  

**1**

**DURGASOFT, # 202, 2 Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,**

**nd**

**OOPs Part - 2**

# Agenda

### Inheritance

* **Has-A Relationship**
* **IS-A Relationship**
* **IS-A vs HAS-A Relationship**
* **Composition vs Aggregation**
* **Types of Inheritance**
  + **Single Inheritance**
  + **Multi Level Inheritance**
  + **Hierarchical Inheritance**
  + **Multiple Inheritance**
  + **Hybrid Inheritance**
  + **Cyclic Inheritance**

### Method Resolution Order (MRO)

* **super() Method**

  

**2**

**DURGASOFT, # 202, 2 Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,**

**nd**

# Using members of one class inside another 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)**

### 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.

#### Demo Program-1:

|  |
| --- |
| **1) class Engine:** |
| **2) a=10** |
| **3) def init (self):** |
| **4) self.b=20** |
| **5) def m1(self):** |
| **6) print('Engine Specific Functionality')** |
| **7) class Car:** |
| **8) def init (self):** |
| **9) self.engine=Engine()** |
| **10) def m2(self):** |
| **11) print('Car using Engine Class Functionality')** |
| **12) print(self.engine.a)** |
| **13) print(self.engine.b)** |
| **14) self.engine.m1()** |
| **15) c=Car()** |
| **16) c.m2()** |

**Output:**

Car using Engine Class Functionality 10

20

Engine Specific Functionality

#### Demo Program-2:

|  |  |
| --- | --- |
| **1)** | **class Car:** |
| **2)** | **def init (self,name,model,color):** |
| **3)** | **self.name=name** |
| **4)** | **self.model=model** |
| **5)** | **self.color=color** |
| **6)** | **def getinfo(self):** |

  

**3**

**DURGASOFT, # 202, 2 Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,**

**nd**

|  |
| --- |
| **7) print("Car Name:{} , Model:{} and Color:{}".format(self.name,self.model,self.color))** |
| **8)** |
| **9) class Employee:** |
| **10) def init (self,ename,eno,car):** |
| **11) self.ename=ename** |
| **12) self.eno=eno** |
| **13) self.car=car** |
| **14) def empinfo(self):** |
| **15) print("Employee Name:",self.ename)** |
| **16) print("Employee Number:",self.eno)** |
| **17) print("Employee Car Info:")** |
| **18) self.car.getinfo()** |
| **19) c=Car("Innova","2.5V","Grey")** |
| **20) e=Employee('Durga',10000,c)** |
| 21) **e.empinfo()** |

#### Output:

Employee Name: Durga Employee Number: 10000 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-3:

|  |
| --- |
| **1) class X:** |
| **2) a=10** |
| **3) def init (self):** |
| **4) self.b=20** |
| **5) def m1(self):** |
| **6) print("m1 method of X class")** |
| **7) class Y:** |
| **8) c=30** |
| **9) def init (self):** |
| **10) self.d=40** |
| **11) def m2(self):** |
| **12) print("m2 method of Y class")** |
| **13) def m3(self):** |
| **14) x1=X()** |
| **15) print(x1.a)** |
| **16) print(x1.b)** |
| **17) x1.m1()** |
| **18) print(Y.c)** |
| **19) print(self.d)** |
| **20) self.m2()** |
| **21) print("m3 method of Y class")** |
| **22) y1=Y()** |

  

23) **y1.m3()**

#### Output:

10

20

m1 method of X class 30

40

m2 method of Y class m3 method of Y class

### 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.

#### Syntax :

class childclass(parentclass):

#### Demo Program for inheritance:

|  |
| --- |
| **1) class P:** |
| **2) a=10** |
| **3) def init (self):** |
| **4) self.b=10** |
| **5) def m1(self):** |
| **6) print('Parent instance method')** |
| **7) @classmethod** |
| **8) def m2(cls):** |
| **9) print('Parent class method')** |
| **10) @staticmethod** |
| **11) def m3():** |
| **12) print('Parent static method')** |
| **13)** |
| **14) class C(P):** |
| **15) pass** |
| **16)** |
| **17) c=C()** |
| **18) print(c.a)** |
| **19) print(c.b)** |
| **20) c.m1()** |
| **21) c.m2()** |
| 22) **c.m3()** |

**Output:**

10

10

**4**

**DURGASOFT, # 202, 2 Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,**

**nd**

  

**5**

**DURGASOFT, # 202, 2 Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,**

**nd**

Parent instance method Parent class method Parent static method

#### Eg:

|  |
| --- |
| **1) class P:** |
| **2) 10 methods** |
| **3) class C(P):** |
| 4) **5 methods** |

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 contains 15 methods.

