(7.3) Composition, Inheritance, MRO, Super()

1. Relationship between class

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

1. By Composition (Has-A Relationship)

2. By Inheritance (IS-A Relationship)

1. By Composition (Has-A Relationship)

By using Class Name or by creating object we can access members of one class inside another class is nothing but composition (Has-A Relationship).
The main advantage of Has-A Relationship is Code Reusability

Program - 1

In [2]:
```
class Engine:
    a=10
    def __init__(self):
        self.b=20
    def m1(self):
        print('Engine Specific Functionality')

class Car:
    def __init__(self):
        self.engine=Engine()
    def m2(self):
        print('Car using Engine Class Functionality')
        print(self.engine.a)
        print(self.engine.b)
        self.engine.m1()

c=Car()
c.m2()
```

```
Car using Engine Class Functionality
10
20
Engine Specific Functionality
```

Program - 2

In [4]:
```
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.color

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()

c=Car("Innova","2.5V","Grey")
e=Employee('Durga',10000,c)
e.empinfo()
```

```
Employee Name: Durga
Employee Number: 10000
Employee Car Info:
Car Name:Innova , Model:2.5V and Color:Grey
```

program - 3

In [5]:

```
class X:
    a=10
    def __init__(self):
        self.b=20
    def m1(self):
        print("m1 method of X class")

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

What ever 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 and we can extend existing functionality with some more extra functionality.

class childclass(parentclass):

Parent Class Also known as Super Class and Child Class is also known as Sub-Class

Parent Class Also known as Base Class and Child Class is also known as Derived Class

Program - 1

In [10]:
```
class P:
    a=10
    def __init__(self):
        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):
    def m4(self):
        print('This is child class')

c=C()
print(c.a)
print(c.b)
c.m1()
c.m2()
c.m3()
c.m4()
```

```
10
20
Parent instance method
Parent class method
Parent static method
This is child class
```

Program - 2

In [14]:
```
class Person:
    def __init__(self,name,age):
        self.name=name
        self.age=age
    def eatndrink(self):
        print('Eat Biryani and Drink Beer')

class Employee(Person):
    def __init__(self,name,age,eno,esal):
        super().__init__(name,age)     # Line - 1
        self.eno=eno
        self.esal=esal
    def work(self):
        print("Coding Python is very easy just like drinking Chilled Beer")
    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('Durga', 48, 100, 10000)
e.eatndrink()
```

```
e.work()
e.empinfo()
```

```
Eat Biryani and Drink Beer
Coding Python is very easy just like drinking Chilled Beer
Employee Name: Durga
Employee Age: 48
Employee Number: 100
Employee Salary: 10000
```

Note :

If we comment Line-1 then variable 'name' and 'age' is not available to the child class.

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.

3. IS-A vs HAS-A Relationship

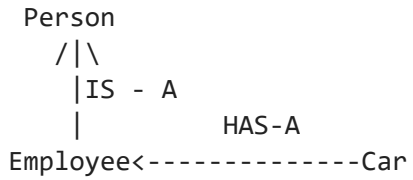
If we want to extend existing functionality with some more extra functionality then we should go for IS-A Relationship

If we dont 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



In [18]:

```
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.color

class Person:
    def __init__(self,name,age):
        self.name=name
        self.age=age
    def eatndrink(self):
        print('Eat Biryani and Drink Beer')

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 very easy just like drinking Chilled Beer")
```

```
    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()


c=Car("Innova","2.5V","Grey")
e=Employee('Durga',48,100,10000,c)
e.eatndrink()
e.work()
e.empinfo()
```

