Python Programming

RGM College of Engineering & Technology (Autonomous)

Department of Computer Science & Engineering

AY:2021-2022

PYTHON’S OBJECT ORIENTED PROGRAMMING - 8

Guido Van Rossum

Agenda:

1. Using Members of One class inside another class

    i. By Composition (HAS-A Relationship)

    ii. By Inheritance (IS-A Relationship)

2. Types of Inheritance

3. Method Resolution Order (MRO) Algorithm

1. Using Members of One class inside another class

We can use members of one class inside another class by using the following two ways:

i. By Composition (HAS-A Relationship)

ii. By Inheritance (IS-A Relationship)

1. By Composition (HAS-A Relationship):

- By using object reference, we can access members of one class inside another class. This approach is nothing but composition or HAS-A Relationship.
- The main advantage of HAS-A Relationship is Code Reusability.

Note:

Composing bigger object with small small individual objects is called as Composition.

Eg: Car object is composed with engine object, tyres object, seats object and brakes object etc..

- class car has a engine reference
- class car has a tyres reference
- class car has a brakes reference and so on…

This type of relationship is known as **Composition** or **HAS-A** relationship**.**

Demo Program 1 on HAS-A Relationship:

```python
class Engine:

    def useEngine(self):
        print("Engine Specific Functionality ")

class car:

    def __init__(self):
        self.engine = Engine()

    def useCar(self):
        print('Car required Engine Functionality ')
        self.engine.useEngine()

c = car()
c.useCar()
```

```
Car required Engine Functionality
Engine Specific Functionality
```

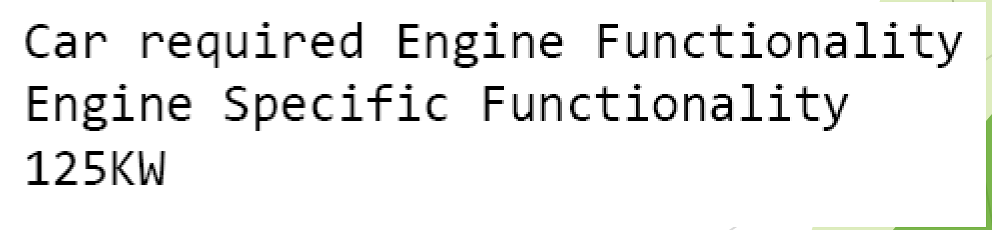
Note:

In the above program class car **has a** Engine reference.

Not only methods, variables of a class also can be used in another class.

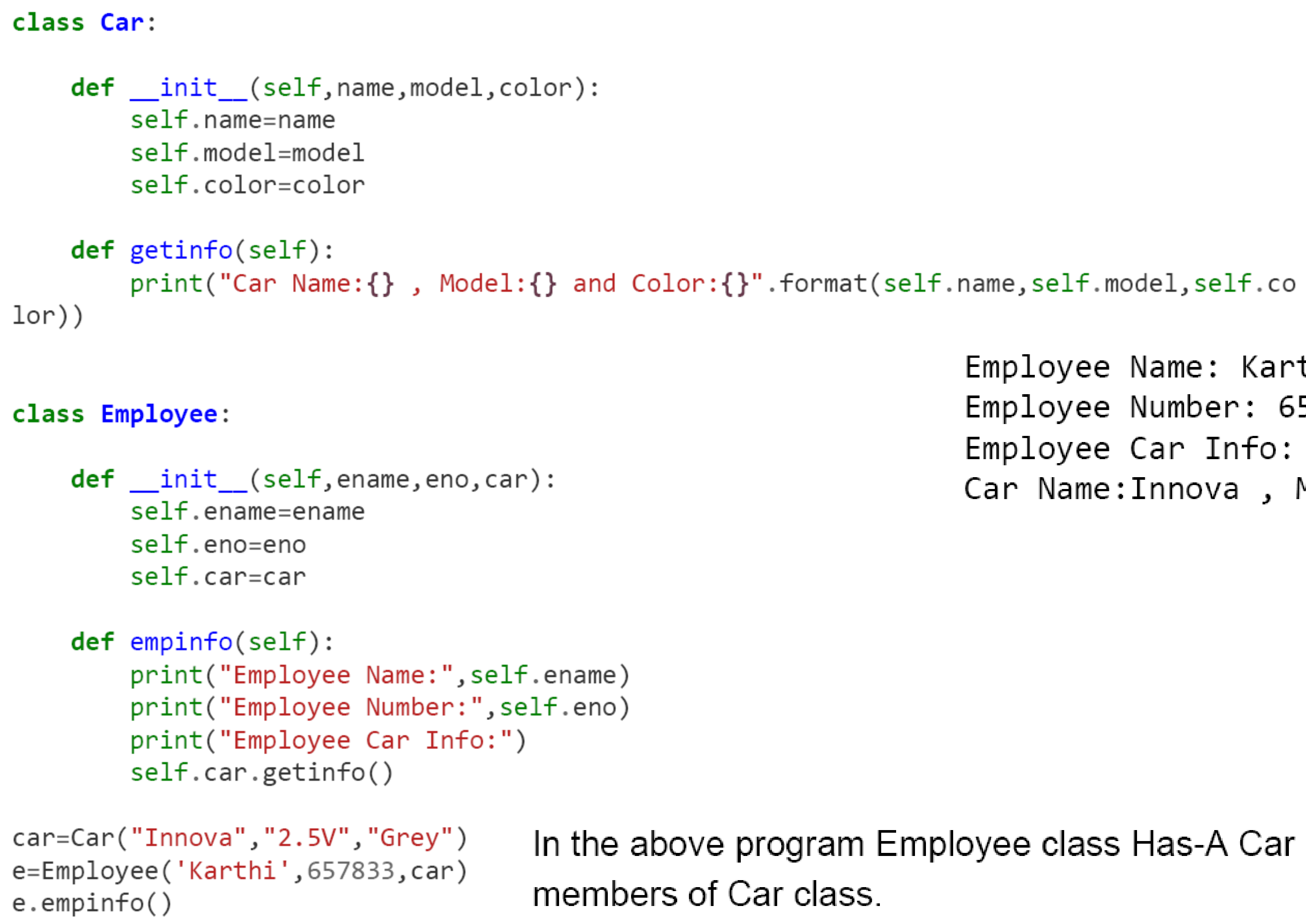
Demo Program 2 on HAS-A Relationship:

```python
class Engine:

    def __init__(self):
        self.power = '125KW'

    def useEngine(self):
        print("Engine Specific Functionality ")

class car:

    def __init__(self):
        self.engine = Engine()

    def useCar(self):
        print('Car required Engine Functionality ')
        self.engine.useEngine()
        print(self.engine.power)

c = car()
c.useCar()
```

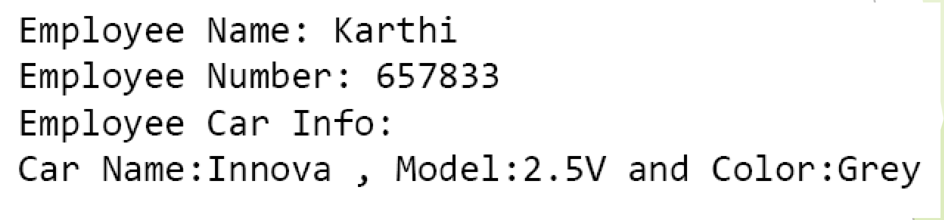


```
Car required Engine Functionality
Engine Specific Functionality
125KW
```

Demo Program 3 on HAS-A Relationship:



```
class Car:

    def __init__(self,name,model,color):
        self.name=name
        self.model=model
        self.color=color

    def getinfo(self):
        print("Car Name:{} , Model:{} and Color:{}".format(self.name,self.model,self.co
lor))


class Employee:

    def __init__(self,ename,eno,car):
        self.ename=ename
        self.eno=eno
        self.car=car

    def empinfo(self):
        print("Employee Name:",self.ename)
        print("Employee Number:",self.eno)
        print("Employee Car Info:")
        self.car.getinfo()

car=Car("Innova","2.5V","Grey")
e=Employee('Karthi',657833,car)
e.empinfo()
```



```
Employee Name: Karthi
Employee Number: 657833
Employee Car Info:
Car Name:Innova , Model:2.5V and Color:Grey
```

In the above program Employee class Has-A Car reference and hence Employee class can access all members of Car class.

Demo Program 4 on HAS-A Relationship:

Problem Description:

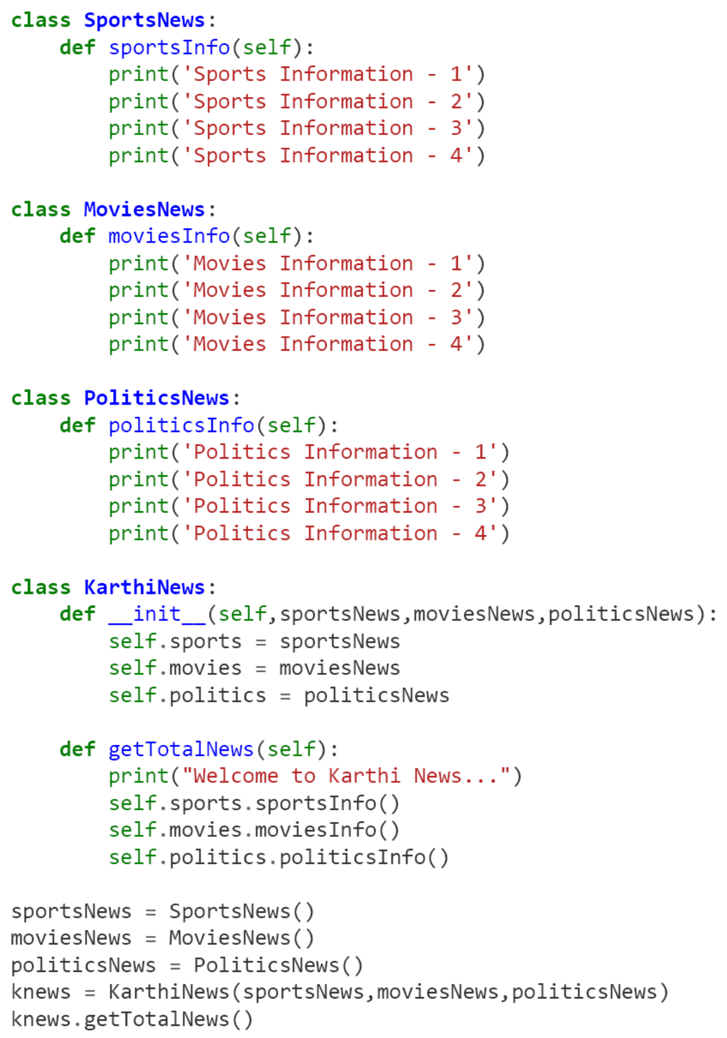
- Suppose I want to start one news channel (KarthiNews), in that I want to include sports news, political news, movies news and so on.
- Assume that to provide all these information already separate classes are defined.
- Being KarthiNews class what I need to do is, use the functionalities of all those classes inside and display whatever information provided by those classes.

Way 1:

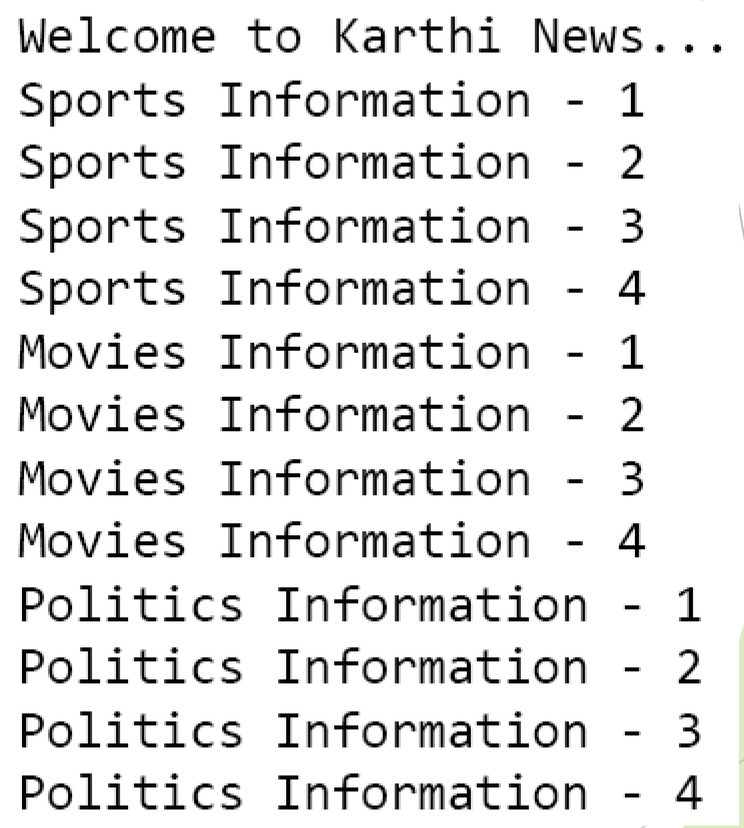
```python
class SportsNews:
    def sportsInfo(self):
        print('Sports Information - 1')
        print('Sports Information - 2')
        print('Sports Information - 3')
        print('Sports Information - 4')

class MoviesNews:
    def moviesInfo(self):
        print('Movies Information - 1')
        print('Movies Information - 2')
        print('Movies Information - 3')
        print('Movies Information - 4')

class PoliticsNews:
    def politicsInfo(self):
        print('Politics Information - 1')
        print('Politics Information - 2')
        print('Politics Information - 3')
        print('Politics Information - 4')

class KarthiNews:
    def __init__(self):
        self.sports = SportsNews()
        self.movies = MoviesNews()
        self.politics = PoliticsNews()

    def getTotalNews(self):
        print("Welcome to Karthi News...")
        self.sports.sportsInfo()
        self.movies.moviesInfo()
        self.politics.politicsInfo()

knews = KarthiNews()
knews.getTotalNews()
```

```
Welcome to Karthi News...
Sports Information - 1
Sports Information - 2
Sports Information - 3
Sports Information - 4
Movies Information - 1
Movies Information - 2
Movies Information - 3
Movies Information - 4
Politics Information - 1
Politics Information - 2
Politics Information - 3
Politics Information - 4
```

Way 2:



```python
class SportsNews:
    def sportsInfo(self):
        print('Sports Information - 1')
        print('Sports Information - 2')
        print('Sports Information - 3')
        print('Sports Information - 4')

class MoviesNews:
    def moviesInfo(self):
        print('Movies Information - 1')
        print('Movies Information - 2')
        print('Movies Information - 3')
        print('Movies Information - 4')

class PoliticsNews:
    def politicsInfo(self):
        print('Politics Information - 1')
        print('Politics Information - 2')
        print('Politics Information - 3')
        print('Politics Information - 4')

class KarthiNews:
    def __init__(self,sportsNews,moviesNews,politicsNews):
        self.sports = sportsNews
        self.movies = moviesNews
        self.politics = politicsNews

    def getTotalNews(self):
        print("Welcome to Karthi News...")
        self.sports.sportsInfo()
        self.movies.moviesInfo()
        self.politics.politicsInfo()

sportsNews = SportsNews()
moviesNews = MoviesNews()
politicsNews = PoliticsNews()
knews = KarthiNews(sportsNews,moviesNews,politicsNews)
knews.getTotalNews()
```



```
Welcome to Karthi News...
Sports Information - 1
Sports Information - 2
Sports Information - 3
Sports Information - 4
Movies Information - 1
Movies Information - 2
Movies Information - 3
Movies Information - 4
Politics Information - 1
Politics Information - 2
Politics Information - 3
Politics Information - 4
```

Demo Program 5 on HAS-A Relationship:

```python
class X:

    a = 10                  # static variable (class X)

    def __init__(self):
        self.b = 20

    def m1(self):
        print("m1 method of X class")

class Y:

    c = 30                  # static variable (class Y)

    def __init__(self):
        self.d = 40

    def m2(self):
        print("m2 method of Y class")

    def m3(self):
        x1=X()
        print(x1.a)           # 10
        print(x1.b)           # 20
        x1.m1()               # m1 method of X class
        print(Y.c)            # 30
        print(self.d)         # 40
        self.m2()             # m2 method of Y class
        print("m3 method of Y class")  # m3 method of Y class

y1 = Y()
y1.m3()
```

```
10
20
m1 method of X class
30
40
m2 method of Y class
m3 method of Y class
```

2. By Inheritance (IS-A Relationship):

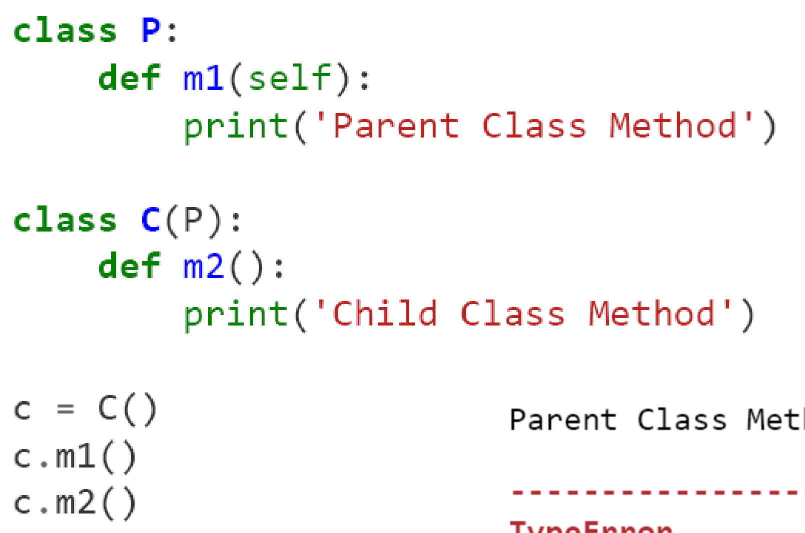
- It is a Parent to Child Relationship.
- Parent class members are by default available to the child class and hence child class can reuse parent class functionality without rewriting. Whatever variables, methods and constructors available in the parent class by default available to the child classes and we are not required to rewrite. Hence the main advantage of inheritance is Code Reusability.
- Child class can define new members also. Hence child class can extend Parent class functionality. We can extend existing functionality with some more extra functionality (i.e., Code Extendibility).