#### Note:

What ever members present in Parent class are by default available to the child class through inheritance.

#### Demo Program:

|  |
| --- |
| **1) class P:** |
| **2) def m1(self):** |
| **3) print("Parent class method")** |
| **4) class C(P):** |
| **5) def m2(self):** |
| **6) print("Child class method")** |
| **7)** |
| **8) c=C();** |
| **9) c.m1()** |
| **10) c.m2()** |

**Output:**

Parent class method Child class method

What ever 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

|  |  |
| --- | --- |
| **1)** | **class P:** |
| **2)** | **a=10** |
| **3)** | **def init (self):** |
| **4)** | **self.b=20** |
| **5)** | **class C(P):** |
| **6)** | **c=30** |
| **7)** | **def init (self):** |
| **8)** | **super(). init ()===>Line-1** |

  

**6**

**DURGASOFT, # 202, 2 Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,**

**nd**

|  |
| --- |
| **9) self.d=30** |
| **10)** |
| **11) c1=C()** |
| **12) print(c1.a,c1.b,c1.c,c1.d)** |

If we comment Line-1 then variable b is not available to the child class.

#### Demo program for inheritance:

|  |
| --- |
| **1) class Person:** |
| **2) def init (self,name,age):** |
| **3) self.name=name** |
| **4) self.age=age** |
| **5) def eatndrink(self):** |
| **6) print('Eat Biryani and Drink Beer')** |
| **7)** |
| **8) class Employee(Person):** |
| **9) def init (self,name,age,eno,esal):** |
| **10) super(). init (name,age)** |
| **11) self.eno=eno** |
| **12) self.esal=esal** |
| **13)** |
| **14) def work(self):** |
| **15) print("Coding Python is very easy just like drinking Chilled Beer")** |
| **16) def empinfo(self):** |
| **17) print("Employee Name:",self.name)** |
| **18) print("Employee Age:",self.age)** |
| **19) print("Employee Number:",self.eno)** |
| **20) print("Employee Salary:",self.esal)** |
| **21)** |
| **22) e=Employee('Durga', 48, 100, 10000)** |
| **23) e.eatndrink()** |
| **24) e.work()** |
| **25) e.empinfo()** |

**Output:**

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

  

**7**

**DURGASOFT, # 202, 2 Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,**

**nd**

# 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 Diagram-1

|  |
| --- |
| **1) class Car:** |
| **2) def init (self,name,model,color):** |
| **3) self.name=name** |
| **4) self.model=model** |
| **5) self.color=color** |
| **6) def getinfo(self):** |
| **7) print("\tCar Name:{} \n\t Model:{} \n\t Color:{}".format(self.name,self.model,self.col**  **or))** |
| **8)** |
| **9) class Person:** |
| **10) def init (self,name,age):** |
| **11) self.name=name** |
| **12) self.age=age** |
| **13) def eatndrink(self):** |
| **14) print('Eat Biryani and Drink Beer')** |
| **15)** |
| **16) class Employee(Person):** |
| **17) def init (self,name,age,eno,esal,car):** |
| **18) super(). init (name,age)** |
| **19) self.eno=eno** |
| **20) self.esal=esal** |
| **21) self.car=car** |
| **22) def work(self):** |
| **23) print("Coding Python is very easy just like drinking Chilled Beer")** |
| **24) def empinfo(self):** |
| **25) print("Employee Name:",self.name)** |
| **26) print("Employee Age:",self.age)** |
| **27) print("Employee Number:",self.eno)** |
| **28) print("Employee Salary:",self.esal)** |
| **29) print("Employee Car Info:")** |
| **30) self.car.getinfo()** |
| **31)** |
| **32) c=Car("Innova","2.5V","Grey")** |
| **33) e=Employee('Durga',48,100,10000,c)** |
| **34) e.eatndrink()** |

  

#### Output:

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 Color:Grey

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

# 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.

Diagram-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.

