OOPs in Python

Class and Object

What Is Object-Oriented Programming?

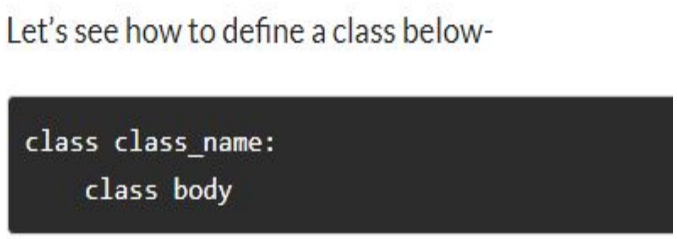
Object-Oriented Programming(OOP), is all about creating “objects”. An object is a group of interrelated variables and functions. These variables are often referred to as properties of the object and functions are referred to as the behavior of the objects. These objects provide a better and clear structure for the program.

For example, a car can be an object. If we consider the car as an object then its properties would be – its color, its model, its price, its brand, etc. And its behavior/function would be acceleration, slowing down, gear change.

Objects:

Objects are an instance of a class. It is an entity that has state and behavior.

- A class is a collection of objects
- When we define a class only the description or a blueprint of the object is created. There is no memory allocation until we create its object. The objector instance contains real data or information.
- Instantiation is nothing but creating a new object/instance of a class. Let's create the object of the above class we defined



Let's see how to define a class below-

```
class class_name:
    class body
```

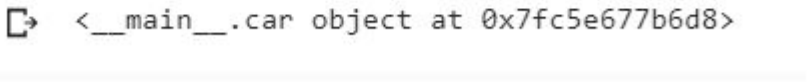
Consider the case of a car showroom. You want to store the details of each car. Let's start by defining a class first-

```
class Car:
    pass
```

```
obj1 = Car()
```

Try printing this object-

```
print(obj1)
```



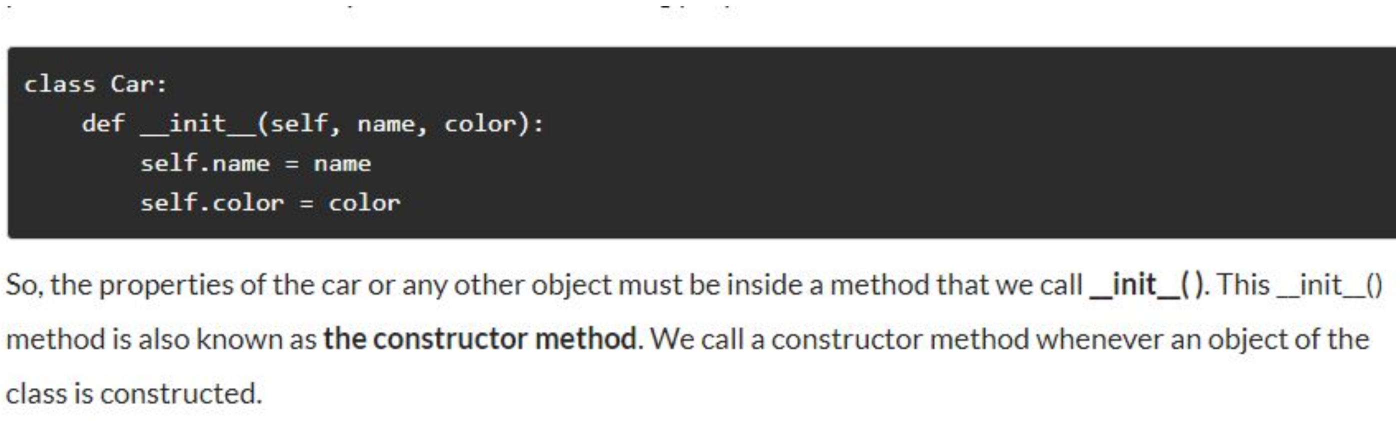
```
<__main__.car object at 0x7fc5e677b6d8>
```

Since our class was empty, it returns the address where the object is stored i.e 0x7fc5e677b6d8

Class Constructor

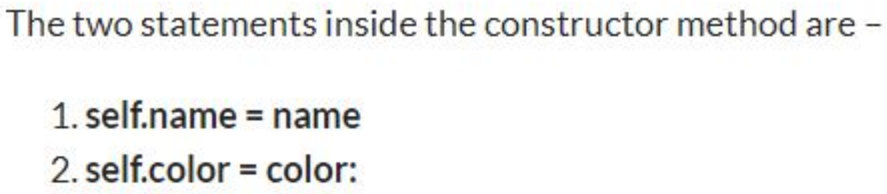
The job of the class constructor is to assign the values to the data members of the class when an object of the class is created.

There can be various properties of a car such as its name, color, model, brand name, engine power, weight, price, etc



```
class Car:
    def __init__(self, name, color):
        self.name = name
        self.color = color
```

So, the properties of the car or any other object must be inside a method that we call **__init__( )**. This __init__() method is also known as **the constructor method**. We call a constructor method whenever an object of the class is constructed.

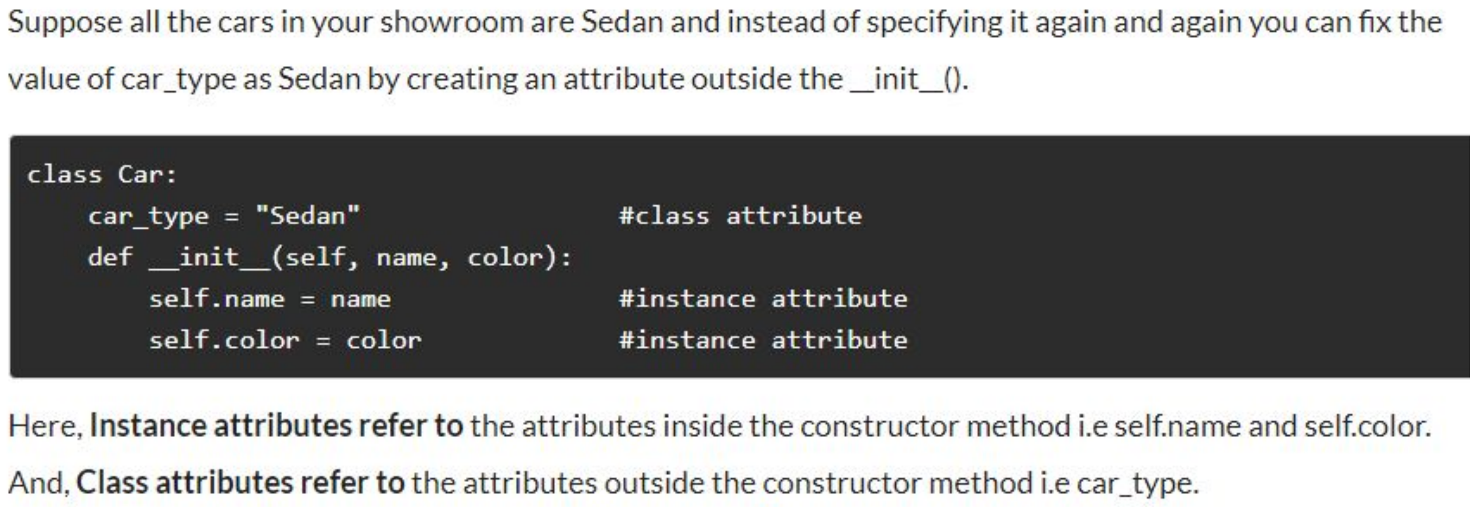


The two statements inside the constructor method are –

1. **self.name = name**
2. **self.color = color:**

This will create new attributes namely name and color and then assign the value of the respective parameters to them.

The “self” keyword represents the instance of the class. By using the “self” keyword we can access the attributes and methods of the class.



