**Batch: P1 - 2 Roll No.: 16014022050**

**Experiment / assignment / tutorial No.: 7**

**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):

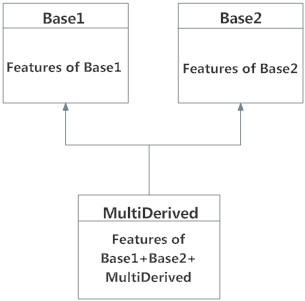
 . . . .

class Base2 (object):

. . . .

class Derived (Base1, Base2):

. . . .

 *Multiple Inheritance in Python*

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

class Person(object):

. . .

# Inherited or Sub class (Note Person in bracket)

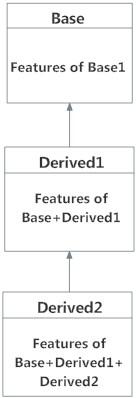
class Child(Base):

. . .

# Inherited or Sub class (Note Child in bracket)

class GrandChild(Child):

. . . .

 *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.

e.g.: 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

e.g.: 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:

e.g: 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.

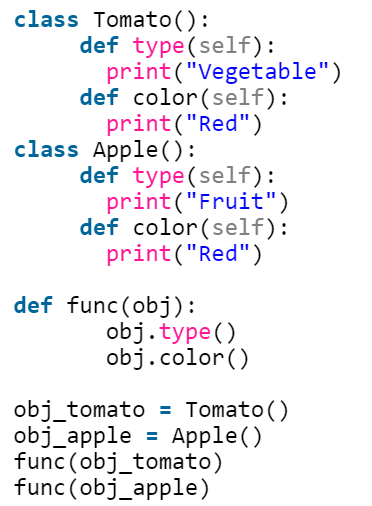
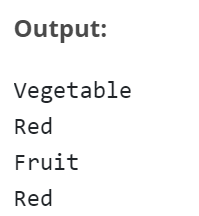
There are different methods to use polymorphism in Python. You can use different function, class methods or objects to define polymorphism.

**Polymorphism with Function and Objects -**

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

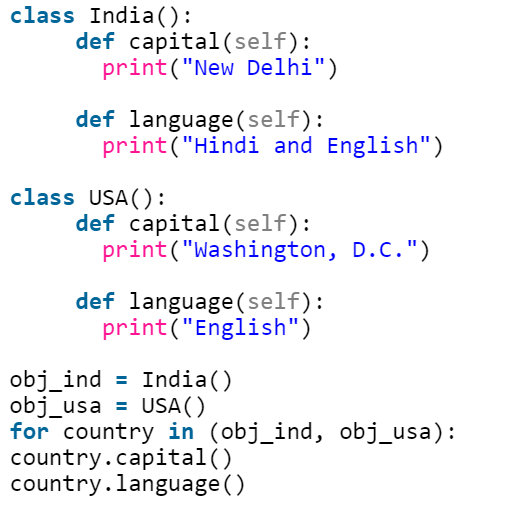
e.g.:

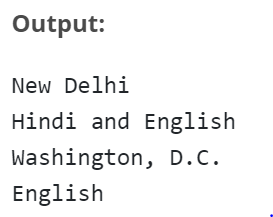
Create a function called “func()” which will take an object which we will name “obj”. and let give the function something to do that uses the ‘obj’ object which is passed to it. Now, call the methods type() and color(), each of which is defined in the two classes ‘Tomato’ and ‘Apple’ by creating instances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



**Polymorphism with Class Methods -**

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

e.g:

