**Inheritance**

The mechanism of deriving a new class from an old one (existing class) such that the new class inherit all the members (variables and methods) of old class is called inheritance or derivation.

Inheritance is the capability of one class to derive or inherit the properties from another class.

The old class is referred to as the Super class and the new one is called the Sub class.

* Parent Class - Base Class or Super Class
* Child Class - Derived Class or Sub Class

All classes in python are built from a single super class called ‘object’ so whenever we create a class in python, object will become super class for them internally.

class Mobile(object):

class Mobile:

*In Python 3.x, “class Test(object)” and “class Test” are same.*

*In Python 2. x, “class Test(object)” creates a class with the object as a parent (called a new-style class), and “class Test” creates an old-style class (without an objecting parent).*

**Benefits of inheritance are:**

* It represents real-world relationships well.
* It provides the **reusability** of a code. We don’t have to write the same code again and again. Also, it allows us to add more features to a class without modifying it.
* It is transitive in nature, which means that if class B inherits from another class A, then all the subclasses of B would automatically inherit from class A.

Syntax:

Class BaseClass:

{Body}

Class DerivedClass(BaseClass):

{Body}

Example of Inheritance

*# A Python program to demonstrate inheritance*

*class Person(object):*

*# Constructor*

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

*self.name = name*

*self.id = id*

*# To check if this person is an employee*

*def Display(self):*

*print(self.name, self.id)*

*# Driver code*

*emp = Person("Satyam", 102) # An Object of Person*

*emp.Display()*

## Creating a Child Class

Here **Emp** is another class which is going to inherit the properties of the **Person class**(base class

*class Emp(Person):*

*def Print(self):*

*print("Emp class called")*

*Emp\_details = Emp("Mayank", 103)*

*# calling parent class function*

*Emp\_details.Display()*

*# Calling child class function*

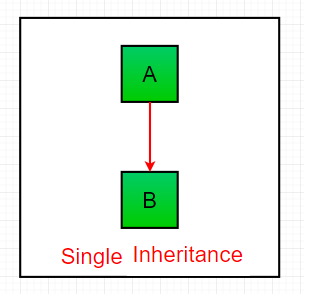
*Emp\_details.Print()*

Types of inheritance

* Single Inheritance
* Multi-level Inheritance
* Hierarchical Inheritance
* Multiple Inheritance
* Hybrid Inheritance

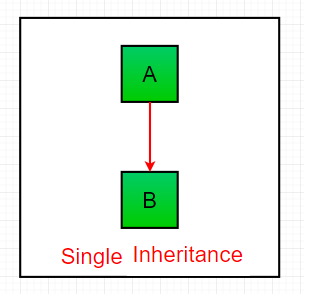
1. **Single Inheritance**

If a class is derived from one base class (Parent Class), it is called Single Inheritance. thus enabling code reusability and the addition of new features to existing code.



OR

OBJECT



Syntax:-

class ParentClassName(object):

members of Parent Class

class ChildClassName(ParentClassName):

members of Child Class

*NOTE:*

* *We can access Parent Class Variables and Methods using Child Class Object*
* *We can also access Parent Class Variables and Methods using Parent Class Object*
* *We cannot access Child Class Variables and Methods using Parent Class Object*
* *By default, The constructor in the parent class is available to the child class.*

*class Father:*

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

*self.money = 2000*

*print("Father Class Constructor")*

*class Son (Father):*

*def disp(self):*

*print(“Son Class Instance Method:”,self.money)*

*s = Son( )*

*s.disp()*

What will happen if we define constructor in both classes ?

If we write constructor in the both classes, parent class and child class then the parent class constructor is not available to the child class. In this case only child class constructor is accessible which means child class constructor is replacing parent class constructor.

Constructor overriding is used when programmer want to modify the existing behavior of a constructor.

*#What will happen if we define constructor in both classes ?*

*class Father:*

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

*self.money=2000*

*print("Father Class Constructor")*

*class Son (Father):*

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

*print("son class Constructor")*

*def disp(self):*

*print("Son Class Instance Method:",self.money)*

*s = Son( )*

*s.disp()*

**Constructor with super() method**

If we write constructor in the both classes, parent class and child class then the parent class constructor is not available to the child class.