Syntax to make a class as Parent class to a child class:

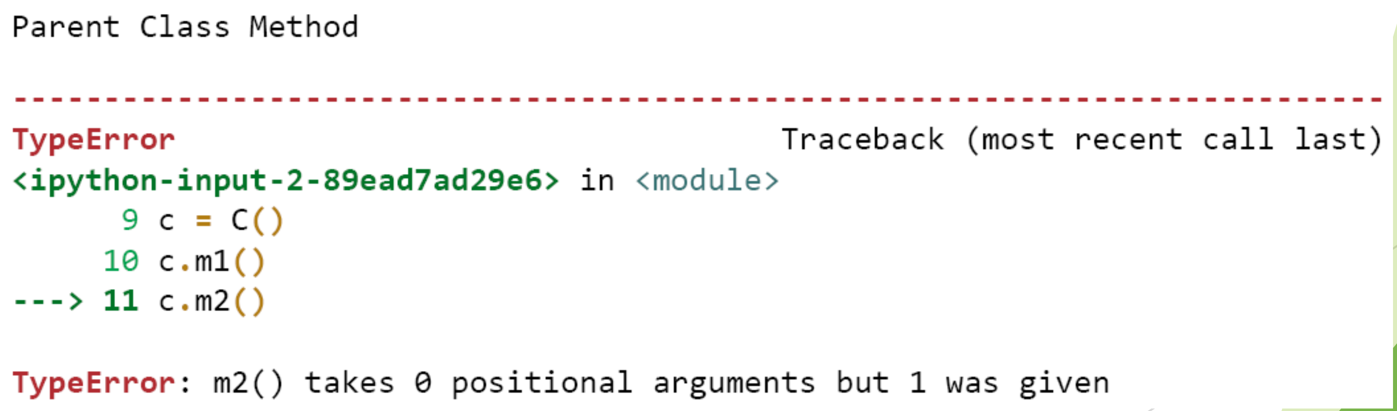
class child_class(parent_class):

Demo Program 1 on Inheritance:

I.



```
class P:
    def m1(self):
        print('Parent Class Method')

class C(P):
    def m2():
        print('Child Class Method')

c = C()
c.m1()
c.m2()
```



```
Parent Class Method

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-2-89ead7ad29e6> in <module>
      9 c = C()
     10 c.m1()
---> 11 c.m2()

TypeError: m2() takes 0 positional arguments but 1 was given
```

II.

```
class P:
    def m1(self):
        print('Parent Class Method')

class C(P):
    def m2(selfa):
        print('Child Class Method')

c = C()
c.m1()
c.m2()
```

```
Parent Class Method
Child Class Method
```

III.

```
class P:
    def m1(self):
        print('Parent Class Method')

class C(P):
    def m2(self):
        print('Child Class Method')

c = C()
c.m1()
c.m2()
```

```
Parent Class Method
Child Class Method
```

Demo Program 2 on Inheritance:

```python
class P:

    a = 10

    def __init__(self):
        print('Parent Constructor')
        self.b = 20

    def m1(self):
        print('Parent Instance Method'

    @classmethod
    def m2(cls):
        print('Parent Class Method')

    @staticmethod
    def m3():
        print('Parent Static Method')

class C(P):
    pass

c=C()         # Parent Constructor
print(c.a)    # 10
print(c.b)    # 20
c.m1()        # Parent Instance Method
c.m2()        # Parent Class Method
c.m3()        # Parent Static Method
```

```
Parent Constructor
10
20
Parent Instance Method
Parent Class Method
Parent Static Method
```

Consider the following case,

```
class P:
    10 methods
class C(P):
    5 methods
```

Conclusions:

- In the above example, Parent class contains 10 methods and these methods automatically available to the child class and we are not required to rewrite those methods(Code Reusability) Hence child class contains15 methods.
- What ever members present in Parent class are by default available to the child class through inheritance.

Demo Program 3 on Inheritance:

```
class P:
    def m1(self):
        print("Parent class method")

class C(P):
    def m2(self):
        print("Child class method")

c=C()
c.m1()
c.m2()
```

```
Parent class method
Child class method
```

Note:

- What ever the methods present in Parent class are automatically available to the child class and hence on the child class reference we can call both parent class methods and child class methods. (Similarly variables also)

Demo Program 4 on Inheritance: Developing Employee and Person classes with Inheritance.

```
class Person:

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

    def eatndrink(self):
        print('Eat Biryani and Drink Cool Drink')

class Employee(Person):

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

    def work(self):
        print("Coding Python Programs")

    def empInfo(self):
        print("Employee Name : ",self.name)
        print("Employee Age : ",self.age)
        print("Employee Number : ",self.eno)
        print("Employee Salary : ",self.esal)

e = Employee('Karthi',7,635428,49000)
e.eatndrink()
e.work()
e.empInfo()
```

```
Eat Biryani and Drink Cool Drink
Coding Python Programs
Employee Name :  Karthi
Employee Age :  7
Employee Number :  635428
Employee Salary :  49000
```

Important Note:

In the above code, the following two statements are there in both classes.

```
self.name=name
self.age=age
```

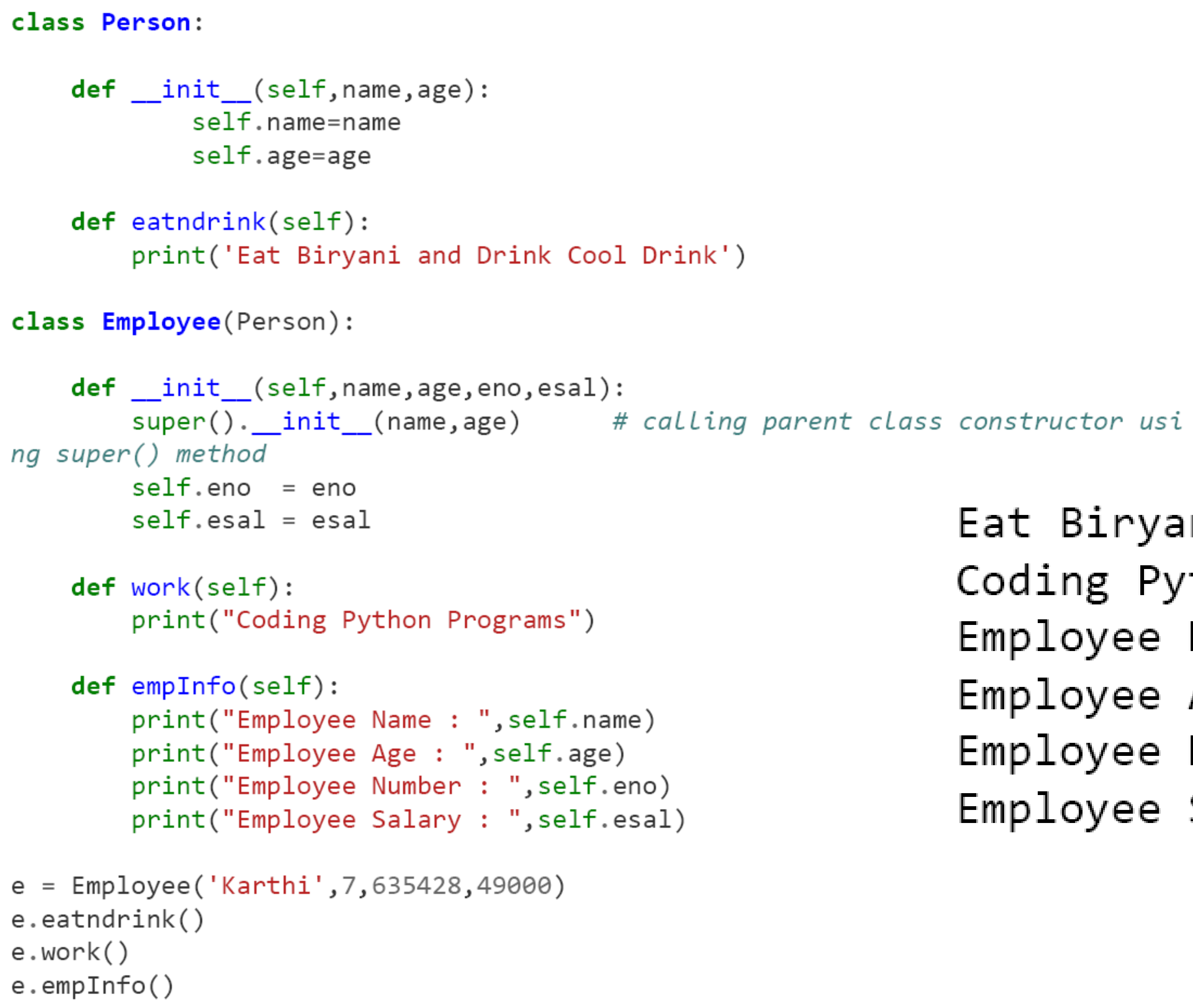
Why we have to repeat these statements in the child class, if already available in the parent class?

- To avoid this repetition, from the child class constructor, we can call the parent class constructor by passing name and age as the arguments.

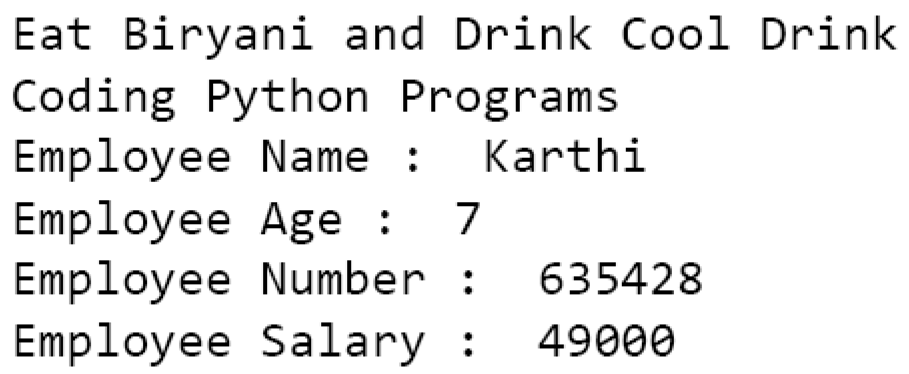
How you can call the parent class constructor from the child class constructor?

Answer: From child class constructor, if you want to access parent class members, we required to use super()method.

See the below code:



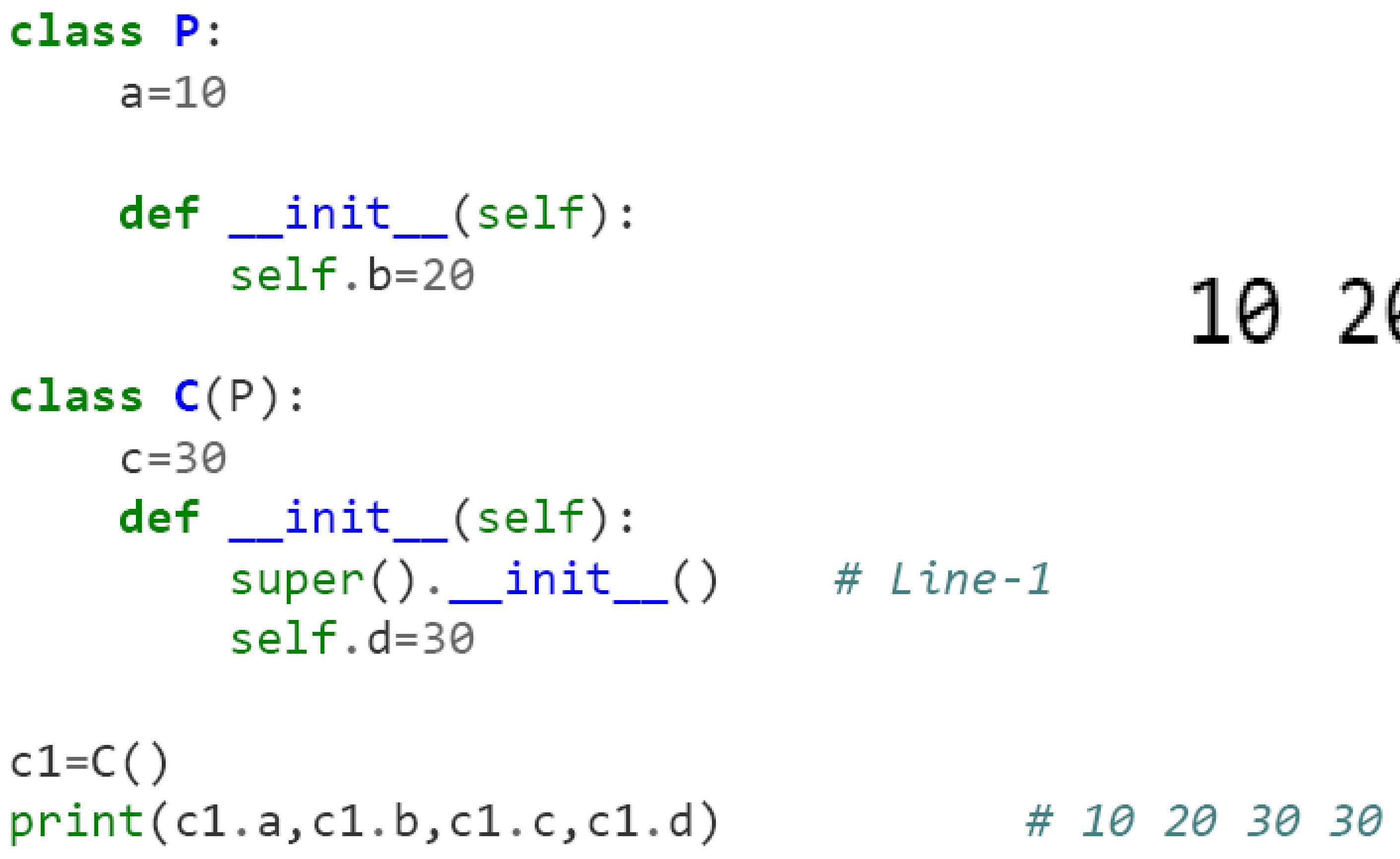
```
class Person:

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

    def eatndrink(self):
        print('Eat Biryani and Drink Cool Drink')

class Employee(Person):

    def __init__(self,name,age,eno,esal):
        super().__init__(name,age)      # calling parent class constructor usi
ng super() method
        self.eno  = eno
        self.esal = esal

    def work(self):
        print("Coding Python Programs")

    def empInfo(self):
        print("Employee Name : ",self.name)
        print("Employee Age : ",self.age)
        print("Employee Number : ",self.eno)
        print("Employee Salary : ",self.esal)

e = Employee('Karthi',7,635428,49000)
e.eatndrink()
e.work()
e.empInfo()
```



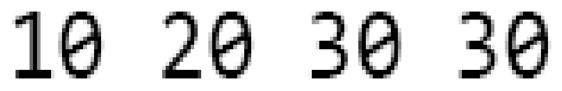
```
Eat Biryani and Drink Cool Drink
Coding Python Programs
Employee Name :  Karthi
Employee Age :  7
Employee Number :  635428
Employee Salary :  49000
```

Demo Program 5 on Inheritance:

I.



```
class P:
    a=10

    def __init__(self):
        self.b=20

class C(P):
    c=30
    def __init__(self):
        super().__init__()     # Line-1
        self.d=30

c1=C()
print(c1.a,c1.b,c1.c,c1.d)             # 10 20 30 30
```



```
10 20 30 30
```

II.

```
class P:
    a=10

    def __init__(self):
        self.b=20

class C(P):
    c=30
    def __init__(self):
        # super().__init__()    # variable 'b' is not available to the child c
lass
        self.d=30

c1=C()
print(c1.a,c1.b,c1.c,c1.d)           # 10 20 30 30
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-4-8935db3b8f36> in <module>
     12
     13 c1=C()
---> 14 print(c1.a,c1.b,c1.c,c1.d)           # 10 20 30 30

AttributeError: 'C' object has no attribute 'b'
```

Inheritance Importance:

Scenario 1:

Suppose I want to build Loan functionality.

