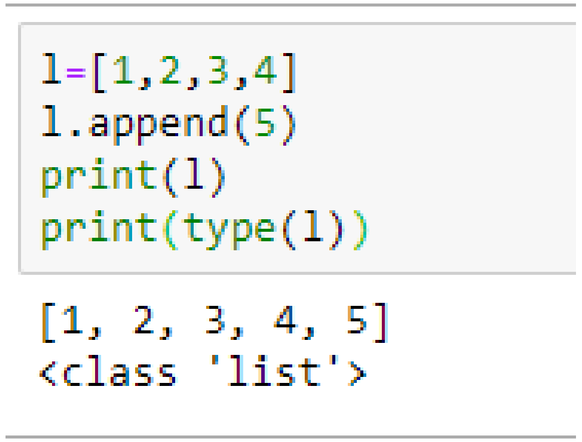
MODULE-2

OBJECT-ORIENTED PROGRAMMING

- Classes and objects
- Creating classes in python
- Creating objects in python
- The constructor method
- Encapsulation
- Inheritance
- Polymorphism

CLASSES AND OBJECTS

- In python, everything is an object
- We can write a class to represent properties (attributes) and actions (behaviour) of object. Properties can be represented by variables, Actions can be represented by Methods.
- Ex:



```
l=[1,2,3,4]
l.append(5)
print(l)
print(type(l))
```

```
[1, 2, 3, 4, 5]
<class 'list'>
```

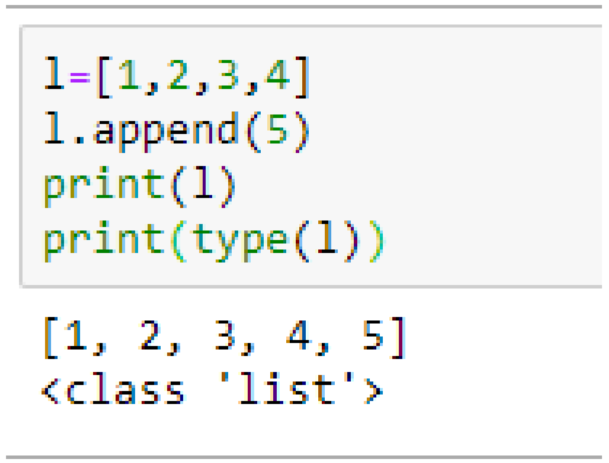
- Class
- Object
- Reference variable

CLASS, OBJECT, REFERENCE VARIABLE

- **Class** is a blueprint to construct objects, it defines all properties and operations of object
- **Object** is physical existence of a class or real world entity. Memory will be allocated for object.
- Per class any number of objects can be created

referencevariable = classname()

- **Reference variable** is pointing to our object, by using it we can perform operations on the object.
- In this example, ‘l’ is reference variable



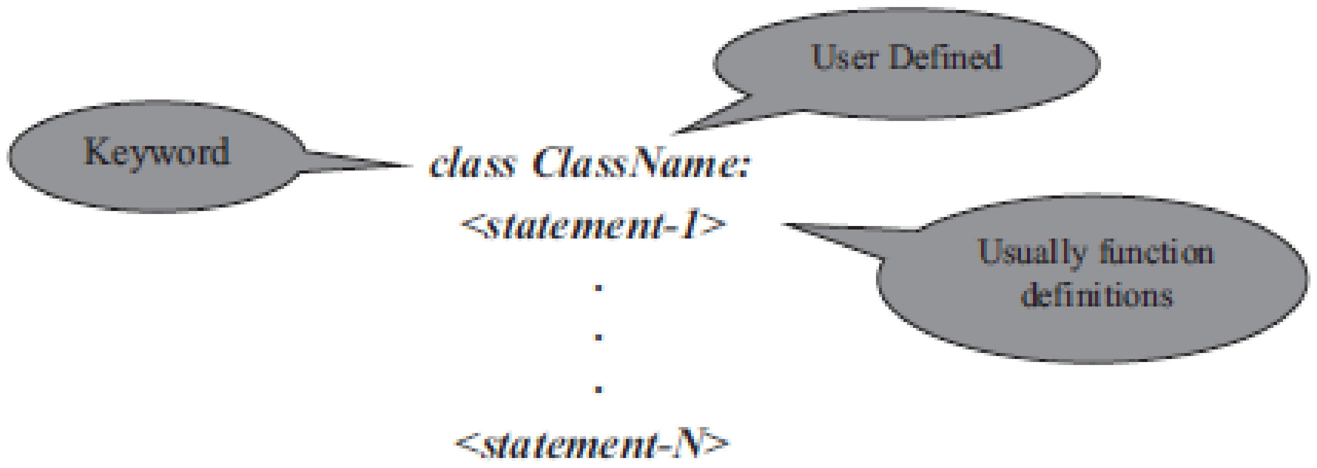
```
l=[1,2,3,4]
l.append(5)
print(l)
print(type(l))
```

```
[1, 2, 3, 4, 5]
<class 'list'>
```

CREATING CLASSES IN PYTHON

- Class contains both variables , methods

```
class className:
    ''' documentation string '''
    variables: instance variables, static and local variables(properties)
    methods: instance methods, static methods, class methods(actions/ behaviour)
```



CLASS

- class classname:

  ‘’’ docstring’’’

  Variables(properties)

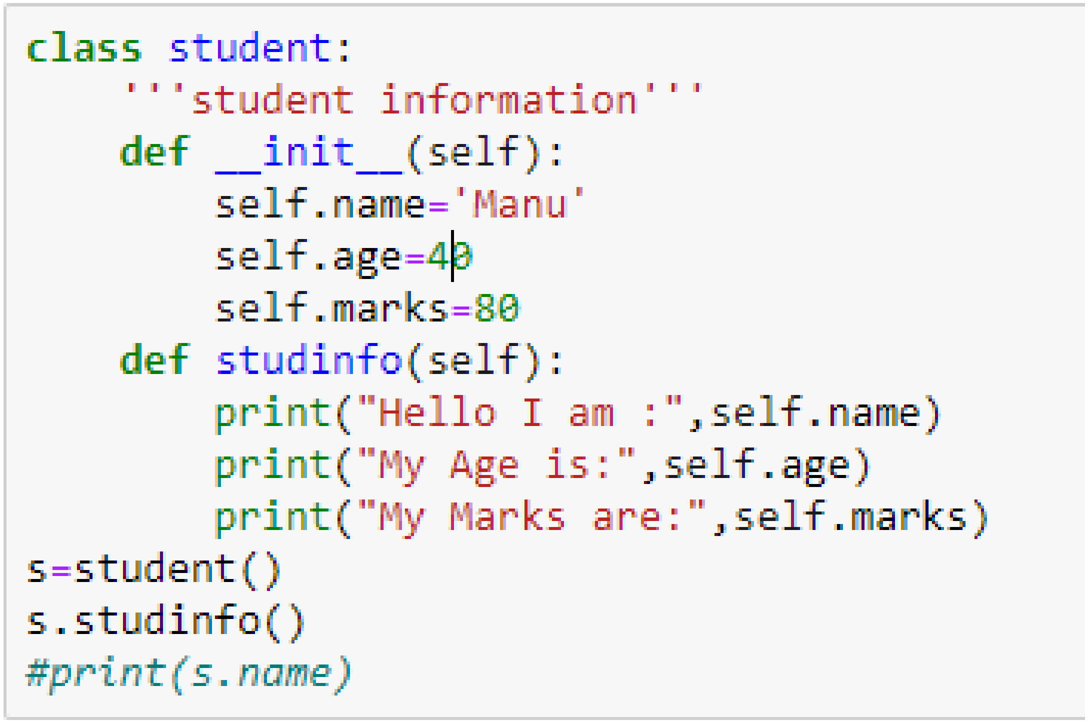
  Methods(actions/behaviour)

- Variables:
  - Instance variables
  - Static variables
  - Local variables
- Methods:
  - Instance methods
  - Static methods
  - Class methods

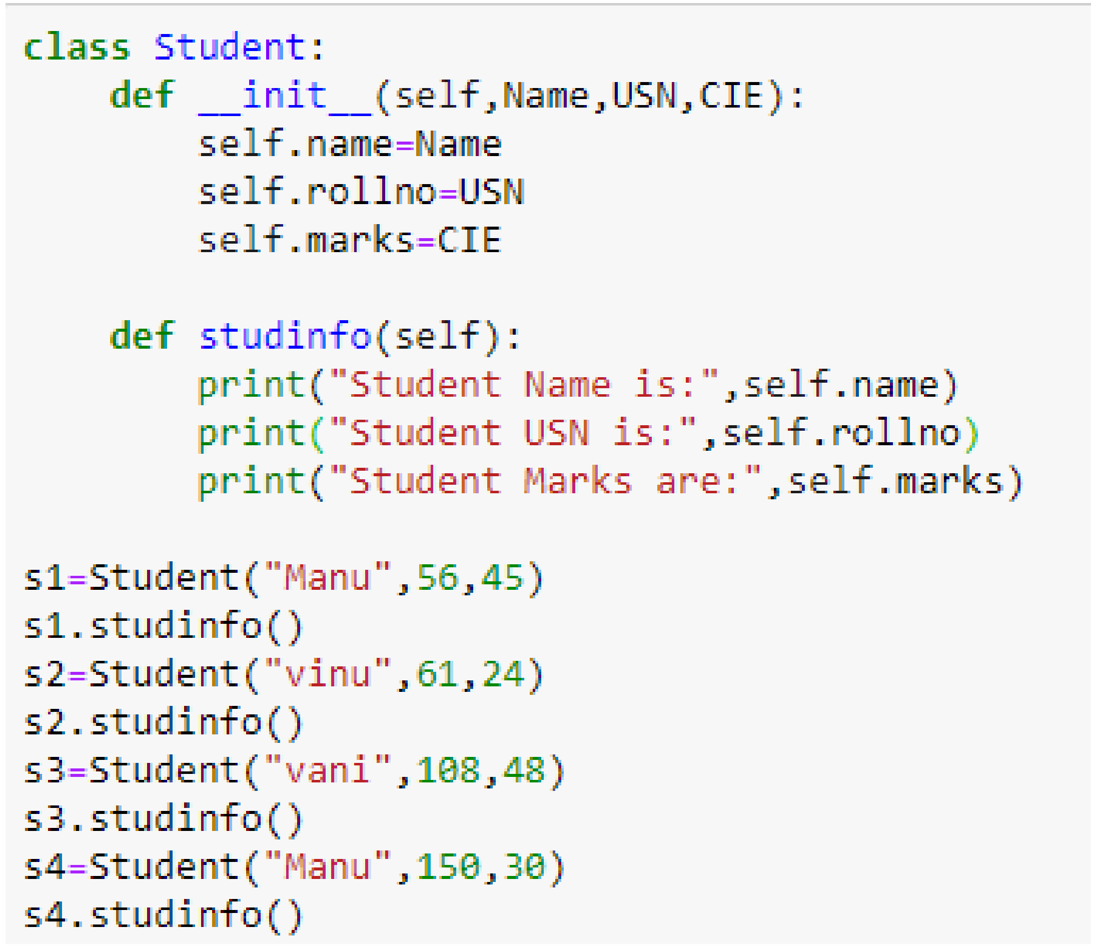
CREATING OBJECTS IN PYTHON

- Object refers to a particular instance of a class where the object contains variables and methods defined in the class.
- Class objects support two kinds of operations:
  - attribute references and
  - instantiation.
- The term attribute refers to any name (variables or methods) following a dot. This is a syntactic construct.
- The act of creating an object from a class is called *instantiation.*
- The names in a class are referenced by objects and are called *attribute references*.
- There are two kinds of attribute references, data attributes and method attributes.
- Variables defined within the methods are called *instance variables* and are used to store data values.
- New instance variables are associated with each of the objects that are created for a class. These instance variables are also called ***data attributes***.
- **Method attributes** are methods inside a class and are referenced by objects of a class. Attribute references use the standard dot notation syntax as supported in Python.

