Method Over-riding

In [14]:
```
class Employee:

    def setNoOfWorkHrs(self):
        self.noOfWorkHrs = 40

    def displayNoOfhrs(self):
        print(self.noOfWorkHrs)

class Trainee(Employee):

    def setNoOfWorkHrs(self):
        self.noOfWorkHrs = 45

    def resetNoOfWorkingHrs(self):
        super().setNoOfWorkHrs()
```

In [15]:
```
emp = Employee()
```

In [16]:
```
emp.setNoOfWorkHrs()
emp.displayNoOfhrs()
```

```
40
```

In [17]:
```
trainee = Trainee()
trainee.setNoOfWorkHrs()
```

In [18]:
```
trainee.displayNoOfhrs()
```

```
45
```

In [19]:
```
#Using super(). ->You can access any of your methods of your base class
```

In [20]:
```
trainee.resetNoOfWorkingHrs()
trainee.displayNoOfhrs()
```

```
40
```

Method Over-riding:

In [23]:
```
class A:

    def meth(self):
        print('This belongs to class A')
    pass

class B(A):
    pass

class C(A):
    pass

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

In [24]:
```
d = D()
d.meth()
```

```
This belongs to class A
```

In [25]:
```
class A:

    def meth(self):
        print('This belongs to class A')
    pass

class B(A):
    def meth(self):
        print('This belongs to class B')

class C(A):
    pass

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

In [26]:
```
d = D()
d.meth()
```

```
This belongs to class B
```

In [27]:
```
class A:

    def meth(self):
        print('This belongs to class A')
    pass

class B(A):
    pass

class C(A):
    def meth(self):
        print('This belongs to class C')

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

In [28]:
```
d = D()
d.meth()
```

```
This belongs to class C
```

In [29]:
```
class A:

    def meth(self):
        print('This belongs to class A')
    pass

class B(A):
    def meth(self):
        print('This belongs to class B')

class C(A):
    def meth(self):
        print('This belongs to class C')
```

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

In [30]:
```
d = D()
d.meth()
```

```
This belongs to class B
```

Operator Overloading:

In [47]:
```
class Square:

    def __init__(self,side):
        self.side = side

    def __add__(sq1,sq2):
        return (4*sq1.side) + (4*sq2.side)
```

In [48]:
```
squareOne = Square(5)
squareTwo = Square(10)
```

In [49]:
```
print('Sum of sides of both squares = ', squareOne+squareTwo)
#TypeError: unsupported operand type(s) for +: 'Square' and 'Square'
#Special methods begin and end with 2 underscores
```

```
Sum of sides of both squares =  60
```

Abstract Base Class(ABC):

ABC is ca class which does not have a definition on its own. It has abstarct methods which forces the implementation in its derived classes

In [51]:
```
#ABCMeta is the class which has the properties of a ABC
#@abstarctmethod decorator is used  for making an abstarct method
#@abstractmethod and ABCMeta both belong the ABC module of python
#ABC class wont allow to instantiate objects for that class, ABC can only be inherited
```

In [52]:
```
from abc import ABCMeta, abstractmethod

class Shape(metaclass = ABCMeta):

    @abstractmethod
    def area(self):
        return 0

class Square(Shape):

    side = 4

    def area(self):
        print('Area is :', (self.side*self.side))
```

In [53]:
```
square = Square()
```

In [54]:
```
square.area()
```

```
Area is : 16
```

In [55]:
```
class Square(Shape):

    side = 4

    def area(self):
        print('Area is :', (self.side*self.side))
```

In [56]:
```
sq = Square()
sq.area()
```

```
Area is : 16
```

In [ ]:
```

```