I need to handle the following 3 types of loans:

1. Home loans
2. Personal Loans
3. Vehicle Loans

Let us see that,

If I don't aware about Inheritance how I write the code?

and

If I know about Inheritance how I write the code?

Without Inheritance:

```
class Homeloan:
     300 methods

class Vehicleloan:
     300 methods

class PersonalLoan:
     300 methods
```

In above code,

- I wrote total 900 methods.
- Assume that 90 hours of development time required.

If I show this code to some expert person. Then Immediately that expert asked me that, Do you know the inheritance concept?

He said that,

- If you use Inheritance, you may save half of the development time.
- If you use Inheritance, you may save half of the length of code.

Then I asked him that how can I use inheritance to my application?

Then, he told the following tasks to me:

- First identify the common methods in all these 3 classes (Assume that 250 common methods).
- Develop a class which consists of such common code, so that redundancy can be reduced.

With Inheritance:

```
class loan:
    250 common methods
class Homeloan(loan):
    50 Homeloan specific methods

class Vehicleloan(loan):
    50 Vehicle loan specific methods

class PersonalLoan(loan):
    50 Personal loan specific methods
```

In the above code,

- 400 methods are enough to develop my application.
- 40 hours of development time is required.

Conclusion:

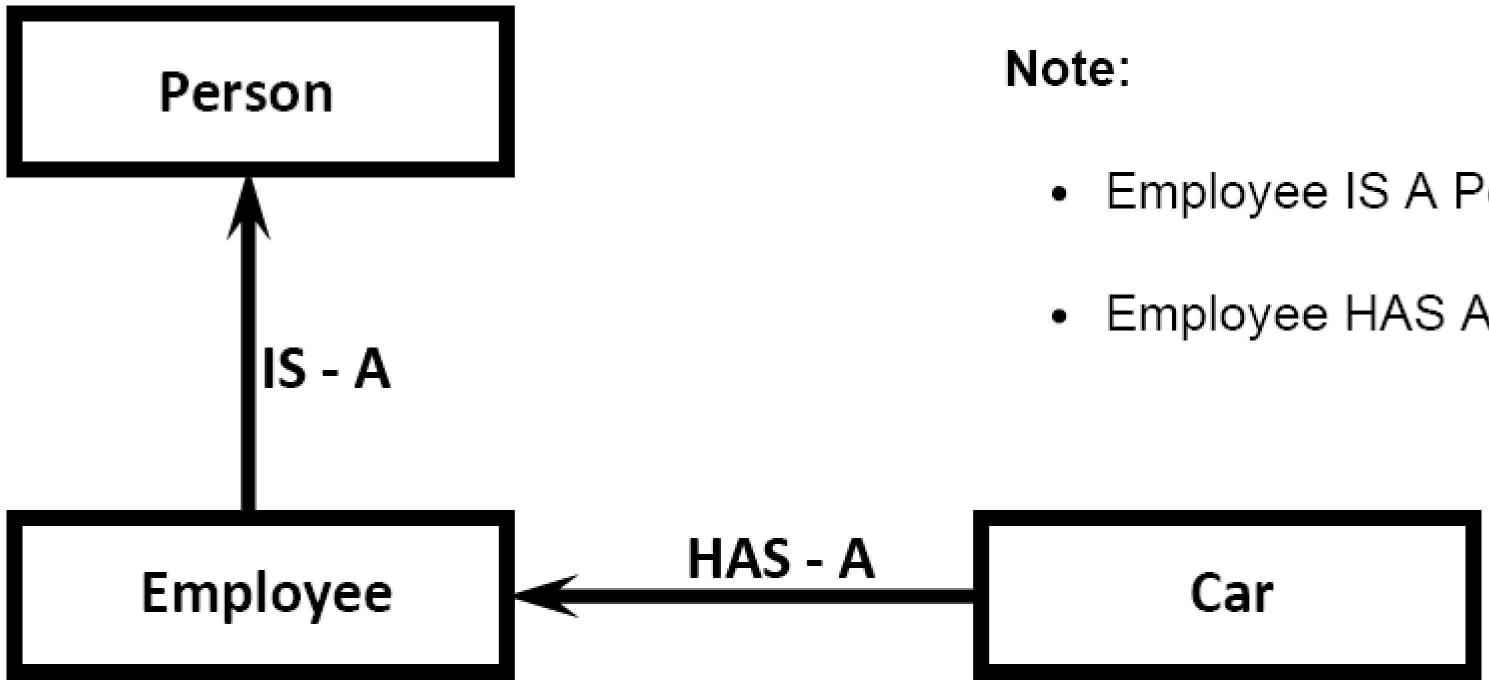
- If you feel comfortable so far, then we will discuss about, what are the different types of inheritances and how they implement them with suitable examples.

IS-A vs HAS-A Relationships

- If we want to extend existing functionality with some more extra functionality then we should go for IS-A Relationship.
- If we don't want to extend and just we have to use existing functionality then we should go for HAS-A Relationship.

Eg:

- Employee class extends Person class Functionality.
- But, Employee class just uses Car functionality but not extending.



Note:

- Employee IS A Person
- Employee HAS A Car

Demo Program:

```python
class Car:

    def __init__(self,name,model,color):
        self.name=name
        self.model=model
        self.color=color

    def getinfo(self):
        print("\tCar Name:{} \n\t Model:{} \n\t Color:{}".format(self.name,sel
f.model,self.color))

class Person:

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

    def eatndrink(self):
        print('Eat Biryani and Drink Cool Drink')

class Employee(Person):

    def __init__(self,name,age,eno,esal,car):
        super().__init__(name,age)
        self.eno=eno
        self.esal=esal
        self.car=car

    def work(self):
        print("Coding Python is programs... ")

    def empinfo(self):
        print("Employee Name:",self.name)
        print("Employee Age:",self.age)
        print("Employee Number:",self.eno)
        print("Employee Salary:",self.esal)
        print("Employee Car Info:")
        self.car.getinfo()   # Employee using Car class functionality

c=Car("Innova","2.5V","Grey")
e=Employee('Karthi',7,654378,10000,c)
e.eatndrink()  # Employee using Person class functionality
e.work()
e.empinfo()
```

```
Eat Biryani and Drink Cool Drink
Coding Python is programs...
Employee Name: Karthi
Employee Age: 7
Employee Number: 654378
Employee Salary: 10000
Employee Car Info:
        Car Name:Innova
         Model:2.5V
         Color:Grey
```

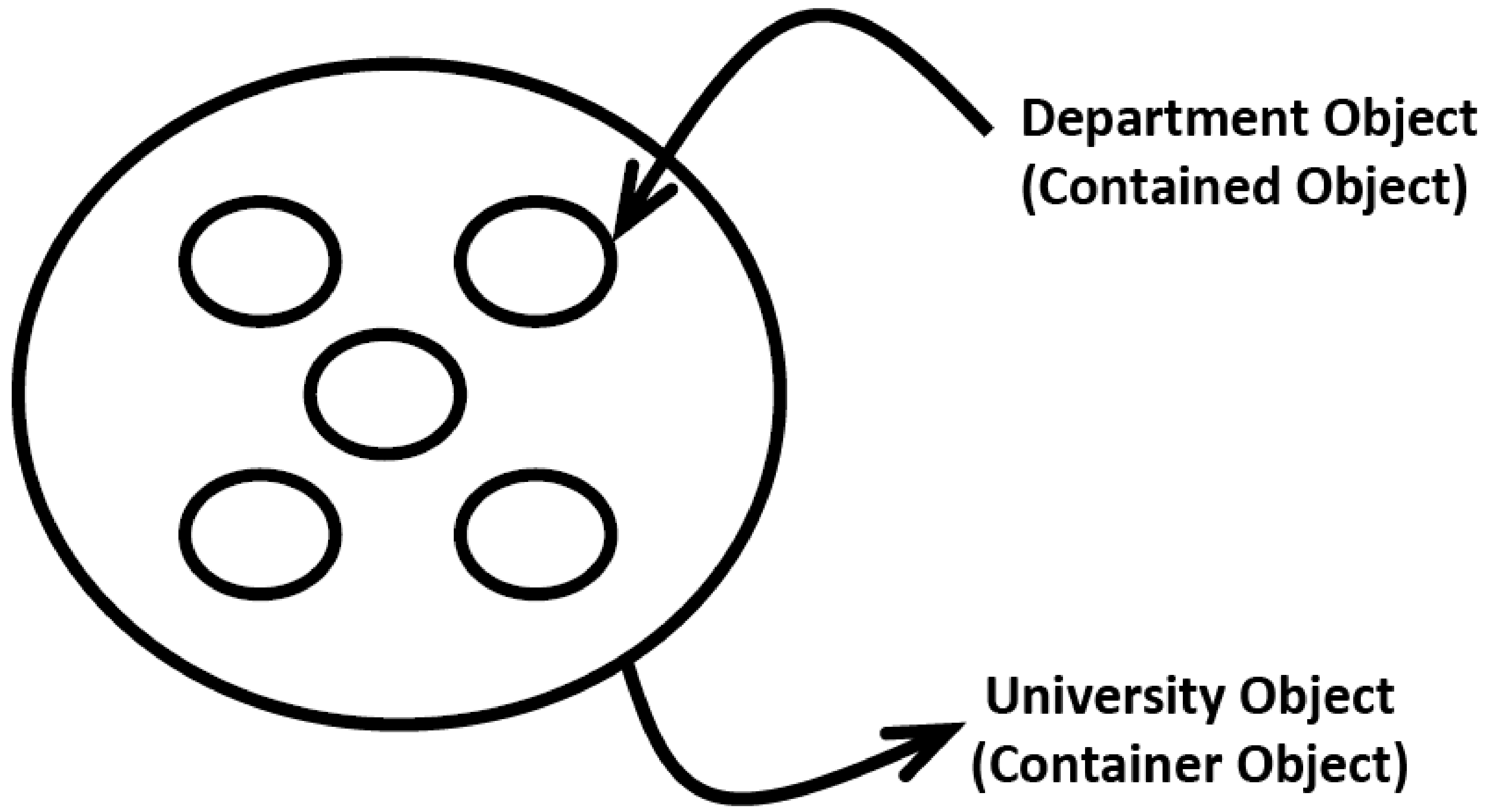
Composition vs Aggregation

1. Composition:

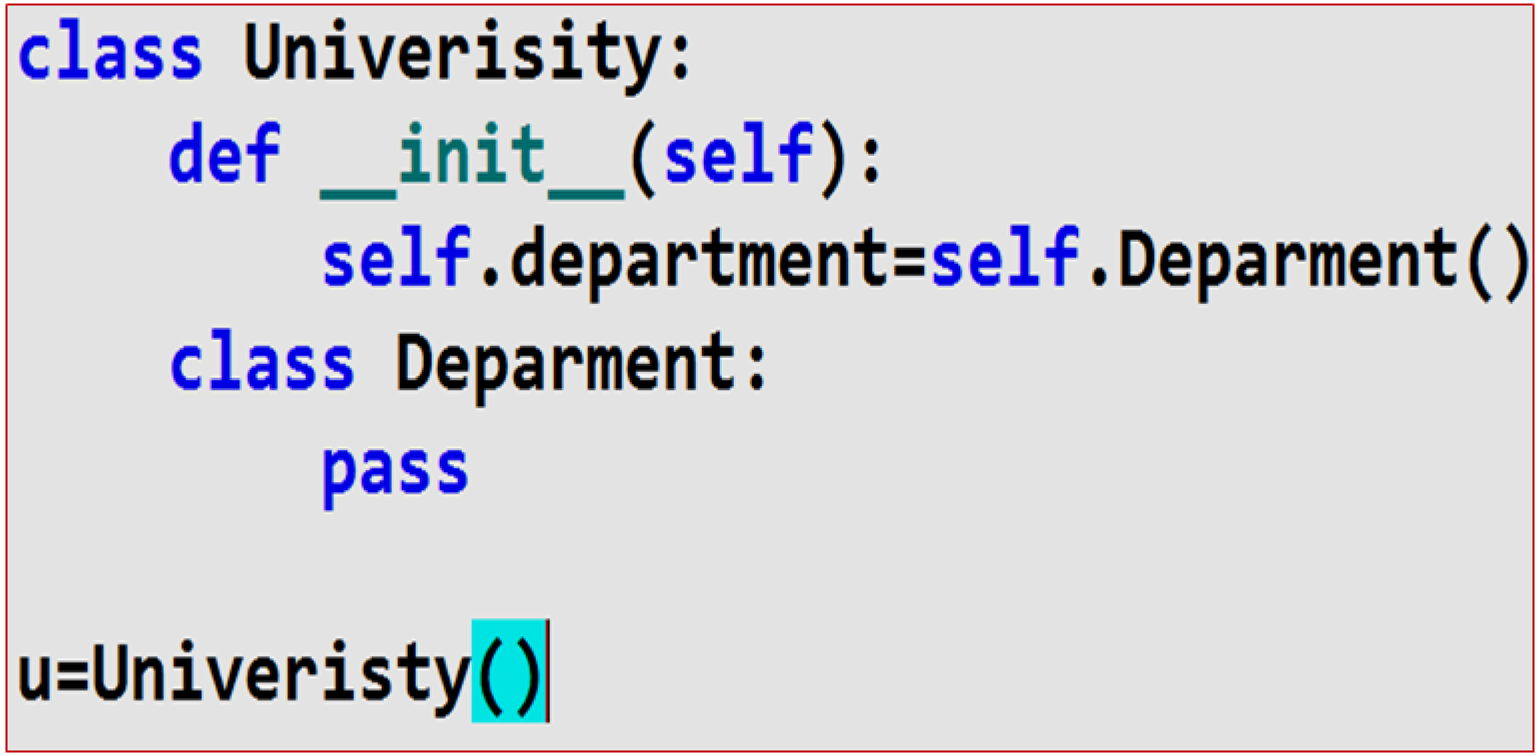
- Without existing container object if there is no chance of existing contained object then the container and contained objects are strongly associated and that strong association is nothing but Composition.

Eg:

- University contains several Departments and without existing university object there is no chance of existing Department object. Hence University and Department objects are strongly associated and this strong association is nothing but Composition.


