MASTER OF COMPUTER APPLICATIONS

SEMESTER 1

PYTHON PROGRAMMING

Unit 8

Object Oriented Programming

Table of Contents

1. INTRODUCTION

Welcome to Unit 8, where we dive into the world of Object-Oriented Programming (OOP) in Python, focusing on Classes and Objects, two fundamental pillars of OOP. In the previous unit, Unit 7, we took a closer look at Exception Handling. You learned about common error types, the importance of try, except, and finally blocks for robust error handling, and how to raise custom exceptions for better control flow in your programs. This knowledge is crucial for writing resilient and fault-tolerant Python code, ensuring your applications can handle unexpected situations gracefully.

Now, in Unit 8, we shift gears to explore how Python implements OOP principles, allowing us to model real-world problems effectively. This unit will introduce you to Classes, the blueprints for creating Objects, which are instances of these Classes. You'll learn how to define your own Classes, create Objects from these Classes, and understand concepts like Object References and Object Identity, which are pivotal in understanding how Python manages memory and object lifecycle. Moreover, we'll delve into Creating Methods within Classes, enabling you to encapsulate functionality specific to an Object's behavior, making your code modular and reusable.

Embarking on this journey requires a mindset shift from procedural to object-oriented programming. Start by familiarizing yourself with the syntax and semantics of Class definitions in Python. Practice creating simple Classes and Objects to get a feel for how they work. Experiment with adding methods to your Classes and observe how Objects of those Classes use these methods. Pay attention to Object References and how Python assigns memory addresses to Objects, as understanding this will significantly improve your debugging skills. Engage with the provided examples, modify them, and try creating your own scenarios where you can apply these concepts. Remember, the best way to learn is by doing, so don't hesitate to write plenty of code as you work through this unit.

1.1 Learning Objectives

After studying this unit, you will be able to:

- *Identify the basic concepts of Classes and Objects in Python.*
- *Explain the relationship between a Class and an Object.*
- *Construct Classes in Python and instantiate Objects from them.*

- *Differentiate between Object References and Object Identities.*
- *Critique the use of different types of Methods within a Class.*
- *Design a Python program using Encapsulation and Polymorphism principles.*

2. CLASS IN PYTHON

To understand the term class, take an analogy like cats, jaguars, tigers, cheetah, and more. All these animals come under the cat family. On the other hand, dogs, wolves, foxes, and such come under the dog family. You can consider them subclasses that come under the larger class of animals. It is the concept of classes that are considered in Python as well.

In Python, a class is defined as an object constructor. It acts as a blueprint to create objects. Class in Python is different from what we understand from it using real-life situations because its definition also integrates the supported methods. All instances, or objects, of a class follow a method. Thus, all the subclasses of that class will follow the method as well.

2.1. Creation of Class in Python

To create a class in Python, you will use the class keyword The name of the class immediately follows the statement. An example is shown below.

Example:

```
class ClassName:
    a = 6
```

- Here, a is an attribute of the class. Note that the class's name starts with a capital letter, as is the convention. The class contains a documentation string that is practical to obtain through ClassName._doc_.
- This documentation string, also known as docstring, is the first string inside the class. It has a small details of the class, and including a docstring is not necessary. However, it is advised to add it.

After class creation, an associated new class object is created quickly. This class object has the same name as the class. With the class object's help, you can access different class attributes and instantiate new objects. As you can see, the class does not inherit anything from any other class. Later in this chapter, we will study the inheritance property.

If you create a class with no attribute, the interpreter will give an error. Thus, you either have to put an attribute inside the class or write the keyword “pass”. This keyword tells the interpreter that it can move on to the next command line.

Also, note that everything inside the class is indented. In Python, indentation is an essential part of the syntax. Being indented means that the command or code is associated with the class. In the future, you will notice, indentation is used in the case of "if", "for", and other such cases.

3. OBJECT IN PYTHON

We have covered the fact that everything in Python is treated as an object. Variables, functions, lists, tuples, dictionaries, sets, etc., every entity defined in Python becomes objects. When you define an integer, it belongs to an integer class, just like a lion belongs to the animal class. Every object in Python has its own class. In a class, you define data structures that can hold data members. These data members, such as variables and constructs, are accessed by an object of the class.

In order to check the type of user-defined classes and all objects, the function isinstance(object, type) is used.

