Course Name: Fundamentals of Programming & Data Science	**Course Code:** CMPE-112L
Assignment Type: Lab	**Dated:** 15th April, 2024
Semester: 2nd	**Session:** 2023
Lab/Project/Assignment #: 9	**CLOs to be covered: CLO 3**
Lab Title: Introduction to OOP and Inheritance	**Teacher Name:** Engr. Afeef Obaid

Lab Evaluation:

CLO 3	Adhere to plagiarism guidelines					
Levels (Marks)	**Level 1**	**Level 2**	**Level 3**	**Level 4**	**Level 5**	**Level 6**
(10)						
					Total	**/10**

Rubrics for Current Lab Evaluation

Scale	**Marks**	**Level**	**Rubric**
Excellent	**10**	L1	Submitted all lab tasks, BONUS task, have good understanding.
Very Good	**8**	L2	Submitted the lab tasks but have good understanding
Good	**6**	L3	Submitted the lab tasks but have weak understanding.
Basic	**4**	L4	Submitted the lab tasks but have no understanding.
Barely Acceptable	**2**	L5	Submitted only one lab task.
Not Acceptable	**0**	L6	Did not attempt

LAB No. 10

Lab Goals/Objectives:

By reading this manual, students will be able:

- To understand the concept of Classes in Python
- To Become familiar with the concept of Objects in Python
- To learn about Constructor in Python
- To learn about Inheritance in Python

Equipment Required: Computer system with Pycharm IDE and python package installed on it

Introduction to Object-oriented programming

In programming languages, mainly there are two approaches that are used to write program or code.

1) Procedural Programming

2) Object-Oriented Programming

The procedure we are following till now is the "**Procedural Programming**" approach. So, in this Lab, we will learn about Object Oriented Programming (OOP). The basic idea of object-oriented programming (OOP) in Python is to use classes and objects to represent real-world concepts and entities.

Features of OOP

- **Encapsulation**

  One of the key features of OOP in Python is encapsulation, which means that the internal state of an object is hidden and can only be accessed or modified through the object's methods. This helps to protect the object's data and prevent it from being modified in unexpected ways.

- **Inheritance**

  Another key feature of OOP in Python is inheritance, which allows new classes to be created that inherit the properties and methods of an existing class. This allows for code reuse and makes it easy to create new classes that have similar functionality to existing classes.

- **Polymorphism**

  Polymorphism is also supported in Python, which means that objects of different classes can be treated as if they were objects of a common class. This allows for greater flexibility in code and makes it easier to write code that can work with multiple types of objects.

In summary, OOP in Python allows developers to model real-world concepts and entities using classes and objects, encapsulate data, reuse code through inheritance, and write more flexible code through polymorphism.

Python Class

A class is a blueprint or the prototype from which the objects are being created. It is a logical entity that contains some attributes and methods. It defines the properties and methods that an object of that class will have. Properties are the data or state of an object, and methods are the actions or behaviors that an object can perform.

To understand the need for creating a class let's consider an example, let's say you wanted to track the number of parrots that may have different attributes like breed, and age. If a list is used, the first element could be the parrot's breed while the second element could represent its age. Let's suppose there are 100 different parrots, then how would you know which element is supposed to be which? What if you wanted to add other properties to these parrots? This lacks organization and it's the exact need for classes.

Some points on Python class:

- Classes are created by keyword class.
- Attributes are the variables that belong to a class.
- Attributes are always public and can be accessed using the dot (.) operator. Eg.: Myclass.Myattribute
- Methods are the function that belongs to a class
- Methods are always public and can be accessed using the dot (.) operator. Eg.: Myclass.MyMethod

Syntax

```
class ClassName:
  # Statement-1
  .
  .
  .
  # Statement-N
```

Example

```
class parrot:
  name=""
  age=0
  language=""
  def spec(self):
    print(f"My name is {self.name}, my age is {self.age} and i can speak {self.language}")
```

Python Object

An object is an instance of a class, and it contains its own data and methods. For example, you have created a class called "Parrot" that has properties such as name, age and language, and a method such as spec(). Each instance of the Parrot class would be a unique object with its own name, age and language but they would all have the same methods of spec().

