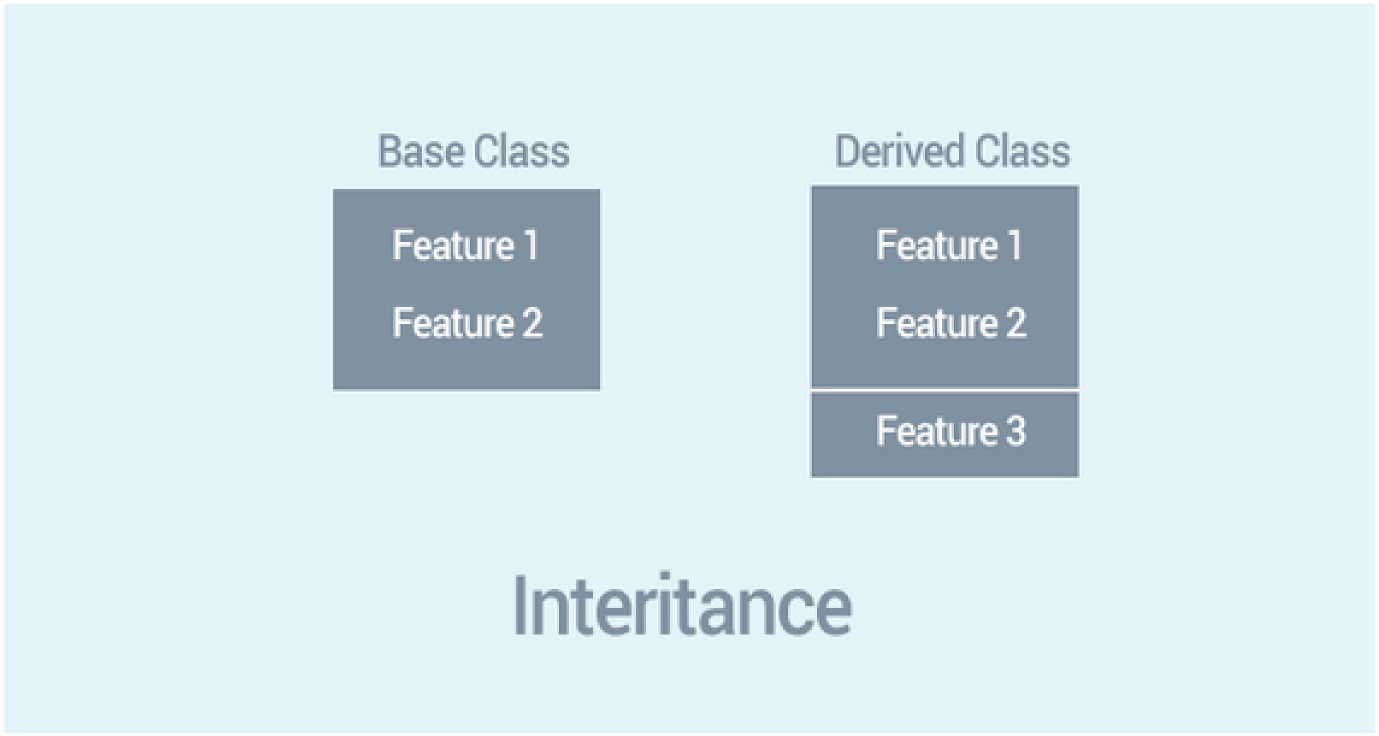
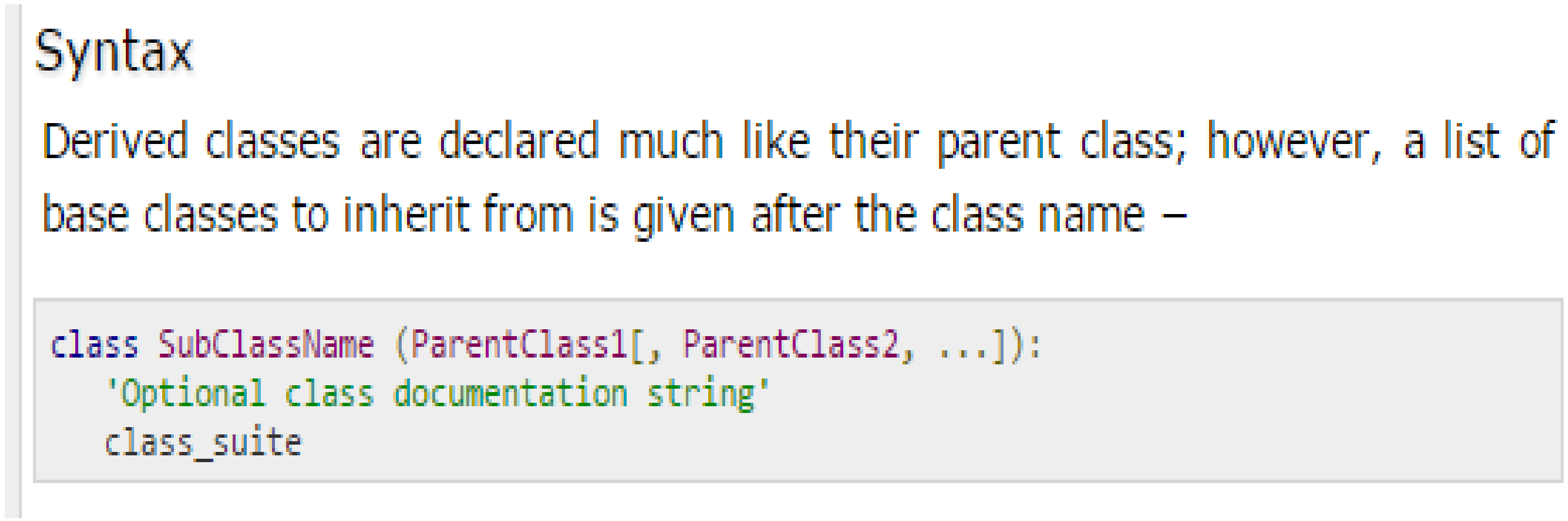
OOPS
IN
PYTHON

Inheritance enable us to define a class that takes all the functionality from parent class and allows us to add more. In this article, you will learn to use inheritance in Python.



- Instead of starting from scratch, you can create a class by deriving it from a preexisting class by listing the parent class in parentheses after the new class name.
- The child class inherits the attributes of its parent class, and you can use those attributes as if they were defined in the child class. A child class can also override data members and methods from the parent.

