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

Ans The 'else' block in a try-except statement is used to specify a block of code that should run if no exceptions are raised within the 'try' block. It provides an opportunity to execute code when the 'try' block runs successfully, without any exceptions being raised. In other words, the 'else' block is executed when there are no exceptions in the 'try' block.

try:

file = open("data.txt", "r")

data = file.read()

file.close()

except FileNotFoundError:

print("File not found.")

except IOError:

print("Error reading the file.")

else:

# If no exceptions occurred, process the data

lines = data.split('\n')

for line in lines:

print("Processed:", line)

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

Ans

Yes, a try-except block can be nested inside another try-except block. This is known as nested exception handling, and it allows you to handle exceptions at different levels of granularity, providing more specific error handling for different parts of your code.

Here's an example to illustrate nested exception handling:

```python

try:

# Outer try block

num1 = int(input("Enter the first number: "))

num2 = int(input("Enter the second number: "))

result = num1 / num2

except ValueError:

print("Invalid input. Please enter valid integers.")

except ZeroDivisionError:

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

except Exception as e:

print("An error occurred:", e)

else:

try:

# Inner try block

if result < 0:

raise ValueError("Result is negative.")

except ValueError as ve:

print("Inner error:", ve)

else:

print("Result:", result)

finally:

print("Outer finally block always executes.")

```

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

Ans

In Python, you can create a custom exception class by defining a new class that inherits from the built-in `Exception` class or one of its subclasses. Custom exception classes are useful when you want to define specific exceptions tailored to your application's needs. Here's an example of how to create a custom exception class and demonstrate its usage:

```python

class CustomError(Exception):

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

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

# Usage of the custom exception

try:

age = int(input("Enter your age: "))

if age < 0:

raise CustomError("Age cannot be negative.")

elif age < 18:

raise CustomError("You must be 18 or older.")

else:

print("You are eligible.")

except CustomError as ce:

print("Custom error:", ce)

except ValueError:

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

except Exception as e:

print("An error occurred:", e)

```

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

Ans

Python provides a variety of built-in exceptions to handle common error conditions that may occur during program execution. Here are some common built-in exceptions in Python:

1. `SyntaxError`: Raised when there is a syntax error in your code.

2. `IndentationError`: Raised when there is an issue with the indentation of your code, such as inconsistent spacing.

3. `NameError`: Raised when an identifier (variable or function) is not found in the local or global scope.

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

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

6. `KeyError`: Raised when a dictionary is accessed with a key that does not exist.

7. `IndexError`: Raised when you try to access an index that is out of range for a sequence (e.g., list, tuple).

8. `FileNotFoundError`: Raised when an attempt to open or manipulate a file fails because the file does not exist.

9. `ZeroDivisionError`: Raised when attempting to divide by zero.

10. `AttributeError`: Raised when an attribute (method or property) is not found on an object.

11. `OverflowError` and `MemoryError`: Raised when a numeric operation exceeds the limits of available memory or system resources.

12. `AssertionError`: Raised when an `assert` statement fails.

13. `IOError`: Raised when an I/O operation (e.g., reading or writing to a file) encounters an issue.

14. `ImportError` and `ModuleNotFoundError`: Raised when there is a problem importing a module.

15. `KeyboardInterrupt`: Raised when the user interrupts the program's execution (e.g., by pressing Ctrl+C).

16. `RuntimeError`: A generic error that can be raised in various situations.

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

Ans Logging in Python refers to the practice of recording information, messages, and events that occur during the execution of a program. It is a crucial aspect of software development for several reasons:

1. \*\*Debugging and Troubleshooting:\*\* Logging provides a systematic way to capture information about what happened in your code. When an error or unexpected behavior occurs, logs can be invaluable for identifying the root cause, as they often contain details about the state of the application at the time of the issue.

2. \*\*Monitoring and Maintenance:\*\* In production environments, logging is essential for monitoring the health and performance of an application. It allows developers and system administrators to keep an eye on the application's behavior, detect issues in real-time, and respond quickly to problems.

3. \*\*Auditing and Compliance:\*\* For applications that handle sensitive data or need to adhere to compliance regulations, logging can be used to create an audit trail. It records who did what and when, helping with accountability and ensuring compliance requirements are met.

