Q1 What is an exception in python? Write the difference between exception and syntax errors.

ANS: In Python, an exception is an error that occurs during the execution of a program. When a statement or expression in a Python script causes an error at runtime, Python raises an exception. This interrupts the normal flow of the program if the error is not handled properly.

Syntax Errors:

Syntax errors, also known as parsing errors, occur when the code is not valid Python syntax. These errors are detected by the Python parser when it is trying to interpret the script. Examples include missing parentheses or quotes, incorrect indentation, or using Python keywords incorrectly. Syntax errors prevent the script from running at all.

Exceptions: Exceptions, on the other hand, occur when the script is syntactically correct but encounters an error during execution. These can happen for various reasons such as trying to access a file that does not exist, dividing by zero, or calling a method on an object that doesn't support it. Unlike syntax errors, exceptions are not necessarily fatal to the program—they can be handled using try-except blocks to gracefully manage the error and continue execution if possible.

Key Differences:

Occurrence:

- **Syntax Errors:** Detected by the Python parser during the compilation phase before the code runs. They prevent the script from executing.
- **Exceptions:** Occur during the execution of the program, when Python encounters an error condition that it cannot cope with automatically.

Cause:

- • **Syntax Errors:** Due to incorrect syntax in the code.
- **Exceptions:** Due to runtime conditions such as unexpected user input, network issues, or arithmetic errors.

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

Ans: When an exception is not handled in Python, it typically leads to the termination of the program abruptly. Here's what happens:

Exception Propagation: If an exception occurs in your Python code and is not caught (handled) by a try-except block or similar mechanism, Python will propagate the exception up the call stack. This means the exception moves from the point where it occurred up through the calling chain of functions or methods until it either gets handled or reaches the top level of the program.

Program Termination: If the exception reaches the top level of the program without being caught, Python prints a traceback (which shows the sequence of function calls that led to the exception) and terminates the program.

Example :

```
def divide(x, y):

    result = x / y

    return result

result = divide(10, 0)

print(result)
```

Q3 Which python statements are used to catch and handled exception ? explain with an example.

Ans: **Try and except Statements**

The try statement allows you to define a block of code where exceptions may occur. You can then use one or more except clauses immediately following the try block to specify how to handle specific exceptions that might occur within the try block.

Syntax:

```
try:

    # Code that may raise exceptions

    # ...

except ExceptionType1 as e1:

    # Handle exception of type ExceptionType1

    # e1 is an instance of ExceptionType1

    # Optional: you can access information about the exception using `e1`

except ExceptionType2 as e2:

    # Handle exception of type ExceptionType2

    # e2 is an instance of ExceptionType2

    # Optional: you can access information about the exception using `e2`

else:
    # Optional block that executes if no exception is raised in the try block

finally:

    # Optional block that always executes regardless of whether an exception occurred or not
```

EXAMPLE:

```
def divide(x, y):

    try:

        result = x / y

    except ZeroDivisionError:

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

    except TypeError as e:

        print(f"Error: {e}")

    else:

        print(f"The result of {x} / {y} is {result}.")

    finally:

        print("Division operation completed.”)

divide(10, 2)

divide(10, 0)

divide('10', 2)
```

Q4 Explain with an example.

a) **Try and else**
b) **Finally**
c) **Ralse**

Ans: Try and else:

The else block in a try statement is executed if no exceptions are raised in the try block. It is typically used to perform actions that should only occur if the try block completes successfully without any exceptions being raised.

Example:

```
def divide(x, y):

    try:

        result = x / y

    except ZeroDivisionError:

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

    else:

        print(f"The result of {x} / {y} is {result}.”

divide(10, 2)   divide(10, 0)
```

FINALLY:

The finally block in a try statement is always executed, regardless of whether an exception was raised or not. It is typically used to ensure that some cleanup or finalization steps are executed, such as closing files or releasing resources.

EXAMPLE:

```
def read_file(filename):

    try:

        file = open(filename, 'r')

        content = file.read()

        print(f"File content: {content}")

    except FileNotFoundError:

        print(f"Error: File '{filename}' not found.")

    finally:

        if 'file' in locals():
```

```
        file.close()

read_file('example.txt')

read_file('nonexistent.txt')
```

RALSE:

The raise statement in Python is used to explicitly raise an exception. You can raise built-in exceptions or custom exceptions to indicate error conditions or specific situations in your code.

EXAMPLE:

```
def greet(name):

    if not isinstance(name, str):

        raise TypeError("Name must be a string.")

    print(f"Hello, {name}!"

greet("Alice")

greet(123)
```

Q5 What are Custom exception in python? Why do we need custom exceptions? Explain with an example.

ANS: In Python, custom exceptions are user-defined exceptions that extend the built-in exceptions provided by Python. They allow you to create specific exception classes tailored to your application's needs, providing more meaningful error messages and allowing for specialized handling of exceptional conditions.

Why do we need custom exceptions?

Custom exceptions are useful for several reasons:

1. **Clarity and Readability:** By defining custom exceptions, you can provide more descriptive error messages that clearly indicate what went wrong in your application.
2. **Modularity and Extensibility:** Custom exceptions help in organizing and managing exception handling logic in a modular way. They can encapsulate specific error conditions related to your application domain.
3. **Error Handling Control:** With custom exceptions, you gain more control over how errors are propagated and handled throughout your codebase. This can lead to more robust error handling strategies.

- **Explanation:**

  - InsufficientFundsError is a custom exception class that inherits from Python's built-in Exception class.
  - It takes amount and balance as parameters to provide a detailed error message when raised.

□ **Usage in Account class:**

  - The Account class has a withdraw method that attempts to withdraw a specified amount from the account.
  - If the amount exceeds the balance, it raises InsufficientFundsError.

Handling the exception:

  - In the example, a try-except block is used to catch InsufficientFundsError.
  - When caught, the error message (e) is printed, providing clear information about the attempted withdrawal amount and the current account balance.

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

ANS: # Custom exception class

```
class CustomException(Exception):

    def __init__(self, message):

        self.message = message

        super().__init__(self.message)

def example_function(x):

    if x < 0:

        raise CustomException("Input value cannot be negative.")

try:

    value = -10

    example_function(value)

except CustomException as e:

    print(f"Custom Exception caught: {e.message}")
```