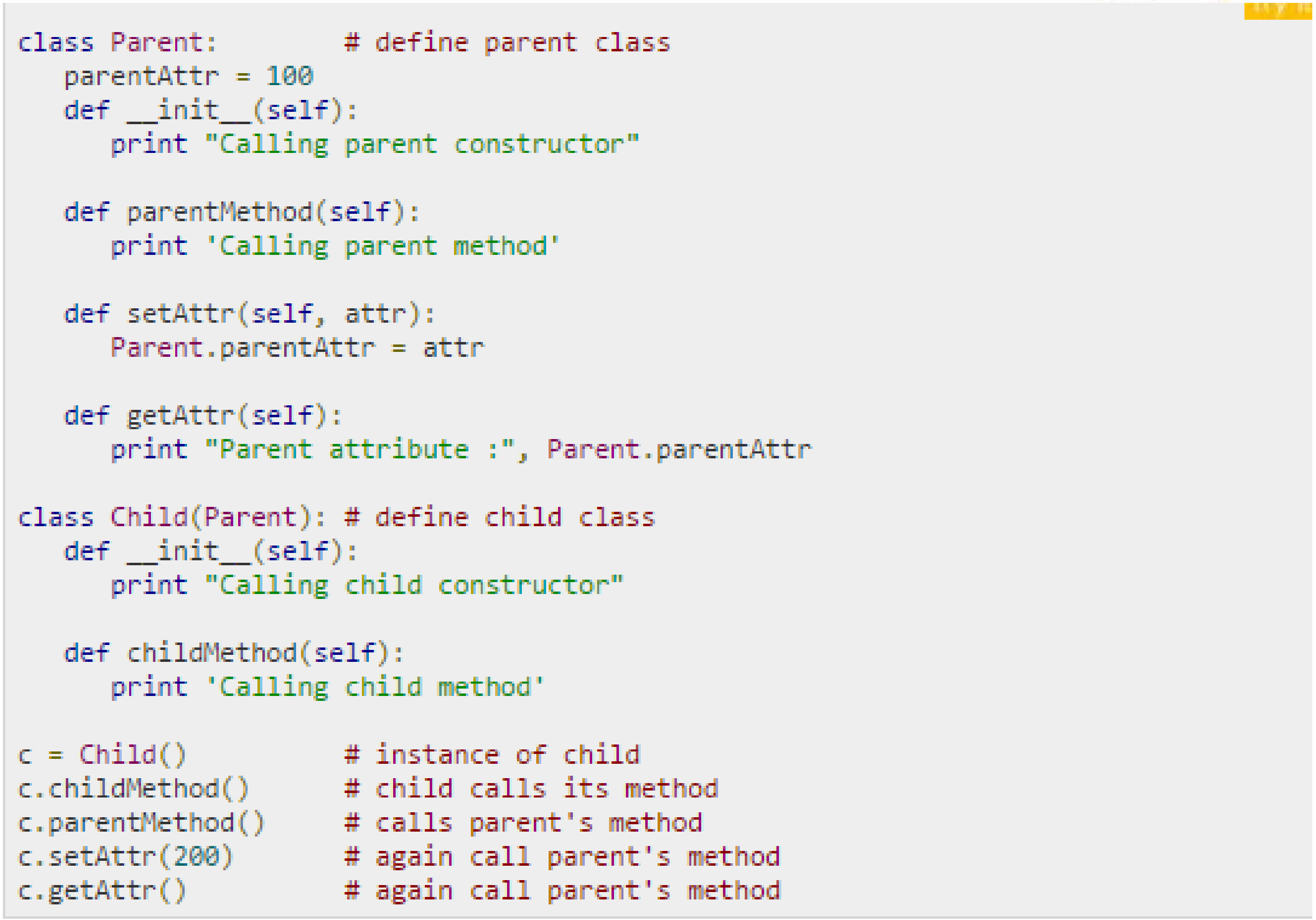
- The Syntax is:



