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 only if no exception occurs in the corresponding try block. It allows you to specify a block of code to be executed when the try block completes successfully, without any exceptions being raised.*

*Here's an example scenario where the 'else' block can be useful:*

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

*try:*

*result = a / b*

*except ZeroDivisionError:*

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

*else:*

*print("Division result:", result)*

*# Test the function*

*divide\_numbers(10, 2) # No exception occurs, else block is executed*

*divide\_numbers(10, 0) # ZeroDivisionError occurs, else block is not executed*

*In this example, the function divide\_numbers divides two numbers a and b. The try block attempts the division operation, and if no exception occurs, the else block is executed and prints the division result. However, if a ZeroDivisionError occurs, the except block is executed, and the else block is skipped.*

*The else block allows you to separate the code that handles exceptions from the code that executes when no exception occurs. It can be useful for performing additional actions or providing specific output based on the successful execution of the try block.*

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. It allows for more fine-grained handling of exceptions in different parts of the code.*

*Here's an example to illustrate nested exception handling:*

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

*try:*

*try:*

*result = a / b*

*except ZeroDivisionError:*

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

*else:*

*print("Inner try block executed successfully.")*

*except TypeError:*

*print("Error: Invalid operand types.")*

*else:*

*print("Outer try block executed successfully.")*

*# Test the function*

*divide\_numbers(10, 2) # Both try blocks executed successfully*

*divide\_numbers(10, 0) # Inner try block raises ZeroDivisionError*

*divide\_numbers(10, '2') # Outer try block raises TypeError*

*In this example, there are two levels of try-except blocks. The inner try block performs the division operation, and if a ZeroDivisionError occurs, it is caught by the inner except block. The outer try block encapsulates the inner try-except block and provides a higher-level exception handling. If a TypeError occurs during the division operation, it is caught by the outer except block.*

*Nested exception handling allows for handling different types of exceptions at different levels of the code hierarchy. It provides flexibility in handling exceptions based on their specific context and allows for more precise error handling.*

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

demonstrates its usage.

Answer:

*In Python, you can create a custom exception class by defining a new class that inherits from the built-in Exception class or any of its subclasses. Here's an example that demonstrates how to create and use a custom exception class:*

*class CustomException(Exception):*

*pass*

*def validate\_age(age):*

*if age < 0:*

*raise CustomException("Age cannot be negative.")*

*elif age < 18:*

*raise CustomException("Age must be at least 18 to access this content.")*

*else:*

*print("Access granted. Enjoy the content!")*

*# Test the function*

*try:*

*validate\_age(15) # Raises CustomException: "Age must be at least 18 to access this content."*

*except CustomException as e:*

*print("Error:", str(e))*

*try:*

*validate\_age(-5) # Raises CustomException: "Age cannot be negative."*

*except CustomException as e:*

*print("Error:", str(e))*

*validate\_age(25) # Access granted. Enjoy the content!*

*In this example, we define a custom exception class CustomException by inheriting from the base Exception class. The CustomException class does not have any additional functionality or attributes, so we simply use the pass statement.*

*The validate\_age function takes an age parameter and raises the CustomException in case of invalid age values. In the example, we test the function with different age values. When an exception is raised, we catch it using the try-except block and print the error message.*

*Creating a custom exception class allows you to define and raise your own exceptions that are specific to your application or domain. It provides a way to differentiate and handle specific exceptional conditions in your code.*

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

Answer:

*Python provides several built-in exceptions that cover a wide range of common errors and exceptional conditions. Some of the most commonly used built-in exceptions in Python include:*

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

*ValueError: Raised when a function receives an argument of the correct type but an inappropriate value.*

*IndexError: Raised when an index is out of range for a sequence.*

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

*FileNotFoundError: Raised when a file or directory is not found.*

*SyntaxError: Raised when there is a syntax error in the code.*

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

*IOError: Raised when an input/output operation fails.*

*NameError: Raised when a local or global name is not found.*

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

*These are just a few examples of the built-in exceptions available in Python. Each exception type serves a specific purpose and helps to handle different types of errors or exceptional conditions that may occur during program execution.*

