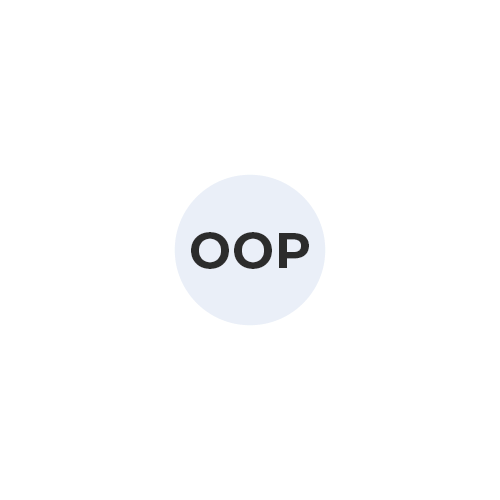
**Aim:** Practical based on OOP concept using Python

**IDE:**

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:

# 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):

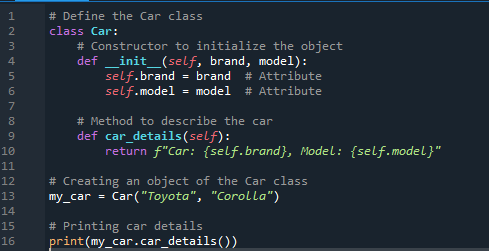
return f"Car: {self.brand}, Model: {self.model}"

# Creating an object of the Car class

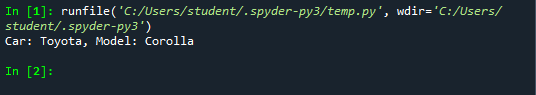
my\_car = Car("Toyota", "Corolla")

print(my\_car.car\_details())

Code:



Output:



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

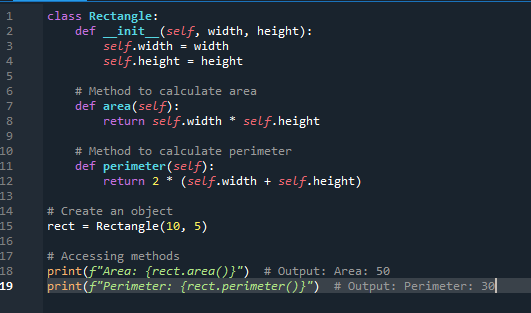
rect = Rectangle(10, 5)

# Accessing methods

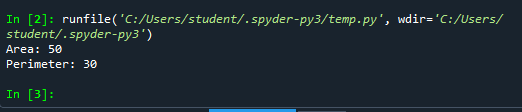
print(f"Area: {rect.area()}") # Output: Area: 50

print(f"Perimeter: {rect.perimeter()}") # Output: Perimeter: 30

Code:

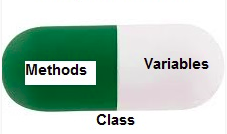


Output:



**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)

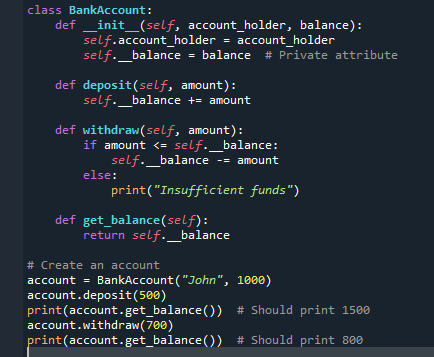
account.deposit(500)

print(account.get\_balance()) #

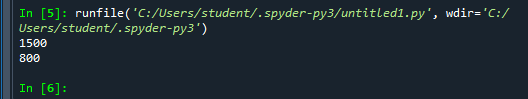
account.withdraw(700)

print(account.get\_balance()) #

Code:



Output



**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!"

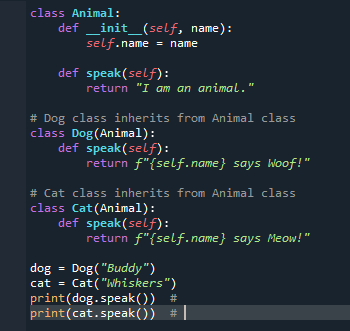
dog = Dog("Buddy")

cat = Cat("Whiskers")

