Exception Handling

1. What is an Exception?
2. What is the main objective of Exception Handling?
3. What is the meaning of Exception?

➢An unwanted and unexpected event that disrupts the normal flow of a program is called an Exception.

Examples of Exceptions:

FileNotFoundError

ZeroDivisionError

ValueError

➢The main objective of exception handling is Graceful Termination / Normal Termination of the application.

(i.e., we should not block our resources and we should not miss anything.)

What is the meaning of Exception?

S1:This weekend I am planning to go to my native place by bus.

S2:This weekend, I am planning to go to my native place by bus. If the bus is not available, then I will try for a train. If the train is not available, then I will try for a flight. Still, if it is not available, then I will go for a cab.

- Exception handling does not mean repairing an exception. We have to define an alternative way to continue the rest of the program normally…
- This way of defining alternatives is nothing but exception handling.

There are 2 types of errors:

- 1. Syntax Error
- 2. Run Time Error

1. Syntax Error:

Invalid syntax in the program. Not following syntax rules, then we get a syntax error.

Example:

```
x = 10
if x = = 10
   print("x value 10“)            Syntax Error: Invalid syntax
```

Example:

```
print "Hello World“
```

Syntax Error: Missing parentheses in call to print

Once all syntax is corrected, only then will program execution start.

•

2. Run Time Error (Exception):

There are no syntax errors in the program, but while executing it, we may get run-time errors.

- Run-time errors occur due to:

$$\left.\begin{array}{l}\text{Invalid input}\\ \text{Memory issue}\\ \text{Wrong programming logic}\end{array}\right\}\quad \text{AT Runtime}$$

```
 x = int(input("Enter x value: "))
y = int(input("Enter y value: "))
print("Result:", x / y)
```

If x = 10 and y = 2, no error.

If x = 10 and y = 0, it results in ZeroDivisionError.

Because of end-user input, this happens

Another error:

x = "Ten"

This causes a ValueError.

- File Handling Example:

```
f = open("abcabc.txt“, ’r’)       # If the file is not available
print(f.read())
```

FileNotFoundError

try:

Read data from Remote file located at London } Main Code: Reads data from the remote file.

except FileNotFoundError:

Use local files and continue the rest of the program normally

Alternative Code: Uses local files if the remote file is unavailable

Default Exception Handling in Python

- Every exception in Python is an object. For every exception type, its corresponding class is available.
- Whenever an exception occurs, the Python Virtual Machine (PVM) will create the corresponding exception object and check for handling code.
- If handling code is available, it will be executed, and the rest of the program will continue normally.
- If not, the PVM will terminate the program abnormally and print the corresponding exception information to the console. The rest of the program won't execute.
- To prevent abnormal termination, we should handle exceptions explicitly using try-except blocks.