```
Eat Biryani and Drink Beer
Coding Python is very easy just like drinking Chilled Beer
Employee Name: Durga
Employee Age: 48
Employee Number: 100
Employee Salary: 10000
Employee Car Info:
Car Name:Innova , Model:2.5V and Color:Grey
```

In the above example Employee class extends Person class functionality but just uses Car class functionality.

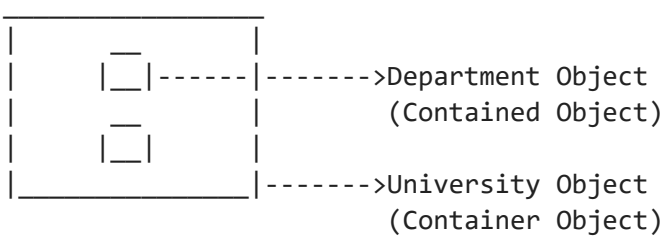
4. Composition vs Aggregation

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.

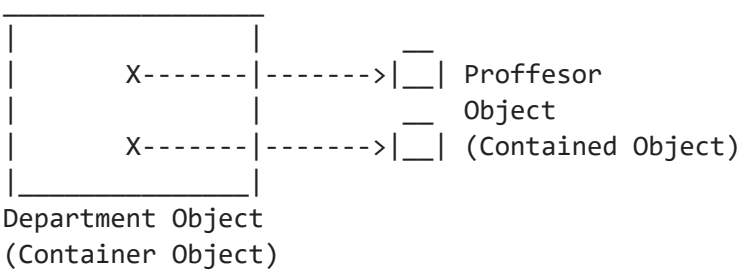


Aggregation

> 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.



In [19]:
```
class Student:
    collegeName='DURGASOFT'
    def __init__(self,name):
        self.name=name

print(Student.collegeName)
s=Student('Durga')
print(s.name)
```

```
DURGASOFT
Durga
```

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

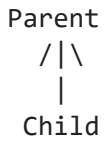
The relation between object and its instance variables is always Composition where as the relation between object and static variables is Aggregation.

2. Types of Inheritance

Single Level Inheritance
Multi Level Inheritance
Hierarchical Inheritance
Multiple Inheritance
Hybrid Inheritance

1. Single Level Inheritance

one parent one child



Program - 1

In [2]:
```
class Parent:
    def bike(self):
        print("Parent class has a hero honda splendra")
    def car(self):
        print("Parent has Maruti Alto 800")
    def home(self):
        print("A old Haveli")

class Child(Parent):
    def mobile(self):
        print("Child has a mobile iphone 10 R")


c = Child()
c.bike()
c.car()
c.mobile()
c.home()
```

```
Parent class has a hero honda splendra
Parent has Maruti Alto 800
Child has a mobile iphone 10 R
A old Haveli
```

In [3]:
```
print(dir(Parent))
```

```
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__
ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le_
_', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr_
_', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'bike', 'ca
r', 'home']
```

In [4]:
```
print(dir(Child))
```

```
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__
ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le_
_', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr_
_', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'bike', 'ca
r', 'home', 'mobile']
```

Program - 2

In [5]:
```
class A:
    def __init__(self,name):
        self.name = name
    def __str__(self):
        return self.name.title()
    def get_name(self):
        return self.name
    def set_name(self,name):
        self.name = name

class B(A):
    pass # A.__init__, A.__str__, A.get_name, A.set_name


a = B("Pankaj Yadav")
print(a)
print(a.get_name())
```

```
Pankaj Yadav
Pankaj Yadav
```

In [6]:
```
a.set_name('Sachin Yadav')
print(a)
```

```
Sachin Yadav
```

In [7]:
```
print(dir(A))
```

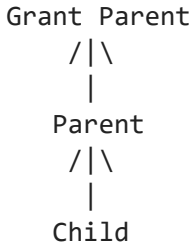
```
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__
ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le_
_', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr_
_', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'get_name',
'set_name']
```

In [8]:
```
print(dir(B))
```

```
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__
ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le_
_', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr_
_', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'get_name',
'set_name']
```

2. Multilevel Inheritance

Grant Parent to Parent and Parent to Child



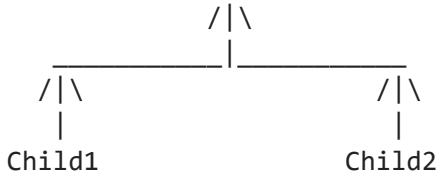
In [10]:
```
class A:
    def hello(self):
        print("Hello World")

class B(A):
    def hi(self):
        print("Hi World")

class C(B):
    def bye(self):
        print("Bye World")


c = C()
c.hello()
c.hi()
c.bye()
```

```
Hello World
Hi World
Bye World
```

3. Heirarchical Inheritance

Single Parent Class Multiple Child Classes

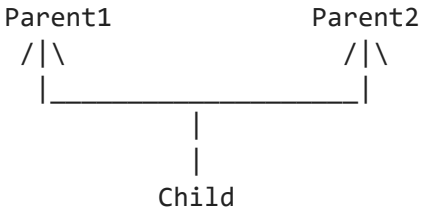
Parent



In [11]:
```
class A:
    def bike(self):
        print("Parent's bike")
    def car(self):
        print("Parent's Car")

