Experiment – 2.1

Aim: Write a Python program that defines a Car class with attributes like make, model, and year, and methods like start() to start the car and stop() to stop it.

Description:

This program provides a foundational example of Object-Oriented Programming (OOP) in Python. It defines a class named Car, which serves as a blueprint for creating car objects. The class is structured to have data attributes (make, model, and year) and behavioral methods (start() and stop()). The __init__ method, also known as the constructor, is used to initialize the attributes of each Car object upon its creation. The start() and stop() methods perform actions associated with the object by printing messages that incorporate the car's attributes. The program then demonstrates the usage of this class by creating two different car objects and calling their respective methods to produce specific output.

Algorithm

1. Start the program.
2. Define a class named Car.
3. Inside the Car class, define the __init__ method, which accepts make, model, and year as parameters.
4. Assign the values of the parameters to the corresponding attributes (self.make, self.model, self.year) of the object being created.
5. Define the start() method, which prints a message indicating the car is starting, using its attributes.
6. Define the stop() method, which prints a message indicating the car is stopping, using its attributes.
7. Create the first object, car1, by calling the Car class constructor with the arguments "Toyota", "Camry", and 2022.
8. Create the second object, car2, by calling the Car class constructor with the arguments "Toyota", "Camry", and 2032.
9. Call the start() method on the car1 object.
10. Call the stop() method on the car2 object.
11. End the program.

Program:

```
class Car:
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year
    def start(self):
        print(f"The {self.year} {self.make} {self.model} is starting.")
    def stop(self):
        print(f"The {self.year} {self.make} {self.model} is stopping.")
# Example usage
car1 = Car("Kia", "K5", 2022)
car2=Car("Kia", "K5", 2032)
car1.start()
car2.stop()
```

Output:

```
=============== RESTART: C:/Users/krish/Desktop/python lab/2.1.py ==============
The 2022 Kia K5 is starting.
The 2032 Kia K5 is stopping.
```

Experiment – 2.2

Aim: Write a Python program that demonstrates inheritance by creating a base class Animal and derived classes like Dog, Cat, etc., each with their specific behaviors.

Description: This program is an excellent demonstration of inheritance, a core concept of Object-Oriented Programming (OOP) in Python. Inheritance allows a class to inherit attributes and methods from another class. The program begins by defining a base class, Animal, which includes a common attribute (name) and a shared behavior (eat()).

Two new classes, Dog and Cat, are then created. These are known as derived or child classes, and they inherit from the Animal base class. This means that both Dog and Cat objects automatically possess the name attribute and the eat() method without needing to be defined again. In addition to the inherited behaviors, each derived class defines its own unique method: Dog has a bark() method, and Cat has a meow() method. The program concludes by creating instances of Dog and Cat and shows how they can use both the inherited eat() method and their own specific methods.

Algorithm

1. Start the program.
2. Define the base class Animal with an __init__ method to initialize an instance with a name attribute.
3. Define a generic eat() method within the Animal class.
4. Define the Dog class, specifying that it inherits from Animal.
5. Define a bark() method specific to the Dog class.
6. Define the Cat class, specifying that it inherits from Animal.
7. Define a meow() method specific to the Cat class.
8. Create an instance of the Dog class, named dog, with the name "Buddy".
9. Call the inherited eat() method on the dog object.
10. Call the bark() method on the dog object.
11. Create an instance of the Cat class, named cat, with the name "Whiskers".
12. Call the inherited eat() method on the cat object.
13. Call the meow() method on the cat object.
14. End the program.

PROGRAM:

```
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.")
class Cat(Animal):
    def meow(self):
        print(f"{self.name} is meowing.")
# Example usage
dog = Dog("DON")
cat = Cat("Robin")
dog.eat()
dog.bark()
cat.eat()
cat.meow()
```

OUTPUT:

```
=============== RESTART: C:/Users/krish/Desktop/python lab/2.2.py ==============
DON is eating.
DON is barking.
Robin is eating.
Robin is meowing.
```

Experiment – 2.3

Aim: Write a python program to define a base class called Animal with a method make_sound(). Implement derived classes like Dog, Cat, and Bird that override the make_sound() method to produce different sounds. Demonstrate polymorphism by calling the method on objects of different classes.

