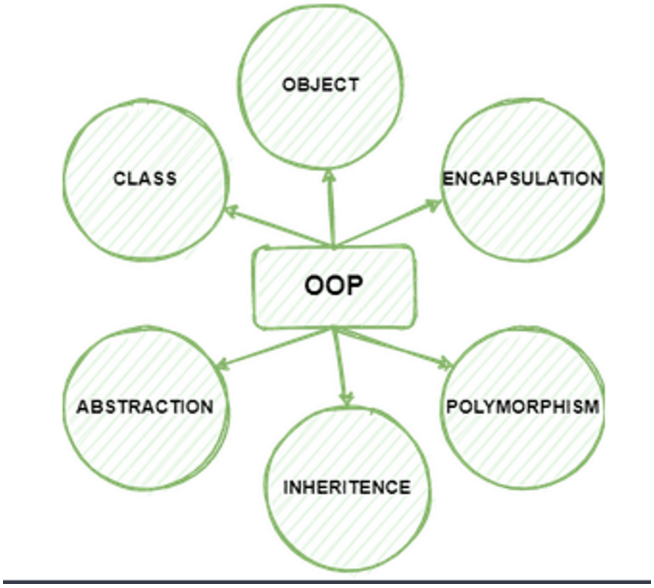
OOPS

07 July 2023 21:38



OOPs Concepts in Python

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

1. **What is OOPS?**

   Object-oriented programming provides a means of structuring programs so that properties and behaviors are bundled into individual**objects**.

   Python is a versatile programming language that supports various programming styles, including object-oriented programming (OOP) through the use of**objects**and**classes**.

2. **What is CLASS and OBJECTS?**

   An **object** is any entity that has**attributes/properties**and**behaviors**.

   For example,

   a parrot is an object. It has

   - **attributes**- name, age, color, etc.
   - **behavior**- dancing, singing, etc.

   a person is an object. It has

   - **attributes**- name, age, address , etc.
   - **behavior**- walking, talking, breathing, and running, etc.

   a email is an object. It has

   - **attributes**- recipient list, subject, and body , etc.
   - **behavior**- adding attachments and sending, etc.

A **class** is a blueprint for that object.

For example, let's say you want to track employees in an organization. You need to store some basic information about each employee, such as their name, age, position, and the year they started working.

One way to do this is to represent each employee as a list:

```
kirk=["James Kirk",34,"Captain",2265]
spock=["Spock",35,"Science Officer",2254]
mccoy=["Leonard McCoy","Chief Medical Officer",2266]
```

There are a number of issues with this approach.

- First, it can make larger code files more difficult to manage. If you reference `kirk[0]` several lines away from where the `kirk` list is declared, will you remember that the element with index `0` is the employee's name?
- Second, it can introduce errors if not every employee has the same number of elements in the list. In the `mccoy` list above, the age is missing, so `mccoy[1]` will return `"Chief Medical Officer"` instead of Dr. McCoy's age.

A great way to make this type of code more manageable and more maintainable is to use **classes**.

- **Creating EMPTY CLASS**

```
class parrot:
    pass

obj=parrot()
```

- Creating class with variables and accessing variables using objects

```
class Parrot:

    # class attribute
    name = ""
    age = 0

# create parrot1 object
parrot1 = Parrot()
parrot1.name = "Blu"
parrot1.age = 10

# create another object parrot2
parrot2 = Parrot()
parrot2.name = "Woo"
parrot2.age = 15

# access attributes
print(f"{parrot1.name} is {parrot1.age} years old")
print(f"{parrot2.name} is {parrot2.age} years old")
```

- Class and instance

Class - blue print or defines the data structure or how something should be defined
Instance - an **instance** is an object that is built from a class and contains real data,
Creating a new object from a class is called **instantiating** an object

```
class parrot:
   pass


obj1 = parrot()
print(obj1)

obj2 = parrot()
print(obj2)
```

Constructor construct the class attribute at the run time. Initiate the instance

- **self** represents the instance of the class. By using the "self" we can access the attributes and methods of the class in python. It binds the attributes with the given arguments.
- Self is always pointing to Current Object.

```
class check:
   def __init__(self):
      print("Address of self = ", id(self))

obj = check()
print("Address of class object = ", id(obj))
obj1 = check()
print("Address of class object = ", id(obj1))
```

