Module V: Object Oriented Programming & Exception Handling

What is a Class?

A **class** in Python is a blueprint or template that defines the structure and behavior of objects. It groups related variables (called **attributes**) and functions (called **methods**) into a single logical unit. Classes help in organizing and reusing code efficiently. For example, you can create a class named Car that defines common properties of all cars like brand, model, and color, along with behaviors like starting or stopping the engine. A class doesn't represent a single car; it's just a design from which cars can be made.

Objects in Python

An **object** is an **instance** of a class. When you create an object, Python allocates memory for it and gives it its own set of data defined by the class. Each object can have different values for its attributes but will share the same methods. For instance, if Car is a class, you can create multiple car objects like car1 and car2, each representing a real car with its own brand and model.

Example of Class and Object

```
class Car:
    def start(self):
        print("The car has started.")


# Creating objects
car1 = Car()
car2 = Car()
```

```
# Accessing method using objects

car1.start()

car2.start()
```

Explanation:
In this example, Car is a class that has one method called start(). The objects car1 and car2 are created from the Car class. When we call car1.start() and car2.start(), both objects use the same method defined in the class but act as separate entities in memory.

The self Keyword

The **self** keyword in Python represents the **current object** of the class. It is used to access the class variables and methods within the class. When a method is defined inside a class, it must have self as its first parameter so that Python knows which object is calling that method. Although you can name it anything, using self is a common convention and makes code easier to read and understand.

Example of self

```
class Student:

    def display(self, name):

        print(f"Hello, my name is {name}")


# Creating an object

student1 = Student()


# Calling method

student1.display("Alice")
```

Explanation:
Here, self refers to the object student1. When we call student1.display("Alice"), Python automatically passes the object reference to self, even though we only passed "Alice". This allows the method to know which specific object is being used.

Constructors in Python

A **constructor** in Python is a special method named **__init__()** that automatically runs whenever a new object is created from a class. It is mainly used to **initialize the object's attributes** with values. This method helps set up the object right after it's created, so you don't have to assign values separately later.

Example of a Constructor

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def introduce(self):
        print(f"My name is {self.name} and I am {self.age} years old.")

# Creating objects
person1 = Person("John", 25)
person2 = Person("Emma", 30)

# Calling method
person1.introduce()
person2.introduce()
```

Explanation:
In this example, the __init__() method initializes the name and age attributes for each object created. When person1 = Person("John", 25) is executed, Python automatically calls the constructor to assign "John" and 25 to that object's attributes. This makes each object unique with its own data.

Example code:

```
## Modeling a Bank account

## Define a class for bank account


class BankAccount():

   def __init__(self, owner, balance=0):

      self.owner=owner

      self.balance=balance


   def deposit(self, amount):

      self.balance+=amount

      print(f'{amount} is deposited. New balance is {self.balance}')


   def withdraw(self, amount):

      if amount>self.balance:

         print('Insuffient Funds')


      else:

         self.balance-=amount

         print(f'{amount} is withdrawn. New balance is {self.balance}')
```

```
    def getBalance(self):
        return self.balance


# Create a object
account=BankAccount('Mark',173000)
print(account.balance)
```

Types of Variables

Class Variables:

A **class variable** is a variable that belongs to the **class itself** and is **shared by all objects** created from that class. It is defined **outside of any method**, usually right below the class definition. When one object modifies a class variable, the change reflects in all other objects since they share the same memory for that variable. Class variables are useful when you want a common piece of data for all instances — for example, a bank name that stays the same for all account holders.

Instance Variables:

An **instance variable** belongs to a **specific object**. It is defined inside the **constructor (__init__)** or inside any instance method using the self keyword. Each object has its **own copy** of instance variables, meaning that changing one object's instance data doesn't affect another. These variables store information unique to each object, such as a person's name, balance, or age.

Example of Class and Instance Variables

```
class Example:
    class_var = "Shared by all objects"  # Class variable

    def __init__(self, value):
        self.instance_var = value  # Instance variable

