Exception Handling

__Topics to be covered__

1. Introduction
2. Types of errors
3. Handling Exceptions Using Try – Except – Finally Blocks

EXCEPTION HANDLING

While executing a Python program, it may happen that the program does not execute at all or it can generate unexpected output. This happens when there are syntax errors, run time errors, logical errors or any semantic errors in the code.

SYNTAX ERROR	SEMANTIC ERROR	LOGICAL ERROR	RUN TIME ERROR

It occurs when we put some incorrect punctuation, incorrect word sequence or there are some undefined terms or missing parenthesis.
Syntax errors are also called as **Parsing errors.**
For example:
>>> p=2(num1+num2)
This statement is mathematically correct but python interpreter will raise SYNTAX error as there is no sign present between 2 and parenthesis. The correct statement will be:
>>> p=2*(num1+num2)
	It occurs when the code is correct according to syntax but it is not meaningful. The code will not behave as expected.
	For example:
	A = 10
	B = ”hello”
	Result = A + B
	Here, The code will execute as it has correct syntax but will not generate expected output.
		It may occur when there may be some improper sequence of statements or incorrect use of operator. It will not stop a program from executing but will produce incorrect output. It will generate incorrect output for every value of input.
		For example:
		If we want to find sum of two numbers and write the following code:
		A, B = 10, 15
		C = A * B
		print (“Sum is: “, C)
		Here, the code will generate A * B but we wanted to find Sum. Hence it is a logical error.
			It occurs at the time of program execution. Such errors produce incorrect output for specific values of input. These errors are also called **Exceptions,** which occur when something unexpected happens leading to stop the program execution.
			For example:
			5. Division by zero
			6. Finding square root of negative number.
			7. Insufficient memory available on computer.
			8. Trying to open a file that does not exist.

EXCEPTIONS:

11. Run time errors are known as Exceptions.
12. When an Exception is generated, the program execution is stopped.
13. Removing errors from a code is referred to as EXCEPTION HANDLING.
14. Commonly occurring exceptions are usually defined in the Interpreter. These are known as Built-in Exceptions.

Some Built-in Exceptions are listed as below:

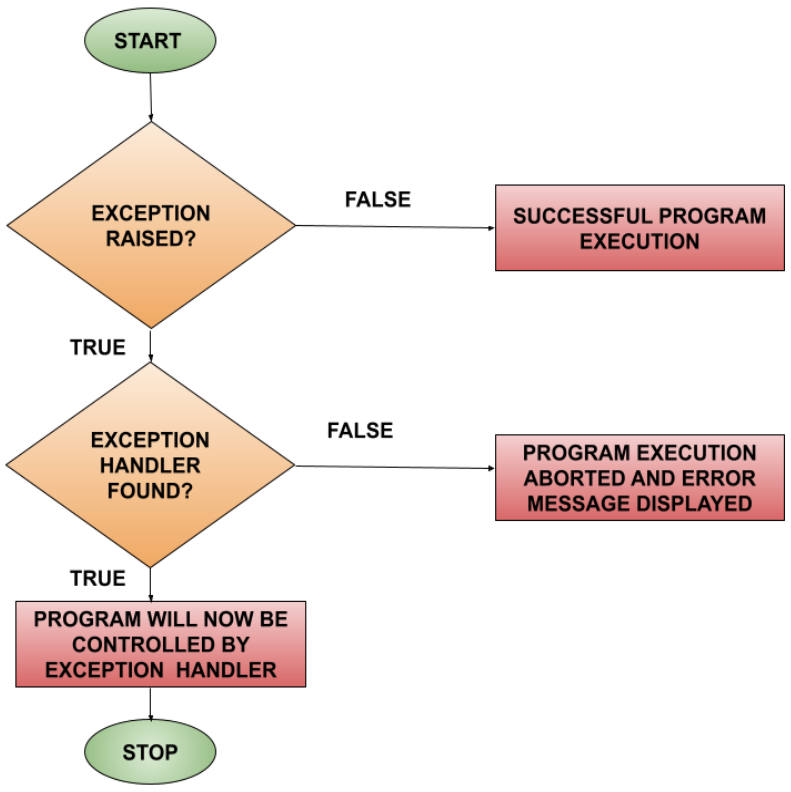
EXCEPTION	DESCRIPTION
ZeroDivisionError	It is raised when an expression or a value is getting divided by zero (0). For example: If c = 0, then p = b/c will result in 'ZeroDivisionError'.
NameError	It is raised when an identifier is not assigned any value earlier and is being used in some expression. For example: p = a*b/c then it will result in 'NameError' when one or more variables are not assigned values.
TypeError	It is raised when variables used in any expression have values of different data types. For example: p=(a+b)/c then it will result in 'TypeError' when the variables a, b and c are of different data types.
Value Error	It is raised when given value of a variable is of right data type but not appropriate according to the expression.
IOError	It is raised when the file specified in a program statement cannot be opened.
IndexError	It is raised when index of a sequence is out of the range.
KeyError	It is raised when a key doesn't exist or not found in a dictionary.