Department Object
(Contained Object)
University Object
(Container Object)

Coding Example:

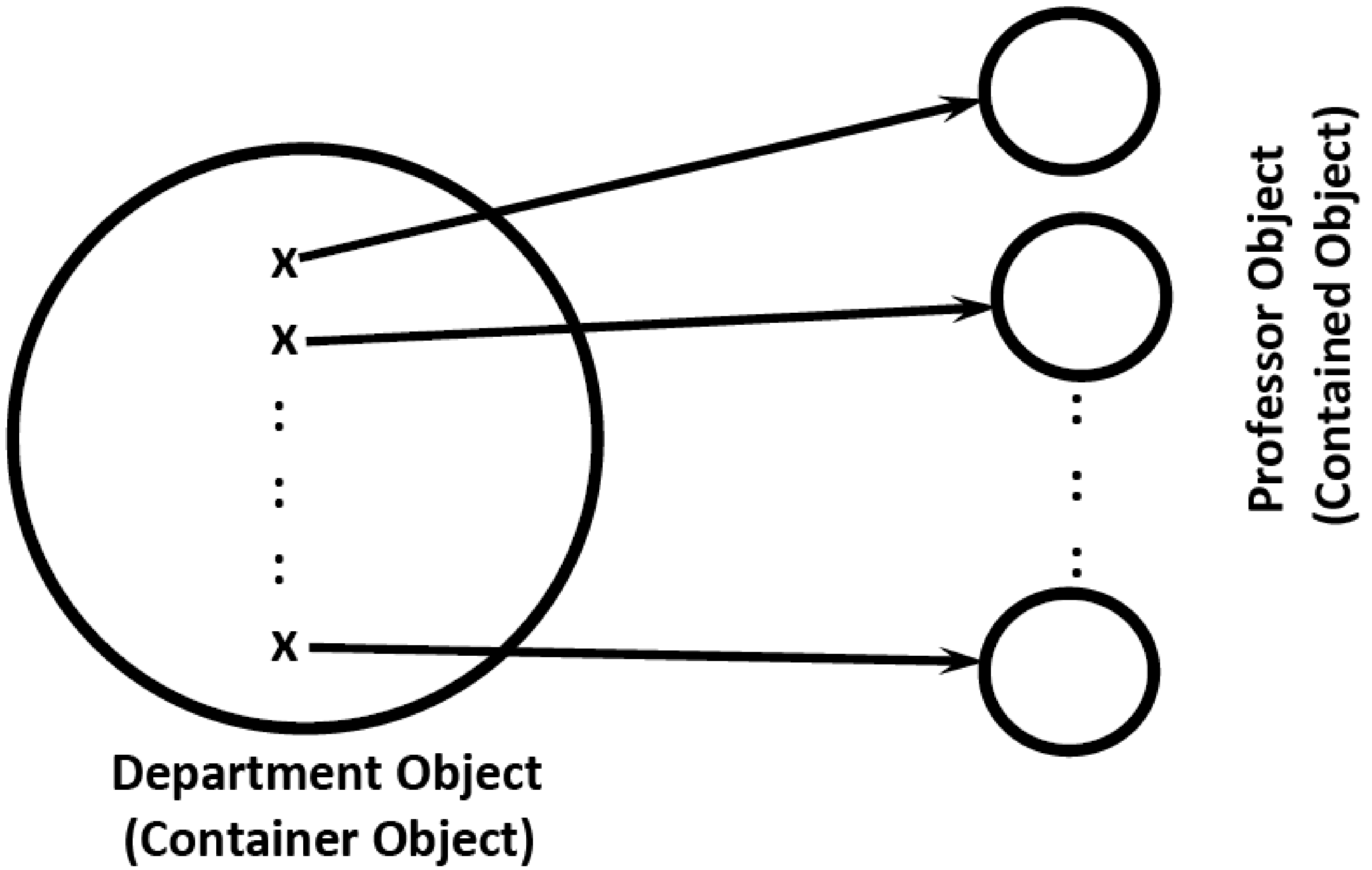

```
class Univerisity:
    def __init__(self):
        self.department=self.Deparment()
    class Deparment:
        pass

u=Univeristy()
```

2. Aggregation:

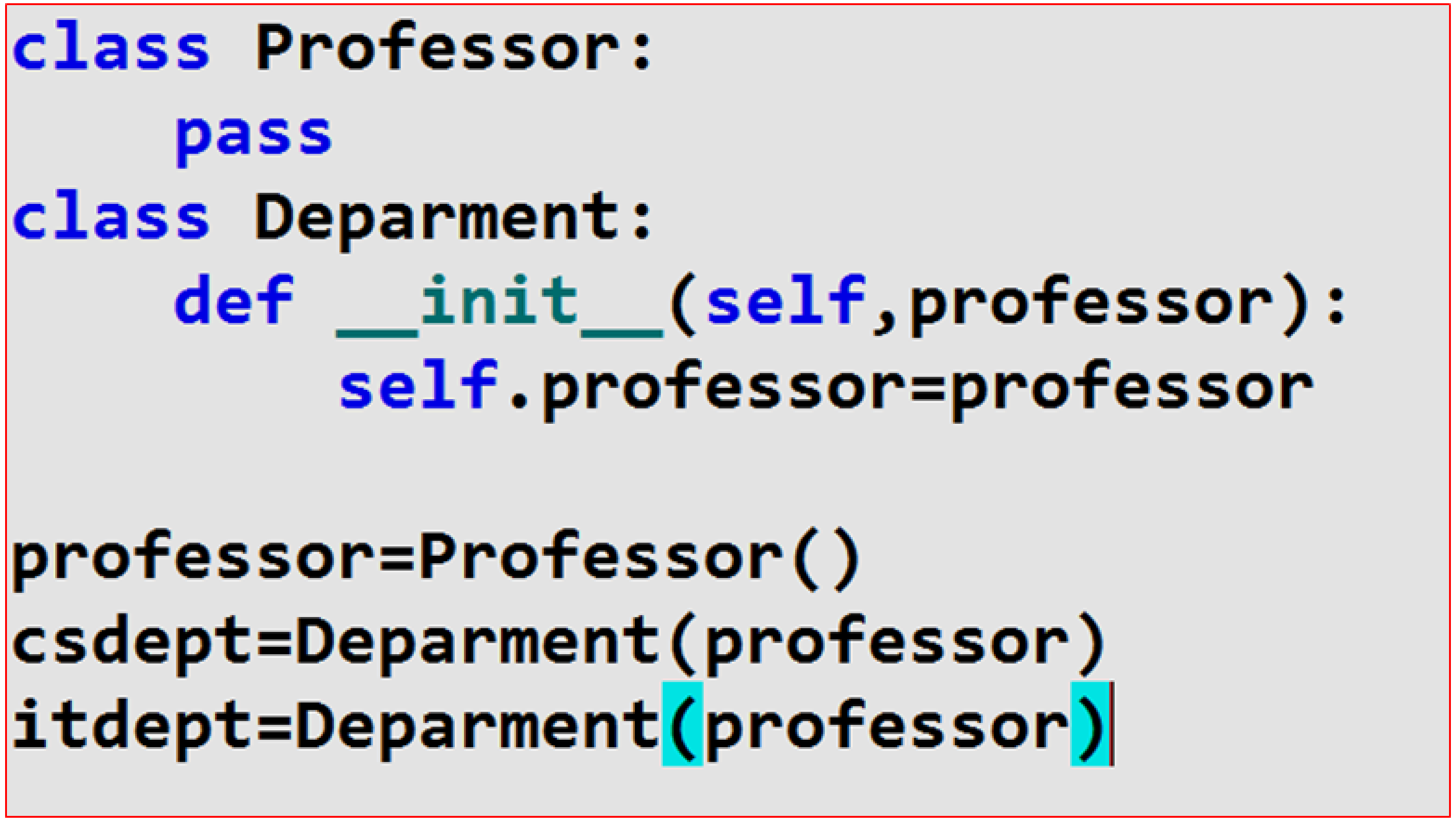
- Without existing container object if there is a chance of existing contained object then the container and contained objects are weakly associated and that weak association is nothing but Aggregation.

Eg:

- Department contains several Professors. Without existing Department still there may be a chance of existing Professor. Hence Department and Professor objects are weakly associated, which is nothing but Aggregation.


X
X
X
Department Object
(Container Object)
Professor Object
(Contained Object)

Coding Example:



```
class Professor:
    pass
class Deparment:
    def __init__(self,professor):
        self.professor=professor

professor=Professor()
csdept=Deparment(professor)
itdept=Deparment(professor)
```

Composition vs Aggregation:

- In Composition objects are strongly associated where as in Aggregation objects are weakly associated.
- In Composition, container object holds directly contained objects, where as in Aggregation container object just holds references of contained objects.

Example Program:

```python
class Student:

    collegeName='RGMCET'

    def __init__(self,name):
        self.name=name
        print(Student.collegeName)

s=Student('Karthi')
print(s.name)
```

```
RGMCET
Karthi
```

Key Observations from the above code:

- In the above example without existing Student object there is no chance of existing his name. Hence Student Object and his name are strongly associated which is nothing but Composition.
- But without existing Student object there may be a chance of existing collegeName. Hence Student object and collegeName are weakly associated which is nothing but Aggregation.

Conclusion:

- The relation between object and its instance variables is always Composition where as the relation between object and static variables is Aggregation.

Note:

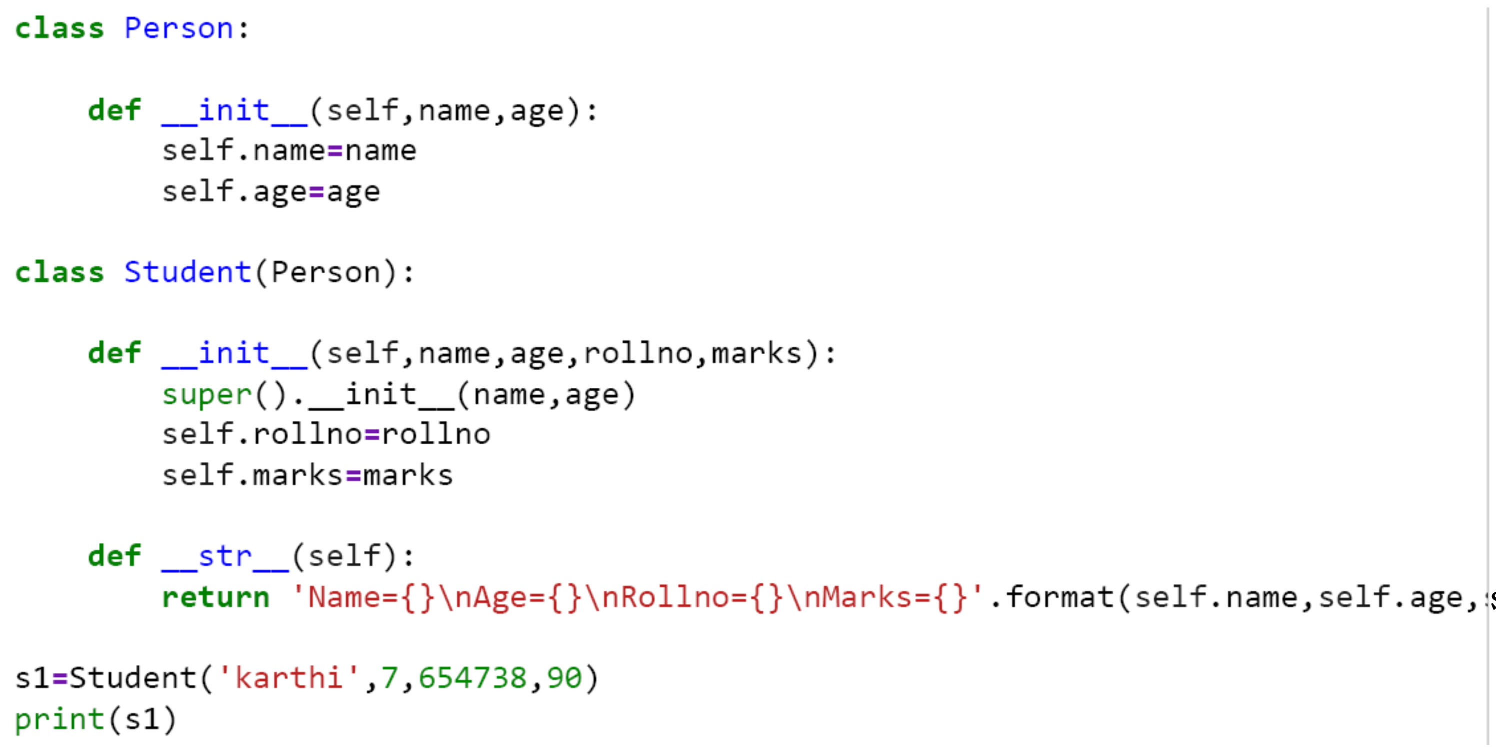
- Whenever we are creating child class object then child class constructor will be executed. If the child class does not contain constructor then parent class constructor will be executed, but parent object won't be created.

Example:

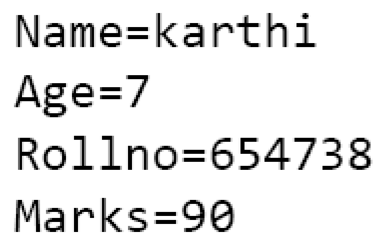
```
class P:

    def __init__(self):
        print(id(self))

class C(P):
    pass

c=C()
print(id(c))
```

```
2770504206536
2770504206536
```

Demo program:



```
class Person:

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

class Student(Person):

    def __init__(self,name,age,rollno,marks):
        super().__init__(name,age)
        self.rollno=rollno
        self.marks=marks

    def __str__(self):
        return 'Name={}\nAge={}\nRollno={}\nMarks={}'.format(self.name,self.age,self .marks)

s1=Student('karthi',7,654738,90)
print(s1)
```



```
Name=karthi
Age=7
Rollno=654738
Marks=90
```

Note: In the above example when ever we are creating child class object both parent and child class constructors got executed to perform initialization of child object.

2. Types of Inheritance

Following are the important Inheritance types:

1. Single Inheritance

2. Multi Level Inheritance

3. Hierarchical Inheritance

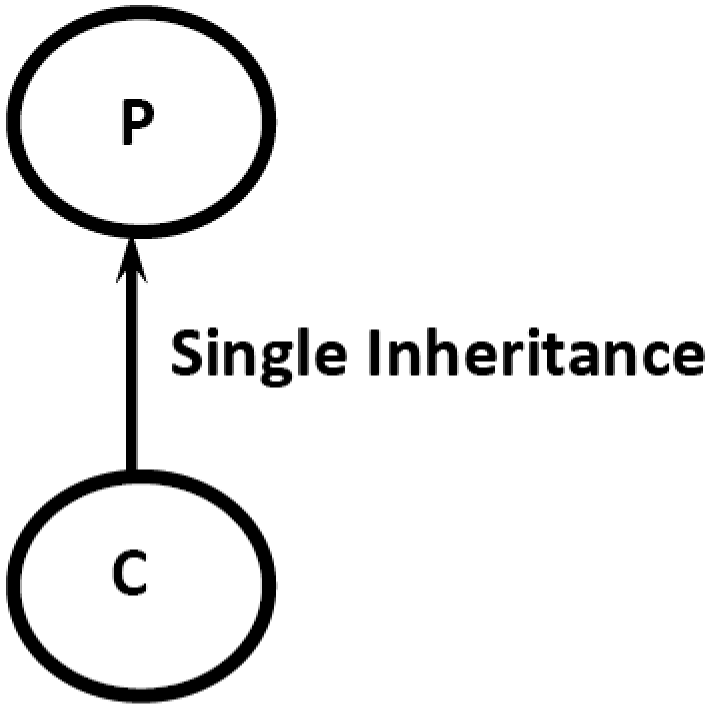
4. Multiple Inheritance

5. Hybrid Inheritance

6. Cyclic Inheritance

1. Single Inheritance:

- The concept of inheriting members from single parent class to single child class is known as single inheritance.
- In simple words, Single parent class and Single child class.