Diagram-3

**8**

**DURGASOFT, # 202, 2 Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,**

**nd**

  

#### Coding Example:

|  |  |
| --- | --- |
| **1)** | **class Student:** |
| **2)** | **collegeName='DURGASOFT'** |
| **3)** | **def init (self,name):** |
| **4)** | **self.name=name** |
| **5)** | **print(Student.collegeName)** |
| **6)** | **s=Student('Durga')** |
| **7)** | **print(s.name)** |

**Output:**

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.

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.

#### Eg:

|  |  |
| --- | --- |
| **1)** | **class P:** |
| **2)** | **def init (self):** |
| **3)** | **print(id(self))** |
| **4)** | **class C(P):** |
| **5)** | **pass** |
| **6)** | **c=C()** |
| **7)** | **print(id(c))** |

**Output:**

6207088

6207088

#### Eg:

**9**

**DURGASOFT, # 202, 2 Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,**

**nd**

|  |  |
| --- | --- |
| **1)** | **class Person:** |
| **2)** | **def init (self,name,age):** |
| **3)** | **self.name=name** |
| **4)** | **self.age=age** |

  

|  |
| --- |
| **5) class Student(Person):** |
| **6) def init (self,name,age,rollno,marks):** |
| **7) super(). init (name,age)** |
| **8) self.rollno=rollno** |
| **9) self.marks=marks** |
| **10) def str (self):** |
| **11) return 'Name={}\nAge={}\nRollno={}\nMarks={}'.format(self.name,self.age,self.rollno**  **,self.marks)** |
| **12) s1=Student('durga',48,101,90)** |
| **13) print(s1)** |

#### Output:

Name=durga Age=48 Rollno=101 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

# Types of Inheritance:

### Single Inheritance:

The concept of inheriting the properties from one class to another class is known as single inheritance.

#### Eg:

|  |  |
| --- | --- |
| **1)** | **class P:** |
| **2)** | **def m1(self):** |
| **3)** | **print("Parent Method")** |
| **4)** | **class C(P):** |
| **5)** | **def m2(self):** |
| **6)** | **print("Child Method")** |
| **7)** | **c=C()** |
| **8)** | **c.m1()** |
| 9) | **c.m2()** |

**Output: Parent Method Child Method**

**10**

**DURGASOFT, # 202, 2 Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,**

**nd**

  

**11**

**DURGASOFT, # 202, 2 Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,**

**nd**

### Multi Level 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

#### Eg:

|  |
| --- |
| **1) class P:** |
| **2) def m1(self):** |
| **3) print("Parent Method")** |
| **4) class C(P):** |
| **5) def m2(self):** |
| **6) print("Child Method")** |
| **7) class CC(C):** |
| **8) def m3(self):** |
| **9) print("Sub Child Method")** |
| **10) c=CC()** |
| **11) c.m1()** |
| **12) c.m2()** |
| **13) c.m3()** |

**Output:**

Parent Method Child Method Sub Child Method

Diagram-5

### Hierarchical Inheritance:

The concept of inheriting properties from one class into multiple classes which are present at same level is known as Hierarchical Inheritance

Diagram-6

|  |
| --- |
| **1) class P:** |
| **2) def m1(self):** |
| **3) print("Parent Method")** |
| **4) class C1(P):** |
| **5) def m2(self):** |
| **6) print("Child1 Method")** |
| **7) class C2(P):** |
| **8) def m3(self):** |
| **9) print("Child2 Method")** |
| **10) c1=C1()** |
| **11) c1.m1()** |
| **12) c1.m2()** |
| **13) c2=C2()** |

  

**12**

**DURGASOFT, # 202, 2 Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,**

**nd**

#### Output:

Parent Method Child1 Method Parent Method Child2 Method

### Multiple Inheritance:

The concept of inheriting the properties from multiple classes into a single class at a time, is known as multiple inheritance.

