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

**Solution 1.**

The try and except blocks in Python are used for exception handling, which allows you to handle and respond to runtime errors or exceptional situations that may occur during the execution of your code.

The try block is used to enclose the code that might raise an exception. It is the block where you anticipate a potential error. If an exception occurs within the try block, the normal flow of the program is interrupted, and the code execution jumps to the except block.

The except block specifies the code that should be executed when a specific exception is raised. It handles the exception and allows you to provide an alternative course of action or handle the error gracefully. You can have multiple except blocks to handle different types of exceptions, allowing you to specify different handling mechanisms based on the type of error encountered.

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

**Solution 2.**

The basic syntax for a try-except block in Python is as follows:

try:

# Code that might raise an exception

# ...

except ExceptionType:

# Code to handle the exception

# ...

In this syntax:

The code that might raise an exception is enclosed within the try block.

ExceptionType is the specific type of exception that you want to catch and handle. It can be a built-in exception class (e.g., ValueError, TypeError, ZeroDivisionError) or a custom exception class that you have defined.

The except block is executed only if an exception of type ExceptionType (or its subclass) is raised within the try block.

Inside the except block, you write the code to handle the exception, which can include error messages, alternative actions, or any other logic you need.

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

**Solution 3.**

If there is no matching except block to handle the exception, the program will terminate, and the error message will be displayed to the user. It's important to handle exceptions appropriately to provide informative error messages or perform alternative actions when errors occur in your program.

1. What is the difference between using a bare except block and specifying a specific exception type?

**Solution 4.**

The difference between using a bare except block and specifying a specific exception type lies in the level of control and specificity in handling exceptions.

Bare except block: When you use a bare except block, without specifying any exception type, it acts as a catch-all for any type of exception that occurs within the try block. It will handle and catch all exceptions, regardless of their type. This can be convenient in some cases where you want to handle all exceptions in a similar manner, but it can also make it difficult to identify and debug specific errors since you don't know the exact exception type that occurred.

Specific exception type: On the other hand, when you specify a specific exception type in the except block, such as except ValueError, you are explicitly catching and handling only that particular type of exception. This allows you to have more fine-grained control over how different types of exceptions are handled. You can have multiple except blocks to handle different types of exceptions in different ways. This approach provides better clarity and allows you to handle different exceptions differently based on their specific requirements.

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

**Solution 5.**

Yes, you can have nested try-except blocks in Python. This means that you can have one try-except block inside another try block or except block.

Here's an example of nested try-except blocks:

try:

# Outer try block

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

num2 = int(input("Enter another number: "))

try:

# Inner try block

result = num1 / num2

print("Result:", result)

except ZeroDivisionError:

print("Error: Division by zero")

except ValueError:

print("Error: Invalid input")

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

**Solution 6.**

Yes, we can use multiple except blocks to handle different types of exceptions. Each except block can handle a specific exception type, allowing us to perform different actions based on the type of exception raised.

Here's an example that demonstrates the use of multiple except blocks:

try:

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

num2 = int(input("Enter another number: "))

result = num1 / num2

print("Result:", result)

except ValueError:

print("Error: Invalid input")

except ZeroDivisionError:

print("Error: Division by zero")

1. Write the reason due to which following errors are raised:
   1. EOFError
   2. FloatingPointError
   3. IndexError
   4. MemoryError
   5. OverflowError
   6. TabError
   7. ValueError

**Solution 7.**

a. EOFError: This error is raised when the input() function reaches the end of the file while trying to read input from the user. It typically occurs when there is an unexpected end of input.

b. FloatingPointError: This error is raised when a floating-point operation encounters an exceptional condition, such as division by zero or an invalid result.

c. IndexError: This error is raised when trying to access an index of a sequence (such as a list or string) that is outside the valid range. It occurs when an invalid index is used to access an element.

d. MemoryError: This error is raised when an operation fails due to insufficient memory. It occurs when the system is unable to allocate the required amount of memory for a program or operation.

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

f. TabError: This error is raised when there is an issue with the indentation of the code. It occurs when inconsistent or incorrect use of tabs and spaces is encountered, typically in the context of Python's indentation-based syntax.

g. ValueError: This error is raised when a function receives an argument of the correct type but with an invalid value. It occurs when an operation or function is called with an argument that is inappropriate or out of range.

1. Write code for the following given scenario and add try-exception block to it.
   1. Program to divide two numbers
   2. Program to convert a string to an integer
   3. Program to access an element in a list
   4. Program to handle a specific exception
   5. Program to handle any exception

**Solution 8.**

a. Program to divide two numbers:

try:

numerator = float(input("Enter the numerator: "))

denominator = float(input("Enter the denominator: "))

result = numerator / denominator

print("Result:", result)

except ZeroDivisionError:

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

b. Program to convert a string to an integer:

try:

num\_str = input("Enter a number: ")

num = int(num\_str)

print("Number:", num)

except ValueError:

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

c. Program to access an element in a list:

try:

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

index = int(input("Enter an index: "))

element = my\_list[index]

print("Element at index", index, ":", element)

except IndexError:

print("Error: Index out of range.")

d. Program to handle a specific exception:

try:

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

if age < 0:

raise ValueError("Age cannot be negative.")

print("Your age:", age)

except ValueError as e:

print("Error:", str(e))

e. Program to handle any exception:

try:

# Code that may raise an exception

pass

except Exception as e:

print("An error occurred:", str(e))