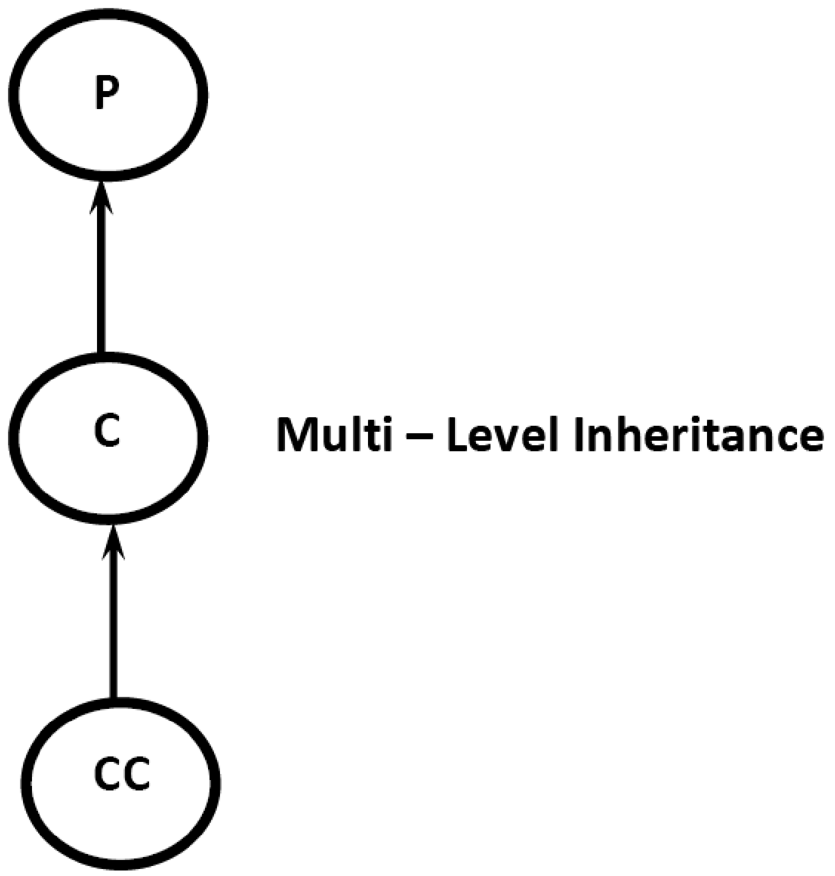
Demo Program on Single Inheritance:

```
class P:

    def m1(self):
        print("Parent Method")

class C(P):
    def m2(self):
        print("Child Method")

c=C()
c.m1()
c.m2()
```

```
Parent Method
Child Method
```

2. Multilevel Inheritance:

- The concept of inheriting the properties from multiple classes to single class with the concept of one after another is known as multilevel inheritance.



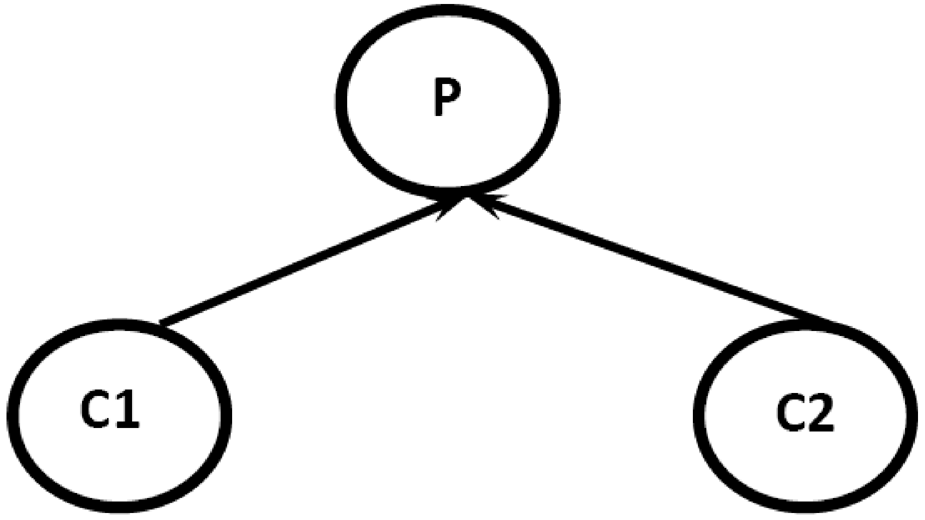
Demo Program on Multi Level Inheritance:

```python
class P:
    def m1(self):
        print("Parent Method")

class C(P):
    def m2(self):
        print("Child Method")

class CC(C):
    def m3(self):
        print("Sub Child Method")

c=CC()
c.m1()
c.m2()
c.m3()
```

```
Parent Method
Child Method
Sub Child Method
```

3. Hierarchical Inheritance:

- The concept of inheriting properties from one class into multiple classes which are present at same level is known as Hierarchical Inheritance.
- In simple words, One Parent but Multiple child classes and all child classes are at same level.



Hierarchical
Inheritance

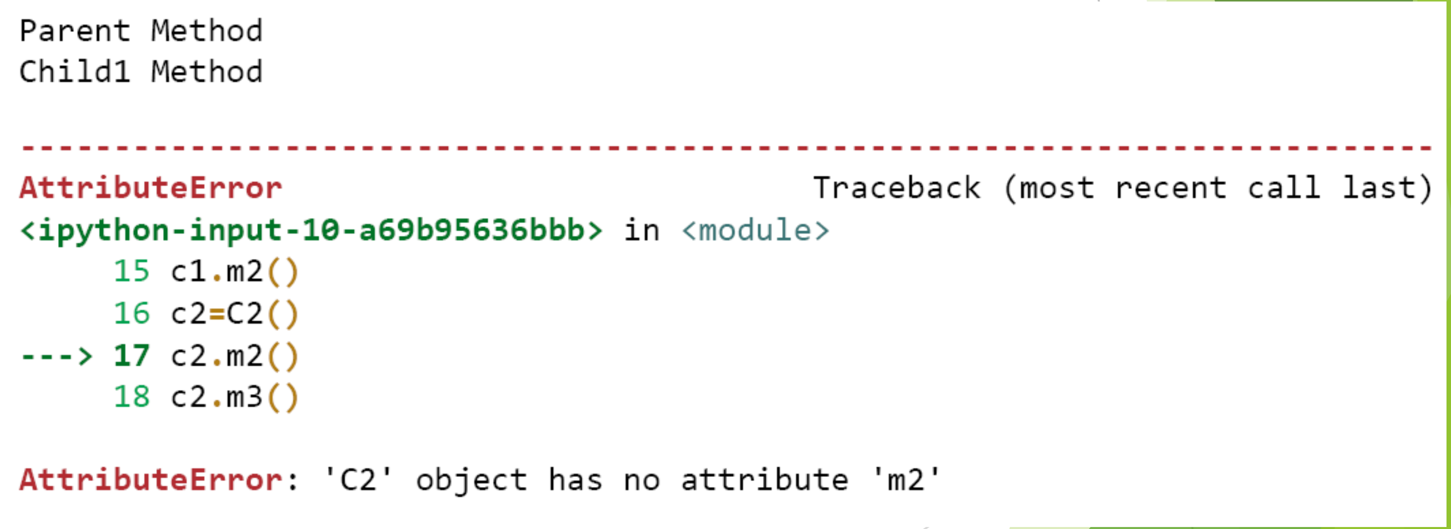
Demo Program 1 on Hierarchical Inheritance:

```
class P:
    def m1(self):
        print("Parent Method")

class C1(P):
    def m2(self):
        print("Child1 Method")

class C2(P):
    def m3(self):
        print("Child2 Method")

c1=C1()
c1.m1()
c1.m2()
c2=C2()
c2.m1()
c2.m3()
```

```
Parent Method
Child1 Method
Parent Method
Child2 Method
```

Demo Program 2 on Hierarchical Inheritance:

```python
class P:
    def m1(self):
        print("Parent Method")

class C1(P):
    def m2(self):
        print("Child1 Method")

class C2(P):
    def m3(self):
        print("Child2 Method")

c1=C1()
c1.m1()
c1.m2()
c2=C2()
c2.m2()
c2.m3()
```



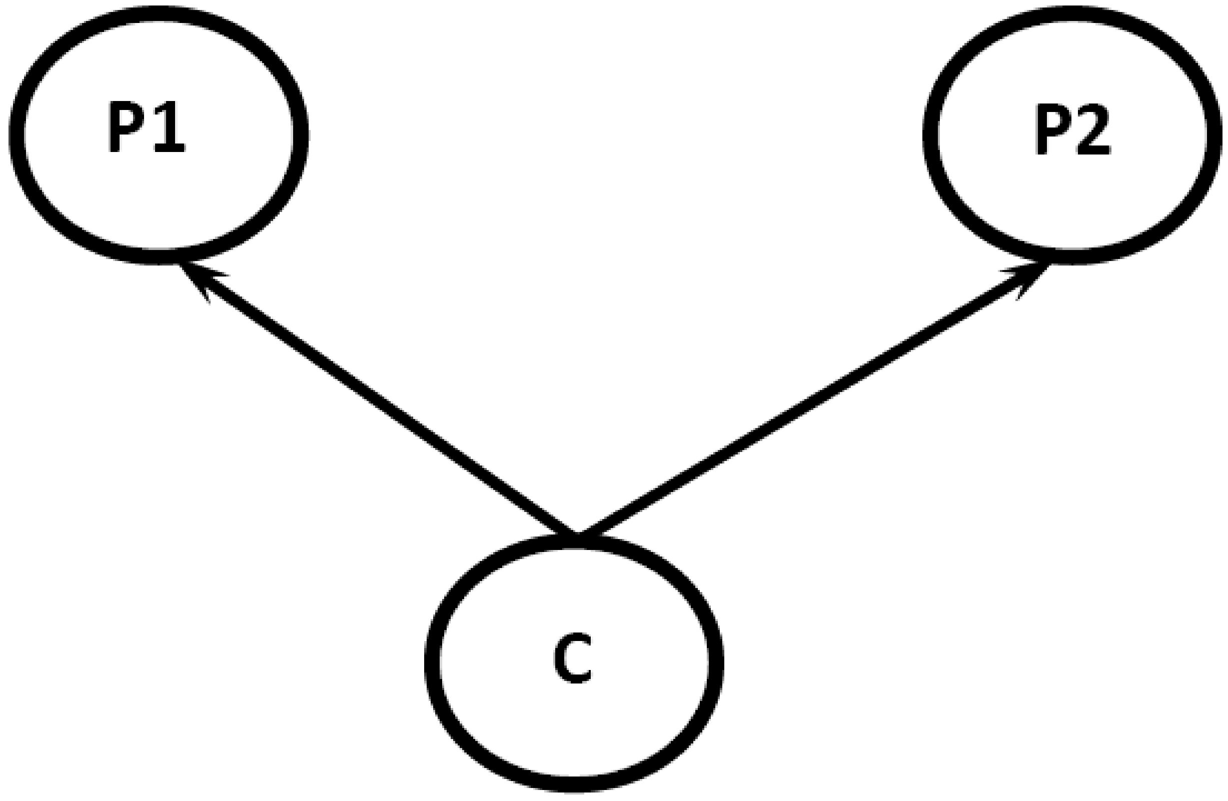
```
Parent Method
Child1 Method

---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-10-a69b95636bbb> in <module>
     15 c1.m2()
     16 c2=C2()
---> 17 c2.m2()
     18 c2.m3()

AttributeError: 'C2' object has no attribute 'm2'
```

4. Multiple Inheritance:

- The concept of inheriting the properties from multiple parent classes into a single child class at a time, is known as multiple inheritance.
- In simple words, it is reverse of Hierarchical Inheritance.

Note:

- **Hierarchical:** One Parent and Multiple Child classes
- **Multiple:** Multiple Parents and Single Child class



Multiple Inheritance

Demo Program on Multiple Inheritance:

```
class P1:
    def m1(self):
        print("Parent1 Method")

class P2:
    def m2(self):
        print("Parent2 Method")

class C(P1,P2):
    def m3(self):
        print("Child2 Method")

c=C()
c.m1()
c.m2()
c.m3()
```

```
Parent1 Method
Parent2 Method
Child2 Method
```

Why Java won't support Multiple Inheritance?

Key Point:

- If the same method is inherited from both parent classes, then Python will always consider the order of Parent classes in the declaration of the child class.

1. **class C(P1,P2):** ===>P1 method will be considered
2. **class C(P2,P1):** ===>P2 method will be considered

Eg 1:

```
class P1:

    def m1(self):
        print("Parent1 Method")

class P2:

    def m1(self):
                print("Parent2 Method")

class C(P1,P2):          # Order is important

    def m2(self):
        print("Child Method")

c=C()
c.m1()
c.m2()
```

```
Parent1 Method
Child Method
```

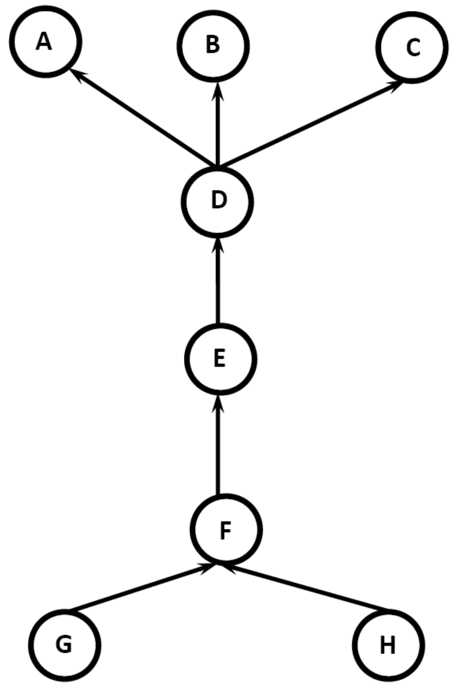
Eg 2:

```
class P1:

    def m1(self):
        print("Parent1 Method")

class P2:

    def m1(self):
                print("Parent2 Method")

class C(P2,P1):                 # Order is important

    def m2(self):
        print("Child Method")

c=C()
c.m1()
c.m2()
```

```
Parent2 Method
Child Method
```

5. Hybrid Inheritance:

- In General, Hybrid means mixing or combination.
- Combination of Single, Multi level, Multiple and Hierarchical inheritances is known as Hybrid Inheritance.



Note: In Hybrid inheritance, method resolution is based on MRO algorithm [Which we will discuss later].

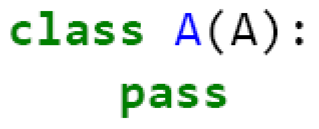
6. Cyclic Inheritance:

- The concept of inheriting members from one class to another class in cyclic way, is called cyclic inheritance.

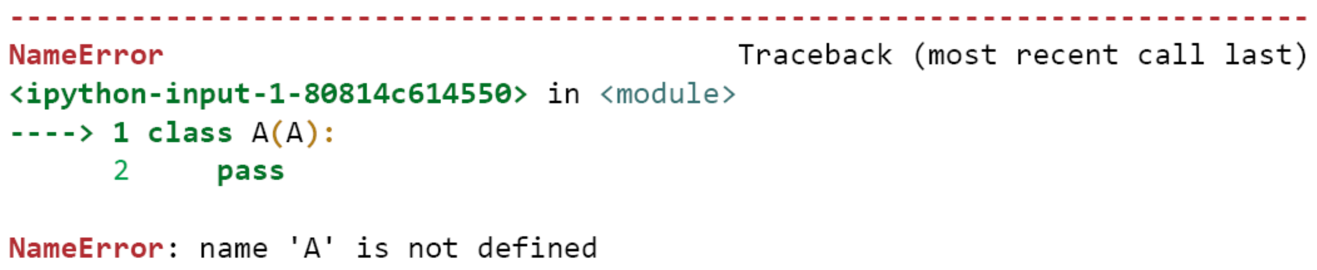
Note:

- Really Cyclic Inheritance is not required. Hence programming languages like java , python won't provide support.

