Class and Object

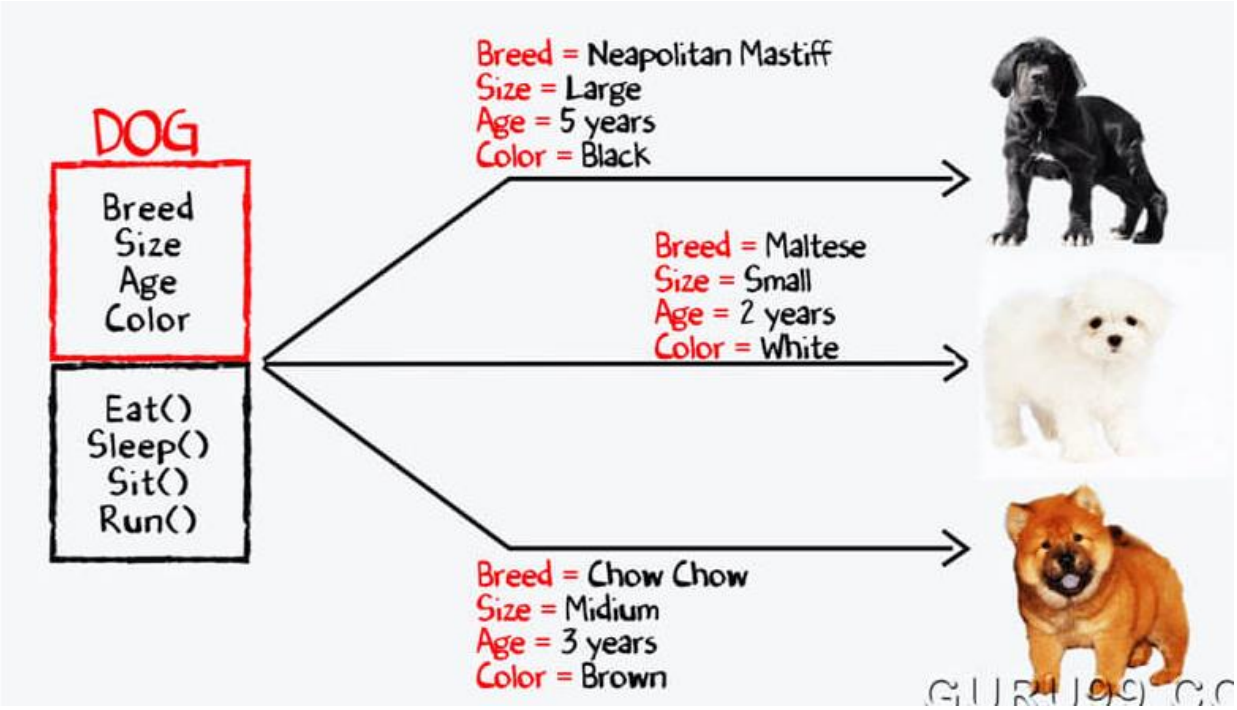
A class is a blueprint of a set of objects, that have a common structure and common behavior. Based on blueprint, actual objects are created.

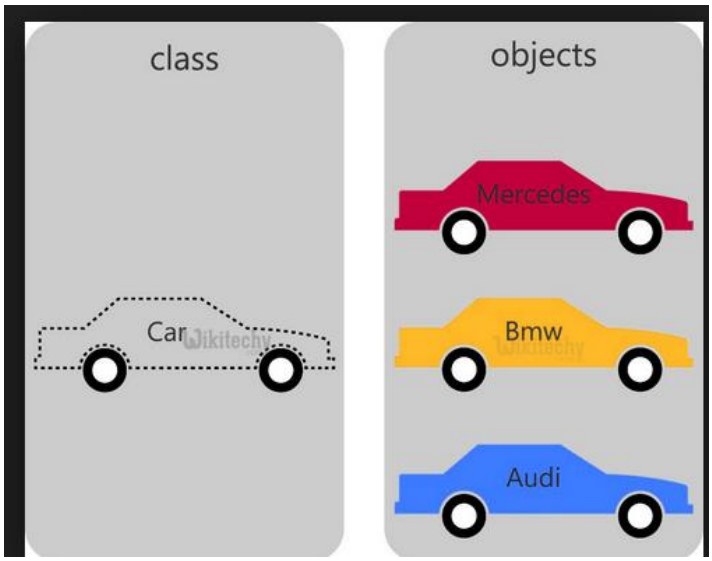
In simple words class define an outline based on which actual objects are created.

