**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:** In a try-except statement in Python, the else block is optional and is executed only if the code inside the try block runs successfully without raising any exceptions. Its role is to contain code that should be executed if no exceptions occur.

**For example:**

def divide\_numbers(a, b):

try:

result = a / b

except ZeroDivisionError:

print("Error: Division by zero")

else:

print("Division successful. Result:", result)

# Example usage:

divide\_numbers(10, 2) # This will print: "Division successful. Result: 5.0"

divide\_numbers(10, 0) # This will print: "Error: Division by zero"

**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 useful when you want to handle different types of exceptions at different levels of your code hierarchy.

**For example:**

def divide\_numbers(a, b, c):

try:

result = a / b

try:

final\_result = result / c

except ZeroDivisionError:

print("Error: Division by zero in inner try block")

except ZeroDivisionError:

print("Error: Division by zero in outer try block")

else:

print("Division successful. Final result:", final\_result)

# Example usage:

divide\_numbers(10, 2, 5) # This will print: "Division successful. Final result: 1.0"

divide\_numbers(10, 0, 5) # This will print: "Error: Division by zero in outer try block"

divide\_numbers(10, 2, 0) # This will print: "Error: Division by zero in inner try block"

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

**Ans 3:** You can create a custom exception class in Python by subclassing the built-in Exception class or any other built-in exception class. Here's how you can create a custom exception class and demonstrate its usage:

class CustomError(Exception):

"""Custom exception class"""

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

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

# Example usage:

def check\_number(x):

if x < 0:

raise CustomError("Number must be non-negative")

try:

check\_number(-5)

except CustomError as e:

print("Caught custom exception:", e)

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

**Ans 4:** Python comes with a variety of built-in exceptions to handle different types of errors that can occur during program execution. Some common built-in exceptions include:

1. SyntaxError: Raised when the Python parser encounters a syntax error.
2. IndentationError: Raised when there is incorrect indentation.
3. NameError: Raised when a local or global name is not found.
4. TypeError: Raised when an operation or function is applied to an object of inappropriate type.
5. ValueError: Raised when a built-in operation or function receives an argument that has the right type but an inappropriate value.
6. KeyError: Raised when a dictionary key is not found.
7. IndexError: Raised when a sequence index is out of range.
8. FileNotFoundError: Raised when a file or directory is requested but cannot be found.
9. IOError: Raised when an input/output operation fails.
10. ZeroDivisionError: Raised when division or modulo by zero occurs.
11. AssertionError: Raised when an assert statement fails.
12. AttributeError: Raised when an attribute reference or assignment fails.
13. OverflowError: Raised when the result of an arithmetic operation is too large to be represented.
14. ImportError: Raised when an import statement fails to find the module definition.
15. KeyboardInterrupt: Raised when the user interrupts program execution, usually by pressing Ctrl+C.

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

**Ans 5:** Logging in Python refers to the process of recording events, messages, and information about the execution of a program. It's a crucial aspect of software development for several reasons:

 **Debugging**: Logging provides a way to track the flow of execution in a program, making it easier to identify and debug issues when they occur.

 **Error Reporting**: By logging error messages, exceptions, and stack traces, developers can gain insights into what went wrong during program execution, aiding in diagnosing and fixing bugs.

 **Monitoring**: Logging allows developers to monitor the behavior and performance of a system in real-time. By logging relevant metrics and events, developers can identify bottlenecks, performance issues, or unusual behavior that might require attention.

 **Auditing and Compliance**: In many applications, logging is essential for auditing purposes and compliance with regulations. By logging relevant actions and events, developers can maintain a record of user interactions and system activities for security and regulatory purposes.

 **Historical Analysis**: Logs provide a historical record of the execution of a program, allowing developers to analyze past behavior, identify trends, and make informed decisions about optimizations or improvements.

 **Communication**: Logging can also serve as a means of communication between different components of a system or between developers. By logging informative messages and status updates, developers can gain insights into the state of a system and its components.

**6. Explain the purpose of log levels in Python logging and provide examples of when each log level would be appropriate.**

**Ans 6:** In Python logging, log levels are used to categorize log messages based on their severity or importance. Python's logging module provides several predefined log levels, each serving a specific purpose.

Here are the standard log levels in Python logging, ordered from least severe to most severe:

 **DEBUG**: Detailed information, typically useful only for diagnosing problems.

 **INFO**: Confirmation that things are working as expected.

 **WARNING**: An indication that something unexpected happened or an issue that might need attention in the future.

 **ERROR**: A serious problem occurred during execution, but the program can still continue.

 **CRITICAL**: A critical error occurred, and the program may not be able to continue running.

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

**Ans 7:** In Python's logging module, log formatters are objects responsible for formatting log records into text. They determine how log messages are presented, including what information is included (such as timestamp, log level, module name, etc.) and how it's formatted.

