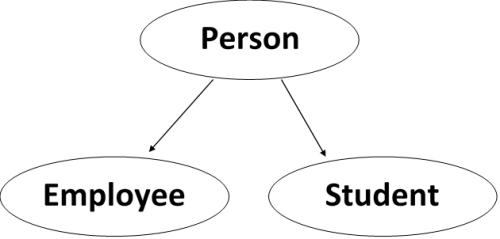
**Inheritance**

**Definition of Inheritance:**

* Defining a class from existing object functionality.
* Classes can inherit from other classes.
* A class can inherit attributes and behaviour methods from another class, called the superclass.
* A class which inherits from a superclass is called a subclass, also called heir class or child class.

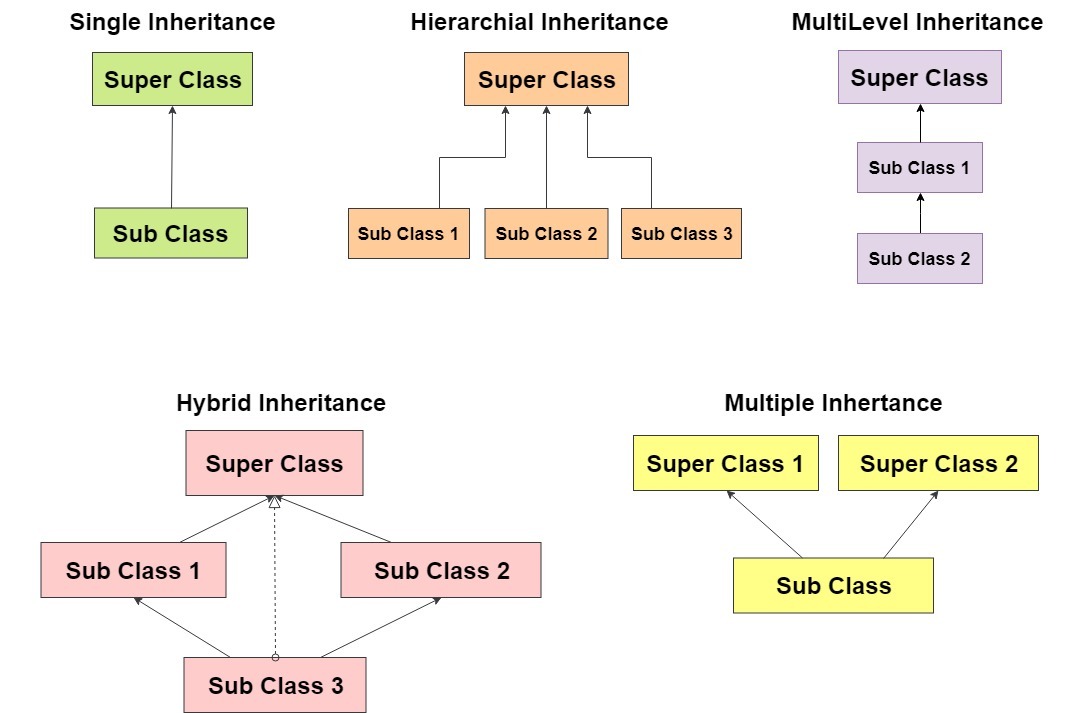


**The syntax for a subclass definition looks like this:**

class DerivedClassName(BaseClassName):

pass

**Types of inheritance:**



**Single inheritance: We need to instantiate Child class to Access Parent class funcationality as well as Child class.**

class Parent :

def m1(self):

print("Parent's functionality")

return

class Child(Parent):

def m2(self):

print("Child's functionality")

return

obj = Child()

obj.m1()

obj.m2()

Class level functionality can be accessed directly by using class name:

class Parent :

def m1():

print("Parent's functionality")

return

class Child(Parent):

def m2():

print("Child's functionality")

return

Child.m1()

Child.m2()

In Child object creation, first Parent object will be created.

Parent and Child objects get memory allocation at same place(combined object)

class Parent :

def m1(self):

print("Parent's object address : ", self)

return

class Child(Parent):

def m2(self):

print("Parent's object address : ", self)

return

obj = Child()

obj.m1()

obj.m2()

**Multi level inheritance: Accessing more than one level of objects functionality.**

class A:

def aa(self):

print("A class method")

return

class B(A):

def bb(self):

print("B class method")

return

class C(B):

def cc(self):

print("C class method")

return

obj = C()

obj.aa()

obj.bb()

obj.cc()