In this case only child class constructor is accessible which means child class constructor is replacing parent class constructor.

**super ( )** method is used to call parent class constructor or methods from the child class

#Constructor with super() method

class Father:

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

self.money=2000

print("Father Class Constructor")

class Son (Father):

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

super().\_\_init\_\_()

print("son class Constructor")

def disp(self):

print("Son Class Instance Method:",self.money)

s = Son( )

s.disp()

1. **Multilevel Inheritance**

In multilevel inheritance, features of the base class and the derived class are further inherited into the new derived class. This is similar to a relationship representing a child and a grandfather.

**Grnadson**

**Son**

**Father**

**object**

Syntax:-

class ParentClassName(object):

members of Parent Class

class ChildClassName(ParentClassName):

members of Child Class

class GrandChildClassName(ChildClassName):

members of Grand Child Class

Example:

*#multilevel inheritance*

*class Grandfather:*

*def \_\_init\_\_(self,grandfathername):*

*self.grandfathername=grandfathername*

*class Father(Grandfather):*

*def \_\_init\_\_(self,fatername, grandfathername):*

*self.fathername=fathername*

*self.grandfathername=grandfathername*

*class Son (Father):*

*def \_\_init\_\_(self,sonname,fathername, grandfathername):*

*self.sonname=sonname*

*self.fathername=fathername*

*self.grandfathername=grandfathername*

*def disp\_name(self):*

*print('Grandfather name :', self.grandfathername)*

*print("Father name :", self.fathername)*

*print("Son name :", self.sonname)*

*s = Son("Sanavi","Rohit","Narhar" )*

*s.disp\_name()*

**3.Multiple Inheritance**

When a class can be derived from more than one base class this type of inheritance is called multiple inheritances. In multiple inheritances, all the features of the base classes are inherited into the derived class.

**Child**

**Parent 2**

**Parent 1**

**object**

Syntax:-

class ParentClassName1(object):

members of Parent Class

class ParentClassName2(object):

members of Parent Class

class ChildClassName(ParentClassName1, ParentClassName2):

members of Child Class

Example:

#Multiple Inheritance

# Python program to demonstrate

# multiple inheritance

# Base class1

*class Mother:*

*mothername = ""*

*def mother(self):*

*print(self.mothername)*

*# Base class2*

*class Father:*

*fathername = ""*

*def father(self):*

*print(self.fathername)*

*# Derived class*

*class Son(Father, Mother):*

*def parents(self):*

*print("Father :", self.fathername)*

*print("Mother :", self.mothername)*

*# Driver's code*

*s1 = Son()*

*s1.fathername = "RAM"*

*s1.mothername = "SITA"*

*s1.parents()*

**Method Resolution Order (MRO)**

It is also called as Diamond shape problem or Diamond inheritance

In the multiple inheritance scenario members of class are searched first in the current class. If not found, the search continues into parent classes in depth-first, left to right manner without searching the same class twice.

* Search for the child class before going to its parent class.
* When a class is inherited from several classes, it searches in the order from left to right in the parent classes.
* It will not visit any class more than once which means a class in the inheritance hierarchy is traversed only once exactly.

Multiple inheritance with super()method

*class Father:*

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

*super().\_\_init\_\_()*

*print("Father constructor")*

*def showF(self):*

*print("Father class")*

*class Mother:*

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

*print("mother constructor")*

*def showM(self):*

*print("Mother class")*

*class Son(Father,Mother):*

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

*super().\_\_init\_\_()*

*print("Son constructor")*

*def showS(self):*

*print("Son class")*

*obj=Son()*

*obj.showS()*

*obj.showM()*

*obj.showF()*

**use of super()**

still if we want to use parent class we have to use super method

If we write method in the both classes, parent class and child class then the parent class’s method is not available to the child class.

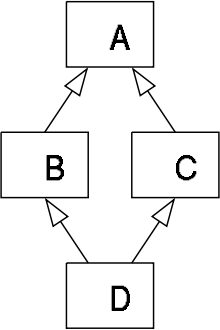
In this case only child class’s method is accessible which means child class’s method is replacing parent class’s method.