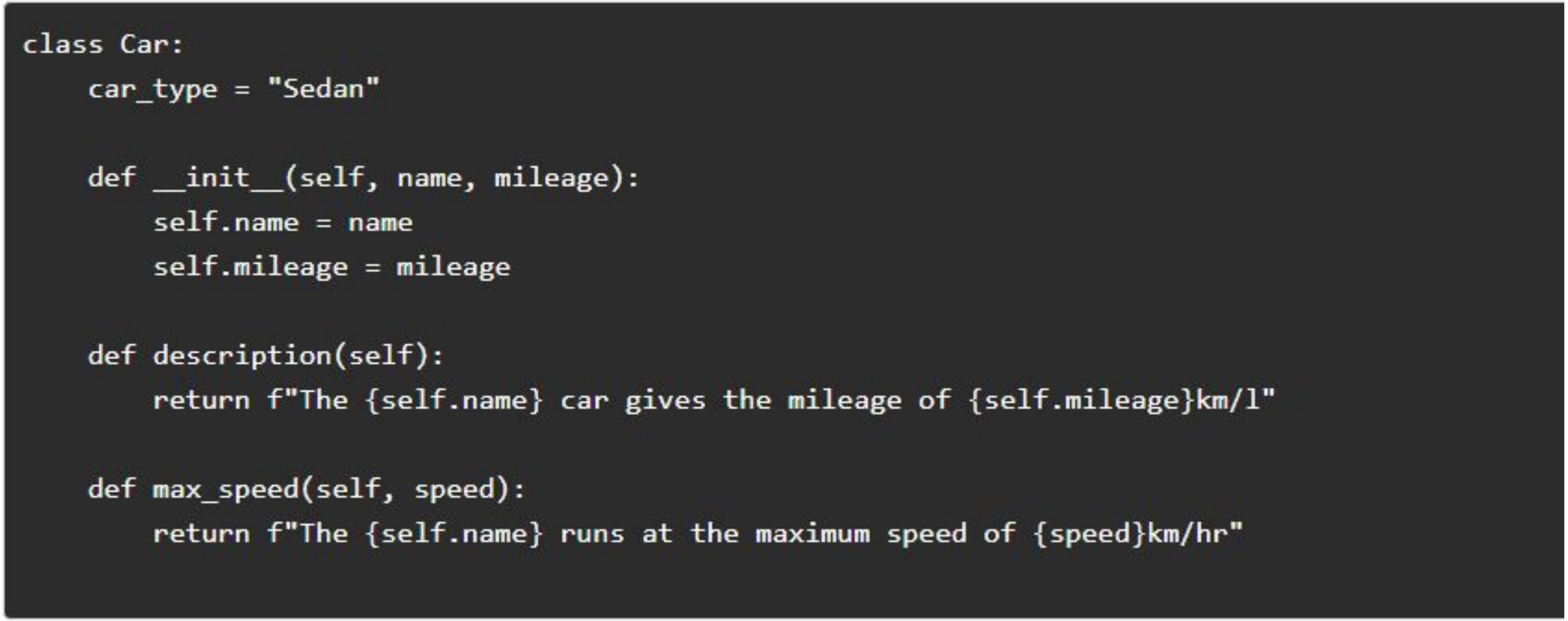
Suppose all the cars in your showroom are Sedan and instead of specifying it again and again you can fix the value of car_type as Sedan by creating an attribute outside the __init__().

```
class Car:
    car_type = "Sedan"                 #class attribute
    def __init__(self, name, color):
        self.name = name               #instance attribute
        self.color = color             #instance attribute
```

Here, **Instance attributes refer to** the attributes inside the constructor method i.e self.name and self.color.

And, **Class attributes refer to** the attributes outside the constructor method i.e car_type.

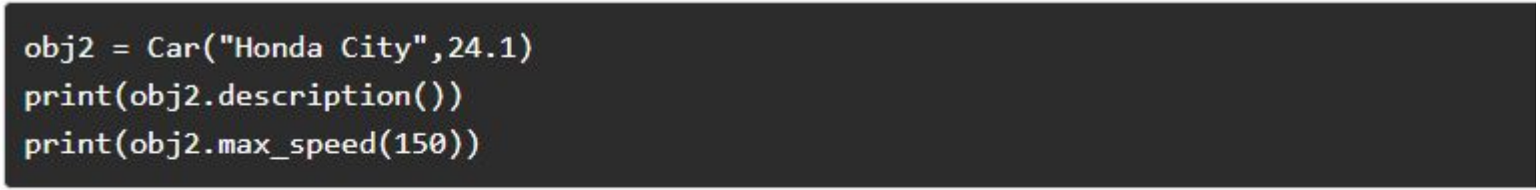
Class Methods



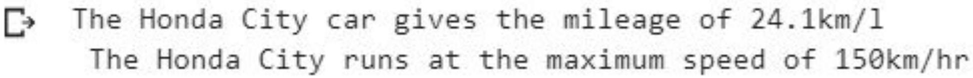
```
class Car:
    car_type = "Sedan"

    def __init__(self, name, mileage):
        self.name = name
        self.mileage = mileage

    def description(self):
        return f"The {self.name} car gives the mileage of {self.mileage}km/l"

    def max_speed(self, speed):
        return f"The {self.name} runs at the maximum speed of {speed}km/hr"
```

The methods defined inside a class other than the constructor method are known as the **instance** methods.

Let’s create an object for the class described in Car

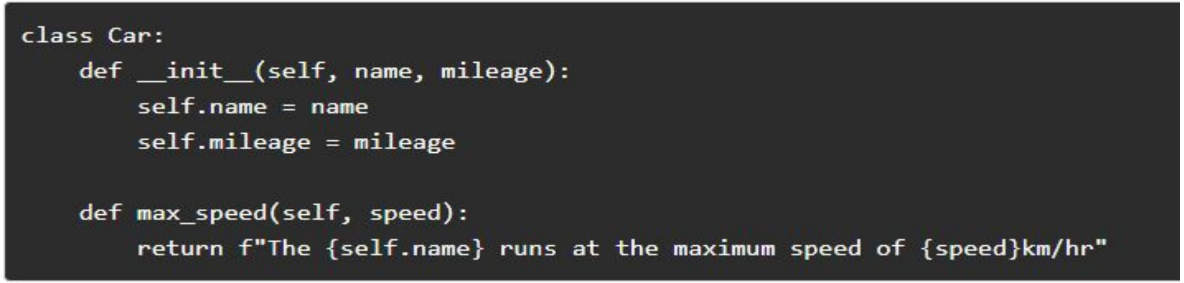


```
obj2 = Car("Honda City",24.1)
print(obj2.description())
print(obj2.max_speed(150))
```

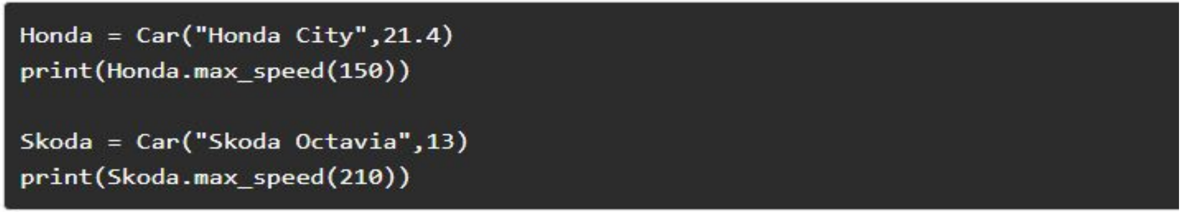


```
The Honda City car gives the mileage of 24.1km/l
 The Honda City runs at the maximum speed of 150km/hr
```