3.1. How to create an object in Python

The method of constructing an object in a class is referred instantiation, and an object is called an instance. The syntax used to define an object inside a class is given below.

ObjectName = ClassName (<required arguements>)

Example Program: Here is a code that displays the name and grade of the student.

```
class Student:
    name = "Ishan"
    grade = 8
stu1 = Student()
print(stu1.name)
print(stu1.grade)
```

The output of the program will be as follows:

```
Ishan
8
```

Here, the object is stu1, and with the dot operator, we accessed the student's name and grade.

4. OBJECT REFERENCES

Imagine an arrow. It is your object, your variable name that you created in the memory. The arrow, once released, will find a place to lodge, which will be its memory location. In Python's terms, whenever you proceed with the variable assignment, that is, assigning a value to a variable, what you are doing is forming an equation. You create the left side of the equation by declaring a variable. This variable gets saved in the memory with the address, say, 1800. Thus, the address 1800 refers to the variables that you created. In Python, everything, including the variable, is an object; therefore, this 1800 becomes your object reference that the variable's name will represent.

The object reference is the indication of the memory location of the object placement. Take an example below to understand the concept deeper.

Example:

```
a = 5
b = 6
c = 8
a = b
b = c
print (a, b)
```

The output of the program will be:

```
6 8
```

The value of a, as assigned to the user, should be 5. Similarly, the value of b should be 6. However, when you wrote 'a = b' and 'b = c', what you essentially did is commanded that a is b, and b is c. Thus, the location of variable 'a' became the location of variable 'b', and the location of 'c' became that of 'b'. Changing the object reference also changed the value stored in that variable.

An object in Python exists as long as it has an object reference. If the number of object references drops to one, then that object is no longer accessible. Thus, we say that the lifetime of the object that began with its creation ended. In the future, we can store other data in this empty memory.

5. OBJECT IDENTITY

Every object in Python has a unique identity. The interpreter ensures that no two objects have the same identity if they both have overlapping lifetimes. When there is no object reference for an object, only then its identity becomes available to be assigned to any other object. Python has a built-in function, id(), to return an object's identity in the form of an integer. This integer represents the location in the memory where the object has been stored.

Example:

```
n = 60
id(n)
```

Output:

```
7811108
```

[This ID will be different for everyone, as it is system dependant]

6. CREATING METHODS IN PYTHON

Before you begin to create Python methods, you will first have to grasp the concept of function. A function is a set of commands or parameters which are closed under a parenthesis. They perform a specific action and return a value. Functions are great tools to simplify and shorten the length of the code. If you find that you have to perform an action repeatedly in a program, then instead of writing the code again and again, all you have to do is write it once in a function. Now, whenever you need to perform a function, you will have to call the function. It will return the value so that your code works impeccably and has a few codes as possible.

Python has various built-in functions that perform the functions that are commonly used across various programs. Take print () for an example. It displays the result that you want to print on the output screen. How does it do it? The commands required to display a result have already been coded in the print () function. Hence, a user does not have to write it again in every program. Similarly, functions like len() and many others are available in the library of Python.

Users can also define and create functions in Python. You will have to use the keywords "def" to define a function. You can add parameters and commands inside the function and then call the function anywhere in the program.

Example: Here is an example of a function defined in Python.

```
def hello (name):
    return "Hello," + name
```

The code creates a function named "hello". It prints the string "hello" with the name entered by the user. Notice that the second line of command is indented to tell the interpreter that it is associated with the function. You can call this function in the program, as is shown below.

```
print (hello ("World"))
```

The output of this will be:

Hello, World.

Now that you have understood the concept of functions let's move on to methods. You must have come across the word methods in the previous units, especially while defining various

data types such as lists, sets, tuples, dictionaries, etc. For example, by using the method append (), to the end of a list you can append an object. Or using the method remove () deletes an object from the sequence data types. As you have already used methods in programs, you must have a vague idea about the term's meaning.

A method is a process through which you represent a class's behaviour the objects associated with the class. In Python, a method works similarly as a function. However, a method must be called on an object.

You can refer to previous units to get the list of methods along with their description in Python.

6.1. How to Create a Method?

A method needs to be created inside a class. The example provided below explains the syntax needed to be used to create a method.

Example:

```
class Hello:
    def __init__(self):
        pass
    def printhello(self, name):
        print(f"Hello, {name}")
        return name
myname = Hello()
myname.printhello('Ayush')
```

Here, the method defined is printhello (). It has parameters and returns a value.