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

Answer:

*Logging in Python is a technique used to record and store informative messages during the execution of a program. It involves the use of the logging module, which provides a flexible and customizable way to generate log messages at different levels of severity.*

*Logging is important in software development for the following reasons:*

*Debugging: Logging helps in identifying and troubleshooting issues or bugs in the code. By logging relevant information, developers can track the flow of the program, monitor variable values, and identify the source of errors.*

*Monitoring and Analysis: Logs provide valuable insights into the behavior of a program during runtime. They can be used to monitor performance, track usage patterns, and analyze system behavior. This information can help optimize the code, identify bottlenecks, and improve overall efficiency.*

*Error Tracking: Logs capture errors, exceptions, and stack traces, allowing developers to diagnose and fix issues. They provide a historical record of errors, which can be invaluable for identifying recurring problems, determining their root causes, and implementing appropriate solutions.*

*Auditing and Compliance: Logging is essential for auditing purposes and ensuring compliance with security and regulatory requirements. By logging relevant activities, such as user actions, system events, or access attempts, organizations can maintain a trail of actions for security analysis, forensic investigation, or compliance audits.*

*Communication and Collaboration: Logs serve as a form of communication between developers, system administrators, and support teams. They provide a shared context and allow different stakeholders to understand the behavior of the system, collaborate on problem-solving, and communicate effectively.*

*Overall, logging plays a crucial role in software development by providing valuable information for debugging, monitoring, analysis, error tracking, compliance, and collaboration. It helps improve the quality and reliability of software systems and facilitates efficient troubleshooting and maintenance.*

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

each log level would be appropriate.

Answer:

*In Python logging, log levels are used to categorize log messages based on their severity or importance. Each log level represents a specific level of severity, and messages can be filtered or handled differently based on their assigned log level. The logging module in Python defines several log levels, each serving a different purpose. Here are the commonly used log levels and their corresponding purposes:*

*DEBUG: This is the lowest log level and is used for detailed information useful for debugging and troubleshooting during development. Debug messages typically provide information about the program's internal state, variable values, or execution flow. Example use cases include printing variable values, function calls, or intermediate results.*

*INFO: The INFO log level is used to provide general informational messages about the program's execution. It gives high-level details about the progress of the program and important milestones. Info messages are helpful for monitoring the program's behavior during normal execution. Example use cases include reporting the start and completion of significant operations, successful connections, or important configuration settings.*

*WARNING: The WARNING log level is used to indicate potential issues or situations that could lead to problems but are not critical. It is typically used for warnings that do not prevent the program from functioning but may require attention. Warning messages often highlight deviations from expected behavior or deprecated features. Example use cases include deprecated function usage, non-fatal errors, or potential performance bottlenecks.*

*ERROR: The ERROR log level is used to indicate errors or exceptional conditions that prevent the program from performing a certain task or result in incorrect behavior. Error messages are logged when something goes wrong that requires immediate attention. Example use cases include handling exceptions, reporting failed operations, or unexpected input.*

*CRITICAL: The CRITICAL log level represents the highest severity level and is used to indicate critical errors or failures that may lead to program termination or major system issues. Critical messages typically require immediate action and indicate a severe problem that needs to be addressed urgently. Example use cases include unrecoverable errors, system failures, or security breaches.*

*Log levels allow developers to control the verbosity and granularity of log messages based on their specific needs. By setting an appropriate log level, developers can filter out less important messages and focus on the ones that require attention. This helps in troubleshooting, diagnosing issues, and analyzing the behavior of the program.*

*For example, during development, setting the log level to DEBUG can provide detailed information about the program's execution and variable values, aiding in debugging. In production environments, it is common to set the log level to INFO to capture high-level information about the system's behavior. In case of critical errors or failures, the log level can be set to CRITICAL to immediately alert the developers or system administrators.*

*It's worth noting that log levels can be configured to log messages of a specific level and above. This allows developers to fine-tune the logging output based on the requirements of the application or system.*

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

message format using formatters?

