ASSIGNMENT 10

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

The try and exception (or catch) blocks are fundamental components of error handling. They are used to handle exceptions or errors that may occur during the execution of a program. Here's a breakdown of their roles:

Try Block:

- The try block contains the code that you want to monitor for exceptions or errors.
- It encloses the code that might cause an exception, such as division by zero, accessing a non-existent file, or any other operation that could result in a runtime error.

Exception (Catch) Block:

- The exception block, also known as the catch block in some programming languages, is used to handle exceptions that are raised within the try block.
- If an exception occurs in the try block, the control is transferred to the corresponding catch block.
- Inside the catch block, you can define how the program should respond to the exception. This can include logging the error, displaying an error message to the user, or taking corrective action to handle the exceptional situation gracefully.

```
try:
    result = 10 / 0
except ZeroDivisionError:
    print("Error: Division by zero")
```

In this example:

- The code inside the try block attempts to divide 10 by 0, which is not allowed and raises a ZeroDivisionError.
- The program then jumps to the except block where you can handle the error by printing an error message.

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

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

3. What happens if an exception occurs inside a try block and there is no matchingexcept 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 an unhandled exception error will be raised.

Termination of the Program: When an exception is raised in a try block, the control flow within the try block is immediately halted. The program will not continue executing the code within the try block after the point where the exception occurred.

Search for a Matching except Block: The program then searches for an except block

that can handle the raised exception. It does this by checking the type of the exception and comparing it to the types specified in the except blocks within the same try-except structure.

No Matching except Block: If no matching except block is found within the same try-except structure, the program will terminate abruptly. The exception information, including the exception type, traceback, and any associated error message, may be displayed to the user or logged for debugging purposes.

```
try:
    result = 10 / 0                    # This will raise a ZeroDivisionError
except FileNotFoundError as e:
    print("Error: File not found")
```

In this example, the code inside the try block raises a ZeroDivisionError. However, the except block specifies FileNotFoundError as the exception type to handle. Since there's no matching except block for ZeroDivisionError, the program will terminate with an unhandled exception:

To avoid such unhandled exceptions, it's essential to have appropriate except blocks that can handle the specific types of exceptions that may occur within the try block or to include a more general catch-all except block if you want to handle unexpected exceptions gracefully.

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

The difference between using a bare (generic) except block and specifying a specific exception type in a try-except statement lies in how exceptions are handled:

Specific Exception Type: When we specify a specific exception type (e.g., except Value Error:), the except block will only catch exceptions of that type or its subclasses. This approach allows you to handle specific types of exceptions in a targeted manner. It is considered good practice because it provides more control over exception handling and ensures that you handle only the expected exceptions.

```
try:
    value = int("abc")
except ValueError:
    print("Invalid value entered")
```

Bare (Generic) except Block: When you use a bare except block (i.e., except: without specifying an exception type), it acts as a catch-all for any exception that occurs within the try block. This approach can catch any type of exception, which might include unexpected and potentially serious errors. It is generally discouraged in most cases because it makes it harder to diagnose and handle specific issues, and it can hide bugs and problems in your code.

```
try:
    value = int("abc")                 # This will raise a ValueError
```

```
    except:
        print("An error occurred")
```

Using a specific exception type allows you to handle known exceptions in a more controlled and precise manner. Using a bare except block should be avoided in most cases because it can make debugging and maintenance challenging, as it catches all exceptions, including those you may not anticipate. It's generally recommended to use specific exception types whenever possible and have a catch-all except block only when you need to handle truly unexpected or generic errors at a high level, such as logging them for diagnostic purposes.

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 that you can place a try-except block inside another try block to handle exceptions at different levels of your code. This can be useful for more fine-grained error handling. Here's an example:

```
try:
                                                    # Outer try block
    num1 = int(input("Enter a numerator: "))
    num2 = int(input("Enter a denominator: "))

    try:
                                                    # Inner try block
        result = num1 / num2
    except ZeroDivisionError:
        print("Inner Except: Division by zero")

except ValueError:
    print("Outer Except: Invalid input for numerator or denominator")

print("Program continues...")
```

In this example:

1. The outer try block attempts to get user input for a numerator and denominator. It expects potentialValueError exceptions if the user enters non-integer values.

2. The inner try block, nested within the outer try block, calculates the result of dividing num1 by num2. It may raise a ZeroDivisionError if the user enters 0 as the denominator

3. There are two except blocks:

- The outer except block catches ValueError exceptions related to user input for the numerator or denominator.

- The inner except block catches ZeroDivisionError exceptions specifically for division by zero within the inner try block.

Depending on the user's input, the program will handle exceptions at different levels. For example, if the user enters a non-integer value for the numerator or denominator, the outer except block will handle the ValueError. If the user enters 0 as the denominator, the inner

except block will handle the ZeroDivisionError.

Nested try-except blocks allow you to provide different error-handling strategies at various levels of your code, making your error handling more robust and specific to the context in which exceptions occur.