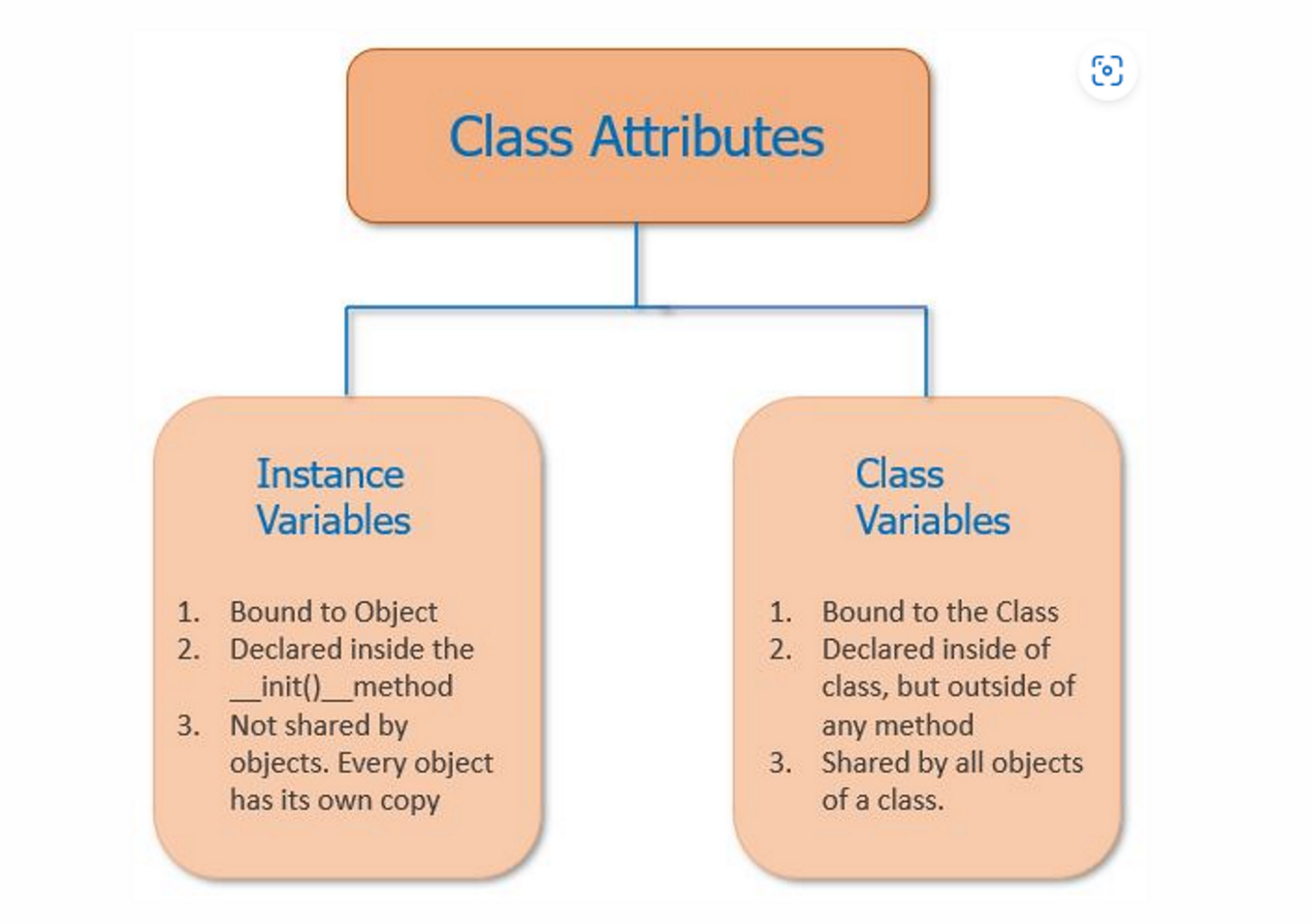
- Self is the first argument to be passed in Constructor and Instance Method.