Answer:

*In Python logging, log formatters are used to define the format of log messages that are emitted by the logging system. Log formatters determine how the log records are formatted before being outputted to the desired logging destinations, such as the console, files, or external services.*

*The logging module in Python provides a built-in Formatter class that allows customization of the log message format. The Formatter class offers various placeholders or format codes that can be used to include specific information in the log message. These placeholders are replaced with actual values from the log record when the log message is formatted.*

*Here are some commonly used placeholders:*

*%(asctime)s: The human-readable timestamp when the log record was created.*

*%(levelname)s: The log level name (e.g., DEBUG, INFO, WARNING, ERROR, CRITICAL).*

*%(message)s: The log message itself.*

*%(name)s: The name of the logger that generated the log record.*

*%(filename)s: The name of the source file where the logging call was made.*

*%(lineno)d: The line number in the source file where the logging call was made.*

*To customize the log message format using a formatter, you can follow these steps:*

*Create an instance of the Formatter class.*

*Configure the desired log message format by specifying the format string with the appropriate placeholders.*

*Set the formatter for the desired logger or handlers using the setFormatter() method.*

*Here's an example of customizing the log message format using a formatter:*

import logging

# Create a logger

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

logger.setLevel(logging.DEBUG)

# Create a formatter

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

# Create a console handler and set the formatter

console\_handler = logging.StreamHandler()

console\_handler.setFormatter(formatter)

# Add the console handler to the logger

logger.addHandler(console\_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')

*In the above example, we create a formatter with the format string '%(asctime)s - %(levelname)s - %(message)s'. This format string includes the timestamp, log level, and message placeholders. We then create a console handler and set the formatter on it using the setFormatter() method. Finally, we add the console handler to the logger and log some messages.*

*The output of the above code will be something like:*

2023-07-05 10:15:30,123 - DEBUG - This is a debug message

2023-07-05 10:15:30,124 - INFO - This is an info message

2023-07-05 10:15:30,125 - WARNING - This is a warning message

*By customizing the log message format using formatters, you can control the information included in the log messages, making them more informative and tailored to your specific needs.*

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 set up a logger hierarchy using the logging module. The logger hierarchy allows you to organize your loggers in a hierarchical structure based on the module or class names. This way, you can configure and capture log messages from different parts of your application separately or collectively.*

*Here's how you can set up logging to capture log messages from multiple modules or classes:*

*Import the logging module:*

import logging

*Create a logger for each module or class:*

logger1 = logging.getLogger('module1')

logger2 = logging.getLogger('module2')

*Set the desired log levels for the loggers:*

logger1.setLevel(logging.DEBUG)

logger2.setLevel(logging.INFO)

*Configure the logging output destinations, such as a file or console handler, for each logger:*

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

console\_handler = logging.StreamHandler()

logger1.addHandler(file\_handler)

logger2.addHandler(console\_handler)

*Log messages from the respective modules or classes using their corresponding loggers:*

logger1.debug('Debug message from module1')

logger2.info('Info message from module2')

*By setting up loggers for each module or class and configuring different log levels and handlers, you can capture log messages separately and direct them to different output destinations. This allows you to control the granularity and destination of the log messages generated by different parts of your application.*

*Note that when setting up logging, it's important to use the same logger names consistently across your modules or classes to ensure that they are correctly associated with the desired loggers in the hierarchy.*

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 in Python serve different purposes and are used in different contexts.*

*Purpose:*

*Logging: The primary purpose of logging is to provide a systematic and configurable way to record messages, events, and errors during the execution of a program. It is used for monitoring, troubleshooting, and debugging purposes.*

*Print statements: Print statements are primarily used for basic output and debugging during development. They are often used to display values of variables, intermediate results, or general information.*

