Assignment 3

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 if no exception occurs within the **try** block. It provides a way to specify code that should be executed when the **try** block runs successfully, without any exceptions being raised.

Ex :

def divide\_numbers(a, b):

try:

result = a / b

except ZeroDivisionError:

print("Error: Division by zero!")

else:

print("Division successful.")

return result

divide\_numbers (10, 5)

Output :

Division Successful

2.0

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

Answer : Yes, we can use a try-except block inside another try-except block. This is called nested exception handling.

Ex :

try: # Outer try block

num1 = 10

num2 = 0

try: # Inner try block

result = numerator / denominator

print("Result:", result)

except ZeroDivisionError:

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

except Exception as e:

print("Error in inner try-except block:", str(e))

except Exception as e:

print("Error in outer try-except block:", str(e))

Output :

Error: Division by zero in inner try-except block

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

Answer :

We can create a custom exception class by inheriting from the built-in **Exception**. By creating it, we can define your own exception and add custom errors.

Ex :

class CustomException(Exception):

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

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

self.additional\_info = "Additional information"

def divide\_numbers(a, b):

try:

if b == 0:

raise CustomException("Error: Division by zero is not allowed")

result = a / b

return result

except CustomException as e:

print(str(e))

print(e.additional\_info)

numerator = 10

denominator = 0

result = divide\_numbers(numerator, denominator)

Output :

Error: Division by zero is not allowed

Additional information

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

Answer :

* 1. ZeroDivisionError
  2. TypeError
  3. ValueError
  4. IndexError:
  5. KeyError
  6. FileNotFoundError
  7. ImportError
  8. NameError
  9. AttributeError
  10. IOError

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

**Answer:** Logging in Python is a built-in module that provides record events, messages, and errors that occur during the execution of a program. It allows developers to capture relevant information and store it in a log file for troubleshooting, debugging, and monitoring purposes

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

Answer:

* 1. **DEBUG**: It is lowest log level. It is used for detailed debugging information about variable values, function calls, and other low-level details.

Ex : import logging

logging.basicConfig(level=logging.DEBUG)

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

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

* 1. **INFO**: It is used for informational messages that provide general status about the program's execution. Info messages are typically used to convey high-level information.

Ex : import logging

logging.basicConfig(level=logging.INFO)

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

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

* 1. **WARNING**: It is used for warning messages that indicate potential issues or unusual situations that require attention.

Ex : import logging

logging.basicConfig(level=logging.WARNING)

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

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

* 1. **ERROR**: It is used for error messages that indicate a failure or error condition. Error messages are logged when an unexpected or exceptional situation occurs, but the program can still continue execution.

Ex : import logging

logging.basicConfig(level=logging.ERROR)

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

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

* 1. **CRITICAL**: It is the highest log level. It is used for critical or severe error messages that indicate a failure or error condition that may cause the program to abort or terminate.

Ex : import logging

logging.basicConfig(level=logging.CRITICAL)

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

logger.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?

Answer : The **logging** module in Python provides the **Formatter** class, which can be used to create log formatters. It accepts a formatting string as a parameter.

* **%s** or **{message}**: Replaced with the log message.
* **%levelname** or **{levelname}**: Replaced with the log level name (e.g., DEBUG, INFO, WARNING, ERROR).
* **%asctime** or **{asctime}**: Replaced with the timestamp of the log message in a specified format.
* **%name** or **{name}**: Replaced with the logger name.
* **%module** or **{module}**: Replaced with the module name.
* **%lineno** or **{lineno}**: Replaced with the line number where the log message is generated.

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

Answer :

import logging

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

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

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

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

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

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

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

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?

Answer :

Logging :

The logging module is designed specifically for capturing and managing log messages.

The logging module offers various features such as configurable log levels, multiple handlers (e.g., file handler, stream handler), log formatting.

Logging is the recommended approach for real-world applications, especially in production environments.

Print :

The **print** statement is primarily used for immediate output of the code.

Print statements offer simplicity and ease of use but provide limited control and flexibility.

Print statements are less suitable for production environments as they are not designed for managing and maintaining a large codebase.

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.

Answer :

import logging

logging.basicConfig(

filename="app.log",

level=logging.INFO,

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

filemode="a"

)

logging.info("Hello, World!")

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.

Answer :

import logging

import datetime

logging.basicConfig(

level=logging.ERROR,

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

handlers=[

logging.StreamHandler(),

logging.FileHandler("errors.log")

]

)

try:

x = 4

if x > 5:

raise ValueError("This is a sample exception")

except Exception as e:

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

logging.error(error\_message)