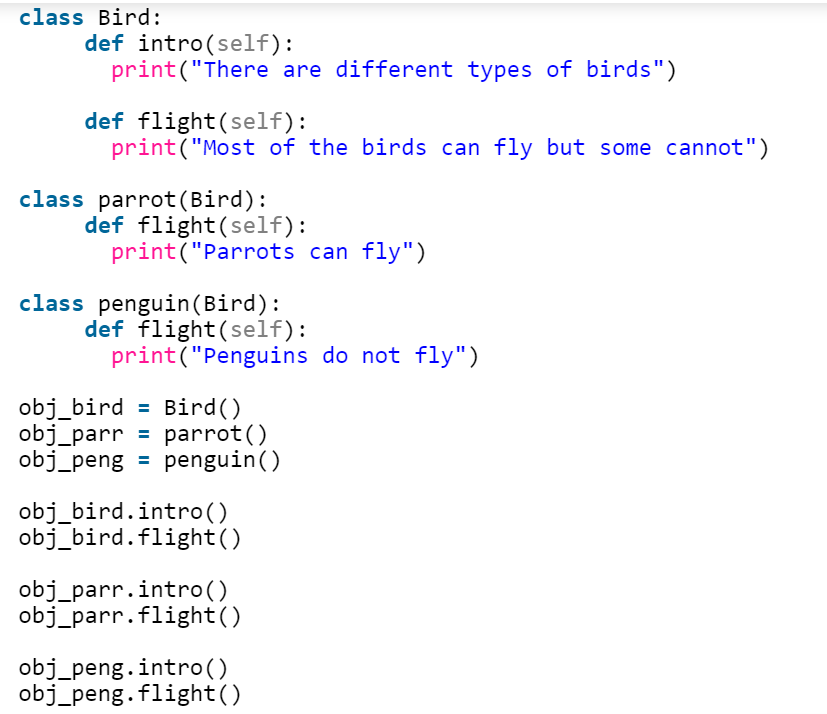
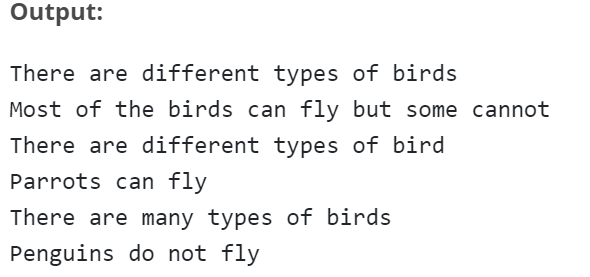


**Polymorphism with Inheritance -**

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.

e.g.:



**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.

**Define Abstract Class in Python -**

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.

e.g.:

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.

e.g.:

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 deposit 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.

1. 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.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**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-in-python>
  5. <https://www.geeksforgeeks.org/inheritance-in-python/>

**Implementation details:**

print(“Ketaki Mahajan / P1-2 / 16014022050”)

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("Deposit successful.\nUpdated balance: ", self.balance)

    def display\_balance(self):

        print(f"Account balance: {self.balance}")

    def withdraw(self, amount):

        if self.balance >= amount:

            self.balance -= amount

            print("Withdrawal successful.\nUpdated balance: ", self.balance)

        else:

            print("Insufficient balance.\nWithdrawal failed. ")

    def compute\_interest(self, rate):

        interest = (rate / 100) \* self.balance

        self.balance += interest

        print("Interest deposited.\nUpdated balance: ", self.balance)

class CurAccount(Account):

    def \_\_init\_\_(self, name, account\_number):

        super().\_\_init\_\_(name, account\_number, "Current Account")

    def check\_balance(self, minimum\_balance, penalty):

        if self.balance < minimum\_balance:

            self.balance -= penalty

            print("Minimum balance is NOT maintained and PENALTY CHARGED. Updated balance: ", self.balance)

class SavAccount(Account):

    def \_\_init\_\_(self, name, account\_number):

        super().\_\_init\_\_(name, account\_number, "Savings Account")

    def check\_balance(self, minimum\_balance, penalty):

        if self.balance < minimum\_balance:

            self.balance -= penalty

            print("Minimum balance is NOT maintained and PENALTY CHARGED. Updated balance: ", self.balance)

def createAccount():

    name = input("Enter customer name: ")

    account\_number = input("Enter account number: ")

    account\_type = input("Enter account type, either Current or Savings: ")

    if account\_type.lower() == "current":

        return CurAccount(name, account\_number)

    elif account\_type.lower() == "savings":

        return SavAccount(name, account\_number)

    else:

        print("INVALID account type.")