EXCEPTION HANDLING:

Every exception has to be handled by the programmer for successful execution of the program. To ensure this we write some additional code to give some proper message to the user if such a condition occurs. This process is known as **EXCEPTION HANDLING.**

Exception handlers separate the main logic of the program from the error detection and correction code. The segment of code where there is any possibility of error or exception, is placed inside one block. The code to be executed in case the exception has occurred, is placed inside another block. These statements for detection and reporting the execution do not affect the main logic of the program.

STEPS FOR EXCEPTION HANDLING:


START
EXCEPTION RAISED?
FALSE
SUCCESSFUL PROGRAM EXECUTION
TRUE
EXCEPTION HANDLER FOUND?
FALSE
PROGRAM EXECUTION ABORTED AND ERROR MESSAGE DISPLAYED
TRUE
PROGRAM WILL NOW BE CONTROLLED BY EXCEPTION HANDLER
STOP

An exception is said to be caught when a code designed for handling that particular exception is executed. In Python, exceptions, if any, are handled by using try-except-finally block. While writing a code, programmer might doubt a particular part of code to raise an exception. Such suspicious lines of code are considered inside a **try** block which will always be followed by an **except** block. The code to handle every possible exception, that may raise in try block, will be written inside the **except** block.

If no exception occurred during execution of program, the program produces desired output successfully. But if an exception is encountered, further execution of the code inside the try block will be stopped and the control flow will be transferred to the except block.

Example 1:

```
n1 = int(input("Enter first number: "))
n2 = int(input("Enter second number: "))
try:
    result = n1 / n2
    print("Division result is: ", result)

except ZeroDivisionError:
    print("Denominator is Zero!! Division not possible.")
```

The output will be:

```
============ RESTART: C:/Users/my lapi/Desktop/exception example.py =====
Enter first number: 12
Enter second number: 2
Division result is:  6.0

============ RESTART: C:/Users/my lapi/Desktop/exception example.py =====
Enter first number: 12
Enter second number: 0
Denominator is Zero!! Division not possible.
```

Example 2:

```
a=int(input("enter a value")) # here user should enter an integer
enter a valueq
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    a=int(input("enter a value")) # here user should enter an integer
ValueError: invalid literal for int() with base 10: 'q'
```

In above example, the user entered a wrong value that raised ValueError. We can handle this exception by using ValueError exception.

```
try:
    a=int(input("enter a value")) # here user should enter an integer
    '''
the user may enter any value which cant be typecasted to integer.
in that case a value error exception will be raised. We can handle that exception
   '''
    print(a)
except ValueError:
    print("enter only integers")
```

Result:

```
enter a valueq
enter only integers
```

Use of multiple "except" block:

Sometimes, it may happen that a single piece of code in a program might have more than one type of error. If such an event happens, we can use multiple **except** blocks for a single **try** block.