What is __init__ ()? If you are familiar with other object-oriented languages such as C++, you might know that a special function is used to initiate the class attributes. This function is called constructor and usually has the same name as the class. In Python, the method __init__ is used for the same purpose. The parameter in this is self, as it calls to the class itself. The keyword “pass” is used as the method cannot be kept empty.

7. ENCAPSULATION

Through encapsulation, you can wrap data and methods in one unit. Thus, other users cannot access the elements of the unit or change them. The method is beneficial to avoid any accidental changes in the data. Encapsulation allows change in an object's variable only through the object's method. Such variables are called private variables.

Encapsulation is easily exemplified by class. It encases all of the elements as well as the data. such as member functions, variables, etc., and does not allow any changes.

7.1. Protected members

The members of a class are not available to access from the exterior of the class. Accessibility is permitted within the class or its subclasses only. In Python, a data member is protected by prefixing it with a single underscore "_".

7.2. Private members

Private members are secured, but these cannot be accessed even by the super or parent class's subclasses, and a double underscore "__" is used as a prefix to inform the interpreter that a data member is private. Note, remember that the init() method is preceded and followed by a double underscore.

Here is a sample code that signifies protected and private members in a Python program.

Sample code:

```
class MyClass:
    def __init__(self):
        self.a = "This is my class"
        self._b = "This is a protected member"
        self.__c = "This is a private member"

class ChildClass(MyClass):
    def __init__(self):
        super().__init__()
        print(self._b)
 obj1 = MyClass()
print(obj1.a)
```

Output:

This is my class

8. POLYMORPHISM

Assume a situation where you have to calculate the price of an item. The price is stored in the form of a list, a tuple, or even a dictionary. Depending upon the type of instance, you will have to put conditions in your program to retrieve the value.

However, is it always possible to know the kind of data type used? What will happen if the data type is changed for some reason? You will have to change the entire code as well. Changing the code, again and again, is neither productive nor uses the intensive features that Python provides

You can let the object decide the operation and carry it out by itself in such a situation. The object can retrieve its value on its own. You will not have to make any corrections or change the data type again. It is possible through polymorphism.

Polymorphism means the occurrence of a state in different forms. Through polymorphism, you can express an entity in various forms and scenarios. Take the example of + sign in Python. for integers, it is used to perform arithmetic addition. When used between two strings, it performs concatenation. Thus, + operator in Python can be used in different forms, depending upon the syntax.

Similarly, functions like len(), del, etc., can be used with various data types. When using the function len(), you do not know whether the data type whose length you can calculate is an integer or a string. The function will return the length of the data type, regardless of its nature. Here is an example.

Example:

```
print (len ("Python"))
print (len ([9, 10, 11]))
```

The output of the program will be:

```
6
3
```

When it comes to classes, polymorphism comes especially handy as, through it, Python permits the usage of methods of the same name across various classes in a program. You can

also define polymorphic functions. The functions will perform the action for which the commands are provided without knowing the object's data type or other attributes.

Example:

```
def add (x, y, z = 0):
   return x + y + z
print (add (2, 3))
print (add (4, 5, 6))
```

Output is shown below:

```
5
15
```

Similarly, through polymorphism, you can apply two distinct class types in the same manner. Another essential function of polymorphism is overriding. When a child class obtains a parent class's properties through inheritance, it is practical to alter a method in the inherited child class. The process involves the reimplementation of the method in child class. This method is called method overriding. All methods of the parent class, except private methods, can be overridden through polymorphism. However, the opposite of this is not possible. You cannot override a method in the subclass through the super or parent class.

9. INHERITANCE

Inheritance is another essential property of object-oriented programming. Through this process, one class can acquire the properties of a different class. The process ensures that a code is reused so that it decreases redundancy. Through inheritance, you can effectively recreate various real-life situations where one class has similar base class properties. Through inheritance, you can make changes in the base class's properties in the child class without the base class alteration.

```
class Employee:
    def __init__(self, name):
        self.name = name
        self.ID = None

    def getName(self):
        return self.name

    def getID(self):
        return self.ID

    def checkemployee(self):
        return False

class Department(Employee):
    def checkemployee(self):
        return True

# Creating an instance of Employee
emp = Employee("Ayush")
emp.ID = "145"  # Setting ID after creation

# Print statements to verify the outputs
print(emp.getName(), emp.getID(), emp.checkemployee())
```

For the child class to identify its parent class, it needs to mention the parent class in its definition. This method is subclassing or calling the constructor of the parent class. The syntax followed for this is as follows.

class subclass_name (superclass_name)

9.1. Types of Inheritance

The types of inheritance are divided based on the number of child and parent classes.