EXAMPLE



```python
class student:
    '''student information'''
    def __init__(self):
        self.name='Manu'
        self.age=40
        self.marks=80
    def studinfo(self):
        print("Hello I am :",self.name)
        print("My Age is:",self.age)
        print("My Marks are:",self.marks)
s=student()
s.studinfo()
#print(s.name)
```



```python
class Student:
    def __init__(self,Name,USN,CIE):
        self.name=Name
        self.rollno=USN
        self.marks=CIE

    def studinfo(self):
        print("Student Name is:",self.name)
        print("Student USN is:",self.rollno)
        print("Student Marks are:",self.marks)

s1=Student("Manu",56,45)
s1.studinfo()
s2=Student("vinu",61,24)
s2.studinfo()
s3=Student("vani",108,48)
s3.studinfo()
s4=Student("Manu",150,30)
s4.studinfo()
```

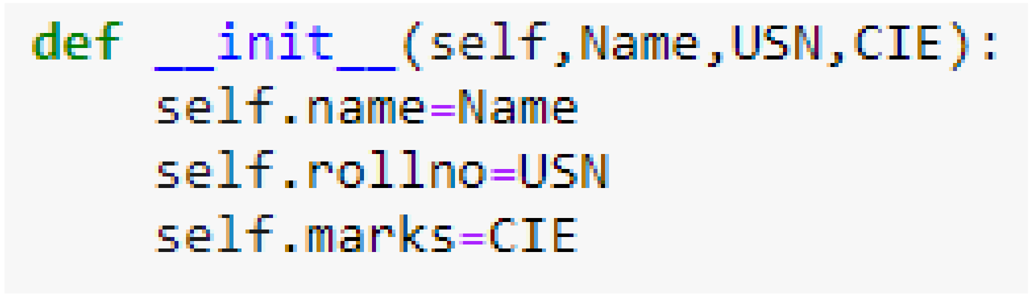
SELF

- Within the class to refer current object, python provides an implicit variable which is ‘self’
- Self variable always pointing to current object
- Self variable is the first argument for constructor and instance methods
- By using self we can declare instance variables, we can access instance variables and instance methods of object
- We can use self within the class only and from outside of class we cannot use
- At the time of calling constructor and instance methods we are not required to provide value for self variable. PVM itself is responsible to provide value
- Instead of ‘self’ we can use any name , but recommended to use self

CONSTRUCTOR CONCEPT

EXAMPLE

- Constructor is a special method in python.
- The name of the constructor should be __init__(self)
- The main purpose of constructor is to declare and initialize instance variables.
- Constructor will be executed automatically at the time of object creation.
- Per object, constructor will be executed only once.
- Constructor can take at least one argument(at least self)
- Constructor is optional and if we are not providing any constructor then python will provide default constructor.



```
def __init__(self,Name,USN,CIE):
    self.name=Name
    self.rollno=USN
    self.marks=CIE
```

- **def __init__(self,name,rollno,marks):**

  self.name=name

  self.rollno=rollno

  self.marks=marks

METHOD VS CONSTRUCTOR

Method	Constructor
Name of method can be any name	Constructor name should be always __init__
Method will be executed if we call that method	Constructor will be executed automatically at the time of object creation
Per object, method can be called any number of times	Per object, Constructor will be executed only once
Inside method we can write logic	Inside Constructor we have to declare and initialize instance variables

TYPES OF VARIABLES

- **Inside Python class 3 types of variables are allowed.**

  1. **Instance Variables (Object Level Variables)**
  2. **Static Variables (Class Level Variables)**
  3. **Local variables (Method Level Variables)**

INSTANCE VARIABLES

- If the value of a variable is varied from object to object, then such type of variables are called instance variables.
- For every object a separate copy of instance variables will be created.
- **Where we can declare Instance variables:**

  1. Inside Constructor by using self variable

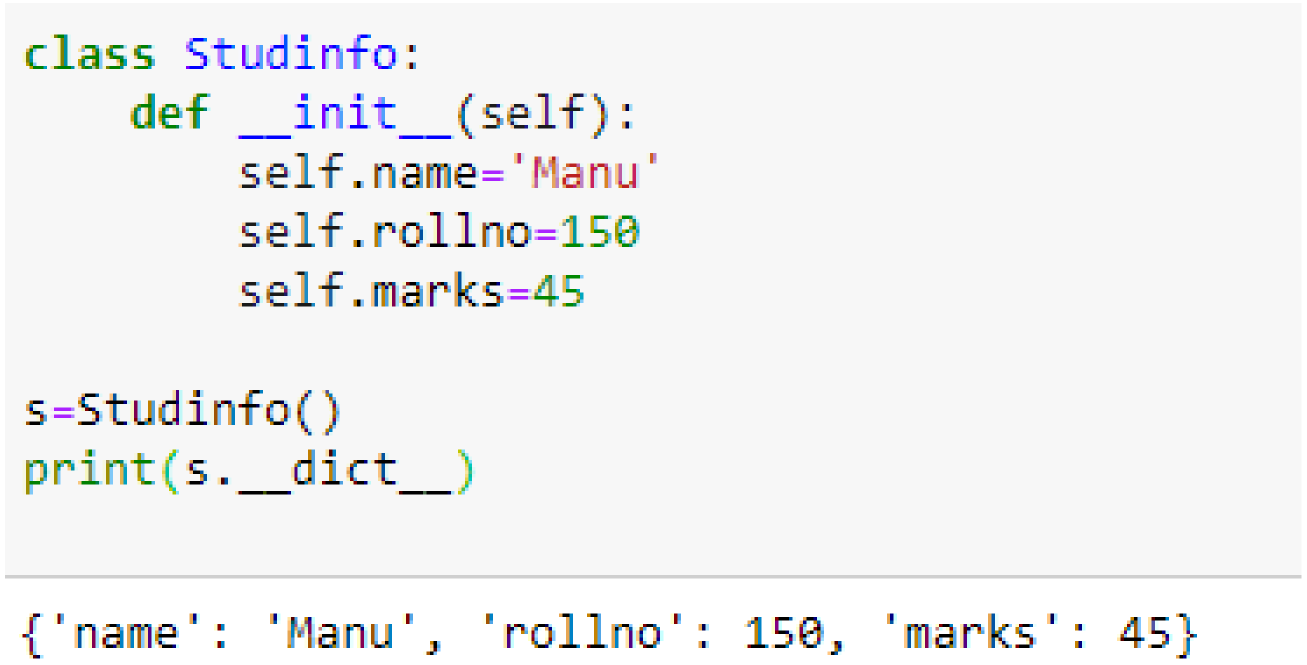
  2. Inside Instance Method by using self variable

  3. Outside of the class by using object reference variable

INSIDE CONSTRUCTOR BY USING SELF VARIABLE:

- We can declare instance variables inside a constructor by **using se**lf keyword.

  Once we creates object, automatically these variables will be added to the object.

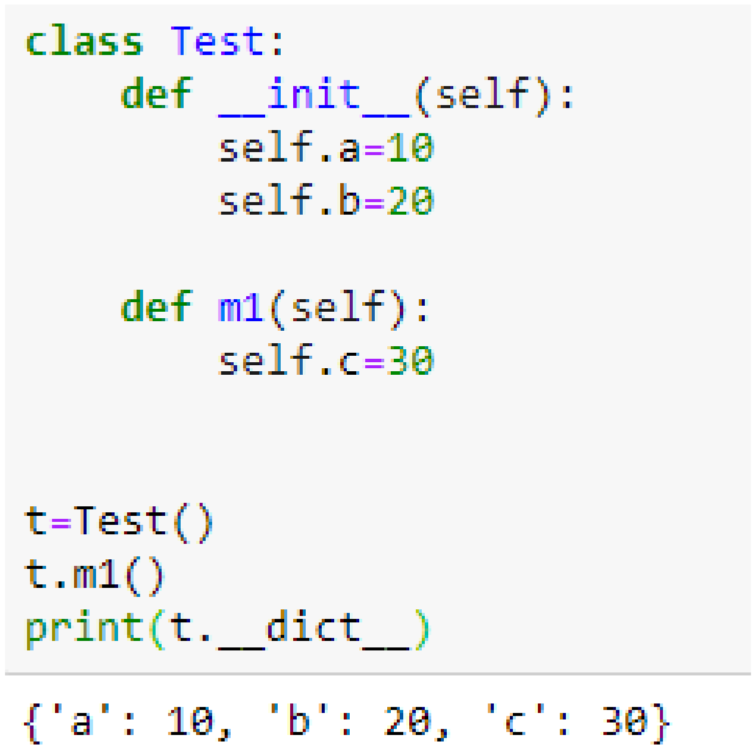


```
class Studinfo:
    def __init__(self):
        self.name='Manu'
        self.rollno=150
        self.marks=45

s=Studinfo()
print(s.__dict__)
```

```
{'name': 'Manu', 'rollno': 150, 'marks': 45}
```

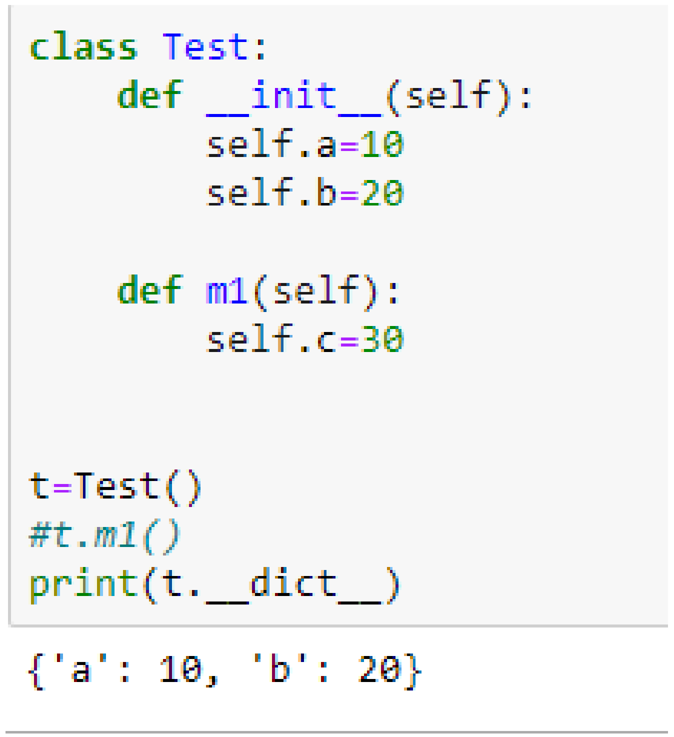
INSIDE INSTANCE METHOD BY USING SELF VARIABLE

- We can also declare instance variables inside instance method by using self variable.
- If any instance variable declared inside instance method, that instance variable will be added once we call that method.



```
class Test:
    def __init__(self):
        self.a=10
        self.b=20

    def m1(self):
        self.c=30


t=Test()
t.m1()
print(t.__dict__)
```

```
{'a': 10, 'b': 20, 'c': 30}
```

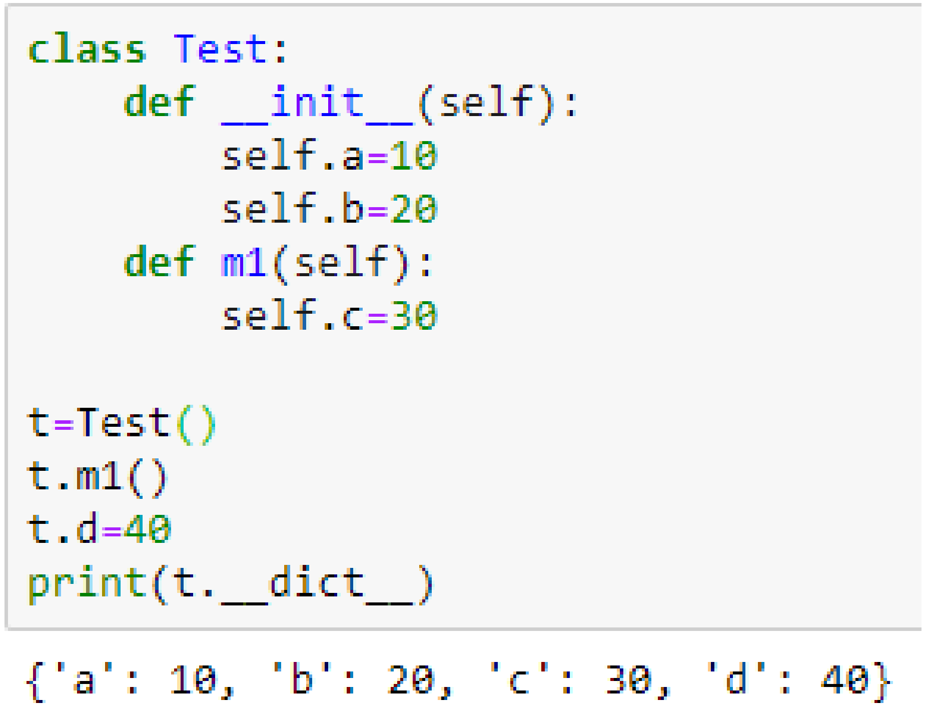


```
class Test:
    def __init__(self):
        self.a=10
        self.b=20

    def m1(self):
        self.c=30


t=Test()
#t.m1()
print(t.__dict__)
```

```
{'a': 10, 'b': 20}
```

OUTSIDE OF THE CLASS BY USING OBJECT REFERENCE VARIABLE:

- We can also add instance variables outside of a class to a particular object.