Eg:

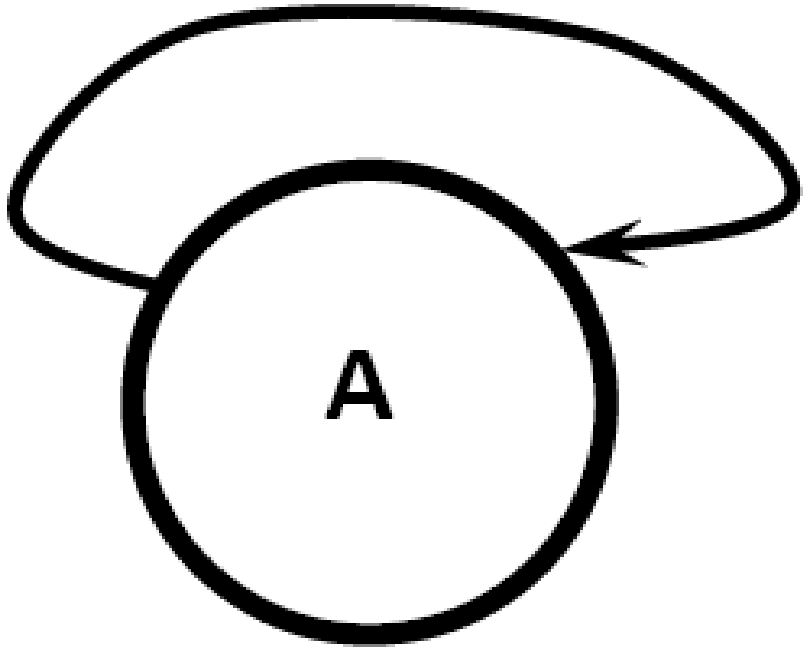


```
class A(A):
    pass
```



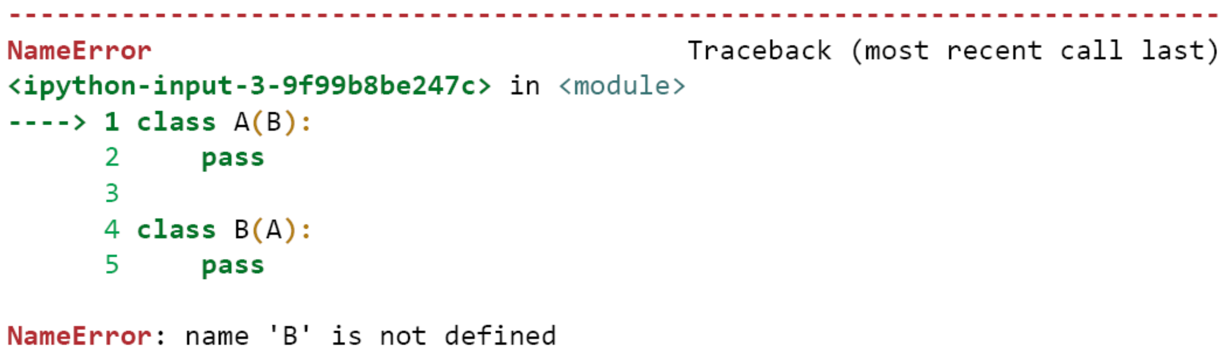
```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-1-80814c614550> in <module>
----> 1 class A(A):
      2     pass

NameError: name 'A' is not defined
```

Diagrammatic representation of previous example:



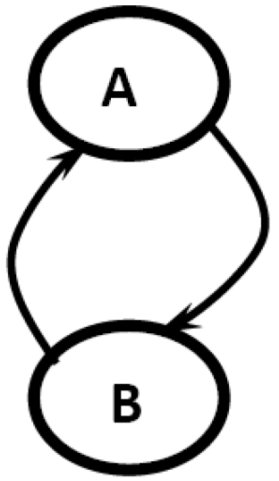
Eg 2:

```
class A(B):
    pass

class B(A):
    pass
```



```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-3-9f99b8be247c> in <module>
----> 1 class A(B):
      2     pass
      3 
      4 class B(A):
      5     pass

NameError: name 'B' is not defined
```

Diagramatic representation of Cyclic Inheritance:



3. Method Resolution Order (MRO) Algorithm

- In Hybrid Inheritance the method resolution order is decided based on MRO algorithm.

Demo Program 1 on Method Resolution Order (MRO):

Consider the following scenario,

$$\begin{tikzcd}
 & A & \\
B \arrow[ur] & & C \arrow[ul] \\
 & D \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

Above diagram contains 4 classes, which involves in the form of Hybrid inheritance.

- Now, If we creates an object of **D** type.

  d = D()

- If we call method 'm1()' by using object 'd'.

  d.m1()

Now the question is, which m1() method will be executed?

- First, it will check in the **D** class for the method m1(). If it is not there, then it will search in the **B** class or **C** class or **A** class.
- In this scenario, from which class, this **m1()** method will be executed will be decided by using Method Resolution Order (MRO) algorithm.

- In the above example, if D doesn't contain m1() method, then check for m1() method in the immediate parents of D (i.e., B & C).
- Then, which is the first parent of D?
- Let us consider B. If B doesn't contain m1() method, then it considers class C for m1() method.
- Because the above scenario is simple, we can find the method resolution in simple manner.
- If you want to know what is the method resolution order (MRO) of any class, there is a way also there. We can find MRO of any class by using **mro()** function.

```
print(classname.mro())
```

Program to find MRO for the taken scenario:

```python
class A:pass
class B(A):pass
class C(A):pass
class D(B,C):pass
print(A.mro())
print(B.mro())
print(C.mro())
print(D.mro())
```

```
[<class '__main__.A'>, <class 'object'>]
[<class '__main__.B'>, <class '__main__.A'>, <class 'object'>]
[<class '__main__.C'>, <class '__main__.A'>, <class 'object'>]
[<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class '__ma
in__.A'>, <class 'object'>]
```

Method Resolution Order (MRO):

- In which order, PVM will give the priority while resolving the methods. That order by default considered as Method Resolution Order (MRO).
- We can find Method Resolution Order (MRO) of any class by using mro() function.

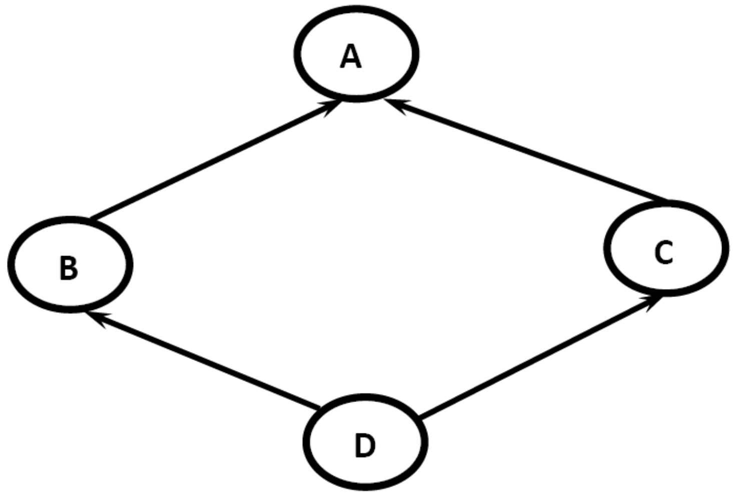
Syntax of mro() function:

```
classname.mro()
```

Examples to find MRO:

I.

```
class A:
    def m1(self):
        print('A class method')
class B(A):
    def m1(self):
        print('B class method')
class C(A):
    def m1(self):
        print('C class method')
class D(B,C):
    def m1(self):
        print('D class method')

d = D()  #DBCAO
d.m1()
```

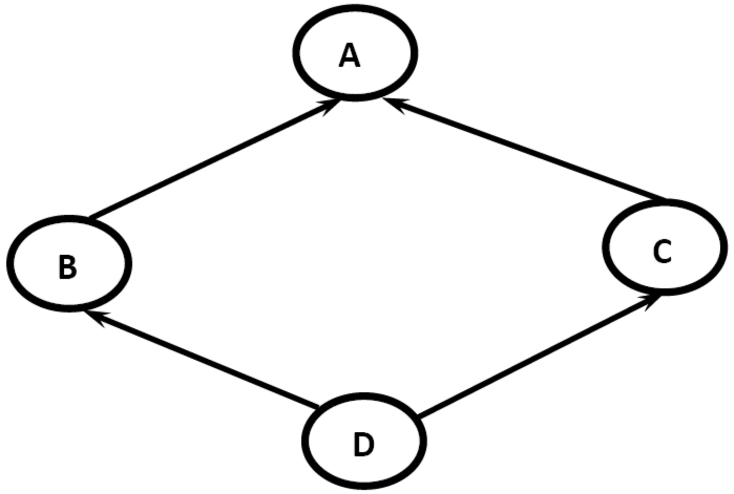


```
D class method
```

Examples to find MRO:

II.

```python
class A:
    def m1(self):
        print('A class method')
class B(A):
    def m1(self):
        print('B class method')
class C(A):
    def m1(self):
        print('C class method')
class D(B,C):
    pass

d = D()  #DBCAO
d.m1()
```

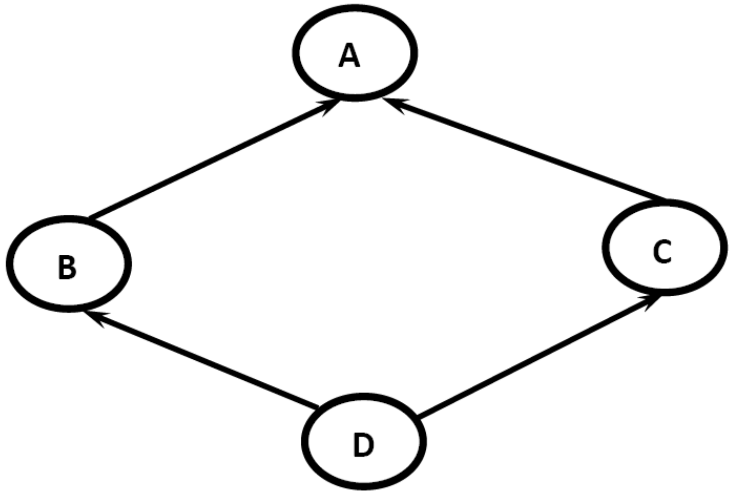

$$\begin{tikzcd} & A & \\ B \arrow[ur] & & C \arrow[ul] \\ & D \arrow[ul] \arrow[ur] & \end{tikzcd}$$

```
B class method
```

Examples to find MRO:

III.

```
class A:
    def m1(self):
        print('A class method')
class B(A):
    pass
class C(A):
    def m1(self):
        print('C class method')
class D(B,C):
    pass

d = D()  #DBCAO
d.m1()
```

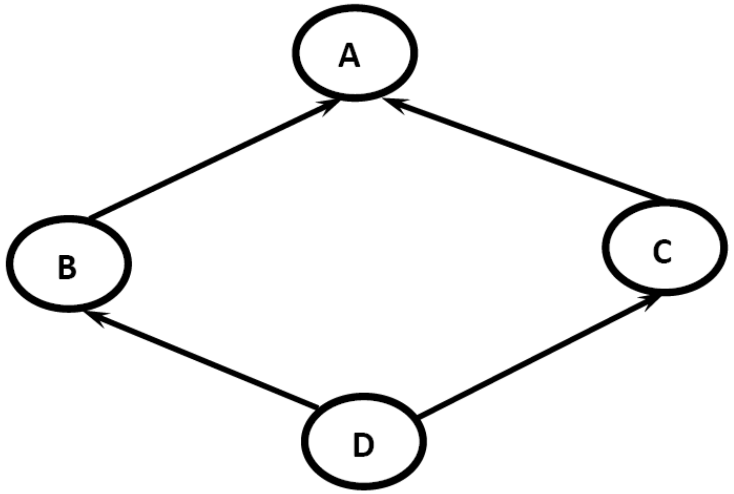


```
C class method
```

Examples to find MRO:

IV.

```
class A:
    def m1(self):
        print('A class method')
class B(A):
    pass
class C(A):
    pass
class D(B,C):
    pass

d = D()  #DBCAO
d.m1()
```

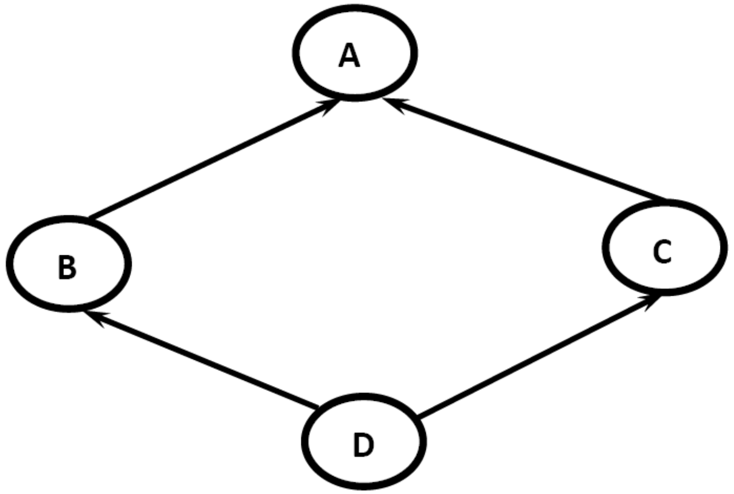


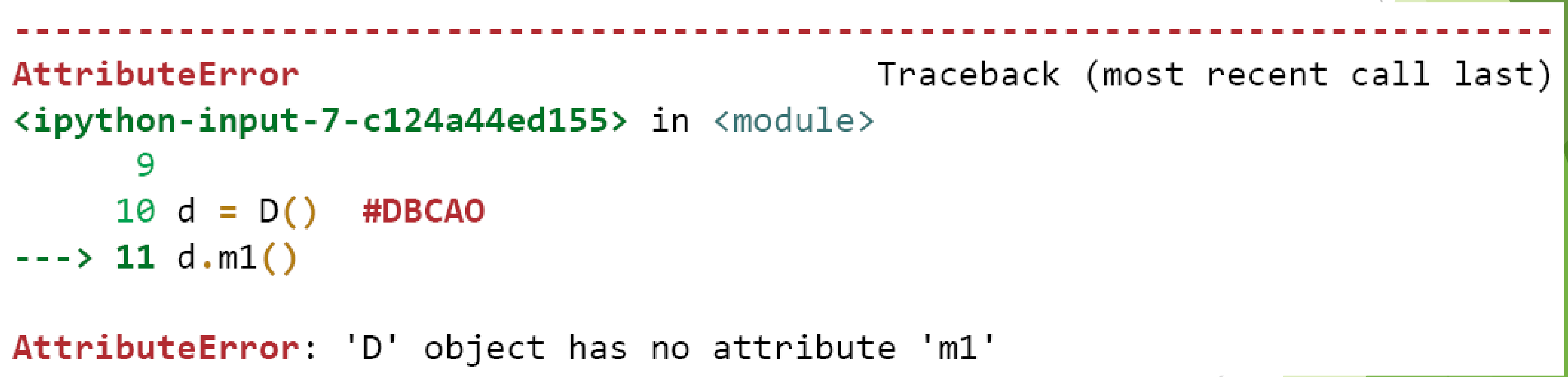
```
A class method
```

Examples to find MRO:

V.

```
class A:
    pass
class B(A):
    pass
class C(A):
    pass
class D(B,C):
    pass

d = D()  #DBCAO
d.m1()
```

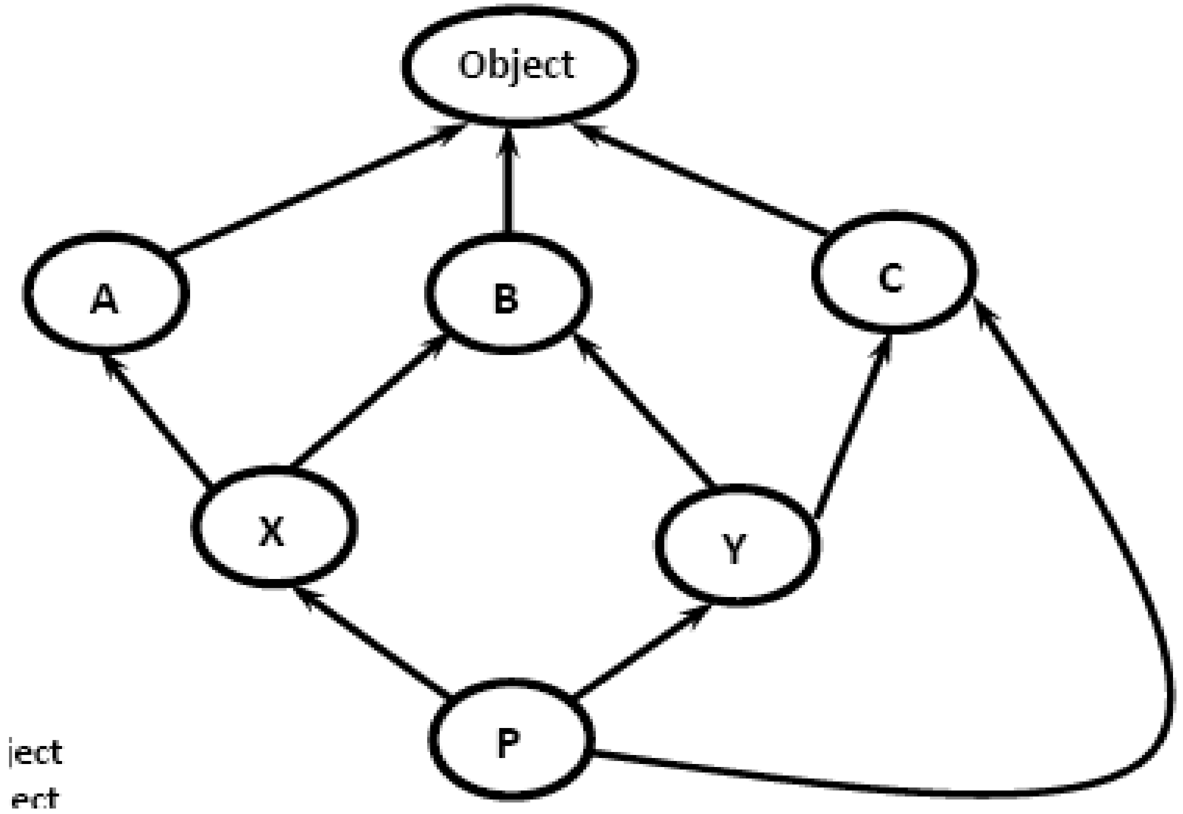




```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-7-c124a44ed155> in <module>
      9
     10 d = D()  #DBCAO
---> 11 d.m1()

AttributeError: 'D' object has no attribute 'm1'
```

