Python Basic Assignment – 23rd June

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 when no exceptions occur in the 'try' block. Its role is to define a set of statements that should run only if the code inside the 'try' block executes successfully without any exceptions being raised.

def process\_data\_from\_file(file\_name):

try:

with open(file\_name, 'r') as file:

data = file.read()

except FileNotFoundError:

print(f"Error: File '{file\_name}' not found.")

except IOError:

print(f"Error: Unable to read from file '{file\_name}'.")

else:

print("Data from the file:")

print(data)

file\_name = "data.txt"

process\_data\_from\_file(file\_name)

In this example, the 'try' block attempts to open and read the specified file. If the file is not found (FileNotFoundError) or there is an issue reading from the file (IOError), the corresponding 'except' block will handle the error and print an error message.

If the file is successfully read without any exceptions, the 'else' block will be executed, and the data processing steps can take place. In this example, we simply print the data read from the file, but in real-world scenarios, more complex data processing tasks can be performed within the 'else' block.

Using the 'else' block in this context allows us to separate the error handling from the normal flow of data processing, enhancing the clarity and maintainability of the code. The 'else' block ensures that the data processing is executed only when the file reading operation is successful, preventing the program from trying to process data that hasn't been read correctly.

1. **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 in Python. This means you can have one "try" block inside another "try" block to handle exceptions at different levels of your code.

def process\_data(file\_name):

try:

with open(file\_name, 'r') as file:

try:

data = file.read()

result = 10 / len(data)

print("Data from the file:", data)

print("Result of division:", result)

except ZeroDivisionError:

print("Error in inner try block: Division by zero.")

except FileNotFoundError:

print(f"Error: File '{file\_name}' not found.")

except IOError:

print(f"Error: Unable to read from file '{file\_name}'.")

# Example usage:

file\_name = "data.txt"

process\_data(file\_name)

Nesting try-except blocks is useful when you need to handle exceptions differently at various levels of your code or when you have specific error handling for different parts of your program. It allows for a more organized and structured approach to exception handling in complex situations.

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

We can create a custom exception class in Python by defining a new class that inherits from the built-in `Exception` class or any other existing exception class. To create a custom exception, you need to define the new class and possibly add some custom behaviour or attributes specific to your application's needs.

class InvalidInputError(Exception):

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

self.message = message

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

def divide\_numbers(a, b):

if b == 0:

raise InvalidInputError("Error: Cannot divide by zero.")

return a / b

try:

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

num2 = int(input("Enter another number (non-zero): "))

result = divide\_numbers(num1, num2)

print("Result of division:", result)

except ValueError:

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

except InvalidInputError as e:

print(e)

except Exception as e:

print("Unexpected error:", str(e))

Creating custom exception classes allows you to handle specific error scenarios more effectively and make your code more readable and maintainable by providing clear and meaningful exception messages for different situations.

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

Python has several common built-in exceptions that represent various error situations. Here are some of the most common ones:

* **ZeroDivisionError**: Raised when division or modulo by zero is encountered.
* **TypeError**: Raised when an operation or function is applied to an object of an inappropriate type.
* **ValueError**: Raised when a built-in operation or 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 it cannot be found.
* **IOError:** Raised when an input/output operation fails, such as the print function unable to write to a file.
* **ImportError**: Raised when an import statement cannot find the module definition or a specific name is not found in a module.
* **NameError**: Raised when a local or global name is not found.
* **SyntaxError**: Raised when there is a syntax error in the code.
* **IndentationError**: Raised when the indentation is incorrect.

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

Logging in Python refers to the process of recording or capturing information, events, and messages during the execution of a program. The Python standard library provides a powerful logging module that allows developers to log various types of messages at different levels of severity. The log messages can then be directed to different outputs, such as the console, files, or even external logging services.