4. \*\*Performance Analysis:\*\* Logs can include timestamps, execution times, and other performance-related data. Analyzing this information can help developers identify bottlenecks and optimize the performance of their software.

5. \*\*Security:\*\* Logging can be used to detect and respond to security incidents. Suspicious activities and unauthorized access attempts can be logged, allowing for rapid response and investigation.

6. \*\*Documentation:\*\* Logs serve as a form of documentation for your application. They provide a historical record of events, which can be useful for understanding the evolution of your codebase and its behavior over time.

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

ans Log levels in Python logging serve the purpose of categorizing log messages based on their importance or severity. They allow you to control which messages get recorded and which ones are ignored, depending on the configured log level. Python's logging module defines several standard log levels, each with its own significance and use cases. Here are the common log levels and when each is appropriate:

1. \*\*DEBUG (10):\*\* The lowest log level, used for detailed debugging information. Debug messages are typically used during development to provide insights into the internal workings of the program. These messages are not usually displayed in production but can be invaluable for diagnosing issues during development and testing.

```python

import logging

logging.basicConfig(level=logging.DEBUG)

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

```

2. \*\*INFO (20):\*\* Informational messages that provide a high-level overview of the application's progress. These messages are typically used to convey important milestones or events in the program's execution.

```python

import logging

logging.basicConfig(level=logging.INFO)

logging.info("Application started.")

```

3. \*\*WARNING (30):\*\* Used to indicate potential issues that are not critical to the program's operation but may require attention. Warning messages can help identify situations where something unexpected has occurred but hasn't caused a failure.

```python

import logging

logging.basicConfig(level=logging.WARNING)

logging.warning("File not found: data.txt")

```

4. \*\*ERROR (40):\*\* Used for reporting errors that prevent a specific part of the program from functioning correctly. Error messages indicate issues that need immediate attention but don't necessarily result in a program crash.

```python

import logging

logging.basicConfig(level=logging.ERROR)

logging.error("An error occurred while processing data.")

```

5. \*\*CRITICAL (50):\*\* The highest log level, reserved for severe errors that may lead to a program crash or other critical failures. Critical messages indicate a situation that requires immediate intervention.

```python

import logging

logging.basicConfig(level=logging.CRITICAL)

logging.critical("Critical error: System is out of memory.")

```

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

Ans

Log formatters in Python logging allow you to customize the format of log messages. They control how log records are rendered into text when they are written to log destinations such as files, the console, or external services. Formatters are essential for producing log messages in a human-readable or machine-readable format that suits your needs.

To customize the log message format using formatters, you typically follow these steps:

1. Create a custom formatter instance.

2. Associate the formatter with a logging handler.

3. Configure the logger to use that handler.

Here's an example of how to create a custom log formatter and apply it to a logger:

```python

import logging

# Step 1: Create a custom log formatter

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

# Step 2: Create a logging handler and associate the formatter

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

file\_handler.setFormatter(formatter)

# Step 3: Create a logger, set its log level, and add the handler

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

logger.setLevel(logging.DEBUG)

logger.addHandler(file\_handler)

# Log messages using the custom formatter

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

logger.info("Informational message.")

logger.error("An error occurred.")

```

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

Ans

Setting up logging to capture log messages from multiple modules or classes in a Python application involves configuring a logging system that can be shared across various parts of your codebase. Here are the steps to achieve this:

1. \*\*Create a Centralized Logger Configuration:\*\*

First, create a centralized logger configuration that sets up the desired log handlers, log levels, and formatters. This configuration should be placed in a separate module or script that's imported by all other modules that need to perform logging. This ensures consistency in logging across your application.

```python

# log\_config.py

import logging

# Create a logger

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

# Set the log level

logger.setLevel(logging.DEBUG)

# Create a file handler

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

# Create a formatter

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

file\_handler.setFormatter(formatter)

# Add the file handler to the logger

logger.addHandler(file\_handler)

```

2. \*\*Import the Centralized Logger Configuration:\*\*

In each module or class where you want to log messages, import the centralized logger configuration from the `log\_config.py` module.

```python

# my\_module.py

import logging

from log\_config import logger

def some\_function():

logger.debug("Debug message from my\_module")

```

3. \*\*Log Messages in Various Modules or Classes:\*\*

In your individual modules or classes, you can now use the logger (`logger`) defined in the centralized configuration to log messages. The logger's configuration (log level, handlers, and formatter) remains consistent across the application.

```python

# another\_module.py

import logging

from log\_config import logger

def another\_function():

logger.info("Informational message from another\_module")

```

4. \*\*Configure Logging Output:\*\*

To control where the log messages go (e.g., to a file, the console, or a remote server), you can configure additional handlers in the centralized logger configuration as needed. You can also customize the log format by modifying the formatter.

