Batch: P5-3 **Roll No.: 16010422185**

Experiment / assignment / tutorial No. 9

Grade: AA / AB / BB / BC / CC / CD /DD

Signature of the Staff In-charge with date

TITLE: Inheritance, Polymorphism and Abstract Class in Python

AIM: **1.** Write a program to implement inheritance to display information of bank account.
2. Write a program to implement polymorphism to display vehicle information

Expected OUTCOME of Experiment:
CO1: Use basic data structures in Python
CO2: Use different Decision Making statements and Functions in Python.
CO3: Apply Object oriented programming concepts in Python
Resource Needed: Python IDE

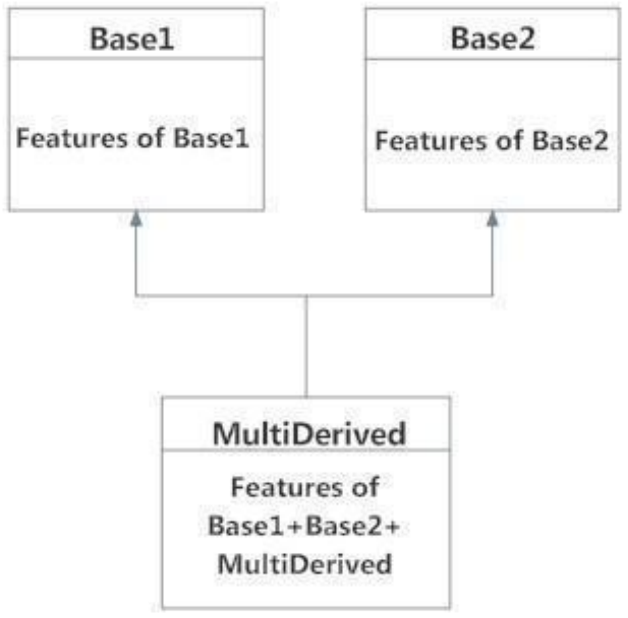
Theory:
Inheritance:
Inheritance is the capability of one class to derive or inherit the properties from some another class. The benefits of inheritance are:
1. It represents real-world relationships well.
2. It provides reusability of a code. We don’t have to write the same code again and again. Also, it allows us to add more features to a class without modifying it.
3. It is transitive in nature, which means that if class B inherits from another class A, then all the subclasses of B would automatically inherit from class A.
Syntax:
```
class Person(object):
   # Constructor
   def __init__(self, name):
      self.name = name
   # Inherited or Sub class (Note Person in bracket)
   class Employee(Person):
   # Here we return true
   def isEmployee(self):
      return True
```

Different forms of Inheritance:

1. Single inheritance: When a child class inherits from only one parent class, it is called as single inheritance. We saw an example above.

2. Multiple inheritance: When a child class inherits from multiple parent classes, it is called as multiple inheritance.

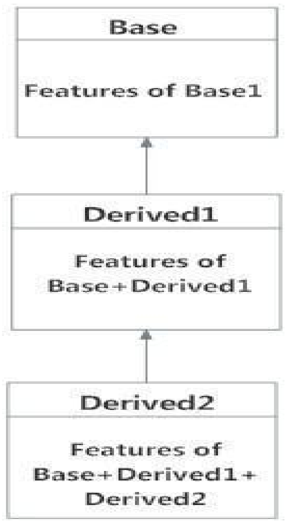
```
class Base1(object):
 ....
class Base2(object):
 ....

class Derived(Base1, Base2):
 ....
```



Multiple Inheritance in Python

3. **Multilevel inheritance**: When we have child and grand child relationship.

```
class Person(object):
     . . .
# Inherited or Sub class (Note Person in bracket)
class Child(Base):
    . . .
# Inherited or Sub class (Note Child in bracket)
class GrandChild(Child):
   ....
```


Base
Features of Base1
Derived1
Features of Base+Derived1
Derived2
Features of Base+Derived1+ Derived2

Multilevel Inheritance

Private members of parent class:

Python doesn't have any mechanism that effectively restricts access to any instance variable or method. Python prescribes a convention of prefixing the name of the variable/method with single or double underscore to emulate the behaviour of protected and private access specifiers.