class B(A):
    pass

class C(A):
    pass

# B, C are siblings
x = B()
y = C()
x.bike()
y.bike()
x.car()
y.bike()
```

```
Parent's bike
Parent's bike
Parent's Car
Parent's bike
```

4. Multiple Inheritance



MRO - Method Resolution Order

First Look for Method in Child Class itself, if method is not there than

we will look for method into parents left to right (at time inheritance)

Bottom -> Left to Right -> Top

Program - 1

In [12]:
```
class Papa:
    def pocket_money(self):
        print("Papa is the source of pocket money.")
    def change_channel(self):
        print("Put Tv on HotStar to watch cricket match b/w india vs england")

class Mummy:
    def love(self):
        print("Lot's and Lot's of love From Mom")
    def change_channel(self):
        print("Put Tv on star Plus I want to see saas bahu serial")
```

```
class Child(Mummy, Papa):
    pass


a = Child()
a.change_channel()
a.pocket_money()
a.love()
```

```
Put Tv on star Plus I want to see saas bahu serial
Papa is the source of pocket money.
Lot's and Lot's of love From Mom
```

In [13]:

```
print(dir(Child))
```

```
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__
ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le_
_', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr_
_', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'change_cha
nnel', 'love', 'pocket_money']
```

Program - 2

In [27]:

```
class A:
    def __init__(self):
        self.name = "Hello! This is From Class A"
    def hello(self):
        print("I am Boss!!")
    def bye(self):
        print("Bye Bye")

class B:
    def __init__(self):
        self.name = "Hello! This is From class B"
    def hello(self):
        print("I am cool!!")
    def good_bye(self):
        print("Good Bye")

class C(A,B):
    def hi(self):
        print("I am Hot!!")
        super().hello() # MRO
    def chalta_method(self):
        super().bye()
        super().good_bye()


c = C()
c.hello()
c.chalta_method()
c.hi()
c.name
```

```
I am Boss!!
Bye Bye
Good Bye
I am Hot!!
I am Boss!!
```

Out[27]:

```
'Hello! This is From Class A'
```

Program - 3

In [28]:
```
class A:
    def __init__(self):
        self.name = "Hello! This is From Class A"
    def hello(self):
        print("I am Boss!!")
    def bye(self):
        print("Bye Bye")

class B:
    def __init__(self):
        self.name = "Hello! This is From Class B"
    def hello(self):
        print("I am Cool!!")
    def good_bye(self):
        print("Good Bye")

class C(B, A):
    def hi(self):
        print("I am Hot!!")
        super().hello() # MRO
    def chalta_method(self):
        super().bye()
        super().good_bye()

a = C()
a.hello()
a.hi()
a.chalta_method()
a.name
```

Out[28]:
```
I am Cool!!
I am Hot!!
I am Cool!!
Bye Bye
Good Bye
'Hello! This is From Class B'
```

5. Hybrid Inheritance

Combination of Single, Multi level, multiple and Hierarchical inheritance is known as Hybrid Inheritance.

Program - 1

In [24]:
```
class A:
    def hi(self):
        print("Class A")

class B(A):
    pass

class C(A):
    def hi(self): # over-riding
        print("Class C")

class D(B, C):
    pass


a = D() # MRO
a.hi()
print(D.mro())
```

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

Program - 2

In [23]:

```
class A:
    def hi(self):
        print("Class A")

class B(A):
    pass

class C:
    def hi(self):
        print("Class C")

class D(B, C):
    pass


a = D()  # MRO
a.hi()
print(D.mro())
```

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

3. Method Resolution Order(MRO)

>In Hybrid Inheritance the method resolution order is decided based on MRO algorithm.
>This algorithm is also known as C3 algorithm.
>Samuele Pedroni proposed this algorithm.
>It follows DLR (Depth First Left to Right)i.e Child will get more priority than Parent. Left Parent will get more priority than Right Parent

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

Head Element vs Tail Terminology

Assume C1,C2,C3,...are classes.
In the list : C1C2C3C4C5....
C1 is considered as Head Element and remaining is considered as Tail.

How to find Merge

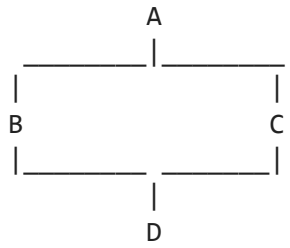
1. Take the head of first list
2. If the head is not in the tail part of any other list,then add this head to the result and remove it from the lists in the merge.
3. If the head is present in the tail part of any other list,then consider the head element of the next list and continue the same process.

Note:

We can find MRO of any class by using mro() function.

print(ClassName.mro())

Problem - 1



In [18]:
```
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 '__main__.A'>, <
class 'object'>]
```

Problem - 2

In [19]:
```
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

print(A.mro())#AO
print(X.mro())#XABO
print(Y.mro())#YBCO
print(P.mro())#PXAYBCO
```

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