Diagram-7

#### Output:

Parent1 Method Parent2 Method Child2 Method

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.

class C(P1,P2): ===>P1 method will be considered class C(P2,P1): ===>P2 method will be considered

#### Eg:

  

|  |  |
| --- | --- |
| **6)** | **print("Parent2 Method")** |
| **7)** | **class C(P1,P2):** |
| **8)** | **def m2(self):** |
| **9)** | **print("Child Method")** |
| **10) c=C()** | |
| **11) c.m1()** | |
| **12) c.m2()** | |

#### Output:

Parent1 Method Child Method

### Hybrid Inheritance:

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

Diagram-8

### 5. Cyclic Inheritance:

The concept of inheriting properties from one class to another class in cyclic way, is called Cyclic inheritance.Python won't support for Cyclic Inheritance of course it is really not required.

#### Eg - 1:

class A(A):pass

NameError: name 'A' is not defined Diagram-9

#### Eg - 2:

|  |
| --- |
| **1) class A(B):** |
| **2) pass** |
| **3) class B(A):** |
| 4) **pass** |

NameError: name 'B' is not defined Diagram-10

**13**

**DURGASOFT, # 202, 2 Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,**

**nd**

  

**14**

**DURGASOFT, # 202, 2 Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,**

**nd**

# 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:

Take the head of first list

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.

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())

### Demo Program-1 for Method Resolution Order:

Diagram-11

mro(A)=A,object mro(B)=B,A,object mro(C)=C,A,object mro(D)=D,B,C,A,object

#### test.py:

|  |
| --- |
| **1) class A:pass** |
| **2) class B(A):pass** |
| **3) class C(A):pass** |
| **4) class D(B,C):pass** |
| **5) print(A.mro())** |
| **6) print(B.mro())** |
| **7) print(C.mro())** |
| **8) print(D.mro())** |

  

#### Output:

[<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'>]

### Demo Program-2 for Method Resolution Order:

Diagram-12

mro(A)=A,object mro(B)=B,object mro(C)=C,object mro(X)=X,A,B,object mro(Y)=Y,B,C,object mro(P)=P,X,A,Y,B,C,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)

= 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

#### test.py:

|  |
| --- |
| **1) class A:pass** |
| **2) class B:pass** |
| **3) class C:pass** |
| **4) class X(A,B):pass** |
| **5) class Y(B,C):pass** |
| **6) class P(X,Y,C):pass** |
| **7) print(A.mro())#AO** |
| **8) print(X.mro())#XABO** |
| **9) print(Y.mro())#YBCO** |
| **10) print(P.mro())#PXAYBCO** |

**Output:**

[<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'>]

**15**

**DURGASOFT, # 202, 2 Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,**

**nd**

  

[<class ' main .P'>, <class ' main .X'>, <class ' main .A'>, <class ' main .Y'>, <class ' main .B'>,

<class ' main .C'>, <class 'object'>]

#### test.py:

|  |
| --- |
| **1) class A:** |
| **2) def m1(self):** |
| **3) print('A class Method')** |
| **4) class B:** |
| **5) def m1(self):** |
| **6) print('B class Method')** |
| **7) class C:** |
| **8) def m1(self):** |
| **9) print('C class Method')** |
| **10) class X(A,B):** |
| **11) def m1(self):** |
| **12) print('X class Method')** |
| **13) class Y(B,C):** |
| **14) def m1(self):** |
| **15) print('Y class Method')** |
| **16) class P(X,Y,C):** |
| **17) def m1(self):** |
| **18) print('P class Method')** |
| **19) p=P()** |
| **20) p.m1()** |

**Output:**

P class Method

In the above example P class m1() method will be considered.If P class does not contain m1() method then as per MRO, X class method will be considered. If X class does not contain then A class method will be considered and this process will be continued.