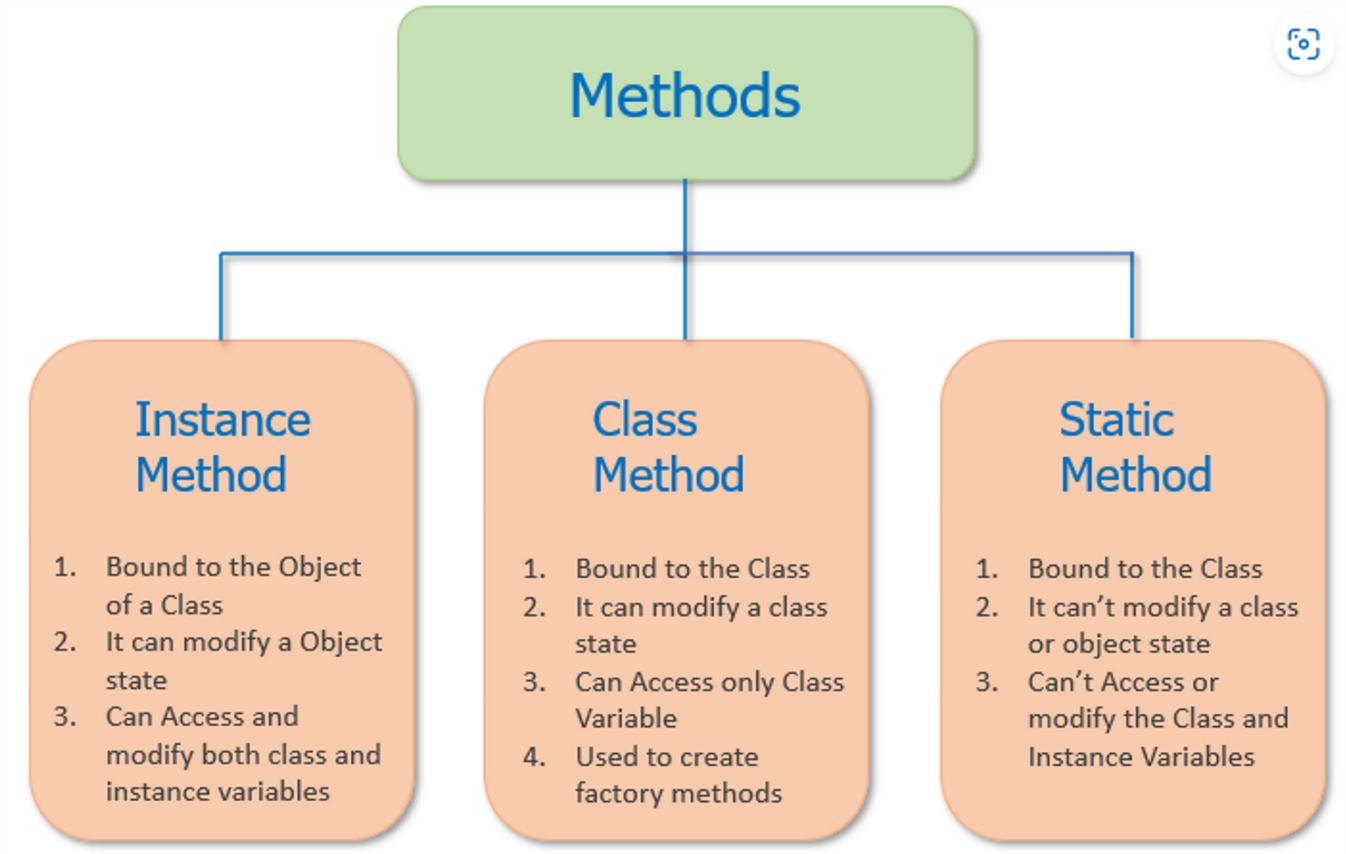
Self is a convention and not a Python keyword .
self is parameter in Instance Method and user can use another parameter name in place of it. But it is advisable to use self because it increases the readability of code, and it is also a good programming practice.

```
# Write Python3 code here

class this_is_class:
def __init__(in_place_of_self):
print("we have used another "
"parameter name in place of self")

object = this_is_class()
```

```
class Parrot:
   # class attribute
   name = ""
   age = 0

   def __init__(self, name, age):
```

```
        self.age=age
        self.name=name
        print(f"executed at the time of object creation or instantiation")
        print(f"{self.name} is {self.age} years old")

# create parrot1 object
parrot1 = Parrot("Blu1", 20)
parrot1.name = "Blu"
parrot1.age = 3

# create another object parrot2
parrot2 = Parrot("woo1", 20)
parrot2.name = "Woo"
parrot2.age = 15

# access attributes
print("executed after class var are executed")
print(f"{parrot1.name} is {parrot1.age} years old")
print(f"{parrot2.name} is {parrot2.age} years old")
```