for e.g.

class Dog(blueprint)

characteristics/data/properties behavior/method





Properties	Methods
Color	Ignition
Length	Changing
Width	gears
Height	Halting
Seating capacity	Turning
Turning Radius	
Fuel tank capacity	

The class is also termed as object factory because based on class specification multiple objects of same type can be created.
The class is also working as user define data type, which is used to create multiple objects of same type.
While creating a class, the attributes are represented through data items which can hold value to define the objects and behavior is represented through methods or member function.

Object are instance of class
Once a class is defined, multiple instance based on class's specification can be created.
Classes are just conceptual entities but object are real entities.

Syntax to define a class
class <class-name>:
    ['''doc string''']
    data
    method
Consider the following example that creates a class BankAccount.

```
class BankAccount:
    def __init__(self):
        self.balance=0
    def deposite(self, amount):
        self.balance = self.balance + amount
        return self.balance
    def withdraw(self, amount):
        if amount > self.balance:
            return "Insufficient balance"
        self.balance -= amount
        return self.balance
```

- The definition in the class suite begin with keyword def. There are three method define in it.
- Every member function definition must have the first parameter as self.
- The __init__ method is very important method as it is used to create an instance i.e. object and the instance variable.
  self.<instance-variable> = initial value
- The __init__ method is a special method sometimes also known as constructor method as it is used to construct the objects of the class.

- If you have not written an __init__ method in class, Python will provide a default one.

Creating an instance

An object is created or instantiated by using the definition of a class. The syntax to create an object is as given below:

<object-name> = <class-name>()

To create object of class BankAccount define above you may write:

ac1213 = BankAccount()

When the above statement is executed, Python will automatically invoke the __init__() method that you created in the class.

If you have not written an __init__() method in a class, Python will provide a default one.

Now this object can be use to access instance variable and instance method of the class using following syntax

<object-name>.<instance-variable-name>

<object-name>.<methodName>(parameter values for parameters other than self)

for e.g.

ac1213.balance = 5000

ac1213.deposite(3000)

Creating Object by Passing Values

Variable in a class

There can be two types of variables in a class. These are:

i) Instance variable

ii) Class variable

1. Instance variable

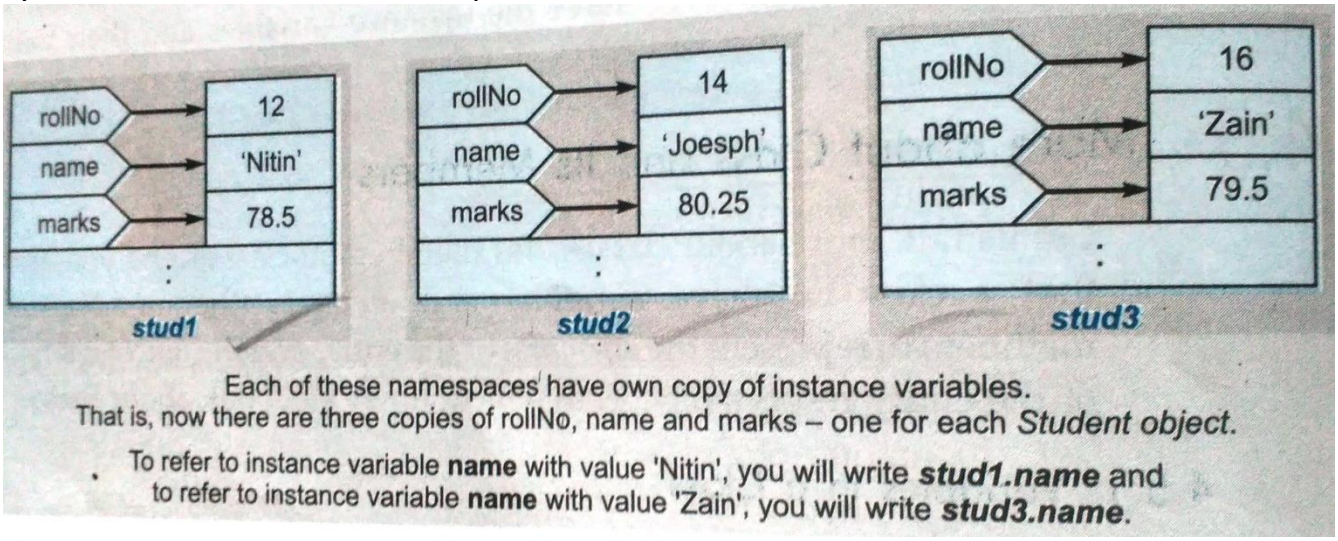
The instance variable are the ones that are created for each object. In other words, every object has individual copy of instance variable.

```
class Student:
  def __init__(self, rno, name, mks)
      self.rollno = rno
      self.name=name
      self.marks=mks
```

Now, when we instantiate this class say, by writing following statements:

```
      stud1 = Student(12, "Nitin", 78.5)
      stud2 = Student(14, "Joseph", 80.25)
      stud3 = Student(16, "Zain", 79.5)
```

Python will create three namespace – one for each student



Each of these namespaces have own copy of instance variables.
That is, now there are three copies of rollNo, name and marks – one for each *Student object*.

To refer to instance variable **name** with value 'Nitin', you will write ***stud1.name*** and to refer to instance variable **name** with value 'Zain', you will write ***stud3.name***.