We don’t always want the instance variables of the parent class to be inherited by the child class i.e. we can make some of the instance variables of the parent class private, which won’t be available to the child class.

All members in a Python class are public by default. Any member can be accessed from outside the class environment.

Example: Public Attributes

```
class employee:
   def __init__(self, name, sal):
      self.name=name
      self.salary=sal
e1= employee(1000)
```

print(e1.salary)

Python's convention to make an instance variable protected is to add a prefix _ (single underscore) to it. This effectively prevents it to be accessed, unless it is from within a sub-class. This doesn't prevent instance variables from accessing or modifying the instance

Example: Protected Attributes

```
class employee:
    def __init__(self, name, sal):
        self._name=name # protected attribute
        self._salary=sal # protected attribute
```

A double underscore __ prefixed to a variable makes it private. It gives a strong suggestion not to touch it from outside the class. Any attempt to do so will result in an AttributeError:

Example: Private Attributes

```
class employee:
    def __init__(self, name, sal):
        self.__name=name # private attribute
        self.__salary=sal # private attribute
```

Python performs name mangling of private variables. Every member with double underscore will be changed to _object._class__variable. If so required, it can still be accessed from outside the class, but the practice should be refrained.

```
e1=Employee("Bill",10000)
print(e1._Employee__salary)
e1._Employee__salary=20000
print(e1._Employee__salary)
```

super() method and method resolution order(MRO)

In Python, super() built-in has two major use cases:

 Allows us to avoid using base class explicitly
 Working with Multiple Inheritance

super() with Single Inheritance:

In case of single inheritance, it allows us to refer base class by super().

```
class Mammal(object):
  def __init__(self, mammalName):
    print(mammalName, 'is a warm-blooded animal.')

class Dog(Mammal):
  def __init__(self):
    print('Dog has four legs.')
    super().__init__('Dog') # instead of Mammal.__init__(self, 'Dog')

d1 = Dog()
```

The super() builtin returns a proxy object, a substitute object that has ability to call method of the base class via delegation. This is called indirection (ability to reference base object with super())

Since the indirection is computed at the runtime, we can use point to different base class at different time (if we need to).

Polymorphism:

Polymorphism is taken from the Greek words Poly (many) and morphism (forms). It means that the same function name can be used for different types. This makes programming more intuitive and easier.

Polymorphism in Python:

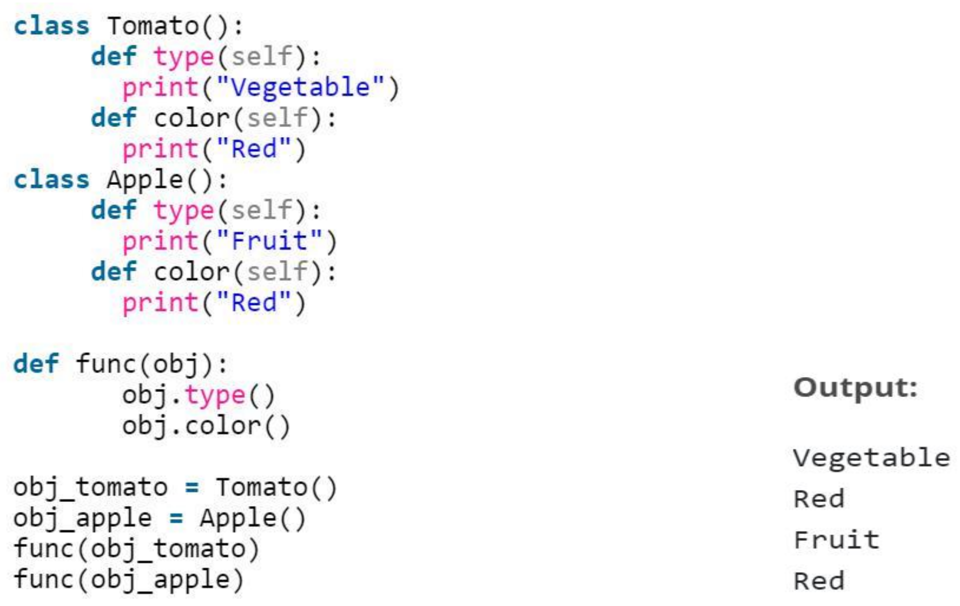
A child class inherits all the methods from the parent class. However, in some situations, the method inherited from the parent class doesn't quite fit into the child class. In such cases, you will have to re-implement method in the child class.
There are different methods to use polymorphism in Python. You can use different function, class methods or objects to define polymorphism.

