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

The try and except blocks in Python are used for exception handling, which allows you to gracefully handle errors or exceptional situations that might occur during the execution of your code. Here's how they work:

try Block: The code that might raise an exception is placed within a try block. It's a way to enclose potentially error-prone code in a controlled environment.

except Block: If an exception occurs inside the try block, the corresponding except block is executed. It handles the exception, preventing the program from crashing and providing an opportunity to handle the error or take corrective actions.

Example:

try:

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

result = 10 / num

print("Result:", result)

except ZeroDivisionError:

print("Error: Cannot divide by zero.")

except ValueError:

print("Error: Please enter a valid number.")

In this example, the try block takes user input, tries to perform a division, and prints the result. If the user enters zero (causing a ZeroDivisionError) or a non-numeric input (causing a ValueError), the corresponding except block handles the error and provides a user-friendly error message.

Using try and except blocks helps make your code more robust by preventing crashes and providing a mechanism to handle exceptional situations gracefully.

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

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

# ...

Here's a breakdown of the syntax:

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

After the try block, you can have one or more except blocks to handle specific types of exceptions.

If an exception of the specified type occurs within the try block, the corresponding except block is executed to handle the exception.

You can have multiple except blocks to handle different types of exceptions. Additionally, you can use a generic except block without specifying an exception type to catch any unexpected exceptions. However, it's generally a good practice to handle specific exceptions whenever possible.

Example:

try:

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

result = 10 / num

print("Result:", result)

except ZeroDivisionError:

print("Error: Cannot divide by zero.")

except ValueError:

print("Error: Please enter a valid number.")

In this example, the try block takes user input, tries to perform a division, and handles ZeroDivisionError and ValueError exceptions with separate except blocks.

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

If an exception occurs inside a try block and there is no matching except block to handle that specific type of exception, the exception will not be caught and will propagate up the call stack. This can lead to a program crash with an error message indicating the unhandled exception.

In other words, if there's no appropriate except block to handle the exception type raised within the try block, the exception will "escape" the try block and potentially terminate the program.

To prevent this from happening, it's a good practice to include a generic except block that catches unexpected exceptions and provides appropriate error handling or logging. However, relying solely on a generic except block without proper handling for specific exception types might hide valuable information about what went wrong in your program.

Example:

try:

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

result = 10 / num

print("Result:", result)

except ValueError:

print("Error: Please enter a valid number.")

In this example, if the user enters zero, causing a ZeroDivisionError, and there's no corresponding except ZeroDivisionError block, the program will terminate with an unhandled exception.

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

Using a bare except block and specifying a specific exception type have distinct differences in how they handle exceptions:

Bare except Block:

A bare except block catches all exceptions, regardless of their type.

It can make it difficult to diagnose and troubleshoot specific issues because you lose information about the type of exception that occurred.

It's generally not recommended to use a bare except block, as it can lead to hiding errors and unexpected behavior.

Example:

try:

# ...

except:

# This block catches all exceptions, including those you might not expect.

# Handle exceptions here.

Specific Exception Type:

Specifying a specific exception type in an except block allows you to catch and handle only that particular type of exception.

It provides better control over exception handling, making it easier to understand and diagnose issues.

It's recommended to use specific except blocks to handle expected exceptions and provide appropriate error handling or recovery.

Example

try:

# ...

except ValueError:

# This block catches only ValueError exceptions.

# Handle ValueError exceptions here.

In summary, using a specific exception type in an except block is a better practice because it provides more accurate and controlled handling of exceptions. Using a bare except block should generally be avoided to ensure proper debugging and error management.

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

Yes, you can have nested try and except blocks in Python. This means you can place one try block inside another, and each try block can have its own set of corresponding except blocks. This allows you to handle different levels of exceptions and provide more specific error handling for different parts of your code.

Example:

try:

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

try:

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

result = numerator / denominator

except ZeroDivisionError:

print("Error: Cannot divide by zero.")

except ValueError:

print("Error: Please enter valid numeric inputs.")

In this example, there are nested try blocks. The outer try block handles potential ValueError exceptions that might occur when getting the numerator input. Inside the outer try block, there's an inner try block that handles potential ZeroDivisionError exceptions that might occur when getting the denominator input. Each level of exception is handled independently with appropriate error messages.

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

Yes, you can use multiple except blocks to handle different types of exceptions in Python. Each except block can be used to catch and handle a specific type of exception. Here's an example:

try:

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

result = 10 / num

print("Result:", result)

except ZeroDivisionError:

print("Error: Cannot divide by zero.")

except ValueError:

print("Error: Please enter a valid number.")

except Exception as e:

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

In this example:

* The first except ZeroDivisionError block handles the division by zero error.
* The second except ValueError block handles the error that might occur if the user enters a non-numeric value.
* The third except Exception block serves as a catch-all for any other unexpected exceptions and provides a general error message along with the exception information.
* It's a good practice to use specific except blocks for expected exceptions and provide more general error handling for unexpected exceptions using a generic except block like except Exception.

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

1. **EOFError**
2. **FloatingPointError**
3. **IndexError**
4. **MemoryError**
5. **OverflowError**
6. **TabError**
7. **ValueError**

**a. EOFError:**

* Raised when the input() function reaches the end of file (EOF) condition.
* Occurs if you're trying to read from the input and the input stream is exhausted.

**b. FloatingPointError:**

* Raised when a floating-point operation (like division by zero) results in an undefined or infinite value.
* Occurs when there's an issue with arithmetic operations involving floating-point numbers.

**c. IndexError:**

* Raised when trying to access an index that is out of range of a sequence (like a list, tuple, or string).
* Occurs when attempting to access an element that doesn't exist at a particular index.

**d. MemoryError:**

* Raised when an operation cannot be completed due to a lack of available memory.
* Occurs when the system does not have enough memory to allocate for an operation.

**e. OverflowError:**

* Raised when an arithmetic operation exceeds the limits of the data type.
* Occurs when attempting to perform a calculation that produces a result outside the range that can be represented.

**f. TabError:**

* Raised when inconsistent use of tabs and spaces is encountered in the indentation of code.
* Occurs when there's a mix of tabs and spaces in an indented block, which can confuse Python's parser.

**g. ValueError:**

* Raised when a function receives an argument of correct data type but an inappropriate value.
* Occurs when the input value is not suitable for the operation being performed.

**8. 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**

the code for each scenario along with added try-except blocks:

**a. Program to Divide Two Numbers:**

try:

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

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

result = numerator / denominator

print("Result:", result)

except ZeroDivisionError:

print("Error: Cannot divide by zero.")

except ValueError:

print("Error: Please enter valid numeric inputs.")

except Exception as e:

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

**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 number.")

**c. Program to Access an Element in a List:**

try:

my\_list = [1, 2, 3]

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

value = my\_list[index]

print("Value at index", index, ":", value)

except IndexError:

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

except ValueError:

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

**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 ve:

print("Error:", ve)

**e. Program to Handle Any Exception:**

try:

value = int(input("Enter a value: "))

result = 10 / value

print("Result:", result)

except Exception as e:

print("An error occurred:", e)

These examples demonstrate various ways to use try-except blocks to handle different types of exceptions in Python.