An object consists of:

State: It is represented by the attributes of an object. It also reflects the properties of an object.

Behavior: It is represented by the methods of an object. It also reflects the response of an object to other objects.

Identity: It gives a unique name to an object and enables one object to interact with other objects.

Example

```
class parrot:
  name=""
  age=0
  language=""
```

```
    def spec(self):
        print(f"My name is {self.name}, my age is {self.age} and i can speak {self.language}")
obj1=parrot()
obj1.name=input("Enter Name of Parrot ")
obj1.age=int(input("Enter age of Parrot "))
obj1.language=input("Enter language of Parrot ")
obj2=parrot()
obj2.name=input("Enter Name of Parrot ")
obj2.age=int(input("Enter age of Parrot "))
obj2.language=input("Enter language of Parrot ")
obj1.spec()
obj2.spec()
```

Python Self Keyword

In object-oriented programming, whenever we define methods for a class, we use self as the first parameter in each case as we at the definition of a class called parrot.

We know that class is a blueprint for the objects. This blueprint can be used to create multiple numbers of objects as we have created two objects of above parrot class.

The self keyword is used to represent an instance (object) of the given class. In above case, the two parrot objects obj1 and obj2 have their own name, age and language attributes. If there was no self argument, the same class couldn't hold the information for both these objects.

However, since the class is just a blueprint, self allows access to the attributes and methods of each object in python. This allows each object to have its own attributes and methods. Thus, even long before creating these objects, we reference the objects as self while defining the class.

Empty Class

Generally, a class contains data members known as attributes of the class and member functions that are used to modify the attributes of the class. But we can make a class without data members and member functions, this class is called Empty class.

In Python, to write an empty class **pass** statement is used. **pass** is a special statement in Python that does nothing. It only works as a dummy statement. However, objects of an empty class can also be created

Example

```
class parrot:
    pass
obj=parrot()
```

Constructors in Python

Constructors are generally used for instantiating an object. The task of constructors is to initialize(assign values) to the data members of the class when an object of the class is created. In Python the __init__() method is called the constructor and is always called when an object is created.

Syntax

```
def __init__(self):
  # body of the constructor
```

Types of constructors:

Default constructor: The default constructor is a simple constructor which doesn't accept any arguments. Its definition has only one argument which is a reference to the instance being constructed.

Example

```
class parrot:
  name=""
```

```
  age=0
  language=""
  def __init__(self):
    self.name="Grey Parrot"
    self.age=2
    self.language="English"
    print(f"My name is {self.name}, my age is {self.age} and i can speak {self.language}")
  def spec(self):
    print(f"My name is {self.name}, my age is {self.age} and i can speak {self.language}")
obj1=parrot()
obj1.name=input("Enter Name of Parrot ")
obj1.age=int(input("Enter age of Parrot "))
obj1.language=input("Enter language of Parrot ")
obj1.spec()
```

Parameterized constructor: constructor with parameters is known as parameterized constructor. The parameterized constructor takes its first argument as a reference to the instance being constructed known as self and the rest of the arguments are provided by the programmer.

Example

```
class parrot:
  name=""
  age=0
  language=""
  def __init__(self,n,a,l):
    self.name=n
    self.age=a
    self.language=l
  def spec(self):
    print(f"My name is {self.name}, my age is {self.age} and i can speak {self.language}")
n=input("Enter Name of Parrot ")
a=int(input("Enter age of Parrot "))
l=input("Enter language of Parrot ")
obj1=parrot(n,a,l)
obj1.spec()
```

Inheritance in Python

One of the core concepts in object-oriented programming (OOP) languages is inheritance. It is a mechanism that allows you to create a hierarchy of classes that share a set of properties and methods by deriving a class from another class. Inheritance is the capability of one class to derive or inherit the properties from another class.