Single Inheritance: The attributes of a single parent class are passed on to the child class.

Example:

```
class Animal:  # Parent Class
  def __init__(self, name):
    self.name = name

  def speak(self):
    pass

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

# Creating an instance of Dog
dog = Dog("Buddy")
print(dog.speak())
```

Output:

Buddy says Woof!

Multiple Inheritance: When a derived class gets its properties to form more than one superclass, it is called multiple inheritances. The subclass derives all the properties of the parent classes.

Example:

```
class Father:
  def skills(self):
    print("Gardening, Programming")
```

```
class Mother:
    def skills(self):
        print("Cooking, Art")

class Child(Father, Mother):  # Child class inheriting from both Father and Mother
    def skills(self):
        Father.skills(self)
        Mother.skills(self)
        print("Sports")

# Creating an instance of Child
child = Child()
child.skills()
```

Output:

```
Gardening, Programming
Cooking, Art
Sports
```

Multilevel Inheritance: When another generation of classes further inherits the parent and child class features, it is classed as a multilevel inheritance.

Example:

```
class Grandfather:
    def height(self):
        print("Height is 6 feet")

class Father(Grandfather):  # Inherits from Grandfather
    def personality(self):
        print("Great personality")

class Son(Father):  # Inherits from Father
    def skills(self):
        print("Excellent in studies")

# Creating an instance of Son
```

```
son = Son()
son.height()  # From Grandfather
son.personality()  # From Father
son.skills()  # From Son itself
```

Hierarchical Inheritance: When more than one derived class receive the properties from one individual parent class, then it is the case of hierarchical inheritance.

Example:

```
class Vehicle:
    def general_usage(self):
        print("General use: Transportation")

class Car(Vehicle):  # Inherits from Vehicle
    def specific_usage(self):
        print("Specific use: Drive to work")

class Truck(Vehicle):  # Inherits from Vehicle
    def specific_usage(self):
        print("Specific use: Transport goods")

car = Car()
truck = Truck()

car.general_usage()  # From Vehicle
car.specific_usage()  # From Car

truck.general_usage()  # From Vehicle
truck.specific_usage()  # From Truck
```

Hybrid Inheritance: Hybrid inheritance occurs when various types of inheritances are used together.

Example:

```
class School:
    def function1(self):
        print("This function is in school.")
```

```
class Teacher1(School):
    def function2(self):
        print("This function is in teacher 1.")

class Teacher2(School):
    def function3(self):
        print("This function is in teacher 2.")

class Student(Teacher1, Teacher2):
    def function4(self):
        print("This function is in student.")

# Creating an instance of Student
student = Student()
student.function1()  # From School
student.function2()  # From Teacher1
student.function3()  # From Teacher2
student.function4()  # From Student itself
```

10. SUMMARY

As we wrap up our exploration into the world of Python's object-oriented programming, we've delved deep into the constructs that make Python such a versatile and powerful language. From the foundational concepts of classes and objects to the advanced principles of encapsulation and polymorphism, our journey has been both enlightening and practical.

Let's take a moment to reflect on what we've learned. We started by understanding that classes serve as blueprints for creating objects. Each object, then, is an instance of a class, equipped with attributes and methods defined within its class. This relationship between classes and objects is fundamental to grasping Python's approach to object-oriented programming. By creating classes, we set the stage for objects to come into existence, each with its own unique properties and behaviors.

We then moved on to the concept of object references and identities. Remember our analogy of the arrow? Just as an arrow finds a place to lodge, a variable in Python points to a location in memory where the object resides. This concept is crucial for managing and manipulating data within our programs effectively.

We didn't stop there. We delved into creating methods within classes, enabling our objects to perform actions. Whether it's a simple method to greet a user or a more complex method that calculates the area of a shape, methods breathe life into our classes, making them functional and dynamic.