Example 1:

```
try:
    n1 = int(input("Enter first number: "))
    n2 = int(input("Enter second number: "))
    result = n1 / n2
    print("Division result is: ", result)

except ZeroDivisionError:
    print("Denominator is Zero!! Division not possible.")

except ValueError:
    print("Kindly enter only Integer Values.")
```

The output will be:

```
Enter first number: 25
Enter second number: a
Kindly enter only Integer Values.
```

Example 2:

```
try:
    file = open("nonexistent_file.txt", "r")
    data = file.read()
    file.close()
    num = int(data)
    result = 10 / num
    print("Result:", result)
except FileNotFoundError:
    print("Error: File not found.")
except ZeroDivisionError:
    print("Error: Cannot divide by zero.")
except ValueError:
    print("Error: Invalid data. Please make sure the file contains a valid number.")
```

We can also handle exceptions without naming them.

```
#handling exceptions without naming them.
try:
    n1 = int(input("Enter first number: "))
    n2 = int(input("Enter second number: "))
    result = n1 / n2
    print("Division result is: ", result)

except ValueError:
    print("Kindly enter only Integer Values.")

except:
    print("oops!! There are some Exceptions.")
```

The output will be:

```
Enter first number: 56
Enter second number: 0
oops!! There are some Exceptions.
```

Default exception messages can also be displayed when we are not handling exceptions by name.

```
try:
    n1 = int(input("Enter first number: "))
    n2 = int(input("Enter second number: "))
    result = n1 / n2
    print("Division operation performed")
except ValueError:
    print("Kindly enter only Integer Values.")
except Exception as e:
    print("There is an Exception, ", str(e))
else:
    print("Division result is: ", result)
finally:
    print("Have a good day!!")
```

The output will be:

```
Enter first number: 12
Enter second number: 0
There is an Exception,  division by zero
Have a good day!!
```

try…except…else Clause:

Just like Conditional and Iterative statements we can use an optional **else** clause along with the **try…except** clause. An except block will be executed only when some exceptions will be raised in the try block. But if there is no error then except blocks will not be executed. **In this case, else clause will be executed.**

```
try:
    n1 = int(input("Enter first number: "))
    n2 = int(input("Enter second number: "))
    result = n1 / n2
    print("Division operation performed")
except ValueError:
    print("Kindly enter only Integer Values.")
except:
    print("oops!! There are some Exceptions.")
else:
    print("Division result is: ", result)
```

The output will be:

```
Enter first number: 12
Enter second number: 2
Division operation performed
Division result is:  6.0
```

finally CLAUSE:

The **try…except…else** block in python has an optional **finally** clause. The statements inside the **finally** block are always executed whether an exception has occurred in the try block or not. If we want to use **finally** block, it should always be placed at the end of the clause i.e. after all except blocks and the else block.

```
try:
    n1 = int(input("Enter first number: "))
    n2 = int(input("Enter second number: "))
    result = n1 / n2
    print("Division operation performed")
except ValueError:
    print("Kindly enter only Integer Values.")
except:
    print("oops!! There are some Exceptions.")
else:
    print("Division result is: ", result)
finally:
    print("Have a good day!!")
```

The output will be:

```
Enter first number: 12
Enter Second number: 2
Division operation performed.
Have a good day.
```

Exercise

1 In a try-except block, can there be multiple 'except' clauses?
a) No, there can be only one 'except' clause.
b) Yes, but only if the exceptions are of the same type.
c) Yes, it allows handling different exceptions separately.
d) No, 'except' clauses are not allowed in a try-except block.

2 When might you use the 'finally' block in exception handling?
a) To handle exceptions that are expected to occur frequently.
b) To provide a resolution for every possible error.
c) To close resources that were opened in the 'try' block, regardless of whether an exception occurred or not.
d) To avoid having to use 'except' blocks.

3 What will be the output of the following code snippet?

```
try:
    result = 10 / 0
except ZeroDivisionError:
    print("Division by zero!")
except ArithmeticError:
    print("Arithmetic error occurred!")
else:
    print("No error!")
```

a) Division by zero!
b) Arithmetic error occurred!
c) No error!
d) This code will raise a syntax error.

4 Which of the following is NOT a standard built-in exception in Python?
a) ValueError
b) IndexError
c) NullPointerException
d) KeyError

5 What is an exception in programming?
a) An error that occurs during runtime
b) A warning message from the compiler
c) A comment in the code
d) A statement that terminates the program

6 What is the purpose of the "try" block in a try-except construct?
a) To handle the exception by executing specific code
b) To specify the type of exception to be thrown
c) To define a custom exception class
d) To ensure a specific block of code always executes