        return None

account = createAccount()

if account:

    while True:

        print("\n1. Deposit\n2. Display Balance\n3. Deposit Interest\n4. Withdraw\n5. Check Minimum Balance \n6. Exit")

        choice = input("Enter choice from 1 to 6: ")

        if choice == "1":

            amount = float(input("Enter deposit amount: "))

            account.deposit(amount)

        elif choice == "2":

            account.display\_balance()

        elif choice == "3":

            interest = float(input("Enter interest rate: "))

            account.compute\_interest(amount)

        elif choice == "4":

            interest = float(input("Enter withdrawal amount: "))

            account.withdraw(amount)

        elif choice == "5":

            min\_balance = float(input("Enter the minimum balance: "))

            penalty = float(input("Enter the penalty amount: "))

            amount.check\_balancee(min\_balance, penalty)

        elif choice == "6":

            break

        else:

            print("Invalid Choice.\nEnter number from 1 to 6.")

print(“Ketaki Mahajan / P1-2 / 16014022050”)

from abc import ABC, abstractmethod

class Vehicle(ABC):

    @abstractmethod

    def speed(self):

        pass

class FourWheeler(Vehicle):

    def \_\_init\_\_(self, name, speed):

        self.name = name

        self.speed\_val = speed

    def speed(self):

        print(f"Name: {self.name}")

        print(f"Speed: {self.speed\_val} km/h")

class TwoWheeler(Vehicle):

    def \_\_init\_\_(self, name, speed):

        self.name = name

        self.speed\_val = speed

    def speed(self):

        print(f"Name: {self.name}")

        print(f"Speed: {self.speed\_val} km/h")

name = input("Enter vehicle name: ")

four\_wheeler\_speed = float(input("Enter speed of four-wheeler: "))

two\_wheeler\_speed = float(input("Enter speed of two-wheeler: "))

four\_wheeler = FourWheeler(name, four\_wheeler\_speed)

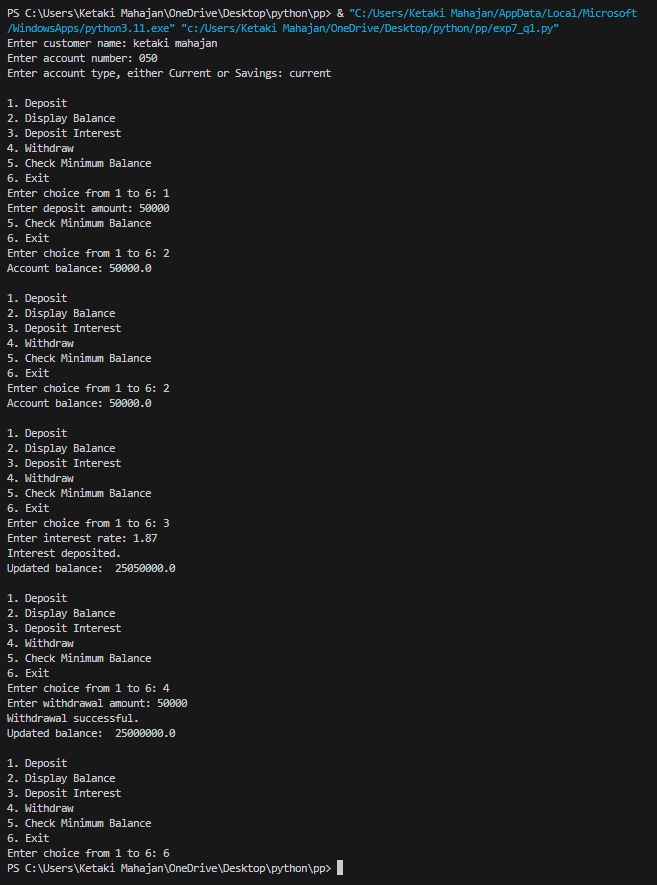
two\_wheeler = TwoWheeler(name, two\_wheeler\_speed)

four\_wheeler.speed()