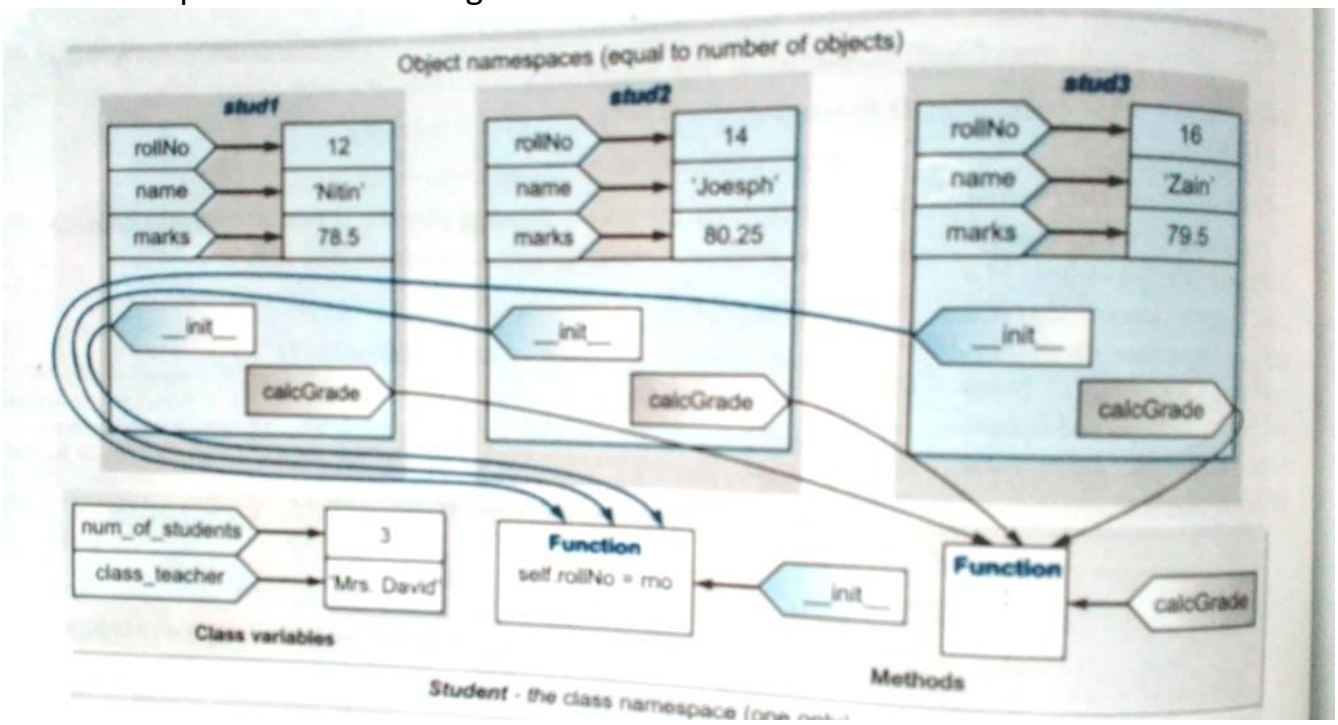
2. Class Variable

The class variables are those variable for which only one copy is created per class and are available to all the class-members. Consider the same class Student, in which if we declare variable(s) outside all methods.

```
class Student:
  num_of_students =0
  class_teacher = "Mr. David"
  def __init__(self, rno, name, mks)
```

```
        self.rollno = rno
        self.name=name
        self.marks=mks
```

The two variables that are declared outside all method definitions will be part of class namespace as shown in fig.



The class namespace contains all the method's definitions and stores the copy of class variables. The contents of class name space are shared among all objects. That is the single copy of class variable is sharable among all objects. Also the objects do not maintain separate copy of methods, rather they refer to the method's definition stored in class namespace.

Although methods reside in class namespace, they are generally called using object names, that is:

- class variables are accessed as ;
  <class-name>.<class_variable_name>
- instance variables are accessed as:
  <objectname>.<instance_variable_name>
- methods are accessed/invoked as:
  <objectName>.<methodName>

Ground rule says when you invoke something using object name, firstly Python will search the object namespace and if it is not found there it searches the class namespace.

Even if you prefix a class variable with an object name, it will translate to class-name because as per scope rules:

i) a variable name used with object name is first searched in object namcspace.
ii) if not found in object namespace, Python will search for it in class namespace.

Methods in a class

There are generally two types of methods in a class:

i) Instance method
ii) Static method

Instance methods

These are the methods that are invoked using instance name or object name. An instance method have the following characteristics :

- The first parameter in the definition of an instance method has to be reference "self" to the instance of the class.
- an instance method can be invoke with object or instance of class; while calling an instance method we need not pass value for parameter self. This parameter is automatically sent to the method at the time of calling .

```
class BankAccount:
        def transfer(self, target, amount):
                :
        def deposite(self, amount)
                :
```

Thhe method call statement for these methods would look like:

```
object.transfer(ac3030, 4500)
object.deposite(5000)
```

Static methods

In Python, a class can also have static methods. A static method is a method which is designed to be called with a class name, i.e. as :

<class-name>.<method-name>()

In other words, static methods are the methods that do not involve a particular instance object, but that somehow involve the class. There are some specific technical details associated with static methods. These are :

- no self parameter is received by a static method.

- The line before method definition must be @staticmethod, which is a decorator, i.e. a special message telling Python that this method is a static method , not an instance method.
- static method can access only class variables, i.e. instance member cannot access inside it.

```
class Student:
        ''' This class store
          detil of Students '''
       no_of_students=0
       classTeacher="Mr. David"
       @staticmethod
       def student_info():
           print(Student.no_of_students, " students have
     been created!!")
          #print("Rollno", self.rollno)
          #error !! instance member cannot access inside
     static method
```

Properties of Python Objects

- Objects are mutable
  for e.g.
  stud1 = Student(101, “ABC”, 65.6)
  stud1.marks = 78.8
- Objects are dynamic
- Object reference will be copied when we assign one reference to another
- Object compare with == operator will be compare object reference rather than actual objects.

