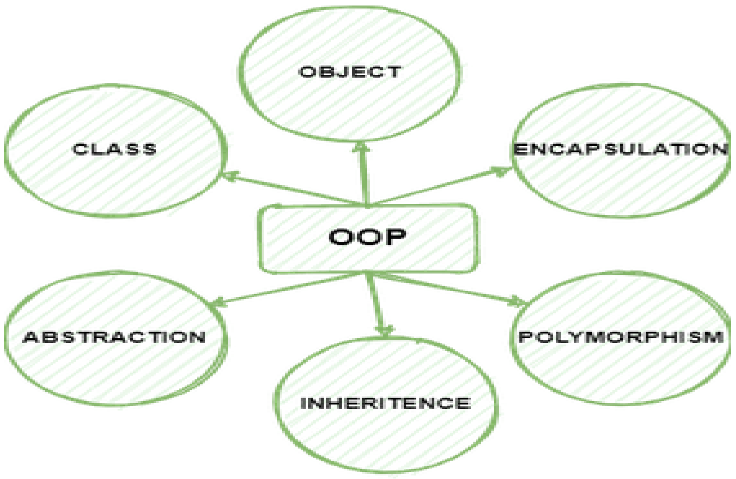
UNIT 2 : OBJECT ORIENTED PROGRAMMING IN PYTHON

The main concept of OOPs is to bind the data and the functions that work on that together as a single unit so that no other part of the code can access this data.

OOPs Concepts in Python

- Class
- Objects
- Polymorphism
- Encapsulation
- Inheritance
- Data Abstraction



Class

1. A class is a collection of variables and fuctions.
2. when function defined inside the class it is called as methods.
3. Classes are created by keyword class.
4. Attributes are the variables that belong to a class.
5. Attributes are always public and can be accessed using the dot (.) operator. Eg.: Myclass.Myattribute
6. Classes provide a means of bundling data and functionality together.

Objects

1. An object is an instance for the class.
2. with the help of object we can access the members of the class.
3. The object is an entity that has a state and behavior associated with it.
4. It may be any real-world object like a mouse, keyboard, chair, table, pen, etc. Integers, strings, floating-point numbers, even arrays, and dictionaries, are all objects.

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 1

```
class  MyClass:
   x =  5
p1 = MyClass()     #p1 is the object
print(p1.x)
```

#EXAMPLE 2

```
class  Person:
    def  __init__(self, name, age):
        self.name = name
       self.age = age
p1 = Person("John",  36)
print(p1.name)
print(p1.age)
```

1. All classes have a function called __init__(), which is always executed when the class is being initiated.
2. Use the __init__() function to assign values to object properties, or other operations that are necessary to do when the object is being created

Inheritance

1. Inheritance is the capability of one class to derive or inherit the properties from another class.
2. The class that derives properties is called the derived class or child class and the class from which the properties are being derived is called the base class or parent class.
3. It represents real-world relationships well.
4. It provides the reusability of a code.
5. We don' t have to write the same code again and again.
6. Also, it allows us to add more features to a class without modifying it.

- **Types of Inheritance**
  1. **Single Inheritance**: Single-level inheritance enables a derived class to inherit characteristics from a single-parent class.
  2. **Multilevel Inheritance:** Multi-level inheritance enables a derived class to inherit properties from an immediate parent class which in turn inherits properties from his parent class.
  3. **Hierarchical Inheritance:** Hierarchical-level inheritance enables more than one derived class to inherit properties from a parent class.
  4. **Multiple Inheritance:** Multiple-level inheritance enables one derived class to inherit properties from more than one base class.

Q] Create a **Bus** child class that inherits from the Vehicle class. The default fare charge of any vehicle is **seating capacity * 100**. If Vehicle is **Bus** instance, we need to add an extra 10% on full fare as a maintenance charge. So total fare for bus instance will become the **final amount = total fare + 10% of the total fare.**

