PD LAB

ASSIGNMENT - 10

Name: Raunak Thanawala

Registration Number: 231070051

Branch: Computer Engineering

Batch: 3

**Aim:-**

To study Classes and Objects in python

**Theory:-**

1. **Classes and Objects**

* Class:
  + A class is a blueprint or template for creating objects.
  + It defines the properties and behaviors that the objects created from it will have.
* Object:
  + An instance of a class.
  + An object is a specific realization of a class, containing actual values for the attributes defined by the class.

1. **Attributes**

* Class Attributes:
  + Variables defined within a class but outside any method.
  + They are shared by all instances of the class.
* Instance Attributes:
  + Variables that are specific to each instance of a class.
  + Typically, they are defined within the \_\_init\_\_ method, which is called when an object is created.

1. **Methods**
   * Instance Methods:
     + Functions defined inside a class that operate on instances of the class.
     + They are usually used to manipulate instance attributes.
   * Class Methods:
     + Defined using the @classmethod decorator.
     + They receive the class itself as the first parameter (cls) and are used to manipulate class attributes or create alternative constructors.
   * Static Methods:
     + Defined using the @staticmethod decorator.
     + They do not take self or cls as a parameter and act as utility functions within a class.
2. **Constructors**

* Initializer (\_\_init\_\_):
  + A special method called when an object is instantiated.
  + Used to initialize instance attributes.

1. **Destructors**

* Destructor (\_\_del\_\_):
  + A special method called when an object is about to be destroyed.
  + Used for cleanup tasks before the object is removed from memory.

1. **Encapsulation**

* The concept of restricting access to certain data and methods within a class.
* In Python, this is achieved by prefixing attributes or methods with a single or double underscore, marking them as protected or private, respectively.

1. **Inheritance**

* A mechanism allowing a new class to inherit attributes and methods from an existing class.
* This enables code reuse and establishes a relationship between classes (such as a parent-child or superclass-subclass relationship).

1. **Polymorphism**

* The ability to define methods in different ways across different classes or instances.
* It allows objects of different types to be treated uniformly based on shared methods or interfaces.

1. **Abstraction**

* Hiding implementation details and exposing only essential information and behaviors.
* Abstraction is achieved by using abstract classes or interfaces that define common behaviors without specifying implementation.

1. **Magic Methods (Dunder Methods)**

* Special methods surrounded by double underscores, such as \_\_init\_\_, \_\_str\_\_, and \_\_len\_\_.
* These methods allow customization of object behavior in Python, enabling operators and functions to work with user-defined objects.

**Code:**

class Pet:

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

self.species = species

def make\_sound(self):

return "Generic Sound"

class Dog(Pet):

def \_\_init\_\_(self, breed, color):

super().\_\_init\_\_("Dog")

self.breed = breed

self.color = color

def make\_sound(self):

return "Bark"

class Cat(Pet):

def \_\_init\_\_(self, breed, color):

super().\_\_init\_\_("Cat")

self.breed = breed

self.color = color

def make\_sound(self):

return "Meow"

Rodger = Dog("Pug", "brown")

Buzo = Dog("Bulldog", "black")

Kitty = Cat("Persian", "white")

print("Rodger details: ")

print("Species:", Rodger.species)

print("Breed:", Rodger.breed)

print("Color:", Rodger.color)

print("Sound:", Rodger.make\_sound())

print("\nBuzo details: ")

print("Species:", Buzo.species)

print("Breed:", Buzo.breed)

print("Color:", Buzo.color)

print("Sound:", Buzo.make\_sound())

print("\nKitty details: ")

print("Species:", Kitty.species)

print("Breed:", Kitty.breed)

print("Color:", Kitty.color)

print("Sound:", Kitty.make\_sound())

pets = [Rodger, Buzo, Kitty]

for pet in pets:

print(f"{pet.species} makes sound: {pet.make\_sound()}")

class GFG:

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

self.name = name

self.company = company

def show(self):

print("Hello my name is " + self.name+" and I" + " work in "+self.company+".")

obj = GFG("John", "GeeksForGeeks")

obj.show()

class Geek:

def \_\_init\_\_(somename, name, company):

somename.name = name

somename.company = company

def show(somename):

print("Hello my name is " + somename.name + " and I work in "+somename.company+".")

obj = Geek("James", "W3Schools")

obj.show()

class MyClass:

pass

class Strin:

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

self.name = name

self.college = college

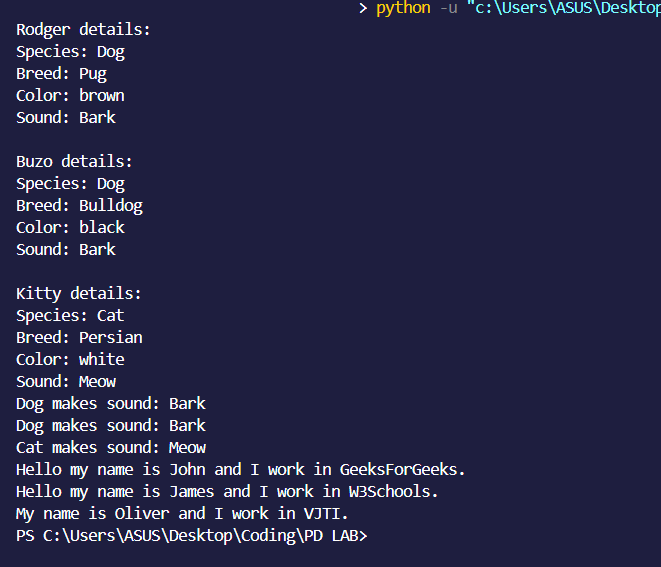
def \_\_str\_\_ (self):

return f"My name is {self.name} and I work in {self.college}."

obj = Strin("Oliver" , "VJTI")

print(obj)

**OUTPUT:**

****

**CONCLUSION:**

Thus we have written a program that shows classes and objects illustrated in python.

We have also given a brief explanation about the classes and objects in python.