Object Oriented Programming

BE(CSE) II-Semester

Prepared by
S. Durga Devi,
Assistant Professor,
CSE,CBIT

Unit-III

Classes and Objects: Introduction

- Classes and Objects
- __init__ method
- Class variables and Object variables
- Public and Private Data members
- calling methods from other methods
- built-in class attributes
- garbage collection
- class methods
- static methods.

Classes and objects

- Class: class is a plan or blue print to create objects.
- Object: physical existence in memory
- Class contains properties(attributes) and actions(methods)
- Properties represented as variables
- Actions as methods.
- Classes are implemented in python my using class keyword

- Syntax for class:

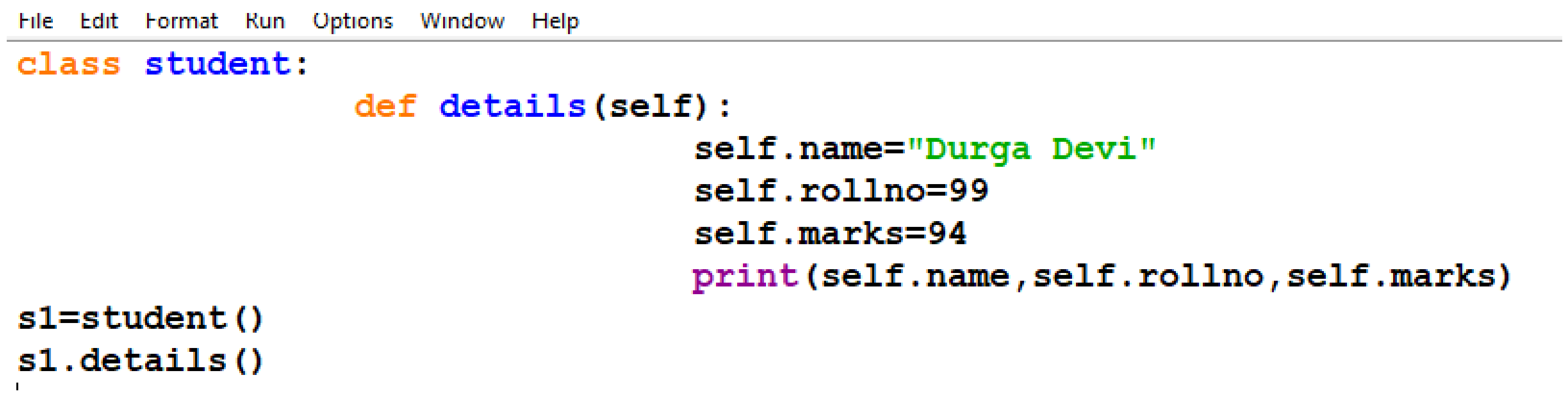
```
class classname:
        variable#instance variables, static and local variables
        methods # instance, static and class methods
```

Object creation
Reference variable =classname()

Reference variable or objectname are same

Members of a class accessed through objects
 - objectname.membername

S. Durga Devi ,CSE,CBIT



```python
class student:
                def details(self):
                                self.name="Durga Devi"
                                self.rollno=99
                                self.marks=94
                                print(self.name,self.rollno,self.marks)
s1=student()
s1.details()
```

Output

```
Durga Devi 99 94
```

constructor

- Constructor is a special method used to initialize instance variable when the objects are created.
- Constructor defined with __init__(two underscores)
- Constructor is called when the objects are created.
- Constructor takes first argument as self.

- **self variable:**

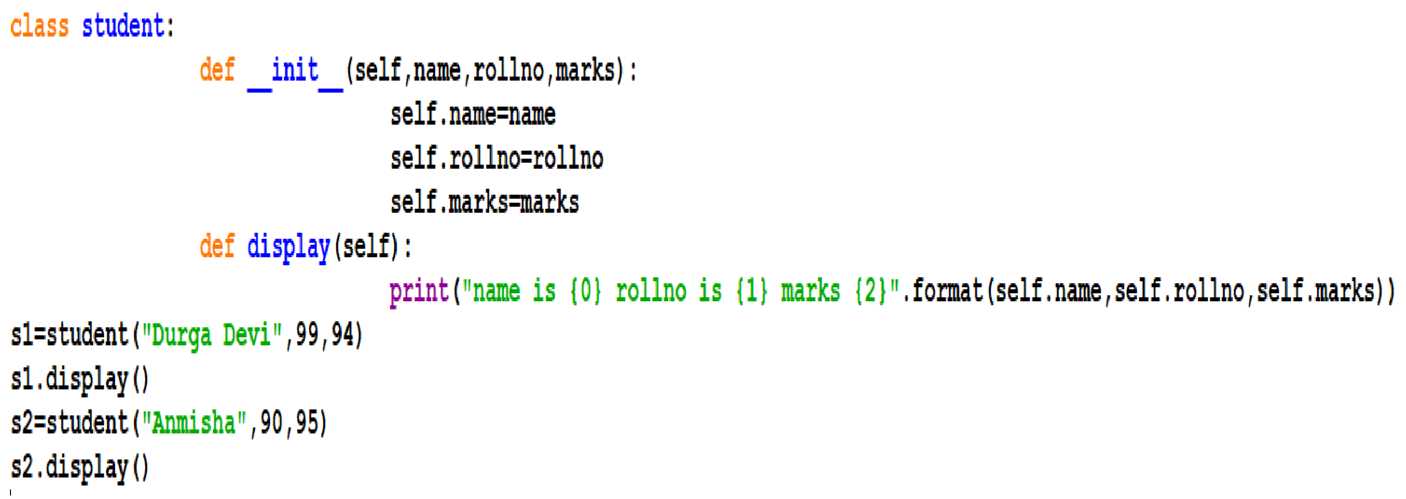
  the reference variable pointing to current object is called self varible.

- Self is used to access instance variables and instance methods.
- Self should be first parameter for instance method and constructor.
- Self is used within the class only.
- Default first argument to a constructor and method is object .
- Example

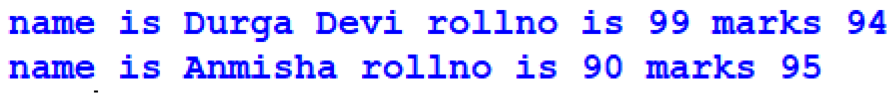
```
class student:
                def f1(self):
                                print("address of self is same as s1",id(self))
                                print("yes self points current object")
s1=student()
print("address of s1 is ",id(s1))
s1.f1()
```

```
address of s1 is  55632176
address of self is same as s1 55632176
yes self points current object
```

Example on constructor



```
class student:
        def __init__(self,name,rollno,marks):
                self.name=name
                self.rollno=rollno
                self.marks=marks
        def display(self):
                print("name is {0} rollno is {1} marks {2}".format(self.name,self.rollno,self.marks))
s1=student("Durga Devi",99,94)
s1.display()
s2=student("Anmisha",90,95)
s2.display()
```



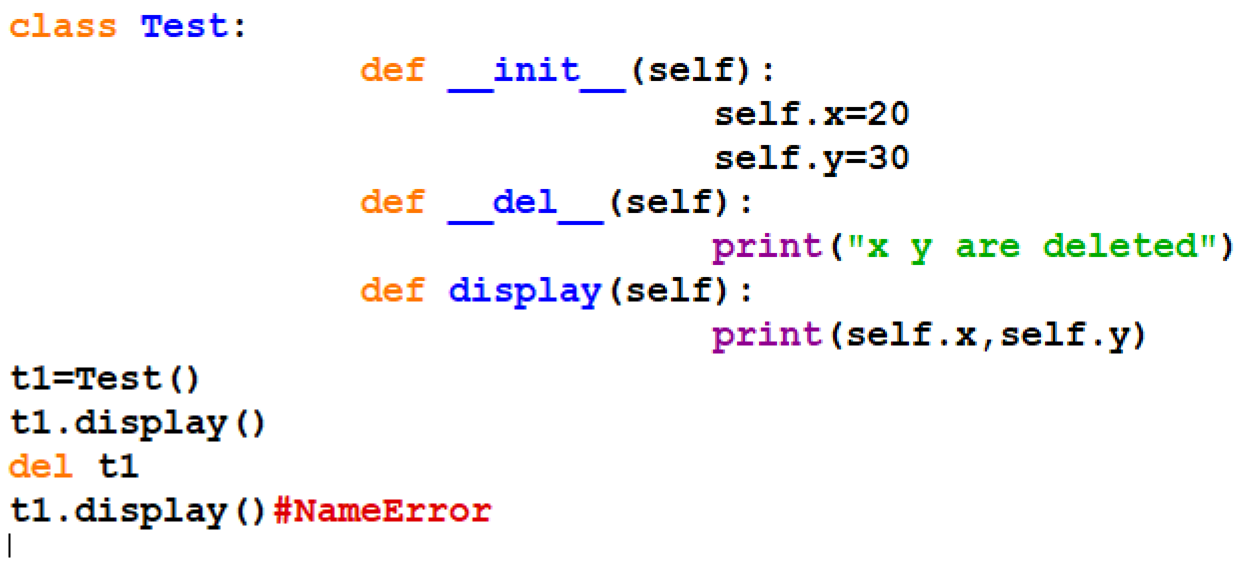
```
name is Durga Devi rollno is 99 marks 94
name is Anmisha rollno is 90 marks 95
```

Difference between method and constructor

method	constructor
Method name can be any valid identifier	name must be __init__(self)
method has business logic	Initialize variables
invoked through object name and method name	Invoked automatically when objects are created

__del__() method

- __init__() method is used to initialize the object when it is created.
- __del__() method is opposite to __init__() method.
- __del__() method is used to destruct the objects which are no longer needed and its occupied memory return back to system so that deallocated memory reused by other objects.



```
class Test:
                def __init__(self):
                                self.x=20
                                self.y=30
                def __del__(self):
                                print("x y are deleted")
                def display(self):
                                print(self.x,self.y)
t1=Test()
t1.display()
del t1
t1.display()#NameError
```

Types of variables

- Three types of variables

1. Instance variables(object level variables)
2. Static variables(class level variables)
3. Local variables(method level variables)

1. Instance variables

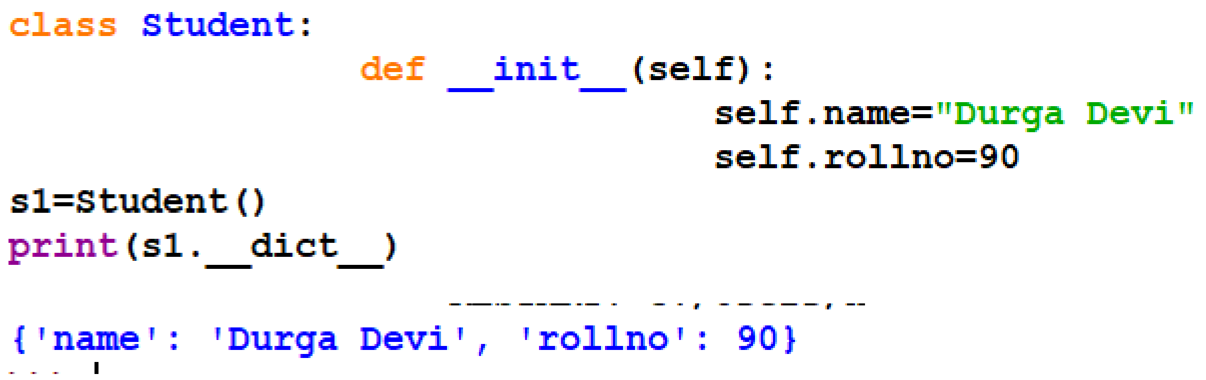
- Instance variable: value of the variable changed from object to object.
- A separate copy of the variable is created for each object.

Where instance variables are declared?

1. Inside constructor using self
2. Inside instance method using self
3. Out side class using reference variable.

1. Inside constructor using self:

- ➢ Instance variables are defined inside a constructor using self key word.
- ➢ Instance variables are initialized when the objects are created.
- ➢ Instance variables are added to object when objects are created

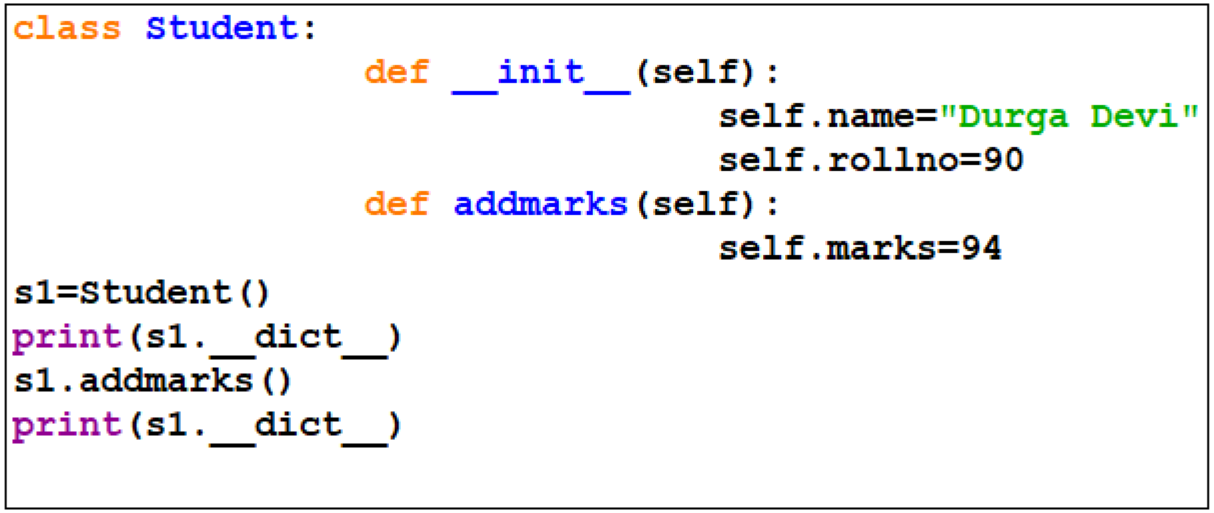


```
class Student:
                def __init__(self):
                                self.name="Durga Devi"
                                self.rollno=90
s1=Student()
print(s1.__dict__)
```

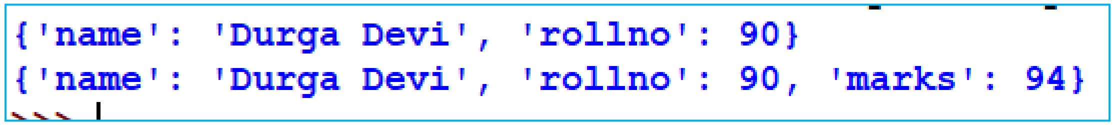
```
{'name': 'Durga Devi', 'rollno': 90}
```

2. Inside instance method using self

- Instance variables defined inside instance method using self keyword.
- Instance variables added to object once the method is called.



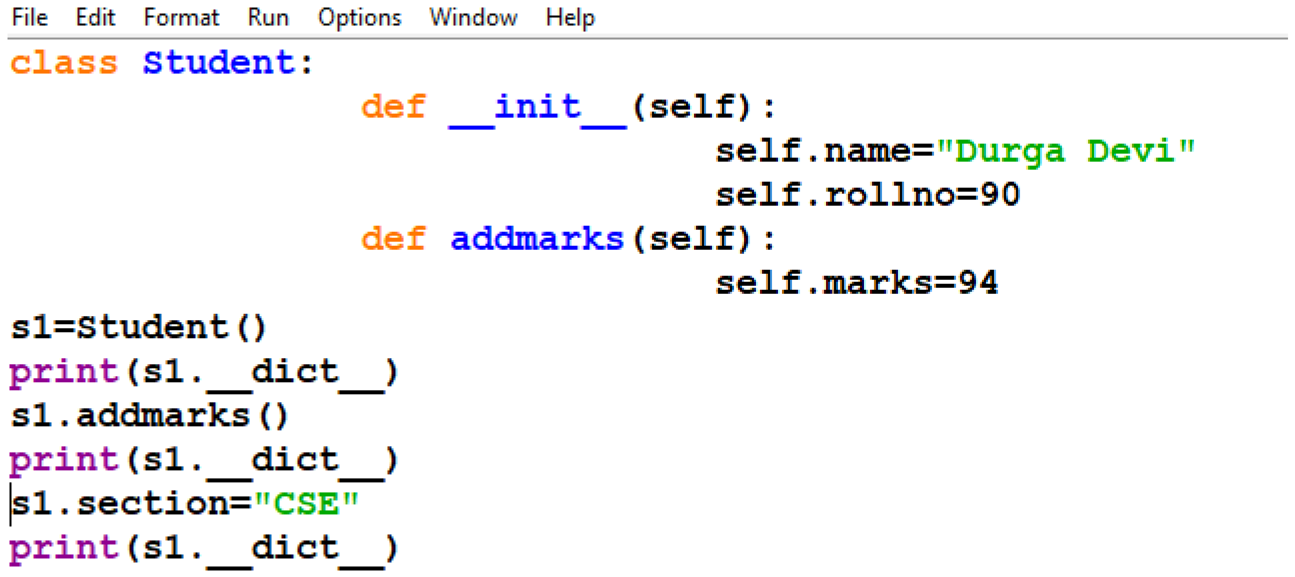
```
class Student:
                def __init__(self):
                                self.name="Durga Devi"
                                self.rollno=90
                def addmarks(self):
                                self.marks=94
s1=Student()
print(s1.__dict__)
s1.addmarks()
print(s1.__dict__)
```



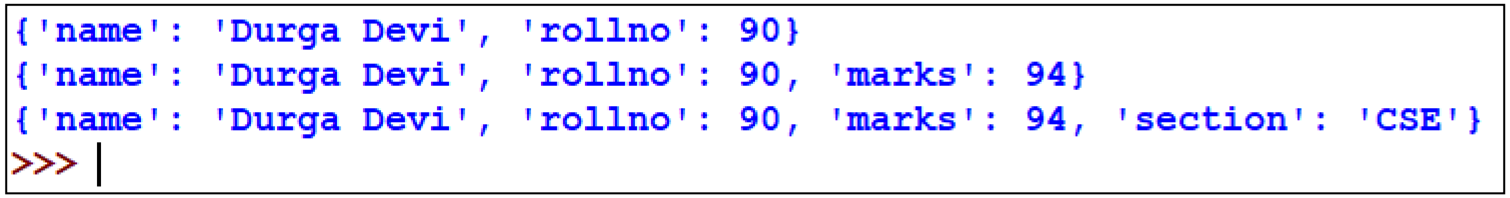
```
{'name': 'Durga Devi', 'rollno': 90}
{'name': 'Durga Devi', 'rollno': 90, 'marks': 94}
```

3. Outside class using object reference variable

- Instance variables added to a particular object



```
class Student:
                def __init__(self):
                                self.name="Durga Devi"
                                self.rollno=90
                def addmarks(self):
                                self.marks=94
s1=Student()
print(s1.__dict__)
s1.addmarks()
print(s1.__dict__)
s1.section="CSE"
print(s1.__dict__)
```



```
{'name': 'Durga Devi', 'rollno': 90}
{'name': 'Durga Devi', 'rollno': 90, 'marks': 94}
{'name': 'Durga Devi', 'rollno': 90, 'marks': 94, 'section': 'CSE'}
>>> 
```

Deletion of instance variables

- Delete inside class

  del self.variable_name

- Delete outside class

  del Objectreferencevariable.variablename

- Instance variables which are deleted from one object will not effect to another object.

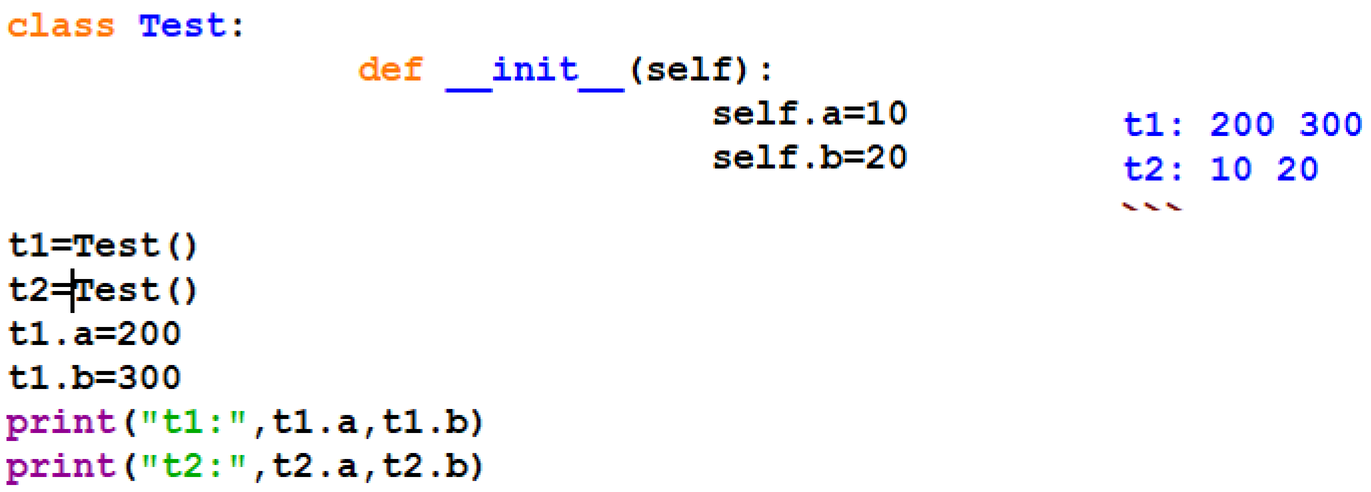
```
class Test:
        def __init__(self):
                self.a=10
                self.b=20
                print(self.a)
                del self.a


        def m1(self):
                self.c=30

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

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

- 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()
t2=Test()
t1.a=200
t1.b=300
print("t1:",t1.a,t1.b)
print("t2:",t2.a,t2.b)
```

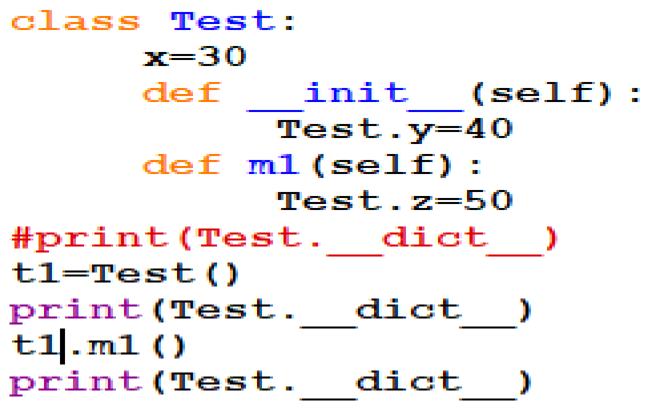
```
t1: 200 300
t2: 10 20
```

2. Static variables

- Static variables are also called class variables.
- Variables which are defined in a class but out side of a methods are called static variables.
- Static variable a copy will be created that copy is shared by all the objects created for that class.
- Static variables are accessed by either class name or object name.
- When the static variables are modified using self or object name those changes will not affect other objects
- To affect changes to all the objects, access static variables using class name.

Declaring static variables

- Out side the method and inside the class we can declare in general.
- Inside constructor using class name
- Inside instance method using class name
- Inside class method using class name or cls variable
- Inside static method using class name.



```
class Test:
        x=30
        def __init__(self):
                Test.y=40
        def m1(self):
                Test.z=50
#print(Test.__dict__)
t1=Test()
print(Test.__dict__)
t1.m1()
print(Test.__dict__)
```

classVariableDeclaration.py

Accessing static variables

- 1.inside constructor: by using either self or class name
- 2. inside instance method: by using either self or class name
- 3. inside class method: by using either class variable or class name
- 4. inside static method: by using class name
- 5. From outside of class: by using either object reference or class name

Class variables

```
#use of class or static variables
# class variables are shared by the more than one object of the same class
class Birthday:
                choclate=60
                def __init__(self,name):
                                self.name=name
                                print("happy birthday ",self.name)
                                print("thanks for choclate")
                                Birthday.choclate=Birthday.choclate-1
                def nochoclates(self):
                                print("no choclates left",self.choclate)
ram=Birthday("Hasan")
ram.nochoclates()
jamesbond=Birthday("Hasan")
jamesbond.nochoclates()
```

classVariableEx.py

```
class Test:
                count=0
                def __init__(self):
                                Test.count=Test.count+1
                                print("no of objects created are",self.count)
t1=Test()
t2=Test()
t3=Test()
```

```
no of objects created are 1
no of objects created are 2
no of objects created are 3
>>>
```

3. Local variables

- Local variables: variables which are declared inside of a method.
- Local variables scope and life time will be within the method
- Out side of a method local variables cannot be accessed.

Access modifiers

- In object oriented programming access modifiers are used to restrict access to members of a class(variables and methods).
- Access modifiers also called access specifiers.
- Access modifiers secure data from the unauthorized access and prevent data from being modified.
- Three access modifiers in python.

1. Public
2. Private
3. Protected.

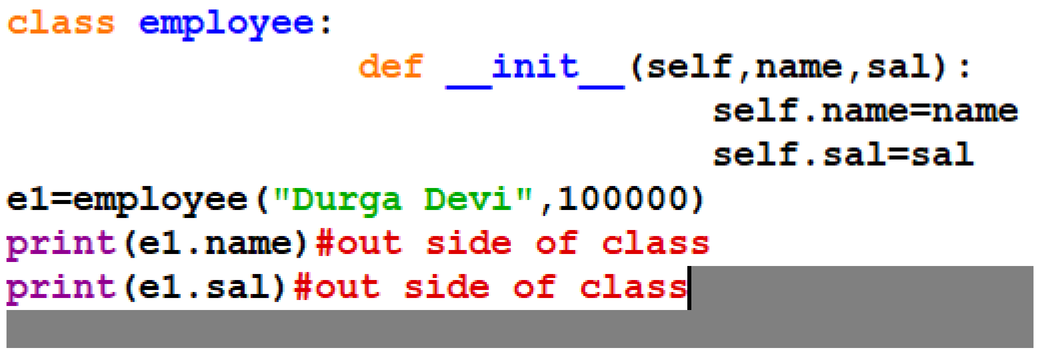
Public and private data members

1. Public members:

   - default all members in python are public

   - public members can be accessed out side of the class through an object of a class.

Example



```
class employee:
                def __init__(self,name,sal):
                                self.name=name
                                self.sal=sal
e1=employee("Durga Devi",100000)
print(e1.name)#out side of class
print(e1.sal)#out side of class
```

- 2. private members:

# Defined with two underscore(__) followed by member name.

# Private members accessed within a class not outside of a class.

Example

```
class employee:
                def __init__(self,name,sal):
                                self.__name=name
                                self.__sal=sal
e1=employee("Durga Devi",100000)
print(e1.__name)#AttributeError: 'employee' object has no attribute '__name'
print(e1.__sal)
```

Is there any way to access private members outside of a class?

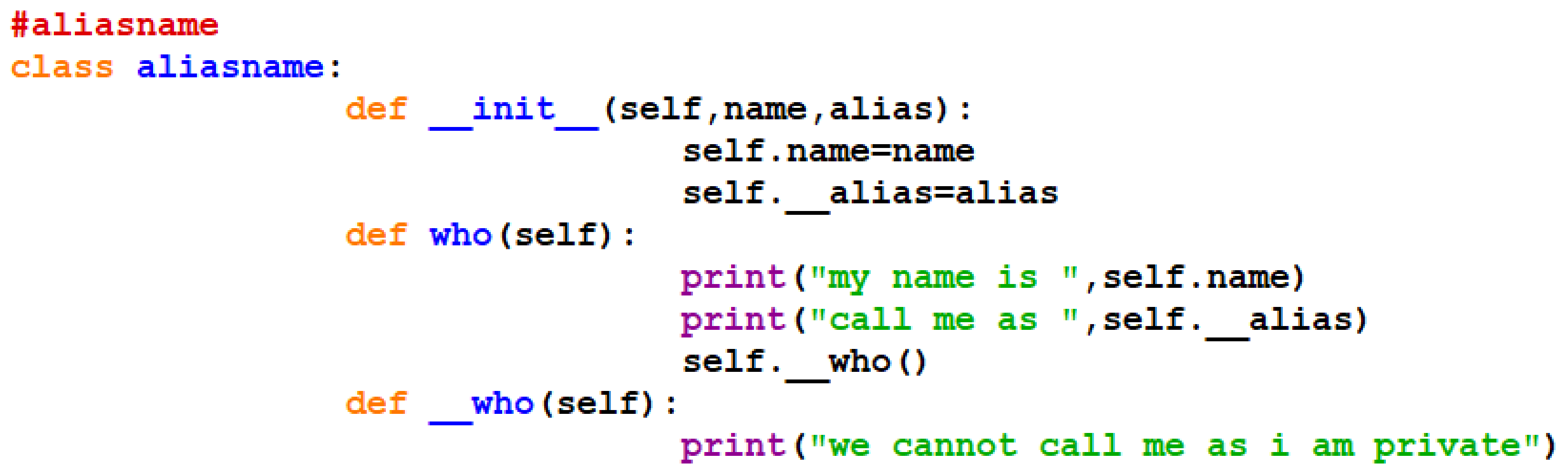
Object._classname__membernames

e1._employee__sal

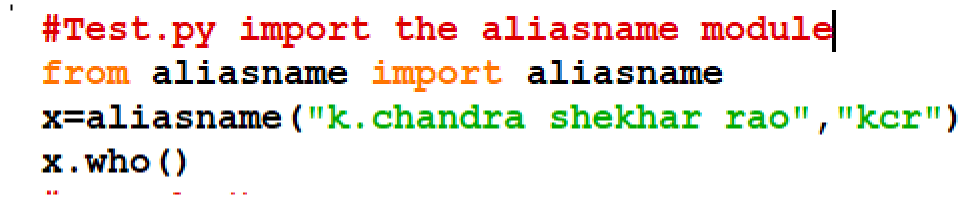
S. Durga Devi ,CSE,CBIT

Private methods in python

aliasname.py

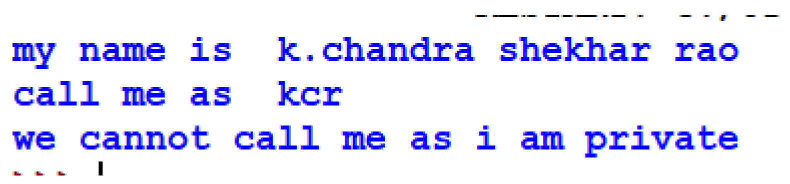


```
#aliasname
class aliasname:
                def __init__(self,name,alias):
                                self.name=name
                                self.__alias=alias
                def who(self):
                                print("my name is ",self.name)
                                print("call me as ",self.__alias)
                                self.__who()
                def __who(self):
                                print("we cannot call me as i am private")
```



```
 #Test.py import the aliasname module
 from aliasname import aliasname
 x=aliasname("k.chandra shekhar rao","kcr")
 x.who()
```

output



```
my name is  k.chandra shekhar rao
call me as  kcr
we cannot call me as i am private
```

```
#Test.py import the aliasname module
from aliasname import aliasname
x=aliasname("k.chandra shekhar rao","kcr")
x.who()
x.__who()
```

output

```
my name is  k.chandra shekhar rao
call me as  kcr
we cannot call me as i am private
Traceback (most recent call last):
  File "C:/Users/hp/Desktop/Test.py", line 5, in <module>
    x.__who()
AttributeError: 'aliasname' object has no attribute '__who'
```

3. Protected members:

- Accessed from a class and its sub class.
- Defined with single underscore symbol.

Can we access members of one class from other class

- We can access members of one class from other class

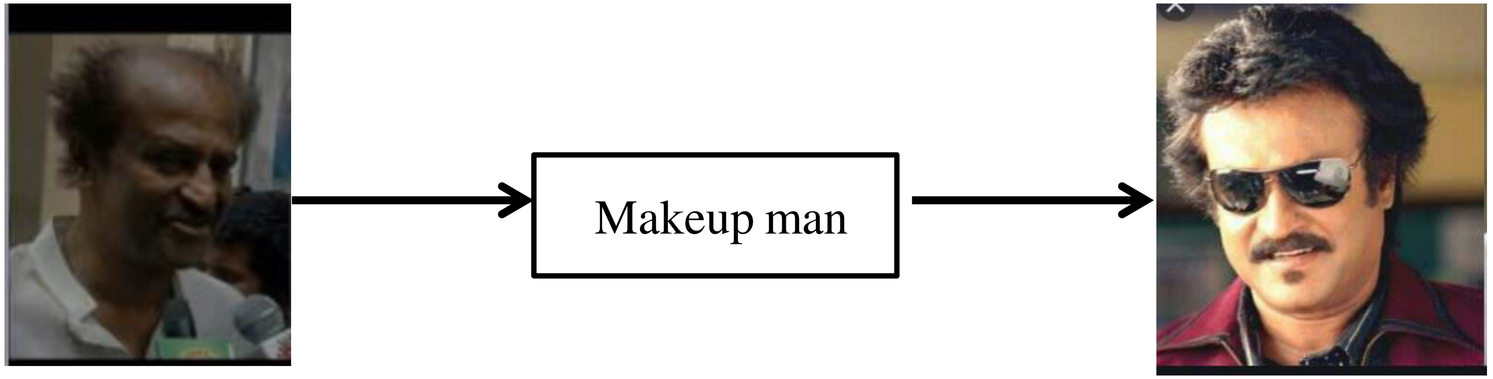
```
class Employee:
                def __init__(self,eno,ename,esal):
                                self.eno=eno
                                self.ename=ename
                                self.esal=esal
                def display(self):
                                print('Employee Number:',self.eno)
                                print('Employee Name:',self.ename)
                                print('Employee Salary:',self.esal)
class Test:
                def modify(emp):
                                emp.esal=emp.esal+10000
                                emp.display()
e=Employee(100,'Durga',10000)
Test.modify(e)
```

```
Employee Number: 100
Employee Name: Durga
Employee Salary: 20000
```

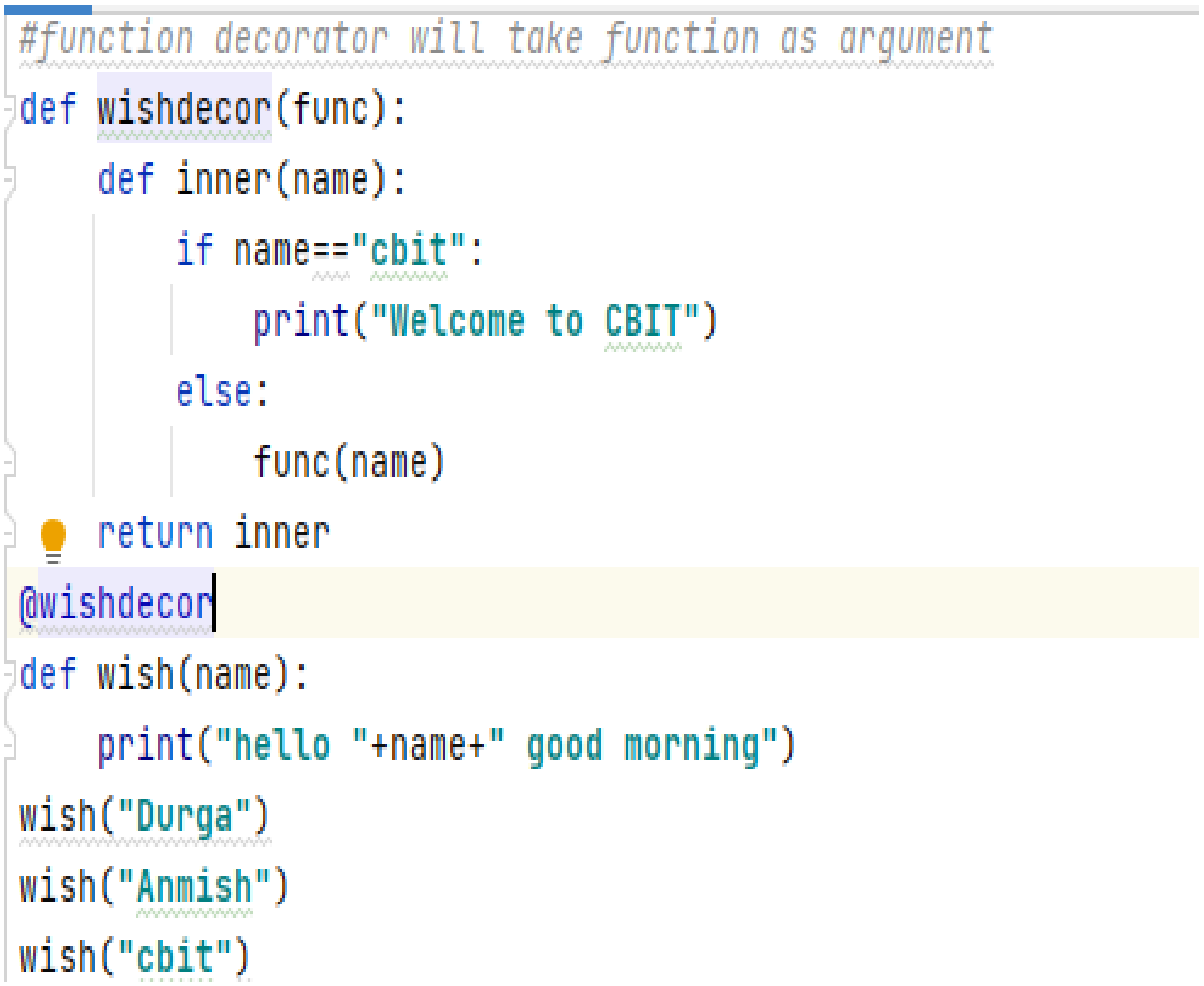
Function decorators

- Function decorator is a function which takes function as an argument and adds the extra functionality and returns the output function with extra added functionality.

$$\begin{tikzcd}\boxed{\text{Input function}} \arrow[r] & \boxed{\text{Decorator function}} \arrow[r, "\text{Adds new functionality}"] & \boxed{\text{output function}}\end{tikzcd}$$



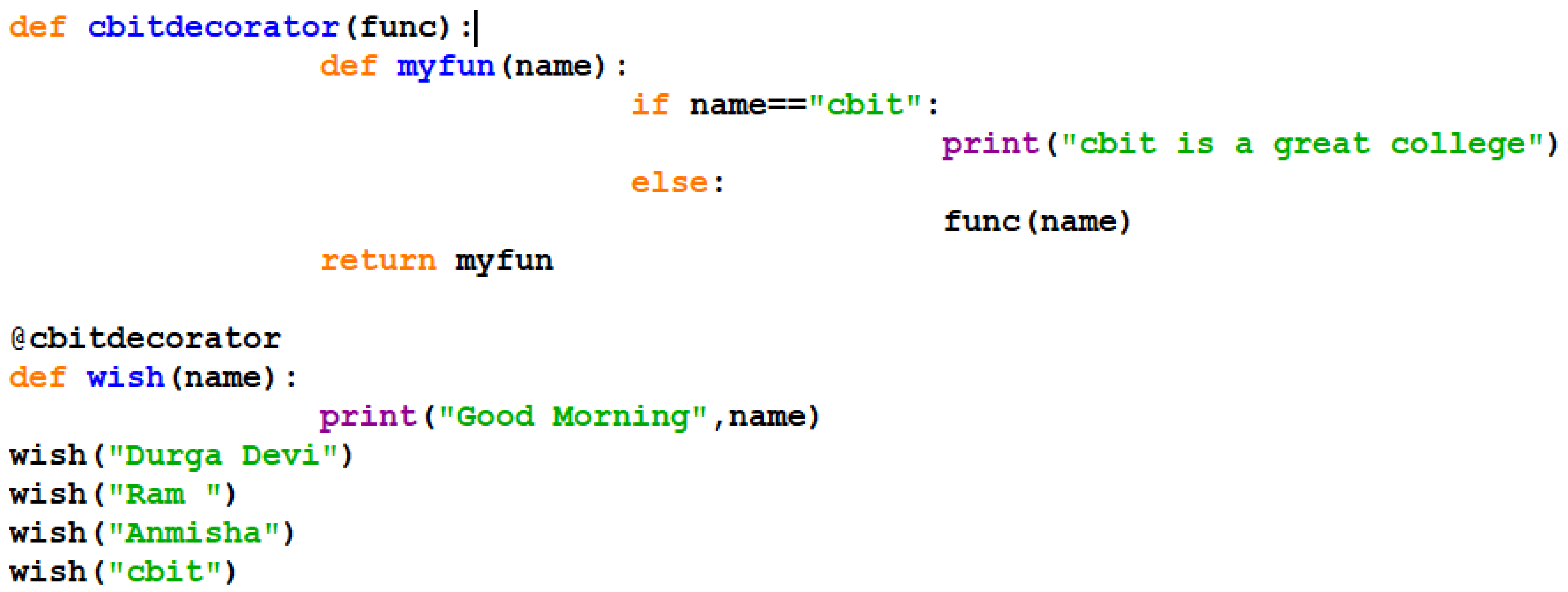
- Decorator function will not modify existing function adds the extra functionality and returns the new function.



```
#function decorator will take function as argument
def wishdecor(func):
    def inner(name):
        if name=="cbit":
            print("Welcome to CBIT")
        else:
            func(name)
    return inner
@wishdecor
def wish(name):
    print("hello "+name+" good morning")
wish("Durga")
wish("Anmish")
wish("cbit")
```

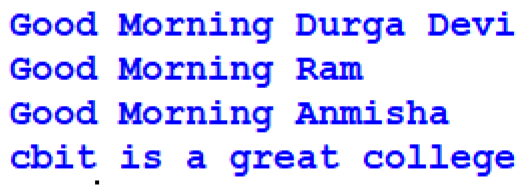
Function decorator

Example



```
def cbitdecorator(func):
        def myfun(name):
                if name=="cbit":
                        print("cbit is a great college")
                else:
                        func(name)
        return myfun

@cbitdecorator
def wish(name):
        print("Good Morning",name)
wish("Durga Devi")
wish("Ram ")
wish("Anmisha")
wish("cbit")
```



```
Good Morning Durga Devi
Good Morning Ram
Good Morning Anmisha
cbit is a great college
```

Types of methods

1. Instance method(object related method)
2. Class method
3. Static method

1. Instance method: if method takes self as first argument is called instance method and uses the instance variables.

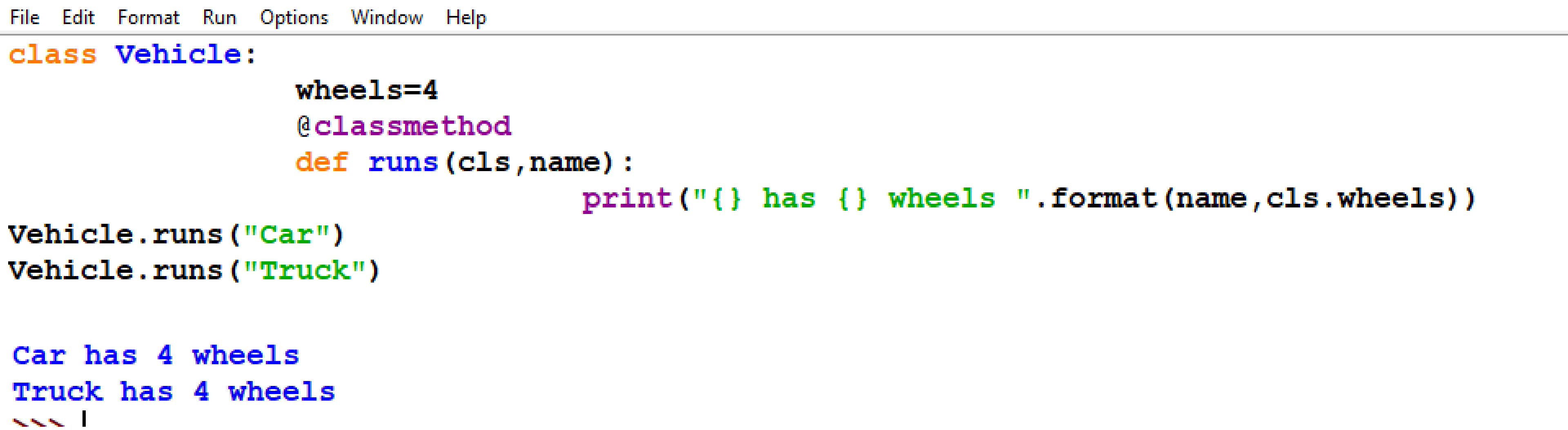
- def functionname(self):

- Instance methods are called with self inside a class and outside the class instance methods are called with object reference variable.

```
class Test:
            def details(self):
                        self.x=20
                        self.y=30
                        print(self.x,self.y)
            def display(self):
                        self.details()
t=Test()
t.display()# 20 30
t.details()#20 30
```

2. Class methods

- Class methods: if a method uses only class variables such methods are declared as class methods.
- Used to get class level data.
- For every class python virtual machine creates one reference variable to maintain class level data.
- Class methods are declared as explicitly using @classmethod decorator.
- For class methods keyword cls is to be used as first argument rather than self.
- Class methods to be called by using class name or object reference variable.

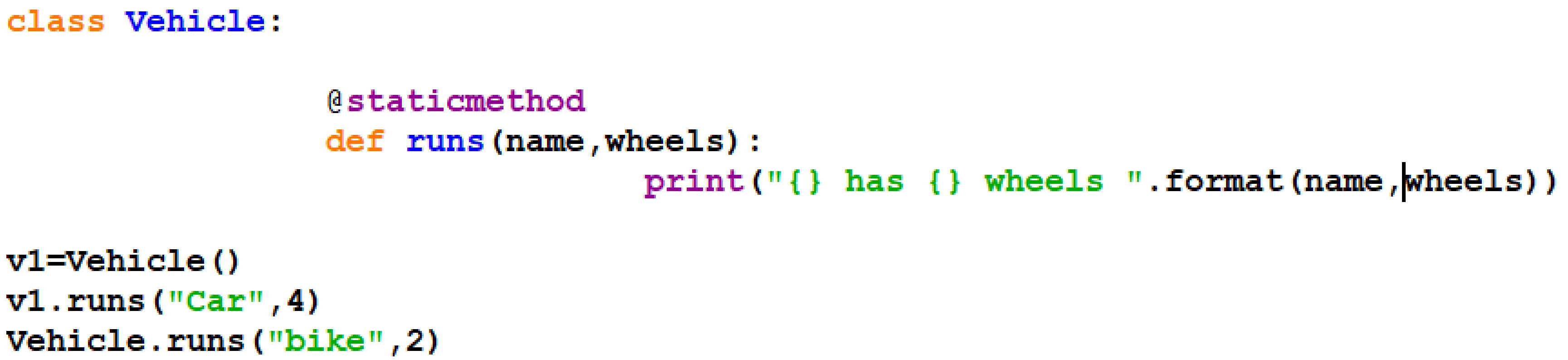


```
class Vehicle:
                wheels=4
                @classmethod
                def runs(cls,name):
                                print("{} has {} wheels ".format(name,cls.wheels))
Vehicle.runs("Car")
Vehicle.runs("Truck")
```

```
Car has 4 wheels
Truck has 4 wheels
>>> 
```

3. Static method

- Static method: the method does not use any instance or class variables is called static method.
- Used like a utility function which does not access properties of a class still a part of the class.
- Static methods does not use any argument of type such as self and cls.
- Static methods explicitly defined as @staticmethod.
- Static methods are defined like a normal function.
- Static methods called with class name or reference variable.



```
class Vehicle:

                @staticmethod
                def runs(name,wheels):
                                print("{} has {} wheels ".format(name,wheels))

v1=Vehicle()
v1.runs("Car",4)
Vehicle.runs("bike",2)
```

- Built in class attributes
- Garbage collector

Built-in class attributes

- Every python class has a some built in attributes.
- These built in attributes used as classname.__attributename.

1. **__doc__:** used to get class documentation string if defined or if not defined returns None
2. **__dict__:** list of identifiers(namespace) used in a class.
3. **__name__:** name of the class
4. **__module__:** name of module in which class defined
5. **__base__:** list of base classes in form of tuple.

```
class Student:
                '''This class is for storing student details'''
                studentcount=0
                def __init__(self,name,rollno):
                                self.name=name
                                self.rollno=rollno
                def displaycount(self):
                                print("no of students=",Student.studentcount)
                def details_student(self):
                                print(self.name,self.rollno)
print("student.__doc__:  ", Student.__doc__)
print("student.__module__:  ", Student.__module__)
print("student.__name__:  ", Student.__name__)
print("student.__bases__:  ", Student.__bases__)
print("student.__dict__ :",Student.__dict__)
```

```
student.__doc__:   This class is for storing student details
student.__module__:   __main__
student.__name__:   Student
student.__bases__:   (<class 'object'>,)
student.__dict__ : {'__module__': '__main__', '__doc__': 'This class is for stor
ing student details', 'studentcount': 0, '__init__': <function Student.__init__
at 0x038DA300>, 'displaycount': <function Student.displaycount at 0x03B4B858>, '
details_student': <function Student.details_student at 0x03B4B8A0>, '__dict__':
<attribute '__dict__' of 'Student' objects>, '__weakref__': <attribute '__weakre
f__' of 'Student' objects>}
```

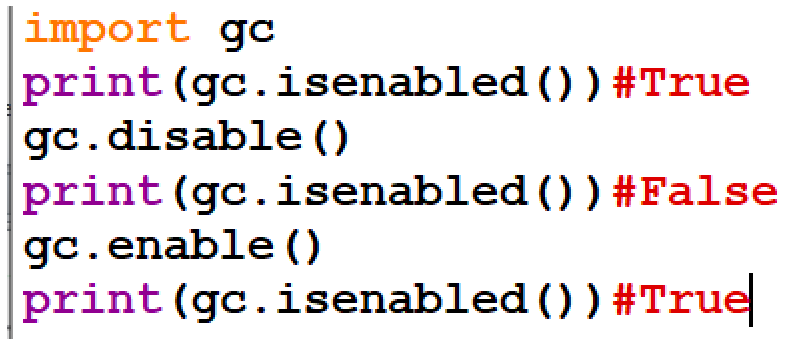
Garbage collection

- Programming languages like C++ , programmer is responsible to create and destroy the objects in a program. If programmer neglect to destroy unused objects, total memory filled with unused objects leads to memory problems and application goes down due to out of memory error.
- In python and java, programmer is only responsible to create objects. Python has program which takes care of destroying unused objects in the program. That program is called garbage collector.
- Main purpose of garbage collector is to destroy unused objects.
- The process of collecting unused objects in a program is called garbage collection.
- If object does not have any reference variable is collected by the garbage collector.

How to enable and disable garbage collector

- By default garbage collector is enabled for every program, we can also disable garbage collector explicitly
- We can use module named gc

1. gc.isenabled() : returns True if garbage collector is enabled.
2. gc.disable(): disable garbage collector explicitly.
3. gc.enable(): enable garbage collector explicitly.



```
import gc
print(gc.isenabled())#True
gc.disable()
print(gc.isenabled())#False
gc.enable()
print(gc.isenabled())#True
```

- Before destroying objects garbage collector calls destructors to perform clean up activities like resource deallocation, closing data base connections.
- Destructor is a special method and its name should be __del__

- Example

```
class Test:
        def __init__(self):
                print("object is initialized")
        def __del__(self):
                print("object is deallocated")
t1=Test()
t1=None
print("garbage collector called destructor")
```

```
object is initialized
object is deallocated
garbage collector called destructor
```

Question bank

- How do you initialize objects? Explain with an example
- What is the importance of self variable?
- What are the differences between constructor and method?
- What are the types of the variables ?
- Where do you use instance variables ? Give example for each.
- Can we add instance variables to object outside of the class? Yes
- Differentiate between instance variables and static variable.
- Compare two date formats means date specified with '/' or with'-' using static method.
- What is a function decorator? What is the use of it?
- How do you clean unused objects in python? Explain with one example
- Identify built in class attributes of class?