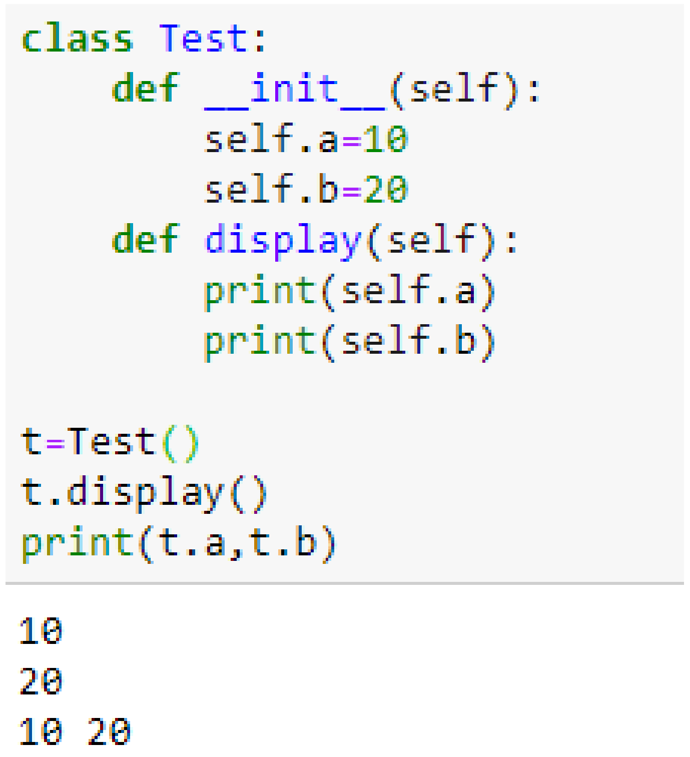


```
class Test:
    def __init__(self):
        self.a=10
        self.b=20
    def m1(self):
        self.c=30

t=Test()
t.m1()
t.d=40
print(t.__dict__)
```

```
{'a': 10, 'b': 20, 'c': 30, 'd': 40}
```

HOW TO ACCESS INSTANCE VARIABLES

- We can access instance variables with in the class by using **self variable** and outside of the class by using object reference.

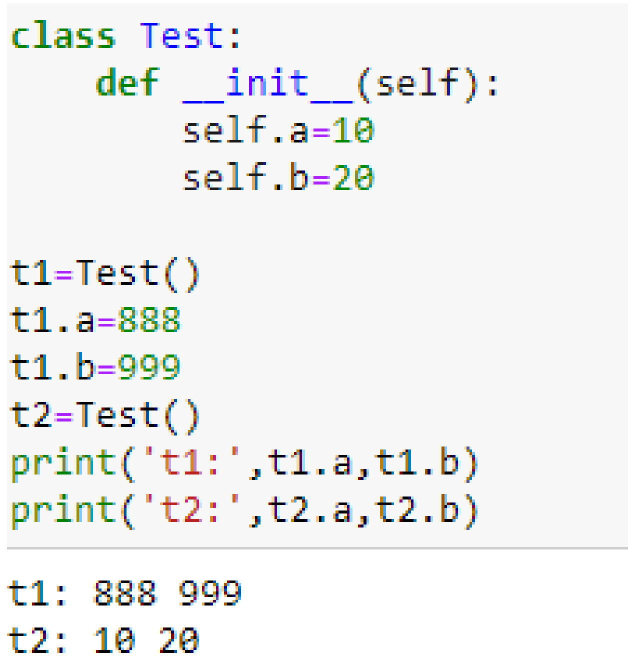


```
class Test:
    def __init__(self):
        self.a=10
        self.b=20
    def display(self):
        print(self.a)
        print(self.b)

t=Test()
t.display()
print(t.a,t.b)
```

```
10
20
10 20
```

EXAMPLE

- If we change the values of instance variables of one object then those changes won't be reflected to the remaining objects, because for every object we are separate copy of instance variables are available.

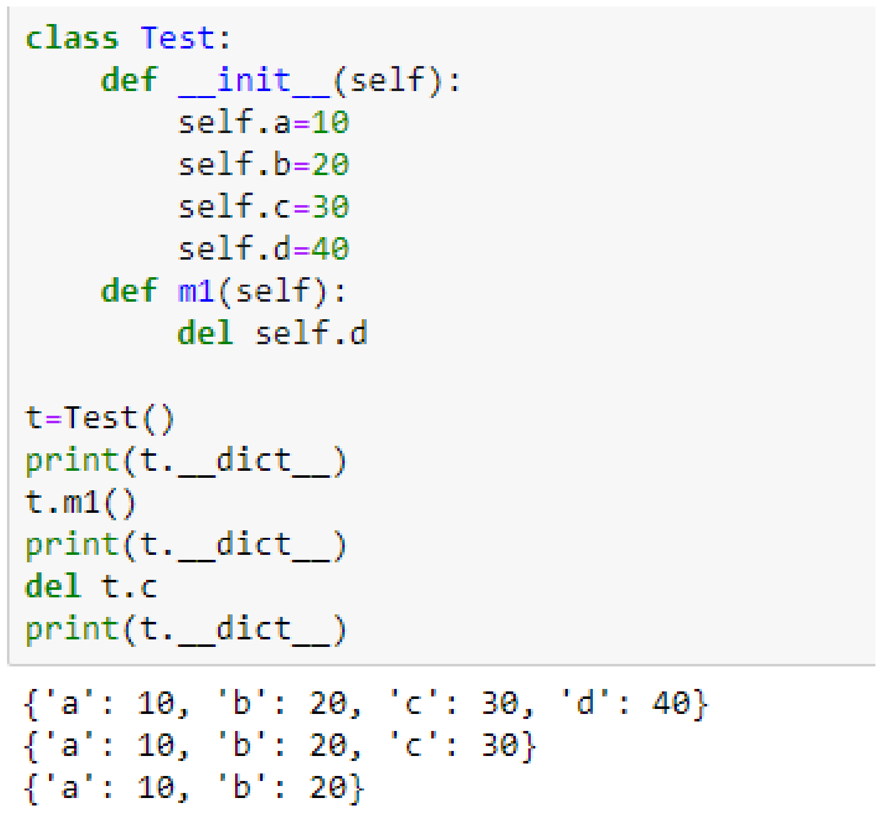


```
class Test:
    def __init__(self):
        self.a=10
        self.b=20

t1=Test()
t1.a=888
t1.b=999
t2=Test()
print('t1:',t1.a,t1.b)
print('t2:',t2.a,t2.b)
```

```
t1: 888 999
t2: 10 20
```

HOW TO DELETE INSTANCE VARIABLE FROM THE OBJECT

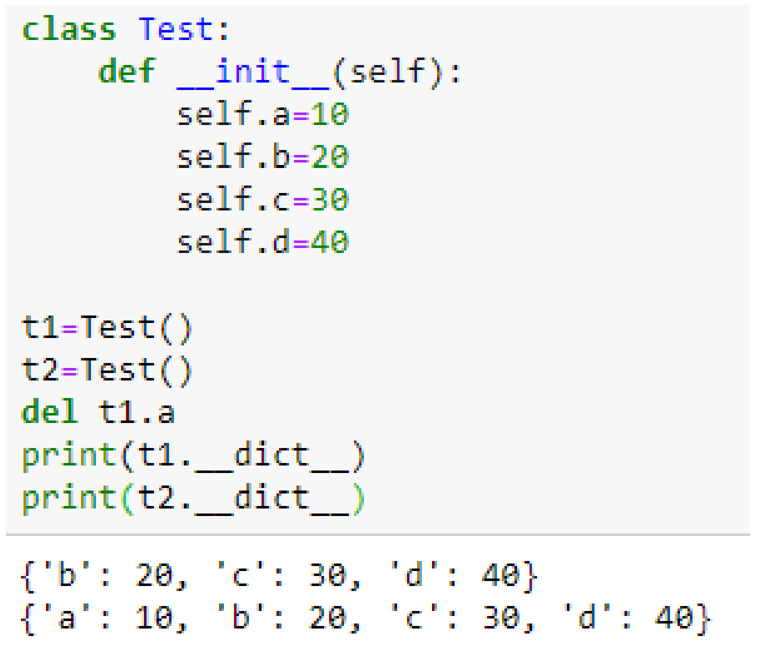
- Within a class we can delete instance variable as follows: **del self.variableName**
- From outside of class we can delete instance variables as follows: **del objectreference.variableName**



```
class Test:
    def __init__(self):
        self.a=10
        self.b=20
        self.c=30
        self.d=40
    def m1(self):
        del self.d

t=Test()
print(t.__dict__)
t.m1()
print(t.__dict__)
del t.c
print(t.__dict__)
```

```
{'a': 10, 'b': 20, 'c': 30, 'd': 40}
{'a': 10, 'b': 20, 'c': 30}
{'a': 10, 'b': 20}
```

Note: the instance variables which are deleted from one object, will not be deleted from other objects.



```
class Test:
    def __init__(self):
        self.a=10
        self.b=20
        self.c=30
        self.d=40

t1=Test()
t2=Test()
del t1.a
print(t1.__dict__)
print(t2.__dict__)
```

```
{'b': 20, 'c': 30, 'd': 40}
{'a': 10, 'b': 20, 'c': 30, 'd': 40}
```

STATIC VARIABLES

- If the value of a variable is not varied from object to object, such type of variables we have to declare with in the class directly but outside of methods. Such type of variables are called Static variables.
- For total class only one copy of static variable will be created and shared by all objects of that class.
- We can access static variables either by class name or by object reference. But recommended to use class name.
- In the case of instance variables for every object a separate copy will be created, but in the case of static variables for total class only one copy will be created and shared by every object of that class.