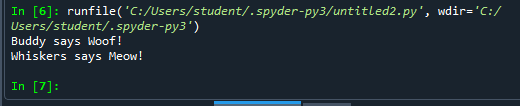
print(dog.speak()) #

print(cat.speak()) #

Code:



Output



**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()

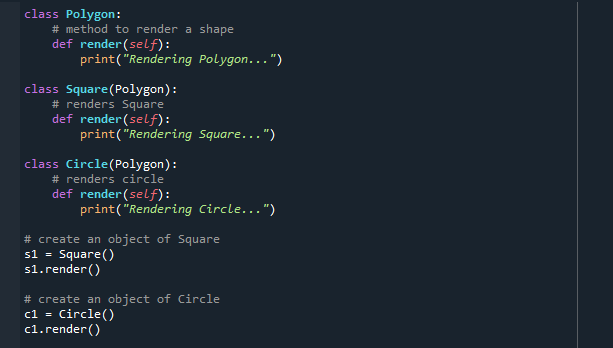
s1.render()

# create an object of Circle

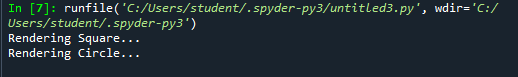
c1 = Circle()

c1.render()

Code:



Output:



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):

@abstractmethod

def area(self):

pass

class Circle(Shape):

def \_\_init\_\_(self, radius):

self.radius = radius

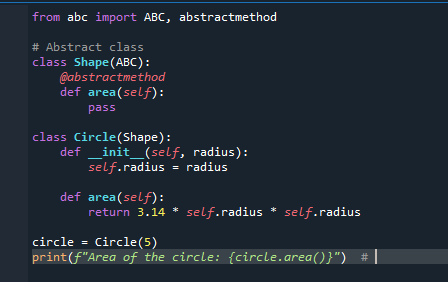
def area(self):

return 3.14 \* self.radius \* self.radius

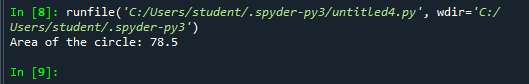
circle = Circle(5)

print(f"Area of the circle: {circle.area()}") #

Code:



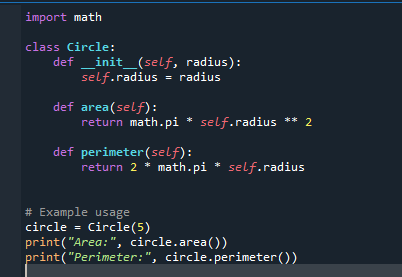
Output:



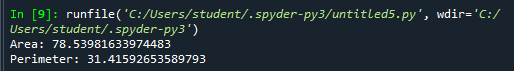
**Post Lab Exercise:**

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

Code:

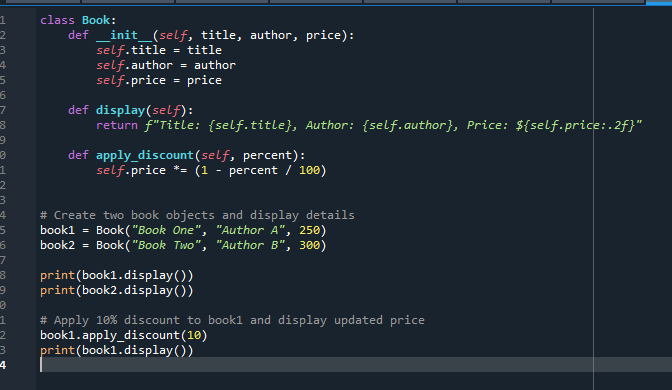


Output:

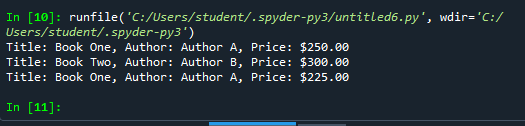


* Create a class Book that stores details like the title, author, and price of a book. Add methods to display the details of the book and apply a discount to the price. (a) Create two objects for different books and display their details. (b) Apply a 10% discount to one of the books and display the updated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

Code:



Output:



Github:

<https://github.com/hemanth-singampalli/pwp.git>