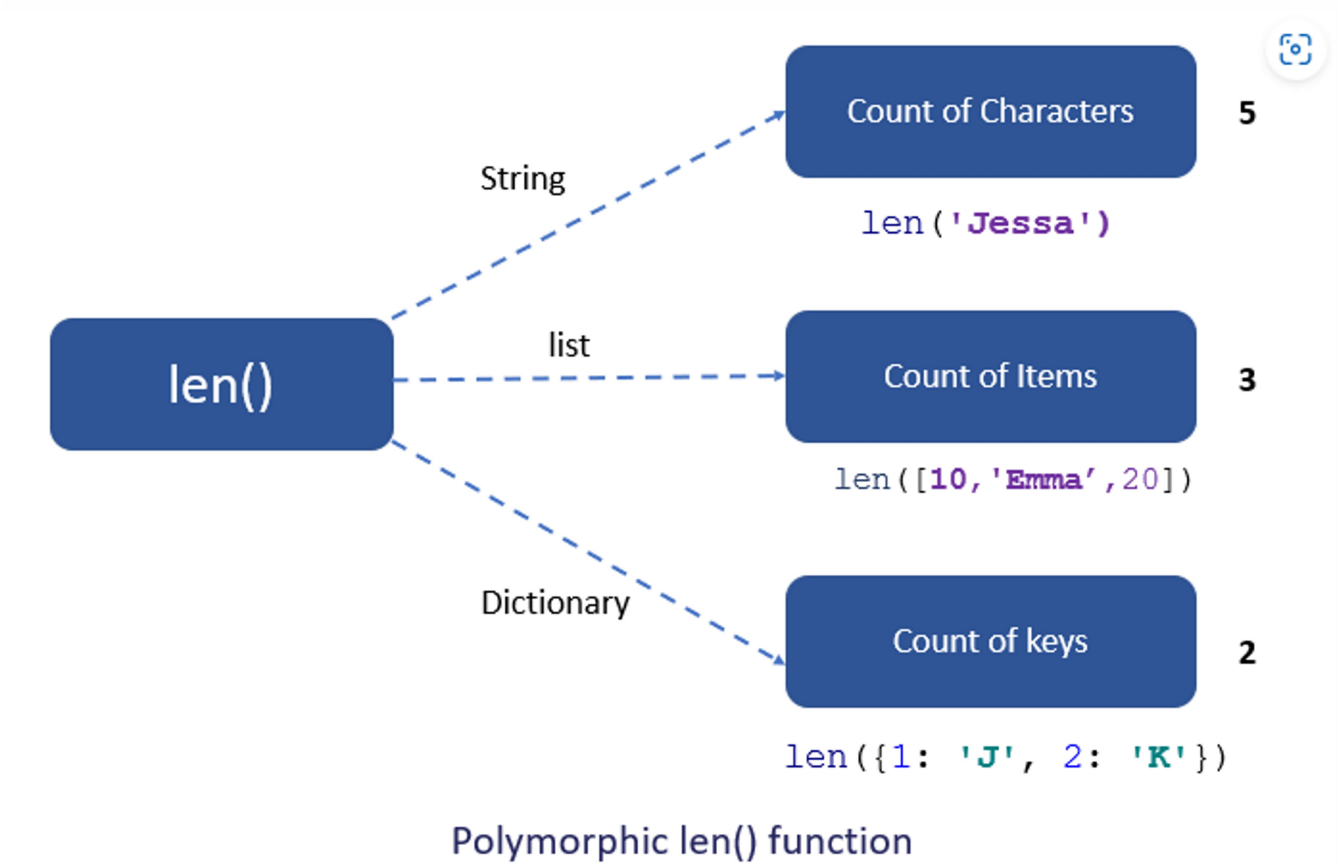
class method vs static method vs instance method

```
# class methods demo
class Student:
    # class variable
    school_name = 'ABC School'

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

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

    # instance method
    def change_age(self, new_age):
        # modify instance variable
        self.age = new_age

    # class method
    @classmethod
```

```
    def modify_school_name(cls, new_name):
        # modify class variable
        cls.school_name = new_name

s1 = Student("Harry", 12)

# call instance methods
s1.show()
s1.change_age(14)

# call class method
Student.modify_school_name('XYZ School')
# call instance methods
s1.show()
```

3. What is Polymorphism?

Polymorphism in Python is the ability of anobjectto take many forms
Example:



Polymorphic len() function

Polymorphic len() function

Polymorphism With Inheritance

Polymorphism is mainly used with inheritance. In**inheritance**, child class inherits the attributes and methods of a parent class. The existing class is called a base class or parent class, and the new class is called a subclass or child class or derived class.

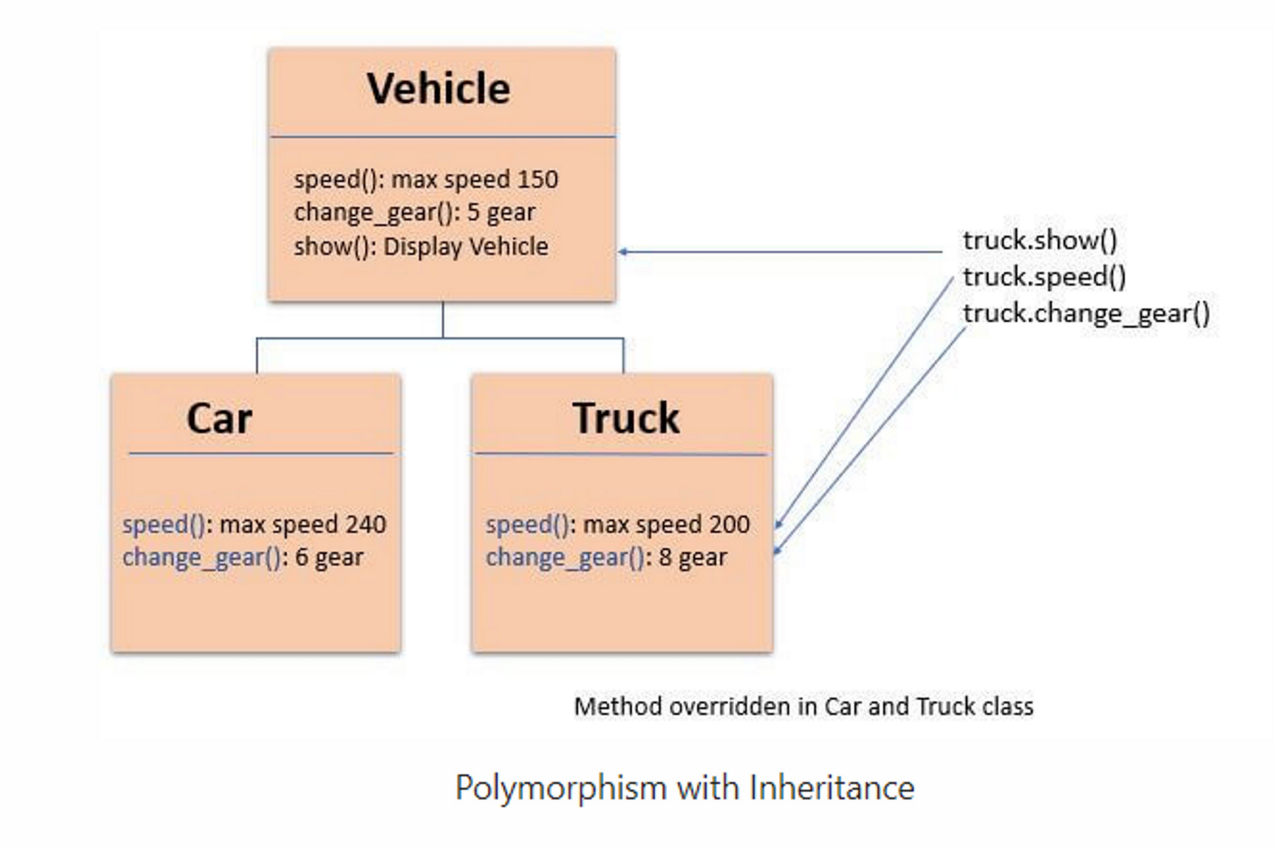
Using**method overriding**polymorphism allows us to defines methods in the childclassthat have the same name as the methods in the parent class. This**process of re-implementing the inherited method in the child class**is known as Method Overriding.