Built-In Class Attributes

Every Python class has some built-in attributes which can be accessed using dot operator like any other attribute. Python classes have following predefine attributes:

Attribute	Type	Description
__dict__	Dictionary	This attribute returns a dictionary containing the instance variables of that object <object-name>.__dict__
__name__	String	It return the name of class <class-name>.__name__
__doc__	String or None	This attribute returns the docstring of the class, if defined; otherwise, it returns None.
__bases__	Tuple of classes	This attribute return the base class of the class.
__module__	String	This attribute returns module name in which the class is defined.

Student class based on above concepts

```
class Student:
        ''' This class store
          detil of Students '''
        no_of_students=0
        classTeacher="Mr. David"
        def __init__(self, rollno, sname, marks, grade):
                self.rollno = rollno
                self.sname = sname
                self.marks=marks
                self.grade=grade
                Student.no_of_students +=1
        def showStudent(self):
                print("Rollno :", self.rollno)
                print("Name :", self.sname)
                print("Marks :", self.marks)
                print("Grade :", self.grade)
        @staticmethod
        def student_info():
                print(Student.no_of_students, " students have
been created!!")
                #print("Rollno", self.rollno) error !!
instance member cannot access inside static method
```

```
#============main=============
s1 = Student(12, "Niitn", 68.5, "B")
s2 = Student(14, "Joseph", 80.25, "A")
#print("Student's detail so far ", Student.no_of_students,
" students")
print("Student's detail so far ", Student.student_info(),
" students")
print("Class Teacher Name : ", Student.classTeacher)
s1.showStudent()
print("==================")
s2.showStudent()
print("Built in class Attribute=======")
print(s1.__dict__)
print(s2.__dict__)
print(Student.__doc__)
print("Class Name :", Student.__name__)
print("Module Name:", Student.__module__)
print("Base class :", Student.__base__)
```

Some Standard Methods

When you create classes in Python, you can use some special methods for specific purpose. These are:

- The __init__() method
- The __del__() method
- The __str__() method

The __init__() method

The init method (short for “initializer”) is a special method that gets invoked when an object is instantiated. The structure of an init method is like the one shown below:

class <class-name>:
    :
    def __init__(self, <parameter>):
        <suite>

The __init__ method support all the variant of parameters: positional, default, keyword etc. Usually the method will set various instance variable via self.

The __str__() method

The __str__ method like other special method is defined within a class. This method is internally called when one of the two things happen:

- You try to print the object of the class using print statement, e.g. print(stud1)
- the built-in str() function is applied to an object of that class, e.g. str(stud1)

For instance we have Point class and its object pt1 defined as shown below

```
class Point:
        def __init__(self, x, y):
                self.x  = x
                self.y = y
   :
p1 = Point(2, 3)
```

Now if we give a code like str(p1), Python will automatically invoke __str__() method of class. We can write desire functionality for str() in the __str__() method, it must return a string. If it is not define in the class then it give its default implementation i.e . name of object type and its code.

```
class Point:
        def __init__(self, x, y):
                self.x  = x
                self.y = y
                print("Point object created!!")
p1 = Point(2, 3)
print(p1)
p2 = p1
print(p2)
Point object created!!
<__main__.Point object at 0x03F71190>
<__main__.Point object at 0x03F71190>
```

if we define __str__() method in our class then the output will be according to our __str__() implementation

```
class Point:
        def __init__(self, x, y):
                self.x  = x
                self.y = y
                print("Point object created!!")
   def __str__(self):
                s = "(" + str(self.x) + ", "+ str(self.y) +
")"
                return s
p1 = Point(2, 3)
print(p1)
p2 = p1
```

```
print(p2)
The output will be:
Point object created!!
(2, 3)
(2, 3)
```

The __del__() method

Like constructor, there is no real destructor but Python offers a method __del__() that work like a destructor. It is called when the instance is about to be destroyed. It takes the following form:

```
class <class-name>:
        :
        :
        def __del__(self):
        <suite>
```

This is invoked implicitly i.e. automatically invoked by Python when an object is deleted, which object is to be delete it decide by garbage collector.