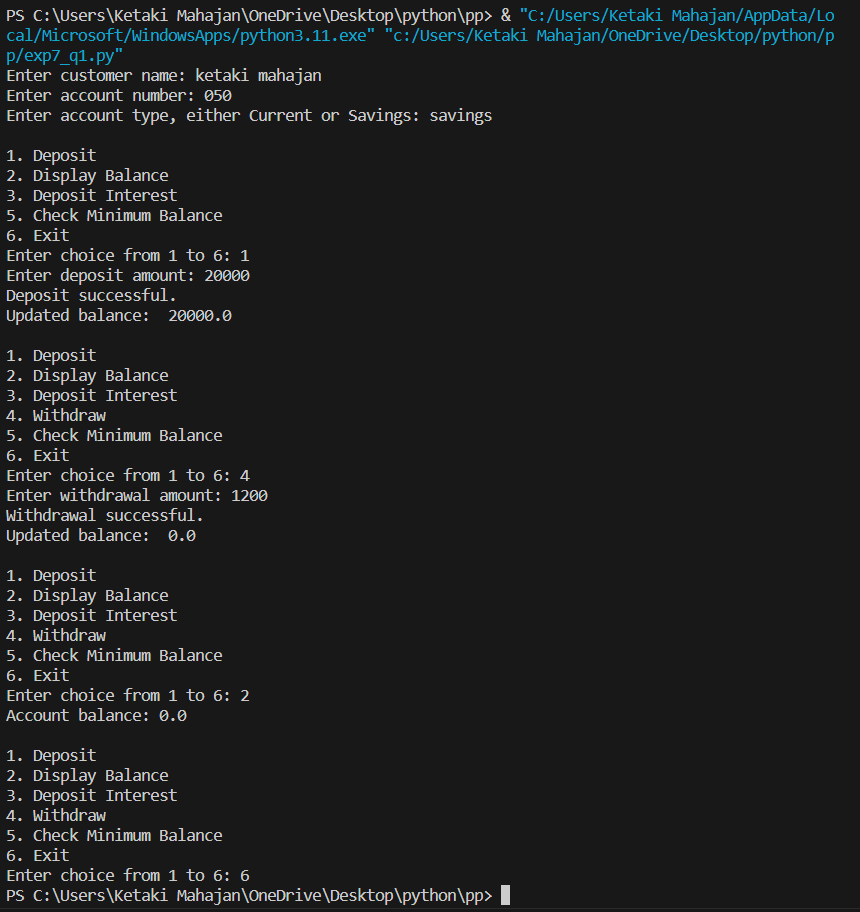
two\_wheeler.speed()

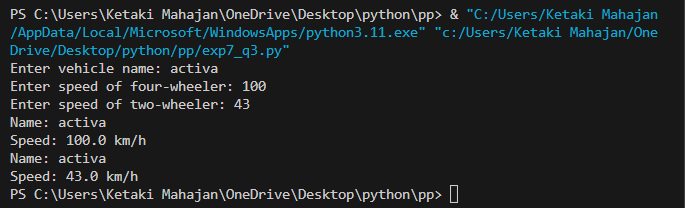
**Output(s):**

***(current account)***



***(savings account)***





**Conclusion:**

In conclusion, we have gained an understanding of inheritance in classes and its benefits. We have explored the usage of essential functions such as super(), isinstance(), and issubclass() within classes. Additionally, we have successfully implemented and grasped the concepts of polymorphism and abstraction

**Post Lab Questions:**

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

**isinstance() Function:** The isinstance() function is used to check if an object belongs to a specific class or any of its derived classes. It returns True if the object is an instance of the specified class or a subclass, and False otherwise.