**Hierarchical inheritance:**

**class Vehicle:**

**def fuel(self):**

**return "Petrol"**

**class Car(Vehicle):**

**def fuel(self):**

**return "CNG"**

**class Bike(Vehicle):**

**pass**

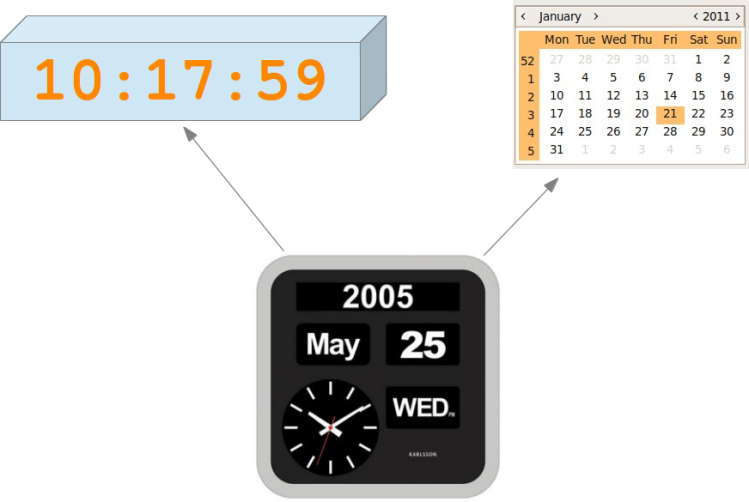
**c = Car()**

**print("Car fuel type : ", c.fuel())**

**b = Bike()**

**print("Bike fuel type : ", b.fuel())**

**Multiple inheritance:**



class Time :

def getTime(self):

print("Print System time")

return

class Date :

def getDate(self):

print("Print System Date")

return

class Watch(Time,Date):

def display(self):

self.getTime()

self.getDate()

return

obj = Watch()

obj.display()

Object initialization:

* Assigning values to object level variables of class.
* Object initialization must be done through constructor.
* Parent class constructor need to be called from the Child class in object creation process.

Following example shows how to access Parent class constructor from Child class constructor.

Connecting constructors in object creation process is called ‘Constructor Chaining’.

class Parent :

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

print("Parent's object creation")

return

class Child(Parent):

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

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

print("Child's object creation")

return

obj = Child()

**Initializing Parent Object in Child object creation process:**

class Parent :

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

self.a = a

return

class Child(Parent):

def \_\_init\_\_(self, a, b):

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

self.b = b

return

obj = Child(10,20)

print("Parent's a value : ", obj.a)

print("Child's b value : ", obj.b)

**Initializing objects in Multiple inheritance:**

class Parent1 :

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

self.a = a

return

class Parent2 :

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

self.b = b

return

class Child(Parent1, Parent2):

def \_\_init\_\_(self,a,b):

Parent1.\_\_init\_\_(self,a)

Parent2.\_\_init\_\_(self,b)

return

def display(self):

print("a value : ", self.a)

print("b value : ", self.b)

return

obj = Child(10,20)

obj.display()

Overriding:

Defining a method in the Child class with the same name and same number of arguments of Parent class method.

Method overriding is the concept of updating existing object functionality if it is not sufficient to extended class.

class SamsungGuru :

def call(self):

print("GURU - Calling....")

return

def camera(self):

print("GURU - VGA camera....")

return

class SamsungGalaxy(SamsungGuru):

# we can access existing(call) functionality

def videoCall(self):

print("Galaxy - Video calling....")

return

def camera(self):

print("Galaxy - 12MP camera....")

return

galaxy = SamsungGalaxy()

galaxy.call() # existing method

galaxy.camera() # updated method

galaxy.videoCall() # new method

Accessing Overridden functionality of Parent class:

class Parent:

def fun(self):

print("Parent")

return

class Child(Parent):

def fun(self):

super().fun()

print("Child")

return

def getIdentity(self):

self.fun()

return

obj = Child()

obj.getIdentity()

We can access the functionality of all the objects in the hierarchy from the child class by specifying class type along with object address.

class Grand:

def fun(self):

print("Grand parent")

return

class Parent(Grand):

def fun(self):

print("Parent")

return

class Child(Parent):

def fun(self):

print("Child")

return

def getIdentity(self):

self.fun()

super(Child, self).fun()

super(Parent , self).fun()

return

obj = Child()

obj.getIdentity()

**Accessing overridden functionality in Multiple inheritance:**

class Parent1:

def fun(self):

print("Parent-1")

return

class Parent2:

def fun(self):

print("Parent-2")

return

class Child(Parent1,Parent2):

def fun(self):

super().fun()

print("Child")

return

def getIdentity(self):

self.fun()

return

obj = Child()

obj.getIdentity()

Private variables in Python:

A variable declaration representation using \_\_identity is called private variable.

We can access private variable only with in the class.

