17th June Assignment

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

Ans- The try and except block is a way to handle errors that may occur in your code. The try block lets you test a block of code for errors. The except block lets you handle the error. The else block lets you execute code when there is no error. The finally block lets you execute code, regardless of the result of the try- and except blocks.

When an error occurs, or exception as we call it, Python will normally stop and generate an error message. These exceptions can be handled using the try statement.

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

Ans-

```
#  Divide two numbers and print the quotient

num1 = input("Enter the first number: ")

num2 = input("Enter the second number: ")

try:

  # convert the inputs to integers and divide them

  quotient = int(num1) // int(num2)

  print("The quotient is:", quotient)

except:

  # handle the error if the inputs are not valid or the division is not possible

  print("Invalid input or division by zero")
```

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, the exception will propagate to the outer scope and terminate the program, unless it is handled by another try-except block in the call stack. This is called unhandled exception.

Example:

```
def divide(x, y):

  try:

    return x / y
```

```
    except ZeroDivisionError:
        print("Cannot divide by zero")
def main():
    try:
        a = int(input("Enter a number: "))
        b = int(input("Enter another number: "))
        result = divide(a, b)
        print(f"The result is {result}")
    except ValueError:
        print("Invalid input")
main()
```

The divide function has a try-except block to handle the ZeroDivisionError exception. it does not handle any other type of exception. The main function has a try-except block to handle the ValueError exception that may occur when converting the user input to integers. However, it does not handle any other type of exception either.

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

Ans- The difference between using a bare except block and specifying a specific exception type is that the former will catch all exceptions indiscriminately, while the latter will catch only the specified exception type or its subclasses. This has several implications for the robustness and readability of the code.

Some of the disadvantages of using a bare except block are:

- It can hide bugs or unexpected errors that are not related to the intended purpose of the try-except block. For example, if you use a bare except block to handle a file not found error, you might also catch a syntax error or a name error that you did not anticipate.
- It can prevent the program from exiting gracefully when the user wants to interrupt it or when the system needs to shut it down. For example, if you use a bare except block in a loop, you might catch a KeyboardInterrupt exception that is raised when the user presses Ctrl+C, or a System Exit exception that is raised when the system calls sys.exit().

It is recommended to always specify an exception type in except blocks, unless you have a very good reason to catch all exceptions. This way, you can ensure that you only handle the exceptions that you know how to handle, and let the rest propagate to the outer scope or terminate the program with an informative error message. You can also use multiple

except blocks to handle different types of exceptions differently, or use a generic except Exception block to catch all regular exceptions but not system-related ones.

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

Ans- Yes, you can have nested try-except blocks in Python. This means that you can place a try-except block inside another try-except block, or inside a try or except clause. This can be useful when you want to handle different types of exceptions at different levels of granularity, or when you want to perform some additional actions before propagating the exception to the outer scope.

Example:

```
string_to_number("3.14")
3.14
>>> string_to_number("abc")
Could not write to log file
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in string_to_number

ValueError: abc is not a valid number
```

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

Ans- Yes, we can use multiple exception blocks in Python. This means that we can handle different types of exceptions differently, depending on the logic and requirements of our program. There are two ways to use multiple exception blocks in Python.

```
try:
    even_numbers = [2,4,6,8]
    print(even_numbers[5] / 0) # this will cause an error
except ZeroDivisionError:
    print("Denominator cannot be 0.")
except IndexError:
    print("Index Out of Bound.")
```

Output:

```
Index Out of Bound.
```

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: Raised when the input() function does not get any input from the user or the file. This usually happens when the user stops giving input or the file ends.

b. FloatingPointError: Raised when a decimal number calculation fails. This exception is only raised when Python is configured with a special option, or when the math.faulthandler() function is called.

c. IndexError: Raised when you try to access an element of a list, string, or other sequence that does not exist. For example, if you try to access list[10] when the list has only 5 elements.

d. MemoryError: Raised when a program runs out of memory and cannot continue. The associated value is a string indicating what kind of operation ran out of memory.

e. OverflowError: Raised when the result of a number calculation is too big to be stored. This cannot happen for whole numbers but for decimal numbers and some other number types.

f. TabError: Raised when the indentation of a program is not consistent and uses both tabs and spaces. This is a subclass of IndentationError.

g. ValueError: Raised when an operation or function gets an argument that has the right type but a wrong value, and the situation is not described by a more specific exception such as IndexError.

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. Program to divide two numbers

```
def divide (a, b):
    try:
```

```
    # Try to perform the division and return the result
      return a / b
    except ZeroDivisionError:
# Handle the case when b is zero
      print("Cannot divide by zero")
# Test the function with some examples
    print(divide(10, 2))
    print(divide(5, 0))
```

Output:

5.0

Cannot divide by zero

 # b. Program to convert a string to an integer

```
def convert(s):
    try:
    # Try to convert the string to an integer and return it
      return int(s)
    except ValueError:
    # Handle the case when the string is not a valid integer
      print("Invalid integer")

# Test the function with some examples
print(convert("42"))
print(convert("abc"))
```

Output:

```
42
Invalid integer
```

 # c. Program to access an element in a list

```
def access(lst, index):
    try:
    # Try to access the element at the given index and return it
      return lst[index]
```

```
    except IndexError:
    # Handle the case when the index is out of range
      print("Index out of range")

# Test the function with some examples
lst = [1, 2, 3, 4, 5]
print(access(lst, 2))
print(access(lst, 10))
```

Output:

```
3
Index out of range
```

 # d. Program to handle a specific exception

```
def handle_specific():
    try:
    # Try to do something that might raise an exception
        x = int(input("Enter a number: "))
        y = 10 / x
        print(y)
    except ZeroDivisionError:
    # Handle the specific exception of dividing by zero
      print("You entered zero")
    except Exception as e:
    # Handle any other exception and print its type and message
      print("Something went wrong")
    print(type(e))
    print(e)

# Test the function with some examples
handle_specific()
handle_specific()
handle_specific()
```

Output:

Enter a number: 2, Prints 5.0

Enter a number: 0, Prints You entered zero

Enter a number: abc, Prints Something went wrong, Prints <class 'ValueError'>, Prints invalid literal for int() with base 10: 'abc'

 # e. Program to handle any exception

```
def handle_any():
    try:
    # Try to do something that might raise an exception
        x = int(input("Enter a number: "))
        y = 10 / x
        print(y)
    except Exception as e:
    # Handle any exception and print its type and message
      print("Something went wrong")
    print(type(e))
    print(e)


# Test the function with Examples
handle_any()
handle_any()
handle_any()
```

Output:

Enter a number: 2, Prints 5.0

Enter a number: 0, Prints Something went wrong, Prints <class 'ZeroDivisionError'>, Prints division by zero

Enter a number: abc, Prints Something went wrong, Prints <class 'ValueError'>, Prints invalid literal for int() with base 10: 'abc'