Encapsulation

Thursday, July 24, 2025 7:16 PM

It is the phenomenon of providing security to the data members present inside the class.

It is also known as access specifiers.

Access specifiers tells whether the data can be accessed outside the class or not, about the scope of usability.

Types of Access specifiers:
Access Specifiers are of 3 types:

i. Public Access Specifiers
ii. Protected Access Specifiers
iii. Private Access Specifiers

1. Public Access Specifier:
   - It can be accessed outside the class and can be modified also outside the class.
   - By default, every data member and methods come under public only.

   Syntax:

```
Class Cname:
        Var=value
        Def Mname(self,args):
                S B
        @classmethod
        Def Mname1(cls,args):
                S B
        @staticmethod
        Def mname2(args):
                S B
```

   Eg:

```
class A:
   a=10
   b=20
   def __init__(self,c,d):
      self.c=c
      self.d=d
   @classmethod
   def disp(cls):
      print(cls.a,cls.b)
   @staticmethod
   def msg():
      print('Hii Everyone')
ob=A(30,40)
A.disp()
print(ob.c)
A.msg()
```

2. Protected Access Specifier:
    - It can also be accessed and modified outside the class.
    - Difference is of syntax only from public access specifier.
    - Inside the class, acts as protected class but outside it acts as normal variable only.

    Syntax:

```
Class Cname:
        _Var=value
        Def _Mname(self,args):
                S B
        @classmethod
        Def _Mname1(cls,args):
                S B
        @staticmethod
        Def _mname2(args):
                S B
```

    Eg:

```
class A:
   _a=10
   _b=20
   def __init__(self,_c,_d):
      self._c=_c
      self._d=_d
   @classmethod
   def _disp(cls):
      print(cls._a,cls._b)
   @staticmethod
   def _msg():
      print('Hii Everyone')
ob=A(30,40)
A._disp()
print(ob._c)
A._msg()
```

3. Private Access Specifiers:
    - It provides the security to the data.
    - It can be accessed and modified inside the class only.
    - It restricts uses outside the class.

```
Class Cname:
        __Var=value
        Def __Mname(self,args):
                S B
        @classmethod
        Def __Mname1(cls,args):
                S B
        @staticmethod
        Def __mname2(args):
                S B
```

To access and mpdify the data outside the class, there are 3 ways:

i. Syntax Method
ii. Using Getter and setter function
iii. Property decorator

i. Syntax Method:

- Obj/Cname._Cname__var/Mname(args) --> Access the data
- Obj/Cname._Cname__var=new_value --> Modify the data

Eg:

```
class A:
   __a=10
   __b=20
   def __init__(self,c,d):
      self.__c=c
      self.__d=d
   @classmethod
   def __disp(cls):
      print(cls.__a,cls.__b)
   @staticmethod
   def __msg():
      print('Hii Everyone')
ob=A(30,40)
A._A__disp()
print(ob._A__c)
A._A__msg()
A._A__a=50
print(A._A__a)
```

ii. Using Getter and setter function:
Creating methods to get the value and methods to set the value.
It is mostly used to extract private members of objects.

Syntax:

```
Class Cname:
        Def __init__(self,var):
                Self.__var=var
        Def get_var(self):
                Return self.__var
        Def set_var(self,new):
                Self.__var=new
Obj=Cname(val)
Print(obj.get_var())
Obj.set_var(new_value)
```

Eg:

```
class School:
   __sname='DPS'
```

```
    def __init__(self,mark,name):
        self.__mark=mark
        self.__name=name
    def get_mark(self):
        return self.__mark
    def set_mark(self,new):
        self.__mark=new
st1=School(78)
print(st1.get_mark())
st1.set_mark(92)
print(st1.get_mark())
print(School._School__sname)
```

iii. Using Property decorator:
It is used to access the private member outside the class with general syntax, i.e cname.mname or obj.mname

4 steps to use property decorator:

i. Use @property over the get function.
ii. Both the function name and method name should be changed to the variable name.
iii. Setter function should have @var_name.setter.
iv. Getter and setter function is required.

First we should have fetter and setter function then only we can use property decorator.

Syntax:

```
Class Cname:
        Def __init__(self,var):
                Self.var=var
        @property
        Def var(self):
                Return Self.__var
        @var.setter
        Def var(self,new):
                Self.__var=new
Obj=Cname(values)
Print(obj.var)
Obj.var=new_value
```

Eg:

```
class Comp:
    def __init__(self,sal,phno):
        self.__sal=sal
        self.__phno=phno
    @property
    def sal(self):
        return self.__sal
    @sal.setter
    def sal(self,new):
```

```
        self.__sal=new
    @property
    def phno(self):
        return self.__phno
    @phno.setter
    def phno(self,new):
        self.__phno=new
e1=Comp(25000,8765432109)
print(e1.sal)
print(e1.phno)
e1.sal=45000
print(e1.sal)
e1.phno=9876543210
print(e1.phno)
```