7 Which of the following exceptions in Python is not a built-in exception?
a) ValueError

b) KeyError
c) CustomError
d) IndexError

8 Assertion (A): In Python, the "try" block is used to enclose code that might raise an exception.
Reasoning (R): The "try" block is where the program attempts to execute code that might result in an exception. If an exception occurs, it is handled in the corresponding "except" block.
A. Both A and R are true and R is correct explanation of A
B. Both A and R are true but R is not correct explanation of A
C. A is True but R is False
D. R is True but A is False

9 Assertion (A): The "finally" block in Python is always executed, regardless of whether an exception is raised or not.
Reasoning (R): The "finally" block contains code that is guaranteed to execute, whether an exception occurs within the "try" block or not.
A. Both A and R are true and R is correct explanation of A
B. Both A and R are true but R is not correct explanation of A
C. A is True but R is False
D. R is True but A is False

10 Assertion (A): Python allows multiple "except" blocks to be used within a single "try" block to handle different exceptions.
Reasoning (R): By using multiple "except" blocks with different exception types, Python provides the flexibility to handle various types of exceptions separately.
A. Both A and R are true and R is correct explanation of A
B. Both A and R are true but R is not correct explanation of A
C. A is True but R is False
D. R is True but A is False

11 Code snippet:

```
try:
    num = int(input("Enter a number: "))
    result = 10 / num
    print("Result:", result)
except ZeroDivisionError:
    print("Error: Cannot divide by zero.")
except ValueError:
    print("Error: Invalid input. Please enter a valid number.")
```

Predict the output when:
a) The user enters "0" .
b) The user enters "5".
c) The user enters "abc".

12 State whether the following statement is True or False: An exception may be raised even if the program is syntactically correct.

13 What will be the output of the following code if the input is “e”:

```
try:
    value = int("abc")
    result = 10 / 0
except ValueError:
    print("Error: Invalid value conversion")
except ZeroDivisionError:
    print("Error: Division by zero")
```

14 What will be the output of the following code if the input is:
i. 2 ii. 2.2

```
try:
    num = int(input("Enter a number: "))
except ValueError:
    print("Error: Invalid input")
else:
    print("Entered number: ",num)
```

15 Rewrite the following code after handling all possible exceptions.

```
    num = int(input("Enter a number: "))
    result = 10 / num
    print("Result:", result)
```

16 Consider the code given below:

```
L = ['s', 45,23]
Result = 0
for x in L:
     print ("The element is ", x)
     Result += x
print("The addition of all elements of L is: ", Result)
```

Which of the following error will be raised by the given Python code?

a) NameError
b) ValueError
c) TypeError
d) IOError

17 Code snippet:

```
try:
    num1 = int(input("Enter the first number: "))
    num2 = int(input("Enter the second number: "))
    result = num1 / num2
except ZeroDivisionError:
    print("Error: Cannot divide by zero.")
except ValueError:
    print("Error: Invalid input. Please enter a valid number.")
else:
    print("Result:", result)
finally:
    print("Finally block executed.")
```

Predict the output when:
a) The user enters "10" for both numbers.

b) The user enters "5" for the first number and "0" for the second number.
c) The user enters "abc" for both numbers.

18 Which of the following statements is true?
a). The standard exceptions are automatically imported in Python programs.
b). All raised standard exceptions must be handled in Python.
c). When there is deviation from the rules of a programming language, a semantic error is thrown.
d). If any exception is thrown in try block, else block is executed.

19 Identify the statement(s) from the following options which will raise TypeError exception(s):
a) print('5')
b) print(5 * 3)
c) print('5' +3)
d) print('5' + '3')

Programming based question:

1 Create a simple calculator program that takes two numbers and an operator (+, -, *, /) as input. Implement exception handling to handle cases like division by zero and invalid operators.

2 Build a program that asks the user for an integer input. Use exception handling to ensure that the input is a valid integer. If the user provides invalid input, prompt them to retry until a valid integer is entered.

3 Create a program that connects to a database and performs database operations (e.g., insert, update, delete). Use exception handling to deal with database-related exceptions, such as connection errors or SQL syntax error.

4 Create a program that reads data from a file specified by the user. Implement exception handling to catch and handle the "FileNotFoundError" exception if the file does not exist.

5 Define a dictionary with some key-value pairs. Ask the user for a key and attempt to access the corresponding value from the dictionary. Handle the "KeyError" exception and display a message if the key is not found.