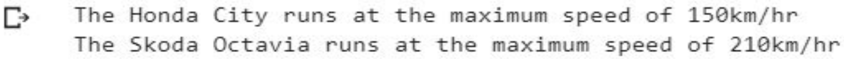
.Creating more than one object of a class



```
class Car:
    def __init__(self, name, mileage):
        self.name = name
        self.mileage = mileage

    def max_speed(self, speed):
        return f"The {self.name} runs at the maximum speed of {speed}km/hr"
```



```
Honda = Car("Honda City",21.4)
print(Honda.max_speed(150))

Skoda = Car("Skoda Octavia",13)
print(Skoda.max_speed(210))
```



```
The Honda City runs at the maximum speed of 150km/hr
The Skoda Octavia runs at the maximum speed of 210km/hr
```

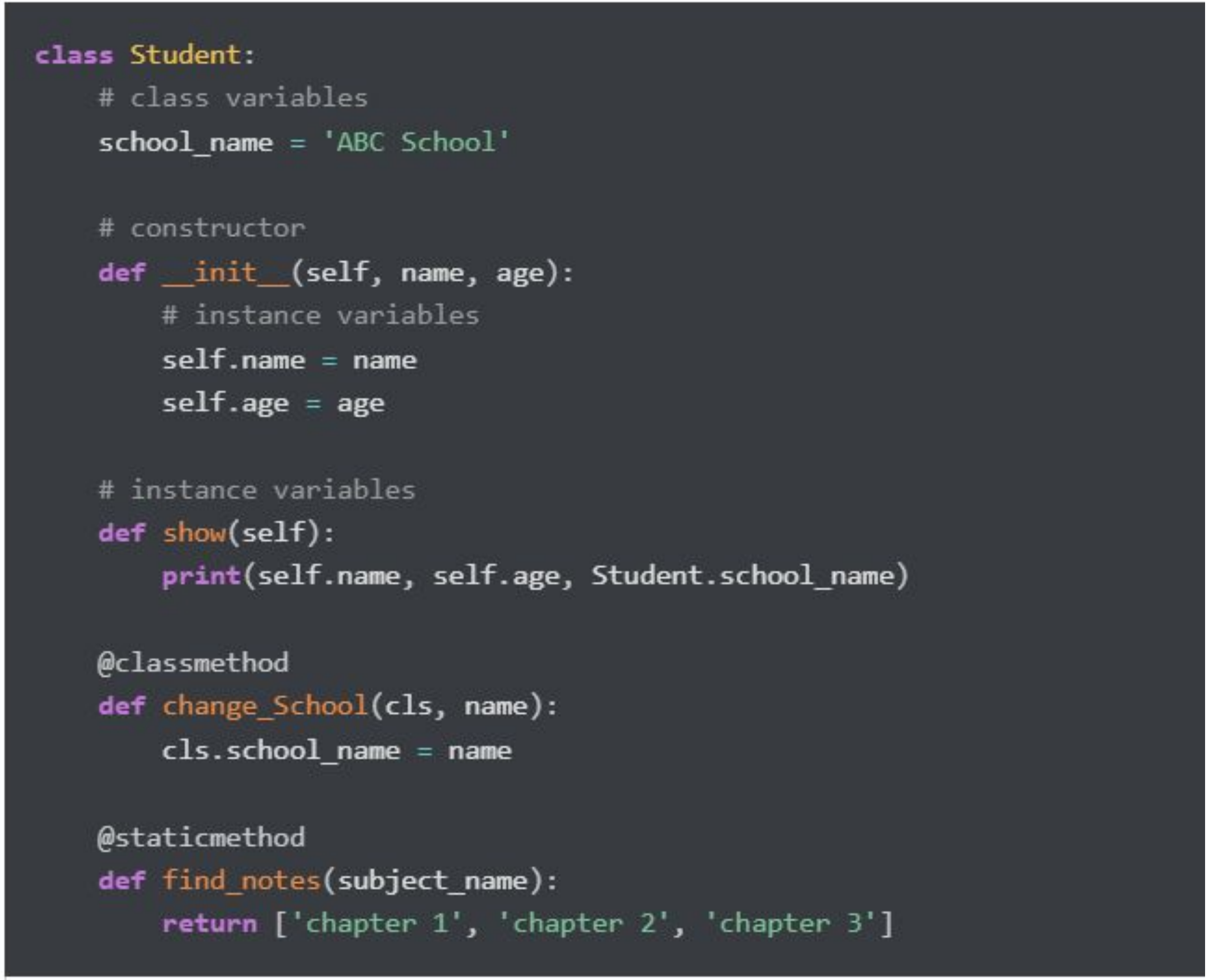
In Object-oriented programming, when we design a class, we use the following three methods

- Instance method performs a set of actions on the data/value provided by the instance variables. If we use instance variables inside a method, such methods are called instance methods.
- Class method is method that is called on the class itself, not on a specific object instance. Therefore, it belongs to a class level, and all class instances share a class method.
- Static method is a general utility method that performs a task in isolation. This method doesn’t have access to the instance and class variable

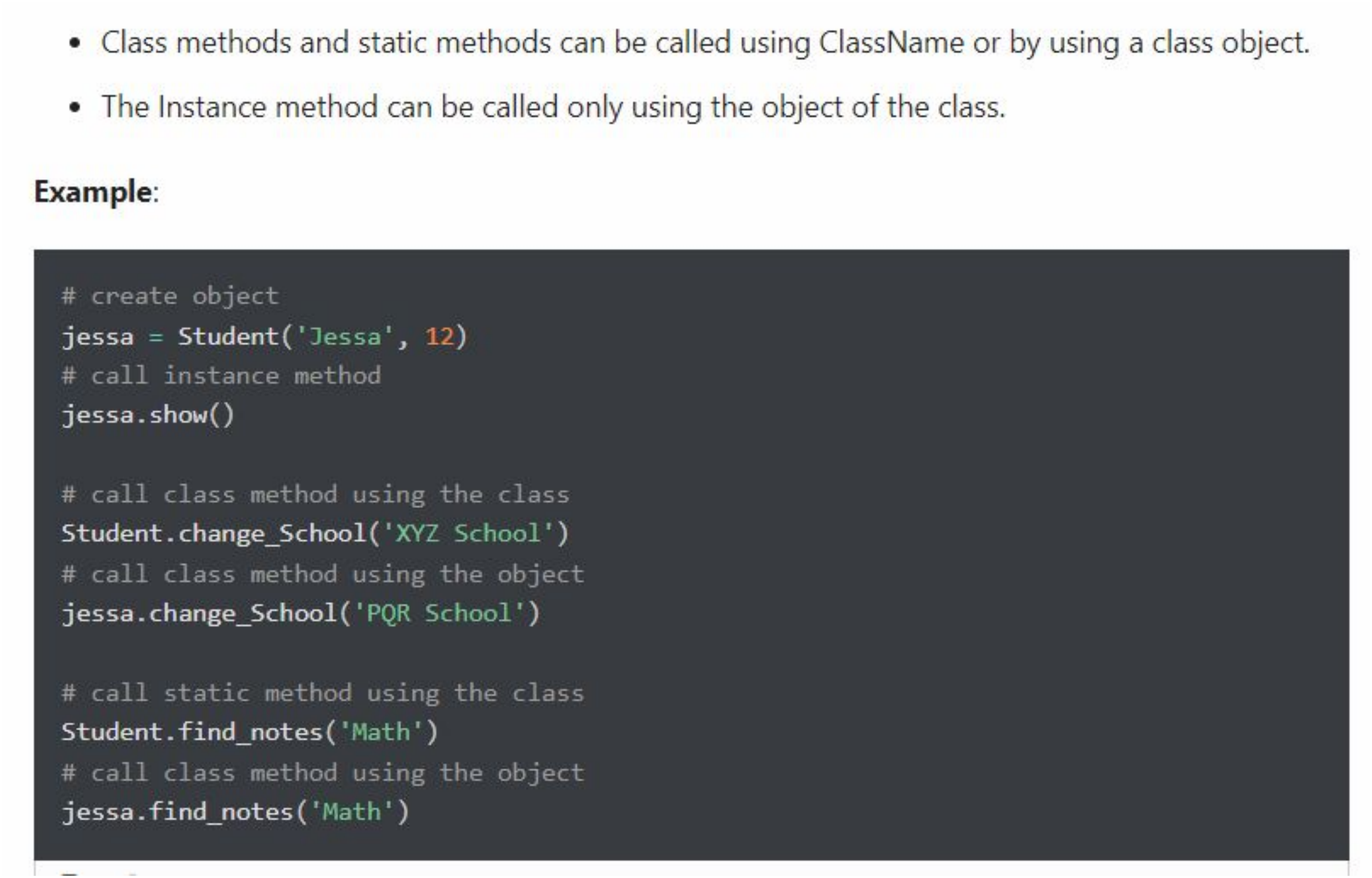
- All three methods are defined inside a class, and it is pretty similar to defining a regular function.
- Any method we create in a class will automatically be created as an instance method. We must explicitly tell Python that it is a class method or static method.
- Use the `@classmethod` decorator or the `classmethod()` function to define the class method
- Use the `@staticmethod` decorator or the `staticmethod()` function to define a static method.