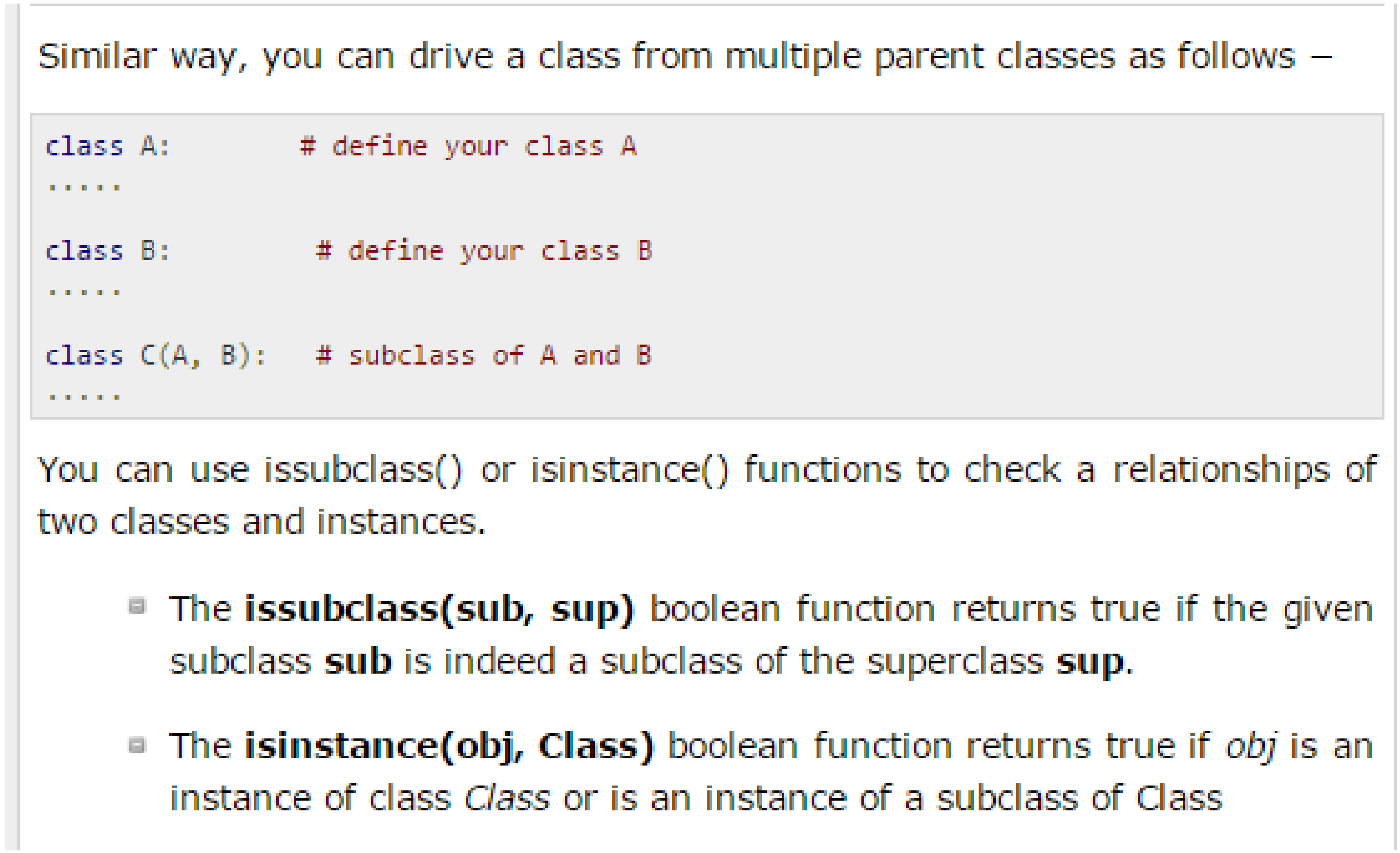
Syntax

Derived classes are declared much like their parent class; however, a list of base classes to inherit from is given after the class name −

```
class SubClassName (ParentClass1[, ParentClass2, ...]):
   'Optional class documentation string'
   class_suite
```



```python
class Parent:        # define parent class
   parentAttr = 100
   def __init__(self):
      print "Calling parent constructor"

   def parentMethod(self):
      print 'Calling parent method'

   def setAttr(self, attr):
      Parent.parentAttr = attr

   def getAttr(self):
      print "Parent attribute :", Parent.parentAttr

class Child(Parent): # define child class
   def __init__(self):
      print "Calling child constructor"

   def childMethod(self):
      print 'Calling child method'

c = Child()          # instance of child
c.childMethod()      # child calls its method
c.parentMethod()     # calls parent's method
c.setAttr(200)       # again call parent's method
c.getAttr()          # again call parent's method
```



Similar way, you can drive a class from multiple parent classes as follows –

```
class A:        # define your class A
.....

class B:         # define your class B
.....

class C(A, B):   # subclass of A and B
.....
```

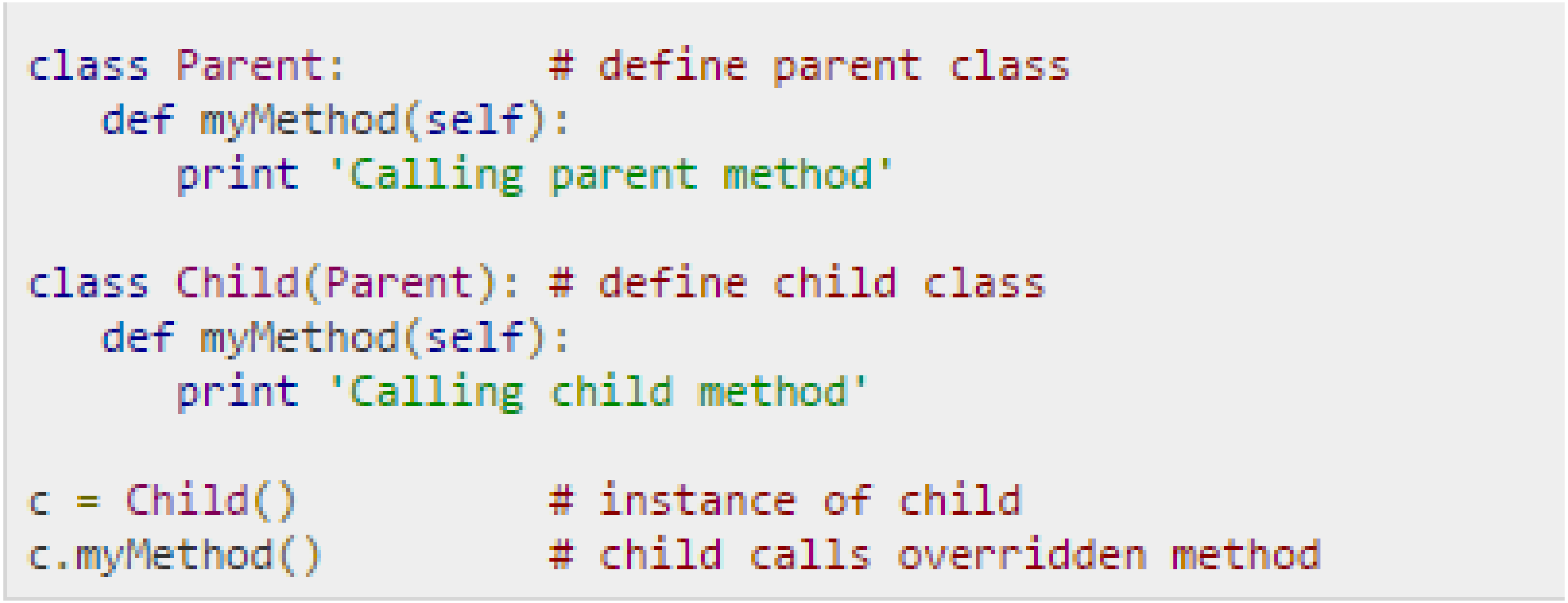
You can use issubclass() or isinstance() functions to check a relationships of two classes and instances.

- The **issubclass(sub, sup)** boolean function returns true if the given subclass **sub** is indeed a subclass of the superclass **sup**.
- The **isinstance(obj, Class)** boolean function returns true if *obj* is an instance of class *Class* or is an instance of a subclass of Class

Overriding Methods

You can always override your parent class methods. One reason for overriding parent's methods is because you may want special or different functionality in your subclass.