class Vehicle: # base class is vehicle from which 2 derived classes 'Car' and 'Motorcycle' are created

    pass

class Car(Vehicle):

    pass

class Motorcycle(Vehicle):

    pass

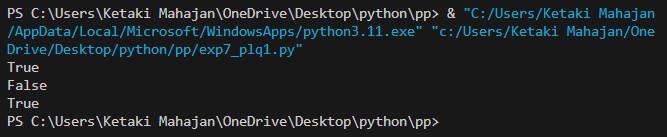
car = Car()

motorcycle = Motorcycle()

print(isinstance(car, Vehicle)) # checks if car object is instance of Vehicle class; As Car is derived from Vehicle, output is True

print(isinstance(motorcycle, Car)) # checks if motorcycle object is instance of Car class; As motorcycle is not an instance of Car, but rather an instance of Motorcycle, output is False

print(isinstance(motorcycle, Vehicle)) # checks if motorcycle object is instance of Vehicle class; As Motorcycle is derived from Vehicle, output is True.



**issubclass() Function:** The issubclass() function is used to check if a class is a subclass of another class. It returns True if the first class is a subclass of the second class, and False otherwise.

class Vehicle:

    pass

class Car(Vehicle):

    pass

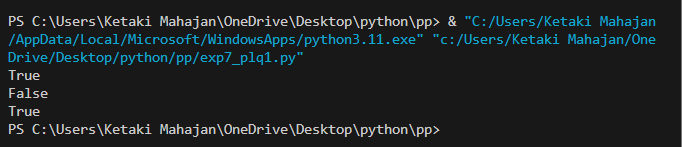
class Motorcycle(Vehicle):

    pass

print(issubclass(Car, Vehicle)) # checks if Car is subclass of Vehicle; As Car is derived from Vehicle, output is True

print(issubclass(Motorcycle, Car)) # checks if Motorcycle is subclass of Car; As Motorcycle is not derived from Car, output is False

print(issubclass(Motorcycle, Vehicle)) # checks if Motorcycle is subclass of Vehicle; As Motorcycle is derived from Vehicle, output is True



1. **Explain difference between inheritance and abstract class with example.**

**Inheritance** is a mechanism that allows a class to inherit properties and methods from another class. The class that inherits from another class is called a derived class or subclass, while the class being inherited from is called the base class or superclass.

The derived class inherits all the attributes and behaviors of the base class and can also add its own attributes and behaviors or override the inherited ones. This promotes code reuse and allows for creating specialized classes based on existing ones.

**e.g.:**

**class Animal:**

**def sound(self):**

**print("Animal makes a sound.")**

**class Dog(Animal):**

**def sound(self):**

**print("Dog barks.")**

**dog = Dog()**

**dog.sound()**

In this example, we have a base class Animal with a sound() method. The Dog class is derived from Animal and overrides the sound() method with its own implementation. When we create an instance of Dog and call the sound() method, it outputs "Dog barks."

An **abstract class** is a class that cannot be instantiated directly but can be used as a blueprint for creating derived classes. It is meant to be subclassed, and it may contain one or more abstract methods. An abstract method is a method that has a declaration but no implementation in the abstract class. Each subclass must provide an implementation for the abstract methods.

Abstract classes are useful when you want to define a common interface or behavior that should be shared by multiple subclasses, but you want to enforce that each subclass implements specific methods.

**e.g:**

**from abc import ABC, abstractmethod**

**class Shape(ABC):**

**@abstractmethod**

**def area(self):**

**pass**

**class Rectangle(Shape):**

**def \_\_init\_\_(self, length, width):**

**self.length = length**

**self.width = width**

**def area(self):**

**return self.length \* self.width**

**rectangle = Rectangle(5, 3)**

**print(rectangle.area())**

In this example, Shape is an abstract class that defines the abstract method area(). The Rectangle class inherits from Shape and provides an implementation for the area() method. The rectangle object, an instance of the Rectangle class, can call the area() method to calculate and return the area of the rectangle.

**Date: \_\_\_\_\_\_\_\_\_\_\_\_\_ Signature of faculty in-charge**