Example:

- Use `self` as the first parameter in the instance method when defining it. The `self` parameter refers to the current object.
- On the other hand, Use `cls` as the first parameter in the class method when defining it. The `cls` refers to the class.
- A static method doesn't take instance or class as a parameter because they don't have access to the instance variables and class variables.



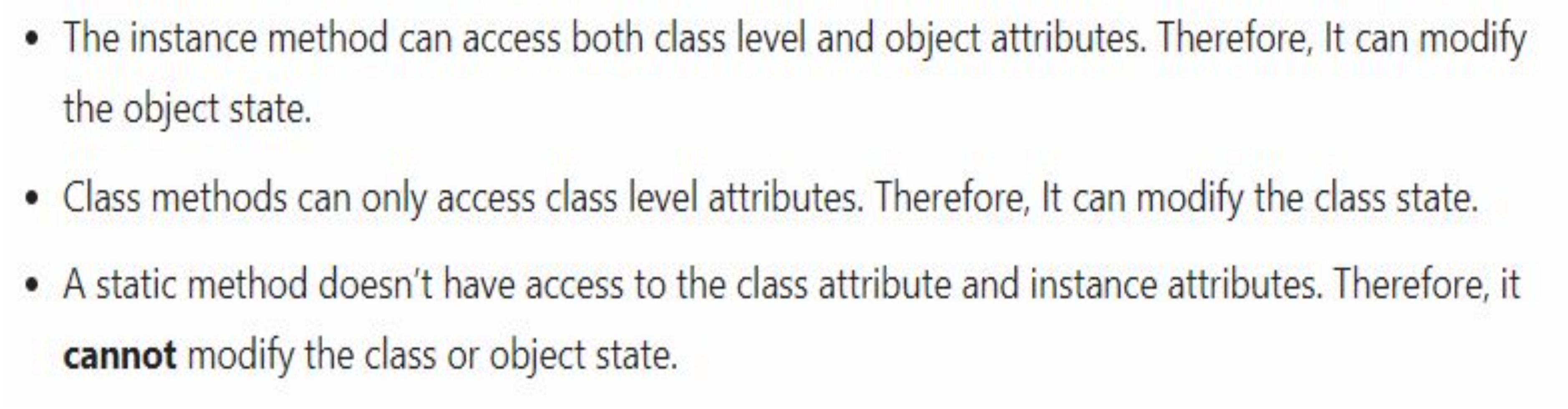
```python
class Student:
    # class variables
    school_name = 'ABC School'

    # constructor
    def __init__(self, name, age):
        # instance variables
        self.name = name
        self.age = age

    # instance variables
    def show(self):
        print(self.name, self.age, Student.school_name)

    @classmethod
    def change_School(cls, name):
        cls.school_name = name

    @staticmethod
    def find_notes(subject_name):
        return ['chapter 1', 'chapter 2', 'chapter 3']
```



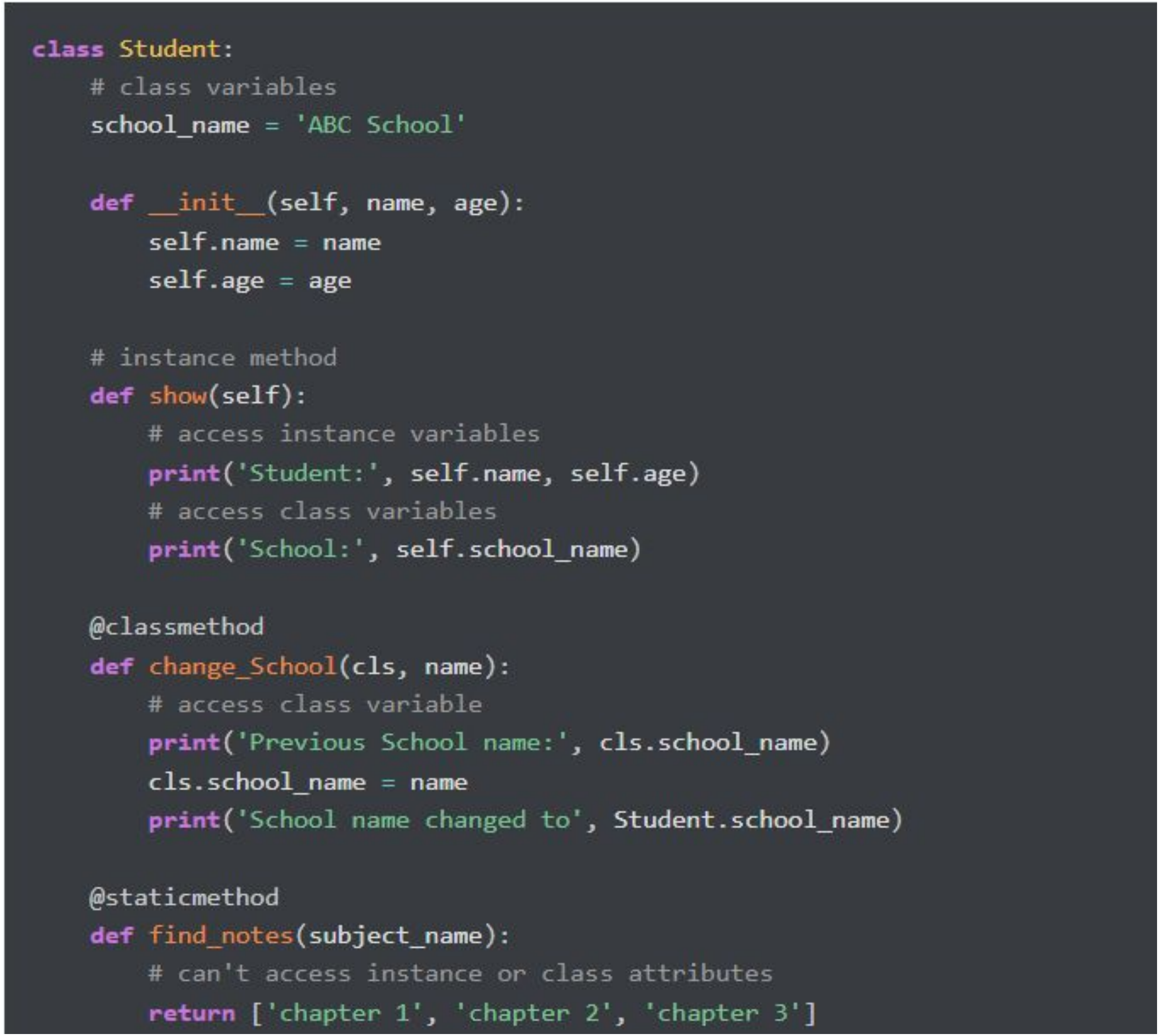
- Class methods and static methods can be called using ClassName or by using a class object.
- The Instance method can be called only using the object of the class.

