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

Ans: The try and except block is used in programming to handle exceptions, which are unexpected or exceptional events that occur during the execution of a program. The try block contains the code that might raise an exception, and the except block defines the code that is executed if an exception occurs.

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

Ans:
```
try:
    # Code that might raise an exception
    # ...
except ExceptionType1:
    # Code to handle exceptions of type ExceptionType1
    # ...
except ExceptionType2:
    # Code to handle exceptions of type ExceptionType2
    # ...
else:
    # Optional code that runs if no exception is raised in the try block
    # ...
finally:
    # Optional code that always runs, whether an exception is raised or not
    # ...
```

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

Ans: If an exception occurs inside a try block and there is no matching except block to handle that specific exception, the exception will propagate up the call stack until it reaches an appropriate except block that can handle it. If no such except block is found anywhere in the call stack, the program will terminate, and an error message or stack trace will be displayed.

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

Ans: Bare `except` block: A bare `except` block is used to catch any exception that occurs within the corresponding `try` block. It does not specify a particular exception type to be caught. This means that any type of exception, whether it is a built-in exception or a custom exception, will be caught and handled by the bare `except` block.

Specifying a specific exception type: Instead of using a bare `except` block, it is often preferable to specify the particular exception type(s) you want to catch and handle. By doing this, you have more control over the exception handling process and can tailor your response to different types of exceptions.

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

Ans: Yes, it is possible to have nested `try-except` blocks in Python. Nested `try-except` blocks allow you to handle exceptions at different levels of your code, providing more granular error handling

```
try:
    # Outer try block
    # Code that may raise exceptions

    try:
        # Inner try block
        # Code that may raise exceptions

    except ExceptionType2:
        # Inner except block
        # Exception handling specific to ExceptionType2

    except ExceptionType3:
        # Inner except block
        # Exception handling specific to ExceptionType3

except ExceptionType1:
    # Outer except block
    # Exception handling specific to ExceptionType1
```

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

Ans: Yes, we can use multiple exception blocks in a programming language that supports exception handling. In most programming languages, multiple exception blocks can be used to catch and handle different types of exceptions separately.

```
try:
    # Code that may raise exceptions
    x = int(input("Enter a number: "))
    result = 10 / x
    print("Result:", result)
except ValueError:
    print("Invalid input. Please enter a valid number.")
except ZeroDivisionError:
    print("Cannot divide by zero.")
except Exception as e:
    print("An error occurred:", str(e))
```

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

Ans: a. `EOFError` is raised when the end of a file or input stream is reached unexpectedly. It typically occurs when a program is trying to read data from a file or

receive input from a user, but the expected input is not present or the input operation fails for some reason.

b. `FloatingPointError` is raised when a floating-point arithmetic operation fails. This error typically occurs when there is an exceptional condition in a floating-point calculation, such as division by zero or an invalid operation like taking the square root of a negative number.

c. `IndexError` is raised when an index value is out of range for a sequence or container. It occurs when you try to access an element using an index that is either negative or greater than the length of the sequence or container.

d. `MemoryError` is raised when an operation fails due to insufficient memory. This error occurs when a program tries to allocate more memory than is available in the system, usually when working with large data structures or performing memory-intensive operations.

e. `OverflowError` is raised when a mathematical operation exceeds the maximum representable value for a numeric type. It occurs when the result of an arithmetic operation is too large to be represented within the allowed range of the data type.

f. `TabError` is raised when inconsistent or improper indentation is detected in Python code. This error typically occurs when mixing tabs and spaces for indentation or when the indentation levels are not aligned correctly.

g. `ValueError` is raised when a function receives an argument of the correct type but an inappropriate value. This error occurs when the input value is outside the range of acceptable values or does not meet other criteria defined by the function or operation being performed.

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

Ans: a)

```
def divide_numbers(num1, num2):
    try:
        result = num1 / num2
        return result
    except ZeroDivisionError:
        print("Error: Cannot divide by zero.")
```

b)

```
def convert_to_integer(string_num):
    try:
        integer_num = int(string_num)
        return integer_num
    except ValueError:
        print("Error: Invalid input. Cannot convert to integer.")
```

c)

```
def access_list_element(lst, index):
    try:
        element = lst[index]
        return element
    except IndexError:
        print("Error: Index out of range.")
```

d)

```
def handle_specific_exception(num1, num2):
    try:
        result = num1 / num2
        return result
    except ZeroDivisionError:
        print("Error: Cannot divide by zero.")
```

e)

```
def handle_any_exception(num1, num2):
    try:
        result = num1 / num2
        return result
    except Exception as e:
        print("An error occurred:", str(e))
```