Polymorphism with Function and Objects:

You can create a function that can take any object, allowing for polymorphism.

Example:

Create a function called “func()” which will take an object which we will name “obj”. and let give the function something to do that uses the ‘obj’ object which is passed to it. Now, call the methods type() and color(), each of which is defined in the two classes ‘Tomato’ and ‘Apple’ by creating instances of both the ‘Tomato’ and ‘Apple’ classes if they do not exist:



```
class Tomato():
     def type(self):
       print("Vegetable")
     def color(self):
       print("Red")
class Apple():
     def type(self):
       print("Fruit")
     def color(self):
       print("Red")

def func(obj):
       obj.type()
       obj.color()

obj_tomato = Tomato()
obj_apple = Apple()
func(obj_tomato)
func(obj_apple)
```

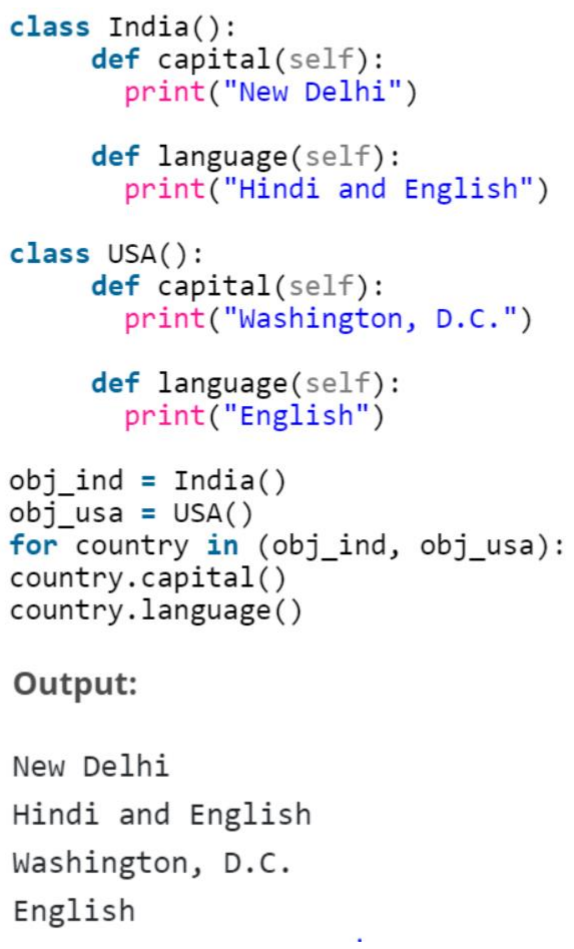
Output:

```
Vegetable
Red
Fruit
Red
```

<u>Polymorphism with Class Methods:</u>

Python uses two different class types in the same way. Here, you have to create a for loop that iterates through a tuple of objects. Next, you have to call the methods without being concerned about which class type each object is. We assume that these methods actually exist in each class

<u>Example:</u>



```
class India():
     def capital(self):
       print("New Delhi")

     def language(self):
       print("Hindi and English")

class USA():
     def capital(self):
       print("Washington, D.C.")

     def language(self):
       print("English")

obj_ind = India()
obj_usa = USA()
for country in (obj_ind, obj_usa):
country.capital()
country.language()
```

Output:

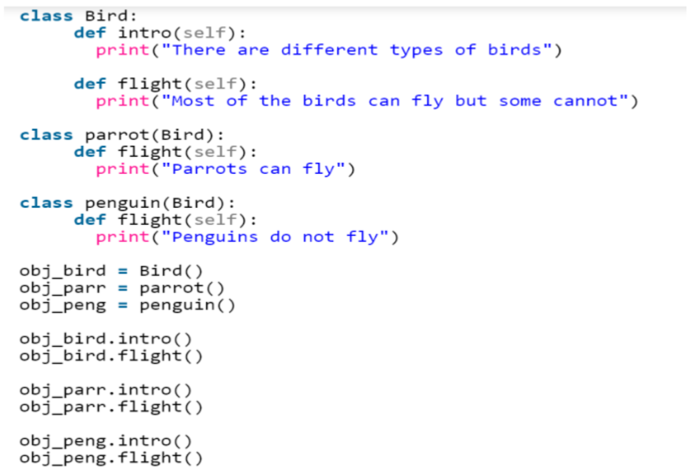
```
New Delhi
Hindi and English
Washington, D.C.
English
```

