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

The role of a try-except block in Python is to handle exceptions or errors that may occur during the execution of a program. By using a try-except block, you can write code that attempts a risky operation (the "try" block) and, if an exception occurs, gracefully handle the error in the "except" block, preventing the program from crashing.

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

```
try:
    # Code that may raise an exception
except ExceptionType:
    # Code to handle the exception
```

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 exception type, the program will terminate, and Python will print a traceback to the standard error output (usually the console or terminal). This traceback provides information about the exception and the point in the code where it occurred.

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

using specific exception types in the except block is preferred because it allows for targeted error handling and makes it easier to understand and maintain the code. Avoid using bare except blocks as they can hide errors and make it harder to identify and fix problems in your Python programs.

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

Yes, you can have nested try-except blocks in Python. This means you can place one try-except block inside another try or except block. Nesting try-except blocks allows you to handle exceptions at different levels of granularity, providing more specific error handling for different parts of your code.

```
def divide_numbers(a, b):
    try:
        result = a / b
        print("Division result:", result)
    except ZeroDivisionError:
        print("Cannot divide by zero.")

try:
    num1 = int(input("Enter the first number: "))
    try:
        num2 = int(input("Enter the second number: "))
        divide_numbers(num1, num2)
    except ValueError:
        print("Invalid input. Please enter a valid number.")
```

```
except KeyboardInterrupt:
    print("\nOperation interrupted by the user.")
```

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

Yes, you can use multiple exception blocks in Python to handle different types of exceptions separately. Each exception block will catch the specific type of exception it's designed for, allowing you to provide customized error handling for each exception type.

```
def divide_numbers(a, b):
    try:
        result = a / b
        print("Division result:", result)
    except ZeroDivisionError:
        print("Cannot divide by zero.")
    except ValueError:
        print("Invalid input. Please enter a valid number.")
    except TypeError:
        print("Unsupported data type. Please provide valid
numeric values.")
    except Exception as e:
        print("An unexpected error occurred:", e)

try:
    num1 = int(input("Enter the first number: "))
    num2 = int(input("Enter the second number: "))
    divide_numbers(num1, num2)
except KeyboardInterrupt:
    print("\nOperation interrupted by the user.")
```

7. **Write the reason due to which following errors are raised:**
    a. **EOFError**
    b. **FloatingPointError**
    c. **IndexError**
    d. **MemoryError**
    e. **OverflowError**
    f. **TabError**
    g. **ValueError**

a. EOFError:

Raised when an input function (like input() or raw_input()) reaches the end of the file (EOF) and no more data can be read.

b. FloatingPointError:

Raised when an invalid or exceptional floating-point operation occurs, such as division by zero or an operation resulting in an overflow or underflow.

c. IndexError:
Raised when trying to access an index in a sequence (e.g., list, tuple, or string) that is out of range, i.e., an index that does not exist.

d. MemoryError:
Raised when an operation cannot be completed due to insufficient memory to allocate a required object or data.

e. OverflowError:
Raised when the result of an arithmetic operation is too large to be represented within the numeric type being used (e.g., exceeding the maximum value of an integer).

f. TabError:
Raised when there is an inconsistent use of tabs and spaces for indentation within the same block of code, causing an indentation error.

g. ValueError:
Raised when a built-in operation or function receives an argument with the correct type but an inappropriate value (e.g., trying to convert a non-numeric string to an integer).

8. **Write code for the following given scenario and add try-exception block to it.**
    a. **Program to divide two numbers**
    b. **Program to convert a string to an integer**
    c. **Program to access an element in a list**
    d. **Program to handle a specific exception**
    e. **Program to handle any exception**

a. Program to divide two numbers

```
def divide_numbers(a, b):
    try:
        result = a / b
        print("Division result:", result)
    except ZeroDivisionError:
        print("Cannot divide by zero.")
    except Exception as e:
        print("An unexpected error occurred:", e)

try:
    num1 = int(input("Enter the first number: "))
    num2 = int(input("Enter the second number: "))
    divide_numbers(num1, num2)
except KeyboardInterrupt:
    print("\nOperation interrupted by the user.")
```

b. Program to convert a string to an integer

```
def convert_to_integer(s):
    try:
        num = int(s)
        print("Integer value:", num)
    except ValueError:
        print("Invalid input. Please enter a valid integer.")
    except Exception as e:
        print("An unexpected error occurred:", e)

try:
    input_str = input("Enter a number: ")
    convert_to_integer(input_str)
except KeyboardInterrupt:
    print("\nOperation interrupted by the user.")
```

c. Program to access an element in a list

```
def access_list_element(lst, index):
    try:
        value = lst[index]
        print("Value at index", index, ":", value)
    except IndexError:
        print("Index out of range. Please provide a valid
index.")
    except Exception as e:
        print("An unexpected error occurred:", e)

try:
    my_list = [10, 20, 30, 40]
    index = int(input("Enter an index: "))
    access_list_element(my_list, index)
except KeyboardInterrupt:
    print("\nOperation interrupted by the user.")
```

d. Program to handle a specific exception

```
try:
    num1 = int(input("Enter the first number: "))
    num2 = int(input("Enter the second number: "))
    result = num1 / num2
    print("Division result:", result)
except ZeroDivisionError:
    print("Cannot divide by zero.")
except ValueError:
    print("Invalid input. Please enter valid numbers.")
except Exception as e:
```

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

e. Program to handle any exception

```
try:
    num1 = int(input("Enter the first number: "))
    num2 = int(input("Enter the second number: "))
    result = num1 / num2
    print("Division result:", result)
except Exception as e:
    print("An error occurred:", e)
```