Advantage of method overriding

- It is effective when we want to extend the functionality by altering the inherited method. Or the method inherited from the parent class doesn't fulfill the need of a child class, so we need to re-implement the same method in the child class in a different way.
- Method overriding is useful when a parent class has multiple child classes, and one of that child class wants to redefine the method. The other child classes can use the parent class method. Due to this, we don't need to modification the parent class code

In polymorphism,**Python first checks the object's class type and executes the appropriate method**when we call the method. For example, If you create the Car object, then Python calls the`speed()`method from a Car class.

Let's see how it works with the help of an example.



Polymorphism with Inheritance

```
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()
```

Method Overloading

The process of calling the same method with different parameters is known as method overloading. Python does not support method overloading. Python considers only the latest defined method even if you overload the method. Python will raise a TypeError if you overload the method.

```
def addition(a, b):
    c = a + b
    print(c)


def addition(a, b, c):
    d = a + b + c
    print(d)


# the below line shows an error
# addition(4, 5)

# This line will call the second product method
addition(3, 7, 5)  #It will take last defined function
```

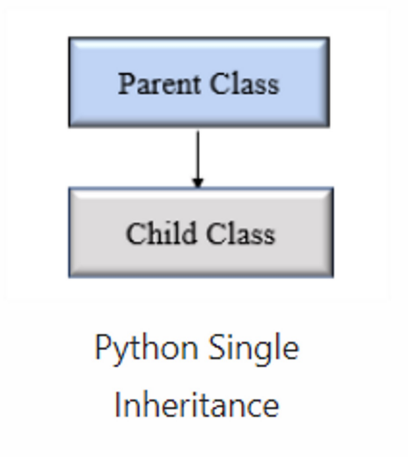
4. Inheritance in Python

Inheritance in Python

The**process of inheriting the properties of the parent class into a child class is called inheritance**. The existing class is called a base class or parent class and the new class is called a subclass or child class or derived class.

1. Single inheritance
2. Multiple Inheritance
3. Multilevel inheritance
4. Hierarchical Inheritance
5. Hybrid Inheritance

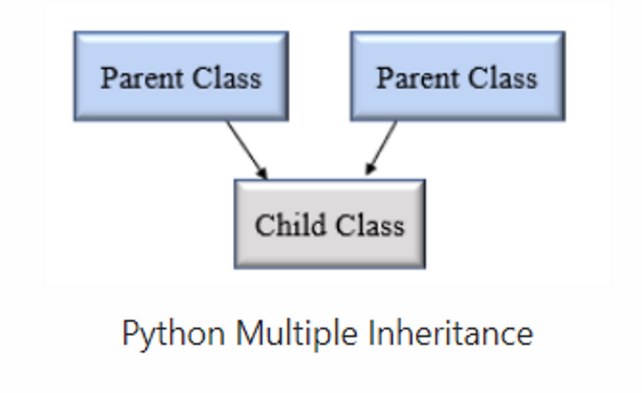
Single inheritance - In single inheritance, a child class inherits from a single-parent class. Here is one child class and one parent class.



Python Single Inheritance

```
# Base class
class Vehicle:
    def Vehicle_info(self):
        print('Inside Vehicle class')

# Child class
class Car(Vehicle):
    def car_info(self):
        print('Inside Car class')

# Create object of Car
car = Car()

# access Vehicle's info using car object
car.Vehicle_info()
car.car_info()
```

multiple inheritance - one child class can inherit from multiple parent classes. So here is one child class and multiple parent classes.