Access class level variable within the class:

class A:

\_\_a=10

def display():

print("private a value : ", A.\_\_a)

return

A.display()

Access class level variable from outside class gives error:

class A:

\_\_a=10

class B:

def display():

print("private a value : ", A.\_\_a)

return

B.display()

Access object level private variable within the class :

class A:

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

self.\_\_a = a

return

def display(self):

print("private a value : ", self.\_\_a)

return

obj = A(10)

obj.display()

Access object level private variable from another class from another class:

class A:

def \_\_init\_\_(self, a, b):

self.a = a

self.\_\_b = b

return

class B:

def display():

obj = A(10,20)

print("public a value : ", obj.a)

print("private b value : ", obj.\_\_b)

return

B.display()

Protected variable access using Parent-Child class :

* Representing a variable using ‘ \_ ‘ is called protected variable.
* Protected variable can be accessed only if parent-child relation is present.

class A:

def \_\_init\_\_(self, a, b):

self.a = a

self.\_b = b # protected variable

return

class B(A):

def display():

obj = A(10,20)

print("public a value : ", obj.a)

print("private b value : ", obj.\_b)

return

B.display()

Accessing Protected functionality of Parent class:

class Parent:

def \_\_init\_\_(self, x, y):

self.\_\_x = x

self.\_y = y

class Child(Parent):

def \_\_init\_\_(self, x, y):

Parent.\_\_init\_\_(self,x,y)

super().\_\_init\_\_(x,y)

def getPrint(self):

print("x value : ", self.\_\_x)

print("y value : ", self.\_y)

return

obj = Child(10,20)

obj.getPrint()

Private function can be accessed with in the class:

class A:

def \_\_fun(self):

print("Hello")

return

def access(self):

self.\_\_fun()

return

obj = A()

obj.access()

Compiler error when we access a private function from another class:

class A:

def \_\_fun(self):

print("Hello")

return

class B:

def access():

obj = A()

obj.\_\_fun()

return

B.access()

Overridden function accessing in Multiple inheritance:

class Parent1:

def fun(self):

print("Parent1")

return

class Parent2:

def fun(self):

print("Parent2")

return

class Child(Parent1,Parent2):

def fun(self):

print("Child")

return

def access(self):

self.fun()

super().fun()

return

obj = Child()

obj.access()

**Accessing all the Parents functionality in Multiple inheritance as follows:**

class Parent1:

def fun(self):

print("Parent1")

return

class Parent2:

def fun(self):

print("Parent2")

return

class Child(Parent1,Parent2):

def fun(self):

print("Child")

return

def access(self):

self.fun()

Parent1.fun(self)

Parent2.fun(self)

return

obj = Child()

obj.access()

Hybrid or Diamond inheritance:

* When we access the functionality of Parent class and it is present in both the parent classes, it will access from left.

class A:

def fun(self):

print("A class fun")

return

class B(A):

def fun(self):

print("B class fun")

return

class C(A):

def fun(self):

print("C class fun")

return

class D(B,C):

pass

obj = D()

obj.fun()

**Check this out:**

class A:

def fun(self):

print("A class fun")

return

class B(A):

pass

class C(A):

def fun(self):

print("C class fun")

return

class D(B,C):

pass

obj = D()

obj.fun()

**Check this example:**

class A:

def fun(self):

print("A class fun")

return

class B(A):

pass

class C(A):

def fun(self):

print("C class fun")

return

class D(B,C):

pass

obj = D()

A.fun(obj)

B.fun(obj)

C.fun(obj)

D.fun(obj)

Accessing Parent class constructor from Child class using super() method:

class A :

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

self.\_a = x

class B(A):

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

super(B,self).\_\_init\_\_(x)

def getDetails(self):

print("Child's a value : " , self.\_a)

return

obj = B(10)

obj.getDetails()

Encapsulation:

* Concept from capsule
* It is the concept of protecting object
* We can apply encapsulation by following few rules.

class Emp:

def \_\_init\_\_(self, eno=0, ename=''):

self.\_\_eno = eno

self.\_\_ename = ename

return

def getEno(self):

return self.\_\_eno

def setEno(self,eno):

self.\_\_eno = eno

return

def getEname(self):

return self.\_\_ename

def setEname(self,ename):

self.\_\_ename = ename

return

class Access:

def main():

e1 = Emp(101 , 'Syam')

e2 = Emp()

e2.setEno(102)

e2.setEname('Annie')

# print("e1-ename : ", e1.\_\_eno) #error :

print("e1-ename : ", e1.getEname())

print("e2-ename : ", e2.getEname())

return

if \_\_name\_\_ == '\_\_main\_\_':

Access.main()

Abstraction:

class Parent:

def m1(self):

pass

class Child(Parent):

def m1(self):

print("m1")

return

def m2(self):

print("m2")

return

obj = Child()

obj.m1()

obj.m2()