Example

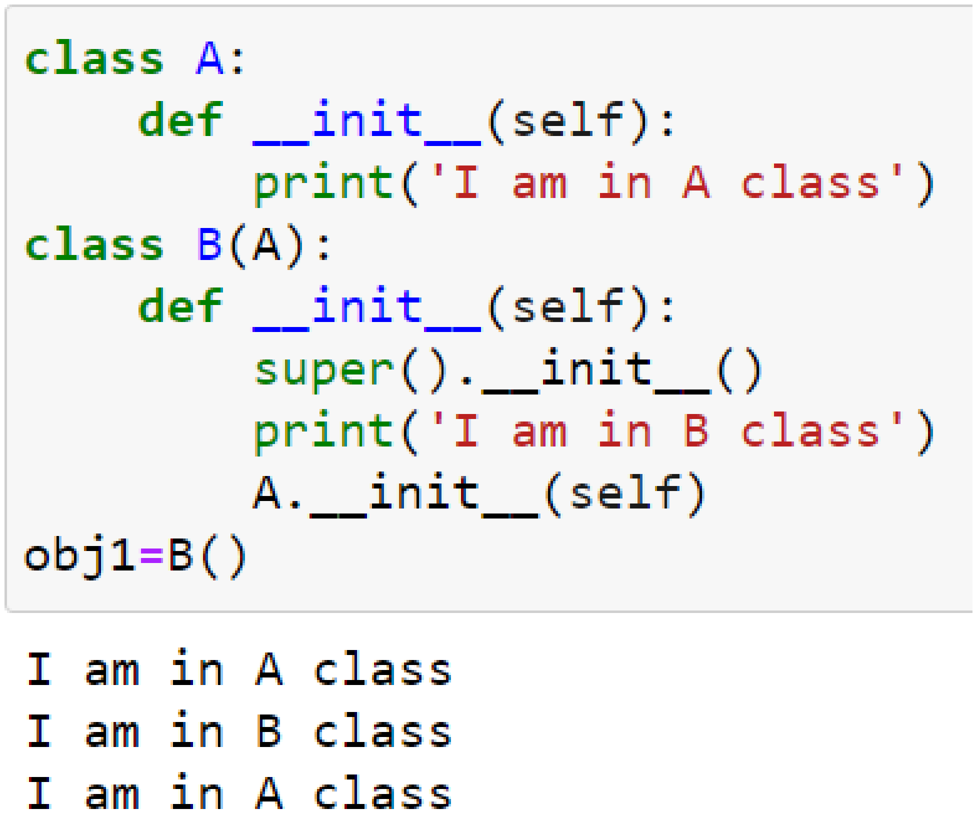


```python
class Parent:        # define parent class
   def myMethod(self):
      print 'Calling parent method'

class Child(Parent): # define child class
   def myMethod(self):
      print 'Calling child method'

c = Child()          # instance of child
c.myMethod()         # child calls overridden method
```

super()

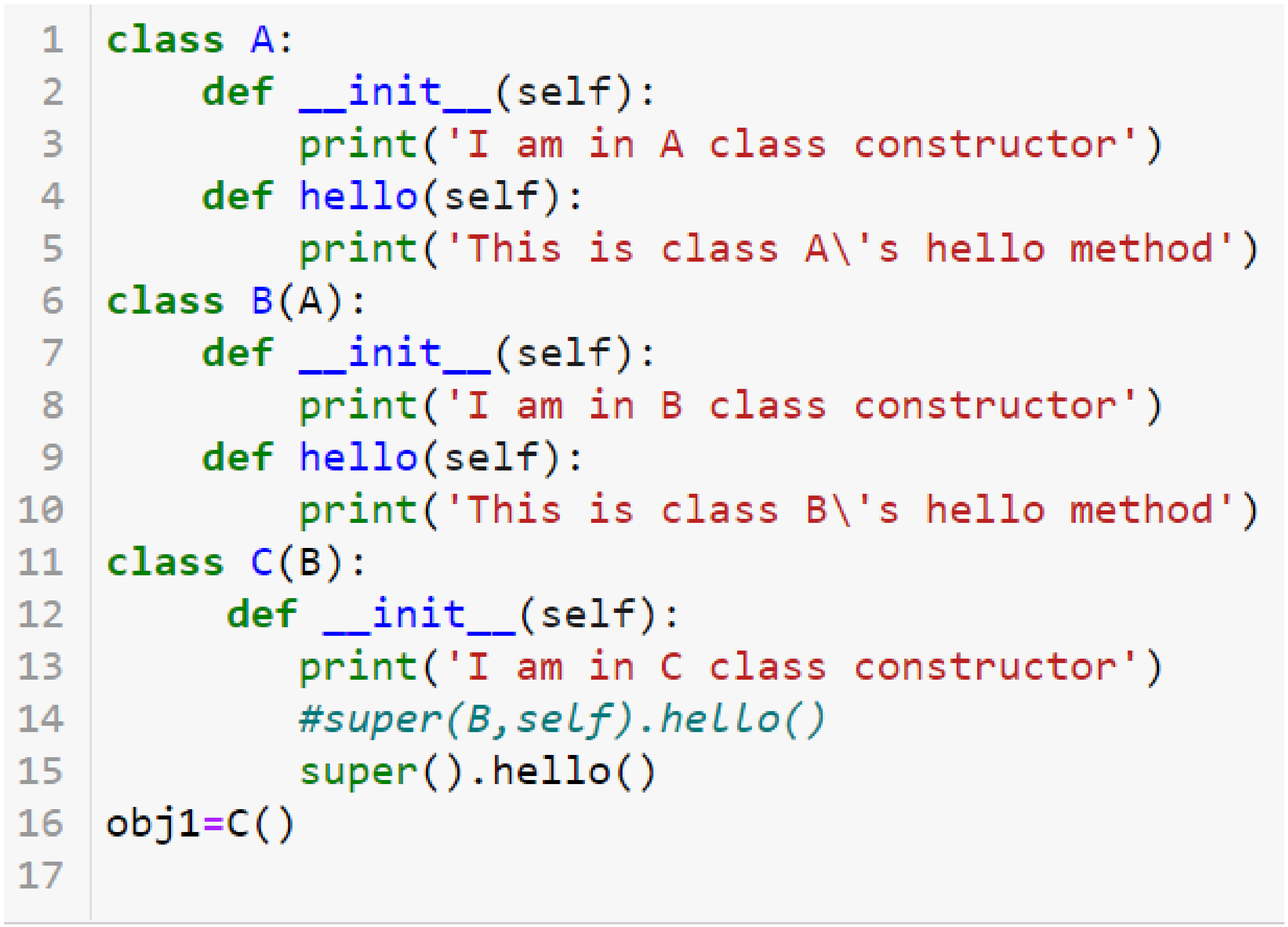
Python super() function provides us the facility to refer to the parent class explicitly. It is basically useful where we have to call superclass functions. It returns the proxy object that allows us to refer parent class by ‘super’.

super()



```python
class A:
    def __init__(self):
        print('I am in A class')
class B(A):
    def __init__(self):
        super().__init__()
        print('I am in B class')
        A.__init__(self)
obj1=B()
```

```
I am in A class
I am in B class
I am in A class
```