You can customize the log message format using formatters by creating a Formatter object and specifying the desired format string.

Example of how you can customize the log message format using formatters in Python:

import logging

# Create a custom formatter

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

# Create a logger

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

logger.setLevel(logging.DEBUG)

# Create a file handler and set the formatter

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

file\_handler.setFormatter(formatter)

# Add the file handler to the logger

logger.addHandler(file\_handler)

# Log some messages

logger.debug('This is a debug message')

logger.info('This is an info message')

logger.warning('This is a warning message')

logger.error('This is an error message')

logger.critical('This is a critical message')

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

**Ans 8:** To set up logging to capture log messages from multiple modules or classes in a Python application, you can follow these steps:

1. **Create a Logger**: First, create a logger instance using the logging.getLogger() method. This logger will be used to emit log messages.
2. **Set Logger Level**: Set the level for the logger using the setLevel() method. This level acts as a filter for which log messages will be emitted by the logger.
3. **Create Handlers**: Create one or more handlers to define where the log messages should be sent. Handlers can direct log messages to different destinations such as files, streams, or external services.
4. **Set Handler Level**: Optionally, set the level for each handler using the setLevel() method. This level acts as a filter for which log messages will be handled by the handler.
5. **Set Formatter**: Optionally, set a formatter for each handler using the setFormatter() method. Formatters define the format of the log messages.
6. **Add Handlers to Logger**: Add the handlers to the logger using the addHandler() method. This associates the handlers with the logger, so they will handle log messages emitted by the logger.

**For Example :**

import logging

# Create a logger

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

logger.setLevel(logging.DEBUG) # Set logger level to capture all messages

# Create a file handler

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

file\_handler.setLevel(logging.INFO) # Set handler level to capture only INFO messages

# Create a stream handler

stream\_handler = logging.StreamHandler()

stream\_handler.setLevel(logging.ERROR) # Set handler level to capture only ERROR messages

# Create a formatter and set it for the handlers

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

file\_handler.setFormatter(formatter)

stream\_handler.setFormatter(formatter)

# Add handlers to the logger

logger.addHandler(file\_handler)

logger.addHandler(stream\_handler)

# Log messages from multiple modules or classes

logger.debug('This is a debug message')

logger.info('This is an info message')

logger.warning('This is a warning message')

logger.error('This is an error message')

logger.critical('This is a 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 9 :** The primary difference between logging and print statements in Python lies in their purpose and functionality:

1. **Purpose**:

* **Logging**: The logging module in Python is specifically designed for recording events and diagnostic information. It allows you to capture different levels of severity (e.g., DEBUG, INFO, WARNING, ERROR, CRITICAL) and route these messages to various destinations (e.g., console, files, databases).
* **Print Statements**: Print statements are primarily used for basic debugging or displaying information during program execution. They output information to the console without any consideration for severity levels or structured logging.

2 **Functionality**:

* **Logging**: With logging, you can configure different log levels to control the verbosity of the output. You can customize the format of log messages, direct them to different handlers (e.g., file, console), and easily filter or disable certain types of log messages based on their severity. Additionally, logging provides more structured information, including timestamps and logger names.
* **Print Statements**: Print statements are straightforward and simply output the provided text to the console. They don't offer features like different severity levels, log formatting, or the ability to disable output selectively.

**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:**

import logging

# Configure logging

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

# 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 11:**

import logging

import traceback

import sys

def main():

try:

# Your program logic here

result = 1 / 0 # This will raise a ZeroDivisionError for demonstration purposes

except Exception as e:

# Log the exception

log\_exception(e)

def log\_exception(exception):

# Configure logging to console

console\_handler = logging.StreamHandler(sys.stdout)

console\_handler.setLevel(logging.ERROR)

console\_formatter = logging.Formatter('%(asctime)s - ERROR - %(message)s')

console\_handler.setFormatter(console\_formatter)

logging.getLogger().addHandler(console\_handler)

# Configure logging to file

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

file\_handler.setLevel(logging.ERROR)

file\_formatter = logging.Formatter('%(asctime)s - ERROR - %(message)s')

file\_handler.setFormatter(file\_formatter)

logging.getLogger().addHandler(file\_handler)

# Log the exception

logging.error(f"{type(exception).\_\_name\_\_}: {exception}")

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

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

main()