The method resolution in the following order:PXAYBCO

### Demo Program-3 for Method Resolution Order:

Diagram-13

mro(o)=object mro(D)=D,object mro(E)=E,object mro(F)=F,object mro(B)=B,D,E,object mro(C)=C,D,F,object

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

=A+Merge(BDEO,CDFO,BC)

**16**

**DURGASOFT, # 202, 2 Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,**

**nd**

  

**17**

**DURGASOFT, # 202, 2 Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,**

**nd**

=A+B+Merge(DEO,CDFO,C)

=A+B+C+Merge(DEO,DFO)

=A+B+C+D+Merge(EO,FO)

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

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

=A+B+C+D+E+F+O

#### test.py:

|  |
| --- |
| **1) class D:pass** |
| **2) class E:pass** |
| **3) class F:pass** |
| **4) class B(D,E):pass** |
| **5) class C(D,F):pass** |
| **6) class A(B,C):pass** |
| **7) print(D.mro())** |
| **8) print(B.mro())** |
| **9) print(C.mro())** |
| **10) print(A.mro())** |

**Output:**

[<class ' main .D'>, <class 'object'>]

[<class ' main .B'>, <class ' main .D'>, <class ' main .E'>, <class 'object'>] [<class ' main .C'>, <class ' main .D'>, <class ' main .F'>, <class 'object'>]

[<class ' main .A'>, <class ' main .B'>, <class ' main .C'>, <class ' main .D'>, <class ' main .E'>,

<class ' main .F'>, <class 'object'>]

# super() Method:

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

#### Demo Program-1 for super():

|  |
| --- |
| **1) class Person:** |
| **2) def init (self,name,age):** |
| **3) self.name=name** |
| **4) self.age=age** |
| **5) def display(self):** |
| **6) print('Name:',self.name)** |
| **7) print('Age:',self.age)** |
| **8)** |
| **9) class Student(Person):** |
| **10) def init (self,name,age,rollno,marks):** |
| **11) super(). init (name,age)** |
| **12) self.rollno=rollno** |
| **13) self.marks=marks** |

  

|  |  |
| --- | --- |
| **14)** | |
| **15)** | **def display(self):** |
| **16)** | **super().display()** |
| **17)** | **print('Roll No:',self.rollno)** |
| **18)** | **print('Marks:',self.marks)** |
| **19)** | |
| **20) s1=Student('Durga',22,101,90)** | |
| **21) s1.display()** | |

#### Output:

Name: Durga Age: 22

Roll No: 101

Marks: 90

In the above program we are using super() method to call parent class constructor and display() method

#### Demo Program-2 for super():

|  |
| --- |
| **1) class P:** |
| **2) a=10** |
| **3) def init (self):** |
| **4) self.b=10** |
| **5) def m1(self):** |
| **6) print('Parent instance method')** |
| **7) @classmethod** |
| **8) def m2(cls):** |
| **9) print('Parent class method')** |
| **10) @staticmethod** |
| **11) def m3():** |
| **12) print('Parent static method')** |
| **13)** |
| **14) class C(P):** |
| **15) a=888** |
| **16) def init (self):** |
| **17) self.b=999** |
| **18) super(). init ()** |
| **19) print(super().a)** |
| **20) super().m1()** |
| **21) super().m2()** |
| **22) super().m3()** |
| **23)** |
| **24) c=C()** |

**Output:**

10

Parent instance method Parent class method

**18**

**DURGASOFT, # 202, 2 Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,**

**nd**

  

**19**

**DURGASOFT, # 202, 2 Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,**

**nd**

Parent static method

In the above example we are using super() to call various members of Parent class.

**How to call method of a particular Super class:**

We can use the following approaches

#### super(D,self).m1()

It will call m1() method of super class of D.

#### A.m1(self)

It will call A class m1() method

|  |  |
| --- | --- |
| **1)** | **class A:** |
| **2)** | **def m1(self):** |
| **3)** | **print('A class Method')** |
| **4) class B(A):** | |
| **5)** | **def m1(self):** |
| **6)** | **print('B class Method')** |
| **7) class C(B):** | |
| **8)** | **def m1(self):** |
| **9)** | **print('C class Method')** |
| **10) class D(C):** | |
| **11)** | **def m1(self):** |
| **12)** | **print('D class Method')** |
| **13) class E(D):** | |
| **14)** | **def m1(self):** |
| **15)** | **A.m1(self)** |
| **16)** | |
| **17) e=E()** | |
| **18) e.m1()** | |

#### Output:

A class Method

  

**20**

**DURGASOFT, # 202, 2 Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,**

**nd**

## Various Important Points about super():

Case-1: 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().