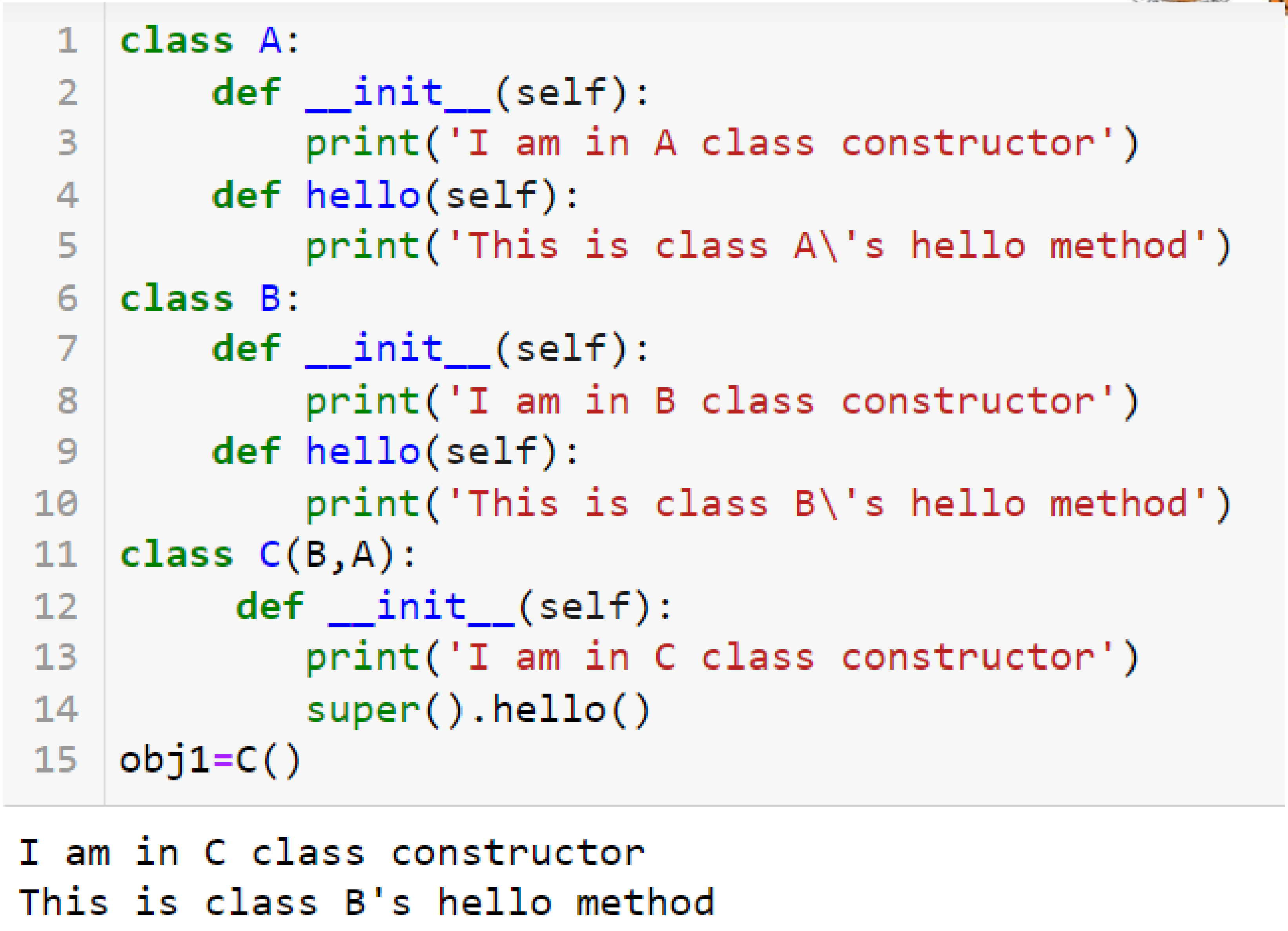


```python
class A:
    def __init__(self):
        print('I am in A class constructor')
    def hello(self):
        print('This is class A\'s hello method')
class B(A):
    def __init__(self):
        print('I am in B class constructor')
    def hello(self):
        print('This is class B\'s hello method')
class C(B):
     def __init__(self):
        print('I am in C class constructor')
        #super(B,self).hello()
        super().hello()
obj1=C()

```

```
I am in C class constructor
This is class B's hello method
```

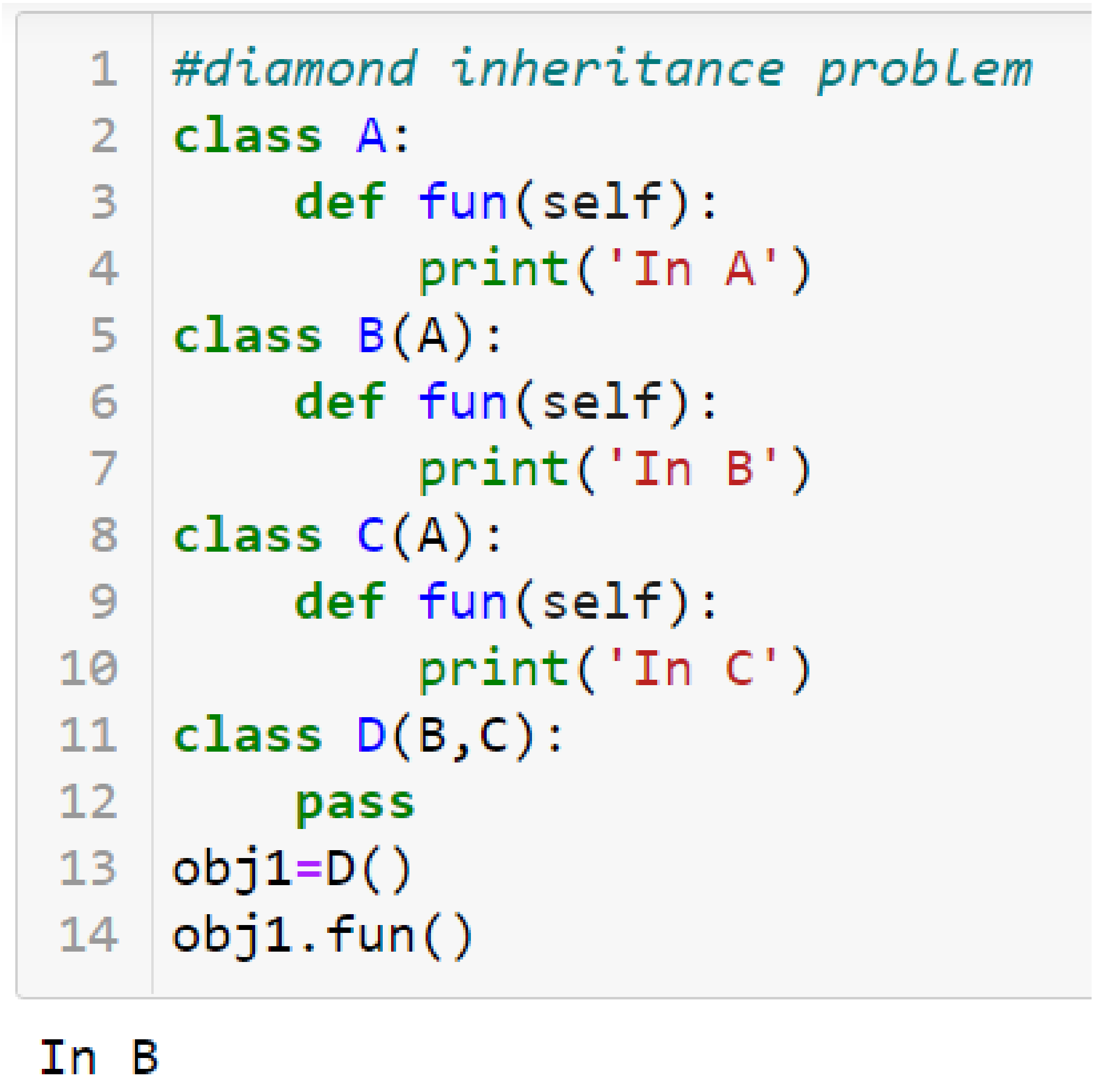
```python
class A:
    def __init__(self):
        print('I am in A class constructor')
    def hello(self):
        print('This is class A\'s hello method')
class B:
    def __init__(self):
        print('I am in B class constructor')
    def hello(self):
        print('This is class B\'s hello method')
class C(A,B):
     def __init__(self):
        print('I am in C class constructor')
        super().hello()
obj1=C()
```

```
I am in C class constructor
This is class A's hello method
```



```python
class A:
    def __init__(self):
        print('I am in A class constructor')
    def hello(self):
        print('This is class A\'s hello method')
class B:
    def __init__(self):
        print('I am in B class constructor')
    def hello(self):
        print('This is class B\'s hello method')
class C(B,A):
     def __init__(self):
        print('I am in C class constructor')
        super().hello()
obj1=C()
```