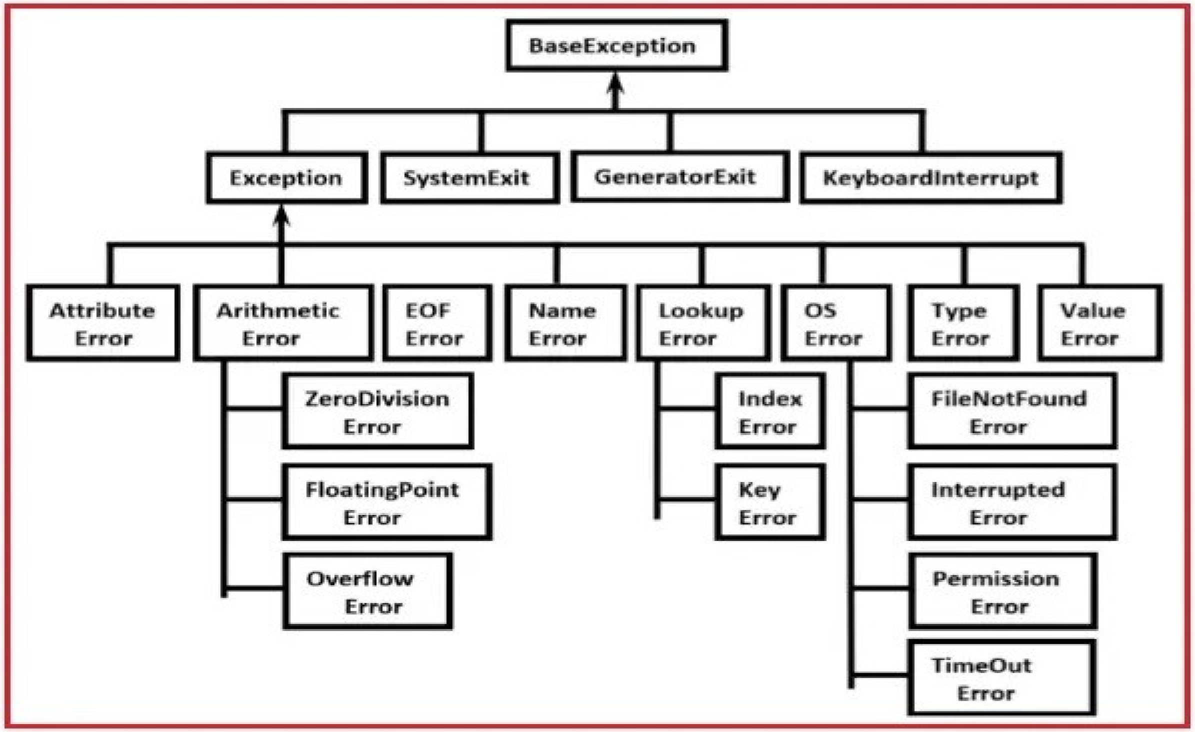
Example:

```
print("Hello")
 print(10 / 0)  # Error: ZeroDivisionError
 print("Hi")
```

Python's Exception Hierarchy

- Every exception in Python is a class.
- All exception classes are child classes of Base Exception, either directly or indirectly.
- Hence, Base Exception acts as the root for Python’s exception hierarchy.
- As a programmers, most of the time, we need to focus on handling exceptions and their child classes.

Exception Hierarchy Diagram



Customized Exception Handling using try-except

- It is highly recommended to handle exceptions.
- The code that may raise an exception is called Risky Code, and we must place it inside a try block.
- The corresponding handling code should be placed inside an except block.
- Syntax:
- try:

```
try:
        # Risky Code
except:
        # Handling Code / Alternative Code
```

Example (Without Exception Handling):

```
print("Hello")
print(10 / 0)  # Causes ZeroDivisionError
print("Hi")  # This will not execute
```

Example (With Exception Handling):

```
print("Hello")
try:
                print(10 / 0)  # Risky Code
except ZeroDivisionError:
               print(10 / 2)  # Handling Code / Alternative code
print("Hi")
```

Control Flow in try-except

```
try:
        stmt1
        stmt2  # Risky code Main Flow
        stmt3
except xxx:
        stmt4  # Alternative flow (Normal code)
stmt5
```

(2)

1. If there is no exception

Execution order: 1, 2, 3, 5 → Normal Termination (NT)

2. If an exception is raised at stmt-2 and the corresponding except block is matched

Execution order: 1, 4, 5 → Normal Termination (NT)

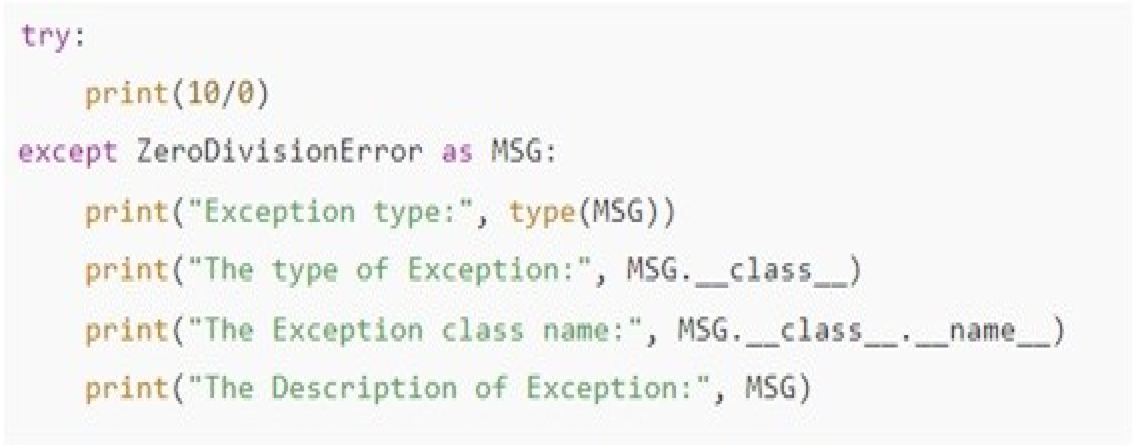
3. If an exception is raised at stmt-2 and the corresponding except block is not matched