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

Yes, you can use multiple exception blocks (also known as multiple except blocks) in a try-except statement in Python to handle different types of exceptions. This allows you to handle various exceptions in a more fine-grained and specific manner. Here's an example:

```
try:
                                                    # Code that may raise exceptions
    value = int(input("Enter an integer: "))
    result = 10 / value
    file = open("nonexistent.txt", "r")

except ValueError:
                                                # Handle ValueError (e.g., non-integer input)
    print("Invalid input. Please enter an integer.")
except ZeroDivisionError:
                                            # Handle ZeroDivisionError (e.g., division by zero)
    print("Division by zero is not allowed.")
except FileNotFoundError:
                                            # Handle FileNotFoundError (e.g., file not found)
    print("File not found.")
except Exception as e:
                                        # Handle any other exceptions not explicitly caught above
    print(f"An unexpected error occurred: {e}")

print("Program continues...")
```

In this example:

1. The try block contains code that may raise various exceptions:It attempts to convert user input to an integer, which can raise a ValueError if the input is not a valid integer.It performs a division operation, which can raise a ZeroDivisionError if the user enters 0 as the input.It tries to open a file that does not exist, which can raise a FileNotFoundError.

2. There are multiple except blocks, each specifically designed to catch and handle a particular type of exception:The first except block catches ValueError and provides an error message for invalid input.The second except block catches ZeroDivisionError and handles division by zero.The third except block catches FileNotFoundError and handles file not found errors.The fourth except block is a catch-all, which can handle any other exceptions that were not explicitly caught. It also prints the specific error message associated with the caught exception.

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**

Here are explanations for the errors you've mentioned:

a.EOFError (End of File Error):

- An EOFError is raised when an input operation (usually involving reading from a file or standard input) reaches the end of the file or stream unexpectedly.
- This error typically occurs when you attempt to read more data than is available in the input source.

b. FloatingPointError:

- A FloatingPointError is raised when a floating-point operation cannot be executed correctly.
- Common scenarios include division by zero for floating-point numbers or mathematical operations that result in an unrepresentable value due to limitations in the floating-point representation.

c. IndexError:

- An IndexError occurs when you try to access an index of a sequence (e.g., a list, tuple, or string) that is outside the valid range of indices.
- For example, trying to access a non-existent element of a list using an out-of-range index would result in an IndexError.

d. MemoryError:

- A MemoryError is raised when the Python interpreter runs out of memory while trying to allocate memory for an object or data structure.
- This typically happens when you attempt to create very large data structures or when the system has insufficient memory available.

e. OverflowError:

- AnOverflowError occurs when a numerical operation results in a value that is too large to be represented in the available memory or numeric range.
- For instance, trying to calculate an extremely large factorial can lead to an OverflowError.

f. TabError:

- A TabError is raised when there are inconsistent or improper use of tabs and spaces in Python code, particularly in indentation.
- Python relies on consistent indentation to determine the structure of code blocks, and mixing tabs and spaces incorrectly can lead to this error.

g. ValueError

- A ValueError is raised when a function receives an argument of the correct data type but with an invalid or inappropriate value.
- For example, trying to convert a string that is not a valid integer to an integer using int() would raise a ValueError.

8. **Write code for the following given scenario and add try-exception block to it.**

   a. **Program to divide two numbers**

```
try:
    num1=int(input("enter numerator number "))
    num2=int(input("enter denominator number "))
    result=num1/num2
    print(result)

except ValueError :
    print("Enter Proper integer value  ")

except ZeroDivisionError as ex:
    print("your denominator number  is zero error is ",ex)
except Exception as ex:
    print(ex)
```

   b. **Program to convert a string to an integer**

```
try:
    str=int(input("enter string you want to convert  number "))

except ValueError :
    print("Enter Proper numeric  string value  ")
```

   c. **Program to access an element in a list**

```
my_list = [10, 20, 30, 40, 50]

try:
    index = int(input("Enter an index: "))
    element = my_list[index]
    print(f"Element at index {index}: {element}")
except IndexError:
    # Handle an IndexError (e.g., index out of range)
    print("Index out of range.")
except ValueError:
    # Handle a ValueError (e.g., if the user enters a non-integer)
    print("Invalid input. Please enter an integer.")
except Exception as e:
    # Handle any other exceptions that may occur
    print(f"An unexpected error occurred: {e}")
```

   d. **Program to handle a specific exception**

```
try:
    num1=int(input("enter numerator number "))
    num2=int(input("enter denominator number "))
    result=num1/num2
    print(result)

except ValueError :
    print("Enter Proper integer value  ")

except ZeroDivisionError as ex:
    print("your denominator number  is zero error is ",ex)
```

   e. **Program to handle any exception**

```
try:
    num1=int(input("enter numerator number "))
    num2=int(input("enter denominator number "))
    result=num1/num2
    print(result)

except Exception as ex:
    print(ex)
```