VARIOUS PLACES TO DECLARE STATIC VARIABLES

- In general we can declare within the class directly but from out side of any method
- Inside constructor by using class name
- Inside instance method by using class name
- Inside class method by using either class name or cls variable
- Inside static method by using class name
- from outside of class by using classname

HOW TO ACCESS STATIC VARIABLES

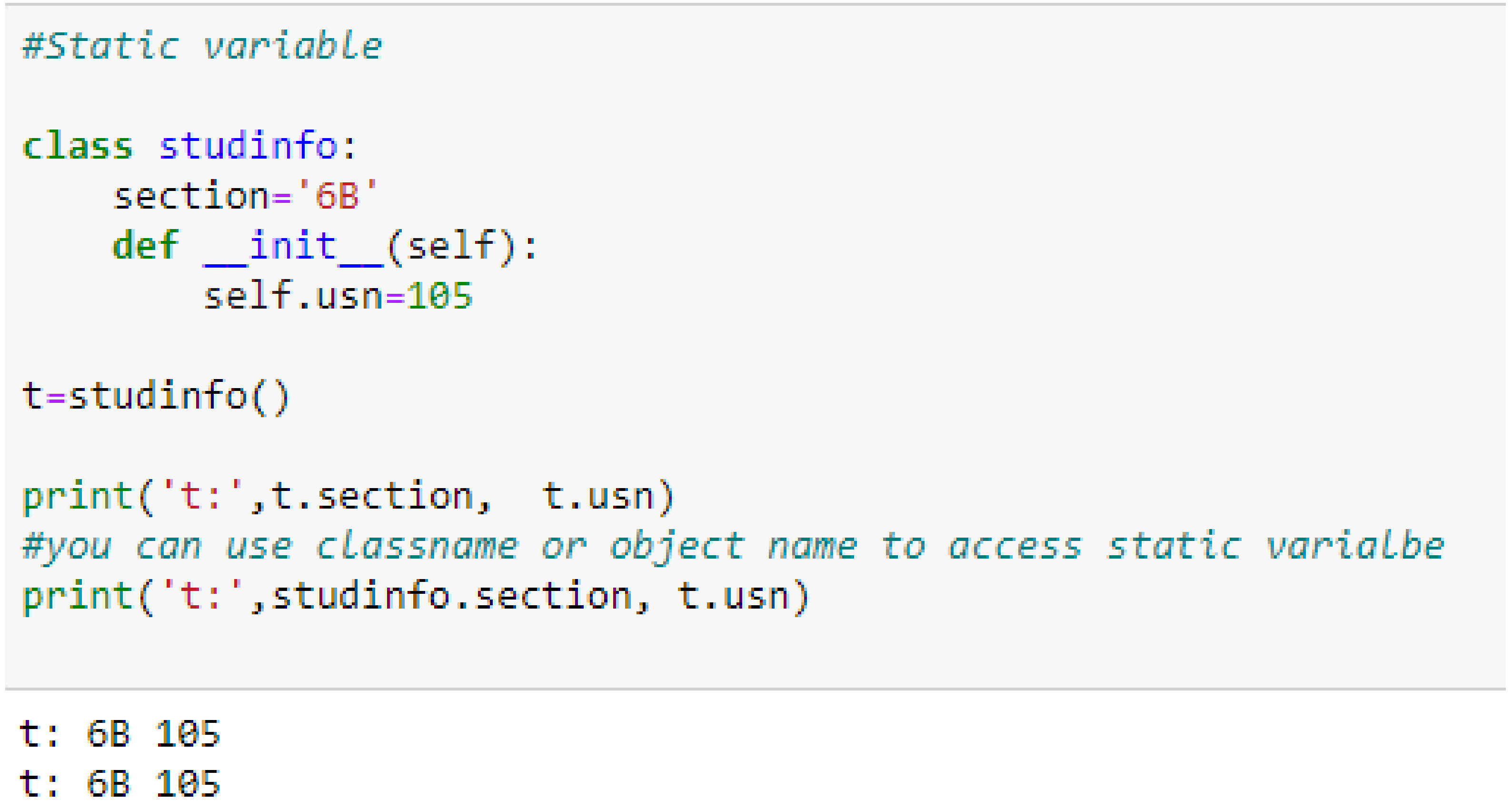
inside constructor	:	by using either self or classname
inside instance method	:	by using either self or classname
inside class method	:	by using either cls variable or classname
inside static method	:	by using classname
From outside of class	:	by using either object reference or classnmae

Where we can modify the value of static variable:

- Anywhere either with in the class or outside of class we can modify by using classname.
- But inside class method, by using cls variable.

DECLARE WITHIN THE CLASS DIRECTLY, BUT FROM OUT SIDE OF ANY METHOD

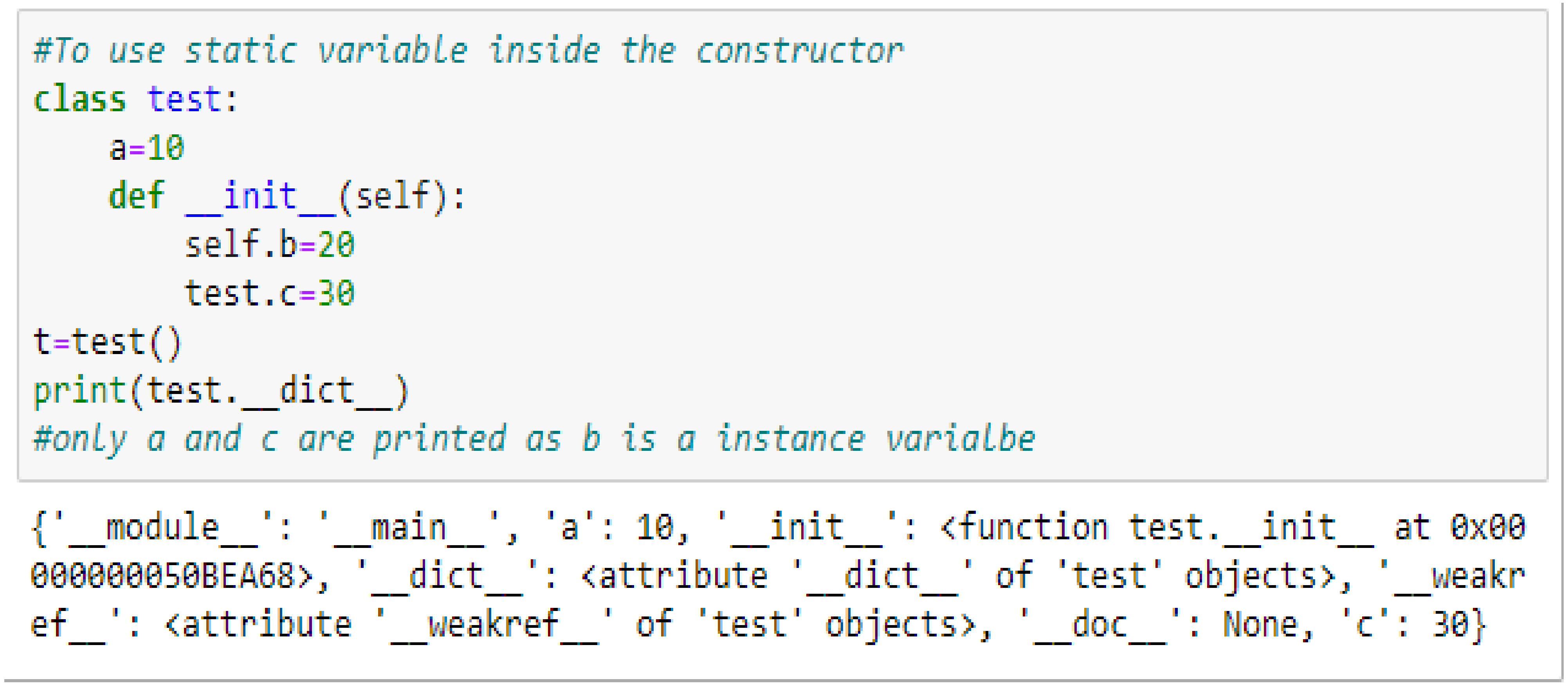
- In general we can declare within the class directly but from out side of any method



```
#Static variable

class studinfo:
    section='6B'
    def __init__(self):
        self.usn=105

t=studinfo()

print('t:',t.section,  t.usn)
#you can use classname or object name to access static varialbe
print('t:',studinfo.section, t.usn)
```

```
t: 6B 105
t: 6B 105
```

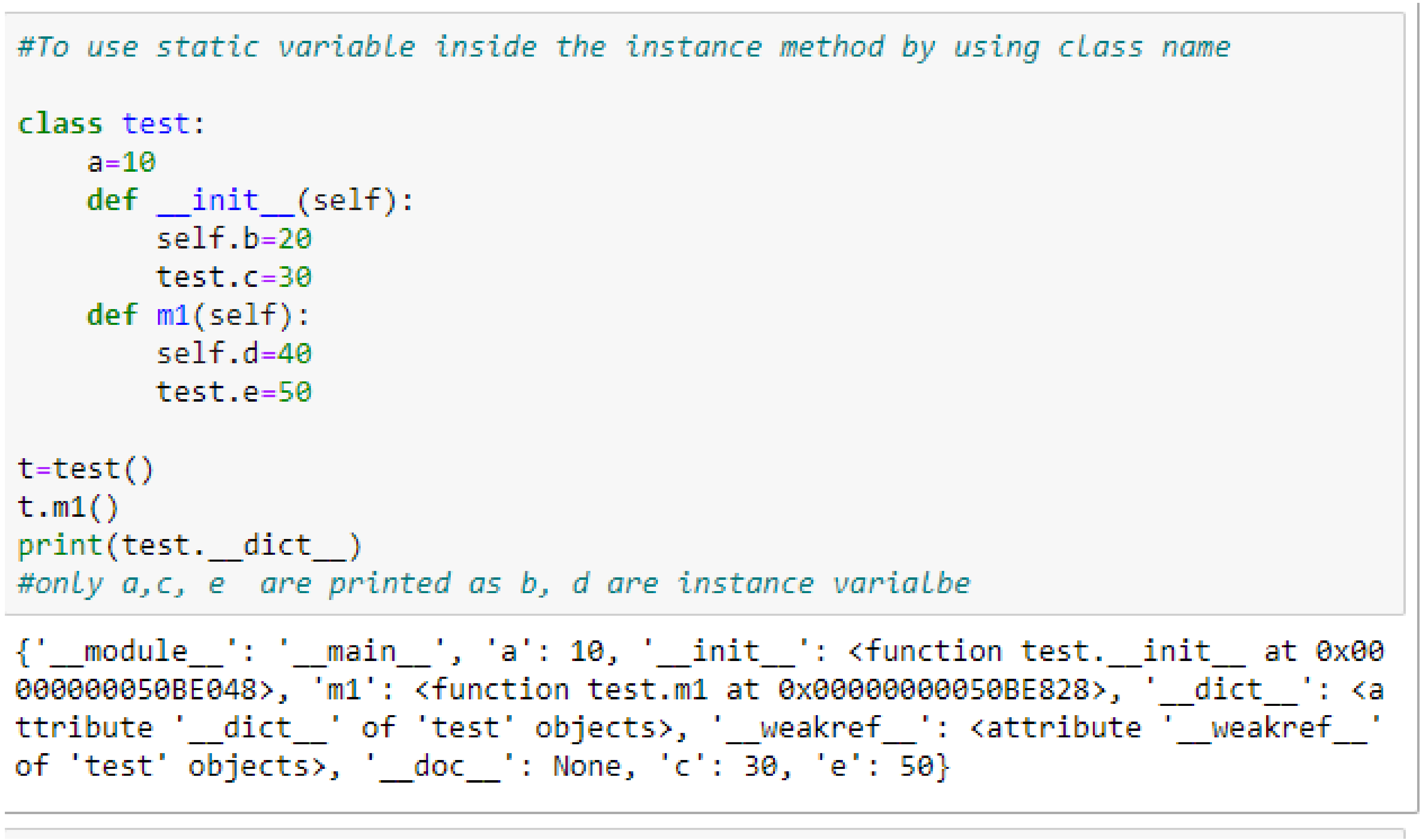
INSIDE CONSTRUCTOR BY USING CLASS NAME



```
#To use static variable inside the constructor
class test:
    a=10
    def __init__(self):
        self.b=20
        test.c=30
t=test()
print(test.__dict__)
#only a and c are printed as b is a instance varialbe
```

```
{'__module__': '__main__', 'a': 10, '__init__': <function test.__init__ at 0x00
000000050BEA68>, '__dict__': <attribute '__dict__' of 'test' objects>, '__weakr
ef__': <attribute '__weakref__' of 'test' objects>, '__doc__': None, 'c': 30}
```