Execution order: 1 → Abnormal Termination (AT)

4. If an exception is raised at stmt-4 and stmt-5

   Abnormal Termination (AT)

- Within the try block, if an exception is raised anywhere, the remaining statements in the try block will not execute, even if the exception is handled. Hence inside try block take only risky code and length of try block should be as less as possible.
- **In addition to try block, there may be chance of raising exceptions inside except and finally blocks also.**
- **3. If any statement which is NOT part of try block raises an exception, then it is always Abnormal Termination.**



```
try:
    print(10/0)
except ZeroDivisionError as MSG:
    print("Exception type:", type(MSG))
    print("The type of Exception:", MSG.__class__)
    print("The Exception class name:", MSG.__class__.__name__)
    print("The Description of Exception:", MSG)
```

try with multiple exception blocks

The way of handling an exception varies from exception to exception.

Hence, for every possible exception type, we should place a separate except block.

Using try with multiple except blocks is possible and recommended.

Syntax:

```
try:
            # Risky Code
except ZeroDivisionError:
            # Perform alternate arithmetic operation
except FileNotFoundError:
            # Use local file instead of remote file
```

Based on the raised exception, the corresponding except block will be executed.

Example:

```
try:
    x = int(input("Enter first value: "))
    y = int(input("Enter second value: "))
    print("The result:", x / y)
except ZeroDivisionError:
    print("Cannot divide by zero")
except ValueError:
     print("Please provide numerical
    values only")
```

$$\begin{tikzcd}
\text{BaseException} \\
\text{Exception} \arrow[u] \\
\text{ArithmeticError} \arrow[u] \\
\text{ZeroDivisionError} \arrow[u] \\
\text{FloatingPointError} \arrow[u] \\
\text{OverflowError} \arrow[u]
\end{tikzcd}$$

Default except Block in Python

- A default except block can be used to handle any type of exception.
- Generally, it is used to print exception information to the console.
- This block catches all exceptions, so it should always be placed at the end of multiple except blocks.

Syntax

```
except:
    # Statements to handle any exception
```

```
try:
        x = int(input("Enter first number: "))
        y = int(input("Enter second number: "))
        print("Result:", x / y)
except ZeroDivisionError:
        print("ZeroDivisionError: Cannot divide by zero")
except:
        print("Default Except Block: An unexpected error occurred")
```

➢

finally Block in Python

The finally block is used for resource cleanup (e.g., closing database connections, files, etc.).
It executes regardless of whether an exception occurs or not.
Ensures that necessary cleanup always happens, even if an error occurs.

Example (Without finally)

```
try:
    open DB connection
    Read data from DB
    close DB connection
    # Resource deallocation
except:
    # Handling Code
```

Issue: If an exception occurs before closing the DB connection, the connection remains open, causing resource leaks.

Example (With finally)

```
try:
    open DB connection
    Read data from DB
except:
    # Handling Code
finally:
    close DB connection
    # Cleanup code (Executes always)
```

Solution: The DB connection is closed whether an exception occurs or not.

Syntax

```
try:
        # Risky Code
except:
        # Handling Code
finally:
        # Cleanup Code / Resource Deallocation (Always Executes)
```

1. The finally block always executes, whether an exception is handled or not.
2. Used for closing resources like files, databases, or network connections.
3. Ensures proper resource management in Python programs.