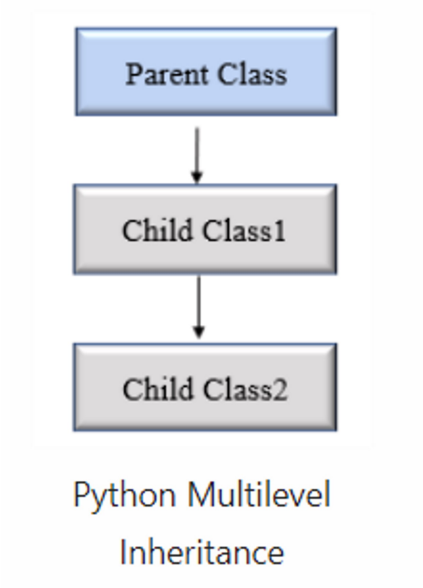
Python Multiple Inheritance

```
# parent1 class
class Vehicle1:
    def Vehicle1_info(self):
        print('Inside Vehicle1 class')

# parent2 class
class Vehicle2:
    def Vehicle2_info(self):
        print('Inside Vehicle2 class')

# Child class
class Car(Vehicle1, Vehicle2):
    def car_info(self):
        print('Inside Car class')


# Create object of Car
car = Car()

# access Vehicle's info using car object
car.Vehicle1_info()
car.Vehicle2_info()
car.car_info()
```

```
# parent1 class
class Vehicle:
    def vehicle_info(self, name):
        print(f'Vehicle name: {name}')

# parent2 class
class MotorSpeed:
    def speed_info(self, speed):
        print(f'Vehicle speed: {speed}')

# Child class
class Car(Vehicle, MotorSpeed):
    def price_info(self, price):
        print(f'Vehicle price: {price}')
```

```
# Create object of Car
car = Car()

# access Vehicle's info using car object
car.vehicle_info("tata")
car.speed_info("230kmp")
car.price_info("2L")

# Create object of Car
car1 = Car()

# access Vehicle's info using car object
car1.vehicle_info("i10")
car1.speed_info("430kmp")
car1.price_info("10L")
```

multilevel inheritance - a class inherits from a child class or derived class. Suppose three classes A, B, C. A is the superclass, B is the child class of A, C is the child class of B. In other words, we can say a**chain of classes**is**called multilevel inheritance.**

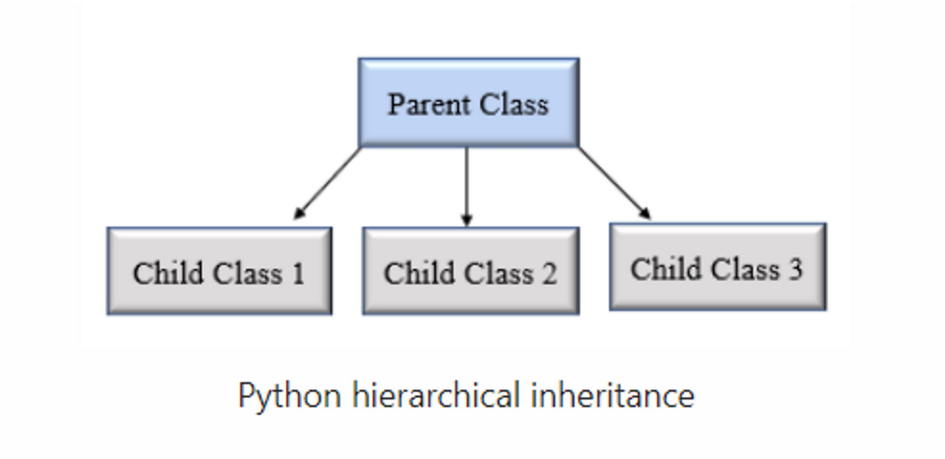


Python Multilevel Inheritance

```
# Base class
class Vehicle:
   def Vehicle_info(self):
      print('Inside Vehicle class')

# Child class
class Car(Vehicle):
   def car_info(self):
      print('Inside Car class')

# Child class
class SportsCar(Car):
   def sports_car_info(self):
      print('Inside SportsCar class')
```

```
# Create object of SportsCar
s_car = SportsCar()

# access Vehicle's and Car info using SportsCar object
s_car.Vehicle_info()
s_car.car_info()
s_car.sports_car_info()
```

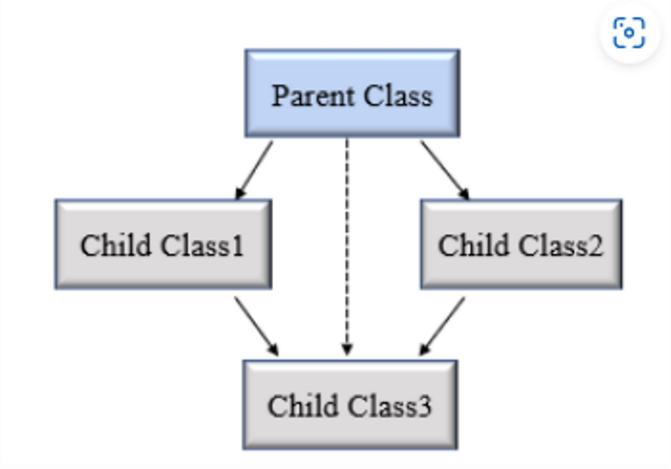
Hierarchical inheritance - more than one child class is derived from a single parent class. In other words, we can say one parent class and multiple child classes.