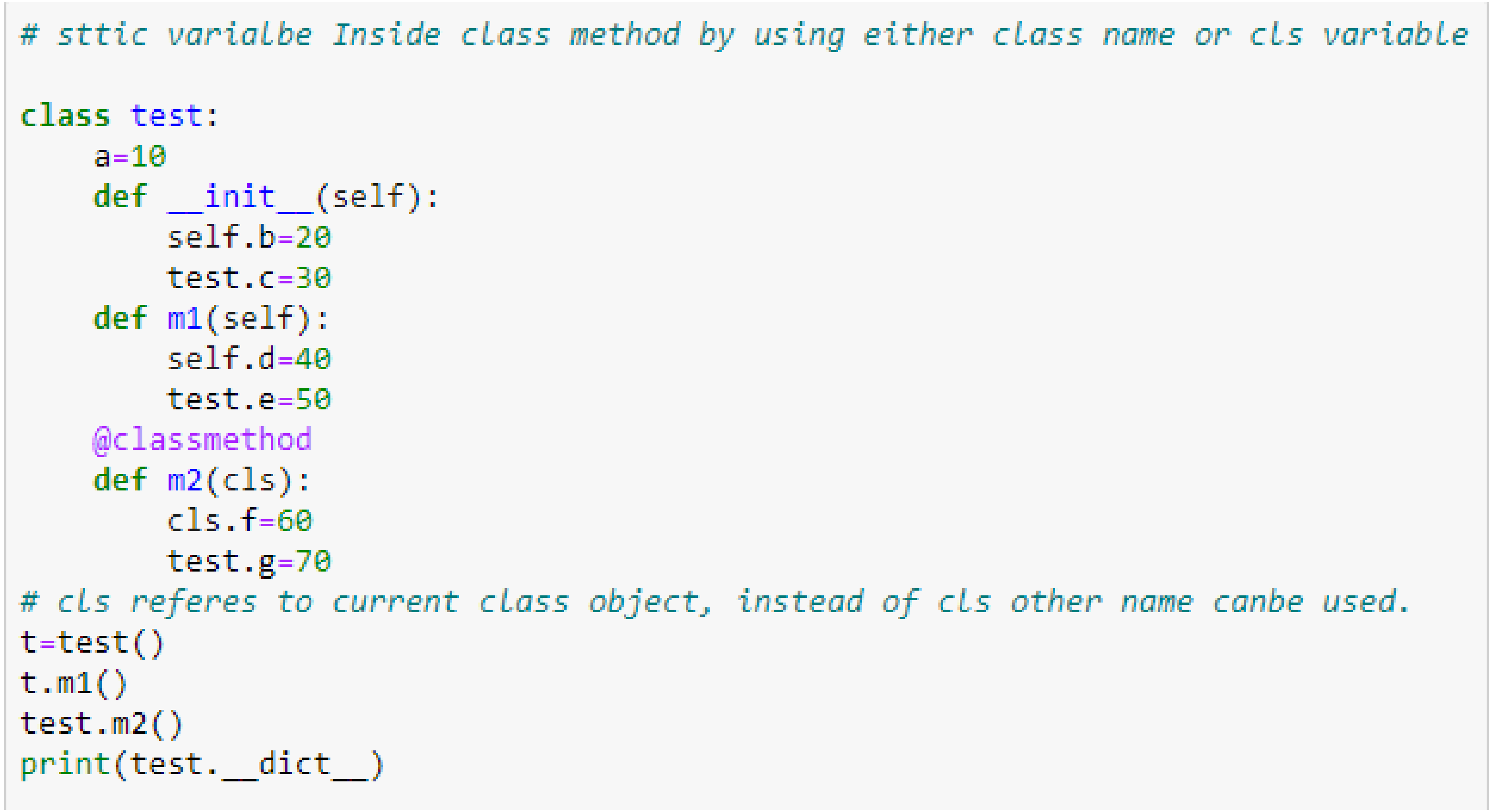
INSIDE INSTANCE METHOD BY USING CLASS NAME



```
#To use static variable inside the instance method by using class name

class test:
    a=10
    def __init__(self):
        self.b=20
        test.c=30
    def m1(self):
        self.d=40
        test.e=50

t=test()
t.m1()
print(test.__dict__)
#only a,c, e  are printed as b, d are instance varialbe
```

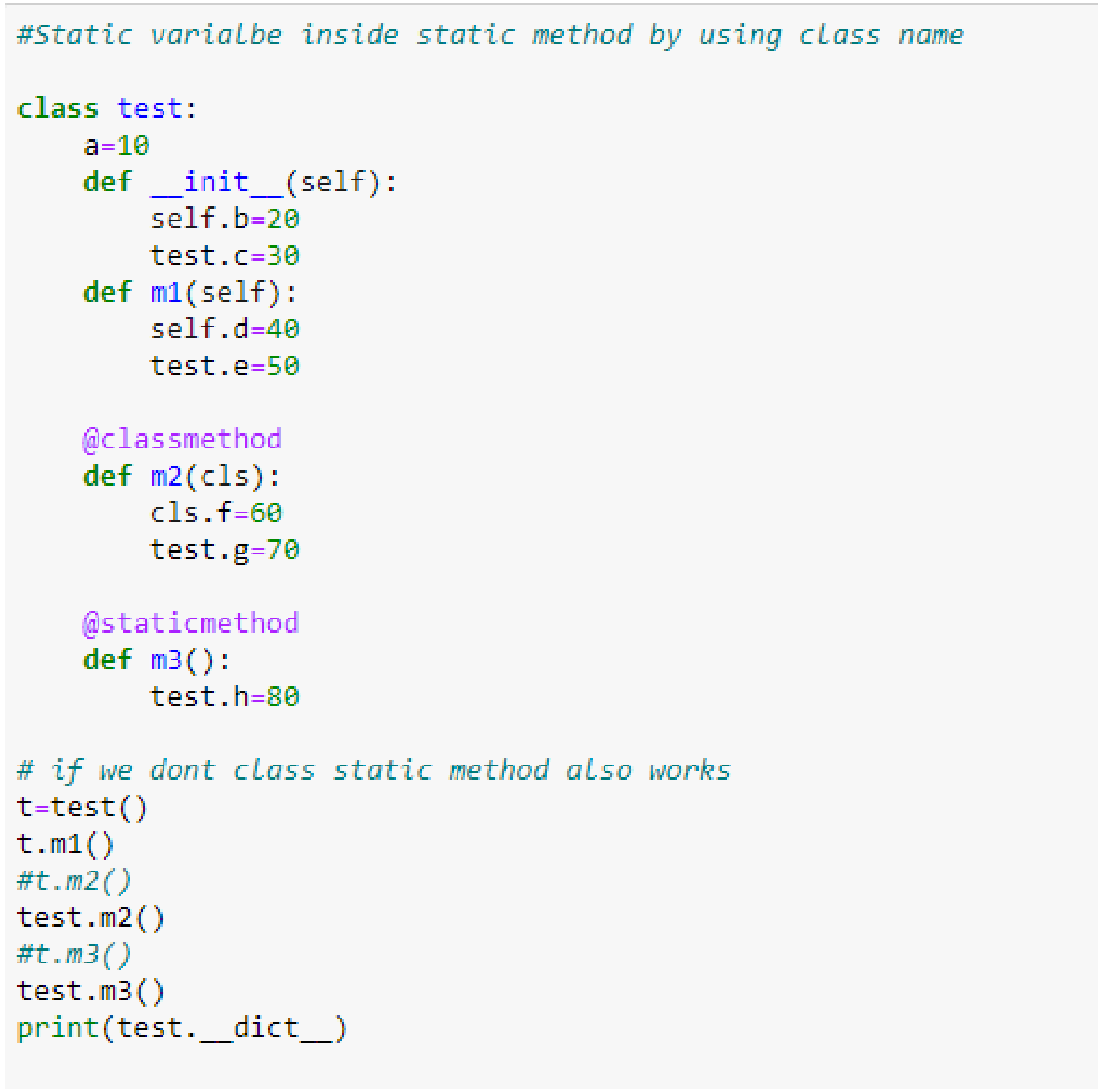
```
{'__module__': '__main__', 'a': 10, '__init__': <function test.__init__ at 0x00
000000050BE048>, 'm1': <function test.m1 at 0x00000000050BE828>, '__dict__': <a
ttribute '__dict__' of 'test' objects>, '__weakref__': <attribute '__weakref__'
of 'test' objects>, '__doc__': None, 'c': 30, 'e': 50}
```

INSIDE CLASS METHOD BY USING EITHER CLASS NAME OR CLS VARIABLE



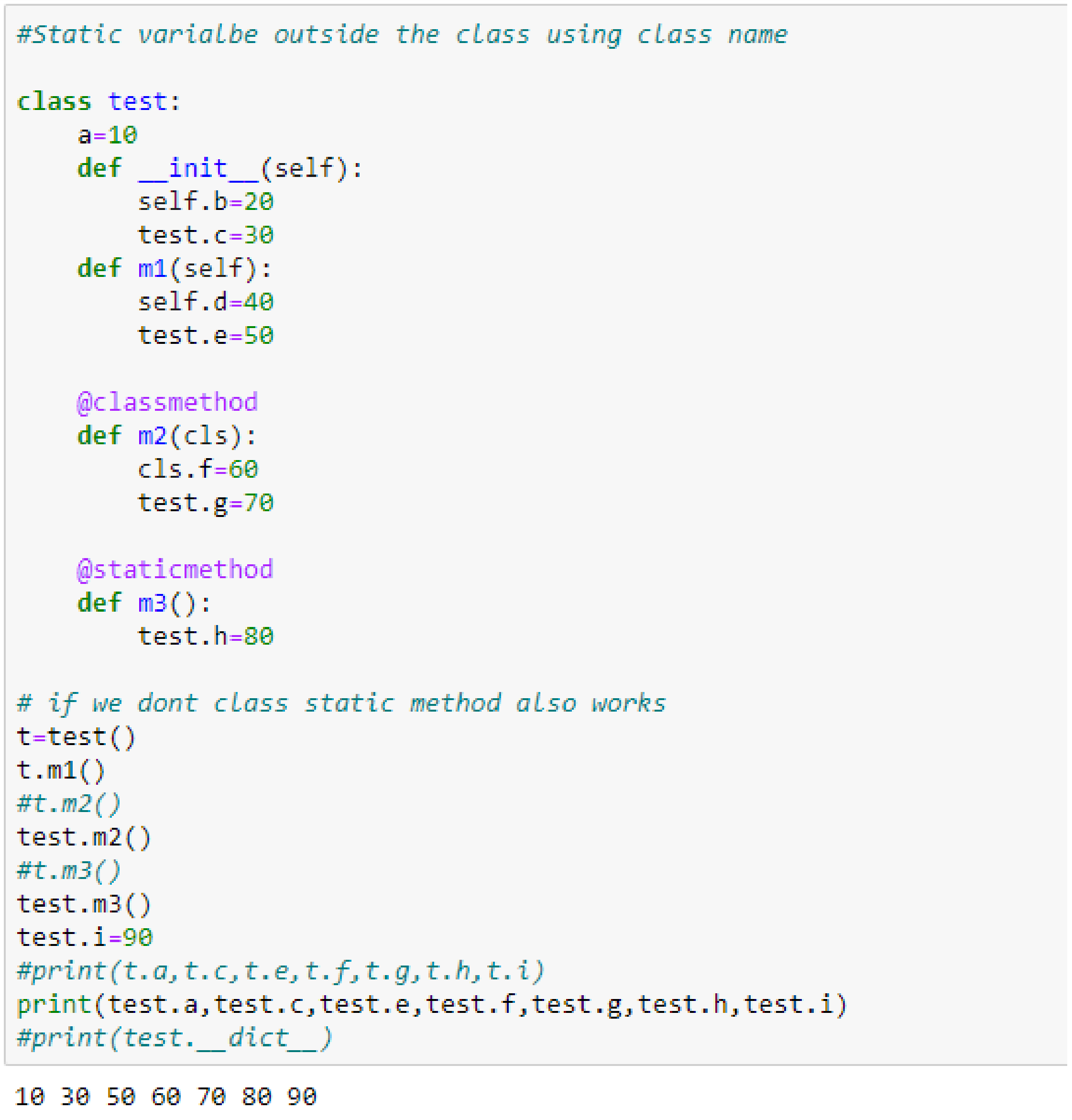
```
# sttic varialbe Inside class method by using either class name or cls variable

class test:
    a=10
    def __init__(self):
        self.b=20
        test.c=30
    def m1(self):
        self.d=40
        test.e=50
    @classmethod
    def m2(cls):
        cls.f=60
        test.g=70
# cls referes to current class object, instead of cls other name canbe used.
t=test()
t.m1()
test.m2()
print(test.__dict__)
```

INSIDE STATIC METHOD BY USING CLASS NAME



```
#Static varialbe inside static method by using class name

class test:
    a=10
    def __init__(self):
        self.b=20
        test.c=30
    def m1(self):
        self.d=40
        test.e=50

    @classmethod
    def m2(cls):
        cls.f=60
        test.g=70

    @staticmethod
    def m3():
        test.h=80

# if we dont class static method also works
t=test()
t.m1()
#t.m2()
test.m2()
#t.m3()
test.m3()
print(test.__dict__)
```

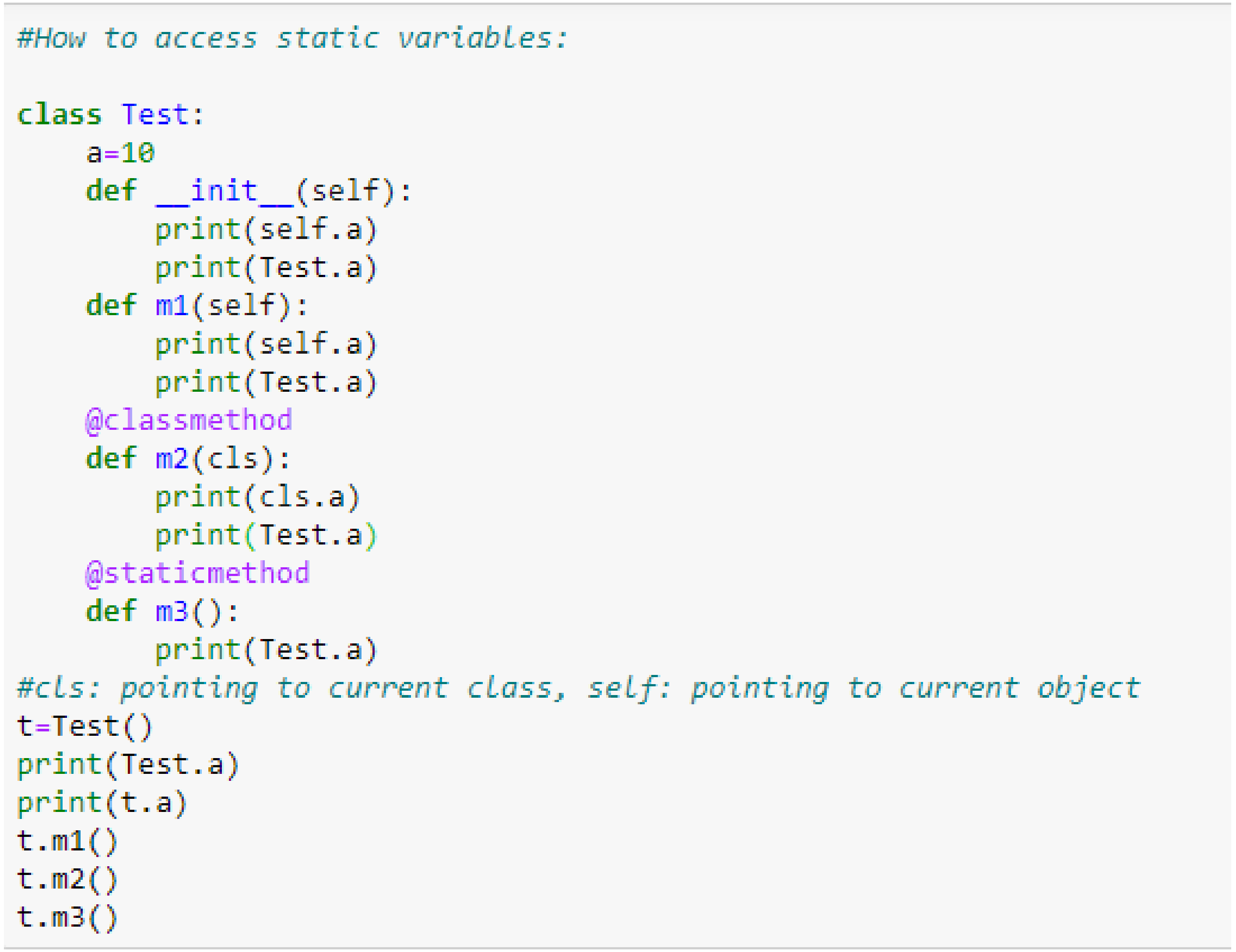
FROM OUTSIDE OF CLASS BY USING CLASSNAME



```python
#Static varialbe outside the class using class name

class test:
    a=10
    def __init__(self):
        self.b=20
        test.c=30
    def m1(self):
        self.d=40
        test.e=50

    @classmethod
    def m2(cls):
        cls.f=60
        test.g=70

    @staticmethod
    def m3():
        test.h=80

# if we dont class static method also works
t=test()
t.m1()
#t.m2()
test.m2()
#t.m3()
test.m3()
test.i=90
#print(t.a,t.c,t.e,t.f,t.g,t.h,t.i)
print(test.a,test.c,test.e,test.f,test.g,test.h,test.i)
#print(test.__dict__)
```

```
10 30 50 60 70 80 90
```

ACCESSING STATIC VARIABLES



```
#How to access static variables:

class Test:
    a=10
    def __init__(self):
        print(self.a)
        print(Test.a)
    def m1(self):
        print(self.a)
        print(Test.a)
    @classmethod
    def m2(cls):
        print(cls.a)
        print(Test.a)
    @staticmethod
    def m3():
        print(Test.a)
#cls: pointing to current class, self: pointing to current object
t=Test()
print(Test.a)
print(t.a)
t.m1()
t.m2()
t.m3()
```

DELETING STATIC VARIABLE

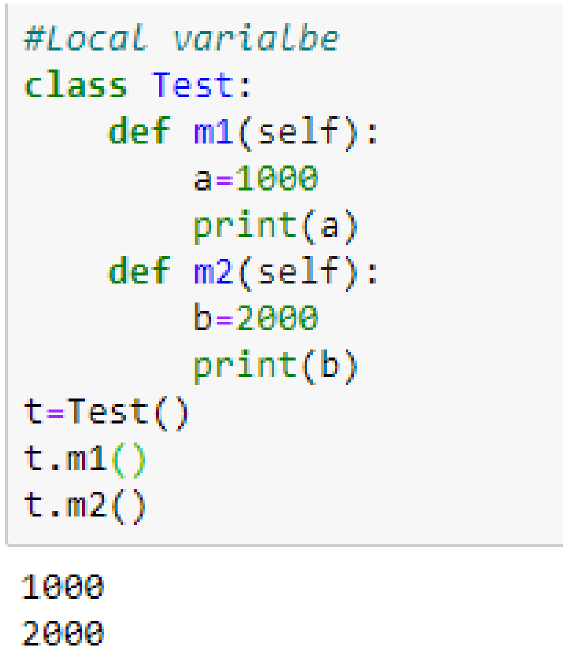
- If we change the value of static variable by using either self or object reference variable, then the value of static variable won't be changed, just a new instance variable with that name will be added to that particular object.
- **How to delete static variables of a class:**

  We can delete static variables from anywhere by using the following syntax:

  - ➢ del classname.variablename
  - ➢ But inside classmethod we can also use cls variable
    - ✓ del cls.variablename
- By using object reference variable/self we can read static variables, but we cannot modify or delete.
- If we are trying to modify, then a new instance variable will be added to that particular object.
- If we are trying to delete then we will get error.
- We can modify or delete static variables only by using classname or cls variable.

LOCAL VARIABLES

- Sometimes to meet temporary requirements of programmer, we can declare variables inside a method directly, such type of variables are called local variable or temporary variables.
- Local variables will be created at the time of method execution and destroyed once method completes.
- Local variables of a method cannot be accessed from outside of method.



```
#Local varialbe
class Test:
    def m1(self):
        a=1000
        print(a)
    def m2(self):
        b=2000
        print(b)
t=Test()
t.m1()
t.m2()
```

```
1000
2000
```

TYPES OF METHODS

- **Inside Python class 3 types of methods are allowed**

1. Instance Methods

2. Class Methods

3. Static Methods

INSTANCE METHODS

- Inside method implementation if we are using instance variables then such type of methods are called instance methods.
- Inside instance method declaration, we have to pass self variable.
    - def m1(self):
- By using self variable inside method we are able to access instance variables.
- Within the class we can call instance method by using self variable and from outside of the class we can call by using object reference.

CLASS METHODS

- Inside method implementation if we are using only class variables (static variables), then such type of methods we should declare as class method.
- We can declare class method explicitly by using @classmethod decorator.
- For class method we should provide cls variable at the time of declaration
- We can call classmethod by using classname or object reference variable.

STATIC METHODS

- In general these methods are general utility methods.
- Inside these methods we won't use any instance or class variables.
- Here we won't provide self or cls arguments at the time of declaration.
- We can declare static method explicitly by using @staticmethod decorator
- We can access static methods by using classname or object reference
- Note: In general we can use only instance and static methods.

  Inside static method we can access class level variables by using class name.
- class methods are most rarely used methods in python.

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 (Resource deallocation activities like close database connection etc).
- Once destructor execution completed then Garbage Collector automatically destroys that object.
- Note: The job of destructor is not to destroy object and it is just to perform clean up activities.
- Note: If the object does not contain any reference variable then only it is eligible for GC ie if the reference count is zero then only object eligible for GC

GARBAGE COLLECTION

- In Python, We have some assistant which is always running in the background to destroy useless objects. Because this assistant, the chance of failing Python program with memory problems is very less. This assistant is nothing but Garbage Collector.
- Hence the main objective of Garbage Collector is to destroy useless objects.
- If an object does not have any reference variable then that object eligible for Garbage Collection.

HOW TO ENABLE AND DISABLE GARBAGE COLLECTOR IN OUR PROGRAM:

- By default Garbage collector is enabled, but we can disable based on our requirement. In this context we can use the following functions of
- gc module.
- 1. gc.isenabled()

  Returns True if GC enabled
- 2. gc.disable()

  To disable GC explicitly
- 3. gc.enable()

  To enable GC explicitly

ENCAPSULATION

- Encapsulation is one of the fundamental concepts in object-oriented programming (OOP).
- Encapsulation is the process of combining variables that store data and methods that work on those variables into a single unit called class.
- In encapsulation, the variables are not accessed directly; it is accessed through the methods present in the class.
- Encapsulation ensures that the object's internal representation (its state and behaviour) are hidden from the rest of the application. Thus, encapsulation makes the concept of data hiding possible. Imagine if some programmer is able to access the data stored in a variable from outside the class then there would be a very high danger of them writing their own (not encapsulated) code to deal with your data stored in a variable. This would, at the very least, lead to code duplication (i.e., useless efforts) and to inconsistencies if the implementations are not perfectly compatible.
- Data hiding means that in order to access data stored in a variable, everybody must use the methods that are provided, so that they are the same for everybody.
- An application using a class does not need to know its internal workings or how it is implemented. The program simply creates an object and uses it to invoke the methods of that class.

ENCAPSULATION AND ABSTRACTION

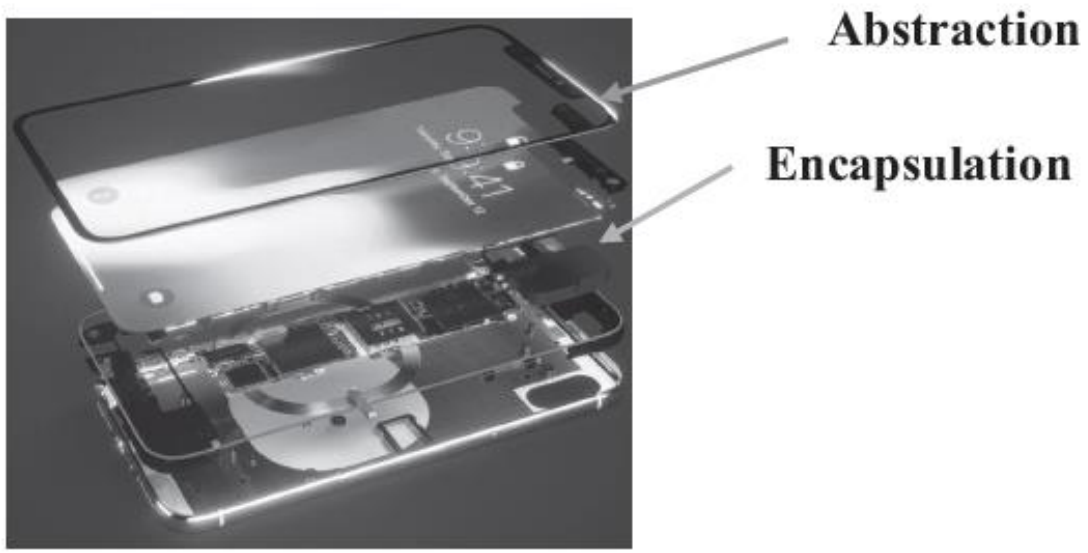


FIGURE 11.6
Demonstration of Abstraction and Encapsulation concept.

- Abstraction is a process where you show only "relevant" variables that are used to access data and "hide" implementation details of an object from the user.
- Consider your mobile phone you just need to know what buttons are to be pressed to send a message or make a call. What happens when you press a button, how your messages are sent, and how your calls are connected are all abstracted away from the user.
- Encapsulation → Information Hiding
- Abstraction → Implementation Hiding

Program to demonstrate the difference between abstraction and encapsulation

```
1. class foo:
2.     def __init__(self, a, b):
3.         self.a = a
4.         self.b = b

5.     def add(self):
6.         return self.a + self.b

7. foo_object = foo(3,4)
8. foo_object.add()
```

In the above program, the internal representation of an object of *foo* class ①–⑥ is hidden outside the class → Encapsulation. Any accessible member (data/method) of an object of *foo* is restricted and can only be accessed by that object ⑦–⑧. Implementation of *add()* method is hidden → Abstraction.

```python
#Program 11.14: Given a point(x, y), Write Python Program to Find Whether it Lies in the
#First, Second, Third or Fourth Quadrant of x - y Plane
class Quadrant:
    def __init__(self, x, y):
        self.x_coord = x
        self.y_coord = y
    def determine_quadrant(self):
        if self.x_coord > 0 and self.y_coord > 0:
            print(f"Point with coordinates {(self.x_coord, self.y_coord)} lies in the FIRST Quadrant")
        elif self.x_coord < 0 and self.y_coord < 0:
            print(f"Point with coordinates {(self.x_coord, self.y_coord)} lies in the THIRD Quadrant")
        elif self.x_coord > 0 and self.y_coord < 0:
            print(f"Point with coordinates {(self.x_coord, self.y_coord)} lies in the FOURTH Quadrant")
        elif self.x_coord < 0 and self.y_coord > 0:
            print(f"Point with coordinates {(self.x_coord, self.y_coord)} lies in the SECOND Quadrant")
def main():
    point = Quadrant(-180, 180)
    point.determine_quadrant()
if __name__ == "__main__":
    main()
```

```
Point with coordinates (-180, 180) lies in the SECOND Quadrant
```

USING PRIVATE INSTANCE VARIABLES, AND METHODS

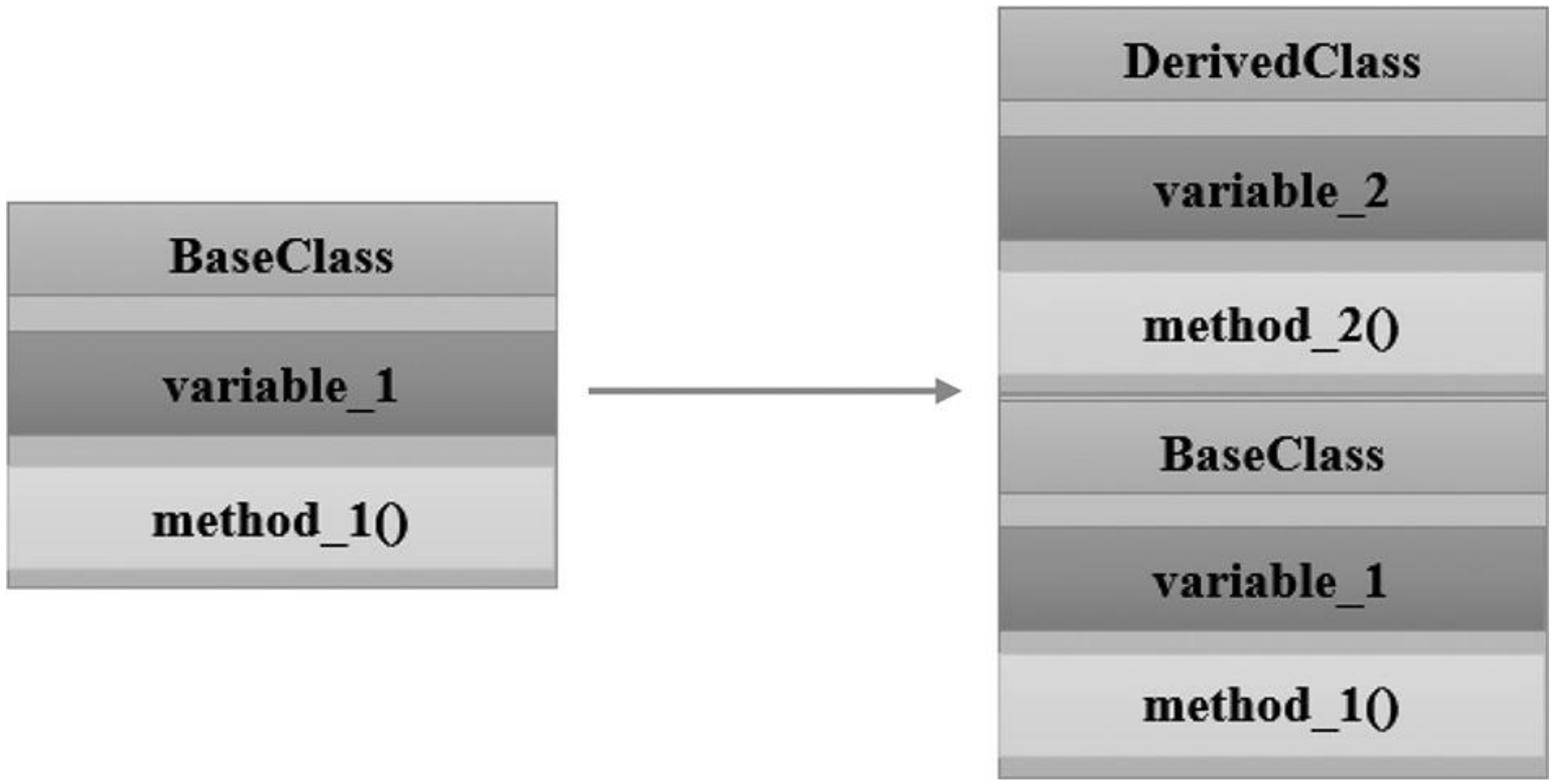
- Instance variables or methods, which can be accessed within the same class and can’t be seen outside, are called private instance variables or private methods.
- Since there is a valid use-case for class-only private members (namely to avoid name clashes of names with names defined by subclasses), there is support for such a mechanism, which is called name mangling.
- In Python, an identifier prefixed with a double underscore (e.g., __spam) and with no trailing underscores should be treated as private (whether it is a method or an instance variable).
- Any identifier of the form __spam is textually replaced with _classname__spam, where classname is the current class name with a leading underscore(s) stripped. This mangling is done without regard to the syntactic position of the identifier, as long as it occurs within the definition of a class.
- Name mangling is helpful for letting subclasses override methods without breaking intraclass method calls.
- Name mangling is intended to give classes an easy way to define “private” instance variables and methods, without having to worry about instance variables defined by derived classes.
- Note that the mangling rules are designed mostly to avoid accidents; it still is possible to access or modify a variable that is considered private.

```python
#Program 11.15: Program to Demonstrate Private Instance Variables in Python
class PrivateDemo:
    def __init__(self):
        self.nonprivateinstance = "I'm not a private instance"
        self.__privateinstance = "I'm private instance"
    def display_privateinstance(self):
        print(f"{self.__privateinstance} used within the method of a class")
def main():
    demo = PrivateDemo()
    print("Invoke Method having private instance")
    print(demo.display_privateinstance())
    print("Invoke non-private instance variable")
    print(demo.nonprivateinstance)
    print("Get attributes of the object")
    print(demo.__dict__)
    print("Trying to access private instance variable outside the class results in an error")
    print(demo.__privateinstance)
if __name__ == "__main__":
    main()
```

INHERITANCE

- Inheritance enables new classes to receive or inherit variables and methods of existing classes. Inheritance is a way to express a relationship between classes.
- If you want to build a new class, which is already similar to one that already exists, then instead of creating a new class from scratch you can reference the existing class and indicate what is different by overriding some of its behaviour or by adding some new functionality.
- A class that is used as the basis for inheritance is called a *superclass* or *base class*. A class that inherits from a base class is called a *subclass* or *derived class*. The terms *parent class* and *child class* are also acceptable terms to use respectively.
- A derived class inherits variables and methods from its base class while adding additional variables and methods of its own. Inheritance easily enables reusing of existing code.

RELATIONSHIP BETWEEN BASE CLASS AND DERIVED CLASS

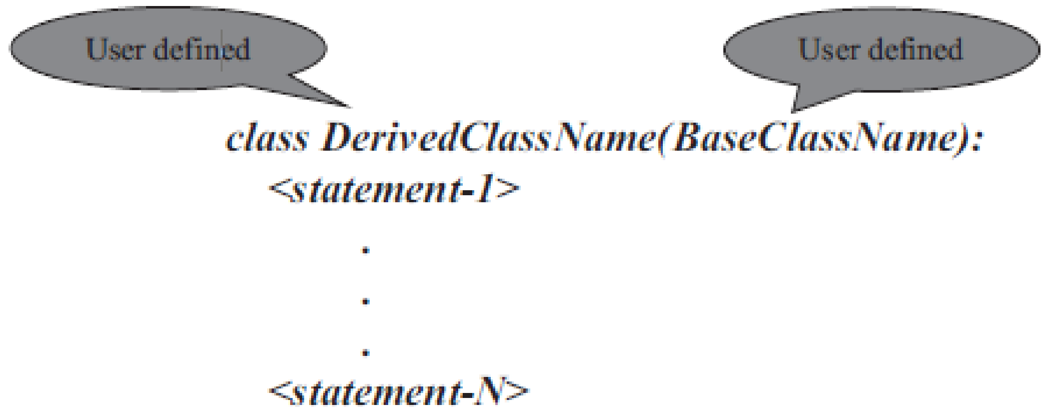


Class *BaseClass*, on the left, has one variable and one method.

Class *DerivedClass*, on the right, is derived from *BaseClass* and contains an additional variable and an additional method.

SYNTAX FOR A DERIVED CLASS

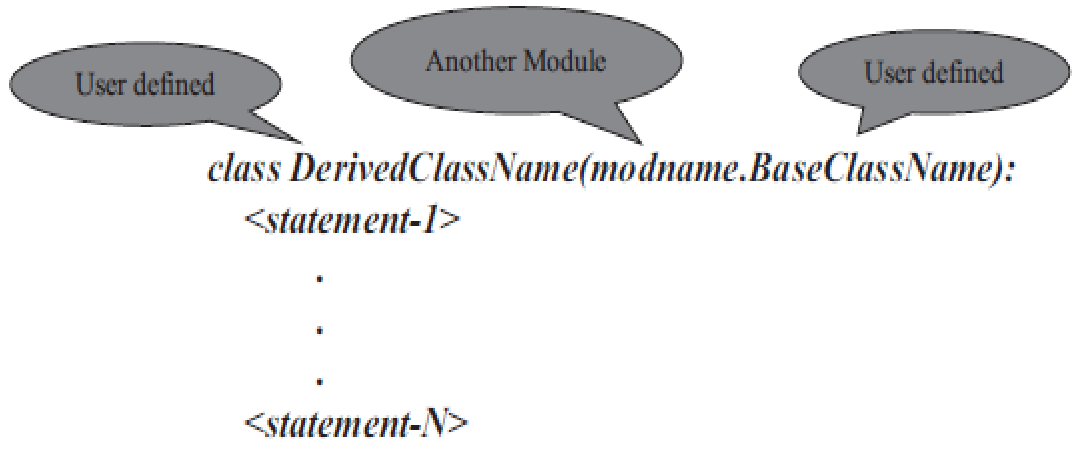
- The syntax for a derived class definition looks like this:



```
class DerivedClassName(BaseClassName):
    <statement-1>
        .
        .
        .
    <statement-N>
```

- To create a derived class, you add a *BaseClassName* after the *DerivedClassName* within the parenthesis followed by a colon. The derived class is said to directly inherit from the listed base class.

In place of a base class name, other arbitrary expressions are also allowed. This can be useful, for example, when the base class is defined in another module:



```
class DerivedClassName(modname.BaseClassName):
    <statement-1>
        .
        .
        .
    <statement-N>
```

ACCESSING THE INHERITED VARIABLES AND METHODS

- Execution of a derived class definition proceeds the same as for a base class.
- When the derived class object is constructed, the base class is also remembered.
- This is used for resolving variable and method attributes.
- If a requested attribute is not found in the derived class, the search proceeds to look in the base class.
- This rule is applied recursively if the base class itself is derived from some other class.
- Inherited variables and methods are accessed just as if they had been created in the derived class itself

BASE AND DERIVED CLASS RELATIONSHIP

```python
#Program 11.16: Program to Demonstrate Base and Derived Class Relationship
#Without Using __init__() Method in a Derived Class
class FootBall:
        def __init__(self, country, division, no_of_times):
            self.country = country
            self.division = division
            self.no_of_times = no_of_times
        def fifa(self):
            print(f"{self.country} national football team is placed in '{self.division}'- FIFA division")

class WorldChampions(FootBall):
        def world_championship(self):
            print(f"{self.country} national football team is {self.no_of_times} times world champions")

def main():
    germany = WorldChampions("Germany", "UEFA", 4)
    germany.fifa()
    germany.world_championship()

if __name__ == "__main__":
    main()
```

USING *SUPER()* FUNCTION AND OVERRIDING BASE CLASS METHODS

- In a single inheritance, the built-in *super()* function can be used to refer to base classes without naming them explicitly, thus making the code more maintainable.
- If you need to access the data attributes from the base class in addition to the data attributes being specified in the derived class's __init__() method, then you must explicitly call the base class __init__() method using *super()*, since that will not happen automatically.
- However, if you do not need any data attributes from the base class, then no need to use *super()* function to invoke base class __init__() method.
- The syntax for using *super()* in derived class __init__() method definition looks like this: ***super().__init__(base_class_parameter(s))***
- Its usage is shown below:

```
class DerivedClassName(BaseClassName):
    def __init__(self, derived_class_parameter(s), base_class_parameter(s))
        super().__init__(base_class_parameter(s))
        self.derived_class_instance_variable = derived_class_parameter
```

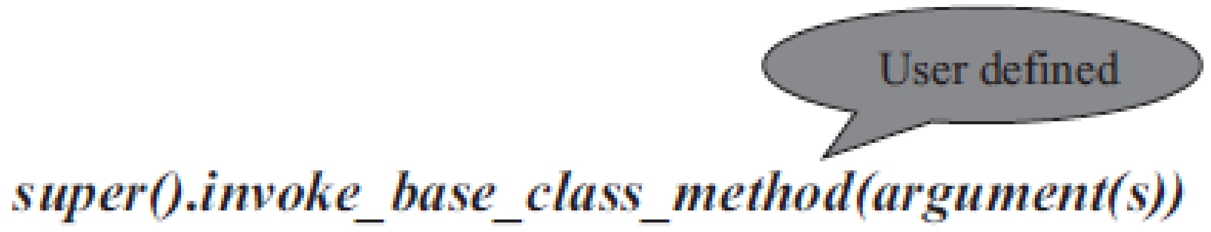
- The derived class __init__() method contains its own parameters along with the parameters specified in the __init__() method of base class. No need to specify self while invoking base class __init__() method using super(). No need to assign base_class_parameters specified in __init__() method of the derived class to any data attributes as the same is taken care of in the __init__() method of base class.

METHOD OVERRIDING

- Sometimes you may want to make use of some of the parent class behaviours but not all of them.
- Method overriding, in object-oriented programming, is a language feature that allows a derived class to provide its own implementation of a method that is already provided in base class.
- Derived classes may override methods of their base class. When you change the definition of parent class methods, you override them. These methods have the same name as those in the base class.
- The method in the derived class and the method in the base class each should have the same method signature.
- Method signature refers to the method name, order and the total number of its parameters. Return types and thrown exceptions are not considered to be a part of the method signature.
- An overriding method in a derived class may want to extend rather than simply replace the base class method of the same name.
- When constructing the base and derived classes, it is important to keep program design in mind so that overriding does not produce unnecessary or redundant code.

SUPER()

- The *super()* function is useful for accessing the base class methods that have been overridden in a derived class without explicitly specifying the base class name. The syntax for using the *super()* function to invoke the base class method is,



super().invoke_base_class_method(argument(s))

- The above expression should be used within a method of the derived class.
- The main advantage of using super() function comes with multiple inheritance.

```python
#Program 11.18: Program to Demonstrate the Overriding of the Base Class Method in the Derived Class
class Book:
    def __init__(self, author, title):
        self.author = author
        self.title = title
    def book_info(self):
        print(f"{self.title} is authored by {self.author}")

class Fiction(Book):
    def __init__(self, author, title, publisher):
        super().__init__(author, title)
        self.publisher = publisher
    def book_info(self):
        print(f"{self.title} is authored by {self.author} and published by {self.publisher}")
    def invoke_base_class_method(self):
        super().book_info()

def main():
    print("Derived Class")
    silva_book = Fiction("Daniel Silva", "Prince of Fire", "Berkley")
    silva_book.book_info()
    silva_book.invoke_base_class_method()
    print("---------------------------------")
    print("Base Class")
    reacher_book = Book("Lee Child", "One Shot")
    reacher_book.book_info()

if __name__ == "__main__":
    main()
#When the same method exists in both the base class and the derived class,
#the method in the derived class will be executed
```

MULTIPLE INHERITANCES

- Python also supports a form of multiple inheritances. A derived class definition with multiple base classes looks like:

```
class DerivedClassName(Base_1, Base_2, Base_3):
    <statement-1>
        .
        .
    <statement-N>
```

- You can call the base class method directly using Base Class name itself without using the super() function.

BaseClassName.methodname(self, arguments)

- Notice the difference between super() function and the base class name in calling the method name.
- Use issubclass() function to check class inheritance. The syntax is, *issubclass(DerivedClassName, BaseClassName).* This function returns Boolean True if DerivedClassName is a derived class of base class BaseClassName.
- The DerivedClassName class is considered a subclass of itself. BaseClassName may be a tuple of classes, in which case every entry in BaseClassName will be checked. In any other case, a TypeError exception is raised.

METHOD RESOLUTION ORDER (MRO)

- Method Resolution Order, or “MRO” in short, denotes the way Python programming language resolves a method found in multiple base classes.
- For a single inheritance, the MRO does not come into play but for multiple inheritances M anRO matters a lot.
- When a derived class inherits from multiple classes, Python programming language needs a way to resolve the methods that are found in multiple base classes as well as in the derived class, which is invoked by object.
- MRO uses the C3 linearization algorithm to determine the order of the methods to be invoked in multiple inheritances while guaranteeing monotonicity.
- MRO applies a set of rules to resolve the method order by constructing linearization.
- Python provides *mro()* method to get information about Method Resolution Order. The syntax to find MRO is,

  class_name.mro()

  where class_name is the name of a class.

Note: Method Resolution Order(MRO) Details, Diamond Problem – Refer Textbook

POLYMORPHISM

- Poly means *many* and morphism means *forms*. Polymorphism is one of the tenets of Object Oriented Programming (OOP). The literal meaning of polymorphism is the condition of occurrence in different forms.
- Polymorphism means that you can have multiple classes where each class implements the same variables or methods in different ways.
- Polymorphism takes advantage of inheritance in order to make this happen.
- A real-world example of polymorphism is suppose when if you are in classroom that time you behave like a student, when you are in market at that time you behave like a customer, when you are at your home at that time you behave like a son or daughter, such that same person is presented as having different behaviors.
- Polymorphism is a very important concept in programming. It refers to the use of a single type entity (method, operator or object) to represent different types in different scenarios.

POLYMORPHISM

- Difference between inheritance and polymorphism is, while inheritance is implemented on classes, polymorphism is implemented on methods.
- In Python, Polymorphism allows us to define methods in the derived class with the same name as defined in their base class.
- Python is a dynamically-typed language and specifically uses duck-typing. Duck-typing in Python allows us to use any object that provides the required methods and variables without forcing it to belong to any particular class.
- In duck-typing, an object's suitability is determined by the presence of methods and variables rather than the actual type of the object. To elaborate, this means that the expression some_obj.foo() will succeed if object some_obj has a foo method, regardless of to which class some_obj object actually belongs to.

OPERATOR OVERLOADING AND MAGIC METHODS

- Operator Overloading is a specific case of polymorphism, where different operators have different implementations depending on their arguments.
- A class can implement certain operations that are invoked by special syntax (such as arithmetic operations or subscripting and slicing) by defining methods with special names called “Magic Methods”. This is Python’s approach to operator overloading, allowing classes to define their own behaviour with respect to language operators.
- Python uses the word “Magic Methods” because these are special methods that you can define to add magic to your program. These magic methods start with double underscores and end with double underscores.
- One of the biggest advantages of using Python’s magic methods is that they provide a simple way to make objects behave like built-in types.
- Consider the standard + (plus) operator. When this operator is used with operands of different standard types, it will have a different meaning. The + operator performs arithmetic addition of two numbers, merges two lists, and concatenates two strings.

Magic Methods for Different Operators and Functions

Binary Operators

Operator	Method	Description
+	__add__(self, other)	Invoked for Addition Operations
-	__sub__(self, other)	Invoked for Subtraction Operations
*	__mul__(self, other)	Invoked for Multiplication Operations
/	__truediv__(self, other)	Invoked for Division Operations
//	__floordiv__(self, other)	Invoked for Floor Division Operations
%	__mod__(self, other)	Invoked for Modulus Operations
**	__pow__(self, other[, modulo])	Invoked for Power Operations
<<	__lshift__(self, other)	Invoked for Left-Shift Operations
>>	__rshift__(self, other)	Invoked for Right-Shift Operations
&	__and__(self, other)	Invoked for Binary AND Operations
^	__xor__(self, other)	Invoked for Binary Exclusive-OR Operations
|	__or__(self, other)	Invoked for Binary OR Operations

Extended Operators

Operator	Method	Description
+=	_iadd__(self, other)	Invoked for Addition Assignment Operations
-=	__isub(self, other)	Invoked for Subtraction Assignment Operations
*=	__imul__(self, other)	Invoked for Multiplication Assignment Operations
/=	__idiv__(self, other)	Invoked for Division Assignment Operations
//=	__ifloordiv__(self, other)	Invoked for Floor Division Assignment Operations
%=	__imod__(self, other)	Invoked for Modulus Assignment Operations
**=	__ipow__(self, other[, modulo])	Invoked for Power Assignment Operations
<<=	__ilshift__(self, other)	Invoked for Left-Shift Assignment Operations
>>=	__irshift__(self, other)	Invoked for Right-Shift Assignment Operations
&=	__iand__(self, other)	Invoked for Binary AND Assignment Operations

Magic Methods for Different Operators and Functions

^=	__ixor__(self, other)	Invoked for Binary Exclusive-OR Assignment Operations
|=	__ior__(self, other)	Invoked for Binary OR Assignment Operations

Unary Operators

Operator	Method	Description
-	__neg__(self)	Invoked for Unary Negation Operator
+	__pos__(self)	Invoked for Unary Plus Operator
abs()	__abs__()	Invoked for built-in function abs(). Returns absolute value
~	__invert__(self)	Invoked for Unary Invert Operator

Conversion Operations

Functions	Method	Description
complex()	__complex__(self)	Invoked for built-in complex() function
int()	__int__(self)	Invoked for built-in int() function
long()	__long__(self)	Invoked for built-in long() function
float()	__float__(self)	Invoked for built-in float() function
oct()	__oct__()	Invoked for built-in oct() function
hex()	__hex__()	Invoked for built-in hex() function

Comparison Operators

Operator	Method	Description
<	__lt__(self, other)	Invoked for Less-Than Operations
<=	__le__(self, other)	Invoked for Less-Than or Equal-To Operations
==	__eq__(self, other)	Invoked for Equality Operations
!=	__ne__(self, other)	Invoked for Inequality Operations
>=	__ge__(self, other)	Invoked for Greater Than or Equal-To Operations
>	__gt__(self, other)	Invoked for Greater Than Operations