Case 1: No Exception Occurs

```
try:
    print("try")
except:   print("except")
finally:  print("finally")
```

Output:

```
        try
        finally
```

Case 2: Exception Occurs and is Handled

```
try:
    print("try")
     print(10 / 0)
# ZeroDivisionError
except ZeroDivisionError:
    print("except")
finally:
    print("finally")
```

Output:

```
          try
          except
          finally
```

Case 3: Exception Occurs and is NOT Handled

```
try:
     print("try")
    print(10 / 0)  #
    ZeroDivisionError
except ValueError:
    # Wrong exception type

    print("except")
finally:
     print("finally")
```

Output:

```
    try
    finally
    ZeroDivisionError
    (Uncaught)
```

os._exit(0) vs finally Block in Python

The finally block always executes, except in one special case:

When os._exit(0) is used, the finally block will NOT execute.

Why?

os._exit(0) immediately terminates the Python process.

It shuts down the Python Virtual Machine (PVM), skipping any cleanup operations.

This means the finally block won't run in this scenario.

Understanding os._exit(0)

The argument 0 represents a successful (normal) termination.

A non-zero argument indicates an abnormal termination.

This exit status is internally used by the operating system.

```
import os
try:
    print("try")
os._exit(0)  # Immediately terminates the program
except ValueError:
     print("except")
finally:
 print("finally")  # This won't execute
```

Output: try

Control Flow in try-except-finally

```
try:
        stmt1
        stmt2  # Risky Code
        stmt3
except:
        stmt4  # Handling Code
finally:
         stmt5  # Cleanup Code (Always Executes)
stmt6  # Normal Code (Outside try-except-finally)
```

1. If there is no exception:
2. If an exception occurs at stmt2 and a corresponding except block exists::
3. If an exception occurs at stmt2, but no matching except block exists:
4. If an exception occurs at finally:
5. If an exception occurs at stmt5 (inside finally) or stmt6:

If there is no exception:

Execution Order:

1 → 2 → 3 → 5 →6

Normal Termination (NT)

If an exception occurs at stmt2 and a corresponding except block exists::

Execution Order:

1 → 4 → 5 → 6

Normal Termination (NT)

If an exception occurs at stmt2, but no matching except block exists:

Execution Order:

1 → 5

Abnormal Termination (AT)

If an exception occurs at finally:

Execution Order:

The finally block executes, but if an exception occurs within it, it leads to abnormal termination.

If an exception occurs at stmt5 (inside finally) or stmt6:

Abnormal Termination (AT)

```
try:
        stmt-1
        stmt-2
        stmt-3
        try:
        stmt-4
        stmt-5
        stmt-6
        except xxx:
        stmt-7      finally :
         stmt-8
              stmt-9
except yyy:
        stmt-10
finally:
        stmt-11
stmt12
```

Case 1: No Exception Occurs

Execution Order: 1 → 2 → 3 → 4 → 5 → 6 → 8 → 9 → 10 → 11 → 12 NT

Case 2: Exception at stmt2, Handled

Execution Order: 1 → 10 → 11 → 12 NT

Case 3: Exception at stmt2, Not Handled

Execution Order: 1 → 11 AT

Case 4: Exception at stmt5, Handled in Inner except Block

Execution Order: 1 → 2 → 3 → 4 → 7 → 8 → 9 → 10 → 11 → 12 NT

Case 5: Exception at stmt5, Not Handled in Inner except Block, But Handled in Outer except Block