# Creating objects
obj1 = Example("Object 1 data")
obj2 = Example("Object 2 data")

print(Example.class_var)
print(obj1.instance_var)
print(obj2.instance_var)
```

Explanation:

Here, class_var is a class variable shared by all objects, while instance_var is unique for each object. If you change Example.class_var, the new value is seen by all objects. But modifying obj1.instance_var affects only that specific object.

Types of Methods in a Class

Python supports **three main types of methods** inside a class:

1. **Instance Method** – Works with **instance variables** and uses the self keyword. It performs operations related to a specific object.

2. **Class Method** – Works with **class variables** and uses the cls keyword. It is defined using the @classmethod decorator and can modify class-level data shared among all objects.
3. **Static Method** – Does not depend on the class or object data. It is defined using the @staticmethod decorator and is usually used for general-purpose functions related to the class.

Code Example: BankAccount

```
class BankAccount:
    bank_name = "SBI"   # Class variable


    def __init__(self, name, balance):
        self.name = name         # Instance variable
        self.balance = balance    # Instance variable


    def deposit(self, amount):  # Instance Method
        self.balance += amount
        print(f"{amount} deposited. New balance: {self.balance}")


    def withdraw(self, amount):  # Instance Method
        if self.balance >= amount:
            self.balance -= amount
            print(f"{amount} withdrawn. New balance: {self.balance}")
        else:
            print("Insufficient balance!")
```

```
@classmethod
    def change_bank(cls, new_bank):  # Class Method
        cls.bank_name = new_bank

    @staticmethod
    def info():  # Static Method
        print("Welcome to our banking system!")

# Using class
BankAccount.info()

# Creating objects
acc1 = BankAccount("Balaji", 7000)
acc1.deposit(500)
acc1.withdraw(300)

# Changing bank name using class method
BankAccount.change_bank("HDFC")
print("New Bank:", BankAccount.bank_name)
```

Explanation of the above Code

1. Class Variable (bank_name)

The variable bank_name is a **class variable** shared by all BankAccount objects. Initially, it is set to "SBI". When the class method changes it to "HDFC", all accounts automatically reflect the new bank name.

2. Instance Variables (name, balance)

The attributes name and balance are **instance variables**, meaning each account object has its own name and balance. For example, acc1 may have "Balaji" as its name and 7000 as its balance. These values belong only to that specific object.

3. Instance Methods (deposit, withdraw)

These methods use the self keyword to work with the object's data.

- The deposit() method increases the balance by a given amount and displays the updated balance.
- The withdraw() method checks if the balance is sufficient before subtracting the amount.
  These methods operate on each object's individual balance.

4. Class Method (change_bank)

The change_bank() method is defined using the @classmethod decorator and uses the cls keyword. It modifies the **class variable** bank_name for the entire class. When BankAccount.change_bank("HDFC") is called, all account objects now belong to "HDFC" instead of "SBI".

5. Static Method (info)

The info() method is defined using @staticmethod. It does not use self or cls, meaning it neither accesses instance variables nor class variables. It simply prints a message. Static methods are used when you want to perform a general task related to the class but not tied to any object.

What is Inheritance in Python

Inheritance is one of the most important concepts in Object-Oriented Programming (OOP). It allows one class (called the **child** or **subclass**) to **inherit properties and behaviors** (variables and methods) from another class (called the **parent** or **superclass**).

Inheritance helps in **code reusability** - you can define common functionality in a base class and extend it in derived classes without rewriting the same code multiple times.
In simple terms, inheritance allows a child class to use the features of its parent class, and it can also add new features or modify existing ones.

Syntax of Inheritance

```
class Parent:
    # parent class members
    pass


class Child(Parent):
    # child class members
    pass
```

Here, the Child class inherits everything from the Parent class, meaning the Child can access all public methods and attributes defined in Parent.

Single Inheritance

In **single inheritance**, a **child class inherits from only one parent class**. This is the simplest form of inheritance and is used when you want to create a class based on another single class.