*Output and flexibility:*

*Logging: Logging provides more flexibility in terms of output destinations, log levels, and log message formatting. It allows you to direct log messages to various destinations like files, console, or external services. Log levels allow you to control the verbosity of the log messages and selectively capture different levels of information. Log message formatting can be customized to include timestamps, log levels, module names, etc.*

*Print statements: Print statements simply write output to the console and are less flexible in terms of customization. They do not offer the same level of control over the formatting or the ability to direct output to different destinations.*

*Application usage:*

*Logging is more suitable for real-world applications where you need a structured and configurable approach to handle logging. It provides better control over what information is logged, where it is logged, and how it is presented. Logging is especially useful for long-running applications, distributed systems, or applications running in production environments where debug information needs to be captured without interfering with the normal output of the application.*

*Print statements are more commonly used during development for quick debugging or understanding the flow of the program. They are less suitable for long-term logging needs as they can clutter the console output and may not provide the level of detail and configurability required for comprehensive logging in production environments.*

*In summary, while print statements are convenient for quick debugging or basic output during development, logging provides a more robust and configurable approach for recording and managing log messages in real-world applications. Logging is better suited for long-term logging needs, troubleshooting, monitoring, and debugging 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:

*To achieve the requirements mentioned, you can use the Python logging module. Here's an example program that logs a message to a file named "app.log" with the specified log level and file appending:*

import logging

# Configure logging

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

# Log the message

logging.info('Hello, World!')

*Explanation:*

*We import the logging module.*

*The basicConfig() function from the logging module is used to configure the logging system. We specify the filename as 'app.log' using the filename parameter. The level parameter is set to logging.INFO to log messages with the "INFO" level or above. The filemode parameter is set to 'a' to append new log entries without overwriting previous ones.*

*We use the logging.info() function to log the message 'Hello, World!' with the specified log level.*

*When you run this program, it will create a file named "app.log" if it doesn't already exist and append the log message "Hello, World!" with the "INFO" log level to the file. Subsequent runs of the program will append new log entries to the same file.*

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:

*To achieve the requirements mentioned, you can use the Python logging module along with exception handling. Here's an example program that logs an error message to the console and a file named "errors.log" when an exception occurs:*

import logging

import datetime

# Configure logging

logging.basicConfig(level=logging.ERROR)

# Create a file handler

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

file\_handler.setLevel(logging.ERROR)

# Create a console handler

console\_handler = logging.StreamHandler()

console\_handler.setLevel(logging.ERROR)

# Create a formatter

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

# Add the formatter to the handlers

file\_handler.setFormatter(formatter)

console\_handler.setFormatter(formatter)

# Get the logger

logger = logging.getLogger()

logger.addHandler(file\_handler)

logger.addHandler(console\_handler)

try:

# Your code here

# ...

# Simulating an exception

raise ValueError('Something went wrong')

except Exception as e:

# Log the exception

logger.error(f'Exception occurred: {type(e).\_\_name\_\_} - {e}', exc\_info=True)

*Explanation:*

*We import the logging and datetime modules.*

*The basicConfig() function from the logging module is used to configure the logging system. We set the logging level to logging.ERROR to only log error messages.*

*We create a FileHandler object named file\_handler and set its level to logging.ERROR.*

*We create a StreamHandler object named console\_handler to log messages to the console with a level of logging.ERROR.*

*We create a Formatter object named formatter to format the log messages. In this case, we include the timestamp, log level, and message in the format.*

*We add the formatter to the file\_handler and console\_handler using the setFormatter() method.*

*We get the root logger using logging.getLogger().*

*We add the file\_handler and console\_handler to the root logger using the addHandler() method.*

*Inside the try block, you can place your code where exceptions may occur. In this example, we simulate an exception by raising a ValueError.*

*Inside the except block, we catch the exception and log an error message using logger.error(). The exc\_info=True argument is used to include the traceback information in the log message.*

*When you run this program, it will log the error message to the console and append it to the "errors.log" file if an exception occurs. The error message will include the exception type, the exception message, and a timestamp.*