Marwadi University	**Marwadi University** **Faculty of Engineering & Technology** **Department of Information and Communication Technology**
Subject: Programming With Python (01CT1309)	**Aim:** Practical based on OOP concept using Python
Experiment No: 14	**Date:** | **Enrollment No: 92400133055**

GITHUB

Aim: Practical based on OOP concept using Python
IDE: Visual Studio Code

Object Oriented Programming is a fundamental concept in Python, empowering developers to build modular, maintainable, and scalable applications. By understanding the core OOP principles classes, objects, inheritance, encapsulation, polymorphism, and abstraction programmers can leverage the full potential of Python's OOP capabilities to design elegant and efficient solutions to complex problems.

OOPs Concepts in Python
Class in Python
Objects in Python
Polymorphism in Python
Encapsulation in Python
Inheritance in Python
Data Abstraction in Python

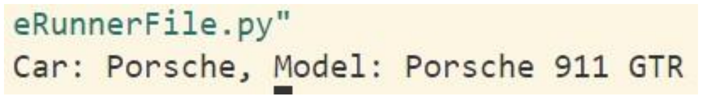
Python Class
A class is a collection of objects. A class contains the blueprints or the prototype from which the objects are being created. It is a logical entity that contains some attributes and methods.

Defining a Class
Example 1: class
Car:

Marwadi University NAAC A+	**Marwadi University** **Faculty of Engineering & Technology** **Department of Information and Communication Technology**
Subject: Programming With Python (01CT1309)	**Aim:** Practical based on OOP concept using Python

Experiment No: 14	**Date:**	**Enrollment No: 92400133055**

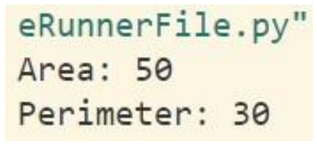
```
   # Constructor to initialize the object
def __init__(self, brand, model):
self.brand = brand  # Attribute
       self.model = model  # Attribute

   # Method to describe the car
def car_details(self):

       return f"Car: {self.brand}, Model: {self.model}"

# Creating an object of the Car class my_car = Car("Porsche", "Porsche
911 GTR")print(my_car.car_details())   Output:
```



```
eRunnerFile.py"
Car: Porsche, Model: Porsche 911 GTR
```

Example 2:
Class with Methods and Attributes

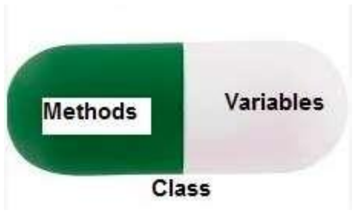
```
class Rectangle:     def
__init__(self, width, height):
self.width = width
       self.height = height

   # Method to calculate area
def area(self):        return
self.width * self.height

   # Method to calculate perimeter
def perimeter(self):        return 2 *
(self.width + self.height)

# Create an object
rect = Rectangle(10, 5)

# Accessing methods
print(f"Area: {rect.area()}")  # Output: Area: 50 print(f"Perimeter:
{rect.perimeter()}")  # Output: Perimeter: 30 Output:
```



```
eRunnerFile.py"
Area: 50
Perimeter: 30
```

Encapsulation

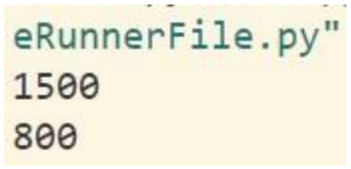
In Python object-oriented programming, Encapsulation is one of the fundamental concepts in object-oriented programming (OOP). It describes the idea of wrapping data and the methods that work on data within one unit. This puts restrictions on accessing variables and methods directly and can prevent the accidental modification of data. To prevent accidental change, an object's variable can only be changed by an object's method. Those types of variables are known as private variables.



Example 3:

```
class BankAccount:     def __init__(self,
account_holder, balance):
self.account_holder = account_holder
self.__balance = balance  # Private attribute

   def deposit(self, amount):
self.__balance += amount

   def withdraw(self, amount):
if amount <= self.__balance:
self.__balance -= amount
else:
         print("Insufficient funds")

   def get_balance(self):
return self.__balance

# Create an account
account = BankAccount("John", 1000)
account.deposit(500)
```

```
print(account.get_balance())            #
account.withdraw(700)
print(account.get_balance())  #
```
Output

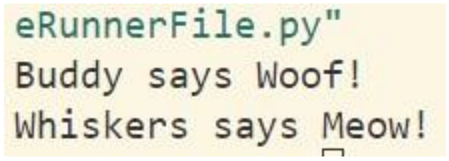


```
eRunnerFile.py"
1500
800
```

Inheritance

Inheritance allows a new class (child class) to inherit attributes and methods from an existing class (parent class). It promotes code reusability.

