What is MRO?
MRO stands for Method Resolution Order
Method Resolution Order(MRO) it denotes the way a programming language resolves a method or attribute.
MRO is a concept used in inheritance. It is the order in which a method is searched for in a classes hierarchy.

Example:

```
 class A:
    def m1(self):
        print("m1 of A")
class B(A):
    def m2(self):
        print("m2 of B")
class C(B):
    def m3(self):
        print("m3 of C")

def main():
    print(C.__mro__)
    objc=C()
    objc.m1()
    objc.m2()
    objc.m3()
main()
```

Output:

```
(<class '__main__.C'>, <class '__main__.B'>, <class '__main__.A'>, <class 'object'>)
m1 of A
m2 of B
m3 of C
```

object class

object class is predefined class. Every class in python inherits the properties and behavior of object class.

isinstance(object, classinfo)
Return True if the object argument is an instance of the classinfo argument, or of a subclass thereof. If object is not an object of the given type, the function always returns False.

Object class is base type or root type for building all user defined classes.

These object class methods are used by python virtual machine for managing objects.

Example:

```
class A: # object class
    '''this is class A inherited from object'''
    pass


print(A.__doc__)
```

Output:

```
======== RESTART: F:/python6pmaug/ooptest44.py ========
this is class A inherited from object
```

__str__() method object class
This returns string representation of object. The method is called by python virtual machine whenever object is printed. By default this method returns address of object. In order to return values object, this method must override.

Example:

```
class Student:
    def __init__(self,rno,name):
        self.__rollno=rno
        self.__name=name
    def __str__(self): # overriding method
        return f'{self.__rollno},{self.__name}'
```

```
def main():
    stud1=Student(101,"naresh")
    print(stud1.__str__())
    list1=list(range(10,60,10))
    print(list1.__str__())

main()
```

Output:

```
======== RESTART: F:/python6pmaug/ooptest45.py ========
101,naresh
[10, 20, 30, 40, 50]
```

Example:

```
class Employee:
    def __init__(self,eno,en):
        self.__empno=eno
        self.__ename=en
    def __str__(self): # overriding method
        return f'{self.__empno},{self.__ename}'
    def __hash__(self): # overriding method
        return self.__empno
    def __eq__(self,other): # overriding method
        if self.__empno==other.__empno:
            return True
        else:
            return False

def main():
    empSet=set()
    empSet.add(Employee(101,"naresh"))
    empSet.add(Employee(102,"suresh"))
    empSet.add(Employee(101,"kishore"))
    for emp in empSet:
        print(emp)
main()
```

Output:

```
======== RESTART: F:/python6pmaug/ooptest46.py ========
101,naresh
```

102,suresh

Inner classes or Nested classes
Defining class inside class is called nested class or inner class.

Advantage of nested classes
1. **Providing security**
2. **Reusability and Modularity**

Types of inner classes
1. Member class
2. Local class

If a class defined