```
mro(p)= P+Merge(mro(X),mro(Y),mro(C),XYC)
      = P+Merge(XABO,YBCO,CO,XYC)
      = P+X+Merge(ABO,YBCO,CO,YC)
      = P+X+A+Merge(BO,YBCO,CO,YC)
      = P+X+A+Y+Merge(BO,BCO,CO,C)
      = P+X+A+Y+B+Merge(O,CO,CO,C)
      = P+X+A+Y+B+C+Merge(O,O,O)
      = P+X+A+Y+B+C+O
```

In [20]:
```
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
```

4. Super() Method

super() is a built-in method which is useful to call the super class constructors,variables and methods from the child class.

Case 1:

From child class constructor and instance method, we can access parent class instance method,static method and class method by using super()

In [37]:

```
class P:
    a=10
    def __init__(self):
        self.b=10
    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):
    a=888
    def __init__(self):
        self.b=999
        super().__init__()
        print(super().a)
        super().m1()
        super().m2()
        super().m3()
    def m1(self):
        print(super().a)
        super().m1()
        super().m2()
        super().m3()
```

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

```
10
Parent instance method
Parent class method
Parent static method
10
Parent instance method
Parent class method
Parent static method
```

call method of a particular Super class

In [46]:

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

e=E()
e.m1()
```

```
A class Method
```

Case 2:

From child class we are not allowed to access parent class instance variables by using super(),Compulsory we should use self only.But we can access parent class static variables by using super().

In [47]:

```
class P:
    a=10
    def __init__(self):
        self.b=20

class C(P):
    def m1(self):
        print(super().a) #valid
        print(self.b) #valid
        print(super().b) #invalid

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

```
10
20
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
C:\Users\PANKAJ~1\AppData\Local\Temp/ipykernel_2408/3652837499.py in <module>
     11
     12 c=C()
```

```
---> 13 c.m1()

C:\Users\PANKAJ~1\AppData\Local\Temp/ipykernel_2408/3652837499.py in m1(self)
      8         print(super().a) #valid
      9         print(self.b) #valid
---> 10         print(super().b) #invalid
     11
     12 c=C()

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

Case 3:

From child class, class method we cannot access parent class instance methods and constructors by using super() directly(but indirectly possible). But we can access parent class static and class methods.

In [48]:
```
class P:
    def __init__(self):
        print('Parent Constructor')
    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):
    @classmethod
    def m1(cls):
        #super().__init__()  #invalid
        #super().m1()    #invalid
        super().m2()
        super().m3()

C.m1()
```

```
Parent class method
Parent static method
```

We can use indirectly as follows

In [49]:
```
class A:
    def __init__(self):
        print('Parent Constructor')
    def m1(self):
        print('Parent instance method')

class B(A):
    @classmethod
    def m1(cls):
        super(B,cls).__init__(cls)
        super(B,cls).m1(cls)

B.m1()
```

```
Parent Constructor
Parent instance method
```

Case 4:

In child class static method we are not allowed to use super() generally (But in special way we can use)

In [50]:
```
class P:
    def __init__(self):
        print('Parent Constructor')
    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):
    @staticmethod
    def m1():
        super().m1() #invalid
        super().m2() #invalid
        super().m3() #invalid

C.m1()
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
C:\Users\PANKAJ~1\AppData\Local\Temp/ipykernel_2408/2242028789.py in <module>
     18         super().m3() #invalid
     19
---> 20 C.m1()

C:\Users\PANKAJ~1\AppData\Local\Temp/ipykernel_2408/2242028789.py in m1()
     14     @staticmethod
     15     def m1():
---> 16         super().m1() #invalid
     17         super().m2() #invalid
     18         super().m3() #invalid

RuntimeError: super(): no arguments
```

We can call parent class static method from child class static method as follows

In [51]:
```
class A:
    @staticmethod
    def m1():
        print('Parent static method')

class B(A):
    @staticmethod
    def m2():
        super(B,B).m1()

B.m2()
```

```
Parent static method
```

In [ ]: