3.Local variables/Temporary variables (Method Level Variables):
1 If we can declare variables inside a method directly.
2 It will be created at the time of method execution
3 They destroyed once method execution completes.
4 They cannot be accessed from outside of method.

Example:
```
class MyClass():
    def MyMethodOne(self):
        #Local Variable
        x=1
        print(x)
    def MyMethodTwo(self):
        #Local Variable
        y=2
        print(y)
MM=MyClass()
MM.MyMethodOne()
MM.MyMethodTwo()
```

Example:
```
class MyClass():
    def MyMethodOne(self):
        #Local Variable
        x=1
        print(x)
    def MyMethodTwo(self):
        #Local Variable
        y=2
        print(y)
        print(x)
MM=MyClass()
MM.MyMethodOne()
MM.MyMethodTwo()
```

Attributes
An attribute is a specification that defines a property of an object, element, or file. It may also refer to or set the specific value for a given instance of such.

There are two types of Attributes:
Built- in Class Attributes
Attributes defined by Users

Built-In Class Attributes
Every Python class has five built-in attributes and they can be accessed using dot (.) operator.

Attribute	Type	Description
__dict__	dictionary	The class name space.
__name__	string	The name of the class.
__bases__	tuple of classes	The classes from which this class inherits
__doc__	string OR None	The class documentation string.
__module__	string	The name of the module in which this class was defined.

Example:

```
class Built_Attr():
    "Welcome to BuiltInAttrs"
print(Built_Attr.__dict__)
print(Built_Attr.__name__)
print(Built_Attr.__bases__)
print(Built_Attr.__doc__)
print(Built_Attr.__module__)
```

Attribute Defined by Users
1 Attributes are created inside the class definition
2 We can dynamically create new attributes for existing instances of a class
3 Attributes can be bound to class names as well

Example:

```
class Employee():
  empCount=100
print(Employee.empCount)
obj=Employee()
print(obj.empCount)
```

Access Modifiers:
PYTHON supports the following list of Access Modifiers:
1 Public:Attributes can be freely used
2 Protected: (Restricted) attributes should only be used under certain conditions
3 Private:Attributes can only be accessed inside of the class definition

Naming	type	
name	public	These attributes can be used freely inside or outside of a class definition
_name	Protected	These attributes should not be used outside of the class definition, unless inside of a subclass definition
__name	Private	This kind of attributes inaccessible and invisible.

Public:
All member variables and methods are public by default in Python
Example:

```
class Nit():
    def __init__(self):
        self.__pri="I am Private"
        self._pro="I am Protected"
        self.pub="I am Public"
ob=Nit()
print(ob.pub) #accessing Public attribute
```

Example:

```
class Nit():
    def __init__(self):
        self.__pri="I am Private"
        self._pro="I am Protected"
        self.pub="I am Public"
ob=Nit()
```

```
print(ob.pub )
ob.pub=ob.pub + '  Can U add me PUBLIC'
print(ob.pub) #Updaing and accessing Public attribute
```

Protected: (Restricted) attributes should only be used under certain conditions

```
class Nit():
    def __init__(self):
        self.__pri="I am Private"
        self._pro="I am Protected"
        self.pub="I am Public"
ob=Nit()
print(ob._pro) #accessing Protected attribute
```

Private:Attributes can only be accessed inside of the class definition

```
class Nit():
    def __init__(self):
        self.__pri="I am Private"
        self._pro="I am Protected"
        self.pub="I am Public"
ob=Nit()
print(ob.__pri) #accessing Private attribute
```

NOTE: AttributeError: 'Nit' object has no attribute '__pri'

Example:

```
class DataBinding():
    def __init__(self):
        self.x="It is Public Access"
        print(self.x)

DD=DataBinding()
print(DD.x)
```

Example:

```
class DataBinding():
    def __init__(self):
        self._x="It is Protected Access"
        print(self._x)

DD=DataBinding()
print(DD._x)
```

Example:

```
class DataBinding():
    def __init__(self):
        self.__x="It is Private Access"
        print(self.__x)

DD=DataBinding()
print(DD.__x)#AttributeError:
```

Python built-in class functions

Python getattr()

It returns the value of the named attribute of an object. If not found, it returns the default value provided to the function.

Syntax:
```
getattr(object, name[, default])
```

Example:
```
class Person:
    exp = "15+"
    name = "KSRaju"
person = Person()
print('The Exp is:', getattr(person, "exp"))
print('The Exp is:', person.exp)
```

Example:
```
class Car():
    brand='Toyota'
    name='Innova'
    model=2021
print(getattr(Car,'name'))
print(getattr(Car,'brand'))
print(Car.brand)
print(Car.model)
```

Example:
```
class Person:
    exp = "15+"
    name = "KSRaju"
person = Person()
print('The Gender is:', getattr(person, 'gen', 'Male'))
```

Python setattr()
It sets the value of given attribute of an object.

Syntax:
```
setattr(object, name, value)
```

Example:
```
class Person:
    name = 'KSRaju'
p = Person()
print('Before modification:', p.name)
setattr(p, 'name', 'NareshIT')
print('After modification:', p.name)
```

Example:
```
class Person:
    name = 'KSRaju'
p = Person()
setattr(p, 'name', None)
print('Name is:', p.name)
setattr(p, 'exp', 15)
print('Exp is:', p.exp)
```

Python hasattr()
It returns true if an object has the given named attribute and false if it does not.

Syntax:
```
hasattr(object, name)
```

```
Example:
class Person:
    exp = 15
    name = 'KSRaju'
person = Person()
print(hasattr(person, 'exp'))
print(hasattr(person, 'salary'))
```

Python delattr()
It deletes an attribute from the object (if the object allows it).

Syntax:
```
delattr(object, name)
```

```
Example:
class Person:
    exp = 15
    name = 'KSRaju'
person = Person()
print(delattr(person, 'exp'))
```

```
Example:
class Student:
    def __init__(self, name, id, age):
        self.name = name
        self.id = id
        self.age = age
#Creates the object of the class Student
s = Student("John", 101, 22)
# prints the attribute name of the object s
print(getattr(s, 'name'))
# reset the value of attribute age to 23
setattr(s, "age", 23)
# prints the modified value of age
print(getattr(s, 'age'))
# prints true if the student contains the attribute with name id
print(hasattr(s, 'id'))
# deletes the attribute age
delattr(s, 'age')
# this will give an error since the attribute age has been deleted
print(s.age)
```

```
EXAMPLES:
class Rectangle:
   def __init__(self, length, breadth):
       self.length = length
       self.breadth = breadth

   def get_perimeter(self):
       return 2 * (self.length + self.breadth)

   def get_area(self):
       return self.length * self.breadth

   def calculate_cost(self):
       area = self.get_area()
       return area * self.unit_cost
r = Rectangle(160, 120)
print("Area of Rectangle: %s cm^2" % (r.get_area()))
```

Example:

```
class Car(object):
   def __init__(self, model, color, company, speed_limit):
       self.color = color
       self.company = company
       self.speed_limit = speed_limit
       self.model = model

   def start(self):
       print("started")

   def stop(self):
         print("stopped")

   def accelarate(self):
       print("accelarating...")

   def change_gear(self, gear_type):
       print("gear changed")

maruthi_suzuki=Car("ertiga", "black", "suzuki", 60)
maruthi_suzuki.start()
maruthi_suzuki.change_gear(0)
maruthi_suzuki.accelarate()
maruthi_suzuki.stop()
```

Class Variable (Static Variable) & Instance/Non-Static Variables
Class or static variables are shared by all objects, Instance variables are unique to each Instance.

Example:

```
class student:
  clg='NareshIT' #Class  Variable
  def __init__(self,rollno,name):
#Defining an Instance Variable
    self.rollno=rollno
    self.name=name

  def display(self):
    print("Student Name:" ,self.name)
    print("Student RollNumber:" ,self.rollno)
    print("Student College:" ,self.clg)

#Declaring an Instance of  Class
student1=student('100005','Raju')
student1.display()

#Declaring an Instance of  Class
student2=student('100006','DRaju')
student2.display()
```

What is Garbage Collection?
Garbage Collection (GC) is a form of automatic memory management.

OR
The concept of removing the unused,unreferenced object from the memory location is known as a garbage collection.

Example:
```
import gc
print(dir(gc))
```

Example:
```
import gc
#Checking Enable or not
print(gc.isenabled())#True
#To disable GC explicitly
gc.disable()
#To enable GC explicitly
gc.enable()
```

Garbage Collection.
There are two types of garbage collection supported by python they are
1.Automatic Garbage Collection
2.Explicit Garbage Collection

1.Automatic Garbage Collection:-
1 After starting execution of program periodically garbage collector program runs internally
2 Whenever any object is going to be removed from memory location the distructor of that class is going to be executed.
3 In distructor we write the resource deallocation statement.

2.Explicit Garbage Collection
The concept of executing the garbage collection program explicitly whenever we required is known as explicit garbage collection.
-By using 'del' keyword we can run garbage collector explicitly.

Destructors:
Destructor is a special method and the name should be del
Just before destroying an object Garbage Collector always calls destructor to perform clean up activities
Once destructor execution completed then Garbage Collector automatically destroys that object.

Note:
The job of destructor is not to destroy object, it is just to perform clean up activities.

Example
```
class Human:
  def __init__(self,name,age):
    print("Hello InIt")

  def walk(self):
      print("Walking")

  def __del__(self):
    print("I am Destroyed")

a=Human("Raju",44)
```

```
a.walk()
del a

Example:Output of the following Script..!
class Account():
    def __init__(self,Id):
        self.Id=Id
        Id=666
Acc=Account(123)
print(Acc.Id)
```