Demo Program 2 on Method Resolution Order (MRO):

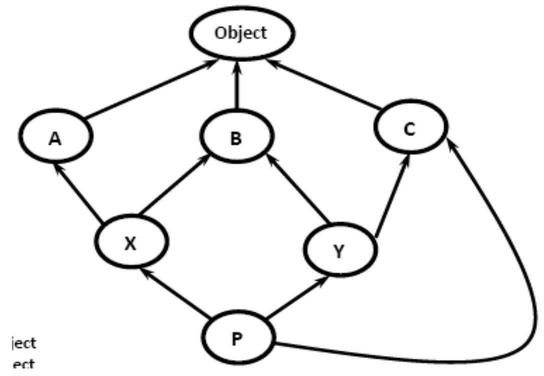
Consider the following scenario,

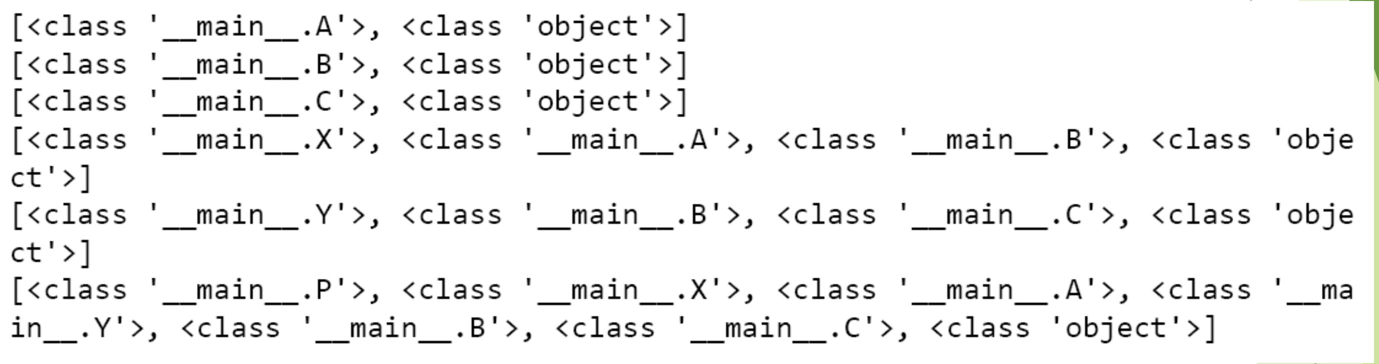


Finding of MRO:

I.

```
class A: pass
class B:pass
class C:pass
class X(A,B):pass
class Y(B,C):pass
class P(X,Y):pass
print(A.mro())  # AO
print(B.mro())  # BO
print(C.mro())  #CO
print(X.mro())  #XABO
print(Y.mro())  #YBCO
print(P.mro())  #PXAYBCO
# IF MULTIPLE LEVELS ARE THERE, WE CAN'T TELL MRO DIRECTLY.
```

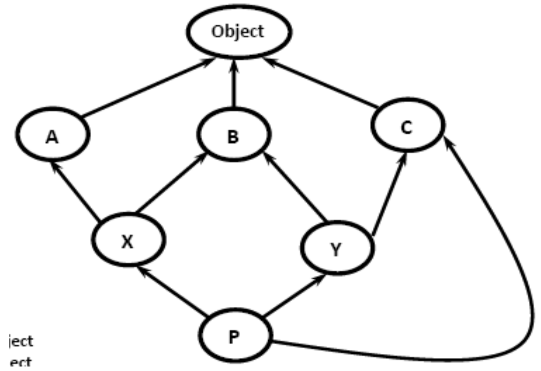




```
[<class '__main__.A'>, <class 'object'>]
[<class '__main__.B'>, <class 'object'>]
[<class '__main__.C'>, <class 'object'>]
[<class '__main__.X'>, <class '__main__.A'>, <class '__main__.B'>, <class 'obje
ct'>]
[<class '__main__.Y'>, <class '__main__.B'>, <class '__main__.C'>, <class 'obje
ct'>]
[<class '__main__.P'>, <class '__main__.X'>, <class '__main__.A'>, <class '__ma
in__.Y'>, <class '__main__.B'>, <class '__main__.C'>, <class 'object'>]
```

II.

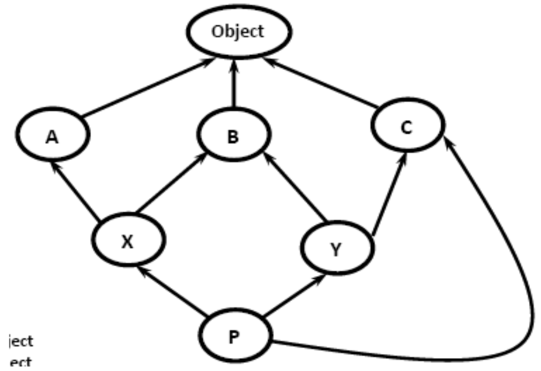
```python
class A:
    def m1(self):
        print('A class Method')

class B:
    def m1(self):
        print('B class Method')

class C:
    def m1(self):
        print('C class Method')

class X(A,B):
    def m1(self):
        print('X class Method')

class Y(B,C):
    def m1(self):
        print('Y class Method')

class P(X,Y,C):
    def m1(self):
        print('P class Method')

p=P()
p.m1()
```



```
P class Method
```

III.

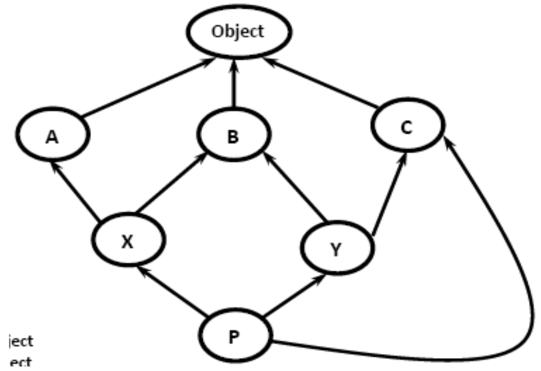
```python
class A:
    def m1(self):
        print('A class Method')

class B:
    def m1(self):
        print('B class Method')

class C:
    def m1(self):
        print('C class Method')

class X(A,B):
    def m1(self):
        print('X class Method')

class Y(B,C):
    def m1(self):
        print('Y class Method')

class P(X,Y,C):
       pass

p=P()
p.m1()
```



```
X class Method
```

IV.

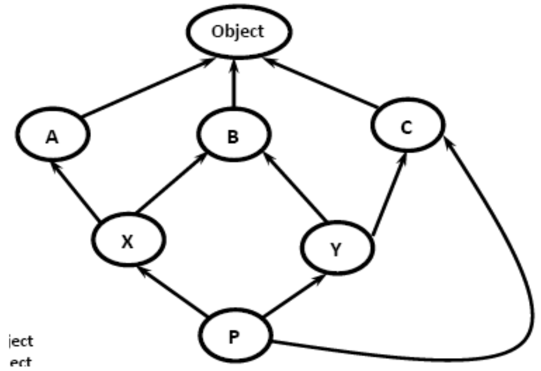
```python
class A:
    def m1(self):
        print('A class Method')

class B:
    def m1(self):
        print('B class Method')

class C:
    def m1(self):
        print('C class Method')

class X(A,B):
    pass

class Y(B,C):
    def m1(self):
        print('Y class Method')

class P(X,Y,C):
        pass

p=P()
p.m1()
```



```
A class Method
```

V.

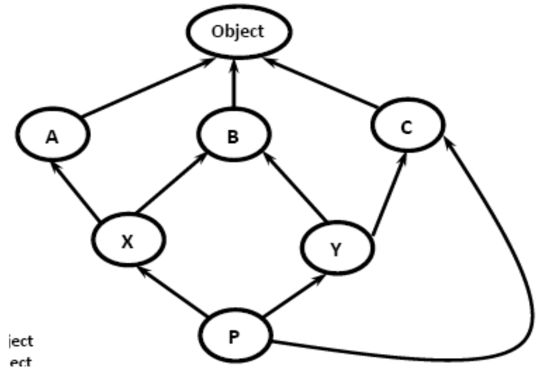
```
class A:
    pass
class B:
    def m1(self):
        print('B class Method')

class C:
    def m1(self):
        print('C class Method')

class X(A,B):
    pass

class Y(B,C):
    def m1(self):
        print('Y class Method')

class P(X,Y,C):
       pass

p=P()
p.m1()
```



```
Y class Method
```

VI.

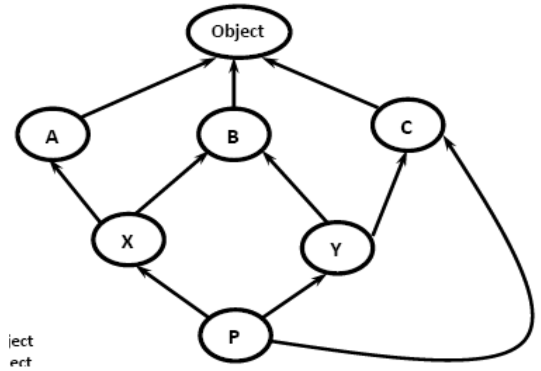
```
class A:
    pass
class B:
    def m1(self):
        print('B class Method')

class C:
    def m1(self):
        print('C class Method')

class X(A,B):
    pass

class Y(B,C):
    pass

class P(X,Y,C):
       pass

p=P()
p.m1()
```



```
B class Method
```

VII.

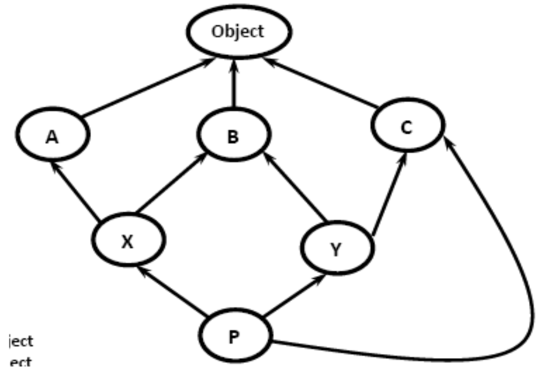
```
class A:
    pass
class B:
    pass

class C:
    def m1(self):
        print('C class Method')

class X(A,B):
    pass

class Y(B,C):
    pass

class P(X,Y,C):
       pass

p=P()
p.m1()
```

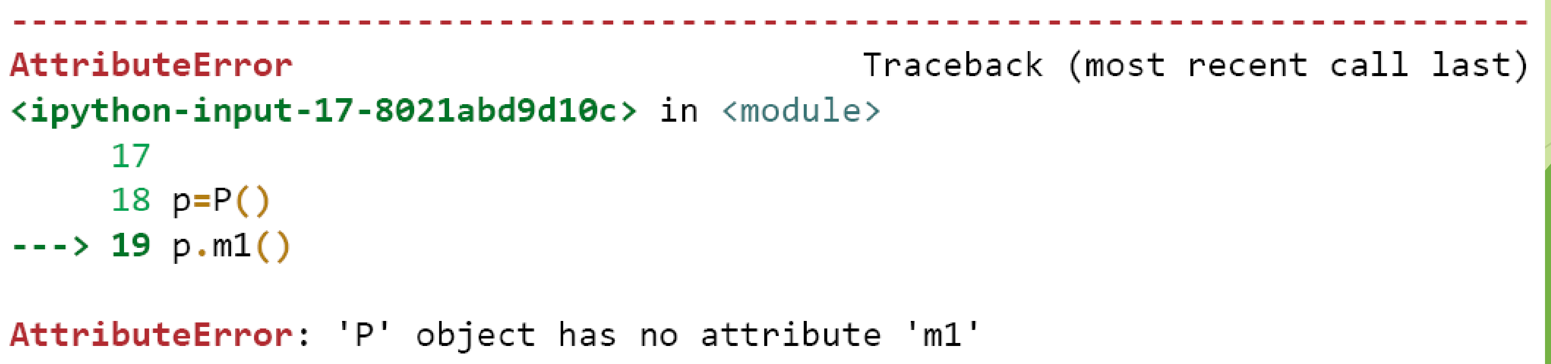


```
C class Method
```

VIII.

```
class A:
    pass
class B:
    pass

class C:
    pass

class X(A,B):
    pass

class Y(B,C):
    pass

class P(X,Y,C):
       pass

p=P()
p.m1()
```





```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-17-8021abd9d10c> in <module>
     17
     18 p=P()
---> 19 p.m1()

AttributeError: 'P' object has no attribute 'm1'
```

- How we are getting this order? What algorithm internally it follows and how that algorithm internally implemented?

We should know the answer for these questions.

MRO Algorithm

- This algorithm is also known as C3 algorithm.
- Samuel Pedroni proposed this algorithm.
- It follows DLR(Depth First Left to Right) approach.

    * Child will get more priority than parent.

    * Left Parent will get more priority than Right Parent.

Formula for calculating the MRO of any class 'X' is :

MRO(X) = X + Merge(MRO(P1),MRO(P2),...,ParentList)

Here, P1,P2 and P3 are the immediate parents of class 'X'.

Note:

- Keep this formula in your mind. We will discuss about this in later sessions.

Key Terminology with respect to MRO Algorithm:

1. Head Element
2. Tail Part

If you want to work with this algorithm merge operation, compulsorily you need to make use of the above terminology.

Assume C1,C2,C3,...are classes

In the List: C1C2C3C4....

First element is considered as Head Element and Remaining is considered as Tail part.

Head Element:

C1

Tail Part:

C2C3C4...

How Merge operation works in MRO Algorithm?

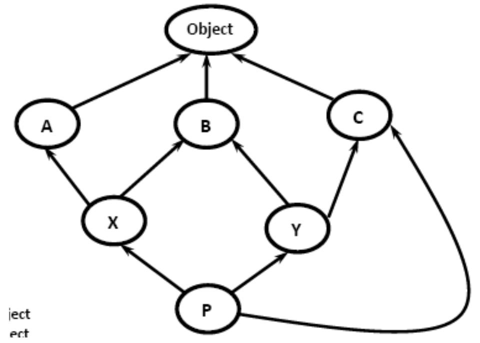
1. Take Head of first list.

2. If the head is not in the tail part of any other list , then add this head element to the result and remove it from all the lists.

3. If the head present in tail part of any other list, then consider head element of the next list and continue same process.

