# Object Oriented Programming- III

## Exception vs Error

An exception is an event that occurs during the execution of a program that disrupts the normal flow of the program's instructions. When an exception is raised, the program execution is immediately halted, and the control is transferred to a specific block of code called an exception handler. Exceptions allow you to handle errors or exceptional situations gracefully by providing a mechanism to catch and recover from them.

On the other hand, an error is a more general term that refers to any unexpected or undesirable behaviour in a program. It can encompass a wide range of issues, including syntax errors, logical errors, runtime errors, or exceptions. Errors are typically caused by mistakes in the code or by external factors, such as incorrect input or resource limitations.

## Exception handling

Exception handling in Python allows you to catch and handle exceptions gracefully  and providing a way to handle unexpected situations. Python provides a try and except block for handling exceptions.

**The basic structure of a try-except block is as follows:**

```
try:
    # Code that may raise an exception
    # ...
except ExceptionType:# Code to handle exception
```

**Here's a step-by-step explanation of how exception handling works in Python:**

- The code within the try block is executed.
- If an exception occurs during the execution of the try block, the remaining code within the try block is skipped.
- The program flow is transferred to the corresponding except block based on the type of exception raised.
- The code within the except block is executed, which handles the exception.
- After the except block is executed, the program continues to run from the point immediately after the try-except block.
- Multiple except blocks can be added to handle different types of exceptions.

**Here's an example that demonstrates exception handling in Python:**

```
try:
    num1 = int(input("Enter a numerator: "))
    num2 = int(input("Enter a denominator: "))
    result = num1 / num2
    print("Result:", result)
except ValueError:
    print("Invalid input. Please enter integers.")
except ZeroDivisionError:
    print("Error: Division by zero is not allowed.")
else:
    print("Division performed successfully.")
```

In this example, the program asks the user for a numerator and a denominator to perform division. If the user enters a non-integer value, a ValueError is raised and caught by the first except block. If the user enters zero as the denominator, a ZeroDivisionError is raised and caught by the second except block. If the division is successful with no exceptions, the else block is executed.

# Try Except and Finally

In Python, the try-except-finally statement provides a way to handle exceptions and. The finally block is executed regardless of whether an exception occurred or was caught.

**Here's the structure of a try-except-finally block:**

```
try:
    # Code that may raise an exception
    # ...
except ExceptionType:
    # Code to handle the exception
    # ...
finally:
    # Code that will always execute
    # ...
```

**The execution flow works as follows:**

- The code within the try block is executed.
- If an exception occurs during the execution of the try block, the remaining code within the try block is skipped.
- If an exception occurs, the program flow is transferred to the corresponding except block based on the type of exception raised.
- The code within the except block is executed to handle the exception.
- Whether an exception occurred or not, the code within the finally block is executed. It is used for cleanup actions that must be performed, such as closing files or releasing resources, regardless of whether an exception occurred or was caught.
- After the finally block is executed, the program continues to run from the point immediately after the try-except-finally block

**Here's an example that demonstrates the usage of try-except-finally:**

```
try:
    file = open("myfile.txt", "r")
    # Perform some operations on the file
except FileNotFoundError:
    print("File not found.")
finally:
    file.close()  # Ensure the file is always closed, even if an
exception occurred
```

In this example, the program attempts to open a file called "myfile.txt" for reading. If the file is not found, a FileNotFoundError is raised and caught by the except block. Regardless of whether an exception occurred or not, the finally block ensures that the file is closed properly by calling the close() method.

# Custom Exceptions

In Python, you can create custom exceptions by defining a new class that inherits from the built-in Exception class or any of its subclasses. This allows you to create your own exception types that can be raised and caught like any other exception.

**Here's an example of how to create a custom exception:**

```
class CustomException(Exception):
    pass
```

In this example, we define a new class called CustomException that inherits from the Exception class. The pass statement indicates that we're not adding any additional behaviour to the exception class, but you can customise it by adding your own methods or attributes.

Once you have defined your custom exception class, you can raise an instance of that exception using the raise statement.

**Here's an example:**

```
def validate_input(value):
    if not isinstance(value, int):
        raise CustomException("Invalid input. Expected an
integer.")

try:
    validate_input("abc")
except CustomException as e:
    print("Error:", str(e))
```

In the above example, we define a function called validate_input() that checks if the provided value is an integer. If it's not, we raise a CustomException with a descriptive error message. In the except block, we catch the CustomException and print the error message.

## Exceptions Resolution Order

In Python, when an exception is raised, the interpreter looks for an appropriate exception handler to handle the exception. The resolution order for exception handling follows a specific hierarchy. Here's the general order in which Python searches for exception handlers:

- The except clauses within the closest enclosing try statement are checked in top-to-bottom order. If the raised exception matches the type specified in an except clause, that block is executed, and the search for an exception handler stops.
- If no matching except clause is found within the closest enclosing try statement, the exception propagates to the next outer try statement (if any) and follows the same resolution process.
- If the exception reaches the topmost level without finding a matching except clause, the default exception handler is invoked.

**To illustrate this resolution order, consider the following example**

```
try:
    # Code block A
    try:
        # Code block B
        raise ValueError("Exception B")
    except IndexError:
        print("IndexError handled")
except ValueError:
    print("ValueError handled")
```

In this example, an exception of type ValueError is raised within Code block B. Python starts searching for an exception handler from the closest enclosing try statement. It checks the except clauses within Code block B, but there is no matching handler for ValueError. So, the exception propagates to the next outer try statement, which is the one enclosing Code block A. Here, the exception matches the except ValueError clause, and the corresponding message "ValueError handled" is printed.

If there were no matching handler for ValueError in the outer try statement enclosing Code block A, the exception would continue propagating to even higher levels until it either finds a matching handler or reaches the topmost level, where the default exception handler would be invoked.