Example 4 class Animal:
```
def __init__(self, name):
       self.name = name

   def speak(self):
return "I am an animal."

# Dog class inherits from Animal class
class Dog(Animal):     def speak(self):
       return f"{self.name} says Woof!"

# Cat class inherits from Animal class
class Cat(Animal):     def speak(self):
       return f"{self.name} says Meow!"

dog  =  Dog("Buddy")
cat = Cat("Whiskers")
print(dog.speak())    #
print(cat.speak())  #
```
Output



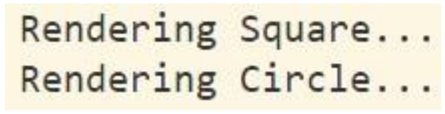
```
eRunnerFile.py"
Buddy says Woof!
Whiskers says Meow!
```

	Marwadi University **Faculty of Engineering & Technology** **Department of Information and Communication Technology**	
Subject: Programming With Python (01CT1309)	**Aim:** Practical based on OOP concept using Python	
Experiment No: 14	**Date:**	**Enrollment No: 92400133055**

Polymorphism

Polymorphism is another important concept of object-oriented programming. It simply means more than one form.
That is, the same entity (method or operator or object) can perform different operations in different scenarios.

```
Example 5:  class
Polygon:
  # method to render a shape
def                    render(self):
print("Rendering       Polygon...")
class   Square(Polygon):           #
renders Square    def render(self):
print("Rendering Square...")

class Circle(Polygon):     #
renders circle     def
render(self):
print("Rendering Circle...")

# create an object of Square s1
= Square()
s1.render()

# create an object of Circle
c1 = Circle()
c1.render() Output:
```



```
Rendering Square...
Rendering Circle...
```

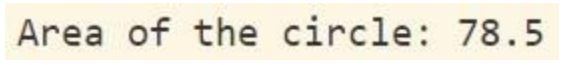
Abstraction

Abstraction focuses on hiding the internal implementation details of a class and exposing only the essential features. Example 6:

```
from abc import ABC, abstractmethod

# Abstract class class
Shape(ABC):
```

Marwadi University · NAAC A+	**Marwadi University** **Faculty of Engineering & Technology** **Department of Information and Communication Technology**	
Subject: Programming With Python (01CT1309)	**Aim:** Practical based on OOP concept using Python	
Experiment No: 14	**Date:**	**Enrollment No: 92400133055**

```
@abstractmethod
def area(self):
    pass

class Circle(Shape):    def
__init__(self, radius):
    self.radius = radius

  def area(self):        return 3.14 *
self.radius * self.radius circle = Circle(5)
print(f"Area of the circle: {circle.area()}")
#  Output:
```



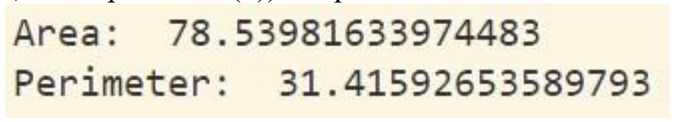
```
Area of the circle: 78.5
```

Post Lab Exercise:

Write a Python program to create a class representing a Circle. Include methods to calculate its area and perimeter.

Code:

```
import math class Circle:        def
area(radius):            return math.pi
* radius ** 2        def
perimeter(radius):
        return 2*math.pi*radius circle=Circle()
print("Area: ", Circle.area(5)) print("Perimeter:
",Circle.perimeter(5)) Output:
```

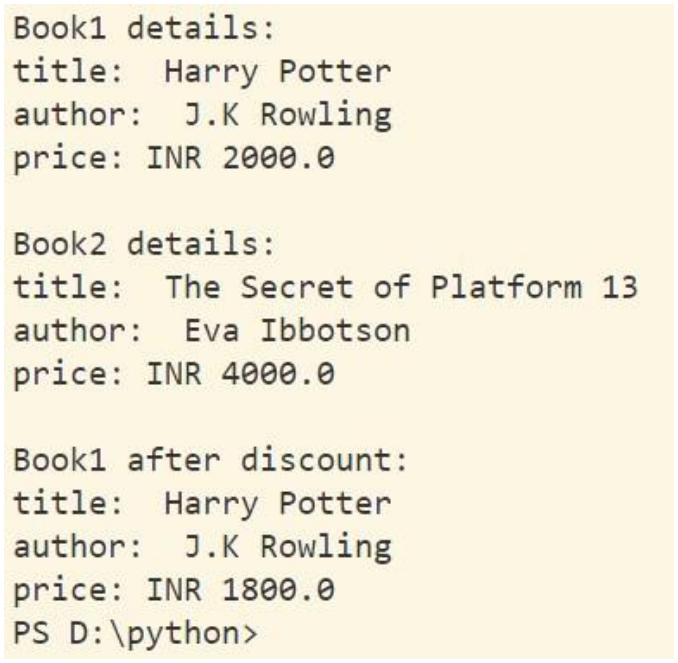


```
Area:  78.53981633974483
Perimeter:  31.41592653589793
```

Create a class `Book` that stores details like the title, author, and price of a book. Add methods to display the details of the book and apply a discount to the price. (a) Create two objects for different books and display their details. (b) Apply a 10% discount to one of the books and display the updated price. Code:

```
class Book:
   def __init__(self,title,author,price):
      self.title=title
self.author=author
self.price=price    def
display(self):
        print("title: ",self.title)
print("author: ",self.author)
print("price: INR",self.price)     def
discount(self,Discount):
self.price *= (1 - Discount / 100)

book1=Book("Harry Potter", "J.K Rowling", 2000.0)

book2=Book("The Secret of Platform 13", "Eva Ibbotson", 4000.0)
print("Book1 details: ") book1.display() print("\nBook2 details: ")
book2.display() book1.discount(10) print("\nBook1 after discount:
") book1.display()
```

Output:



```
Book1 details:
title:  Harry Potter
author:  J.K Rowling
price: INR 2000.0

Book2 details:
title:  The Secret of Platform 13
author:  Eva Ibbotson
price: INR 4000.0

Book1 after discount:
title:  Harry Potter
author:  J.K Rowling
price: INR 1800.0
PS D:\python>
```