**Oops Concept**

Oops" stands for Object-Oriented Programming, and it's a programming paradigm that uses objects – instances of classes – for structuring and organizing code. This paradigm is based on the concept of "objects," which can contain data, in the form of fields (often known as attributes or properties), and code, in the form of procedures (often known as methods). The primary goals of object-oriented programming are to increase the modularity, reusability, and maintainability of code.

**Class:** A class is a blueprint or template for creating objects. It defines a datatype, including attributes and methods that will be common to all objects instantiated from that class.

1. **Object:** An object is an instance of a class. It is a self-contained unit that consists of both data (attributes) and the methods that operate on the data.
2. **Encapsulation:** Encapsulation is the bundling of data (attributes) and the methods that operate on that data into a single unit (class). It helps in hiding the internal state of an object and restricting access to its internal details.
3. **Inheritance:** Inheritance is a mechanism that allows a new class to inherit properties and behaviors from an existing class. The existing class is called the base class or superclass, and the new class is called the derived class or subclass. This promotes code reuse.
4. **Polymorphism:** Polymorphism allows objects of different classes to be treated as objects of a common base class. It enables a single interface to represent different types of objects. There are two types of polymorphism: compile-time (method overloading) and runtime (method overriding).
5. **Abstraction:** Abstraction involves simplifying complex systems by modeling classes based on the essential properties and behaviors they share. It helps in reducing programming complexity and effort.

**Object**

**Example:**

class MyClass:

variable = "blah"

def function(self):

print("This is a message inside the class.")

myobjectx = MyClass()

myobjectx.variable

myobjectx.function()

**Inheritance:**

Inheritance enable us to define a class that takes all the functionality from parent class and allows us to add more. It refers to defining a new class with little or no modification to an existing class. The new class is called derived (or child) class and the one from which it inherits is called the base (or parent) class. The name BaseClassName must be defined in a scope containing the derived class definition.

**Syntax**

class BaseClass:

Body of base class

class DerivedClass(BaseClass):

Body of derived class

**1. Single inheritance :**

Single inheritance enables a derived class to inherit properties and behavior from a single parent class. It allows a derived class to inherit the properties and behavior of a base class, thus enabling code reusability as well as adding new features to the existing code.

**Example**

# Base class

class Animal:

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

self.name = name

def speak(self):

pass

# Derived class (inherits from Animal)

class Dog(Animal):

def speak(self):

return f"{self.name} says Woof!"

# Another derived class (also inherits from Animal)

class Cat(Animal):

def speak(self):

return f"{self.name} says Meow!"

# Creating instances of the derived classes

dog\_instance = Dog("Buddy")

cat\_instance = Cat("Whiskers")

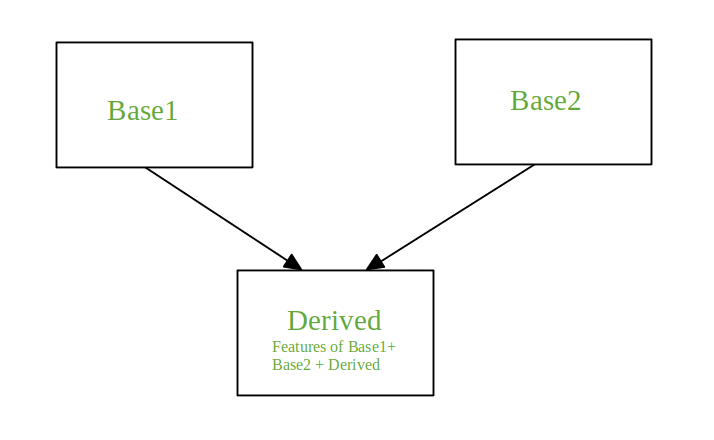
# Calling the speak method on the instances

print(dog\_instance.speak())

print(cat\_instance.speak())

**2. Multiple inheritance:**

Multiple inheritance is a feature of some object-oriented computer programming languages in which an object or class can inherit characteristics and features from more than one parent object or parent class



**Example**

class Mammal:

def mammal\_info(self):

print("Mammals can give direct birth.")

class WingedAnimal:

def winged\_animal\_info(self):

print("Winged animals can flap.")

class Bat(Mammal, WingedAnimal):

def bat\_info(self):

# Calling methods from base classes using super()

super().mammal\_info()

super().winged\_animal\_info()

print("Bats are a combination of mammals and winged animals.")

