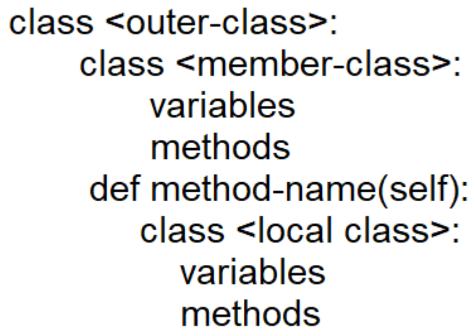
Inner classes or nested classes

A class within class is called inner class or nested class.
Inner classes are two types

1. Member class
2. Local class

If a class is defined as a member of class, it is called as member class.
If a class is defined inside method or block is called local class.



```
class <outer-class>:
        class <member-class>:
                variables
                methods
         def method-name(self):
              class <local class>:
                  variables
                  methods
```

Member class is used anywhere within outer class
Local class is used within declared method.

Applications of inner class

1. Hiding functionality of one class inside another class
2. Dividing functionality of one class into number of classes

Example:

```
class Person:
    class Address: # Member Class
        def __init__(self):
            self.__street=None
            self.__city=None
        def read_address(self):
            self.__street=input("Enter Street ")
            self.__city=input("Enter City ")
        def print_address(self):
            print(f'Street {self.__street}')
            print(f'City {self.__city}')

    def __init__(self):
        self.__name=None
        self.__add1=Person.Address()
        self.__add2=Person.Address()
    def read_person(self):
        self.__name=input("Enter Name ")
        self.__add1.read_address()
        self.__add2.read_address()
    def print_person(self):
        print(f'Name {self.__name}')
        self.__add1.print_address()
        self.__add2.print_address()


p1=Person()
p1.read_person()
p1.print_person()
```

Output:

```
Enter Name naresh
Enter Street ameerpet
Enter City hyd
Enter Street s.r.nager
Enter City hyd
Name naresh
Street ameerpet
City hyd
Street s.r.nager
City hyd
```

Example:

```
class Student:
    class Date: # Member Class
        def __init__(self):
            self.__dd=None
            self.__mm=None
            self.__yy=None
        def readDate(self):
            self.__dd=int(input("Enter dd value"))
            self.__mm=int(input("Enter mm value "))
            self.__yy=int(input("Enter yy value"))
        def printDate(self):
            print(f'{self.__dd}-{self.__mm}-{self.__yy}')

    def __init__(self):
        self.__rollno=None
        self.__dob=Student.Date()
        self.__doj=Student.Date()
    def readStudent(self):
        self.__rollno=int(input("Enter Rollno "))
        print("Enter Date of Birth")
        self.__dob.readDate()
        print("Etner Date of Joining ")
        self.__doj.readDate()
    def printStudent(self):
        print(f'Rollno {self.__rollno}')
        print(f'Date of Birth')
        self.__dob.printDate()
        print(f'Date of Joining')
        self.__doj.printDate()


stud1=Student()
stud1.readStudent()
stud1.printStudent()
```

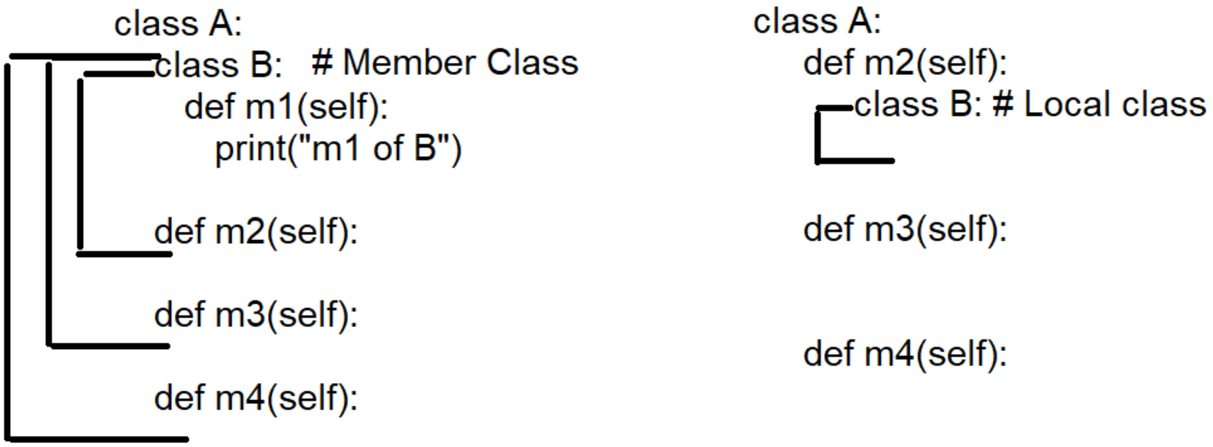
Output:

```
Enter Rollno 1
Enter Date of Birth
Enter dd value10
Enter mm value 3
Enter yy value2000
Etner Date of Joining
Enter dd value10
Enter mm value 3
Enter yy value2023
```

Rollno 1
Date of Birth
10-3-2000
Date of Joining
10-3-2023

Local class

Local class is one type of inner class or nested class.
Local class is defined inside a method. This scope of this class is inside method.



```
class A:
  class B:   # Member Class
    def m1(self):
      print("m1 of B")

  def m2(self):

  def m3(self):

  def m4(self):
```

```
class A:
    def m2(self):
        class B: # Local class

    def m3(self):

    def m4(self):
```

Example:

```
class A:
    def m1(self):
        class B: # Local class
            def m2(self):
                print("m2 of class B")
        objb=B()
        objb.m2()
        print("inside m1 of A")


obja=A()
obja.m1()
```

Output

m2 of class B
inside m1 of A

Overloading

Method overloading is a process of defining more than one method or function with same name and number of arguments or parameters. Python does not support method overloading.

Example:

```
def add(a,b):
    return a+b

def add(a,b,c):
    return a+b+c

def adding(*values):
    s=0
    for value in values:
        s=s+value
    return s


res1=add(10,20,30)
print(res1)
res2=adding(10,20)
res3=adding(10,20,30)
res4=adding(10,20,30,40,50)
print(res2,res3,res4)
```

Output:

```
60
30 60 150
```

Operator Overloading

Python support operator overloading.
Existing operators perform operations on predefined data types but not on user defined data types.

For every operator python provides a method, these methods are inherited from object class.
These operator methods are magic methods.

Operator	**Method**
+	__add__
-	__sub__
*	__mul__
/	__floatdiv__
//	__floordiv__
==	__eq__

Example:

```
class Point:
    def __init__(self):
        self.__x=0
        self.__y=0
    def __add__(self, other):
        p=Point()
        p.__x=self.__x+other.__x
        p.__y=self.__y+other.__y
        return p
    def setX(self,x):
        self.__x=x
    def setY(self,y):
        self.__y=y
    def getX(self):
        return self.__x
    def getY(self):
        return self.__y

p1=Point()
p2=Point()
p1.setX(10)
p1.setY(20)
p2.setX(50)
p2.setY(60)
p3=p1+p2 # p1.__add__(p2)
```

```
print(p1.getX(),p1.getY())
print(p2.getX(),p2.getY())
print(p3.getX(),p3.getY())
```

Output:

```
10 20
50 60
60 80
```