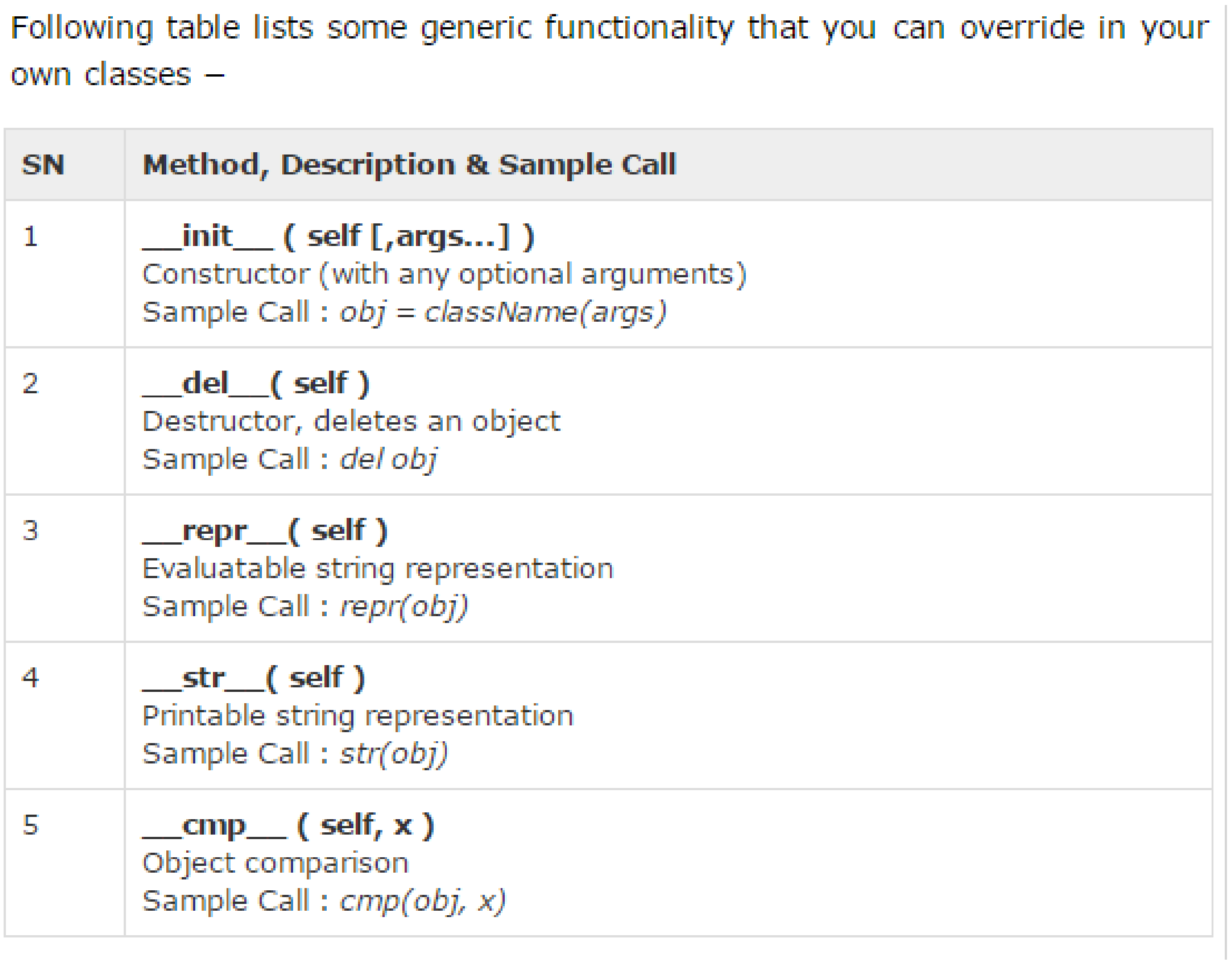
```
I am in C class constructor
This is class B's hello method
```



```python
#diamond inheritance problem
class A:
    def fun(self):
        print('In A')
class B(A):
    def fun(self):
        print('In B')
class C(A):
    def fun(self):
        print('In C')
class D(B,C):
    pass
obj1=D()
obj1.fun()
```

```
In B
```