Execution Order: 1 → 2 → 3 → 4 → 8 → 10 → 11 → 12 NT

Case 6: Exception at stmt5, Not Handled in Any except Block

Execution Order: 1 → 2 → 3 → 4 → 8 → 11 AT

Case 7: Exception at stmt7, Handled

Execution Order: 1 → 2 → 3 → 4 → 5 → 6 → 8 → 10 → 11 → 12 NT

Case 8: Exception at stmt7, Not Handled

Execution Order: 1 → 2 → 3 → 4 → 5 → 6 → 8 → 11 AT

Case 9: Exception at stmt8, Handled

Execution Order: 1 → 2 → 3 → ... → 8 → 10 → 11 → 12 NT

Case 10: Exception at stmt8, Not Handled

Execution Order: 1 → 2 → 3 → ... → 8 → 11 AT

Case 11: Exception at stmt9, Not Handled

Execution Order: 1 → 2 → 3 → ... → 8 → 10 → 11 → 12 NT

Case 12: Exception at stmt9, Not Handled

Execution Order: 1 → 2 → 3 → ... → 8 → 112 AT

Case 13: Exception at stmt10

Execution Order: Always Abnormal Termination (AT)

Case 14: Exception at stmt11 or stmt12 - AT

else Block with try-except-finally, Loops, and Conditionals

1. if-else Statement

   If the condition is false, the else block executes.

```
Example:  x = 10
if x > 10:
 print("x > 10")
else:
print("x is not > 10")                Output:  x is not > 10
```

2. for-else Statement

   The else block runs only if the loop completes without a break.

```
 Example:      for x in range(10):
            print("The value of x:", x)
        else:    print("All values of x are printed") : The value of x: 9
All values of x are printed
```

```
Example with break:    for x in range(10):
      if x > 5:
          break
print("The value of x:", x)
else:
print("All values of x are printed")
```

```
Output:
The value of x: 0
The value of x: 1...
The value of x: 5
```

try-except-else-finally Block in Python

1. try Block → Contains risky code that might raise an exception.
2. except Block → Executes if an exception occurs inside try.
3. else Block → Executes only if no exception occurs in try.
4. finally Block → Executes always, regardless of whether an exception occurred or not.

else executes because try had no exceptions.

finally executes always

Syntax :

```
try:
    # Risky Code
except ExceptionType:
    # Handling          Code (Executes if
    an exception occurs)
else:   # Executes if NO exception occurs
    in try
finally:   # Cleanup Code (Executes
    always)
```

```
try:
 num = int(input("Enter a number: "))
print("You entered:", num)
except ValueError:
    print("Invalid input! Please enter a number.")
else:   print("No exception occurred. Else block
    executed.")
finally:   print("Finally block executed.")
```

Case 1: Valid Input (10)

```
Enter a number: 10
You entered: 10
No exception occurred. Else block executed.
Finally block executed.
```

Example 1:No Exception Occurs

```
try:
    print("try")
except:
    print("except")
else:
    print("else")
finally:
    print("finally")
```

Output:

```
try
else
finally
```

else executes because there was no exception.

finally executes always.

Example 2: Exception Occurs

```
try:
    print("try")
    print(10 / 0)  # ZeroDivisionError
except:
    print("except")
else:
    print("else")
finally:
    print("finally")
```

Output:

```
try
except
finally
```

except executes because of ZeroDivisionError.

else is skipped.

finally executes always.

Example 3: in correct else Placement (Syntax Error)

```
try:
        print("try")
 else:
        print("else")
finally:
        print("finally")
```

Output:

invalid Syntax!

else must always follow except.

The assert statement

The assert statement is useful to ensure that a given condition is True. If it is not ture raise AssertionError.

Syntax

```
        assert  condition, message
```

If the condition is False, then the exception by the name AssertionError is raised along with the message written in the assert statement.

EX

```
try:
        x=int(input('Enter a number between 5 and 10:'))
        assert  x>=5  and  x<=10, "your  input  is  not  correct"
        print('The  number  entered:',x)
except   AssertionError  as  obj:
        print(obj)
```

User-Defined Exceptions

- The programmer create his own exceptions are called ‘user-defined exceptions’ or ‘custom exceptions’
- since all exceptions are classes, the programmer is supposed to create his own exception as a class and make this class as a subclass to the built in ‘Exception’ class.

```
class MyException(Exception):
        def __init__(self,arg):
                self.msg=arg
```

