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

Answer: The 'else' block in a try-except statement is optional and is executed if no exception is raised in the corresponding 'try' block. It allows you to define a code block that should run only when the 'try' block executes successfully, without any exceptions being raised.

Example scenario where 'else' block is useful:

def divide\_numbers(a, b):

try:

result = a / b

except ZeroDivisionError:

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

else:

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

divide\_numbers(10, 2) # Output: Division successful. Result: 5.0

divide\_numbers(10, 0) # Output: Error: Division by zero is not allowed.

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

Answer: Yes, a try-except block can be nested inside another try-except block. This is known as nested exception handling.

Example of nested try-except blocks:

try:

try:

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

result = 10 / x

print("Result:", result)

except ZeroDivisionError:

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

except ValueError:

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

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

Answer: To create a custom exception class in Python, you need to inherit from the `Exception` class or any of its subclasses. You can define your custom exception class with additional attributes or methods as needed.

Example of a custom exception class:

class CustomError(Exception):

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

self.message = message

try:

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

except CustomError as ce:

print("Custom Exception:", ce.message)

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

Answer: Some common built-in exceptions in Python include:

- ZeroDivisionError: Raised when division or modulo operation is performed with zero as the divisor.

- ValueError: Raised when a built-in operation or function receives an argument of the correct data type but an inappropriate value.

- TypeError: Raised when an operation or function is applied to an object of an inappropriate type.

- IndexError: Raised when attempting to access an index that is out of range in a sequence (e.g., list, tuple).

- FileNotFoundError: Raised when a file or directory is requested, but it cannot be found.

- KeyError: Raised when a dictionary key is not found.

- ImportError: Raised when a module or package is not found.

- AttributeError: Raised when an attribute reference or assignment fails.

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

Answer: Logging in Python is a mechanism used to record the events and actions that occur during the execution of a program. It allows developers to capture and save log messages with different levels of severity, which can be used for debugging, monitoring, and analyzing the behavior of the application.

Logging is crucial in software development because:

- It helps in identifying and understanding the cause of errors and exceptions.

- It provides valuable insights into the flow of the program and the values of variables at different points.

- It assists in diagnosing and fixing issues in production environments without disrupting the users.

- It can be used for performance analysis and optimization.

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

Log levels in Python logging define the severity or importance of log messages. Different log levels are used to categorize log messages based on their significance.

Common log levels in Python logging, in increasing order of severity, are:

- DEBUG: Detailed information for debugging purposes.

- INFO: General information about the program's operation.

- WARNING: Indication of a potential issue that does not interrupt the program's flow.

- ERROR: Indication of a more serious problem that may need attention.

- CRITICAL: Critical errors that can cause the program to crash or behave abnormally.

Examples of log level usage:

import logging

# Configuring logging

logging.basicConfig(level=logging.DEBUG)

# Logging messages at different levels

logging.debug("This is a debug message.")

logging.info("This is an info message.")

logging.warning("This is a warning message.")

logging.error("This is an error message.")

logging.critical("This is a critical message.")

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

Answer: Log formatters in Python logging are used to control the format of log messages when they are displayed or saved to a file. They allow developers to customize how the log messages are presented, including date and time, log level, logger name, and the log message itself.

To customize the log message format using formatters, you can use the `logging.Formatter` class and set it on the logger or handler.

Example of custom log message format:

import logging

# Configuring logging with a custom formatter

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

handler = logging.StreamHandler()

handler.setFormatter(formatter)

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

logger.setLevel(logging.DEBUG)

logger.addHandler(handler)

# Logging messages with the custom format

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

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

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

Answer: To capture log messages from multiple modules or classes in a Python application, you can create a single logger instance with a common name, and all modules or classes can use the same logger. This way, all log messages will be directed to the same output or log file.

Example of setting up logging in multiple modules:

module1.py:

import logging

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

def func1():

logger.info("This is a log message from module1.")

module2.py:

import logging

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

def func2():

logger.warning("This is a warning message from module2.")

main.py:

import logging

import module1

import module2

logging.basicConfig(level=logging.DEBUG)

module1.func1()

module2.func2()

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?

Answer: The logging and print statements serve different purposes in Python:

- Logging: Logging is used to record the events and actions that occur during the execution of a program. Log messages can have different log levels to indicate their severity. Logging is more powerful than print statements because it allows you to control the verbosity of log messages, save them to files, and filter them based on log levels. It is particularly useful for real-world applications where you need to monitor and debug code in production environments without interfering with the user experience.

- Print Statements: Print statements are used primarily for debugging purposes and are often temporary. They are useful for inspecting the values of variables and understanding the flow of the code during development. However, in a real-world application, print statements can become noise and clutter the code, making it difficult to maintain and debug. Additionally, print statements are challenging to control or disable in production environments.

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.

Answer:

import logging

# Configuring logging

logging.basicConfig(filename='app.log', level=logging.INFO)

# Logging 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.

Answer:

import logging

import traceback

import sys

# Configuring logging to console and file

logging.basicConfig(filename='errors.log', level=logging.ERROR, format='%(asctime)s - %(levelname)s: %(message)s', datefmt='%Y-%m-%d %H:%M:%S')

try:

# code that may raise an exception

num1 = int(input("Enter a number: "))

num2 = int(input("Enter another number: "))

result = num1 / num2

print("Result:", result)

except Exception as e:

# Log the exception to console and file

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

traceback.print\_exc(file=sys.stdout)