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

**Solution 1.**

The 'else' block in a try-except statement is optional and is executed if no exceptions are raised in the try block. Its role is to specify the code that should be executed when the try block completes successfully without any exceptions.

The 'else' block is useful when you want to perform certain actions only when the code in the try block succeeds without raising any exceptions. It allows you to separate the code that can potentially raise exceptions from the code that should only be executed when no exceptions occur.

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

def divide\_numbers(numerator, denominator):

try:

result = numerator / denominator

except ZeroDivisionError:

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

else:

print("Division result:", result)

# Example usage

divide\_numbers(10, 2)

divide\_numbers(5, 0)

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

**Solution 2.**

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

Nested exception handling allows for more granular handling of specific exceptions at different levels of code execution. It allows for handling different types of exceptions at different levels and provides the flexibility to catch and handle exceptions at appropriate levels based on the specific requirements.

Here's an example to illustrate nested exception handling:

try:

# Outer try block

numerator = int(input("Enter the numerator: "))

denominator = int(input("Enter the denominator: "))

try:

# Inner try block

result = numerator / denominator

print("Division result:", result)

except ZeroDivisionError:

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

except ValueError:

print("Error: Invalid input. Please enter integers only.")

In the above example, there are two levels of exception handling. The outer try block handles a ValueError that may occur when converting the input to integers. If an invalid input (non-integer) is provided, it raises a ValueError, and the corresponding error message is printed.

The inner try block handles a ZeroDivisionError that may occur when dividing the numerator by the denominator. If the denominator is zero, it raises a ZeroDivisionError, and the corresponding error message is printed.

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

**Solution 3.**

To create a custom exception class in Python, you can define a new class that inherits from the built-in Exception class or any of its subclasses. By creating a custom exception class, you can define your own exception types with specific behaviors and attributes.

Here's an example that demonstrates the creation and usage of a custom exception class:

class CustomException(Exception):

pass

def divide\_numbers(a, b):

if b == 0:

raise CustomException("Division by zero is not allowed.")

return a / b

try:

result = divide\_numbers(10, 0)

print("Result:", result)

except CustomException as e:

print("Custom Exception:", e)

By creating and using custom exception classes, you can define your own exception hierarchy and handle specific types of exceptions in your code. This allows for more fine-grained error handling and better organization of error handling logic.

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

**Solution 4.**

Python provides several built-in exceptions that cover a wide range of error conditions. Some common built-in exceptions in Python include:

* SyntaxError: Raised when there is a syntax error in the code.
* TypeError: Raised when an operation or function is performed on an object of inappropriate type.
* NameError: Raised when a local or global name is not found.
* ValueError: Raised when a function receives an argument of the correct type but an inappropriate value.
* IndexError: Raised when a sequence subscript is out of range.
* KeyError: Raised when a dictionary key is not found.
* FileNotFoundError: Raised when a file or directory is requested but cannot be found.
* IOError: Raised when an input/output operation fails.
* ZeroDivisionError: Raised when division or modulo operation is performed with zero as the denominator.

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

**Solution 5.**

Logging in Python refers to the process of recording events or messages during the execution of a program. The logging module in Python provides a flexible and efficient way to log information at different levels of severity. It allows developers to track the flow of execution, monitor the behavior of a program, and troubleshoot issues.

Logging is important in software development for several reasons:

* Debugging and Troubleshooting: Logging helps in identifying and fixing issues by providing valuable information about the execution flow, variable values, and error messages. It allows developers to track down problems and understand what went wrong.
* Monitoring and Analysis: By logging important events and metrics, developers can monitor the behavior of their software in real-time or retrospectively. This enables them to analyze performance, identify bottlenecks, and optimize the code or system.
* Auditing and Compliance: Logging plays a crucial role in auditing and compliance requirements. It allows developers to track user actions, system events, and security-related activities for accountability and regulatory purposes.
* Understanding Application Behavior: By logging relevant information, developers can gain insights into how their application behaves in different scenarios and under various conditions. This helps in understanding usage patterns, identifying usage anomalies, and making informed decisions for improvements.
* Production Support and Maintenance: Logging assists in supporting and maintaining software in production environments. It helps in diagnosing issues, identifying patterns, and providing meaningful logs to support teams for efficient troubleshooting.

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

**Solution 6.**

Log levels in Python logging define the severity or importance of the logged messages. They allow developers to categorize and filter log messages based on their significance. Python logging provides several predefined log levels, each serving a specific purpose. The commonly used log levels, in increasing order of severity, are:

1. DEBUG: The DEBUG level is used for detailed diagnostic information. It is typically used during development and debugging to log fine-grained details, variable values, and internal states of the program. For example:

import logging

logging.basicConfig(level=logging.DEBUG)

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

1. INFO: The INFO level provides general information about the program's execution. It is used to convey important operational details or milestones. Info-level logs are generally suitable for production environments where you want to keep track of the program's progress. For example:

import logging

logging.basicConfig(level=logging.INFO)

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

1. WARNING: The WARNING level is used to indicate potential issues or unexpected situations that may cause problems in the future but do not prevent the program from continuing. Warnings are often used to highlight conditions that could lead to errors or unintended behavior. For example:

import logging

logging.basicConfig(level=logging.WARNING)

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

1. ERROR: The ERROR level is used to indicate errors that occurred during the execution of the program. These errors might impact the normal flow of the program or cause it to terminate prematurely. Error-level logs are typically used to capture exceptions, critical failures, or unexpected conditions. For example:

import logging

logging.basicConfig(level=logging.ERROR)

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

1. CRITICAL: The CRITICAL level represents the most severe log level. It is used for critical errors or exceptional conditions that require immediate attention. Critical-level logs indicate that the program may not be able to continue running. For example:

import logging

logging.basicConfig(level=logging.CRITICAL)

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

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

**Solution 7.**

Log formatters in Python logging allow developers to customize the format of log messages. They determine how the log records are formatted before being emitted. Python logging provides various predefined formatters, but you can also create custom formatters to suit your specific needs.