Benefits of inheritance: Inheritance allows you to inherit the properties of a class, i.e., base class to another, i.e., derived class. The benefits of Inheritance in Python are as follows:

- It represents real-world relationships well.
- It provides the 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.
- 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.
- Inheritance offers a simple, understandable model structure.
- Less development and maintenance expenses result from an inheritance.

Syntax

```
Class BaseClass:
  {Body}

Class DerivedClass(BaseClass):
  {Body}
```

Parent Class

A parent class is a class whose properties are inherited by the child class. Let's create a parent class called Person which has a Display method to display the person's information.

```
class Person:
  def __init__(self,name,id):
    self.name=name
    self.id=id
  def Display_Person_Details(self):
    print(f"Name:{self.name}\nID:{self.id}")
```

```
per=Person("Ali",1)
per.Display_Person_Details()
```

Child Class

A child class is a class that drives the properties from its parent class. Here Programmer is another class that is going to inherit the properties of the Person class(base class).

```
class Person:
   def __init__(self,name,id):
      self.name=name
      self.id=id
   def Display_Person_Details(self):
      print(f"Name:{self.name}\nID:{self.id}")
class Programmer(Person):
   def Display_Programmer_Details(self):
      print(f"I am in Programming class\nName:{self.name}\nID:{self.id}")
per=Programmer("Ali",1)
per.Display_Person_Details()
per.Display_Programmer_Details()
```

Calling Constructor of Parent Class

A child class needs to identify which class is its parent class. This can be done by mentioning the parent class name in the definition of the child class.

If we create the object of child class then only its constructor will call and the attributes of child class will initialize only.

Now the attributes of Parent class will not be available for the child class so we must have to call the constructor of Parent class as well so that its attributes get initialized and available for child class

```
class Person:
   def __init__(self,name,id):
      self.name=name
      self.id=id
   def Display_Person_Details(self):
```

```
      print(f"Name:{self.name}\nID:{self.id}")
class Programmer(Person):
   def __init__(self,name,id,age,Language):
      self.age=age
      self.Language=Language
      Person.__init__(self,name,id)
   def Display_Programmer_Details(self):
      print(f"Age:{self.age}\nProgramming Language:{self.Language}")
per=Programmer("Ali",1,25,"Python")
per.Display_Person_Details()
per.Display_Programmer_Details()
```

The super() Function

The super() function is a built-in function that returns the objects that represent the parent class. It allows to access the parent class's methods and attributes in the child class.

```
class Person:
   def __init__(self,name,id):
      self.name=name
      self.id=id
   def Display_Person_Details(self):
      print(f"Name:{self.name}\nID:{self.id}")
class Programmer(Person):
   def __init__(self,name,id,age,Language):
      self.age=age
      self.Language=Language
      super().__init__(name,id)
   def Display_Programmer_Details(self):
      print(f"Age:{self.age}\nProgramming Language:{self.Language}")
per=Programmer("Ali",1,25,"Python")
per.Display_Person_Details()
per.Display_Programmer_Details()
```

Lab Tasks

- Write the python program to create a class by name Students, and initialize attributes like name, age, and grade while creating an object.

- Write a python program to create a class by name chair with attributes color, height and material and create its three objects and initialize the attributes while creating the objects and add a method disp() for displaying the features of chair objects.

- Write a python program to create a class by name rectangle and initialize attributes of `length` and `width` while creating an object, and add methods of perimeter() and area() for calculating the perimeter and area of the rectangle.

- Write a python program to create a class by name calculator and initialize attributes of `1st_num` and `2nd_num` while creating an object, and add methods of add(), sub(), mul() and div() for calculating the addition, subtraction, multiplication and division of given numbers.

- Write a python program to create a parent class by name circle with attributes radius and method of area(), then create a child class by name cylinder with attribute height and method of volume() that takes the value of area from parent class and multiply it with height to get volume. Calculate the volume of cylinder by creating the object of volume class that initialize the value of radius and height.