**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 executed if no exception is raised in the try block. This is useful for running code that should only execute if the code in the try block runs successfully, i.e., without raising any exceptions.**

**Example scenario: In this example, we try to open a file and read its contents. If no exception occurs (e.g., file is found), we execute the code in the else block.**

**python**

**Copy**

**try:**

**file = open('example.txt', 'r')**

**content = file.read()**

**except FileNotFoundError:**

**print("File not found!")**

**else:**

**print("File read successfully!")**

**print(content)**

**finally:**

**file.close()**

**Here, the else block ensures that the file was read successfully before proceeding to print the content.**

**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 allows handling multiple levels of exceptions within different parts of the code.**

**Example:**

**python**

**Copy**

**try:**

**num = int(input("Enter a number: "))**

**try:**

**result = 10 / num**

**except ZeroDivisionError:**

**print("Cannot divide by zero.")**

**else:**

**print(f"Result of division: {result}")**

**except ValueError:**

**print("Invalid input. Please enter a valid integer.")**

**In this case, the outer try-except handles invalid input (e.g., a non-integer), while the inner try-except handles division by zero errors.**

**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 define a class that inherits from the built-in Exception class. You can add custom behavior or messages to the exception if desired.**

**Example:**

**python**

**Copy**

**class NegativeNumberError(Exception):**

**"""Exception raised for negative numbers."""**

**def \_\_init\_\_(self, value):**

**self.value = value**

**self.message = f"Negative number encountered: {value}"**

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

**def check\_positive\_number(value):**

**if value < 0:**

**raise NegativeNumberError(value)**

**return value**

**try:**

**num = int(input("Enter a positive number: "))**

**check\_positive\_number(num)**

**except NegativeNumberError as e:**

**print(f"Error: {e}")**

**In this example, NegativeNumberError is a custom exception that is raised when a negative number is encountered.**

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

**Here are some common built-in exceptions in Python:**

* **ValueError: Raised when a function receives an argument of the correct type but inappropriate value.**
* **IndexError: Raised when trying to access an index that is out of range for a list or string.**
* **KeyError: Raised when trying to access a dictionary with a key that does not exist.**
* **TypeError: Raised when an operation or function is applied to an object of inappropriate type.**
* **ZeroDivisionError: Raised when dividing by zero.**
* **FileNotFoundError: Raised when trying to open a file that does not exist.**
* **AttributeError: Raised when an invalid attribute reference is made.**

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

**Logging in Python refers to recording runtime information and events during the execution of an application. It provides a way to track and monitor what's happening in the application. Logging helps in debugging, error tracking, performance monitoring, and provides insights into the application's behavior.**

**In software development, logging is important because it:**

* **Helps in identifying and fixing issues.**
* **Provides detailed information about errors and exceptions.**
* **Allows tracking of code execution flow, which is useful for debugging and monitoring.**
* **Helps keep a record of system events for future audits.**

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

**The log levels in Python logging define the severity of the messages being logged. Each level represents a specific level of importance, from the most detailed (debug) to the most severe (critical).**

* **DEBUG: Detailed information, typically useful for diagnosing problems.**
  + ***Example*: Logging function entry and exit points for debugging.**
* **INFO: General information about the program's normal operation.**
  + ***Example*: Informing that a process completed successfully.**
* **WARNING: Indicates a potential problem or unexpected situation.**
  + ***Example*: Deprecation warnings or non-fatal errors.**
* **ERROR: Indicates a serious problem, but the program can still continue running.**
  + ***Example*: An invalid user input that causes failure in a specific operation.**
* **CRITICAL: Indicates a severe error that might cause the program to stop.**
  + ***Example*: System failure, application crash, or hardware failure.**

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

**A log formatter in Python is used to define the structure and format of the log messages. By customizing the format, you can control how timestamps, log levels, messages, etc., are displayed in the logs.**

**Example:**

**python**

**Copy**

**import logging**

**# Define the log format**

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

**# Set up a stream handler and apply the formatter**

**handler = logging.StreamHandler()**

**handler.setFormatter(formatter)**

**# Set up logging with the handler**

**logger = logging.getLogger()**

**logger.addHandler(handler)**

**logger.setLevel(logging.DEBUG)**

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

**Here, the format is set to include the timestamp, logger name, log level, and message.**

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

**You can set up logging using a centralized logging configuration and propagate logs from different modules or classes.**

**Example:**

**python**

**Copy**

**import logging**

**# Set up basic configuration for logging**

**logging.basicConfig(level=logging.INFO, format='%(name)s - %(levelname)s - %(message)s')**

**# In Module1.py**

**logger1 = logging.getLogger('Module1')**

**logger1.info('This is an info message from Module1')**

**# In Module2.py**

**logger2 = logging.getLogger('Module2')**

**logger2.error('This is an error message from Module2')**

**This way, both modules log messages to the same centralized logger.**

**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?**

* **print: Outputs information to the console, primarily used for user interaction and debugging during development.**
* **logging: Records detailed logs with timestamps, severity levels, and can output to different destinations (e.g., console, files, remote systems). It is more suitable for production environments.**

**In a real-world application, you should use logging over print statements because:**

* **It provides more control and customization.**
* **It can be directed to different output locations (files, remote servers).**
* **It supports log levels (e.g., DEBUG, ERROR) to differentiate between normal operations and issues.**
* **It can be turned on or off based on the environment (e.g., only log errors in production).**

**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.**

**python**

**Copy**

**import logging**

**# Set up basic configuration to log to a file**

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

**format='%(asctime)s - %(levelname)s - %(message)s',**

**filemode='a') # Append mode**

**# Log an info 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.**

**python**

**Copy**

**import logging**

**# Set up basic configuration for logging to a file**

**logging.basicConfig(filename='errors.log', level=logging.ERROR,**

**format='%(asctime)s - %(levelname)s - %(message)s')**

**# Set up logging to the console**

**console\_handler = logging.StreamHandler()**

**console\_handler.setLevel(logging.ERROR)**

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

**console\_handler.setFormatter(formatter)**

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

**try:**

**# Some code that might raise an exception**

**x = 1 / 0**

**except ZeroDivisionError as e:**

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

**Top of Form**

**Bottom of Form**