Python hierarchical inheritance

```
class Vehicle:
    def info(self):
        print("This is Vehicle")

class Car(Vehicle):
    def car_info(self, name):
        print("Car name is:", name)

class Truck(Vehicle):
    def truck_info(self, name):
        print("Truck name is:", name)

obj1 = Car()
obj1.info()
obj1.car_info('BMW')

obj2 = Truck()
obj2.info()
obj2.truck_info('Ford')
```

Hybrid Inheritance - When inheritance is consists of multiple types or a combination of different inheritance is called hybrid inheritance.



Python hybrid inheritance

```
class Vehicle:
    def vehicle_info(self):
        print("Inside Vehicle class")

class Car(Vehicle):
    def car_info(self):
        print("Inside Car class")

class Truck(Vehicle):
    def truck_info(self):
        print("Inside Truck class")

# Sports Car can inherits properties of Vehicle and Car
class SportsCar(Car, Vehicle):
    def sports_car_info(self):
        print("Inside SportsCar class")

# create object
s_car = SportsCar()

s_car.vehicle_info()
s_car.car_info()
s_car.sports_car_info()
class Vehicle:
    def vehicle_info(self):
        print("Inside Vehicle class")

class Car(Vehicle):
    def car_info(self):
        print("Inside Car class")

class Truck(Vehicle):
    def truck_info(self):
        print("Inside Truck class")

# Sports Car can inherits properties of Vehicle and Car
class SportsCar(Vehicle, Car, Truck):
    def sports_car_info(self):
        print("Inside SportsCar class")
```

```
# create object
s_car = SportsCar()
print(SportsCar.mro())
s_car.vehicle_info()
s_car.car_info()
s_car.truck_info()
s_car.sports_car_info()
s_car.mro
```

In child class, we can refer to parent class by using the`super()`function. The super function returns a temporary object of the parent class that allows us to call a parent class method inside a child class method.

```
class Company:
    def company_name(self):
        return 'Google'

class Employee(Company):
    def info(self):
        # Calling the superclass method using super()function
        c_name = super().company_name()
        print("Jessa works at", c_name)

# Creating object of child class
emp = Employee()
emp.info()
```

What is Encapsulation in Python?

Encapsulationin Python describes the concept of**bundling data andmethodswithin a single unit**. So

```
class Employee:
    # constructor
    def __init__(self, name, salary, project):
        # data members
        self.name = name
        self.salary = salary
        self.project = project

    # method
    # to display employee's details
    def show(self):
        # accessing public data member
        print("Name: ", self.name, 'Salary:', self.salary)

    # method
    def work(self):
        print(self.name, 'is working on', self.project)

# creating object of a class
emp = Employee('Jessa', 8000, 'NLP')
```

```
# calling public method of the class
emp.show()
emp.work()
```

Using encapsulation, we can hide an object's internal representation from the outside. This is called information hiding.