```
# Parent class
class Animal:
    def speak(self):
        print("Animal makes a sound")


# Child class
class Dog(Animal):
    def bark(self):
        print("Dog barks")


# Create object of Dog
d = Dog()
d.speak()   # Inherited from Animal
d.bark()    # Defined in Dog
```

Explanation:

Here, Animal is the parent class, and Dog is the child class. The Dog class inherits the speak() method from Animal, so when we call d.speak(), it executes the parent class method. The Dog class also defines its own method bark(), which prints “Dog barks.”
This shows that the child class can **use** and **extend** the functionality of its parent class.

Multilevel Inheritance

In **multilevel inheritance**, a class inherits from a child class, which in turn inherits from another parent class - forming a **chain of inheritance**.
In simple words, the grandchild class inherits from both its parent and grandparent classes.

```
class GrandFather:
    def land(self):
        print("GrandFather has land.")


class Father(GrandFather):
    def car(self):
        print("Father has a car.")
```

```
class Son(Father):

    def bike(self):

        print("Son has a bike.")


s = Son()

s.land()

s.car()

s.bike()
```

Explanation:
Here, the class Father inherits from GrandFather, and Son inherits from Father.

- GrandFather -> has land()
- Father -> has car() and inherits land()
- Son -> has bike() and inherits both land() and car()
  When we create an object of Son, it can access all methods from Son, Father, and GrandFather, demonstrating the multilevel chain of inheritance.

Multiple Inheritance

In **multiple inheritance**, a class can inherit from **more than one parent class**. This means the child class can access properties and methods from multiple base classes.

```
class Father:

    def land(self):

        print('Father has land')


class Mother:
    def money(self):
        print('Mother has money')


class Child(Father, Mother):
    def bike(self):
        print('Child has bike')


c = Child()
c.land()
c.money()
c.bike()
```

Explanation:
In this example, the class Child inherits from both Father and Mother.
This means the Child class can access:

- land() method from Father
- money() method from Mother
- bike() method defined inside Child itself
  When c.land() and c.money() are called, the methods of both parent classes are available to the child. This demonstrates that multiple inheritance allows combining functionalities from different sources into one class.

Hierarchical Inheritance

In **hierarchical inheritance**, **multiple child classes inherit from a single parent class**.
This means one parent class is shared among many child classes, allowing them to use common functionality from the same parent.

```
class Father:
    def land(self):
        print('Father has land')

class Child1(Father):
    def money(self):
        print('Child1 has money')

class Child2(Father):
    def bike(self):
        print('Child2 has bike')

c1 = Child1()
c1.land()
c1.money()

c2 = Child2()
c2.land()
c2.bike()
```

Explanation:

Here, Father is the parent class, and both Child1 and Child2 are child classes.

- Father defines the method land().
- Child1 inherits land() and defines its own method money().
- Child2 also inherits land() and defines its own method bike().
  This shows that multiple child classes can share and reuse the code from a single parent class while still having their own unique methods.

What is Method Overriding

Method Overriding is a concept in **inheritance** where a **child class defines a method with the same name as a method in its parent class**.
When this happens, the **child class version of the method overrides (replaces)** the one in the parent class.
It allows the child class to provide a **specific implementation** of a method that already exists in its parent class.
Method overriding is used when the behavior of the parent class method is not suitable for the child class, and we want to change or extend it.

Rules of Method Overriding

1. The parent and child class must have a **method with the same name**.
2. The **child class's method** must have the **same parameters** (number and type).
3. When the child class object calls the method, the **child's method is executed instead of the parent's**.
4. Overriding happens only when **inheritance** exists between classes.

Example: Basic Method Overriding

```
class Animal:
    def sound(self):
        print("Animals make different sounds")


class Dog(Animal):
    def sound(self):
        print("Dog barks")  # overriding parent method


d = Dog()
d.sound()   # Output: Dog barks
```

Explanation:

Here, the class Animal defines a method sound() that prints “Animals make different sounds.”
The class Dog inherits from Animal and also defines a method with the **same name** sound().
When the Dog object d calls d.sound(), Python first checks the method inside the Dog class.
Since Dog has its own sound() method, it **overrides** the parent version, and “Dog barks” is printed instead of the parent’s message.
This is how **method overriding** replaces the parent class behavior with the child class’s version.

Using super() to Call the Parent Method

Sometimes, we may want to **override a parent method** but still **call the parent’s version** as part of the new behavior.
To achieve this, Python provides the **super()** function.
It allows the child class to call a method from its parent class without directly naming the parent.