Base Overloading Methods



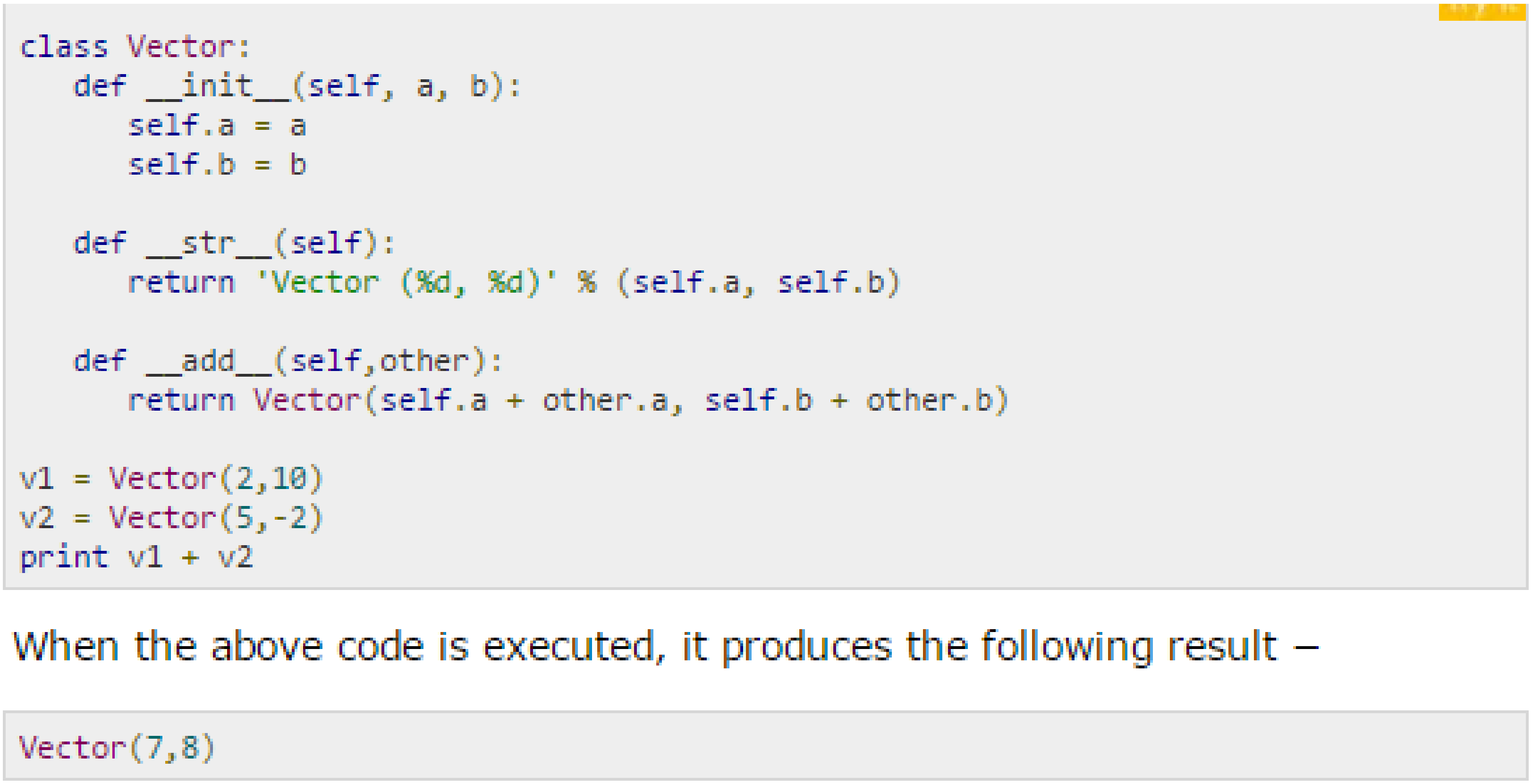
Following table lists some generic functionality that you can override in your own classes −

SN	Method, Description & Sample Call
1	**__init__ (self [,args...])** Constructor (with any optional arguments) Sample Call : *obj = className(args)*
2	**__del__(self)** Destructor, deletes an object Sample Call : *del obj*
3	**__repr__(self)** Evaluatable string representation Sample Call : *repr(obj)*
4	**__str__(self)** Printable string representation Sample Call : *str(obj)*
5	**__cmp__ (self, x)** Object comparison Sample Call : *cmp(obj, x)*

Overloading Operators

- Suppose you have created a Vector class to represent two-dimensional vectors, what happens when you use the plus operator to add them? Most likely Python will yell at you.
- You could, however, define the *__add__* method in your class to perform vector addition and then the plus operator would behave as per expectation

Example



```
class Vector:
   def __init__(self, a, b):
      self.a = a
      self.b = b

   def __str__(self):
      return 'Vector (%d, %d)' % (self.a, self.b)

   def __add__(self,other):
      return Vector(self.a + other.a, self.b + other.b)

v1 = Vector(2,10)
v2 = Vector(5,-2)
print v1 + v2
```

When the above code is executed, it produces the following result –

```
Vector(7,8)
```

Mathematical Operator

Below we have the names of the special functions to overload the mathematical operators in python.

Name	Symbol	Special Function
Addition	`+`	`__add__(self, other)`
Subtraction	`-`	`__sub__(self, other)`
Division	`/`	`__truediv__(self, other)`
Floor Division	`//`	`__floordiv__(self, other)`
Modulus(or Remainder)	`%`	`__mod__(self, other)`
Power	`**`	`__pow__(self, other)`

