Hybird Inheritance Concept:

- Hybrid inheritance is a combination of multiple inheritance and multilevel inheritance.
- The class is derived from the two classes as in the multiple inheritance. However, one of the parent classes is not the base class. It is a derived class.
- Hybrid Inheritance combines more than one form of inheritance. It is a blend of more than one type of inheritance.

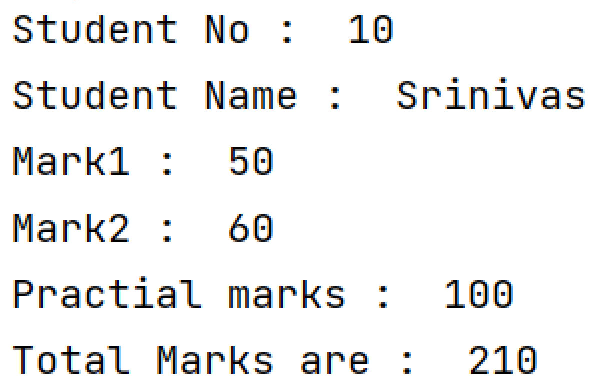
Syntax:

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

Example:

```
class Student:
    def setStudent(self, sno, sname):
        self.sno = sno;
        self.sname = sname;

    def getStudent(self):
        print("Student No : ", self.sno);
        print("Student Name : ", self.sname);

class Marks(Student):
    def setMarks(self, m1, m2):
        self.mark1 = m1;
        self.mark2 = m2;
    def getMarks(self):
        print("Mark1 : ", self.mark1);
        print("Mark2 : ", self.mark2);

class Pratical:
    def setPractial(self, p1):
        self.p1 = p1;

    def getPractial(self):
        print("Practial marks : ", self.p1);

class Result(Marks, Pratical):
    def findTotal(self):
        self.total = self.mark1 + self.mark2 + self.p1;

    def getTotal(self):
        print("Total Marks are : ", self.total);

r = Result();
r.setStudent(10, "Srinivas");
r.setMarks(50, 60);
r.setPractial(100);
r.getStudent()
r.getMarks()
r.getPractial()
r.findTotal()
r.getTotal()
```

Output:



```
Student No :  10
Student Name :  Srinivas
Mark1 :  50
Mark2 :  60
Practial marks :  100
Total Marks are :  210
```

super() method Concept:

The super() builtin returns a proxy object (temporary object of the superclass) that allows us to access methods of the base class.

In Python, the super() function is used to refer to the parent class or superclass. It allows you to call methods defined in the superclass from the subclass, enabling you to extend and customize the functionality inherited from the parent class.

Syntax: *super()*

Return : *Return a proxy object which represents the parent's class.*

Q. What is use of super() method in OOPs concept ?

Example:

```
class Sample:
    def __init__(self):
        print('This is Sample class constructor')

    def message1(self):
        print('Hello Good morning')

class Example(Sample):
    def  __init__(self):
        super().__init__()
        print('This is Example class constructor')

    def message2(self):
        print('Hello Students')
        # accessing  super class
        super(Example, self).message1()
        super().message1()

# s = Sample()
e = Example()
e.message2()
```

Output:

```
This is Sample class constructor
This is Example class constructor
Hello Students
Hello Good morning
Hello Good morning
```

Example:

```
class Animal(object):
  def __init__(self, animal_type):
    print('Animal Type:', animal_type)

class Mammal(Animal):
  def __init__(self):

    # call superclass
    super().__init__('Mammal')

    print('Mammals give birth directly')

dog = Mammal()

# Output:
Animal Type: Mammal
Mammals give birth directly
```

Use of super()

In Python, `super()` has two major use cases:

- Allows us to avoid using the base class name explicitly
- Working with Multiple Inheritance

Example 1: super() with Single Inheritance

In the case of single inheritance, we use `super()` to refer to the base class.

```
class Mammal(object):
  def __init__(self, mammalName):
    print(mammalName, 'is a warm-blooded animal.')

class Dog(Mammal):
  def __init__(self):
    print('Dog has four legs.')
    super().__init__('Dog')

d1 = Dog()
```

Output

```
Dog has four legs.
Dog is a warm-blooded animal.
```

Here, we called the `__init__()` method of the `Mammal` class (from the `Dog` class) using code

```
super().__init__('Dog')
```

instead of

```
Mammal.__init__(self, 'Dog')
```

Since we do not need to specify the name of the base class when we call its members, we can easily change the base class name (if we need to).

```
# changing base class to CanidaeFamily
class Dog(CanidaeFamily):
  def __init__(self):
    print('Dog has four legs.')

    # no need to change this
    super().__init__('Dog')
```

The `super()` builtin returns a proxy object, a substitute object that can call methods of the base class via delegation. This is called indirection (ability to reference base object with `super()`)

Since the indirection is computed at the runtime, we can use different base classes at different times (if we need to).

Example 2: super() with Multiple Inheritance:

```
class Animal:
  def __init__(self, Animal):
    print(Animal, 'is an animal.');

class Mammal(Animal):
  def __init__(self, mammalName):
    print(mammalName, 'is a warm-blooded animal.')
    super().__init__(mammalName)

class NonWingedMammal(Mammal):
  def __init__(self, NonWingedMammal):
    print(NonWingedMammal, "can't fly.")
    super().__init__(NonWingedMammal)

class NonMarineMammal(Mammal):
  def __init__(self, NonMarineMammal):
    print(NonMarineMammal, "can't swim.")
    super().__init__(NonMarineMammal)

class Dog(NonMarineMammal, NonWingedMammal):
  def __init__(self):
    print('Dog has 4 legs.');
    super().__init__('Dog')

d = Dog()
print('')
bat = NonMarineMammal('Bat')
```

Output:

```
Dog has 4 legs.
Dog can't swim.
Dog can't fly.
Dog is a warm-blooded animal.
Dog is an animal.

Bat can't swim.
Bat is a warm-blooded animal.
Bat is an animal.
```

Method Resolution Order (MRO)

Method Resolution Order (MRO) is the order in which methods should be inherited in the presence of multiple inheritance. You can view the MRO by using the `__mro__` attribute.

```
>>> Dog.__mro__

(<class 'Dog'>,

<class 'NonMarineMammal'>,
<class 'NonWingedMammal'>,

<class 'Mammal'>,

<class 'Animal'>,

<class 'object'>)
```

Here is how MRO works:

- A method in the derived calls is always called before the method of the base class. In our example, `Dog` class is called before `NonMarineMammal` or `NoneWingedMammal`. These two classes are called before `Mammal`, which is called before `Animal`, and `Animal` class is called before the `object`.
- If there are multiple parents like `Dog(NonMarineMammal, NonWingedMammal)`, methods of `NonMarineMammal` is invoked first because it appears first.