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

Ans:-
    The "try" and "except" blocks are fundamental constructs in programming languages that provide a way to handle errors and exceptions gracefully. These blocks are particularly important when writing code that might encounter unexpected situations, like runtime errors or exceptional conditions, which could otherwise lead to program crashes or undesired behavior.

(1) Try Block: The code that you suspect might raise an exception is placed within the "try" block. This block contains the code that you want to monitor for exceptions.

(2) Except Block: If an exception occurs within the "try" block, the program flow immediately jumps to the corresponding "except" block. The "except" block contains the code that specifies how to handle the exception. You can define different "except" blocks for different types of exceptions to provide customized handling.

For Example:-

In [1]:

```
try:
    # Code that might raise an exception
    num = int(input("Enter a number: "))
    result = 10 / num
except ZeroDivisionError:
    print("Cannot divide by zero!")
except ValueError:
    print("Invalid input! Please enter a valid number.")
except Exception as e:
    print("An error occurred:", e)
else:
    print("Result:", result)
finally:
    print("This code always executes, regardless of exceptions.")
```

```
Enter a number: 5
Result: 2.0
This code always executes, regardless of exceptions.
```

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

Ans:-
    (1) Try: This is where you enclose the code that might raise an exception. If an exception occurs within this block, the control will immediately jump to the corresponding except block.

   (2) SomeExceptionType: Replace this with the specific type of exception you want to catch. For example, in Python, you could use ValueError, TypeError, IOError, etc. to catch specific types of exceptions. You can also use a more general Exception to catch any type of exception.

(3) Except: This is where you handle the exception. The code within this block will run if the specified exception type is raised within the try block.

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 encounters an appropriate except block or until it reaches the top level of the program. If the exception is not caught anywhere in the call stack, it will result in the termination of the program and an error message will be displayed, providing information about the type of exception and the stack trace leading up to the point where the exception occurred.
For Example:-

In [2]:

```
try:
    x = 10 / 0  # This will raise a ZeroDivisionError
except ValueError:
    print("Caught a ValueError")
```

```
---------------------------------------------------------------------------
-
ZeroDivisionError                         Traceback (most recent call las
t)
Cell In[2], line 2
      1 try:
----> 2     x = 10 / 0  # This will raise a ZeroDivisionError
      3 except ValueError:
      4     print("Caught a ValueError")

ZeroDivisionError: division by zero
```

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

Ans:- (1) Bare except block: A bare except block, also known as a generic or catch-all exception handler, catches and handles all types of exceptions that may occur within its scope. It doesn't differentiate between the types of exceptions raised, which can make it challenging to diagnose the actual problem. This approach is generally not recommended because it can lead to unintended consequences and make debugging difficult.

(2)Specifying a specific exception type: Specifying a specific exception type involves catching only a particular type of exception. This allows you to handle that specific type of exception in a more targeted and controlled manner. It's considered best practice to catch only the exceptions you expect to occur and handle them appropriately.

Key differences:

(a) Precision: Using a specific exception type allows you to handle only the intended exceptions, leaving other exceptions uncaught and potentially letting them propagate up the call stack. This approach is more precise and helps in identifying the root cause of the issue.

(b) Readability: Code with specific exception handling is more readable and self-explanatory. It clearly indicates the types of exceptions that are expected to be encountered and handled.

(c) Debugging: Using a bare except block can make debugging difficult since it obscures the actual cause of the exception. Specific exception handling provides better insight into what went wrong.

(d) Maintainability: Code that uses specific exception handling tends to be more maintainable over time. If new exceptions need to be handled, they can be added to the appropriate except blocks without affecting the handling of other exceptions.

