Q1. What is an Exception in python ? Write the difference between Exception and Syntax errors.

**In Python, an exception is an event that occurs during the execution of a program that disrupts the normal flow of the program's instructions. When an exception occurs, the program execution is immediately halted, and Python raises an exception object. This object contains information about the type of exception and where it occurred in the program.**

**Exceptions are used to handle errors or exceptional situations that may arise during program execution. They allow you to catch and handle specific types of errors, perform appropriate actions, and gracefully recover from exceptional conditions. By using exception handling, you can prevent your program from crashing and provide a fallback mechanism to handle errors in a controlled manner.**

**Syntax errors, on the other hand, are errors that occur when you write code that violates the Python syntax rules. They are detected by the Python interpreter during the parsing phase, before the program is executed. Syntax errors usually indicate typos, missing or misplaced punctuation, or incorrect language constructs.**

**The main differences between exceptions and syntax errors are as follows:**

1. **Detection: Syntax errors are detected by the Python interpreter during the parsing phase before the program is executed. Exceptions, on the other hand, occur during program execution when an exceptional condition arises.**
2. **Timing: Syntax errors prevent the program from running at all, whereas exceptions can occur at any point during program execution.**
3. **Handling: Syntax errors require fixing the code and correcting the syntax violation before the program can be executed. Exceptions, on the other hand, can be handled using exception handling mechanisms, such as try-except blocks, to catch and handle the exceptional condition gracefully.**
4. **Types: Syntax errors are usually generic and do not have specific types associated with them. Exceptions, on the other hand, have specific types that categorize the exceptional condition, such as ValueError, TypeError, FileNotFoundError, etc. These types can be caught individually to handle different types of exceptions differently.**

**In summary, exceptions are raised during program execution to handle exceptional conditions, while syntax errors occur due to violations of Python's syntax rules and prevent the program from running. Exception handling allows you to catch and handle exceptions gracefully, providing a fallback mechanism to recover from errors and continue program execution.**

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

**An exception is not handled, it results in an "unhandled exception" or an "uncaught exception." This means that the exception is not caught and processed by any part of the program, causing the program to terminate abruptly and potentially display an error message or stack trace. The exact behavior can vary depending on the programming language and runtime environment.**

**Let's consider an example in Python:**

**def divide(a, b):**

**return a / b**

**numerator = 10**

**denominator = 0**

**result = divide(numerator, denominator)**

**print("Result:", result)**

a function called divide() that divides two numbers. We attempt to divide numerator by denominator, which is 0. This operation will raise a ZeroDivisionError because dividing by zero is mathematically undefined.

If the exception is not handled, the program will terminate and display an error message like the following:

**Traceback (most recent call last):**

**File "example.py", line 6, in <module>**

**result = divide(numerator, denominator)**

**File "example.py", line 2, in divide**

**return a / b**

**ZeroDivisionError: division by zero**

the exception occurred at the line return a / b in the divide() function, and since it was not handled, the program terminated immediately. The error message provides information about the type of exception (ZeroDivisionError) and the line number where it occurred.

If we want to handle the exception, you can use a try-except block to catch and handle it appropriately:

**def divide(a, b):**

**try:**

**result = a / b**

**return result**

**except ZeroDivisionError:**

**print("Error: Cannot divide by zero!")**

**return None**

**numerator = 10**

**denominator = 0**

**result = divide(numerator, denominator)**

**if result is not None:**

**print("Result:", result)**

In this modified example, the divide() function is wrapped in a try block, and we specifically catch the ‘ZeroDivisionError’. If the exception occurs, we print an error message, return None, and continue the program execution. This way, the program doesn't terminate abruptly, and we have a chance to handle the error gracefully.

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

In Python, the statements used to catch and handle exceptions are try, except, else, and finally. They allow you to handle errors or exceptional situations that may occur during the execution of your program. Here's an explanation of each statement with an example:

1. try block: The code that potentially raises an exception is placed inside a try block. If an exception occurs within the try block, the execution is immediately transferred to the corresponding except block.
2. except block: This block is used to handle the exception raised in the try block. You can specify the type of exception you want to catch or use a generic Exception class to catch any exception. You can also have multiple except blocks to handle different types of exceptions.
3. else block (optional): The else block is executed if no exceptions occur in the try block. It is generally used for code that should run only when no exceptions are raised.
4. finally block (optional): This block is always executed, whether an exception occurs or not. It is used to define cleanup actions, such as closing files or releasing resources.

Here's an example that demonstrates the usage of these exception handling statements:

**try:**

**# Code that may raise an exception**

**x = 10**

**y = 0**

**result = x / y # Division by zero will raise a ZeroDivisionError**

**except ZeroDivisionError:**

**# Handling specific exception**

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

**except Exception as e:**

**# Handling any other exception**

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

**else:**

**# Executed if no exceptions occur**

**print("Result:", result)**

**finally:**

**# Cleanup actions**

**print("Cleanup: Closing resources.")**

**# Output:**

**# Error: Division by zero is not allowed!**

**# Cleanup: Closing resources.**

Here's an example that demonstrates the usage of these exception handling statements:

**try:**

**# Code that may raise an exception**

**x = 10**

**y = 0**

**result = x / y # Division by zero will raise a ZeroDivisionError**

**except ZeroDivisionError:**

**# Handling specific exception**

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

**except Exception as e:**

**# Handling any other exception**

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

**else:**

**# Executed if no exceptions occur**

**print("Result:", result)**

**finally:**

**# Cleanup actions**

**print("Cleanup: Closing resources.")**

**# Output:**

**# Error: Division by zero is not allowed!**

**# Cleanup: Closing resources.**

In this example, the code inside the try block attempts to divide x by y. Since y is set to 0, a ZeroDivisionError is raised. The program then transfers the execution to the corresponding except block, which handles the exception by printing an error message.