Demo Program 3 on Method Resolution Order (MRO):

Consider the following Scenario,



```python
class A: pass
class B:pass
class C:pass
class X(A,B):pass
class Y(B,C):pass
class P(X,Y):pass
print(A.mro())  # AO
print(B.mro())  # BO
print(C.mro())  #CO
print(X.mro())  #XABO
print(Y.mro())  #YBCO
print(P.mro())  #PXAYBCO
```

```
[<class '__main__.A'>, <class 'object'>]
[<class '__main__.B'>, <class 'object'>]
[<class '__main__.C'>, <class 'object'>]
[<class '__main__.X'>, <class '__main__.A'>, <class '__main__.B'>, <class 'obje
ct'>]
[<class '__main__.Y'>, <class '__main__.B'>, <class '__main__.C'>, <class 'obje
ct'>]
[<class '__main__.P'>, <class '__main__.X'>, <class '__main__.A'>, <class '__ma
in__.Y'>, <class '__main__.B'>, <class '__main__.C'>, <class 'object'>]
```

Finding mro(P) by using C3 algorithm:

Formula:

MRO(X)=X+ Merge(MRO(P1),MRO(P2),...,ParentList)

mro(P) = P+Merge(mro(X),mro(Y),mro(C),XYC)

mro(P) = P+Merge(XABO,YBCO,CO,XYC)

mro(P) = P+X+Merge(ABO,YBCO,CO,YC)

mro(P) = P+X+A+Merge(BO,YBCO,CO,YC)

mro(P) = P+X+A+Y+Merge(BO,BCO,CO,C)

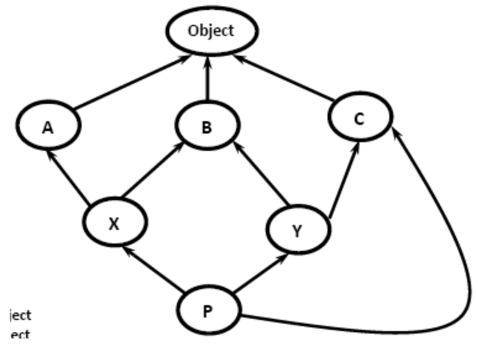
mro(P) = P+X+A+Y+B+Merge(O,CO,CO,C)

mro(P) = P+X+A+Y+B+C+Merge(O,O,O)

mro(P) = P+X+A+Y+B+C+O

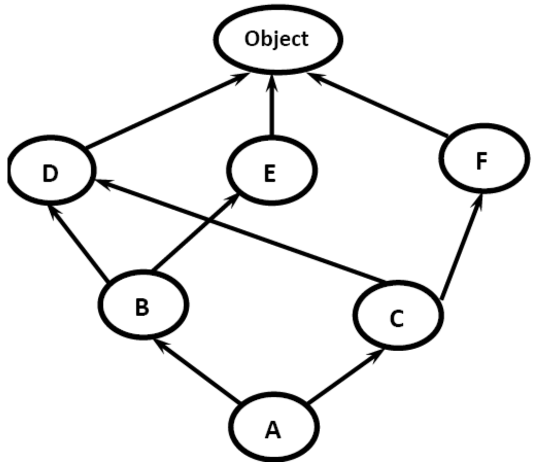
i.e., mro(P)= PXAYBCO

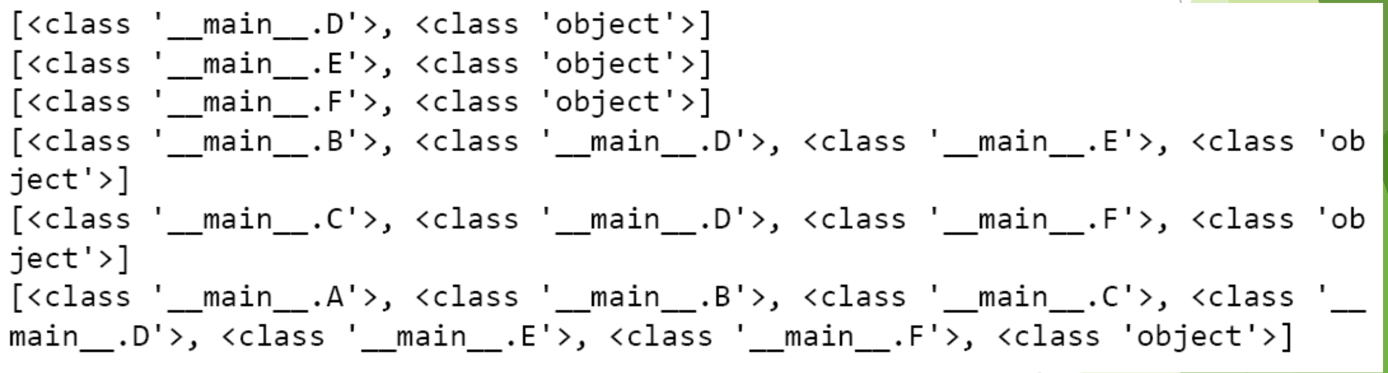
Note: Prooved with our output



Demo Program 4 on Method Resolution Order (MRO) Algorithm:

```
class D:pass
class E:pass
class F:pass
class B(D,E):pass
class C(D,F):pass
class A(B,C):pass
print(D.mro())   # DO
print(E.mro())   # EO
print(F.mro())   # FO
print(B.mro())   # BDEO
print(C.mro())   # CDFO
print(A.mro())   # ABCDEFO
```





```
[<class '__main__.D'>, <class 'object'>]
[<class '__main__.E'>, <class 'object'>]
[<class '__main__.F'>, <class 'object'>]
[<class '__main__.B'>, <class '__main__.D'>, <class '__main__.E'>, <class 'ob
ject'>]
[<class '__main__.C'>, <class '__main__.D'>, <class '__main__.F'>, <class 'ob
ject'>]
[<class '__main__.A'>, <class '__main__.B'>, <class '__main__.C'>, <class '__
main__.D'>, <class '__main__.E'>, <class '__main__.F'>, <class 'object'>]
```

Calculation of mro(A):

Formula: MRO(X)=X+Merge(MRO(P1),MRO(P2),...,ParentList)

mro(A) = A + merge(mro(B),mro(C),BC)

mro(A) = A + merge(BDEO,CDFO,BC)

mro(A) = A + B + merge(DEO,CDFO,C)

mro(A) = A + B + C + merge(DEO,DFO)

mro(A) = A + B + C + D + merge(EO,FO)

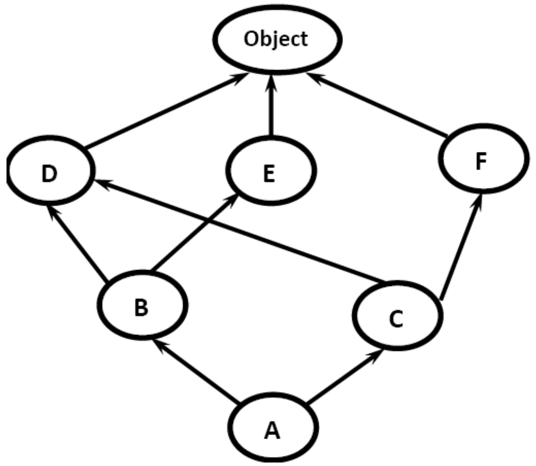
mro(A) = A + B + C + D + E + merge(O,FO)

mro(A) = A + B + C + D + E + F + merge(O,O)

mro(A) = A + B + C + D + E + F + O

mro(A) = ABCDEFO

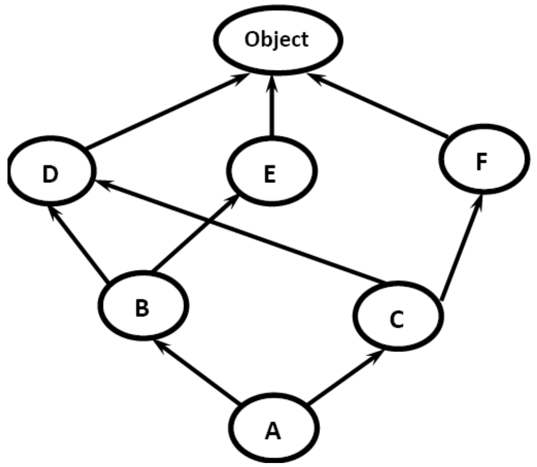
Prooved



Proof through Program:

I.

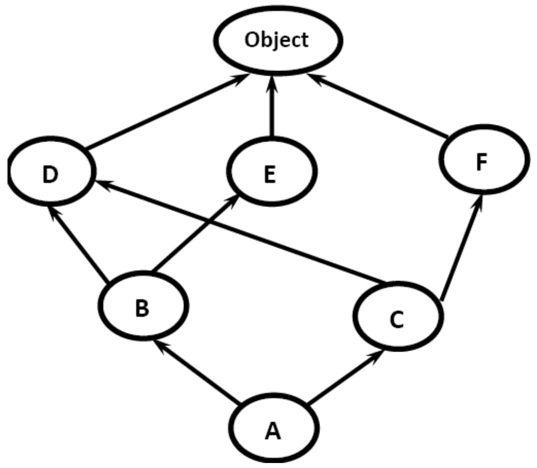
```
class D:
    def m1(self):
        print('D Class Method')
class E:
    def m1(self):
        print('E Class Method')
class F:
    def m1(self):
        print('F Class Method')
class B(D,E):
    def m1(self):
        print('B Class Method')
class C(D,F):
    def m1(self):
        print('C Class Method')
class A(B,C):
    def m1(self):
        print('A Class Method')

a = A()
a.m1()
```



```
A Class Method
```

II.

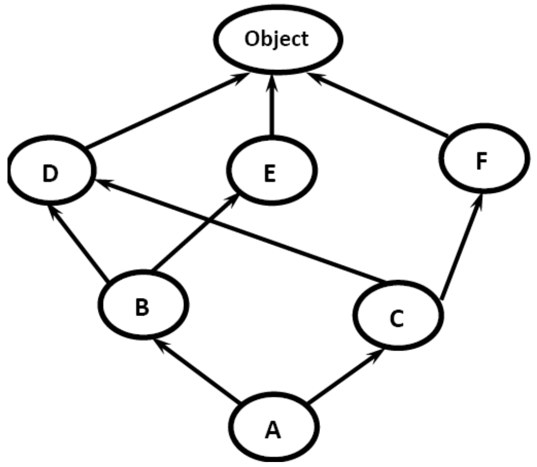
```
class D:
    def m1(self):
        print('D Class Method')
class E:
    def m1(self):
        print('E Class Method')
class F:
    def m1(self):
        print('F Class Method')
class B(D,E):
    def m1(self):
        print('B Class Method')
class C(D,F):
    def m1(self):
        print('C Class Method')
class A(B,C):
    pass
a = A()
a.m1()
```



```
B Class Method
```

III.

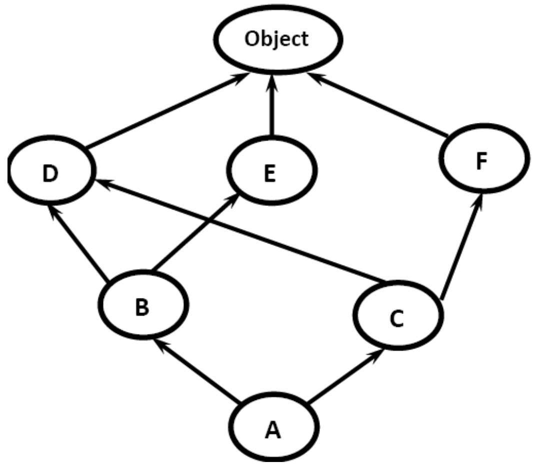
```
class D:
    def m1(self):
        print('D Class Method')
class E:
    def m1(self):
        print('E Class Method')
class F:
    def m1(self):
        print('F Class Method')
class B(D,E):
    pass
class C(D,F):
    def m1(self):
        print('C Class Method')
class A(B,C):
    pass
a = A()
a.m1()
```



```
C Class Method
```

IV.

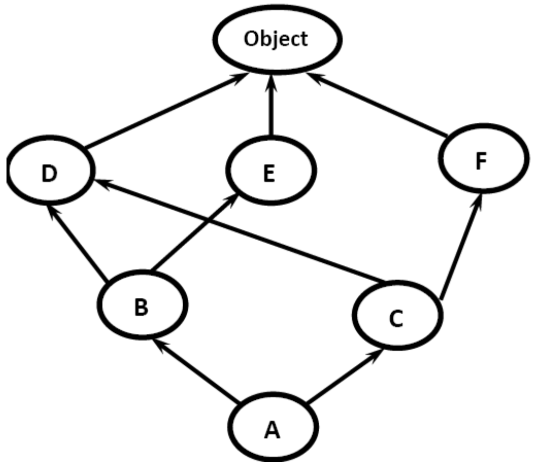
```
class D:
    def m1(self):
        print('D Class Method')
class E:
    def m1(self):
        print('E Class Method')
class F:
    def m1(self):
        print('F Class Method')
class B(D,E):
    pass
class C(D,F):
    pass
class A(B,C):
    pass
a = A()
a.m1()
```



```
D Class Method
```

V.

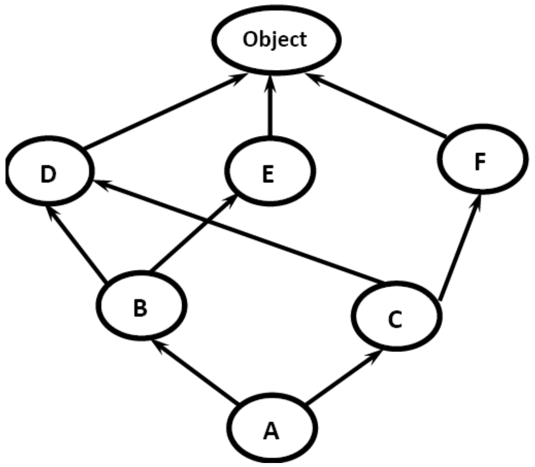
```
class D:
    pass
class E:
    def m1(self):
        print('E Class Method')
class F:
    def m1(self):
        print('F Class Method')
class B(D,E):
    pass
class C(D,F):
    pass
class A(B,C):
    pass
a = A()
a.m1()
```



```
E Class Method
```

VI.

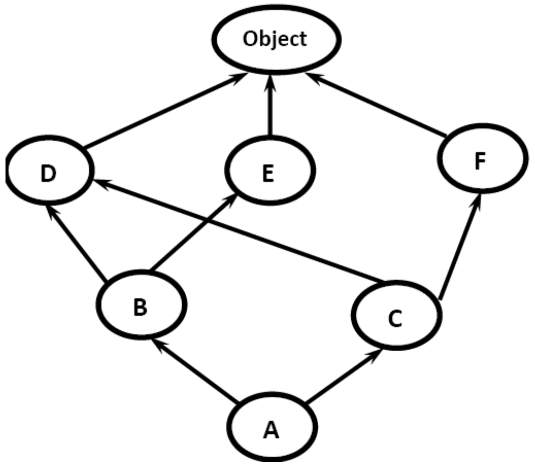
```
class D:
    pass
class E:
    pass
class F:
    def m1(self):
        print('F Class Method')
class B(D,E):
    pass
class C(D,F):
    pass
class A(B,C):
    pass
a = A()
a.m1()
```

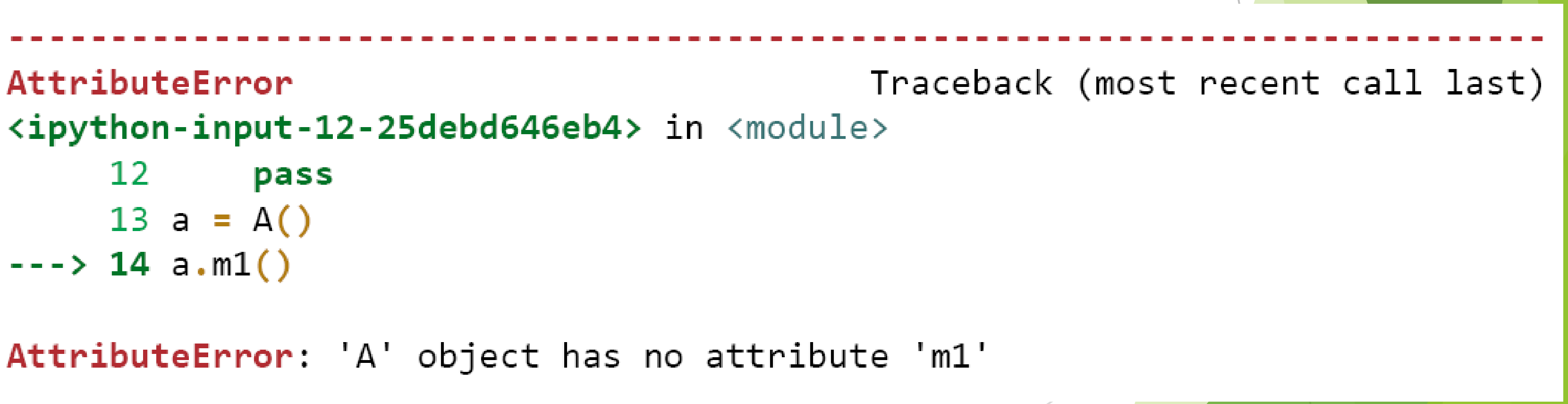


```
F Class Method
```

VII.

```
class D:
        pass
class E:
        pass
class F:
        pass
class B(D,E):
        pass
class C(D,F):
        pass
class A(B,C):
        pass
a = A()
a.m1()
```





```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-12-25debd646eb4> in <module>
     12     pass
     13 a = A()
---> 14 a.m1()

AttributeError: 'A' object has no attribute 'm1'
```

Any question?

If you try to practice programs yourself, then you will learn many things automatically

Spend few minutes and then enjoy the study

Thank You