To customize the log message format using formatters, you need to create an instance of the Formatter class from the logging module and configure it with the desired format. The format string can include placeholders for various attributes of the log record, such as the log level, timestamp, module name, message, etc.

Here's an example that demonstrates customizing the log message format using formatters:

import logging

# Create a formatter

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

# Create a logger and set its level

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

logger.setLevel(logging.DEBUG)

# Create a file handler and set the formatter

file\_handler = logging.FileHandler("mylog.log")

file\_handler.setFormatter(formatter)

# Add the file handler to the logger

logger.addHandler(file\_handler)

# Log some messages

logger.debug("Debug message")

logger.info("Info message")

logger.warning("Warning message")

logger.error("Error message")

In the above example, the Formatter class is instantiated with the format string "%(asctime)s - %(levelname)s - %(message)s". This format string contains three placeholders:

%(asctime)s: Represents the timestamp of the log record.

%(levelname)s: Represents the log level of the record (e.g., DEBUG, INFO, WARNING).

%(message)s: Represents the log message itself.

By configuring the formatter with this format string and attaching it to a file handler, the log messages will be formatted accordingly when written to the file.

1. How can you set up logging to capture log messages from multiple modules or

**Solution 8**

To capture log messages from multiple modules or classes in a Python application, you can follow these steps:

Import the logging module in each module or class where you want to log messages.

Create a logger instance for each module or class using logging.getLogger(\_\_name\_\_). The \_\_name\_\_ attribute represents the module or class name and ensures that logs are tagged with the correct source.

Configure the log level and handlers for each logger instance.

Add the appropriate handlers (e.g., FileHandler, StreamHandler, etc.) to each logger instance to define where the log messages should be directed.

Here's an example to illustrate the setup of logging for multiple modules or classes:

import logging

# Create a logger for module1

logger = logging.getLogger(\_\_name\_\_)

logger.setLevel(logging.DEBUG)

# Create a file handler

file\_handler = logging.FileHandler("module1.log")

file\_handler.setLevel(logging.DEBUG)

# Create a formatter

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

file\_handler.setFormatter(formatter)

# Add the file handler to the logger

logger.addHandler(file\_handler)

# Log some messages

logger.debug("Debug message from module1")

logger.info("Info message from module1")

1. 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?

**Solution 9.**

The logging and print statements in Python serve different purposes and are used in different contexts.

The key differences between logging and print statements are:

Output location: The print statement outputs messages to the standard output (usually the console or terminal), whereas the logging module allows you to direct log messages to various destinations such as files, network streams, or custom handlers.

Flexibility and control: The logging module provides more flexibility and control over the log messages. You can specify different log levels (debug, info, warning, error, etc.), define multiple handlers with different configurations, and control the verbosity of the logs. On the other hand, print statements simply output messages without these additional features.

Logging levels and filtering: The logging module allows you to define different log levels for different parts of your application and selectively filter or handle log messages based on their severity. This can be useful for debugging, troubleshooting, and monitoring purposes. Print statements do not offer this level of control by default.

When to use logging over print statements in a real-world application:

Debugging and troubleshooting: Logging is especially useful during development and debugging stages when you need to track the flow of your application, identify issues, and get detailed information about the program's behavior. It allows you to selectively enable or disable debug statements based on the log level, helping you focus on specific areas of interest.

Production environments: In a production environment, where your application is running live, logging provides a way to capture important events, errors, and metrics. It allows you to gather insights into the application's behavior, monitor its performance, and diagnose issues without interfering with the regular output of the application.

Logging in libraries or reusable code: When writing libraries or reusable code, using logging instead of print statements allows the user of your code to control the verbosity and handle log messages according to their requirements. It provides a more professional and standardized way to communicate information and errors to users of your code.

1. 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.

**Solution 10**

To accomplish 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 an "INFO" log level:

import logging

# Configure the logging

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

# Log the message

logging.info("Hello, World!")

This program sets up the logging configuration using the basicConfig function from the logging module. It specifies the log file name as "app.log" and sets the log level to "INFO" using the level=logging.INFO argument.

Then, the program logs the message "Hello, World!" using the info method from the logging module. This message will be appended to the log file without overwriting previous log entries.

1. 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.

**Solution 11**

To log an error message to both the console and a file named "errors.log" in case of an exception, you can modify the logging configuration to handle both outputs. Here's an example program that accomplishes this

import logging

import datetime

# Configure the logging

logging.basicConfig(level=logging.ERROR,

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

handlers=[

logging.FileHandler('errors.log'),

logging.StreamHandler()

])

try:

# Your code that may raise an exception

# ...

# Simulating an exception

raise ValueError("Something went wrong!")

except Exception as e:

# Log the exception

logging.error(f"Exception occurred: {type(e).\_\_name\_\_} - {e}")

In this program, the logging configuration is set up with a level of logging.ERROR, which captures only error-level log messages. The log format is specified as '%(asctime)s - %(levelname)s - %(message)s', which includes the timestamp, log level, and message.

The logging handlers are configured to include both a FileHandler (to log to the "errors.log" file) and a StreamHandler (to log to the console). By including both handlers, the log messages will be written to both outputs.

Inside the try-except block, you can place your code that may raise an exception. In the example, I have included a line that simulates an exception by raising a ValueError.

If an exception occurs, it is caught in the except block, and the error message is logged using the logging.error method. The message includes the exception type (type(e).\_\_name\_\_) and the actual exception message (e).