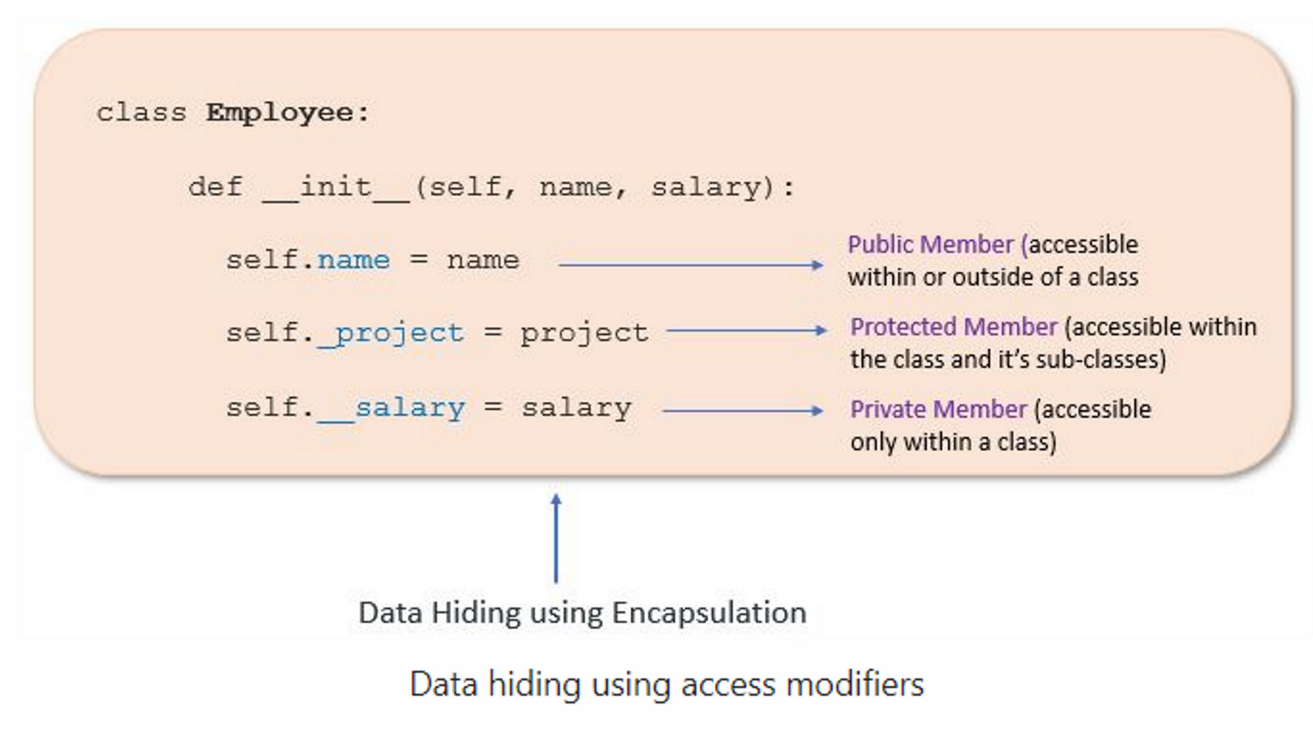
Access Modifiers in Python

Encapsulation can be achieved by declaring the data members and methods of a class either as private or protected. But In Python, we don't have direct access modifiers like public, private, and protected. We can achieve this by using single**underscore**and**doubleunderscores**.

Access modifiers limit access to the variables and methods of a class. Python provides three types of access modifiers private, public, and protected.

- **Public Member**: Accessible anywhere from outside class.
- **Private Member**: Accessible within the class
- **Protected Member**: Accessible within the class and its sub-classes

Access Specifiers	Same Class	Same Package	Derived Class	Other Class
Public	Yes	Yes	Yes	Yes
Protected	Yes	Yes	Yes	No
Private	Yes	No	No	No



Data hiding using access modifiers

Public Member: Accessible anywhere from outside class.

```
class Parent:
    # constructor
    def __init__(self, var1):
        # public data members
        self.var1 = var1

    # public instance methods
    def PublicMethod(self):
        # accessing public data member in publicmethod of same class
        print(f"public data  Var1: {self.var1} accessed thru PublicMethod")

    # creating object of a class


obj = Parent("PUBLIC VARIABLE")

# accessing public data members outside class using obj
print("var1 ", obj.var1)

# calling public method of the class
obj.PublicMethod()

class Parent:
    # constructor
    def __init__(self, var1):
        # public data members
        self.var1 = var1

    # public instance methods
    def PublicMethod(self):
        # accessing public data member in publicmethod of same class
```

```
        print(f"public data  Var1: {self.var1} accessed thru PublicMethod")

    # creating object of a class

class Child(Parent):
    def __init__(self, var1):
        Parent.__init__(self, var1)

    def AccessParentClassPublicMethed(self):
        self.PublicMethod()

obj = Child("PUBLIC VARIABLE")

# accessing public data members outside class using obj
print("var1 ", obj.var1)

# calling public method of the parent class
obj.PublicMethod()
# calling public method of the parent class in child class
obj.AccessParentClassPublicMethed()
```

- **Private Member**: Accessible within the class

```
class Parent:
    # constructor
    def __init__(self, var1):
        # Private data members
        self.__var1 = var1

    # Private instance methods
    def __PrivateMethod(self):
        # accessing Protected data member in Protected method of same class
        print(f"Private data  Var1: {self.__var1} accessed thru Private Method")

    # accessing private variavles and methods using public method
    def accessprivatemethod(self):
        print(f"private var1: {self.__var1}")
        self.__PrivateMethod()

    # creating object of a class

class Child(Parent):
    def __init__(self, var1):
        Parent.__init__(self, var1)

    def AccessParentClassPrivateMethed(self):
        self.__PrivateMethod()

# obj = Parent("PRIVATE VARIABLE PARENT")
obj1 = Child("PRIVATE VARIABLE CHILD")
# directly accessing Protected data members outside class using obj
# print("var1 ", obj.__var1)
# obj.__PrivateMethod()

obj1.AccessParentClassPrivateMethed()
```

- **Protected Member**: Accessible within the class and its sub-classes.

```
class Parent:
    # constructor
    def __init__(self, var1):
        # Protected data members
        self._var1 = var1

    # Protected instance methods
    def _ProtectedMethod(self):
        # accessing Protected data member in Protected method of same class
        print(f"Protected data  Var1: {self._var1} accessed thru Protected Method")

    # creating object of a class

class Child(Parent):
    def __init__(self, var1):
        Parent.__init__(self, var1)

    def AccessParentClassPqrotectedMethed(self):
        self._ProtectedMethod()

obj = Parent("PUBLIC VARIABLE")
obj1 = Child("PUBLIC VARIABLE CHILD")
# directly accessing Protected data members outside class using obj
print("var1 ", obj._var1)
obj._ProtectedMethod()
print(obj1._var1)
obj1.AccessParentClassPqrotectedMethed()
```