**super ( )** method is used to call parent class’s constructor or methods from the child class.

**Explanation of above program**

* The search will start from Son. As the object of Son is created, the constructor of Son is called.
* Son has super().\_\_init\_\_() inside his constructor so its parent class, the one in the left side ‘Father’ class’s constructor is called.
* Father class also has super().\_\_init\_\_() inside his constructor so its parent ‘object’ class’s constructor is called.
* Object does not have any constructor so the search will continue down to right hand side class (Mother) of object class so Mother class’s constructor is called.
* As Mother class also has super(). \_\_inti\_\_() so its parent class ‘object’ constructor is called but as object class already visited, the search will stop here.

**Diamond inheritance** and it looks as follows:



Python follows a depth-first lookup order and hence ends up calling the method from class A. By following the method resolution order, the lookup order as follows.   
Class D -> Class B -> Class C -> Class A   
Python follows depth-first order to resolve the methods and attributes. So in the above example, it executes the method in class B.

Method Resolution Order (MRO) is a order in which methods should be inherited in the case of multiple iheritance. [C3 linearization algorithm](https://en.wikipedia.org/wiki/C3_linearization) is how MRO works under the hood since [version 2.3](https://www.python.org/download/releases/2.3/mro/).

**# Demonstration of MRO**

class X:

pass

class Y:

pass

class Z:

pass

class A(X, Y):

pass

class B(Y, Z):

pass

class M(B, A, Z):

pass

# Output:

# [<class '\_\_main\_\_.M'>, <class '\_\_main\_\_.B'>,

# <class '\_\_main\_\_.A'>, <class '\_\_main\_\_.X'>,

# <class '\_\_main\_\_.Y'>, <class '\_\_main\_\_.Z'>,

# <class 'object'>]

print(M.mro())

classname.mro() => returns list

classname.\_\_mro\_\_() = > tuple format

**4.Hierarchical Inheritance**

When more than one derived class are created from a single base this type of inheritance is called hierarchical inheritance

Syntax:-

class ParentClassName(object):