Example:

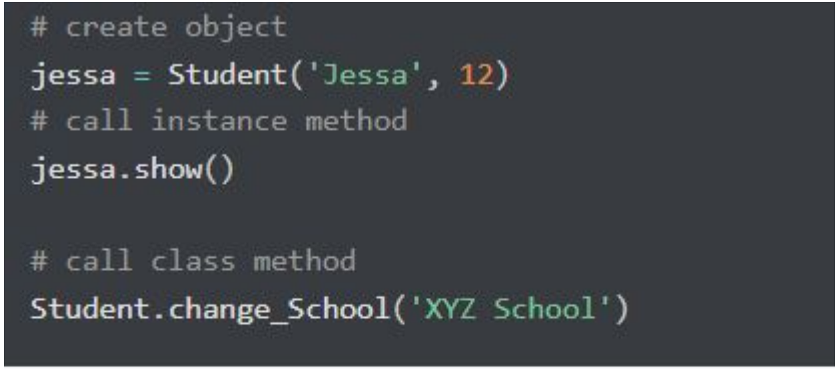
```
# create object
jessa = Student('Jessa', 12)
# call instance method
jessa.show()

# call class method using the class
Student.change_School('XYZ School')
# call class method using the object
jessa.change_School('PQR School')

# call static method using the class
Student.find_notes('Math')
# call class method using the object
jessa.find_notes('Math')
```



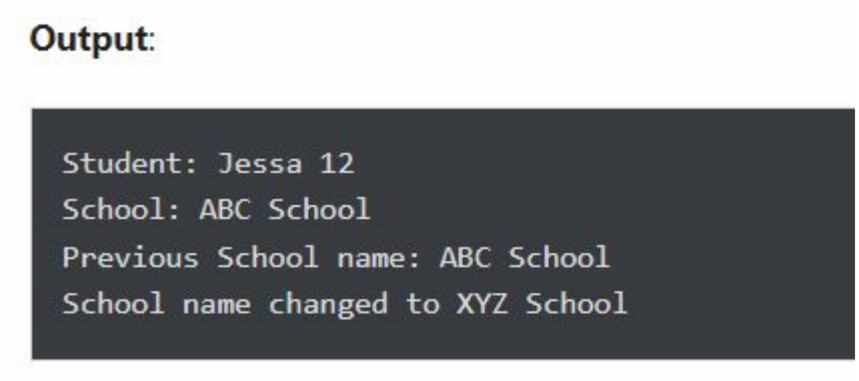
- The instance method can access both class level and object attributes. Therefore, It can modify the object state.
- Class methods can only access class level attributes. Therefore, It can modify the class state.
- A static method doesn’t have access to the class attribute and instance attributes. Therefore, it **cannot** modify the class or object state.



```python
class Student:
    # class variables
    school_name = 'ABC School'

    def __init__(self, name, age):
        self.name = name
        self.age = age

    # instance method
    def show(self):
        # access instance variables
        print('Student:', self.name, self.age)
        # access class variables
        print('School:', self.school_name)

    @classmethod
    def change_School(cls, name):
        # access class variable
        print('Previous School name:', cls.school_name)
        cls.school_name = name
        print('School name changed to', Student.school_name)

    @staticmethod
    def find_notes(subject_name):
        # can't access instance or class attributes
        return ['chapter 1', 'chapter 2', 'chapter 3']
```



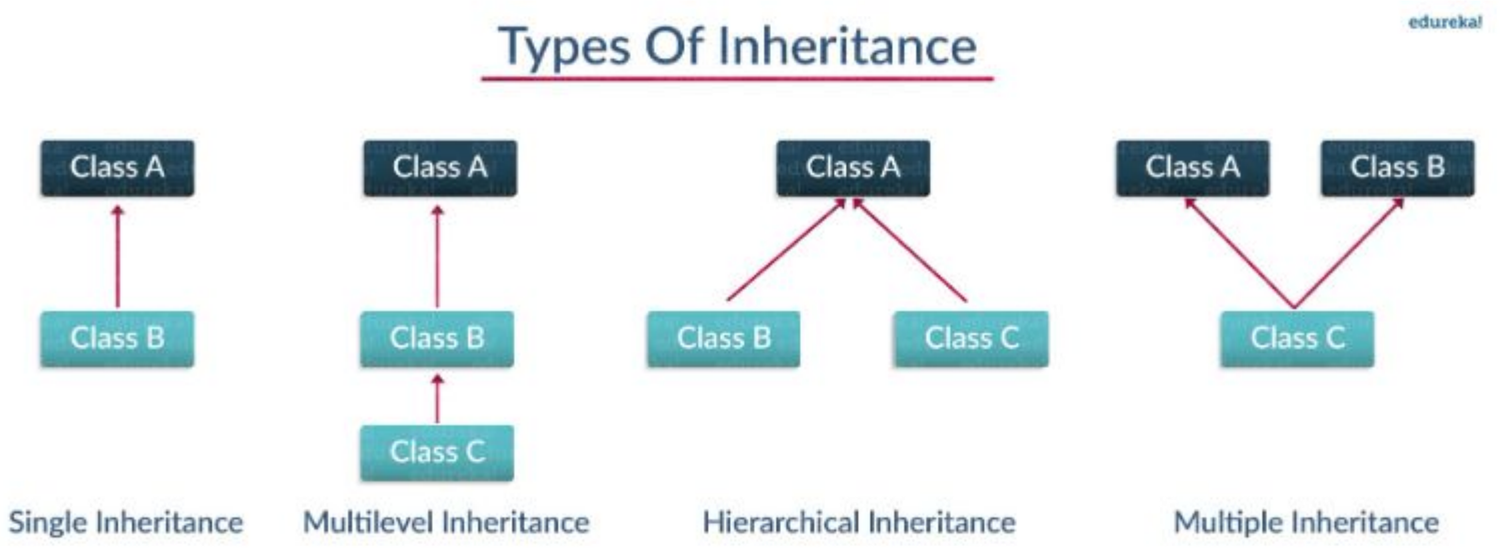
```
# create object
jessa = Student('Jessa', 12)
# call instance method
jessa.show()

# call class method
Student.change_School('XYZ School')
```



Output:

```
Student: Jessa 12
School: ABC School
Previous School name: ABC School
School name changed to XYZ School
```