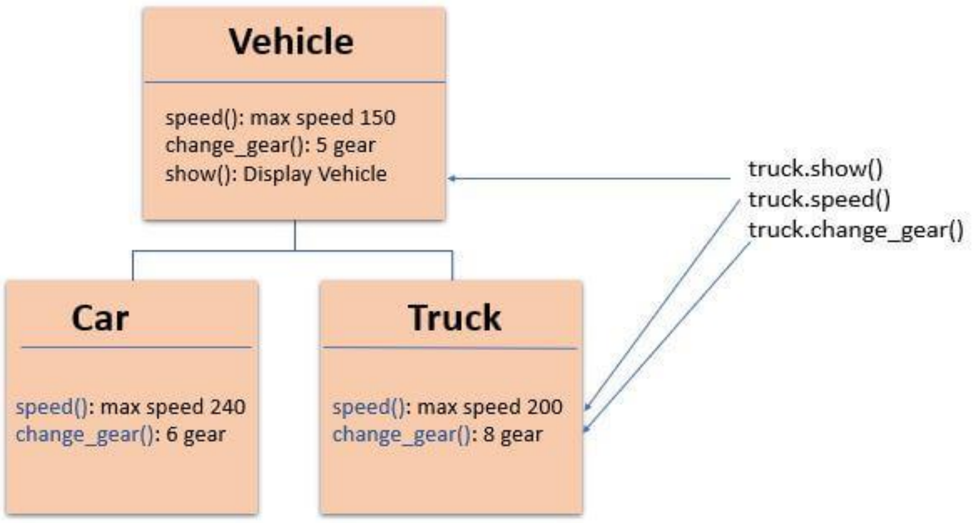
```
class Vehicle:
    def __init__(self, name, mileage, capacity):
```

```
                self.name = name
                self.mileage = mileage
                self.capacity = capacity
            def fare(self):
                return self.capacity * 100
        class Bus(Vehicle):
            pass
        School_bus = Bus("School Volvo", 12, 50)
        print("Total Bus fare is:", School_bus.fare())
```

NOTE : We need to access the parent class from inside a method of a child class.

Polymorphism

1. Polymorphism simply means having many forms.
2. In polymorphism, a method can **process objects differently depending on the class type or data type**.
3. The built-in function len() calculates the length of an object depending upon its type.
4. If an object is a string, it returns the count of characters, and If an object is a list, it returns the count of items in a list.
5. Polymorphism is mainly used with inheritance.

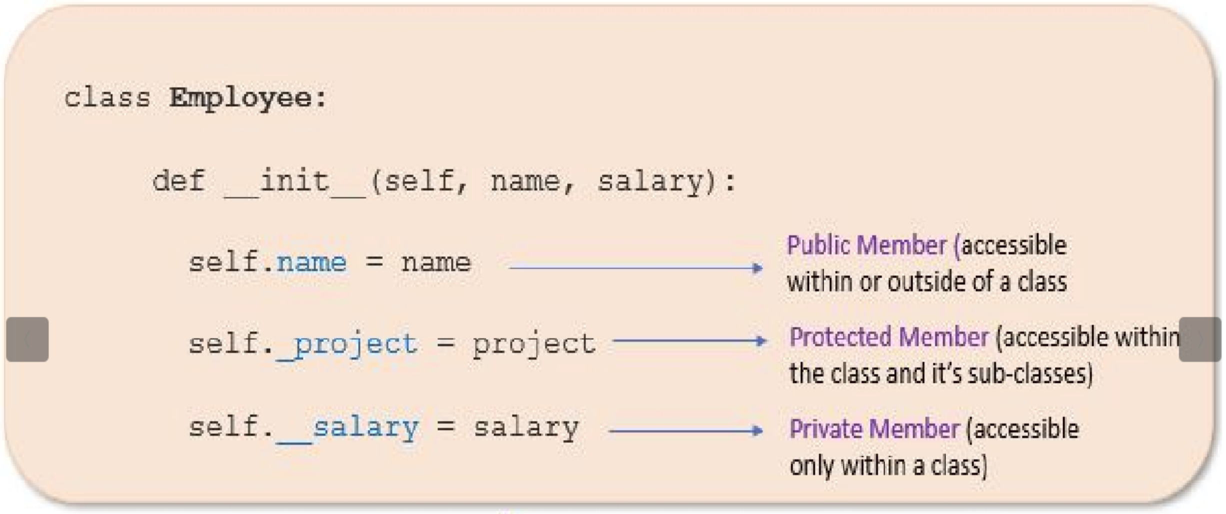


Method overridden in Car and Truck class

```
class Vehicle:
    def __init__(self, name, color, price):
        self.name = name
        self.color = color
        self.price = price
    def show(self):
        print('Details:', self.name, self.color, self.price)
    def max_speed(self):
        print('Vehicle max speed is 150')
    def change_gear(self):
        print('Vehicle change 6 gear')
# inherit from vehicle class
class Car(Vehicle):
    def max_speed(self):
        print('Car max speed is 240')
    def change_gear(self):
        print('Car change 7 gear')
# Car Object
car = Car('Car x1', 'Red', 20000)
car.show()
# calls methods from Car class
car.max_speed()
car.change_gear()
# Vehicle Object
vehicle = Vehicle('Truck x1', 'white', 75000)
vehicle.show()
# calls method from a Vehicle class
vehicle.max_speed()
vehicle.change_gear()
```