Polymorphism with Inheritance:

Polymorphism in python defines methods in the child class that have the same name as the methods in the parent class. In inheritance, the child class inherits the methods from the parent class. Also, it is possible to modify a method in a child class that it has inherited from the parent class.

This is mostly used in cases where the method inherited from the parent class doesn't fit the child class. This process of re-implementing a method in the child class is known as Method Overriding.

Example:



```
class Bird:
     def intro(self):
       print("There are different types of birds")

     def flight(self):
       print("Most of the birds can fly but some cannot")

class parrot(Bird):
     def flight(self):
       print("Parrots can fly")

class penguin(Bird):
     def flight(self):
       print("Penguins do not fly")

obj_bird = Bird()
obj_parr = parrot()
obj_peng = penguin()

obj_bird.intro()
obj_bird.flight()

obj_parr.intro()
obj_parr.flight()

obj_peng.intro()
obj_peng.flight()
```

Output:

```
There are different types of birds
Most of the birds can fly but some cannot
There are different types of bird
Parrots can fly
There are many types of birds
Penguins do not fly
```

Abstract Class:

Abstract Class is concept of object-oriented programming based on DRY (Don't Repeat Yourself) principle. In a large project, code duplication is approximately equal to bug reuse and one developer is impossible to remember all classes' details. Therefore, it's very helpful to use an abstract class to define a common interface for different implementations.

An abstract class has some features, as follows:-

- An abstract class doesn't contain all of the method implementations required to work completely, which means it contains one or more abstract methods. An abstract method is a method that just has a declaration but does not have a detail implementation.
- An abstract class cannot be instantiated. It just provides an interface for subclasses to avoid code duplication. It makes no sense to instantiate an abstract class.
- A derived subclass must implement the abstract methods to create a concrete class that fits the interface defined by the abstract class. Therefore it cannot be instantiated unless all of its abstract methods are overridden.

<u>Define Abstract Class in Python:</u>

Python comes with a module called abc which provides methods for abstract class.

Define a class as an abstract class by abc.ABC and define a method as an abstract method by abc.abstractmethod. ABC is the abbreviation of abstract base class.

Example:

```
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def move(self):
        pass
a = Animal() # TypeError: Can't instantiate abstract class Animal with abstract methods move



class Animal():
    @abstractmethod
    def move(self):
        pass
a = Animal()              # No errors
```

Invoke Methods from Abstract Classes:

An abstract method is not needed to be “totally abstract” in Python. We can define some content in an abstract method and use super() to invoke it in subclasses.

Example:

```
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def move(self):
        print('Animal moves')

class Cat(Animal):
    def move(self):
        super().move()
        print('Cat moves')

c = Cat()
c.move()
```

Output:

```
Animal moves
Cat moves
```

Problem Definition:

1. For given program find output

Sr.No	Program	Output

1	`class Rectangle:`
	`    def __init__(self, length, width):`
	`        self.length = length`
	`        self.width = width`
	
	`    def area(self):`
	`        return self.length * self.width`
	
	`    def perimeter(self):`
	`        return 2 * self.length + 2 * self.width`
	
	`class Square(Rectangle):`
	`    def __init__(self, length):`
	`        super().__init__(length, length)`
	
	`square = Square(4)`
	`print(square.area())`
		16

2	`class Person:`
	`  def __init__(self, fname, lname):`
	`    self.firstname = fname`
	`    self.lastname = lname`
	
	`  def printname(self):`
	`    print(self.firstname, self.lastname)`
	
	`class Student(Person):`
	`  def __init__(self, fname, lname, year):`
	`    super().__init__(fname, lname)`
	`    self.graduationyear = year`
	
