***Exception Handling-1***

**Q1. What is an Exception in python? Write the difference between Exceptions and syntax errors.**

In Python, an Exception is an error that occurs during the execution of a program. When an exception is encountered, the normal flow of execution is disrupted and the program terminates, unless the exception is caught and handled appropriately.

Exceptions can occur for a variety of reasons, such as invalid input, insufficient resources, or unexpected conditions. Some common types of exceptions in Python include ValueError, TypeError, and IOError.

On the other hand, syntax errors are errors that occur when the Python interpreter encounters code that violates the rules of the Python language. These errors occur during the parsing of the code, before the code is executed. Examples of syntax errors include misspelled keywords, missing parentheses, and incorrect indentation.

The main difference between exceptions and syntax errors is that syntax errors are caught by the Python interpreter before the code is executed, whereas exceptions occur during the execution of the code. Syntax errors indicate a problem with the structure of the code, while exceptions indicate a problem with the runtime behavior of the code.

In summary, exceptions are errors that occur during the execution of a program, while syntax errors are errors that occur during the parsing of the code.

**Q2. What happens when an exception is not handled? Explain with an example.**

When an exception is not handled in Python, it results in the termination of the program and the display of an error message that describes the exception that occurred. This error message can help the programmer to identify the cause of the exception and fix the problem in the code.

Here is an example that demonstrates what happens when an exception is not handled:

try:

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

y = 10 / x

print("The result is:", y)

except ValueError:

print("Invalid input. Please enter a number.")

In this example, the program prompts the user to enter a number and then attempts to divide 10 by the input value. If the user enters a non-numeric value, a ValueError exception is raised and the program displays an error message asking the user to enter a number.

However, if we remove the try-except block and attempt to run the code without handling the exception, like this:

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

y = 10 / x

print("The result is:", y)

If the user enters a value of 0 or a non-numeric value, a ZeroDivisionError or a ValueError exception will be raised, respectively. Since we did not handle these exceptions, the program will terminate and display an error message that describes the exception that occurred.

For example, if the user enters 0, the program will terminate and display the following error message:

Traceback (most recent call last):

File "<stdin>", line 2, in <module>

ZeroDivisionError: division by zero

This error message indicates that a ZeroDivisionError exception was raised because we attempted to divide by zero. Since we did not handle this exception, the program terminated without completing its execution.

**Q3. Which Python statements are used to catch and handle exceptions? Explain with an example.**

In Python, the try and except statements are used to catch and handle exceptions.

The try block contains the code that may raise an exception. If an exception occurs within the try block, the code execution is immediately transferred to the except block. The except block is responsible for handling the exception by providing an appropriate response to the error condition.

Here is an example that demonstrates the use of try and except statements to catch and handle an exception:

try:

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

y = 10 / x

print("The result is:", y)

except ValueError:

print("Invalid input. Please enter a number.")

except ZeroDivisionError:

print("Cannot divide by zero.")

In this example, the try block contains the code that prompts the user to enter a number, converts the input to an integer, and then divides 10 by the input value. If the user enters a non-numeric value, a ValueError exception is raised and the program jumps to the except ValueError block, which prints an error message to the console.

If the user enters a value of 0, a ZeroDivisionError exception is raised and the program jumps to the except ZeroDivisionError block, which prints an error message indicating that division by zero is not allowed.

By handling these exceptions with appropriate error messages, the program can continue to execute and prompt the user to enter a valid input value.

It's also possible to use a single except block to catch multiple types of exceptions, like this:

try:

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

y = 10 / x

print("The result is:", y)

except (ValueError, ZeroDivisionError):

print("Invalid input. Please enter a non-zero number.")

In this example, the except block catches both ValueError and ZeroDivisionError exceptions and prints a single error message indicating that the input value is invalid.

**Q4. Explain with an example:**

1. **try and else**
2. **finally**
3. **raise**

try and else statement:

In Python, the else statement can be used in conjunction with the try statement to specify a block of code to be executed if no exceptions are raised in the try block. Here's an example:

try:

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

y = 10 / x

except ValueError:

print("Invalid input. Please enter a number.")

else:

print("The result is:", y)

In this example, the try block prompts the user to enter a number, converts it to an integer, and then divides 10 by the input value. If the user enters a non-numeric value, a ValueError exception is raised and the program jumps to the except block, which prints an error message. If no exception is raised, the else block is executed and the result is printed to the console.

finally statement:

The finally block in Python is used to specify a block of code that will always be executed, regardless of whether an exception is raised or not. Here's an example:

try:

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

y = 10 / x

except ValueError:

print("Invalid input. Please enter a number.")

else:

print("The result is:", y)

finally:

print("Thank you for using this program.")

In this example, the finally block is used to print a message to the console, indicating that the program has completed execution, regardless of whether an exception was raised or not.

raise statement:

The raise statement in Python is used to explicitly raise an exception. Here's an example:

try:

x = int(input("Enter a number between 1 and 10: "))

if x < 1 or x > 10:

raise ValueError("The value must be between 1 and 10.")

except ValueError as e:

print(e)

In this example, the raise statement is used to raise a ValueError exception if the user enters a number that is outside the range of 1 to 10. The exception message is specified in the raise statement, and is printed to the console in the except block.

**Q5. What are Custom Exceptions in python? Why do we need Custom Exceptions? Explain with an example.**

Custom exceptions in Python are user-defined exceptions that can be raised in response to specific error conditions in a program. Custom exceptions are useful when the built-in exceptions provided by Python are not sufficient to capture the specific error condition that needs to be handled.

Custom exceptions can be defined by creating a new class that inherits from the Exception class, or from one of its subclasses. The custom exception class can then be used to raise exceptions in the same way as built-in exceptions.

Here's an example of defining and using a custom exception class in Python:

class NegativeNumberError(Exception):

pass

def square\_root(x):

if x < 0:

raise NegativeNumberError("Cannot calculate the square root of a negative number.")

else:

return x \*\* 0.5

try:

result = square\_root(-4)

print("The square root is:", result)

except NegativeNumberError as e:

print("Error:", e)

In this example, a custom exception class called NegativeNumberError is defined by creating a new class that inherits from the Exception class. The square\_root function checks if the input value is negative and raises a NegativeNumberError exception if it is.

The try block calls the square\_root function with a negative input value, which raises a NegativeNumberError exception. The except block catches the exception and prints the error message to the console.

Custom exceptions provide a way to capture specific error conditions in a program and handle them appropriately, by providing informative error messages or performing specific error recovery operations. They help to make programs more robust and easier to debug by providing clear information about what went wrong and why.

**Q6.Create a custom exception class. Use this class to handle an exception.**

here's an example of defining a custom exception class and using it to handle an exception:

class InvalidAgeError(Exception):

"""Custom exception for invalid age"""

pass

def validate\_age(age):

if age < 18:

raise InvalidAgeError("Age must be greater than or equal to 18")

else:

print("Age is valid")

try:

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

validate\_age(age)

except InvalidAgeError as e:

print("Error:", e)

In this example, we have defined a custom exception class called InvalidAgeError by creating a new class that inherits from the Exception class. The validate\_age function checks if the input age is less than 18 and raises an InvalidAgeError exception if it is.

In the try block, we get the user's age as input and call the validate\_age function with the input value. If the age is less than 18, the validate\_age function raises an InvalidAgeError exception which is caught by the except block. The error message is then printed to the console.

If the age is valid (greater than or equal to 18), the validate\_age function prints a message indicating that the age is valid.