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 gets executed only if no exceptions raised in the try block.

try:

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("Cannot divide by zero.")

else:

print("The result of the division is:", result)

Output:

Enter the first number: 100  
Enter the second number: 20  
The result of the division is: 5.0

Here, in the try block if the user enters valid integers then the program runs successfully, and the else block gets executed and prints the result.

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

Example.

Answer:

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

try:

try:

numerator = 10

denominator = 0

result = numerator / denominator

except ZeroDivisionError:

print("Cannot divide by zero")

some\_value = result \* 2

except NameError:

print("An error occurred while using 'result'")

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

demonstrates its usage.

Answer:

In python you can inherit a custom exception class by inheriting from the built-in Exception class.

def divide(a, b):

if b == 0:

raise CustomException("Cannot divide by zero")

return a / b

try:

result = divide(10, 0)

print("Result:", result)

except CustomException as e:

print("Error:", e.message)

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

Answer:

* **SyntaxError**: Raised when the Python interpreter encounters a syntax error in the code.
* **TypeError**: Raised when an operation or function is applied to an object of an inappropriate type.
* **ValueError**: Raised when a function receives an argument of the correct type but an inappropriate value.
* **NameError**: Raised when a local or global name is not found.
* **IndexError**: Raised when a sequence subscript is out of range.
* **KeyError**: Raised when a dictionary key is not found.
* **FileNotFoundError**: Raised when an attempt to open a file fails because the file does not exist.
* **IOError**: Raised when an input/output operation fails, such as when reading or writing a file.
* **ZeroDivisionError**: Raised when division or modulo operation is performed with zero as the divisor.
* **AttributeError**: Raised when an attribute reference or assignment fails.
* **ImportError**: Raised when an import statement fails to find the module being imported.
* **OverflowError**: Raised when the result of an arithmetic operation is too large to be represented.
* **MemoryError**: Raised when an operation runs out of memory.

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

Answer:

Logging is a built-in module in python that is used to store the information about your script and track events that occur.

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

each log level would be appropriate.

Answer:

There are different purpose of log levels in python logging. Some of them are given below:

Debug – the debug is typically used for detailed information for the debugging processes.

Info – the info is used for providing useful information about the programs execution.

Warning – warning is used to indicate about potential issues and indicates that may cause problems in future.

Error – the error is used to indicate errors or exceptional conditions that prevent program from functioning properly.

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

message format using formatters?

Answer:

In python log formatters are responsible for defining the format of log messages.

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

classes in a Python application?

Answer:

To setup logging to capture log messages from multiple modules or classes in python follow the steps below:

1.import the logging module

2.configure the logging system

3.create a logger instance for every module or classes that needs logging.

4.Add log messages at appropriate places within your code.

5. Run your application and observe the log messages.

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:

Difference between logging and print statements in python are given below:

Logging Record events and errors that occur during the execution of Python programs and print statement Displays the information to the console for the debugging purposes.

Logging is Mainly used in the production environment whereas print is Mainly for debugging.

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!"

Answer:

def log\_message(message):

with open("app.log", "a") as file:

file.write(message + "\n")

log\_message("Hello, World!")

● The log level should be set to "INFO."

Answer:

import logging

def main():

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

logging.info('This is an INFO level log message.')

if \_\_name\_\_ == '\_\_main\_\_':

main()

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:

import logging

import traceback

import datetime

def main():

logging.basicConfig(level=logging.ERROR, format='%(asctime)s - %(levelname)s - %(message)s', filename='errors.log')

try:

raise ValueError("This is a sample exception")

except Exception as e:

print(f"Error: {e.\_\_class\_\_.\_\_name\_\_} - {e}")

logging.error(f"{e.\_\_class\_\_.\_\_name\_\_} - {e}\n{traceback.format\_exc()}")

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

main()