	`x = Student("Wilbert", "Galitz", 2018)`
	`print(x.graduationyear)`
		2018

3	class Bank:
	    def getroi(self):
	        return 10
	
	class SBI:
	        def getroi(self):
	            return 7
	
	class ICICI:
	   def getroi(self):
	        return 8
	
	b1=Bank()
	b2=SBI()
	b3=ICICI()
	print("Bank rate of interest:",b1.getroi())
	print("SBI rate of interest:",b2.getroi())
	print("ICICI rate of interest:",b3.getroi())
		Bank rate of interest: 10
		SBI rate of interest: 7
		ICICI rate of interest: 8

1. Create a class account that stores customer name, account number and type of account. From this derive the classes cur_acct and sav_acct to make them more specific to their requirements. Include necessary member functions in order to achieve the following tasks:

- Accept deposits from a customer and update the balance.
- Display the balance.
- Compute and deposit interest.
- Permit withdrawal and update the balance.
- Check for the minimum balance, impose penalty, necessary and update the balance.

2. Write a program that defines an abstract class called Vehicle containing an abstract method speed (). Derive from it two classes - FourWheeler and TwoWheeler. Create objects of derived classes and call the speed () method using these objects, passing to it the name of vehicle and speed of vehicle. In the speed () method print the vehicle name and the speed of vehicle to which speed () belongs.

Result

Books/ Journals/ Websites referred:

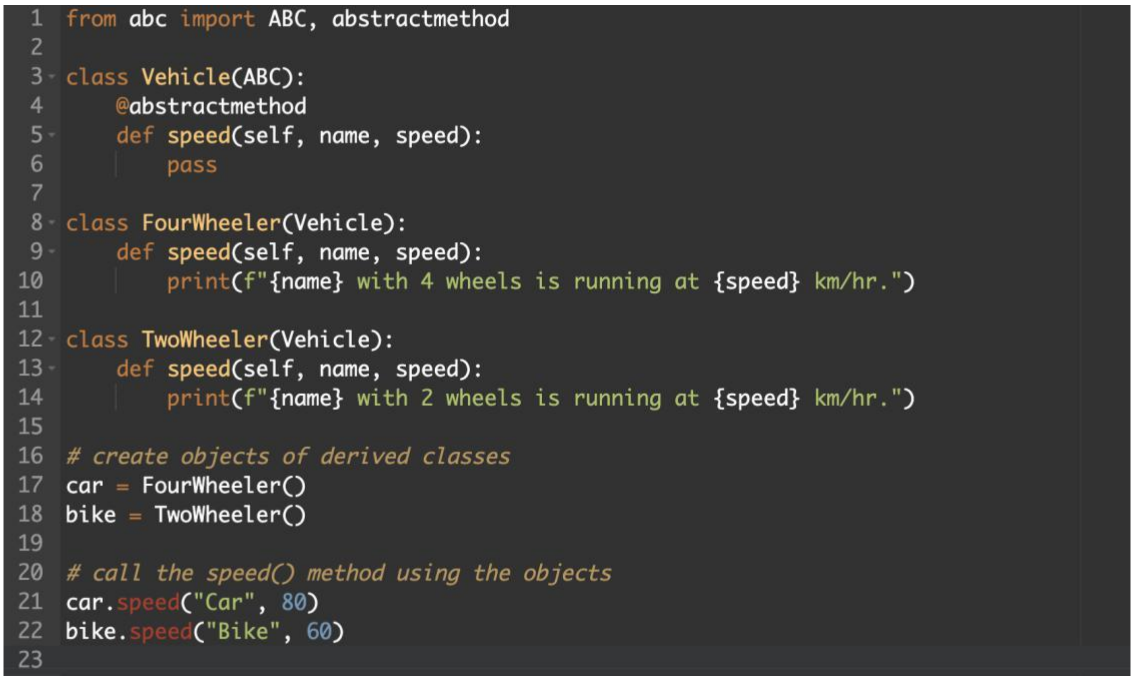
1. **Reema Thareja , "Python Programming: Using Problem Solving Approach", Oxford University Press, First Edition 2017, India**
2. **Sheetal Taneja and Naveen Kumar," Python Programing: A Modular Approach", Pearson India, Second Edition 2018, India**
3. https://www.programiz.com/python-programming/methods/built-in/super
4. https://www.tutorialsteacher.com/python/private-and-protected-access-modifiers-i n-python
5. https://www.geeksforgeeks.org/inheritance-in-python/

