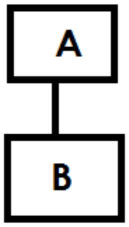
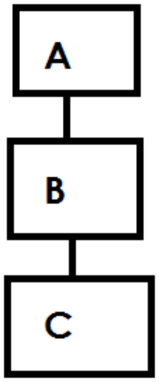
Type of inheritances

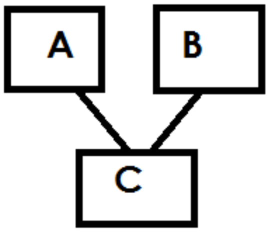
1. Single level inheritance
2. Multi level inheritance
3. Multiple inheritance
4. Hierarchical inheritance
5. Hybrid inheritance



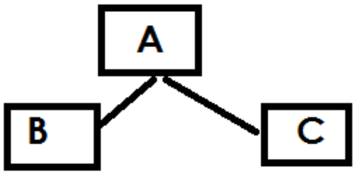
Single level



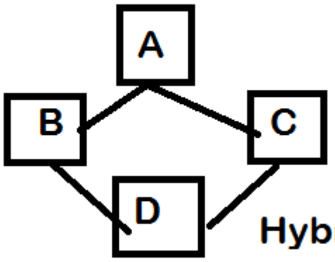
Multilevel



Multiple Inheritance



Hierarchical inheritance



Hybrid Inheritance

Syntax:

```
class <derived-class>/<sub-class>/<child-class>(base-class/superclass/parentclass,….):
        variable
        methods
```

Methods of base class/super class/parent class are inherited automatically within derived class.

Example:

```
# singl level inheritance
class A: # base class/parent class/super class
   def m1(self):
        print("m1 of A class")

class B(A): # derived class/child class/sub class
    def m2(self):
        print("m2 of B class")

def main():
    objb=B()
    objb.m1()
    objb.m2()

main()
```

Output:

```
m1 of A class
m2 of B class
>>>
```

Variables of base class are not inherited automatically within derived class. In order inherit the variables/properties of base class within derived class, the derived class constructor must call the constructor of base class.

Example:

```
class A: # base class/super class
    def __init__(self):
        self.x=100
class B(A): # derived class/sub class
    def __init__(self):
        super().__init__()
        self.y=200
def main():
    objb=B()
    print(objb.y)
    print(objb.x)
main()
```

Output:

```
200
100
>>>
```

Example:

```
# multilevel inheritance

class A:
    def __init__(self):
        self.x=100

class B(A):
    def __init__(self):
        super().__init__()
        self.y=200
class C(B):
    def __init__(self):
        super().__init__()
        self.z=300
def main():
    objc=C()
    print(objc.x,objc.y,objc.z)
main()
```

Output:

```
100 200 300
>>>
```

super() type : return object of immediate super class. Using this object, subclass can invoke the members of super class**.**

Example:

# multiple inheritance

```
class A:
    def __init__(self):
        self.x=100
class B:
    def __init__(self):
        self.y=200
class C(A,B):
    def __init__(self):
        super().__init__()
        B.__init__(self)
        self.z=300
def main():
    objc=C()
    print(objc.x,objc.y,objc.z)
main()
```

Output:

```
100 200 300
>>>
```

Example:

```
# singl level inheritance
class Person:
    def __init__(self):
        self.__name=None
    def set_name(self,n):
        self.__name=n
    def get_name(self):
        return self.__name

class Student(Person):
    def __init__(self):
        super().__init__()
        self.__course=None
    def set_course(self,c):
        self.__course=c
    def get_course(self):
        return self.__course

def main():
    stud1=Student()
    stud1.set_name('naresh')
    stud1.set_course('python')
    print(f'Name: {stud1.get_name()}')
    print(f'Course: {stud1.get_course()}')
main()
```

Output:

```
Name: naresh
Course: python
```

```
>>>
```

Private members of base class/super class are not accessible within derived class. In order to access super class/base class should provide public methods.

Protected members of super class are inherited within subclass.
Public members are used within class, derived class and outside the class.

Example:

```
class A:
    def __init__(self):
        self.x=100  # public
        self._y=200 # protected
        self.__z=300 # private

class B(A):
    def __init__(self):
        super().__init__()

def main():
    objb=B()
    print(objb.x)
    print(objb._y)
    print(objb.__z)
main()
```

Output:

```
100
200
Traceback (most recent call last):
  File "C:/Users/user/Desktop/python6pm/py273.py", line 17, in <module>
    main()
  File "C:/Users/user/Desktop/python6pm/py273.py", line 16, in main
    print(objb.__z)
AttributeError: 'B' object has no attribute '__z'
>>>
```

Example:

#multilevel inheritance

```
class Person:
    def __init__(self):
        self.__name=None
    def set_name(self,n):
        self.__name=n
    def get_name(self):
        return self.__name
class Employee(Person):
    def __init__(self):
        super().__init__()
        self.__job=None
    def set_job(self,j):
        self.__job=j
    def get_job(self):
        return self.__job
class SalariedEmployee(Employee):
    def __init__(self):
        super().__init__()
        self.__salary=None
    def set_salary(self,s):
        self.__salary=s
    def get_salary(self):
        return self.__salary
def main():
    emp1=SalariedEmployee()
    emp1.set_name("naresh")
    emp1.set_job("Manager")
    emp1.set_salary(50000)
    print(emp1.get_name())
    print(emp1.get_job())
    print(emp1.get_salary())
    print(SalariedEmployee.__mro__)
main()
```

Output:

```
naresh
Manager
50000
(<class '__main__.SalariedEmployee'>, <class '__main__.Employee'>,
<class '__main__.Person'>, <class 'object'>)
```

```
>>>
```

What is MRO?

Method Resolution Order is the order in which base classes are searched for a member during lookup.

class.mro()

This method can be overridden by a metaclass to customize the method resolution order for its instances. It is called at class instantiation, and its result is stored in __mro__.