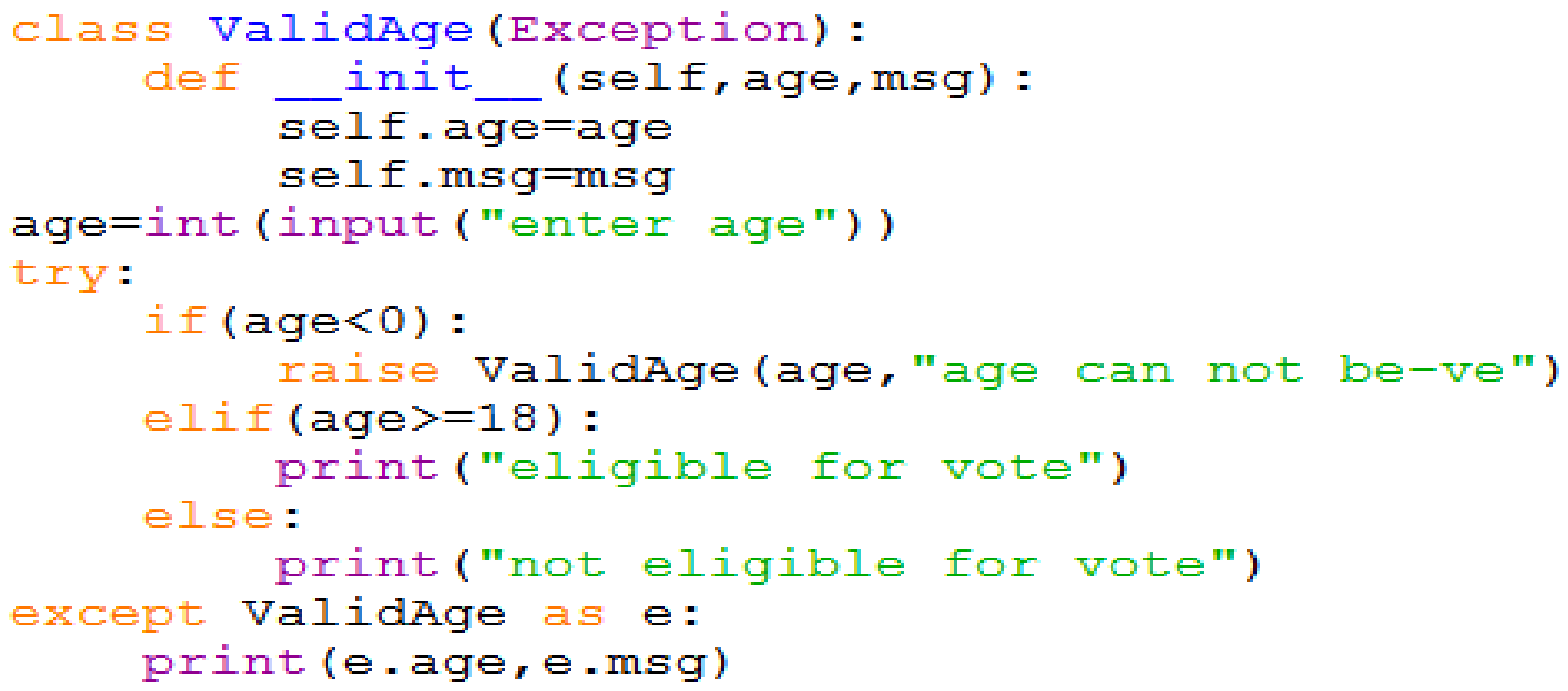
- When the programmer suspects the possibility of exception , he should raise his own exception using ‘raise’ statement as

```
raise MyException(‘message’)
```

- The programmer can insert the code inside a ‘try’ block and catch the exception using except block as

```
try:
        code
except MyException as me:
        pritnt(me)
```

Example program



```
class ValidAge(Exception):
    def __init__(self,age,msg):
        self.age=age
        self.msg=msg
age=int(input("enter age"))
try:
    if(age<0):
        raise ValidAge(age,"age can not be-ve")
    elif(age>=18):
        print("eligible for vote")
    else:
        print("not eligible for vote")
except ValidAge as e:
    print(e.age,e.msg)
```

```
enter age-12
-12 age can not be-ve
>>>
enter age45
eligible for vote
>>>
enter age9
not eligible for vote
```