1. What is the role of try and exception block?

Ans: The "try" and "except" blocks are part of exception handling in many programming languages, including Python. They allow you to handle potential errors or exceptional situations in your code gracefully, without causing your program to terminate unexpectedly.

2. What is the syntax for a basic try-except block?

Ans:

try:

# Code that may raise an exception

# ...

except ExceptionType1:

# Code to handle ExceptionType1

# ...

except ExceptionType2:

# Code to handle ExceptionType2

# ...

except:

# Code to handle any other exceptions

# ...

finally:

# Code that will be executed regardless of whether an exception occurred or not

# ...

3. What happens if an exception occurs inside a try block and there is no matching

except block?

Ans: If an exception occurs inside a try block and there is no matching except block to handle that specific exception, the program will terminate abruptly, and the exception will propagate up the call stack until it reaches an appropriate exception handler. This behavior is known as an unhandled exception.

4. What is the difference between using a bare except block and specifying a specific

exception type?

Ans:

When handling exceptions in Python, there is a difference between using a bare except block and specifying a specific exception type.

Bare except block:

A bare except block is used to catch all exceptions that occur within the corresponding try block. It doesn't specify any particular exception type and acts as a catch-all. While this can be useful in certain situations, it is generally considered bad practice to use a bare except block without any specific reason. The main reason is that it makes it harder to diagnose and handle specific types of exceptions, potentially leading to unexpected behavior or hiding errors that should be addressed. It is generally recommended to catch and handle specific exceptions whenever possible.

Specifying a specific exception type:

When you specify a specific exception type in the except block, you are indicating that you want to handle only that particular type of exception. This allows you to have more fine-grained control over exception handling and enables you to take appropriate actions based on the type of error that occurred. By explicitly stating the exception type, you can handle different exceptions differently, providing more targeted error handling and making your code more robust and maintainable.

5. Can you have nested try-except blocks in Python? If yes, then give an example.

Ans:

es, it is possible to have nested try-except blocks in Python. Nesting try-except blocks allows you to handle exceptions at different levels of your code, providing more fine-grained error handling. Here's an example:

try:

# Outer try-except block

try:

# Inner try-except block

numerator = 10

denominator = 0

result = numerator / denominator

print("Result:", result)

except ZeroDivisionError:

print("Error: Division by zero occurred.")

finally:

print("Inner finally block executed.")

except:

print("An error occurred in the outer try block.")

finally:

print("Outer finally block executed.")

6. Can we use multiple exception blocks, if yes then give an example.

Ans:

Yes, in many programming languages, including Python, you can use multiple exception blocks to handle different types of exceptions separately. Each exception block can specify a different type of exception that you want to handle.

try:

# Some code that may raise exceptions

file = open("nonexistent\_file.txt", "r")

number = 10 / 0 # This will raise a ZeroDivisionError

except FileNotFoundError:

print("File not found!")

except ZeroDivisionError:

print("Cannot divide by zero!")

except Exception as e:

print("An error occurred:", e)

7. Write the reason due to which following errors are raised:

a. EOFError

b. FloatingPointError

c. IndexError

d. MemoryError

e. OverflowError

f. TabError

g. ValueError

Ans:

a. EOFError: This error is raised when the input() function reaches the end of the file (EOF) unexpectedly. It typically occurs when attempting to read data from a file or stream that doesn't have any more content.

b. FloatingPointError: This error is raised when a floating-point operation, such as division by zero or an invalid mathematical operation, results in an undefined or infinite value. It typically occurs when there are numerical computations involving floating-point numbers that cannot be accurately represented.

c. IndexError: This error is raised when you try to access an index of a sequence (such as a list or tuple) that is outside the valid range. It typically occurs when you use an index that is either negative or greater than or equal to the length of the sequence.

d. MemoryError: This error is raised when an operation fails due to insufficient memory available to perform the requested action. It typically occurs when a program tries to allocate more memory than the system can provide.

e. OverflowError: This error is raised when the result of an arithmetic operation exceeds the maximum representable value for a numeric type. It typically occurs when attempting to perform calculations that lead to a value that is too large to be stored.

f. TabError: This error is raised when there are inconsistencies or incorrect usage of indentation in Python code. It typically occurs when tabs and spaces are mixed improperly, especially when working with code blocks like loops or function definitions.

g. ValueError: This error is raised when a function receives an argument of the correct type but an invalid value. It typically occurs when the input data does not meet the expected conditions or constraints defined by the function. For example, passing a string to a function that expects an integer can raise a ValueError.

8. Write code for the following given scenario and add try-exception block to it.

a. Program to divide two numbers

try:

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

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

result = dividend / divisor

print("Result:", result)

except ZeroDivisionError:

print("Error: Cannot divide by zero")

b. Program to convert a string to an integer

try:

string\_num = input("Enter a number: ")

num = int(string\_num)

print("Number:", num)

except ValueError:

print("Error: Invalid number entered")

c. Program to access an element in a list

try:

my\_list = [1, 2, 3, 4, 5]

index = int(input("Enter the index to access: "))

element = my\_list[index]

print("Element:", element)

except IndexError:

print("Error: Index out of range")

d. Program to handle a specific exception

try:

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

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

result = num1 / num2

print("Result:", result)

except ValueError:

print("Error: Invalid input entered")

e. Program to handle any exception

try:

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

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

result = dividend / divisor

print("Result:", result)

except Exception as e:

print("Error:", e)