Garbage Collector in Python

The garbage collection system (GC) of Python is very significant as it performs following important functions:

i) allocates memory for new objects
ii) identify garbage objects, and
iii) reclaim memory from garbage objects(i.e. destroy them)

The __del__() method gets invoked automatically when an object is garbage collected, not when its del statement is issued.

```
class Point:
        def __init__(self, x, y):
                self.x  = x
                self.y = y
                print("Point object created!!")
        def __str__(self):
                s = "(" + str(self.x) + ", "+ str(self.y) +
")"
                return s
        def __del__(self):
                print("Destructor started!!")
p1 = Point(2, 3)
print(p1)
p2 = p1
print(p2)
```

```
del p1
del p2

Point object created!!
(2, 3)
(2, 3)
Destructor started!!
```

Private Attributes

To mark a data member of method as private you prefix the name with a double underscore. When this is done you can only use it from inside the class. In other words, instance variable names starting with two underscore characters cannot be accessed from outside of the class. Without two underscores in the beginning of their names, the members are public and hence can be access from within the class as well as outside the class.

```
class Point:
        def __init__(self, x, y):
                self.x  = x
                self.y = y
                self.__z=0
                print("Point object created!!")
        def __str__(self):
                s = "(" + str(self.x) + ", "+ str(self.y) +
", "+ str(self.__z)+")"
                return s
p1 = Point(4, 5)
#print(p1.x, p1.y, p1.__z)  error priavte member cannot be
access outside the class
it shows this error
  print(p1.x, p1.y, p1.__z)
AttributeError: 'Point' object has no attribute '__z
```

We can write public method to access private member outside the class.

Similarly, when you put two underscores before a method name, it becomes private and hence can only be called from within the class and not from outside the class:

```
class Point:
        def __init__(self, x, y):
                self.x  = x
                self.y = y
                self.__z=0
                print("Point object created!!")
```

```
        def __printDetail(self):
              s = "(" + str(self.x) + ", "+ str(self.y) + ",
              "+ str(self.__z)+")"
              return s
p1 = Point(4, 5)
p1.__printDetail() # error
p1.__printDetail()
AttributeError: 'Point' object has no attribute
'__printDetail'
```

Reading and Modifying Properties of Classes using attr Methods

Till now access the attributes/properties of classes using dot expression, i.e. as :

```
<instance-name>.<attributeName>
```

Python also provides some high level functions which can be use to modify/read properties. These are listed below:

1. getattr(obj, name[, default])

   This function can be used to access the attribute of object. Internally it invokes a built-in method class __getattr__() automatically.

   ```
   getattr(stud1, 'age')
   ```

   will return value of attribute 'age' of instance stud1.

2. hasattr(obj, name)

   This function can be used to check if an attribute exists or not.

   ```
   hasattr(stud1, 'age')
   ```

   return True if an attribute age exists in the class Student

3. setattr(obj, name, value)

   This function can be used to set an attribute of object. If attribute does not exist, then it would be created. Internally it invokes a built-in class method __setattr__().

   ```
   setattr(stud1, 'age', 18)
   ```

4. delattr(obj, name)

   This function can be used to delete an attribute.

