Week -13

Error and Exception Handling: Python errors; exceptions: built in, user defined. How to catch exceptions? Raising exceptions;

Errors:

- An error is an issue in a program that prevents the program from completing its task
- An exception is a condition that interrupts the normal flow of the program

Three types of Error occur in python.

1. Syntax errors :
    - Syntax errors, also known as parsing errors and when the proper syntax of the language is not followed then a syntax error will occur.
2. Runtime Errors
    - Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it.
    - This type of error occurs whenever syntactically correct Python code results in an error
3. Logical errors: Logical errors occur when the code is syntactically correct and run without causing errors, but the actual output is not same as the expected output

Types of Exceptions in Python:

1) Built-in
2) User Defined

Built-in Python Exceptions

Here is the list of default Python exceptions with descriptions:

1. AssertionError: raised when the assert statement fails.
2. EOFError: raised when the input() function meets the end-of-file condition.
3. AttributeError: raised when the attribute assignment or reference fails.
4. TabError: raised when the indentations consist of inconsistent tabs or spaces.
5. ImportError: raised when importing the module fails.
6. IndexError: occurs when the index of a sequence is out of range
7. KeyboardInterrupt: raised when the user inputs interrupt keys (Ctrl + C or Delete).
8. NameError: raised when a variable is not found in the local or global scope.
9. MemoryError: raised when programs run out of memory.
10. ValueError: occurs when the operation or function receives an argument with the right type but the wrong value.
11. ZeroDivisionError: raised when you divide a value or variable with zero.
12. SyntaxError: raised by the parser when the Python syntax is wrong.

13. IndentationError: occurs when there is a wrong indentation.
14. SystemError: raised when the interpreter detects an internal error.

Error Handling

- Like many other programming languages, it possible to program how an error/exception that is raised during runtime must be handled.
- It is also possible to throw custom exceptions in a program.

HANDLING EXCEPTIONS:

- In normal circumstances, the errors will stop the code execution and display the error message. To create stable systems, we need to anticipate these errors and come up with alternative solutions or warning messages.
- Exceptions are handled using the try, except, else and finally keywords.
- The try block lets you test a block of code for errors.
- The except block lets you handle the error.
- The else block executes when try block doesnot raise an exception
- The finally block lets you execute code regardless of the result of the try and except

Example:

```
try:
    print(x)
except:
    print('This is a custom error message')
```

- In the above code the variable x is not defined. Hence an exception will be triggered at runtime.
- But this exception will only cause the code in except block to be executed. Hence the output of the above code will be: 'This is a custom error message'

- We can also handle different types of exception using multiple except statements

```
try:
  print(1/0)
except ZeroDivisionError:
  print("You cannot divide a value with zero")
except:
  print("Something else went wrong")
```

try with else block:

When the `try` statement does not raise an exception, code enters into the `else` block.

Ex:

```
try:
  result = 1/3
except ZeroDivisionError as err:
  print(err)
else:
  print(f"Your answer is {result}")
```

finally block:

The `finally` keyword in the try-except block is always executed, irrespective of whether there is an exception or not.

```
In [4]:
def divide(x,y):
    try:
        result = x/y
    except ZeroDivisionError:
        print("Please change 'y' argument to non-zero value")
    except:
        print("Something went wrong")
    else:
        print(f"Your answer is {result}")
    finally:
        print(" I am always executed ")

divide(1,0)

divide(9,7)

divide(1,"3")
```

```
Please change 'y' argument to non-zero value
 I am always executed
Your answer is 1.2857142857142858
 I am always executed
Something went wrong
 I am always executed
```

Raising Exceptions in Python

As a Python developer, you have the option to throw an exception if certain conditions are met. It allows you to interrupt the program based on your requirement. To throw an exception, we have to use the `raise` keyword followed by an exception name.

Ex1:

```
value = 2000
if value > 1000:
    # raise the ValueError
    raise ValueError("Please add a value lower than 1,000")
else:
    print("Congratulations! You are the winner!!")
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
C:\Users\LAXMIY~1\AppData\Local\Temp/ipykernel_28792/4261113922.py in <module>
      2 if value > 1000:
      3     # raise the ValueError
----> 4     raise ValueError("Please add a value lower than 1,000")
      5 else:
      6     print("Congratulations! You are the winner!!")

ValueError: Please add a value lower than 1,000
```

Ex2:

```
x = -1
if x < 0:
 raise Exception('Only positive numbers are allowed')
```

 We could also raise custom errors of a particular type.

Ex3:

```
x = 'hello'
if type(x) != int:
  raise TypeError('Only integers allowed')
```