**Answer 1:-**

**Try-Catch Block (Exception Handling)**

In many programming languages like Python, Java, and C++, a try block is used to handle exceptions (errors) that might occur in the code. The associated catch (or except in Python) block is used to manage those errors.

**Example in Python**

```
try:
    num = int(input("Enter a number: "))
    result = 10 / num
    print(result)
except ValueError:
    print("Invalid input! Please enter a number.")
except ZeroDivisionError:
    print("Cannot divide by zero!")
finally:
    print("Execution complete.")
```

- try: Wraps the code that might cause an error.
- except: Catches and handles specific exceptions.
- finally: Runs code regardless of whether an error occurs.

**Answer 2:-**

**The syntax of a try-except block :**

```
try:
    # Code that may cause an exception
    risky_operation()
except ExceptionType:
    # Handle the exception
    print("An error occurred")
```

**Example:-**

```
try:
    x = 10 / 0  # This will raise a ZeroDivisionError
except ZeroDivisionError:
    print("Cannot divide by zero!")
```

**Answer 3:-**

If an exception occurs inside a try block and there is no matching except block, the program will terminate with an error message (also called an unhandled exception).
If an exception occurs and there's no matching except block, Python will display a traceback and terminate the program.

**Example**

```
try:
    x = 10 / 0  # Raises ZeroDivisionError
except ValueError:  # This does not match ZeroDivisionError
    print("Invalid value!")

print("This will not execute if an exception is not handled.")
```

**Answer 4:-**

A bare except block and specifying a specific exception type have different behaviors when handling exceptions in Python.

**1. Bare except Block**

```
try:
    x = 10 / 0
except:
    print("An error occurred!")
```

**2. Specifying a Specific Exception Type**

```
try:
    x = 10 / 0
except ZeroDivisionError:
    print("Cannot divide by zero!")
```

**3. Using except Exception: (Better Alternative to Bare except Block)**

```
try:
    x = 10 / 0
except Exception as e:
    print(f"An error occurred: {e}")
```

**Answer 5:-**

Yes, you can have nested try-except blocks in Python. This means you can have a try-except block inside another try block. This is useful when you need to handle exceptions at different levels of execution.

**Example:**

```
def nested_try_example():
    try:
        print("Outer try block")
        x = int(input("Enter a number: "))  # Might raise ValueError

        try:
            print("Inner try block")
            result = 10 / x  # Might raise ZeroDivisionError
            print("Result:", result)
        except ZeroDivisionError:
            print("Inner except: Cannot divide by zero.")

    except ValueError:
        print("Outer except: Invalid input. Please enter a number.")

finally:
        print("Finally block executed.")

nested_try_example()
```

**<u>Answer 6</u>:-**

Yes, you can use multiple except blocks in Python to handle different types of exceptions separately. This allows you to customize the error handling for specific exceptions.

**Example:**

```
try:
    x = int(input("Enter a number: "))  # Might raise ValueError
    result = 10 / x  # Might raise ZeroDivisionError
    my_list = [1, 2, 3]
    print(my_list[5])  # Might raise IndexError

except ValueError:
    print("Invalid input! Please enter a valid number.")

except ZeroDivisionError:
```

```
    print("Cannot divide by zero!")

except IndexError:
    print("Index out of range!")

except Exception as e:
    print(f"An unexpected error occurred: {e}")

finally:
    print("Execution completed.")
```

**Answer 7:-**

**A). EOFError**

An EOFError (End-of-File Error) is raised when an input operation reaches the end of a file or input stream unexpectedly. Here are the common reasons for this error:

- Unexpected End of Input – When using input() in Python and the input stream is closed or no input is provided (e.g., pressing Ctrl+D in Linux/macOS or Ctrl+Z in Windows).
- Reading from an Empty File – When trying to read from a file that has no more data using functions like readline() or read().
- Pickle Module Issues – When pickle.load() attempts to read from an empty or corrupted file.

**B). Flo**

A FloatingPointError is raised when a floating-point operation fails under exceptional conditions. However, in Python, this error is rarely encountered because most floating-point operations return inf (infinity) or nan (not a number) instead of raising an error.

**C). IndexError**

An IndexError is raised when trying to access an index that is out of the valid range for a sequence (like a list, tuple, or string).

**D). Memory error**

A MemoryError is raised when Python runs out of memory while trying to allocate an object that is too large for the available system memory.

E). **OverflowError**

An OverflowError is raised when a numeric calculation exceeds the limit of what Python can represent within its floating-point or integer system. However, Python's integers have arbitrary precision, so this error typically occurs with floating-point operations.

**F). TabError**

A TabError is raised in Python when inconsistent use of tabs and spaces is detected for indentation. Python enforces indentation rules strictly, and mixing tabs and spaces in the same block of code leads to this error.

**G). Value error**

A ValueError is raised in Python when a function receives an argument of the correct type but an inappropriate value.

**Answer 8:-**

**A). Program to divide two numbers**

```
def divide_numbers():
    try:
        num1 = float(input("Enter the first number: "))
        num2 = float(input("Enter the second number: "))

        result = num1 / num2
        print(f"Result: {result}")

    except ZeroDivisionError:
        print("Error: Cannot divide by zero.")
    except ValueError:
        print("Error: Please enter valid numbers.")

divide_numbers()
```

**B). program to convert a string to an integer**

```
def string_to_integer():
    try:
        user_input = input("Enter a number: ")
        num = int(user_input)
        print(f"Converted integer: {num}")
    except ValueError:
        print("Error: Please enter a valid    integer.")
```

```
string_to_integer()
```

**C). Program to access an element in the list**

```
def access_element():
    try:
        my_list = [10, 20, 30, 40, 50]
        index = int(input("Enter the index to access (0-4): "))

        element = my_list[index]
        print(f"Element at index {index}: {element}")

    except IndexError:
        print("Error: Index out of range. Please enter a valid index (0-4).")
    except ValueError:
        print("Error: Please enter a valid integer.")

access_element()
```

**D). Program to handle a specific exception**

```
def divide_numbers():
    try:
        num1 = float(input("Enter the first number: "))
        num2 = float(input("Enter the second number: "))

        result = num1 / num2
        print(f"Result: {result}")

    except ZeroDivisionError:
        print("Error: Cannot divide by zero. Please enter a non-zero denominator.")

divide_numbers()
```

**E). Program to handle any exception**

```
def safe_division():
    try:
        num1 = float(input("Enter the first number: "))
        num2 = float(input("Enter the second number: "))

        result = num1 / num2
        print(f"Result: {result}")
```

```
    except Exception as e:  # Catches any exception
        print(f"An error occurred: {e}")

safe_division()
```