   ```
   detattr(stud1, 'age')
   ```

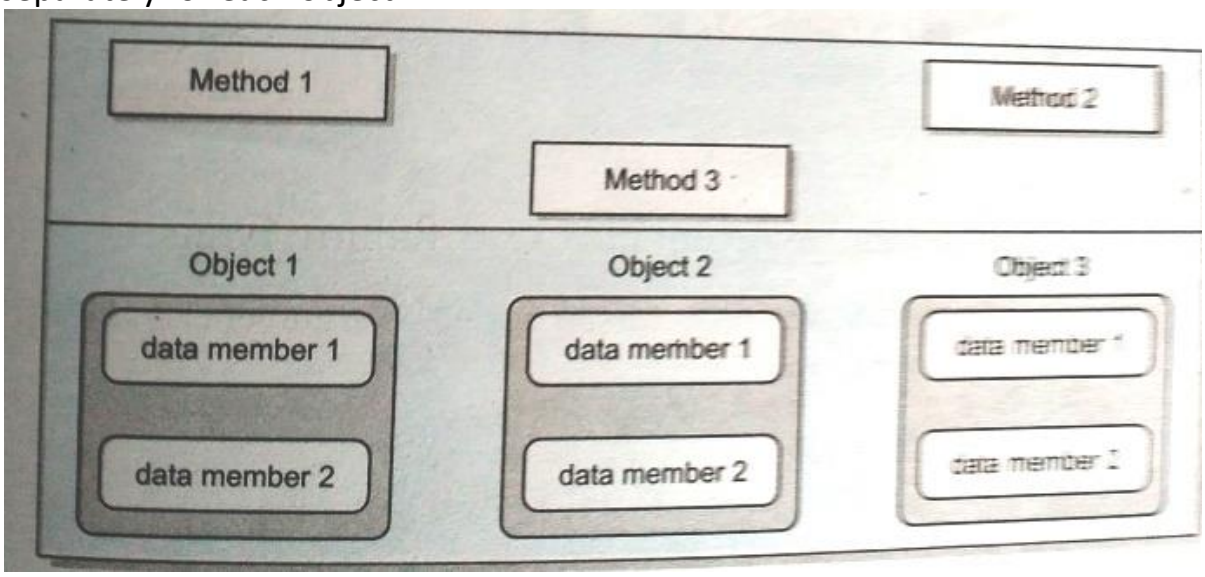
   will delete attribute 'age' from instance stud1.

```
class Employee:
       empCount = 0
       def __init__(self, name, salary):
             self.name = name
             self.salary = salary
             Employee.empCount +=1
       def displayCount(self):
             pass
```

```
        def displayEmployee(self):
                pass
te1 = Employee("Sushma", 20000)
if hasattr(e1, "age"):
        print("Instance e1 has attribute age")
else:
        print("Instance e1 No attribute age")
print("Value of  attribute name : ", getattr(e1, "name"))
setattr(e1, "name", "Arihant")
print("Value of  attribute name : ", getattr(e1, "name"))
delattr(e1, 'name')
if hasattr(e1, "name"):
        print("Instance e1 has attribute name")
else:
        print("Instance e1 No attribute name")
The output is :
Instance e1 No attribute age
Value of  attribute name :  Sushma
Value of  attribute name :  Arihant
Instance e1 No attribute name
```

The "self" Reference

As soon as you define a class, the method are created and placed in the memory space only once, i.e. in the class namespace. That is only one copy of method is maintained that is shared by all the objects of the class. Only space for data members is allocated separately for each object.



This has an associated problem. If only one instance of a method exists, how does it come to know which object's data member is to be manipulated? For example, if method3 is capable of changing the value of datamember2 and we want to change the value of datamember2 of object1. How would the method3 come to know which object's datamember2 is to be changed?
The answer to this problem is 'self', the parameter of every method. The self is a reference to the object that is invoking a method. When a method is called, it is automatically passed an implicit(in built) argument that is **a reference to the object that invoked the method.** This reference is called self. That is if object1 is invoking method3, then an implicit argument is passed to method3 that point to object1 i.e., self now points to object1.

This self reference can be thought of analogous to the ATM card. For instance, in a bank there are many accounts. The account holder can withdraw amount or view their bank-statements through Automatic –Teller-Machines. Now, these ATMs can withdraw from any account in the bank, but which account are they supposed to work upon? This resolved by ATM card, which gives the identification of user and his accounts, from where the amount is withdrawn. Similarly, the self reference is the ATM cards for objects, which identifies the currently-calling object. self is automatically provided and it always refers to currently-calling object.