Encapsulation

1. Encapsulation describes the idea of wrapping data and the methods that work on data within one unit. This puts restrictions on accessing variables and methods directly and can prevent the accidental modification of data.
2. Encapsulation in Python describes the concept of bundling data and methods within a single unit.
3. Encapsulation allows us to restrict accessing variables and methods directly and prevent accidental data modification by creating private data members and methods within a class.
4. Encapsulation is a way to can restrict access to methods and variables from outside of class.
5. Encapsulation can be achieved by declaring the data members and methods of a class either as private or protected.
6. we don' t have direct access modifiers like public, private, and protected. We can achieve this by using single **underscore** and **double underscores**.
    - **Public Member**: Accessible anywhere from otside oclass.
    - **Private Member**: Accessible within the class
    - **Protected Member**: Accessible within the class and its sub-classes



```
class Employee:

    def __init__(self, name, salary):

      self.name = name
      self._project = project
      self.__salary = salary
```

Public Member (accessible within or outside of a class
Protected Member (accessible within the class and it's sub-classes)
Private Member (accessible only within a class)

7. **Public data** members are accessible within and outside of a class. All member variables of the class are by default public.

```
class Employee:
    # constructor
    def __init__(self, name, salary):
        # public data members
        self.name = name
        self.salary = salary
    # public instance methods
    def show(self):
        # accessing public data member
        print("Name: ", self.name, 'Salary:', self.salary)
# creating object of a class
emp = Employee('Jessa', 10000)
# accessing public data members
print("Name: ", emp.name, 'Salary:', emp.salary)
# calling public method of the class
```

8. We can protect variables in the class by marking them private. To define a private variable add two underscores as a prefix at the start of a variable name.
9. **Private members** are accessible only within the class, and we can't access them directly from the class objects.

```
class Employee:
    # constructor
    def __init__(self, name, salary):
        # public data member
        self.name = name
        # private member
        self.__salary = salary
# creating object of a class
emp = Employee('Jessa', 10000)
# accessing private data members
print('Salary:', emp.__salary)
```

10. We can access private members from outside of a class using the following two approaches
    - Create public method to access private members

```
class Employee:
    # constructor
    def __init__(self, name, salary):
        # public data member
        self.name = name
        # private member
        self.__salary = salary
    # public instance methods
    def show(self):
        # private members are accessible from a class
        print("Name: ", self.name, 'Salary:', self.__salary)
```

```
# creating object of a class
emp = Employee('Jessa', 10000)
# calling public method of the class
emp.show()
```

- Use name mangling

11. **Protected members** are accessible within the class and also available to its sub-classes. To define a protected member, prefix the member name with a single underscore _.
12. Protected data members are used when you implement inheritance and want to allow data members access to only child classes.

```
# base class
class Company:
    def __init__(self):
        # Protected member
        self._project = "NLP"
# child class
class Employee(Company):
    def __init__(self, name):
        self.name = name
        Company.__init__(self)
    def show(self):
        print("Employee name :", self.name)
        # Accessing protected member in child class
        print("Working on project :", self._project)
c = Employee("Jessa")
c.show()
# Direct access protected data member
print('Project:', c._project)
```

Advantages of Encapsulation

1. **Security:** The main advantage of using encapsulation is the security of the data. Encapsulation protects an object from unauthorized access. It allows private and protected access levels to prevent accidental data modification.
2. **Data Hiding:** The user would not be knowing what is going on behind the scene. They would only be knowing that to modify a data member, call the setter method. To read a data member, call the getter method. What these setter and getter methods are doing is hidden from them.
3. **Simplicity:** It simplifies the maintenance of the application by keeping classes separated and preventing them from tightly coupling with each other.
4. **Aesthetics:** Bundling data and methods within a class makes code more readable and maintainable