1. What is the role of the 'else' block in a try-except statement? Provide an example scenario where it would be useful.

**Ans.1**

The 'else' block in a try-except statement is used to specify a block of code that should be executed when no exceptions are raised within the 'try' block. It is useful for handling the normal, non-exceptional case. Here's an example scenario where it would be useful:

try:

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

except ValueError:

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

else:

result = value \* 2

print("The result is:", result)

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

**Ans.2** Yes, a try-except block can be nested inside another try-except block. This is known as nested exception handling and allows for more granular handling of exceptions. Here's an example:

try:

try:

x = 10 / 0

except ZeroDivisionError:

print("Inner except block: Division by zero")

except ZeroDivisionError:

print("Outer except block: Division by zero")

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

**Ans.3** To create a custom exception class in Python, you can inherit from the base Exception class or any of its subclasses. Here's an example:

class MyCustomException(Exception):

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

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

**# Usage of the custom exception**

try:

if some\_condition:

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

except MyCustomException as e:

print("Custom exception caught:", e)

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

**Ans.4**

Some common built-in exceptions in Python include:

* SyntaxError: Raised for syntax errors in the code.
* IndentationError: Raised when there is an issue with indentation.
* NameError: Raised when a local or global name is not found.
* TypeError: Raised when an operation or function is applied to an inappropriate object.
* ValueError: Raised when a function receives an argument of the correct type but with an inappropriate value.
* ZeroDivisionError: Raised when division by zero is attempted.
* IndexError: Raised when trying to access an index that is out of range in a sequence.
* FileNotFoundError: Raised when an attempt to open a file fails because the file does not exist.

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

**Ans.5**

Logging in Python is a built-in module that provides a flexible and efficient way to track the execution of a program. It is important in software development for several reasons, including:

* Debugging: Logging helps developers identify issues, errors, and unexpected behavior during development and debugging.
* Monitoring: It allows real-time monitoring of applications in production environments, helping to identify and resolve issues as they occur.
* Auditing: Logging can be used to keep a record of important events and user activities for auditing and compliance purposes.
* Performance analysis: Log messages can provide insights into the performance of an application, helping to optimize code.

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

each log level would be appropriate.

**Ans.6**

Log levels in Python logging serve to categorize log messages based on their severity. Common log levels include:

* + DEBUG: Detailed information for debugging purposes.
  + INFO: General information about the program's progress.
  + WARNING: Indications of potential issues that do not stop the program's execution.
  + ERROR: Indications of errors that caused the program to fail or produce unexpected results.
  + CRITICAL: Severe errors that may lead to application termination.

Example of when to use each log level:

import logging

logging.basicConfig(level=logging.DEBUG)

logging.debug("Debug message: Detailed information")

logging.info("Info message: General progress information")

logging.warning("Warning message: Potential issue")

logging.error("Error message: Program encountered an error")

logging.critical("Critical message: Severe error")

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

**Ans.7**

Log formatters in Python logging allow you to customize the format of log messages. You can define the format of log records, including timestamps, log levels, and the message itself. Here's how you can customize the log message format using formatters:

**import logging**

logging.basicConfig**(format="%(asctime)s - %(name)s - %(levelname)s - %(message)s")**

logger = logging.getLogger**("example")**

logger.warning**("This is a custom log message.")**

8. How can you set up logging to capture log messages from multiple modules or classes in a Python application?

**Ans.8**

To capture log messages from multiple modules or classes in a Python application, you can configure a central logging system and define loggers for each module or class. Use a common configuration to ensure consistency. For **example:**

**# main.py**

import logging

from module1 import logger as module1\_logger

from module2 import logger as module2\_logger

logging.basicConfig(level=logging.DEBUG)

module1\_logger.debug("Module 1 debug message")

module2\_logger.debug("Module 2 debug 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.9**

**The key differences between logging and print statements in Python are**:

* Logging is more flexible and configurable. You can control the format, log levels, and where log messages are directed (e.g., files, console, network).
* Logging is intended for debugging and monitoring in real-world applications, whereas print statements are primarily for debugging during development.
* Logging allows you to categorize and prioritize log messages based on their importance (log levels).
* Log messages can be disabled or enabled based on their log level, which can help troubleshoot or monitor an application without modifying the code.
* Logging is thread-safe and can be used in multi-threaded applications without issues.

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.10**

**Python program to log a message to a file named "app.log" with the specified requirements**:

import logging

# Configure the logger

logging.basicConfig(filename="app.log", level=logging.INFO, format="%(asctime)s - %(levelname)s - %(message)s")

# Log an "INFO" level 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.11**

**Python program to log an error message to the console and a file named "errors.log" if an exception occurs during the program's execution:**

import logging

import datetime

# Configure the logger

logging.basicConfig(level=logging.ERROR)

error\_log\_file = "errors.log"

# Function that may raise an exception

def potentially\_error\_prone\_function():

try:

result = 10 / 0 # Division by zero, which raises an exception

except ZeroDivisionError as e:

# Log the error to the console and the error log file

logging.error(f"Error: {e}")

with open(error\_log\_file, "a") as file:

timestamp = datetime.datetime.now()

file.write(f"[{timestamp}] Error: {e}\n")

# Call the function

potentially\_error\_prone\_function()