Encapsulation and polymorphism were our next stops. Encapsulation allows us to hide and protect the data within our classes, ensuring that only the most relevant information is exposed through well-defined interfaces. Polymorphism, on the other hand, gifted us with the flexibility to use a single interface for different data types. We saw how polymorphism enables us to streamline our code, making it more adaptable and easier to maintain.

Inheritance, a cornerstone of object-oriented programming, was another highlight of our journey. We explored various types of inheritance, from single and multiple to multilevel, hierarchical, and hybrid. Each type serves a unique purpose, allowing us to build complex relationships between classes and leverage existing code to create new functionalities.

As we conclude this unit, I hope you feel empowered by the knowledge and skills you've acquired. Object-oriented programming in Python opens up a world of possibilities, enabling you to tackle complex problems with elegant and efficient solutions. Keep experimenting, keep coding, and most importantly, keep learning. The journey of mastering Python is ongoing, and every step forward is a step towards becoming a more proficient and confident programmer.

11. GLOSSARY

Class: A blueprint for creating objects, defining a set of attributes and methods that characterize any object of the class.

Object: An instance of a class containing real values, not just the layout defined by the class.

Method: A function defined inside a class and used to define the behaviors of an object.

Encapsulation: The bundling of data with the methods that operate on that data, restricting direct access to some of the object's components.

Polymorphism: The ability of different object types to be accessed through the same interface, highlighting the capability to appear in many forms.

Inheritance: A mechanism where a new class is derived from an existing class, inheriting attributes and methods from the base class.

Single Inheritance: A derived class inherits properties from a single parent class.

Multiple Inheritance: A derived class inherits properties from more than one parent class.

Multilevel Inheritance: A form of inheritance where a class is derived from a derived class, creating a "grandchild" class.

Hierarchical Inheritance: Multiple derived classes inherit properties from a single base class.

Hybrid Inheritance: A combination of two or more types of inheritance used in a single program.

Object References: The association of a variable with an object in memory, allowing the variable to access the object's attributes and methods.

Object Identity: A unique identifier for an object, typically represented by its memory address in Python, which can be obtained using the id() function.

init Method: A special method in Python classes, called a constructor, that initializes new objects, setting their initial state.

Protected Members: Attributes or methods in a class that are only accessible within the class and its subclasses, typically denoted by a single underscore prefix (e.g., _protectedAttribute).

Private Members: Attributes or methods in a class that are accessible only within the class and not by its instances or subclasses, denoted by a double underscore prefix (e.g., __privateAttribute).

12. SELF-ASSESSMENT QUESTIONS

1. In Python, a _____ is defined as an object constructor, or a "blueprint" for creating objects.
2. An instance of a class is known as an _____.
3. _____ is the process of wrapping data and functions into a single unit in a class.
4. The method _____ is used in Python classes to initialize new objects.
5. _____ inheritance occurs when a class inherits from more than one base class.
6. Which of the following is NOT a type of inheritance in Python?
   A. Single Inheritance
   B. Double Inheritance
   C. Multilevel Inheritance
   D. Hierarchical Inheritance
7. Which special method in Python classes acts as a constructor?
   A. __init__()
   B. __construct__()
   C. __new__()
   D. __class__()
8. Encapsulation in Python can be implemented by using:
   A. Public members only
   B. Private members using double underscores
   C. Protected members using a single underscore
   D. Both B and C
9. Polymorphism allows:
   A. Classes to inherit methods from multiple base classes
   B. Methods to use objects of different classes
   C. Functions to perform differently based on the input
   D. Objects to change their class after creation
10. Which method is automatically called when an object is created?
    A. __auto__()
    B. __call__()
    C. __init__()

D. __start__()

11. The identity of an object can be obtained using the _____ function.
    A. type()
    B. id()
    C. class()
    D. object()
12. A class variable is accessible:
    A. Only within the class it is declared
    B. From outside the class but not by instances
    C. By instances of the class and by the class itself
    D. Only by the base class
13. Which of the following best defines method overriding?
    A. Changing the behavior of a base class method in the derived class
    B. Using multiple methods with the same name within a class
    C. Overloading a method with different parameters
    D. Renaming a method in a derived class
14. Hybrid inheritance is a combination of:
    A. Only single and multiple inheritances
    B. Single, multiple, and multilevel inheritances
    C. Any two types of inheritance
    D. All types of inheritance
15. In Python, private members of a class are prefixed with:
    A. A single underscore _
    B. A double underscore __
    C. A triple underscore ___
    D. No underscore