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?

Ans

Logging and print statements serve different purposes in Python, and they have distinct characteristics. Here are the key differences and when to use each in a real-world application:

\*\*Logging:\*\*

1. \*\*Purpose:\*\* Logging is a systematic way to record information, messages, and events during the execution of a program. It's primarily used for diagnostic, monitoring, and debugging purposes.

2. \*\*Levels:\*\* Logging allows you to categorize log messages by severity using log levels (e.g., DEBUG, INFO, WARNING, ERROR, CRITICAL). This allows you to control the verbosity of log output.

3. \*\*Destination:\*\* Log messages can be directed to various destinations, such as files, the console, external services, or remote servers. You can configure different handlers for different purposes.

4. \*\*Customization:\*\* Log messages can be customized with timestamps, loggers' names, log levels, and custom format strings. This makes logs suitable for structured data analysis and machine parsing.

5. \*\*Granularity:\*\* Logging provides fine-grained control over what information is recorded, making it suitable for production environments where you want to capture specific events and errors.

\*\*Print Statements:\*\*

1. \*\*Purpose:\*\* Print statements are primarily used for displaying information to the console. They are often used for quick debugging or providing program output for end-users.

2. \*\*Levels:\*\* Print statements do not have levels like log messages. All print statements are treated equally and display the same way.

3. \*\*Destination:\*\* Print statements always display messages to the console where the program is running. They are not easily redirected to other output destinations.

4. \*\*Customization:\*\* Print statements have limited customization options. You can change the separator or end character, but they don't provide advanced formatting or timestamping options.

5. \*\*Granularity:\*\* Print statements are typically used for ad-hoc debugging or providing user-visible output. They are not as suitable for structured logging or capturing detailed diagnostic information.

\*\*When to Use Logging Over Print Statements in a Real-World Application:\*\*

Use logging over print statements in a real-world application when:

1. \*\*Structured Information is Needed:\*\* If you require structured logs with timestamp, log levels, and customizable format for later analysis or monitoring, logging is the better choice.

2. \*\*Fine-Grained Control:\*\* In production environments, where you need to capture specific events, errors, or diagnostic information selectively, logging allows you to control the granularity of the captured data.

3. \*\*Debugging in Production:\*\* For debugging and monitoring production applications, logging is essential because it doesn't interfere with the user experience and allows you to capture issues without disrupting the application's output.

4. \*\*Multiple Output Destinations:\*\* When you need to direct log messages to multiple destinations (e.g., files, console, remote servers) simultaneously, logging provides a flexible way to do so.

5. \*\*Severity Levels:\*\* When you want to categorize messages by severity (e.g., errors, warnings, information) and control which levels are displayed based on the application's current state, logging is more suitable.

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.

Ans

import logging

# Configure the logger

logging.basicConfig(

filename="app.log", # Set the log file name

level=logging.INFO, # Set the log level to INFO

format="%(asctime)s - %(levelname)s - %(message)s", # Define log message format

filemode="a", # Append mode (to append new log entries without overwriting)

)

# Log the message

logging.info("Hello, World!")

# Close the logger (optional but recommended)

logging.shutdown()

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.

Ans

import logging

import traceback

import sys

# Configure the logger

logging.basicConfig(

level=logging.ERROR, # Set the log level to ERROR to capture exceptions

format="%(asctime)s - %(levelname)s - %(message)s", # Define log message format

handlers=[

logging.StreamHandler(sys.stdout), # Log to the console

logging.FileHandler("errors.log") # Log to a file

]

)

try:

# Your code that may raise an exception

result = 10 / 0 # Example: Division by zero

except Exception as e:

# Log the exception with timestamp

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

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

# Close the logger (optional but recommended)

logging.shutdown()