Logging is essential in software development for several reasons:

1. **Debugging and Troubleshooting:** Logging allows developers to capture information about the program's behavior, variable values, and execution flow. When unexpected issues or errors occur, log messages can provide valuable insights into the cause of the problem, helping developers debug and troubleshoot the code effectively.
2. **Monitoring and Auditing**: In production environments, logging plays a critical role in monitoring the health and performance of the application. By logging key events and metrics, developers and system administrators can keep track of the application's behavior and identify potential bottlenecks or performance issues.
3. **Error Reporting**: Logging error messages and stack traces when exceptions occur allows developers to identify and handle errors gracefully. It provides a way to capture unhandled exceptions and report them to developers or system administrators for further investigation.
4. **Data Analysis**: Log messages can be used for data analysis and to gain insights into how the application is being used by end-users. By logging user interactions and behavior, developers can make data-driven decisions to improve the user experience.
5. **Security:** Logging can be used to track and monitor security-related events and potential security breaches. By logging authentication attempts, access control decisions, and critical security events, administrators can detect suspicious activities and take appropriate actions.
6. **Historical Records**: Logging serves as a historical record of events and actions taken by the application. It allows developers to review past events and activities, which can be helpful for forensic analysis or historical reporting.
7. **Explain the purpose of log levels in Python logging and provide examples of when each log level would be appropriate.**

In Python logging, log levels are used to categorize log messages based on their severity or importance. The logging module provides various log levels, each representing a different level of criticality. These log levels allow developers to filter and control the verbosity of log messages based on their significance and the desired level of detail in the logs.

The standard log levels in Python logging, ordered by increasing severity, are as follows:

* **DEBUG**: Used for detailed debugging information. This level is typically used during development to capture fine-grained details that help in troubleshooting and understanding the code's behavior.
* **INFO**: Used for general information about the application's execution. It is used to report key milestones or significant events during the normal operation of the application.
* **WARNING**: Used to indicate potential issues or situations that may cause problems but are not critical. Warnings are used to alert developers about potential errors or unexpected behavior.
* **ERROR**: Used to report errors that caused the application to perform incorrectly or abnormally. These errors might not be fatal, but they indicate significant issues that require attention.
* **CRITICAL**: Used for critical errors that can cause the application to crash or become unusable. This level is reserved for the most severe issues that need immediate attention.

import logging

logging.basicConfig(level=logging.DEBUG)

def divide\_numbers(a, b):

try:

result = a / b

logging.debug(f"Division result: {result}")

logging.info("Division successful.")

return result

except ZeroDivisionError as e:

logging.warning(f"ZeroDivisionError: {e

except Exception as e:

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

raise

num1 = 10

num2 = 0

logging.info(f"Attempting to divide {num1} by {num2}")

result = divide\_numbers(num1, num2)

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

In Python logging, log formatters are objects responsible for determining the format of log messages before they are outputted to the chosen logging destinations, such as the console, log files, or external logging services. The logging module provides the ability to customize the log message format using formatters, allowing developers to control how the log messages are structured and what information is included in each log entry.

The logging.Formatter class is used to create log formatters. The format string consists of placeholders that start with a percentage sign %, followed by a character that specifies the type of information to include in the log message. Common placeholders include:

* %s: The log message.
* %(asctime)s: The timestamp when the log message was created.
* %(levelname)s: The log level name (e.g., DEBUG, INFO, ERROR).
* %(name)s: The name of the logger.
* %(module)s: The name of the module where the log message originated.
* %(lineno)d: The line number in the source code where the log message originated.

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

To capture log messages from multiple modules or classes in a Python application, you can set up logging with a single shared logger that is accessible across all the modules and classes. This allows you to centralize the logging configuration and ensures that all log messages from different parts of the application are directed to the same logging output.

* **Create a Centralized Logger:**

In your main module, create a central logger using the logging.getLogger() method. You can specify a specific name for the logger to make it unique and easily identifiable.

* **Configure the Logging Format and Handlers**:

After creating the central logger, configure the logging format and handlers. You can add multiple handlers to the logger, each directing log messages to different logging destinations, such as the console, log files, or external logging services.

* **Access the Central Logger in Other Modules or Classes:**

In other modules or classes that need to log messages, import the central logger and use it to log the messages.

By using a centralized logger, you can manage and control log messages from various parts of your application efficiently, making it easier to analyze and debug the application as a whole. It also promotes consistency in log message formatting and handling throughout your application.