Inheritance



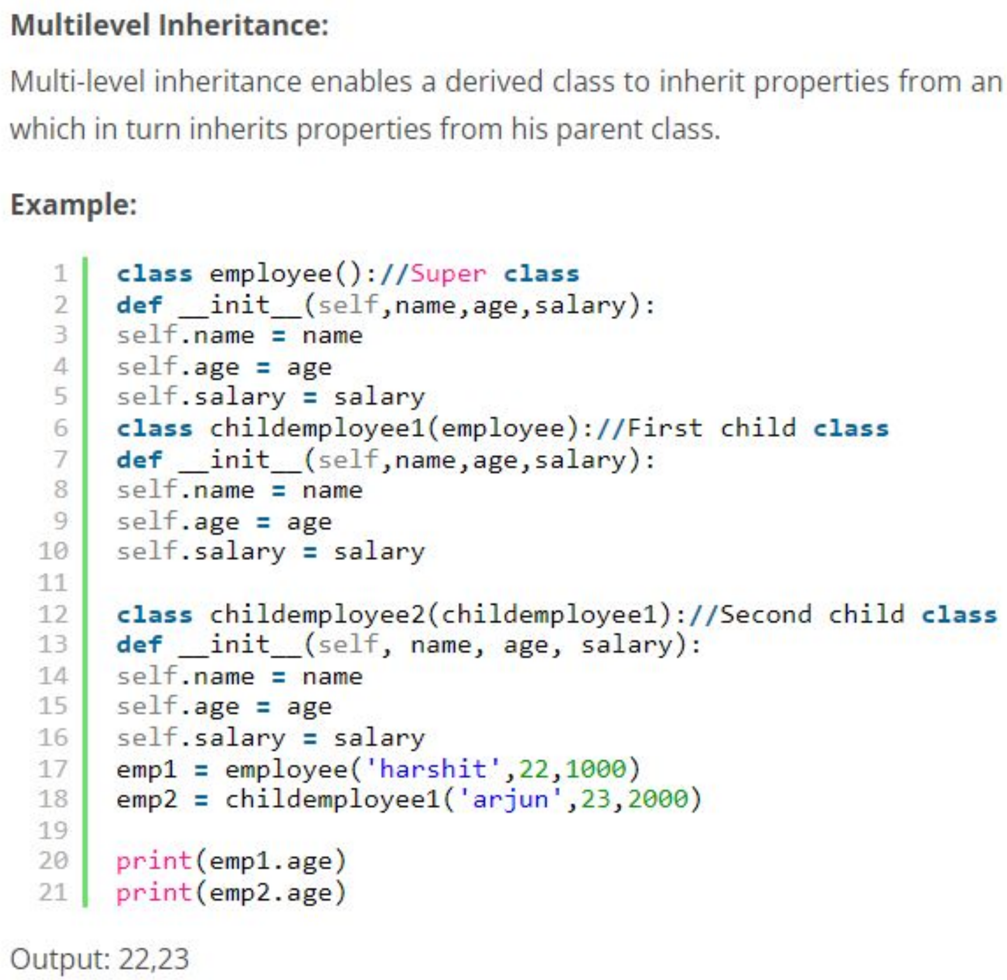
Single Inheritance:

Single level inheritance enables a derived class to inherit characteristics from a single parent class.

Example:

```
class employee1()://This is a parent class
def __init__(self, name, age, salary):
self.name = name
self.age = age
self.salary = salary

class childemployee(employee1)://This is a child class
def __init__(self, name, age, salary,id):
self.name = name
self.age = age
self.salary = salary
self.id = id
emp1 = employee1('harshit',22,1000)

print(emp1.age)
```

Output: 22



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.

Example:

```
class employee()://Super class
def __init__(self,name,age,salary):
self.name = name
self.age = age
self.salary = salary
class childemployee1(employee)://First child class
def __init__(self,name,age,salary):
self.name = name
self.age = age
self.salary = salary

class childemployee2(childemployee1)://Second child class
def __init__(self, name, age, salary):
self.name = name
self.age = age
self.salary = salary
emp1 = employee('harshit',22,1000)
emp2 = childemployee1('arjun',23,2000)

print(emp1.age)
print(emp2.age)
```

Output: 22,23

Explanation:

- It is clearly explained in the code written above, Here I have defined the superclass as employee and child class as **childemployee1**. Now, **childemployee1** acts as a parent for **childemployee2**.

- I have instantiated two objects ’emp1′ and ’emp2′ where I am passing the parameters “name”, “age”, “salary” for emp1 from superclass “employee” and “name”, “age, “salary” and “id” from the parent class “**childemployee1**”

Hierarchical Inheritance:

Hierarchical level inheritance enables more than one derived class to inherit properties from a parent class.

Example:

```
class employee():
def __init__(self, name, age, salary):     //Hierarchical Inheritance
self.name = name
self.age = age
self.salary = salary

class childemployee1(employee):
def __init__(self,name,age,salary):
self.name = name
self.age = age
self.salary = salary

class childemployee2(employee):
def __init__(self, name, age, salary):
self.name = name
self.age = age
self.salary = salary
emp1 = employee('harshit',22,1000)
emp2 = employee('arjun',23,2000)

print(emp1.age)
print(emp2.age)
```

Output: 22,23

Multiple Inheritance:

Multiple level inheritance enables one derived class to inherit properties from more than one base class.

Example:

```
class employee1()://Parent class
    def __init__(self, name, age, salary):
        self.name = name
        self.age = age
        self.salary = salary

class employee2()://Parent class
    def __init__(self,name,age,salary,id):
     self.name = name
     self.age = age
     self.salary = salary
     self.id = id

class childemployee(employee1,employee2):
    def __init__(self, name, age, salary,id):
     self.name = name
     self.age = age
     self.salary = salary
     self.id = id
emp1 = employee1('harshit',22,1000)
emp2 = employee2('arjun',23,2000,1234)

print(emp1.age)
print(emp2.id)
```

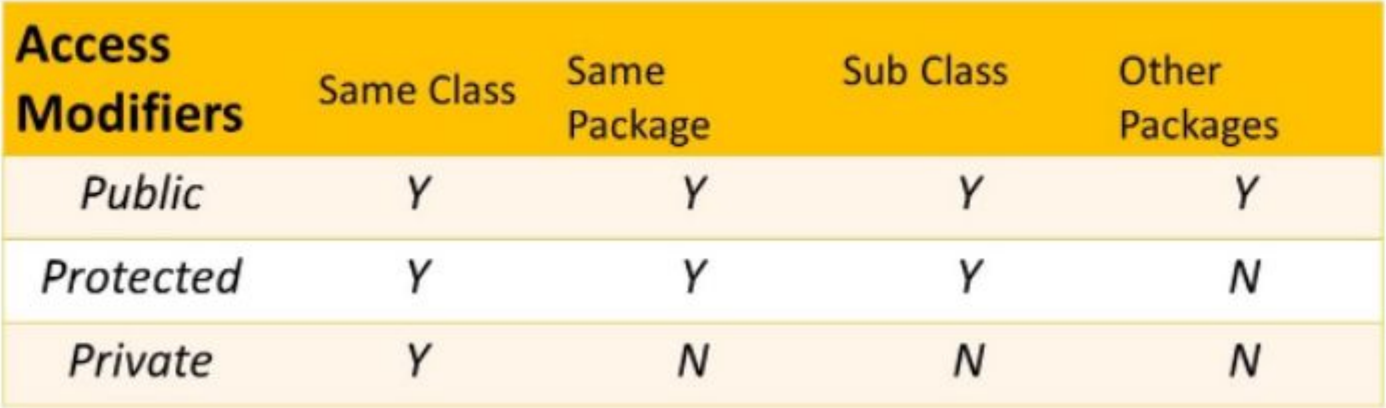
Output: 22,1234

Access Specifiers

Access Modifiers: **Access specifiers or access modifiers in python programming are used to limit the access of class variables and class methods outside of class while implementing the concepts of inheritance. This can be achieved by: Public, Private and Protected keyword.**

We can easily inherit the properties or behaviour of any class using the concept of inheritance. But some classes also holds the data (class variables and class methods) that we don't want other classes to inherit. So, to prevent that data we used access specifiers in python.