13. TERMINAL QUESTIONS

1. Explain the concept of classes in Python and how they act as a blueprint for creating objects.
2. Describe the process and significance of creating objects in Python classes.
3. What is encapsulation in Python, and how does it improve data security within a class?
4. Discuss the __init__ method in Python classes and its role in object initialization.
5. Compare and contrast single inheritance and multiple inheritance with examples.
6. Create a Python class named Book with attributes title and author. Initialize these attributes with the __init__ method and create a method to display the book details.
7. Define a class Circle with a private attribute radius. Provide a method to set the radius value and another method to calculate and return the area of the circle.
8. Implement a Python program demonstrating multilevel inheritance with a base class Animal, a derived class Bird, and another level derived class Sparrow.
9. Write a Python program to demonstrate hierarchical inheritance where a base class Vehicle is inherited by two derived classes Car and Truck, each with a specific method defining their use.
10. Create a Python class that uses encapsulation to hide its attributes and provides methods to get and set the values.
11. Design a Python program that demonstrates the concept of polymorphism using a common method in two different classes.
12. Implement a class Calculator that demonstrates method overriding by inheriting from a BasicCalculator class and overriding a method to add functionality.
13. Create a Python class Rectangle with methods to calculate the area and perimeter. Then, create a derived class Square that overrides the methods to calculate the area and perimeter of a square.
14. Write a program that demonstrates the use of the id() function by creating two different objects and displaying their identities.
15. Implement a class method in Python that takes multiple arguments and demonstrates method overloading.
16. Design a Python class Person with a private attribute age and implement a method to set the age only if it is within a specific range (e.g., 1-100).

17. Write a Python program to demonstrate the use of hybrid inheritance with a practical example involving at least three different types of inheritance.
18. Create a Python program that showcases the use of the isinstance() function with user-defined classes and objects.
19. Explain with a code example how to create and use protected members in a Python class.
20. Demonstrate with a Python program how a child class can modify the behavior of a parent class method (method overriding).

14. ANSWERS

14.1. Self-Assessment Questions

1. Class
2. Object
3. Encapsulation
4. __init__
5. Multiple
6. B) Double Inheritance
7. A) __init__()
8. D) Both B and C
9. C) Functions to perform differently based on the input
10. C) __init__()
11. B) id()
12. C) By instances of the class and by the class itself
13. A) Changing the behavior of a base class method in the derived class
14. D) All types of inheritance
15. B) A double underscore __

14.2. Terminal Questions

1. Refer to the "Class in Python" section to understand the concept and definition of classes in Python.
2. The "Object in Python" section provides insights on how objects are created and used within classes.
3. Encapsulation is detailed in the "Encapsulation" subsection, explaining how data hiding works in Python.
4. The __init__ method is explained under "Creating Methods in Python," focusing on its role in initializing objects.
5. Single and multiple inheritances are discussed in the "Inheritance" section, with clear examples.
6. Create a Book class with title and author attributes. See "Class in Python" for guidance.

7. Implement encapsulation in the Circle class. Refer to "Encapsulation" for private attribute handling.
8. For multilevel inheritance, extend the Animal class to Bird, then to Sparrow. See "Inheritance."
9. Use hierarchical inheritance to derive Car and Truck from Vehicle, each with unique methods. Check "Inheritance."
10. Demonstrate encapsulation with get and set methods in a class. Look into "Encapsulation."
11. Show polymorphism with a shared method in two classes. Refer to "Polymorphism."
12. Override a method in Calculator from BasicCalculator. See "Inheritance" for overriding methods.
13. Use method overriding in Square derived from Rectangle to calculate area and perimeter. "Inheritance" has examples.
14. Utilize the id() function to show object identities. Check "Object Identity."
15. Illustrate method overloading with a class method accepting variable arguments. "Creating Methods in Python" can guide you.
16. Use encapsulation to set age in Person if it's in range. Look into "Encapsulation."
17. Demonstrate hybrid inheritance with a complex inheritance structure. Refer to "Implementation of Inheritance."
18. Apply isinstance() to check object types against user-defined classes. See "Object in Python."
19. Create protected members in a class and access them in a subclass. Check "Encapsulation."
20. Override a parent class method in a child class to alter its behavior. "Implementation of Inheritance" provides insights.