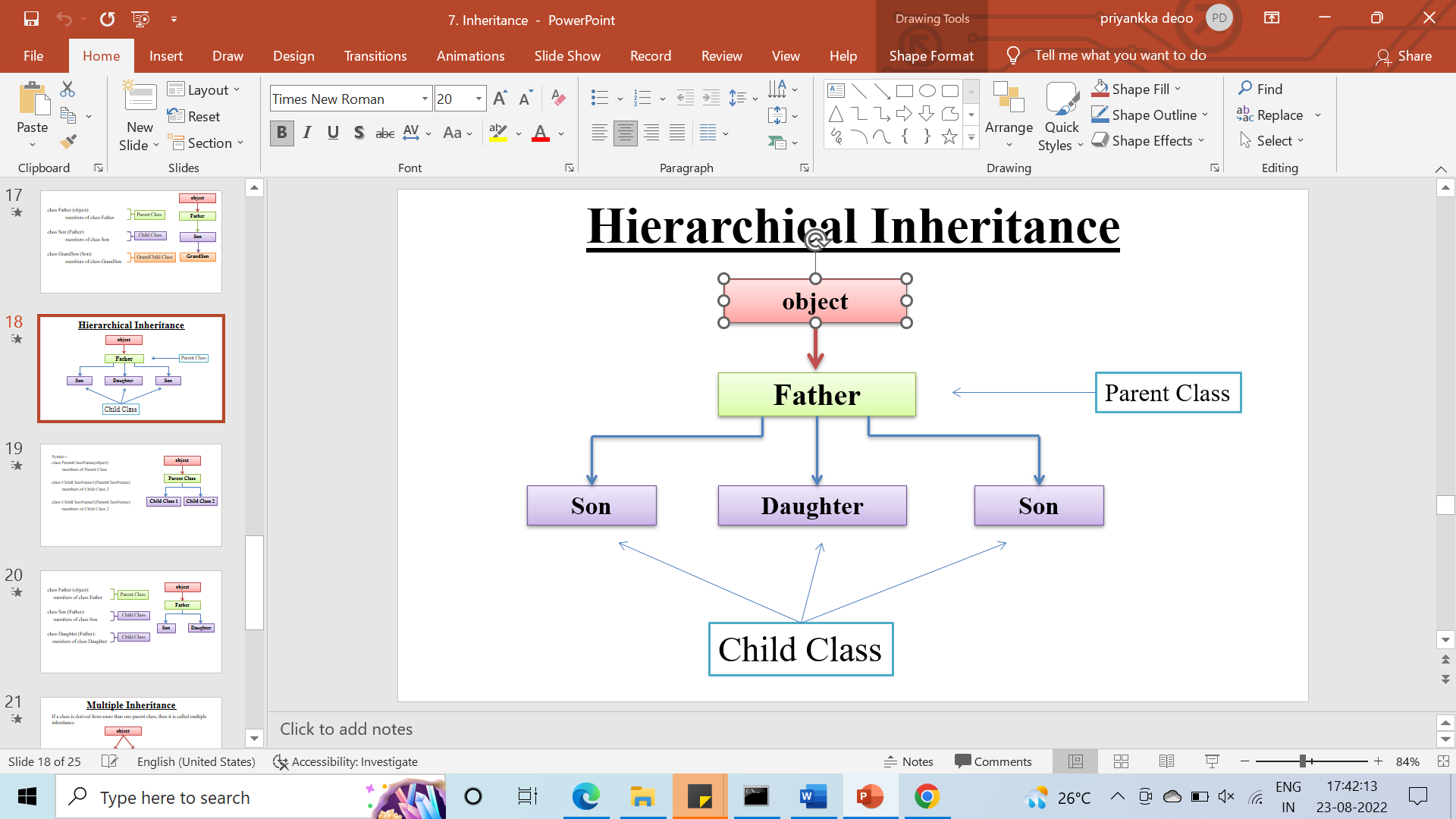
members of Parent Class

class ChildClassName1(ParentClassName):

members of Child Class 2

class ChildClassName2(ParentClassName):

members of Child Class 2



Example

# Hierarchical inheritance

*# Base class*

*class Parent:*

*def func1(self):*

*print("This function is in parent class.")*

*# Derived class1*

*class Child1(Parent):*

*def func2(self):*

*print("This function is in child 1.")*

*# Derivied class2*

*class Child2(Parent):*

*def func3(self):*

*print("This function is in child 2.")*

*# Driver's code*

*object1 = Child1()*

*object2 = Child2()*

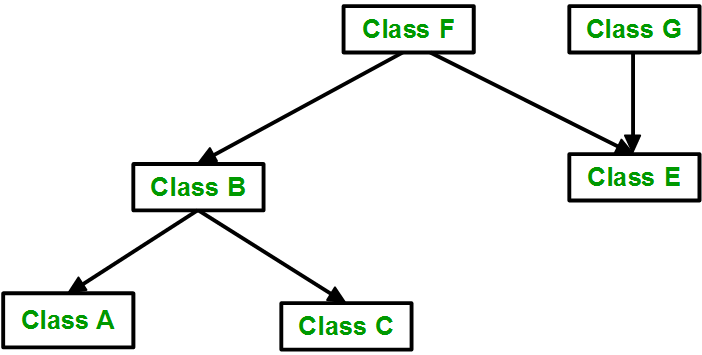
*object1.func1()*

*object1.func2()*

*object2.func1()*

*object2.func3()*

### *5.* **Hybrid Inheritance:**

Inheritance consisting of multiple types of inheritance is called hybrid inheritance.

|  |
| --- |
| *# Python program to demonstrate*  *# hybrid inheritance*      ***class*** *School:*  ***def*** *func1(self):*  ***print****("This function is in school.")*      ***class*** *Student1(School):*  ***def*** *func2(self):*  ***print****("This function is in student 1. ")*      ***class*** *Student2(School):*  ***def*** *func3(self):*  ***print****("This function is in student 2.")*      ***class*** *Student3(Student1, School):*  ***def*** *func4(self):*  ***print****("This function is in student 3.")*      *# Driver's code*  *object* ***=*** *Student3()*  *object.func1()*  *object.func2()* |