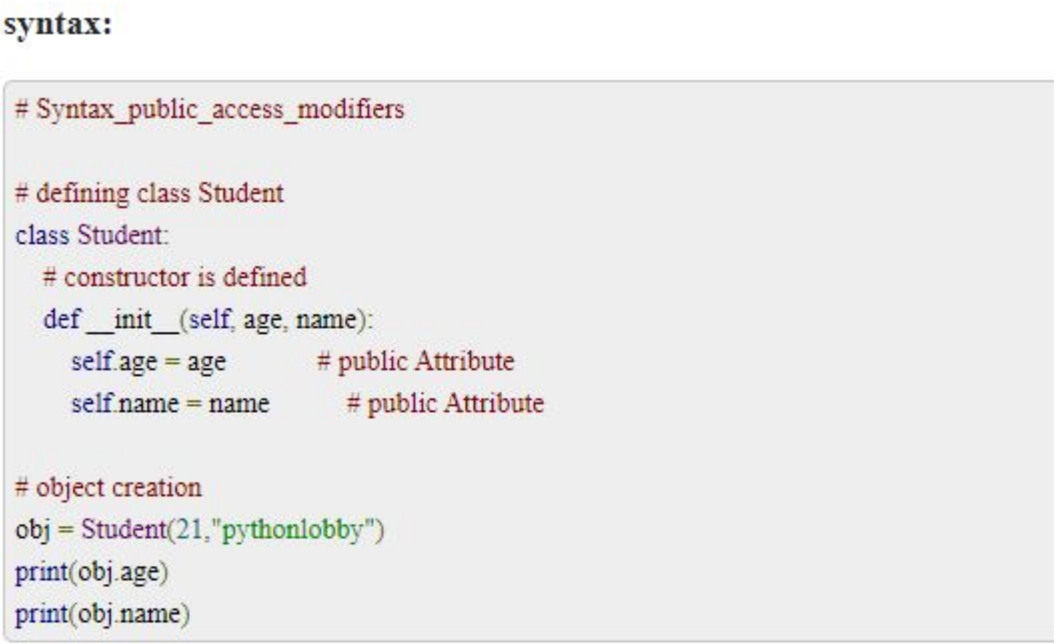
Note: Access modifiers in python are very helpful when we are using the concepts of inheritance. We can also apply the concept of access modifiers to class methods.



Access Modifiers	Same Class	Same Package	Sub Class	Other Packages
Public	*Y*	*Y*	*Y*	*Y*
Protected	*Y*	*Y*	*Y*	*N*
Private	*Y*	*N*	*N*	*N*

Public Access Modifier in Python

All the variables and methods (member functions) in python are by default public. Any instance variable in a class followed by the ‘self’ keyword ie. self.var_name are public accessed.



syntax:

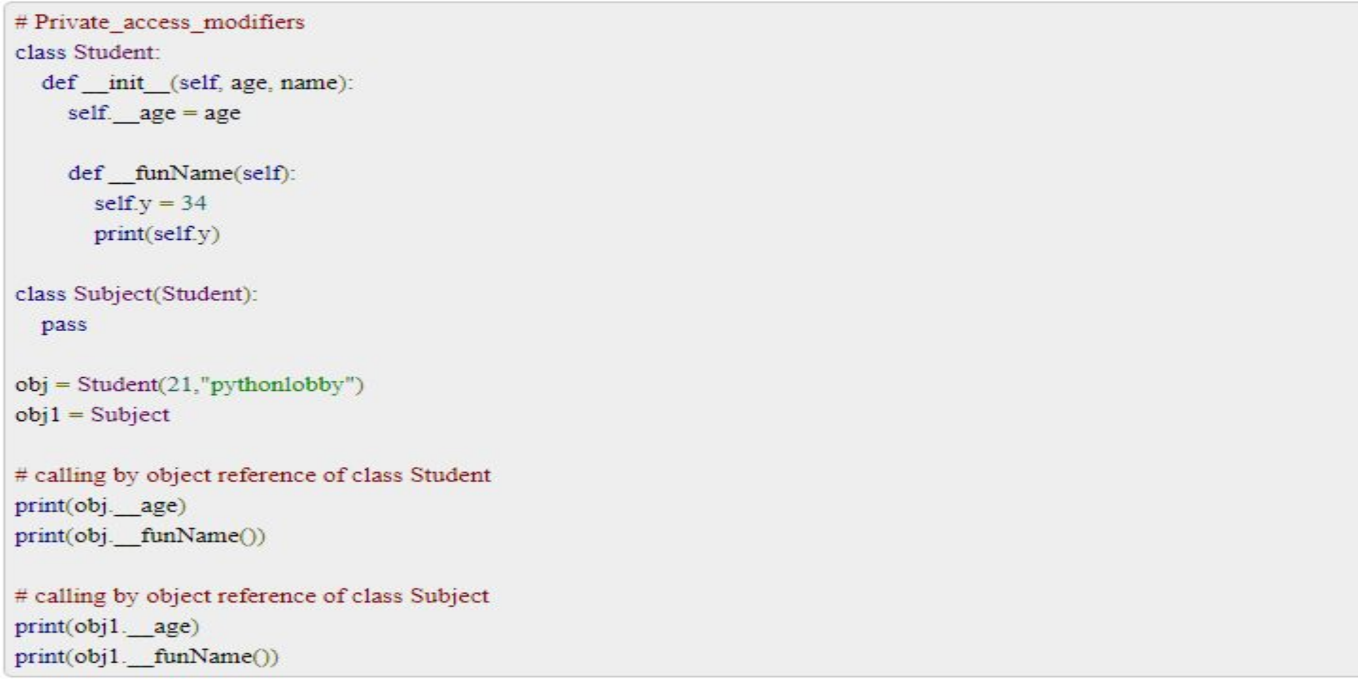
```
# Syntax_public_access_modifiers

# defining class Student
class Student:
    # constructor is defined
    def __init__(self, age, name):
        self.age = age          # public Attribute
        self.name = name          # public Attribute

# object creation
obj = Student(21,"pythonlobby")
print(obj.age)
print(obj.name)
```

Private Access Modifier

Private members of a class (variables or methods) are those members which are only accessible inside the class. We cannot use private members outside of class.

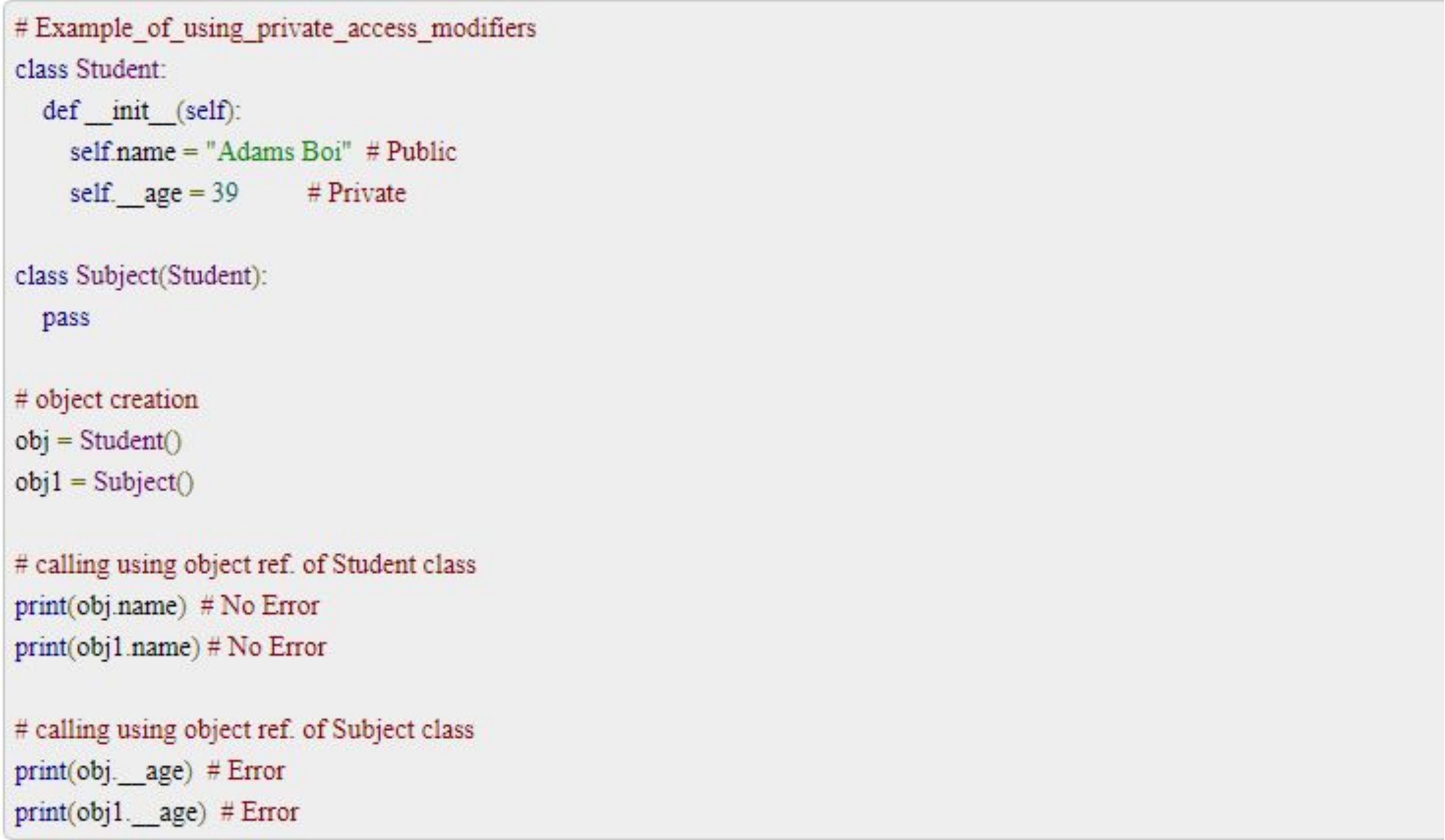
It is also not possible to inherit the private members of any class (parent class) to derived class (child class). Any instance variable in a class followed by `self` keyword and the variable name starting with double underscore ie. self.__varName are the private accessed member of a class.