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?**

The logging and print statements in Python are both used to display information, but they serve different purposes and have distinct use cases in a real-world application.

**Logging:**

* **Purpose:** Logging is a method of capturing and recording information about the application's behavior, events, and errors during its execution.
* **Use Cases**: Logging is commonly used for debugging, monitoring, and analyzing the application in different environments, such as development, testing, and production.
* **Advantages:**

- Allows you to specify different log levels categorize the severity of messages.

- Provides a centralized logging configuration, allowing you to customize log formatting, redirect logs to various outputs (e.g., console, files), and control the verbosity of log messages.

- Supports logging from multiple modules or classes, aggregating all logs in one place for easier analysis.

- Logs can be filtered based on log levels and other criteria, providing more focused and relevant information for debugging and monitoring.

* **Disadvantages:**

- Requires some setup and configuration to use effectively.

- May introduce overhead due to the additional logging operations.

**Print Statements:**

* Purpose: Print statements are primarily used to display temporary or immediate information during the development phase.
* Use Cases: Print statements are often employed for quick debugging, inspecting variable values, and understanding the flow of code during development.
* Advantages:

- Simple and easy to use without any additional setup.

- Provides immediate output, allowing developers to see real-time information during code execution.

- Requires minimal code changes to add print statements for debugging purposes.

* Disadvantages:

- Printing to the console is the default behavior, and the output cannot be easily redirected or filtered like logs.

- Print statements are not suitable for logging in production environments as they might expose sensitive information or clutter the application's output.

In a real-world application, it is generally advisable to use logging over print statements for the following reasons:

1. Debugging and Monitoring: Logging allows for more sophisticated debugging and monitoring capabilities, enabling developers to capture and analyze application behavior in a controlled manner.

2. Log Levels and Filtering: Logging offers different log levels and the ability to filter log messages based on their severity, providing valuable insights without cluttering the output.

3. Centralized Configuration: Logging allows you to set up a centralized logging configuration, making it easier to manage and control log outputs across the application.

4. Production Use: Unlike print statements, logging is suitable for use in production environments where you need to capture important information without revealing sensitive data.

5. Structured Information: With logging, you can provide structured information in log messages, including timestamps, module names, and log levels, making the logs more informative and helpful for debugging and analysis.

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.**

import logging

def setup\_logger():

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

logger.setLevel(logging.INFO)

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

file\_handler.setLevel(logging.INFO)

log\_formatter = logging.Formatter("%(asctime)s - %(levelname)s - %(message)s")

file\_handler.setFormatter(log\_formatter)

logger.addHandler(file\_handler)

return logger

def main():

logger = setup\_logger()

logger.info("Hello, World!")

if \_\_name\_\_ == "\_\_main\_\_":

main()

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.**

import logging

import traceback

import datetime

def setup\_logger():

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

logger.setLevel(logging.ERROR)

file\_handler = logging.FileHandler("errors.log", mode="a")

file\_handler.setLevel(logging.ERROR)

console\_handler = logging.StreamHandler()

console\_handler.setLevel(logging.ERROR)

log\_formatter = logging.Formatter("%(asctime)s - %(levelname)s - %(message)s")

file\_handler.setFormatter(log\_formatter)

console\_handler.setFormatter(log\_formatter)

logger.addHandler(file\_handler)

logger.addHandler(console\_handler)

return logger

def divide\_numbers(a, b):

try:

result = a / b

return result

except Exception as e:

# Log the error message along with the exception type and timestamp

error\_message = f"Exception: {type(e).\_\_name\_\_} - {str(e)}"

logger.error(error\_message)

logger.error(traceback.format\_exc()) # Log the traceback

raise

def main():

try:

# Set up the logger

global logger

logger = setup\_logger()

num1 = 10

num2 = 0

result = divide\_numbers(num1, num2)

print("Result of division:", result)

except Exception as e:

print("An error occurred during execution. Check the 'errors.log' file for details.")

logger.error(f"Main Exception: {type(e).\_\_name\_\_} - {str(e)}")

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

if \_\_name\_\_ == "\_\_main\_\_":

main()