Description:

This program demonstrates polymorphism in Python, a key concept in Object-Oriented Programming (OOP) where a single interface can be used for different underlying data types. The program first defines a base class, Animal, with a generic make_sound() method. It then creates three derived classes—Dog, Cat, and Bird—that inherit from Animal. Each of these derived classes overrides the make_sound() method, providing a unique and specific implementation (e.g., "Woof!", "Meow!", "Tweet!"). The final part of the code showcases polymorphism in action by creating a list of mixed animal objects. It then iterates through this list, and for each object, it calls the make_sound() method. Despite calling the same method name, Python's runtime determines which specific make_sound() implementation to execute based on the object's class, producing the correct sound for each animal.

Algorithm

1. Start the program.
2. Define a base class Animal with a method make_sound() that prints a generic sound.
3. Define the Dog class, inheriting from Animal.
4. Override the make_sound() method within the Dog class to print "Woof!".
5. Define the Cat class, inheriting from Animal.
6. Override the make_sound() method within the Cat class to print "Meow!".
7. Define the Bird class, inheriting from Animal.
8. Override the make_sound() method within the Bird class to print "Tweet!".
9. Create a list named animals containing instances of Dog, Cat, and Bird.
10. Start a for loop to iterate through each object in the animals list.
11. In each iteration, call the make_sound() method on the current animal object.
12. End the program.

PROGRAM:

```
class Animal:
    def make_sound(self):
        print("Some generic animal sound.")

class Dog(Animal):
    def make_sound(self):
        print("BOWW!")

class Cat(Animal):
    def make_sound(self):
        print("Meow!")

class Bird(Animal):
    def make_sound(self):
        print("Tweet Tweet!")
# Polymorphic behavior
animals = [Dog(), Cat(), Bird()]

for animal in animals:
    animal.make_sound()
```

OUTPUT:

```
=============== RESTART: C:/Users/krish/Desktop/python lab/2.3.py ==============
BOWW!
Meow!
Tweet Tweet!
```

Experiment – 2.4

Aim: Write a Python program that demonstrates error handling using the try-except block to handle division by zero.

Description:

This program demonstrates how to handle potential errors in a Python program using a try-except block. This is a fundamental concept for creating robust and user-friendly applications that can gracefully manage unexpected situations. The program's goal is to perform a simple division operation. It prompts the user for a numerator and a denominator. The core of the program is the try block, which attempts to perform the division. If the division is successful, the result is printed. However, if the user enters a non-numerical value, a ValueError is raised, and the program executes the first except block, providing a custom error message. Similarly, if the user enters 0 for the denominator, a ZeroDivisionError is raised, and the second except block is executed, preventing the program from crashing and instead informing the user about the error**.**

Algorithm

1. Start the program.
2. Begin a try block to monitor for potential errors.
3. Inside the try block:
    - Prompt the user to enter a numerator and store it as an integer.
    - Prompt the user to enter a denominator and store it as an integer.
    - Perform the division of the numerator by the denominator and store the result.
    - Print the final result of the division.
4. If a ZeroDivisionError occurs (e.g., the user enters 0 for the denominator), skip the rest of the try block and execute the except ZeroDivisionError block.
5. Inside the except ZeroDivisionError block, print the error message: "Error: Cannot divide by zero."
6. If a ValueError occurs (e.g., the user enters text instead of a number), skip the rest of the try block and execute the except ValueError block.
7. Inside the except ValueError block, print the error message: "Error: Please enter valid integers."

8. End the program.

PROGRAM:

```
try:
    numerator = int(input("Enter numerator: "))
    denominator = int(input("Enter denominator: "))
    result = numerator / denominator
    print(f"Result: {result}")
except ZeroDivisionError:
    print("Error: Cannot divide by zero.")
except ValueError:
    print("Error: Please enter valid integers.")
```

OUTPUT:

```
>>>
========== RESTART: C:/Users/krish/Desktop/python lab/2.4.py ==========
Enter numerator: 10
Enter denominator: 2
Result: 5.0
>>>
========== RESTART: C:/Users/krish/Desktop/python lab/2.4.py ==========
Enter numerator: 5
Enter denominator: 0
Error: Cannot divide by zero.
>>>
========== RESTART: C:/Users/krish/Desktop/python lab/2.4.py ==========
Enter numerator: 5
Enter denominator: h
Error: Please enter valid integers.
>>>
```