If the division were successful, the program would execute the else block and print the result. Finally, regardless of whether an exception occurred or not, the finally block is executed to perform any necessary cleanup actions.

By using these exception handling statements, we can gracefully handle errors and exceptions in your Python programs, making them more robust and reliable.

Q4. Explain with an example:

a. try and else

b. finally

c. raise

a. try and else:

In Python, try and else are used together to handle exceptions and provide alternative code execution paths. The try block is used to enclose the code that may raise an exception, and the else block is executed only if no exceptions occur in the try block.

Here's an example to illustrate how try and else work together:

**def divide\_numbers(a, b):**

**try:**

**result = a / b**

**except ZeroDivisionError:**

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

**else:**

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

**# Example usage:**

**divide\_numbers(6, 2) # Output: The result of the division is: 3.0**

**divide\_numbers(6, 0) # Output: Error: Cannot divide by zero**

In the above example, the divide\_numbers function takes two numbers, a and b, and attempts to divide a by b. Inside the try block, the division operation result = a / b is performed. If no exception occurs, the code inside the else block is executed, which prints the result of the division.

If a ZeroDivisionError exception occurs, indicating an attempt to divide by zero, the code inside the except block is executed instead, printing an error message. The else block is skipped in this case.

b. finally:

In Python, the finally block is used in conjunction with the try block to specify a piece of code that will always be executed, regardless of whether an exception occurs or not. It ensures that certain cleanup actions or finalization steps are performed, regardless of the outcome of the try block.

Here's an example to demonstrate the usage of finally:

**file = None**

**try:**

**file = open("example.txt", "r")**

**# Perform some operations on the file**

**print(file.read())**

**except FileNotFoundError:**

**print("Error: File not found")**

**finally:**

**if file:**

**file.close()**

In the above example, we attempt to open a file named "example.txt" for reading. If the file is successfully opened, we perform some operations (in this case, reading the contents of the file) inside the try block. If a FileNotFoundError occurs because the file doesn't exist, an error message is printed in the except block.

Regardless of whether an exception occurs or not, the finally block is always executed. In this case, we ensure that the file is closed using the close() method, which is called inside the finally block. This ensures that the file is properly closed and any associated resources are released, even if an exception occurs during the execution of the try block.

c. raise:

In Python, the raise keyword is used to explicitly raise an exception. It allows you to manually trigger exceptions when certain conditions are met. You can raise built-in exceptions or create your own custom exceptions.

Here's an example to illustrate the usage of raise:

**def validate\_age(age):**

**if age < 0:**

**raise ValueError("Age cannot be negative")**

**elif age < 18:**

**raise ValueError("You must be at least 18 years old")**

**# Example usage:**

**try:**

**validate\_age(15)**

**except ValueError as error:**

**print(error)**

In the above example, the validate\_age function checks if the provided age is valid. If the age is negative or less than 18, the function raises a `Value

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

In Python, custom exceptions allow you to define your own exception classes based on your specific needs. They are derived from the base Exception class or any other built-in exception class. Custom exceptions are useful when you want to handle specific types of errors in your code and provide meaningful error messages or perform specific actions when those errors occur.

Here's an example to illustrate the need for custom exceptions:

**class InsufficientFundsError(Exception):**

**"""Exception raised for insufficient funds in an account."""**

**def \_\_init\_\_(self, balance, amount):**

**self.balance = balance**

**self.amount = amount**

**self.message = f"Insufficient funds: Available balance is {balance}, but {amount} required."**

**super().\_\_init\_\_(self.message)**

**def withdraw(balance, amount):**

**if amount > balance:**

**raise InsufficientFundsError(balance, amount)**

**else:**

**# Perform withdrawal operation**

**print("Withdrawal successful!")**

**# Example usage**

**try:**

**account\_balance = 1000**

**withdrawal\_amount = 1500**

**withdraw(account\_balance, withdrawal\_amount)**

**except InsufficientFundsError as e:**

**print(e.message)**

**# Additional actions or error handling specific to InsufficientFundsError can be performed here**

In this example, I define a custom exception called InsufficientFundsError that inherits from the base Exception class. It takes the current account balance and the requested withdrawal amount as arguments and generates an error message indicating the insufficient funds.

The withdraw function uses this custom exception to raise an error if the withdrawal amount exceeds the available balance. If the balance is sufficient, it performs the withdrawal operation.

In the example usage, I attempt to withdraw an amount that exceeds the account balance. This triggers the InsufficientFundsError to be raised, and we catch the exception using a try-except block. We can access the error message and handle the exception accordingly.

Custom exceptions help in organizing and clarifying your code by distinguishing different types of errors. They provide a way to handle specific exceptions separately, allowing you to take appropriate actions or provide detailed information to the user about the error.

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

an example of creating a custom exception class and using it to handle an exception:

**class CustomException(Exception):**

**def \_\_init\_\_(self, message):**

**self.message = message**

**super().\_\_init\_\_(self.message)**

**try:**

**raise CustomException("This is a custom exception.")**

**except CustomException as e:**

**print("Custom Exception Caught:", e.message)**

In the code above, define a custom exception class called CustomException by inheriting from the base Exception class. The CustomException class has an \_\_init\_\_ method that takes a message parameter and sets it as an attribute of the exception. The super().\_\_init\_\_(self.message) line ensures that the exception is properly initialized.

Next, use a try-except block to catch the exception. We raise an instance of CustomException using the raise statement, passing the desired error message as an argument. If the CustomException is raised, the code within the except block will be executed.

Inside the except block, we catch the CustomException and print the custom error message using the e.message attribute.