Object Oriented Programming in Python3

Palani Karthikeyan

karthikeyan@krosum.com

Krosum Labs

Introduction to OOPs in Python

- Course Introduction
- Python OOPs – pillars
- Class
- Object
- Method
  - Constructor
  - Non-Constructor
- Inheritance & Types
  - MRO
- Decorator
  - @Classmethod
  - @staticmethod

Krosum Labs

Introduction to OOPs in Python

- Python is a multi-paradigm programming language.
- Meaning, it supports different programming approach.
- The concept of OOP in Python focuses on creating reusable code.

- Class
- Object
- Method
- Inheritance

Krosum Labs

class

- Classes are used to create new user-defined data structures that contain arbitrary information about object.
- We can think class is a blueprint of the object.
- Syntax about class

class classname:

members

class is a keyword, class name is user defined.

Example1

- >>> class box:
  ... pass
  ...
- >>> box
- <class '__main__.box'>
- >>>
- >>> type(box)
- <class 'type'>
- >>>
- Empty class name is called **box**.

Krosum Labs

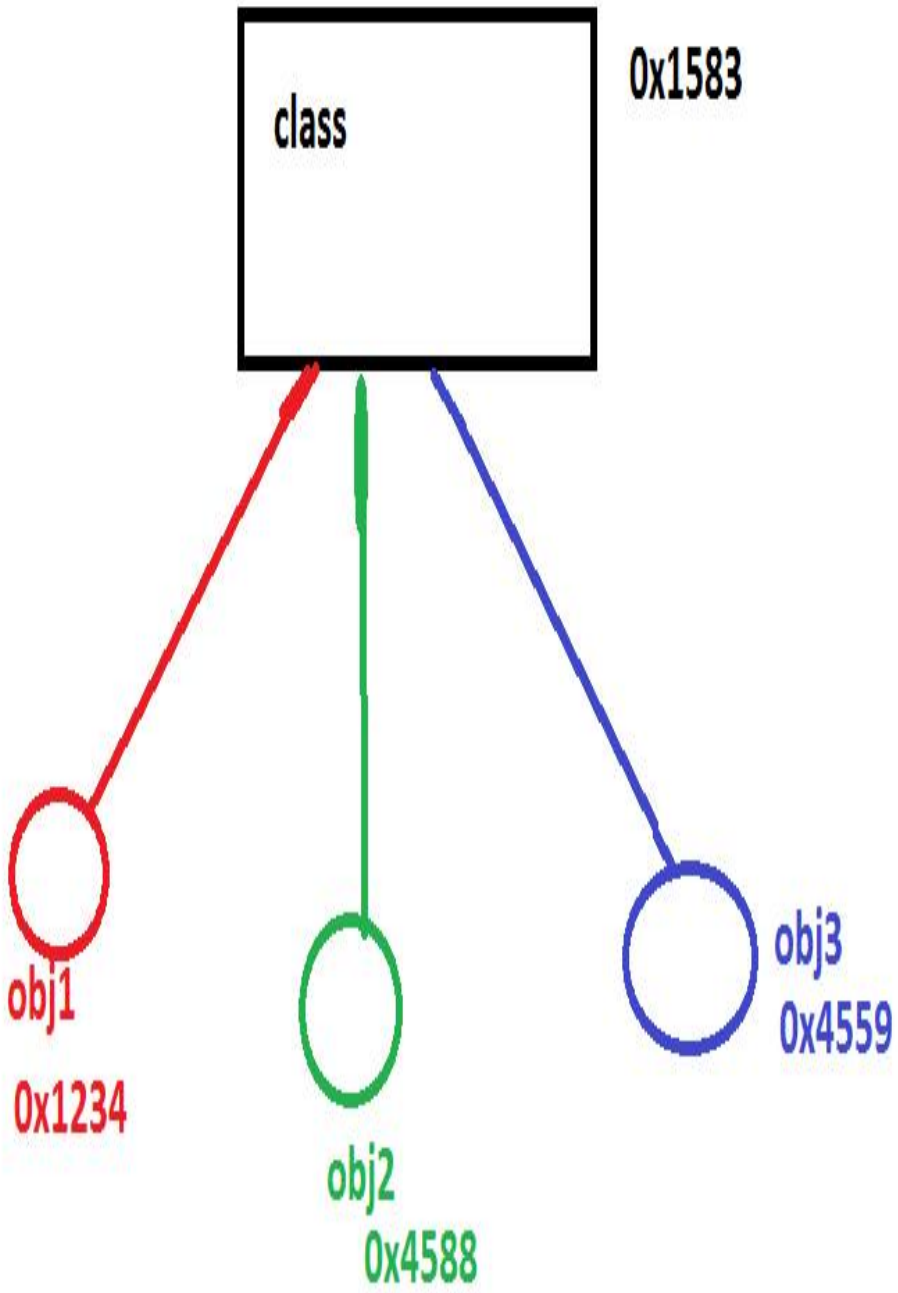
Object

- While the class is the blueprint, an *instance* is a copy of the class with *actual* values, literally an object belonging to a specific class.
- An object (instance) is an instantiation of a class.
- From single class we can create more than one object.

Syntax to create an object in python

object=classname() - constructor

```
>>> class   box:
...             pass
>>> box
<class '__main__.box'>
>>>
>>> box()
<__main__.box object at 0x004FB0D0>
>>>
>>> box()
<__main__.box object at 0x004FB130>
>>> box()
<__main__.box object at 0x004FB170>
>>> obj1=box()
>>> obj2=box()
>>> obj3=box()
```



Example 1

```
>>> class  Box:
...             pass
...
>>> obj1=Box()
>>> Box
<class '__main__.Box'>
>>> obj1
<__main__.Box object at 0x004FB250>
>>> type(Box)
<class 'type'>
>>> type(obj1)
<class '__main__.Box'>
```

```
>>> # procedure style
>>> v1=10
>>> v1
10
>>> # Object Oriented Style
>>> class   Box:
...             v2=100 # class member
...
>>> v2    # we can't access class member directly
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'v2' is not defined
>>>
>>> Box.v2      # using class name- we can access class member
100
>>> obj=Box()  # instantiation of a class
>>> obj.v2      # using class instance we can access class member
100
>>>  v1 # v1 is not a class member – we can access class member
10
```

```
>>> __port=80   #  ordinary variable
>>>
>>> __port
80
>>> def   f1():
...     print(__port)
...
>>> f1()
80
>>> class    Box:
...               var=100 # public variable - we can access outside class using classname
...               __v2=6550 # user defined private variable - we can't access this variable outside th e class
...
>>> Box.var
100
>>> Box.__v2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: type object 'Box' has no attribute '__v2'
>>> obj=Box()
>>> obj.var
100
>>> obj.__v2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Box' object has no attribute '__v2'
```

```
>>> class  Box:
...         server = "Default-Server"
>>> Box.server
'Default-Server'
>>> # using class name we can modify class variable
>>> Box.server="10.20.30.40"
>>> Box.server
'10.20.30.40'
>>> Box.user="root"    # we can create new class  variable
>>> Box.user
'root'
>>> user # this is not a class member
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'user' is not defined
```

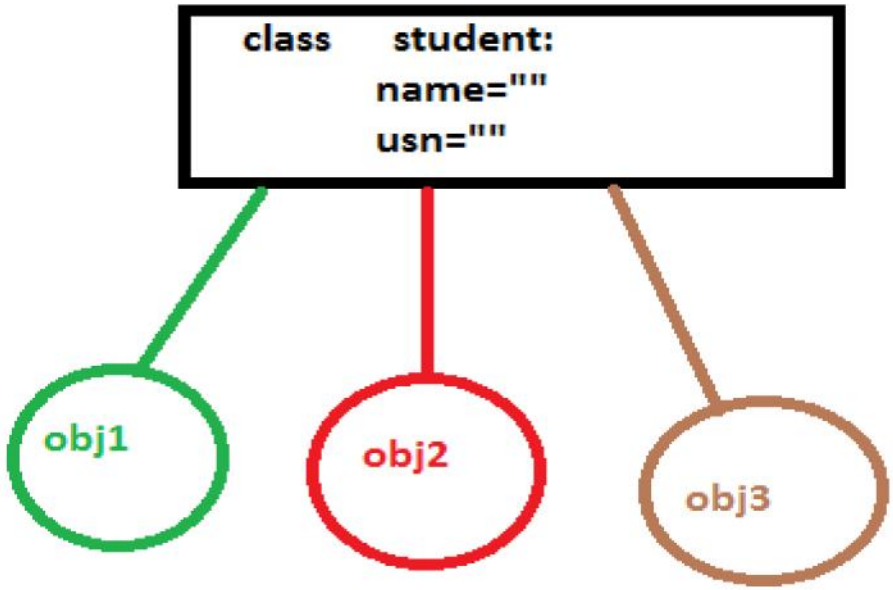
class vs object

```
>>> class  box:
...     server="Default-Server"
>>> obj1=box()
>>> obj2=box()
>>> box.server    # this is not private variable,so we can access  outside the class
'Default-Server'
>>> obj1.server
'Default-Server'
>>> obj2.server
'Default-Server'
```

using object we can assign unique value ,this value is **not overwrite** class and other object variable

```
>>> obj1.server="Unix“
>>> obj2.server="Linux"
>>> obj1.server
'Unix'
>>> obj2.server
'Linux'
>>> box.server
'Default-Server'
```

```
>>> class student:
...     name=""
...     usn=""
>>> obj1=student()
>>> obj2=student()
>>> obj3=student()
>>> obj1.name="Arun"
>>> obj1.usn="001"
>>> obj2.name="Vijay"
>>> obj2.usn="002"
>>> obj3.name="Kumar"
>>> obj3.usn="003"
print("Student1:{}\nStudent2:{}\nStudent3:{}".format(obj1.name,obj2.name,obj3.name))
Student1:Arun
Student2:Vijay
Student3:Kumar
>>> print("Student1:{}\nStudent2:{}\nStudent3:{}".format(obj1.usn,obj2.usn,obj3.usn))
Student1:001
Student2:002
Student3:003
>>>
```



methods

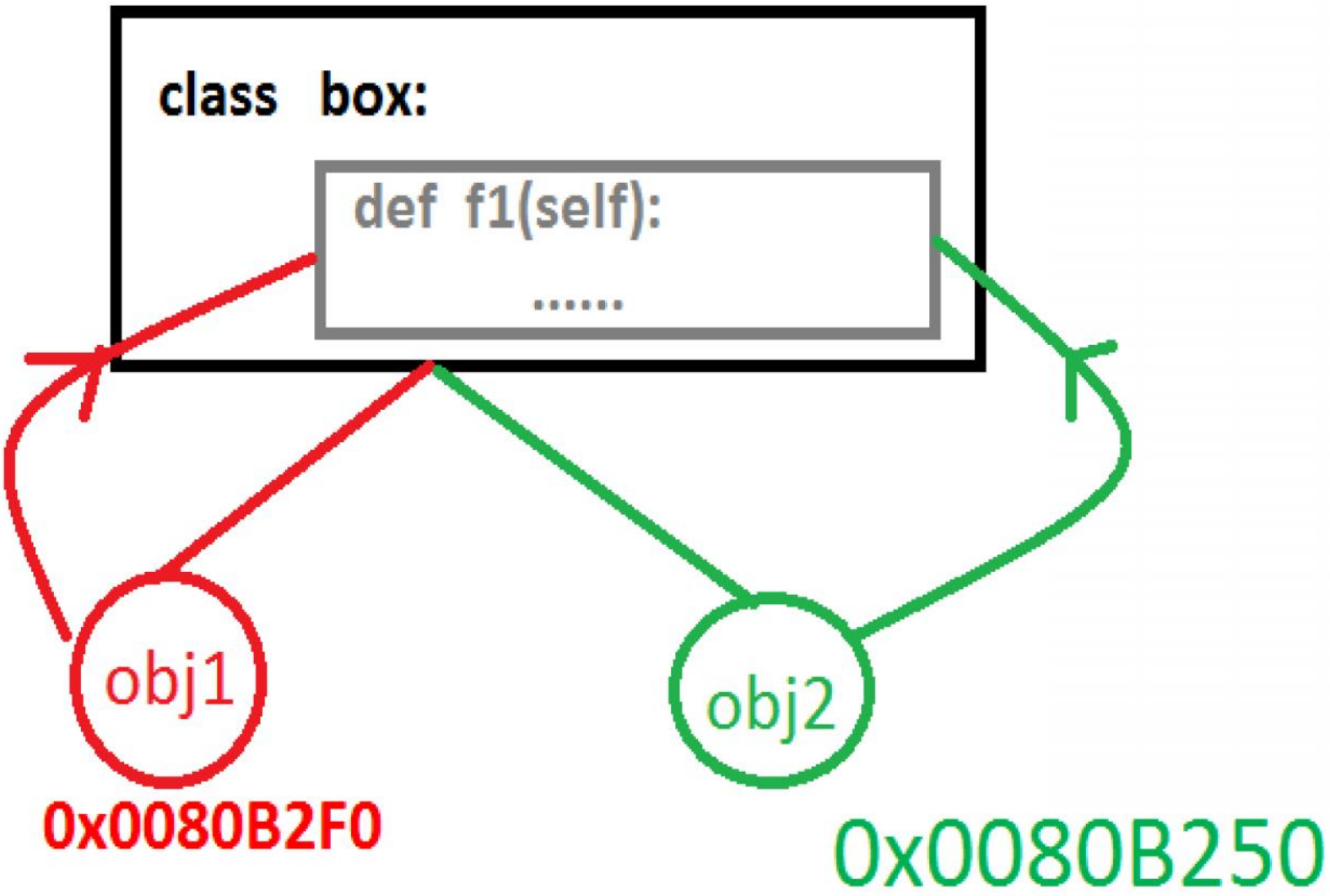
- Methods are functions defined inside the body of a class.
- They are used to define the behaviors of an object.

- >>> class box:
- ... def f1(self):
- ... print("Im f1 method")
- ... print("self value is:{}".format(self))
- ...
- >>> obj1=box() # object1
- >>> obj1
- <__main__.box object at **0x0080B2F0**>
- >>>
- >>> obj1.f1() # f1(obj1)
- Im f1 method
- self value is:<__main__.box object at **0x0080B2F0**>

- >>> **obj2**=box() # object2
- >>> obj2
- <__main__.box object at 0x0080B250>
- >>>
- >>> **obj2**.f1() # f1(obj2)
- Im f1 method
- self value is:<__main__.box object at **0x0080B250**>
- >>>

 # again object1 is calling f1() method

- ... obj1.f1()
- Im f1 method
- self value is:<__main__.box object at 0x0080B2F0>
- >>>

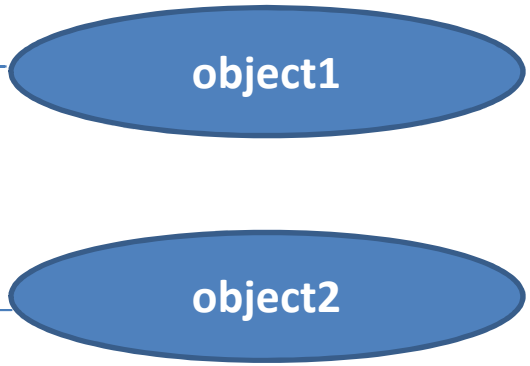

class box:
def f1(self):
.....
obj1
obj2
0x0080B2F0
0x0080B250

- In python class variables can not accessible directly in side the method.
- >>> class box:
- ... var=100
- ... def f1(self):
- ... print("Im f1 block")
- ... print(var)
- ...
- ...
- >>> obj=box()
- >>> obj.f1()
- Im f1 block
- Traceback (most recent call last):
- File "<stdin>", line 1, in <module>
- File "<stdin>", line 5, in f1
- NameError: name 'var' is not defined
- >>>

- Using self(object) we can access class variable

```
>>> class box:
...     var=100
...     def f1(self):
...             print("Im f1 block")
...             print(self.var)
...
>>> obj=box()
>>> obj.f1()
Im f1 block
100
>>>
```

```
>>> class student:
...     name=""
...     usn=""
...
>>> obj1=student()
>>> obj1.name="Arun"
>>> obj1.usn="001"
>>>
>>> obj2=student()
>>> obj2.name="Vijay"
>>> obj2.usn="002"
>>>
>>> print("""
... student name:{}\tUSN:{}
... ------------------------
... student name:{}\tUSN:{}
... ------------------------""".format(obj1.name,obj1.usn,obj2.name,obj2.usn))

student name:Arun       USN:001
------------------------
student name:Vijay      USN:002
------------------------
>>>
```



using method

```
>>> class student:
...     name=""
...     usn=""
...     def f1(self,a1,a2):
...             self.name=a1
...             self.usn=a2
...     def f2(self):
...             print("Student Name:{}\nUSN:{}".format(self.name,self.usn))
...
>>> obj1=student()
>>> obj1.f1("Arun","001")  # Like obj1.name="Arun"; obj2.usn="001"
>>>
>>> obj2=student()
>>> obj2.f1("Vijay","002") # Like obj2.name="Vijay"; obj2.usn="002"
>>>
>>> obj1.f2()
Student Name:Arun
USN:001
>>>
>>> obj2.f2()
Student Name:Vijay
USN:002
>>>
```

using method

```
>>> class student:
...     name=""
...     usn=""
...     def f1(self,a1,a2):
...             self.name=a1
...             self.usn=a2
...     def f2(self):
...             print("Student Name:{}\nUSN:{}".format(self.name,self.usn))
...
>>> obj1=student()
>>> obj1.f1("Arun","001")  # Like obj1.name="Arun"; obj2.usn="001"
>>>
>>> obj2=student()
>>> obj2.f1("Vijay","002") # Like obj2.name="Vijay"; obj2.usn="002"
>>>
>>> obj1.f2()
Student Name:Arun
USN:001
>>>
>>> obj2.f2()
Student Name:Vijay
USN:002
>>>
```

using method

```
>>> class student:
...     name=""
...     usn=""
...     def f1(self,a1,a2):
...             self.name=a1
...             self.usn=a2
...     def f2(self):
...             print("Student Name:{}\nUSN:{}".format(self.name,self.usn))
...
>>> obj1=student()
>>> obj1.f1("Arun","001")  # Like obj1.name="Arun"; obj2.usn="001"
>>>
>>> obj2=student()
>>> obj2.f1("Vijay","002") # Like obj2.name="Vijay"; obj2.usn="002"
>>>
>>> obj1.f2()
Student Name:Arun
USN:001
>>>
>>> obj2.f2()
Student Name:Vijay
USN:002
>>>
```

__variable

```
>>>class box:
        __var=100  # private variable


>>> box.var
Attribute Error

>>> obj=box()
>>> obj.var
Attribute Error
```

```
>>> class box:
...     __var=100
...     def f1(self):
...         print("Var value:{}".format(self.__var))
...
>>> obj=box()
>>> obj.f1()
Var value:100
```

```
>>> class    box:
...     __var=100
...     def f1(self):
...         print("Var value:{}".format(self.__var))
...
>>> obj=box()
>>> obj.f1()
Var value:100
```

```
>>> obj.__var
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'box' object has no attribute '__var'
>>>
>>> box.__var
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: type object 'box' has no attribute '__var'
>>>
```

```
>>> class student:
...     __name=""
...     __usn=""
...     def f1(self,a1,a2):
...             self.__name=a1
...             self.__usn=a2
...     def f2(self):
...             print("Student name:{}".format(self.__name))
...             print("Student USN:{}".format(self.__usn))
...
>>> obj1=student()
>>> obj1.f1("Arun","001")
>>>
>>> obj2=student()
>>> obj2.f1("Vijay","002")
>>>
>>> obj1.f2()
Student name:Arun
Student USN:001
>>>
>>> obj2.f2()
Student name:Vijay
Student USN:002
>>>
```

__init__method

- __init__(self)
- "__init__" is a reserved method in python classes. It is called as a constructor in object oriented terminology.
- This method is called when an object is created from a class and it allows the class to initialize the attributes of the class.
- >>> class box:
- ... def **__init__(self):**
- ... print("Im initialized block")
- ...
- >>> **box()**
- Im initialized block
- <__main__.box object at 0x0080B610>
- >>>
- **>>> obj=box()**
- Im initialized block
- >>>

constructor vs non-constructor

- >>> class box:
- ... def __init__(self):
- ... print("Im constructor")
- ...
- >>> obj=box()
- Im constructor

- >>> class box:
- ... def f1(self):
- ... print("Im non-constructor")
- ...
- >>> obj=box()
- >>> **obj.f1()**
- Im non-constructor

Passing arguments

```
>>> class box:
...     def __init__(self,a1,a2,a3):
...             print(a1)
...             print(a2)
...             print(a3)
...
>>> obj=box("root","Welcome",80)
root
Welcome
80
>>> obj=box(100,["D1","D2","D3","D4"],("T1","T2"))
100
['D1', 'D2', 'D3', 'D4']
('T1', 'T2')
>>> obj=box("data",{"K1":"V1","K2":"V2"},10.3455)
data
{'K1': 'V1', 'K2': 'V2'}
10.3455
>>>
```

Using __init__()

```
>>> class student:
...     __name=""
...     __usn=""
...     def __init__(self,a1,a2):
...             self.__name=a1
...             self.__usn=a2
...     def display(self):
...             print("Student name:{}".format(self.__name))
...             print("Student USN:{}".format(self.__usn))
...
>>> obj1=student("Arun","001")
>>> obj2=student("Vijay","002")
>>>
>>> obj1.display()
Student name:Arun
Student USN:001
>>> obj2.display()
Student name:Vijay
Student USN:002
```

```
>>> class student:
...     __name=""
...     __usn=""
...     __dept=""
...     def __init__(self,a1,a2,a3):
...             self.__name=a1
...             self.__usn=a2
...             self.__dept=a3
...     def display(self):
...             print("Student name:{}".format(self.__name))
...             print("Student USN:{}".format(self.__usn))
...             print("Student Dept:{}".format(self.__dept))
...     def update(self,a1):
...             self.__dept=a1 #updating dept value
...
>>> obj=student("Arun","001","CSE")
>>> obj.display()
Student name:Arun
Student USN:001
Student Dept:CSE
>>>
```

After updating **dept** value

```
>>> class student:
...     __name=""
...     __usn=""
...     __dept=""
...     def __init__(self,a1,a2,a3):
...             self.__name=a1
...             self.__usn=a2
...             self.__dept=a3
...     def display(self):
...             print("Student name:{}".format(self.__name))
...             print("Student USN:{}".format(self.__usn))
...             print("Student Dept:{}".format(self.__dept))
...     def update(self,a1):
...             self.__dept=a1 #updating dept value
...
>>> obj=student("Arun","001","CSE")
>>> obj.display()
Student name:Arun
Student USN:001
Student Dept:CSE
>>>
```

```
>>> obj=student("Arun","001","CSE")
>>> obj.display()
Student name:Arun
Student USN:001
Student Dept:CSE
>>> obj.update("MECH")
>>> obj.display()
Student name:Arun
Student USN:001
Student Dept:MECH
```

python

Krosum Labs

Inheritance

- Inheritance is the capability of one class to derive or inherit the properties from some another class.
- The benefits of inheritance is **reusability.**
- We don’t have to write the same code again and again. Also, it allows us to add more features to a class without modifying it.
- It refers to defining a new class with little or no modification to an existing class.
- The new class is called **derived (or child) class** and the one from which it inherits is called the **base (or parent) class**.

Python Inheritance Syntax

```
class  BaseClass:
        Body of base class class

DerivedClass(BaseClass):
        Body of derived class
```

- Derived class inherits features from the base class, adding new features to it. This results into re-usability of code.

Example

```
>>> class P1:
...         def f1(self):
...                 print("Im from f1 block from P1 class")
...
>>> class P2(P1):
...             def f2(self):
...                     print("Im from f2 block from P2 class")
...
>>>
>>> obj=P2()
>>> obj.f1() # calling parent method
Im from f1 block from P1 class
>>>
>>> obj.f2()
Im from f2 block from P2 class
>>>
```

Example

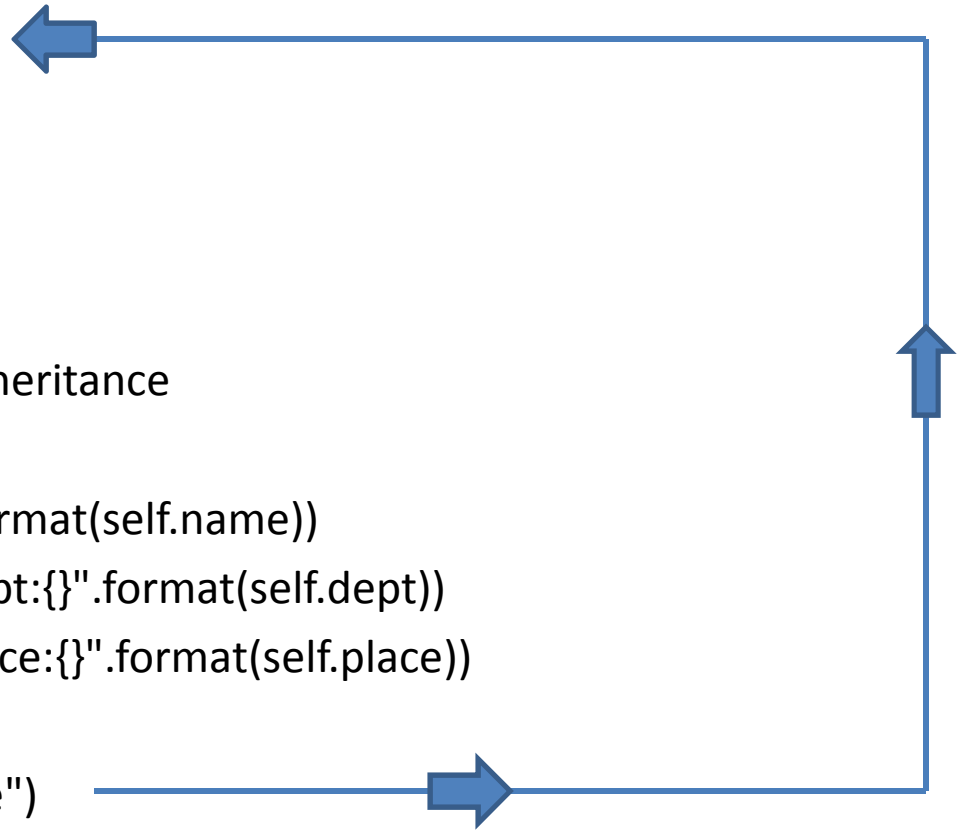
```
class  Enrollment:
        name=""
        dept=""
        place=""
        def f1(self,a1,a2,a3):
            self.name=a1
            self.dept=a2
            self.place=a3

class  Person(Enrollment):   # single inheritance
            def f2(self):
                print("Emp name:{}".format(self.name))
                print("Emp working dept:{}".format(self.dept))
                print("Emp working place:{}".format(self.place))

obj=Person()
obj.f1("Arun","sales","Bangalore")
obj.f2()
```

Example

```
class  Enrollment:
        name=""
        dept=""
        place=""
        def f1(self,a1,a2,a3):
            self.name=a1
            self.dept=a2
            self.place=a3

class  Person(Enrollment):   # single inheritance
            def f2(self):
                print("Emp name:{}".format(self.name))
                print("Emp working dept:{}".format(self.dept))
                print("Emp working place:{}".format(self.place))

obj=Person()
obj.f1("Arun","sales","Bangalore")
obj.f2()
```

Example

```
class   Enrollment:
            name=""
            dept=""
            place=""
            def  __init__(self,a1,a2,a3):
                    self.name=a1
                    self.dept=a2
                    self.place=a3

class   Person(Enrollment): # single inheritance
              def f2(self):
                    print("Emp name:{}".format(self.name))
                    print("Emp working dept:{}".format(self.dept))
                    print("Emp working place:{}".format(self.place))

obj=Person("Arun","sales","Bangalore")
obj.f2()
```



Method Overriding in Python

- In Inheritance parent and child class contains same method name, the method in the derived class (child) overrides that in the base class (parent).
- class P1:

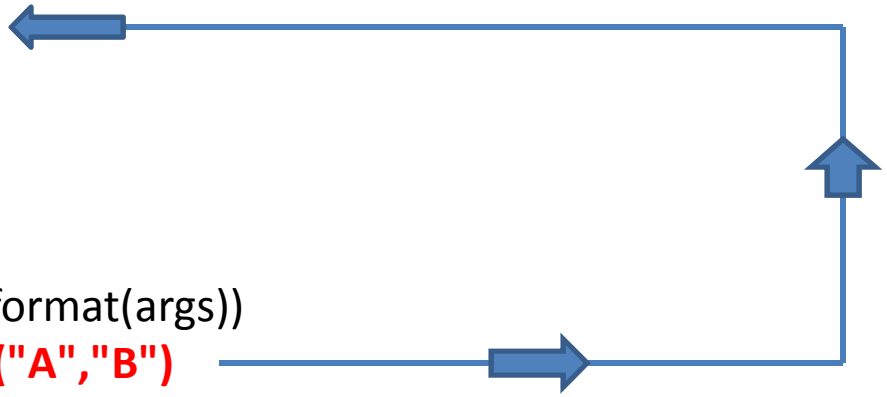
```
    def f1("Hello")
class P2(P1):
    def f1("Welcome")
Obj=P2()
Obj.f1()  → "Welcome"
Obj.f1()  → "Welcome"
```

Super()

- To avoid overriding we can use super() method.
- >>> class P1:
- ... def f1(self):
- ... print("Hello")
- ...
- >>> class P2(P1):
- ... def f1(self):
- ... print("Welcome")
- ... **super(P2,self).f1() # calling parent f1() method**
- ...
- >>> obj=P2()
- >>> obj.f1()
- Welcome
- **Hello**
- >>>

Super()

```
>>> class   A:
...     def   f1(self,*args):
...             print(args)

>>> class  B(A):
...     def   f1(self,*args):
...             print("Child:{}".format(args))
...             super(B,self).f1("A","B")
...             print("Child:{}".format(args))
...
>>> obj=B()
>>> obj.f1(10,20,30,40)
Child:(10, 20, 30, 40)
('A', 'B')
Child:(10, 20, 30, 40)
>>>
```



isinstance() and issubclass()

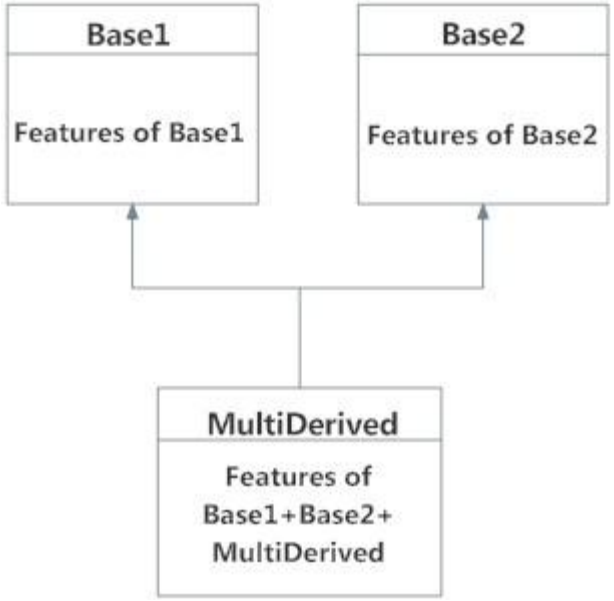
- Two in built
  functions isinstance() and issubclass() are used to check inheritances.
- Function isinstance() returns True if the object is an instance of the class or other classes derived from it.
- Each and every class in Python inherits from the base class object.

```
>>> isinstance(obj,P1)
True
>>> isinstance(obj,P2)
True
>>> issubclass(P2,P1)
True
>>> issubclass(P1,P2)
False
>>>
```

Python Multiple Inheritance

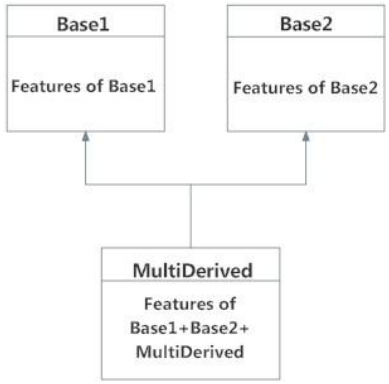
- In multiple inheritance, the features of all the base classes are inherited into the derived class.
- The syntax for multiple inheritance is similar to single inheritance.

```
class Base1:
        pass
class Base2:
        pass
class MultiDerived(Base1, Base2):
        pass
```



Python Multiple Inheritance

```
class Fsinfo:
        fstype="ext"
        user="root"

class Sysinfo:
        kernel="Linux"
        Version=2.6

class  info (Fsinfo,Sysinfo):
        pass

obj=info()
print("Fstype:{}\nUser:{}".format(obj.fstype,obj.user))
print("working kernel name:{}\tVersion:{}".format(obj.kernel,obj.Version))
```

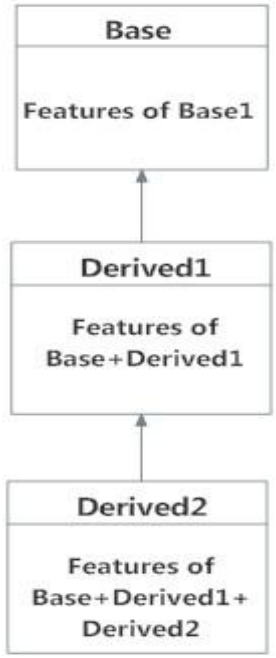



```
C:\Users\User>python  multiple.py
Fstype:ext
User:root
working kernel name:Linux       Version:2.6
```

Multilevel Inheritance in Python

- we can also inherit form a derived class. This is called multilevel inheritance.
- It can be of any depth in Python.
- In multilevel inheritance, features of the base class and the derived class is inherited into the new derived class.

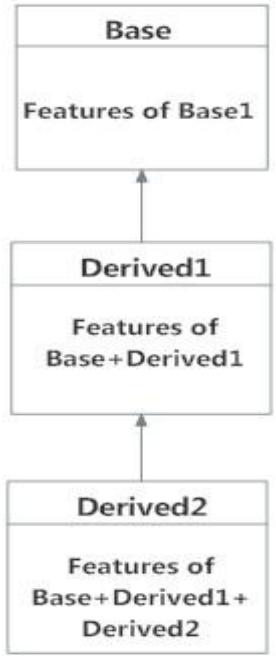
```
class Base:
        pass
class Derived1(Base):
        pass
class Derived2(Derived1):
          pass
```



Multilevel Inheritance in Python

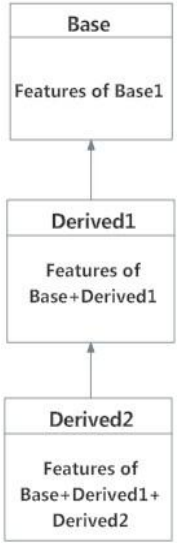
- we can also inherit form a derived class. This is called multilevel inheritance.
- It can be of any depth in Python.
- In multilevel inheritance, features of the base class and the derived class is inherited into the new derived class.

```
class Base:
        pass
class Derived1(Base):
        pass
class Derived2(Derived1):
          pass
```



Multilevel Inheritance in Python

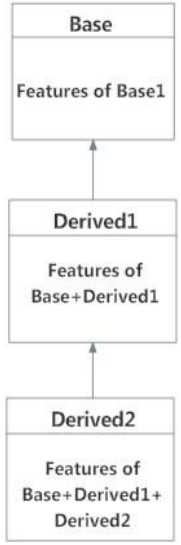
```
class    Version1:
         product_name="Test_App1"
         count=0
         def f1(self):
                  print("Product_Name:{}".format(self.product_name))
                  self.count=self.count+1
class    Version2(Version1):
         COST=12345.67
         def f2(self):
                  print("Cost Value:{}".format(self.COST))
                  self.count=self.count+1
class    Version3(Version2):
         authors=["Mr.Vishnu","Mr.John"]
         def f3(self):
              print("Author details:-")
              print(self.authors)
obj= Version3()
obj.f1()
obj.f2()
obj.f3()
```




```
C:\Users\User> python ml.py
Product_Name:Test_App1
Cost Value:12345.67
Author details:-
['Mr.Vishnu', 'Mr.John']

C:\Users\User>
```

Multilevel Inheritance in Python

```
class Version1:
        product_name="Test_App1"
        count=0
        def f1(self):
                print("Product_Name:{}".format(self.product_name))
                self.count=self.count+1
class Version2(Version1):
        COST=12345.67
        def f2(self):
                print("Cost Value:{}".format(self.COST))
                self.count=self.count+1
class Version3(Version2):
        authors=["Mr.Vishnu","Mr.John"]
        def f3(self):
                print("Author details:-")
                print(self.authors)
```

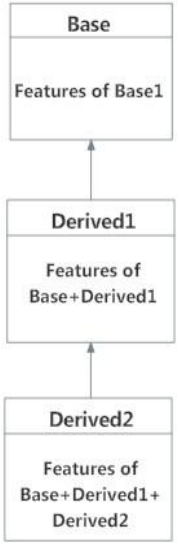



```
obj=Version3()
print("-----(1)----{}".format(obj.count))
obj.f1()
print("-----(2)----{}".format(obj.count))
obj.f2()
print("-----(3)----{}".format(obj.count))
obj.f3()
print("-----(4)----{}".format(obj.count))
```

```
C:\Users\User>python ml.py
-----(1)----0
Product_Name: Test_App1
-----(2)----1
Cost Value: 12345.67
-----(3)----2
Author details:-
['Mr.Vishnu', 'Mr.John']
-----(4)----2
```

Multilevel Inheritance in Python

```
class    Version1:
         product_name="Test_App1"
         count=0
         def f1(self):
                  print("Product_Name:{}".format(self.product_name))
                  self.count=self.count+1
class    Version2(Version1):
         COST=12345.67
         def f2(self):
                  print("Cost Value:{}".format(self.COST))
                  self.count=self.count+1
class    Version3(Version2):
         authors=["Mr.Vishnu","Mr.John"]
         def f3(self):
              print("Author details:-")
              print(self.authors)
obj= Version3()
obj.f1()
obj.f2()
obj.f3()
```

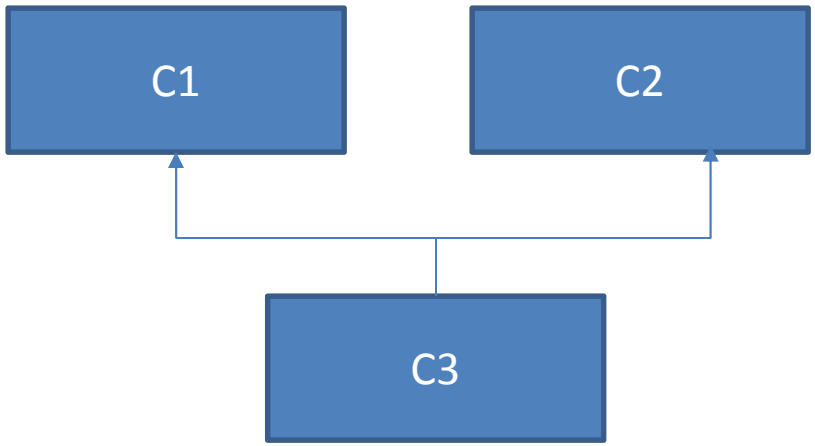


Method Resolution Order (MRO)

- Method Resolution Order (MRO) is the order in which Python looks for a method in a **hierarchy of classes**.
- Multiple inheritance
- single method may be found in multiple super classes.

Method Resolution Order (MRO)

```
>>> class C1:
...     def f1(self):
...             print("F1 block from C1 class")
...
>>> class C2:
...     pass
...
>>> class C3(C1,C2):
...     pass
...
>>>
>>> obj=C3()
>>> obj.f1()
F1 block from C1 class
>>>
>>> C3.mro()
[<class '__main__.C3'>, <class '__main__.C1'>, <class '__main__.C2'>, <class 'object'>]
>>>
```

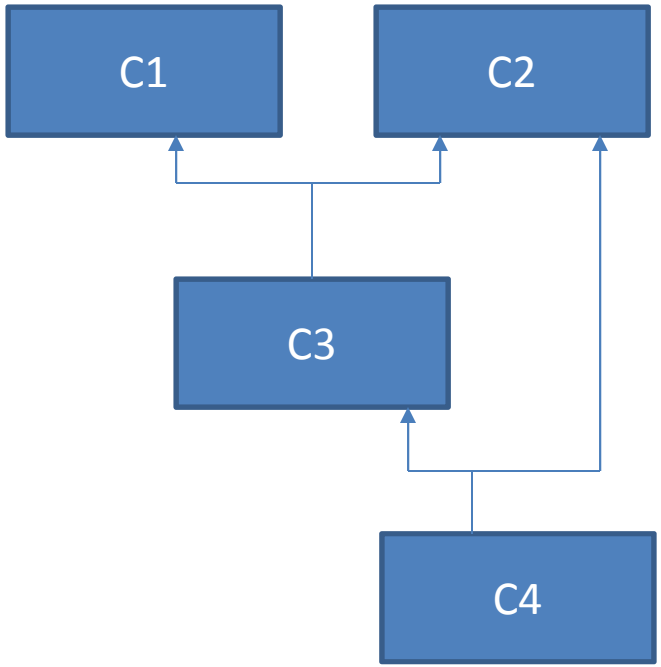


Method Resolution Order (MRO)

```
>>> class C1:
...     def f1(self):
...             print("F1 block-C1 class")
>>> class C2:
...     def f1(self):
...             print("F1 block-C2 class")
>>> class C3(C1,C2):
...     pass
>>> obj=C3()
>>> obj.f1()
F1 block-C1 class
>>>
>>> class C3(C2,C1):
...     pass
...
>>> obj=C3()
>>> obj.f1()
F1 block-C2 class
>>>
>>> C3.mro()
[<class '__main__.C3'>, <class '__main__.C2'>, <class '__main__.C1'>, <class 'object'>]
>>>
```

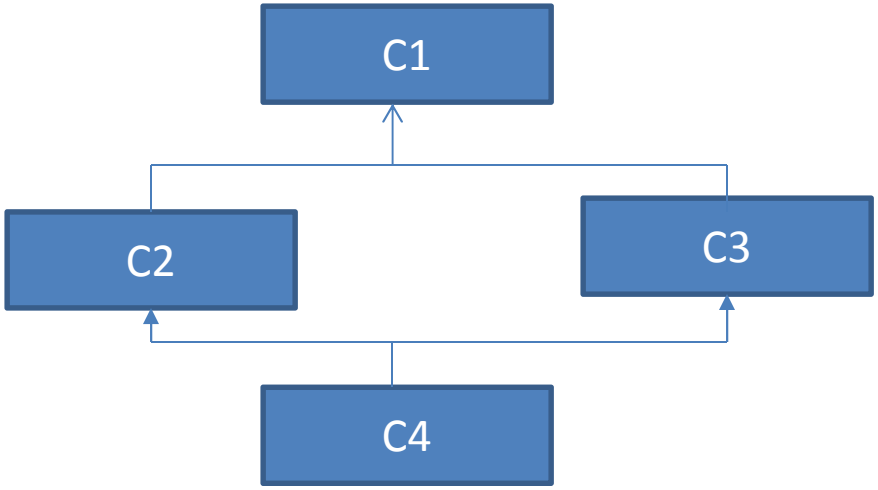
Method Resolution Order (MRO)

```
>>> class C1:
...     def f1(self):
...             print("F1-block from C1 class")
...
>>> class C2:
...     def f1(self):
...             print("F1-block from C2 class")
...
>>> class C3(C1,C2):
...     def f1(self):
...             print("F1-block from C3 class")
...
>>> class C4(C3,C2):
...     pass
...
>>> obj=C4()
>>> obj.f1()
F1-block from C3 class
>>> obj.f1()
F1-block from C3 class
>>> C4.mro()
[<class '__main__.C4'>, <class '__main__.C3'>, <class '__main__.C1'>, <class '__main__.C2'>, <class 'object'>]
```



Method Resolution Order (MRO)

```
>>> class C1:
...     def f1(self):
...             print("F1 Block-From C1 class")
...
>>> class C2(C1):
...     def f1(self):
...             print("F1 Block-From C2 class")
...
>>> class C3(C1):
...     def f1(self):
...             print("F1 Block-From C3 class")
...
>>> class C4(C2,C3):
...     pass
...
>>> obj=C4()
>>> obj.f1()
F1 Block-From C2 class

>>> C4.mro()
[<class '__main__.C4'>, <class '__main__.C2'>, <class '__main__.C3'>, <class '__main__.C1'>, <class 'object'>]
>>>
```



Method Resolution Order (MRO)

- However, that is contradictory to rule of inheritance, as most specific version must be taken first and then least specific (generic) version.

```
>>> class C3(C2,C1):
...     pass
...
>>> C3.mro()
[<class '__main__.C3'>, <class '__main__.C2'>,
<class '__main__.C1'>, <class 'object'>]
```

```
>>> class C1:
...     pass
>>> class C2(C1):
...     pass
>>> class C3(C1,C2):
...     pass
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Cannot create a consistent method resolution
order (MRO) for bases C1, C2
>>>
```