Implementation details:

Q1.

```
class Account:
    def __init__(self, name, account_number, account_type):
        self.name = name
        self.account_number = account_number
        self.account_type = account_type
        self.balance = 0

    def deposit(self, amount):
        self.balance += amount
        print(f"Deposit of {amount} successful. New balance: {self.balance}")

    def display_balance(self):
        print(f"Current balance: {self.balance}")

    def compute_interest(self, rate):
        interest = self.balance * rate / 100
        self.balance += interest
        print(f"Interest of {interest} credited. New balance: {self.balance}")

    def withdraw(self, amount):
        if amount > self.balance:
            print("Insufficient funds.")
        else:
            self.balance -= amount
            print(f"Withdrawal of {amount} successful. New balance: {self.balance}")


class CurrentAccount(Account):
    def __init__(self, name, account_number):
        super().__init__(name, account_number, "Current")
        self.minimum_balance = 0

    def set_minimum_balance(self, amount):
        self.minimum_balance = amount
        print(f"Minimum balance set to {amount}")

    def check_minimum_balance(self):
        if self.balance < self.minimum_balance:
            penalty = 0.05 * (self.minimum_balance - self.balance)
            self.balance -= penalty
            print(f"Minimum balance not maintained. Penalty of {penalty} imposed. New balance: {self.balance}")

    def withdraw(self, amount):
        if amount > self.balance - self.minimum_balance:
            print("Insufficient funds. Minimum balance not maintained.")
        else:
            self.balance -= amount
            print(f"Withdrawal of {amount} successful. New balance: {self.balance}")
            self.check_minimum_balance()


class SavingsAccount(Account):
    def __init__(self, name, account_number):
        super().__init__(name, account_number, "Savings")

    def compute_interest(self, rate):
        if self.balance >= 1000:
            interest = self.balance * rate / 100
            self.balance += interest
            print(f"Interest of {interest} credited. New balance: {self.balance}")
        else:
            print("Minimum balance not maintained. Interest not computed.")


ca = CurrentAccount("John Doe", "12345")
ca.set_minimum_balance(1000)

sa = SavingsAccount("Jane Doe", "67890")


ca.deposit(2000)
ca.withdraw(500)
ca.withdraw(1500)
ca.display_balance()

sa.deposit(2500)
sa.compute_interest(2)
sa.display_balance()
sa.compute_interest(2)
```

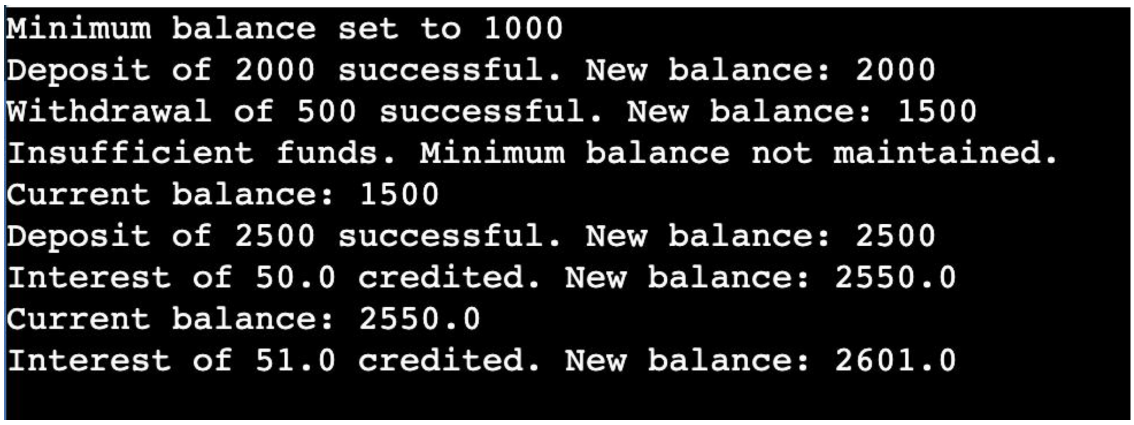
Q2.



```python
from abc import ABC, abstractmethod

class Vehicle(ABC):
    @abstractmethod
    def speed(self, name, speed):
        pass

class FourWheeler(Vehicle):
    def speed(self, name, speed):
        print(f"{name} with 4 wheels is running at {speed} km/hr.")

class TwoWheeler(Vehicle):
    def speed(self, name, speed):
        print(f"{name} with 2 wheels is running at {speed} km/hr.")

# create objects of derived classes
car = FourWheeler()
bike = TwoWheeler()

# call the speed() method using the objects
car.speed("Car", 80)
bike.speed("Bike", 60)