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

Ans:-
    Yes, we can have nested try-except blocks in Python. This means you can place one try-except block inside another. This can be useful when you want to handle different levels of exceptions in a more granular way.
    For Example:-

In [3]:

```
try:
    outer_variable = 10
    inner_variable = 0

    try:
        result = outer_variable / inner_variable
    except ZeroDivisionError:
        print("Error: Division by zero in inner block")
except Exception as e:
    print(f"Outer block caught an exception: {e}")
```

```
Error: Division by zero in inner block
```

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

Ans:-
    Yes, we can use multiple exception blocks in programming languages that support exception handling, such as Python, Java, and many others. Using multiple exception blocks allows us to handle different types of exceptions separately, providing more fine-grained error handling in our code.

In [5]:

```
try:
    num = int(input("Enter a number: "))
    result = 10 / num
    print("Result:", result)
except ZeroDivisionError:
    print("Error: Cannot divide by zero.")
except ValueError:
    print("Invalid input. Please enter a valid number.")
except Exception as e:
    print("An error occurred:", e)
finally:
    print("Exception handling is done.")
```

```
Enter a number: 10
Result: 1.0
Exception handling is done.
```

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:
This error is raised when an operation that requires more data than what is available
is attempted to be performed on an input stream. It stands for "End of File Error" and
commonly occurs when reading from a file or a data stream and reaching the end
unexpectedly.

   (b) FloatingPointError:
This error is raised when a floating-point operation fails to execute properly. It can
occur when trying to perform illegal floating-point calculations, such as dividing by
zero or trying to calculate the square root of a negative number.

   (c) IndexError:
This error is raised when attempting to access an index of a sequence (like a list or
a string) that is out of the allowed range. For example, trying to access an element
at an index that is greater than or equal to the length of the sequence will result in
an IndexError.
```

```
   (d) MemoryError:
This error is raised when an operation that requires more memory than what is
available in the system is attempted. It indicates that the program has exhausted its
available memory resources and cannot allocate any more memory for the operation.

   (e) OverflowError:
This error is raised when a numeric calculation exceeds the limits of its data type's
representable range. It occurs when attempting to store a value that is too large to
be accommodated by the data type, such as an integer that goes beyond the maximum
representable value.

   (f) TabError:
This error is raised when there is an issue with the usage of tabs and spaces for
indentation in Python code. Python uses indentation to define code blocks, and mixing
tabs and spaces inconsistently can lead to a TabError. This error is often encountered
when there's a mismatch between the indentation style used in a file.

   (g) ValueError:
This error is raised when a function receives an argument of the correct data type but
an inappropriate value. For example, trying to convert a string to an integer using
int() when the string is not a valid integer representation will raise a ValueError
```

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

In [6]:

```
# Program to divide two numbers
def divide_numbers(a, b):
    try:
        result = a / b
        return result
    except ZeroDivisionError:
        return "Error: Division by zero is not allowed."

numerator = 10
denominator = 2
print(divide_numbers(numerator, denominator))
```

```
5.0
```

In [7]:

```
# Program to convert a string to an integer
def convert_to_integer(input_str):
    try:
        result = int(input_str)
        return result
    except ValueError:
        return "Error: Unable to convert the input to an integer."

user_input = input("Enter a number: ")
print(convert_to_integer(user_input))
```

```
Enter a number: 59
59
```

In [12]:

```
# Program to access an element in a list
def access_element(input_list, index):
    try:
        result = input_list[index]
        return result
    except IndexError:
        return "Error: Index out of range."

my_list = [1, 2, 3, 4, 5]
desired_index = 9
print(access_element(my_list, desired_index))
```

```
Error: Index out of range.
```

In [13]:

```
# Program to handle a specific exception
def specific_exception_handling(num):
    try:
        result = 10 / num
        return result
    except ZeroDivisionError:
        return "Error: Division by zero is not allowed."

divider = 0
print(specific_exception_handling(divider))
```

```
Error: Division by zero is not allowed.
```

In [14]:

```
# Program to handle any exception
def handle_any_exception(input_value):
    try:
        result = 10 / input_value
        return result
    except Exception as e:
        return f"An error occurred: {str(e)}"

divisor = 0
print(handle_any_exception(divisor))
```

```
An error occurred: division by zero
```

In [ ]:

```

```