# create an object of Bat class

b1 = Bat()

# Calling the method from the Bat class

b1.bat\_info()

In multiple inheritance, the features of all the base classes are inherited into the derived class. The syntax for multiple inheritance is similar to single inheritance.

**Super keyword**   
Working with Multiple Inheritance

Allows us to avoid using base class explicitly

The super() builtin returns a proxy object, a substitute object that has ability to call method of the base class via delegation. Ability to reference base object with super() is called indirection.

**Multilevel Inheritance in Python**

You can inherit a derived class from another derived class. This is known as multilevel inheritance. In Python, multilevel inheritance can be done at any depth. On the other hand, we can also inherit form a derived class. This is called multilevel inheritance. It can be of any depth in Python. In multilevel inheritance, features of the base class and the derived class is inherited into the new derived class.

**Syntax**

class Base:

pass class Derived1(Base):

pass class Derived2(Derived1):

pass

**Example:**

class Parent:

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

self.name = name

def getName(self):

return self.name

class Child(Parent):

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

Parent.\_\_init\_\_(self, name)

self.age = age

def getAge(self):

return self.age

class Grandchild(Child):

def \_\_init\_\_(self, name, age, location):

Child.\_\_init\_\_(self, name, age)

self.location = location

def getLocation(self):

return self.location

# Creating an object of the Grandchild class

gc = Grandchild("Srinivas", 24, "Hyderabad")

# Accessing methods from all levels of the inheritance hierarchy

print(gc.getName()) # Method from Parent class

print(gc.getAge()) # Method from Child class

print(gc.getLocation()) # Method from Grandchild class

**POLYMORPHISM**

Polymorphism is based on the greek words Poly (many) and morphism (forms). We will create a structure that can take or use many forms of objects. Polymorphism can be referred as the process of taking many forms by a same object.

**1. Method overloading** In Python you can define a method in such a way that there are multiple waysto call it. Given a single method or function, we can specify the number of parameters ourself. Depending on the function definition, it can be called with zero, one, two or more parameters. This is known as method overloading.

**Example:**

class Human:

def sayHello(self, name=None):

if name is not None:

print('Hello ' + name)

else:

print('Hello ')

# Create an instance

obj = Human()

# Call the method

obj.sayHello()

# Call the method with a parameter

obj.sayHello('Guido')

**2. Method overriding:** Polymorphic behaviour allows you to specify common methods in an "abstract" level, and implement them in particular instances. Overriding is the ability of a class to change the implementation of a method provided by one of its ancestors. Overriding is a very important part of OOP since it is the feature that makes inheritance exploit its full power. In Python method overriding occurs simply defining in the child class a method with the same name of a method in the parent class. When you define a method in the object you make this latter able to satisfy that method call, so the implementations of its ancestors do not come in play.

**Example:**

class Parent:

# define parent class

def myMethod(self):

print('Calling parent method')

class Child(Parent):

# define child class

def myMethod(self):

print('Calling child method')

c = Child() # instance of child

c.myMethod() # child calls overridden method

**Constructors :**

Class functions that begins with double underscore (\_\_) are called special functions as they have special meaning like \_\_init\_\_() function. This special function gets called whenever a new object of that class is instantiated. This type of function is also called constructors in Object Oriented Programming (OOP). When a class defines an \_\_init\_\_() method, class instantiation automatically invokes \_\_init\_\_() for the newly-created class instance. The automatic destruction of unreferenced objects in Python is also called garbage collection.

**self keyword:** Class methods have only one specific difference from ordinary functions - they must have an extra first name that has to be added to the beginning of the parameter list, but you do not give a value for this parameter when you call the method, Python will provide it. This particular variable refers to the object itself, and by convention,

it is given the name self. In the \_\_init\_\_ method, self refers to the newly created object; in other class methods, it refers to the instance whose method was called.