```

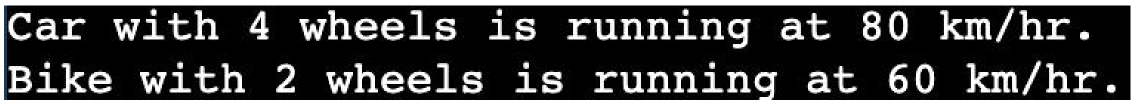
Output(s):

Q1.



```
Minimum balance set to 1000
Deposit of 2000 successful. New balance: 2000
Withdrawal of 500 successful. New balance: 1500
Insufficient funds. Minimum balance not maintained.
Current balance: 1500
Deposit of 2500 successful. New balance: 2500
Interest of 50.0 credited. New balance: 2550.0
Current balance: 2550.0
Interest of 51.0 credited. New balance: 2601.0
```

Q2.



```
Car with 4 wheels is running at 80 km/hr.
Bike with 2 wheels is running at 60 km/hr.
```

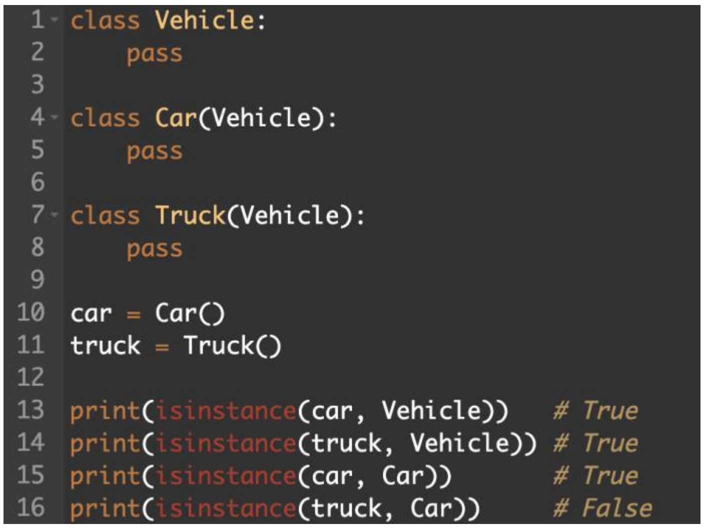
Conclusion:

Post Lab Questions:

1. Explain *isinstance()* and *issubclass()* functions with example

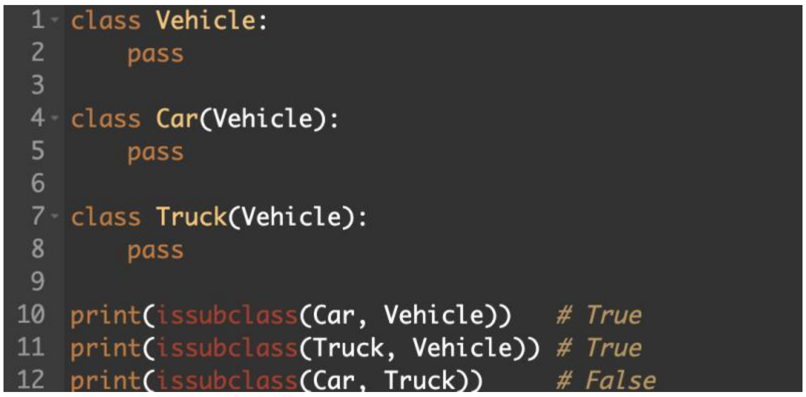
isinstance() and *issubclass()* are two built-in Python functions that are used for checking the type hierarchy of objects and classes.