#### Eg:

|  |
| --- |
| **1) class P:** |
| **2) a=10** |
| **3) def init (self):** |
| **4) self.b=20** |
| **5)** |
| **6) class C(P):** |
| **7) def m1(self):** |
| **8) print(super().a)#valid** |
| **9) print(self.b)#valid** |
| **10) print(super().b)#invalid** |
| **11) c=C()** |
| **12) c.m1()** |

**Output:**

10

20

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

Case-2: From child class constructor and instance method, we can access parent class instance method,static method and class method by using super()

|  |
| --- |
| **1) class P:** |
| **2) def init (self):** |
| **3) print('Parent Constructor')** |
| **4) def m1(self):** |
| **5) print('Parent instance method')** |
| **6) @classmethod** |
| **7) def m2(cls):** |
| **8) print('Parent class method')** |
| **9) @staticmethod** |
| **10) def m3():** |
| **11) print('Parent static method')** |
| **12)** |
| **13) class C(P):** |
| **14) def init (self):** |
| **15) super(). init ()** |
| **16) super().m1()** |
| **17) super().m2()** |
| **18) super().m3()** |
| **19)** |

  

|  |  |
| --- | --- |
| **20)** | **def m1(self):** |
| **21)** | **super(). init ()** |
| **22)** | **super().m1()** |
| **23)** | **super().m2()** |
| **24)** | **super().m3()** |
| **25)** | |
| **26) c=C()** | |
| **27) c.m1()** | |

#### Output:

Parent Constructor Parent instance method Parent class method Parent static method Parent Constructor Parent instance method Parent class method Parent static method

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.

|  |
| --- |
| **1) class P:** |
| **2) def init (self):** |
| **3) print('Parent Constructor')** |
| **4) def m1(self):** |
| **5) print('Parent instance method')** |
| **6) @classmethod** |
| **7) def m2(cls):** |
| **8) print('Parent class method')** |
| **9) @staticmethod** |
| **10) def m3():** |
| **11) print('Parent static method')** |
| **12)** |
| **13) class C(P):** |
| **14) @classmethod** |
| **15) def m1(cls):** |
| **16) #super(). init ()--->invalid** |
| **17) #super().m1()--->invalid** |
| **18) super().m2()** |
| **19) super().m3()** |
| **20)** |
| **21) C.m1()** |

#### Output:

Parent class method Parent static method

**21**

**DURGASOFT, # 202, 2 Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,**

**nd**

  

**From Class Method of Child class,how to call parent class instance methods and constructors:**

|  |
| --- |
| **1) class A:** |
| **2) def init (self):** |
| **3) print('Parent constructor')** |
| **4)** |
| **5) def m1(self):** |
| **6) print('Parent instance method')** |
| **7)** |
| **8) class B(A):** |
| **9) @classmethod** |
| **10) def m2(cls):** |
| **11) super(B,cls). init (cls)** |
| **12) super(B,cls).m1(cls)** |
| **13)** |
| **14) B.m2()** |

#### Output:

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)

|  |
| --- |
| **1) class P:** |
| **2) def init (self):** |
| **3) print('Parent Constructor')** |
| **4) def m1(self):** |
| **5) print('Parent instance method')** |
| **6) @classmethod** |
| **7) def m2(cls):** |
| **8) print('Parent class method')** |
| **9) @staticmethod** |
| **10) def m3():** |
| **11) print('Parent static method')** |
| **12)** |
| **13) class C(P):** |
| **14) @staticmethod** |
| **15) def m1():** |
| **16) super().m1()-->invalid** |
| **17) super().m2()--->invalid** |
| **18) super().m3()--->invalid** |
| **19)** |
| **20) C.m1()** |

RuntimeError: super(): no arguments

**22**

**DURGASOFT, # 202, 2 Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,**

**nd**

  

**23**

**DURGASOFT, # 202, 2 Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,**

**nd**

#### How to call parent class static method from child class static method by using super():

|  |
| --- |
| **1) class A:** |
| **2)** |
| **3) @staticmethod** |
| **4) def m1():** |
| **5) print('Parent static method')** |
| **6)** |
| **7) class B(A):** |
| **8) @staticmethod** |
| **9) def m2():** |
| **10) super(B,B).m1()** |
| **11)** |
| **12) B.m2()** |

**Output:**

Parent static method