```
class Animal:
    def sound(self):
        print("Animals make sound")


class Dog(Animal):
    def sound(self):
        super().sound()    # call parent method
        print("Dog barks")


d = Dog()
d.sound()
```

Explanation:

In this example, both Animal and Dog classes have a method named sound().
The Dog class overrides the method but also calls super().sound() to execute the parent class version before adding its own behavior.
So when we run d.sound(), Python first calls Animal’s sound() method (via super()), printing “Animals make sound,” and then continues with the child’s method, printing “Dog barks.”
The final output is:

Animals make sound

Dog barks

This approach is very useful when you want to **extend the functionality** of the parent method instead of completely replacing it.

What is Data Hiding

Data Hiding is a fundamental concept in Object-Oriented Programming that focuses on **protecting the internal details of an object from direct access** by external code.
It ensures that sensitive or critical data cannot be directly modified or viewed from outside the class.
In Python, data hiding is achieved by **using private variables**, which are defined with a **double underscore (__) prefix** before the variable name.
This means such variables cannot be accessed directly using the object name, helping to maintain **data security and integrity**.

Why Data Hiding is Important

Data hiding helps to:

- Prevent accidental modification of important values.
- Maintain control over how data is accessed or changed.

- Improve security and reliability of the program.
- Enforce **encapsulation**, which is one of the key principles of OOP.

By using private variables, the class controls access to its data through **getter** and **setter** methods, ensuring that values remain valid and consistent.

Example: Basic Data Hiding

```
class BankAccount:

    def __init__(self, owner, balance=0):

        self.owner = owner

        self.__balance = balance  # private variable


    def getBalance(self):

        return self.__balance


# Create an object

account = BankAccount('Balaji Kudumu', 173000)

print(account.getBalance())


# print(account.__balance)  # This will cause an AttributeError
```

Explanation:

In this example, the variable __balance is **private** because it has two underscores before its name.
That means it **cannot be accessed directly** from outside the class using account.__balance.
To safely access it, the class provides a **getter method** named getBalance(), which returns the current balance value.
If you try to print account.__balance, Python will raise an **AttributeError** since

private data is hidden from external access.
This is how data hiding protects important information, such as bank account balances, from unauthorized changes.

Accessing Private Data Safely

Although direct access to private variables is restricted, we can still **read or modify** them safely using special methods called **getters** and **setters**.

- A **getter method** allows reading the value of a private variable.
- A **setter method** allows modifying the value but usually includes validation to prevent invalid data from being stored.

This approach ensures that data remains valid and secure while still allowing controlled access when needed.

Example: Getter and Setter Methods

```
class Account:
    def __init__(self, name, balance):
        self.__name = name
        self.__balance = balance

    # getter
    def get_balance(self):
        return self.__balance

    # setter
    def set_balance(self, new_balance):
        if new_balance >= 0:
            self.__balance = new_balance
        else:
            print("Invalid balance!")


acc = Account("Raj", 2000)

print(acc.get_balance())   # 2000

acc.set_balance(3000)

print(acc.get_balance())   # 3000

acc.set_balance(-500)      # Invalid balance!

print(f"Updated Balance: {acc.get_balance()}")
```

Explanation:

In this example, the class Account defines two private variables — __name and __balance.
The get_balance() method is a **getter**, which retrieves the current balance safely.
The set_balance() method is a **setter**, which allows updating the balance **only if the new value is valid** (non-negative in this case).
When acc.set_balance(3000) is called, the balance updates successfully, but acc.set_balance(-500) shows "Invalid balance!" because the setter protects the variable from invalid data.
This ensures that even though private data is hidden, it can still be **accessed in a controlled and safe manner** through these methods.

Key Points about Data Hiding

- Private variables are created using **__ (double underscore)** before the variable name.
- They **cannot be accessed directly** outside the class.

- Use **getter methods** to read data safely.
- Use **setter methods** to modify data with proper validation.
- Data hiding enforces **encapsulation**, which ensures that an object's internal state is protected and managed only through defined interfaces.

Difference Between an Error and an Exception

In Python, both **Errors** and **Exceptions** indicate that something went wrong while executing a program — but they are **not the same thing**.
They differ mainly in **when they occur**, **how they can be handled**, and **their severity**.

1. Error:

Errors are problems that occur **because of mistakes in the code**, such as syntax issues or logical mistakes, which **stop the program from running**.
Errors are usually **detected by the Python interpreter before the program starts executing** (at compile time).

Errors **cannot be handled** using try-except blocks if they are syntax-related.
You need to **fix the code** before running the program again.

Common Examples of Errors:

- **SyntaxError** → Occurs when Python code is written incorrectly.
- **IndentationError** → Happens when indentation (spacing) is not proper.
- **NameError** → When a variable is not defined before using it.

Example:

```
# Example of a Syntax Error
print("Hello"   # Missing closing parenthesis — SyntaxError


# Example of a Name Error
print(x)  # x is not defined
```

Explanation:

- The first example causes a **SyntaxError** because the closing parenthesis) is missing — the interpreter cannot even start running the program.
- The second one gives a **NameError** because the variable x was never defined.
  Both are **errors** that must be corrected before execution continues.

2. Exception:

Exceptions are issues that occur **while the program is running** (after it starts execution).
They occur when something **unexpected happens**, such as dividing by zero or trying to open a missing file.

Unlike syntax errors, **exceptions can be handled** gracefully using **try–except blocks**.
This allows the program to continue running even after an exception occurs.

Common Examples of Exceptions:

- **ZeroDivisionError** -> Dividing a number by zero.
- **FileNotFoundError** -> Trying to open a file that doesn’t exist.
- **ValueError** -> Passing invalid data to a function.

Example:

```
# Example of an Exception
num1 = 10
num2 = 0

try:
    result = num1 / num2  # ZeroDivisionError
    print(result)
except ZeroDivisionError:
    print("Cannot divide by zero! Please check your input.")
```

Output:

```
Cannot divide by zero! Please check your input.
```

Explanation:

Here, the division num1 / num2 causes a **ZeroDivisionError**, but because it is inside a try-except block, the program **doesn't crash**.
Instead, the exception is **caught** and a friendly message is displayed.
This is what makes exceptions different from errors — they can be **handled safely**.

Handling Exceptions in Python

Exception Handling is a mechanism that allows a program to **respond to runtime errors** (exceptions) in a controlled way, **without crashing** the entire program.

Instead of stopping execution when an error occurs, Python lets you **“catch”** the error and **handle it gracefully** using try–except blocks.

The try–except Block

The try–except block is used to **test a block of code for errors** and **handle them** if any occur.

Basic Structure

```
try:
    # Code that may cause an exception
except SomeException:
    # Code to handle the exception
```

If no exception occurs, the except block is skipped.

Full Form: try–except–else–finally

Python allows four optional parts in an exception-handling block:

1. **try block** -> The code that might raise an exception.
2. **except block** -> The code that runs **if an exception occurs**.
3. **else block** -> The code that runs **if no exception occurs**.
4. **finally block** -> The code that **always runs**, whether an exception occurred or not (often used for cleanup).

Code Example:

```
try:
    num = int(input("Enter a number: "))
    result = 10 / num
except ZeroDivisionError:
    print("You cannot divide by zero!")
except ValueError:
    print("Please enter a valid number!")
else:
    print("Division successful! Result:", result)
finally:
    print("Program finished.")
```

Explanation (Line by Line)

try block:

```
try:
    num = int(input("Enter a number: "))
    result = 10 / num
```

- The code inside the try block is *monitored* for exceptions.
- The statement int(input(...)) can raise a ValueError if the user enters something that is not a number.
- The statement 10 / num can raise a ZeroDivisionError if the user enters 0.

except blocks:

```
except ZeroDivisionError:
    print("You cannot divide by zero!")
except ValueError:
    print("Please enter a valid number!")
```

- Each except block handles a **specific type of exception**.
- If the user enters 0 → ZeroDivisionError occurs → prints “You cannot divide by zero!”

- If the user enters non-numeric data → ValueError occurs → prints "Please enter a valid number!"

else block:

```
else:
    print("Division successful! Result:", result)
```

- The else block runs **only if no exception occurs** in the try block.
- It's often used for code that should run **only when everything goes right**.

finally block:

```
finally:
    print("Program finished.")
```

- The finally block **always executes**, whether an exception occurred or not.
- Typically used for cleanup tasks like closing files, releasing memory, or disconnecting from a database.

Example Output Scenarios

Case 1: Valid input

```
Enter a number: 2
Division successful! Result: 5.0
Program finished.
```

Case 2: Division by zero

```
Enter a number: 0
You cannot divide by zero!
Program finished.
```

Case 3: Invalid input

```
Enter a number: hello

Please enter a valid number!

Program finished.
```

Raising Exceptions in Python

Definition:

Raising an exception means **manually creating and triggering** an error when certain conditions are not met in your program.

You can use the **raise** keyword to stop program execution and display a meaningful error message.

This helps you **enforce rules** or **validate input** in your code.

Example:

```
age = int(input("Enter your age: "))


if age < 18:
   raise ValueError("Age must be at least 18 to register.")
else:
   print("Registration successful!")
```

Explanation:

- The program asks the user to input their age.
- If the entered age is less than 18, Python **raises a ValueError** with a custom message — "Age must be at least 18 to register."
- The raise statement **stops** further execution.

- If the age is 18 or above, it prints "Registration successful!"

Output Scenarios:

Case 1 – Valid Age:

```
Enter your age: 22

Registration successful!
```

Case 2 – Invalid Age:

```
Enter your age: 15

Traceback (most recent call last):

  ...

ValueError: Age must be at least 18 to register.
```

User-Defined Exceptions

Definition:

A **User-Defined Exception** is a **custom error type** created by the programmer. It allows you to define your own rules for when and how an exception should occur, using the **Exception** base class.

This is useful in large applications like **banking systems, inventory management, or file handling**, where specific error messages help in debugging and clarity.

Example: Custom Exception for Bank System

```
# Custom Exception

class InsufficientBalanceError(Exception):

    """Custom Exception for low balance"""

    pass
```

```
class BankAccount:
    bank_name = "SBI"  # class variable shared by all accounts

    def __init__(self, name, balance):
        self.name = name
        self.balance = balance

    # Instance Method - Withdraw
    def withdraw(self, amount):
        if amount > self.balance:
            raise InsufficientBalanceError("Insufficient balance to withdraw.")
        else:
            self.balance -= amount
            print(f"Withdrawal successful! Remaining balance: {self.balance}")

    # Instance Method - Deposit
    def deposit(self, amount):
        self.balance += amount
        print(f"{amount} deposited successfully! New balance: {self.balance}")

    # Class Method - Change Bank Name
    @classmethod
    def change_bank(cls, new_bank):
        cls.bank_name = new_bank
        print(f"Bank name changed to {cls.bank_name}")

    # Static Method - Display Info
    @staticmethod
    def info():
        print("Welcome to our secure banking system!")

# Using the class
try:
    # Display static info
    BankAccount.info()

    # Create object
    acc = BankAccount("Balaji", 1000)

    # Deposit money
    acc.deposit(500)

    # Withdraw more than balance to trigger exception
    acc.withdraw(2000)

except InsufficientBalanceError as e:
    print("Error:", e)

finally:
    print("Transaction completed.")
```

Explanation:

1. **Custom Exception Class (InsufficientBalanceError)**
   - Created by inheriting from Python’s built-in Exception class.
   - Used to raise a meaningful error when the balance is too low.
2. **withdraw() Method**
   - Checks if the withdrawal amount is greater than the available balance.
   - If yes → raises InsufficientBalanceError.
   - Otherwise → deducts money and prints the new balance.
3. **deposit() Method**
   - Adds the specified amount to the current balance.
4. **change_bank() (Class Method)**
   - Changes the bank name for all accounts using cls.
5. **info() (Static Method)**
   - Displays a welcome message; it doesn’t depend on class or object data.
6. **try–except–finally Block**
   - Used to handle the custom exception gracefully.
   - The finally block runs no matter what, to confirm transaction completion.

Output Example:

```
Welcome to our secure banking system!

500 deposited successfully! New balance: 1500

Error: Insufficient balance to withdraw.

Transaction completed.
```