isinstance() function takes two arguments: an object and a class (or a tuple of classes). It returns **True** if the object is an instance of the given class (or one of the classes in the tuple), and **False** otherwise. Here's an example:



```python
class Vehicle:
    pass

class Car(Vehicle):
    pass

class Truck(Vehicle):
    pass

car = Car()
truck = Truck()

print(isinstance(car, Vehicle))   # True
print(isinstance(truck, Vehicle)) # True
print(isinstance(car, Car))       # True
print(isinstance(truck, Car))     # False
```

issubclass() function takes two arguments: two classes. It returns **True** if the first class is a subclass of the second class, or if they are the same class, and **False** otherwise. Here's an example:

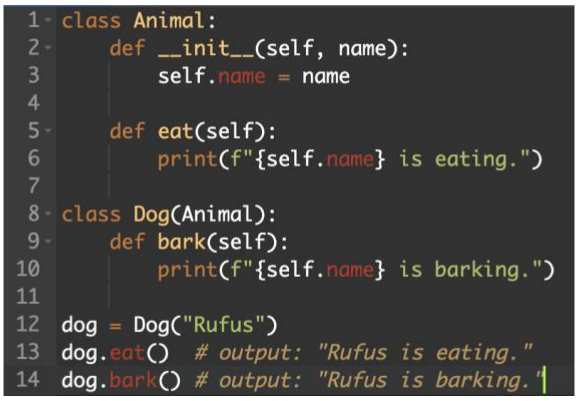


```python
class Vehicle:
    pass

class Car(Vehicle):
    pass

class Truck(Vehicle):
    pass

print(issubclass(Car, Vehicle))   # True
print(issubclass(Truck, Vehicle)) # True
print(issubclass(Car, Truck))     # False
```

2. Explain difference between inheritance and abstract class with example

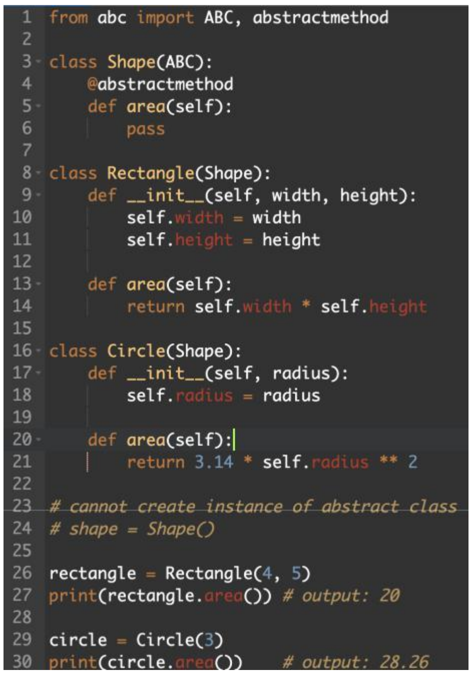
Inheritance and abstract classes are both mechanisms in object-oriented programming for creating classes that are related to each other, but they differ in their implementation and purpose.

Inheritance is a mechanism where a new class is derived from an existing class, inheriting all the attributes and methods of the parent class. The new class is known as the subclass or derived class, and the parent class is known as the superclass or base class. The subclass can override or extend the methods of the superclass, or add new methods and attributes of its own. Here's an example:



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

    def eat(self):
        print(f"{self.name} is eating.")

class Dog(Animal):
    def bark(self):
        print(f"{self.name} is barking.")

dog = Dog("Rufus")
dog.eat()  # output: "Rufus is eating."
dog.bark() # output: "Rufus is barking."
```

An abstract class is a class that cannot be instantiated on its own, but can be subclassed to create concrete classes. Abstract classes define abstract methods, which are methods that are declared but not implemented in the abstract class. The implementation of the abstract methods is left to the concrete subclasses. Abstract classes provide a way to define a common interface for a group of related classes, without specifying their exact implementation. Here's an example:



```python
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

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

    def area(self):
        return self.width * self.height

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

    def area(self):
        return 3.14 * self.radius ** 2

# cannot create instance of abstract class
# shape = Shape()

rectangle = Rectangle(4, 5)
print(rectangle.area()) # output: 20

circle = Circle(3)
print(circle.area())     # output: 28.26
```

Date: _____________ **Signature of faculty in-charge**