Syntax:



```
# Private_access_modifiers
class Student:
   def __init__(self, age, name):
      self.__age = age

      def __funName(self):
         self.y = 34
         print(self.y)

class Subject(Student):
   pass

obj = Student(21,"pythonlobby")
obj1 = Subject

# calling by object reference of class Student
print(obj.__age)
print(obj.__funName())

# calling by object reference of class Subject
print(obj1.__age)
print(obj1.__funName())
```

Example 2:



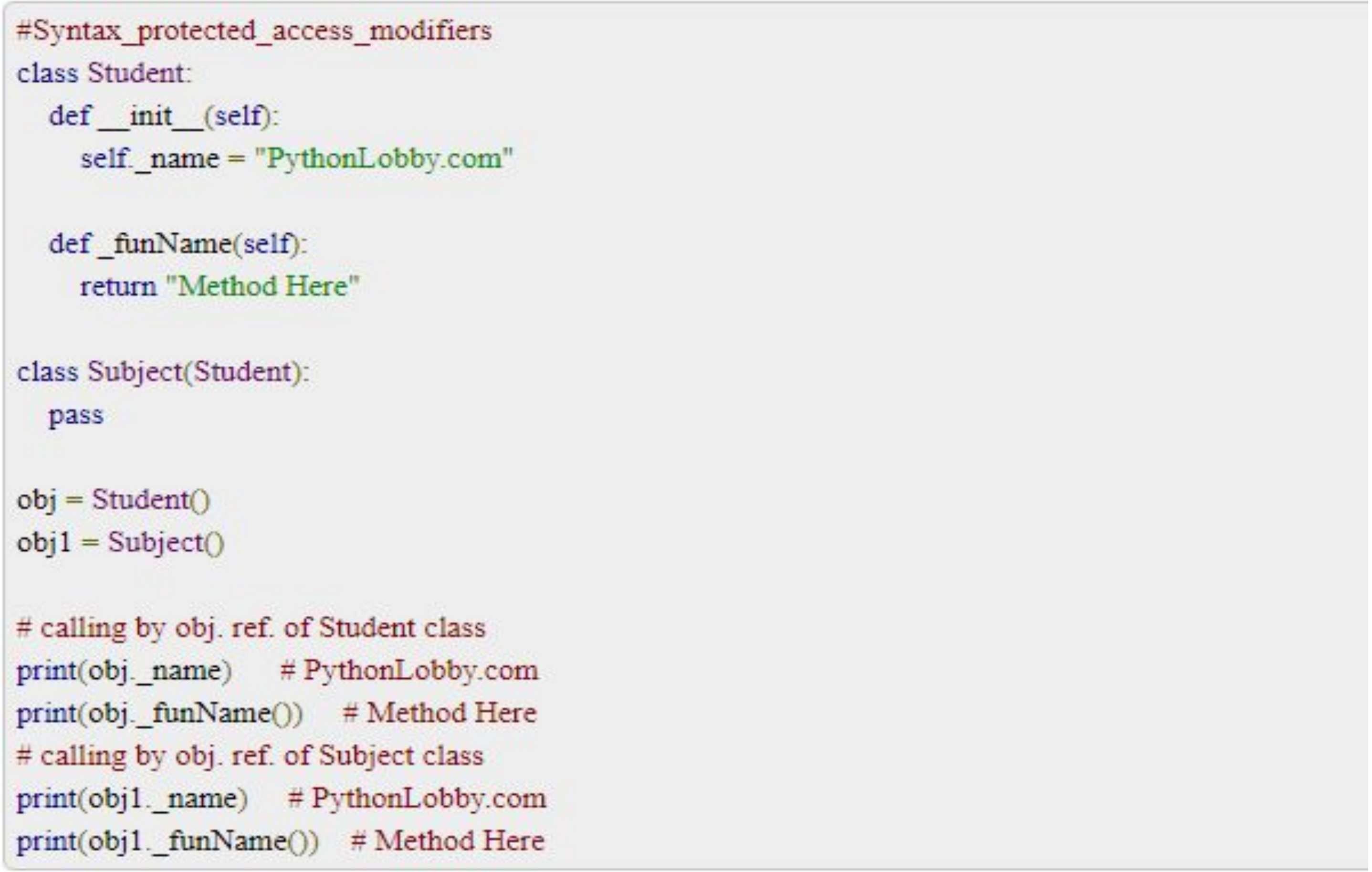
```
# Example_of_using_private_access_modifiers
class Student:
    def __init__(self):
        self.name = "Adams Boi"  # Public
        self.__age = 39         # Private

class Subject(Student):
    pass

# object creation
obj = Student()
obj1 = Subject()

# calling using object ref. of Student class
print(obj.name)  # No Error
print(obj1.name) # No Error

# calling using object ref. of Subject class
print(obj.__age)  # Error
print(obj1.__age)  # Error
```

Protected Access Modifier

Protected variables or we can say protected members of a class are restricted to be used only by the member functions and class members of the same class. And also it can be accessed or inherited by its derived class (child class). We can modify the values of protected variables of a class. The syntax we follow to make any variable protected is to write variable name followed by a single underscore (_) ie. _varName.

- Note:		We can access protected members of class outside of class even we can modify its value also. Now the doubt that arises is, public access modifiers follow the same except its syntax. Actually, protected access modifiers are designed so that responsible programmer would identify by their name convention and do the required operation only on that protected class members or class methods.

Syntax and Example 3:



```
#Syntax_protected_access_modifiers
class Student:
    def __init__(self):
        self._name = "PythonLobby.com"

    def _funName(self):
        return "Method Here"

class Subject(Student):
    pass

obj = Student()
obj1 = Subject()

# calling by obj. ref. of Student class
print(obj._name)     # PythonLobby.com
print(obj._funName())     # Method Here
# calling by obj. ref. of Subject class
print(obj1._name)     # PythonLobby.com
print(obj1._funName())    # Method Here
```