Assignment Operator

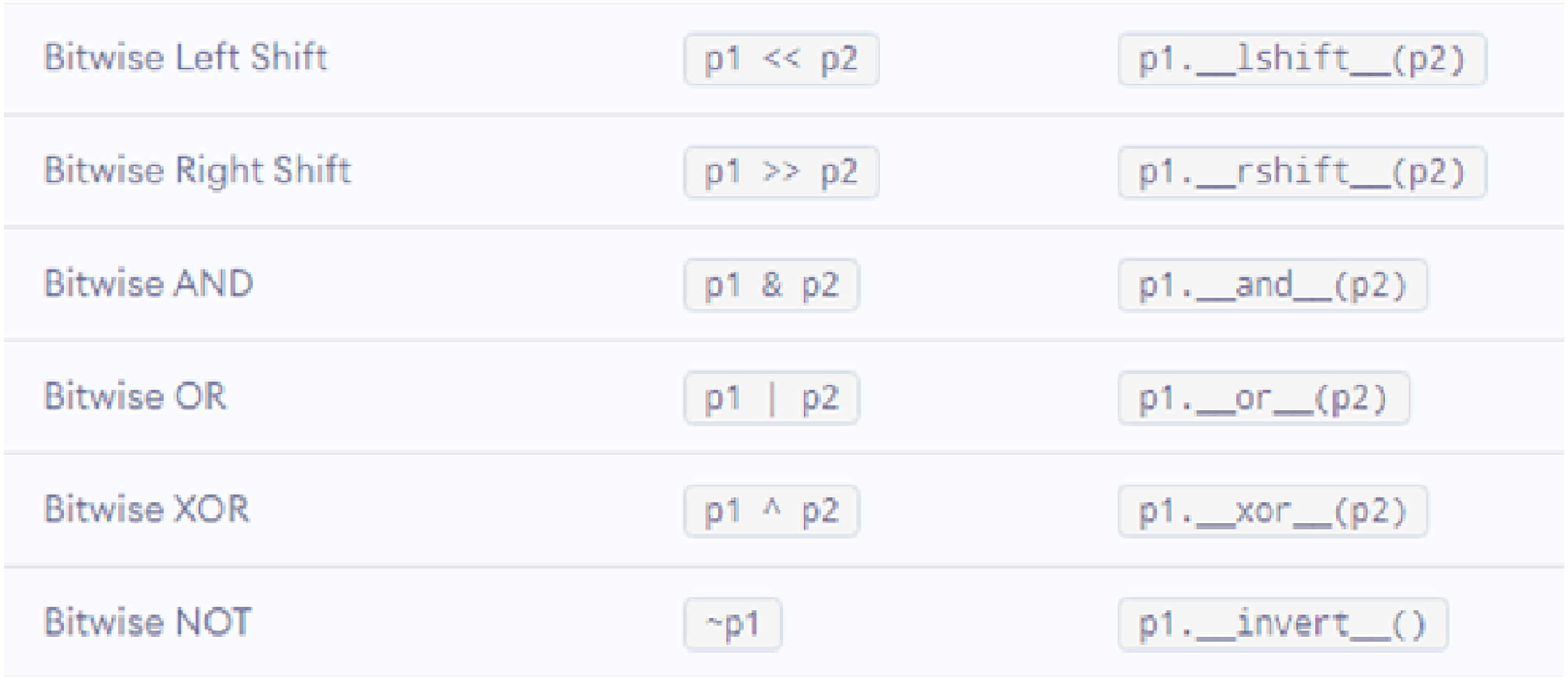
Below we have the names of the special functions to overload the assignment operators in python.

Name	Symbol	Special Function
Increment	`+=`	`__iadd__(self, other)`
Decrement	`-=`	`__isub__(self, other)`
Product	`*=`	`__imul__(self, other)`
Division	`/=`	`__idiv__(self, other)`
Modulus	`%=`	`__imod__(self, other)`
Power	`**=`	`__ipow__(self, other)`

Relational Operator

Below we have the names of the special functions to overload the relational operators in python.

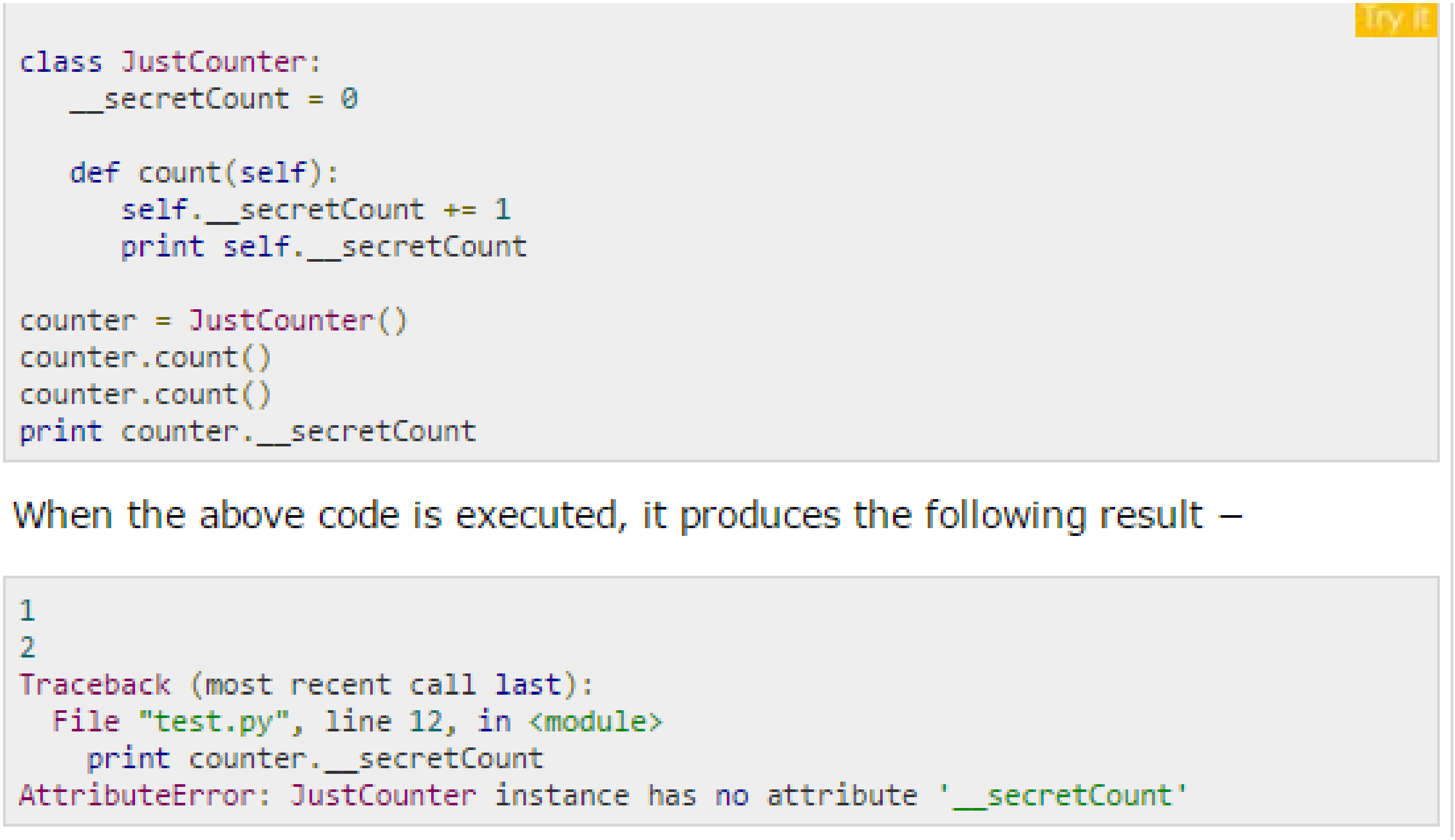
Name	Symbol	Special Function
Less than	`<`	`__lt__(self, other)`
Greater than	`>`	`__gt__(self, other)`
Equal to	`==`	`__eq__(self, other)`
Not equal	`!=`	`__ne__(self, other)`
Less than or equal to	`<=`	`__le__(self, other)`
Greater than or equal to	`> =`	`__gt__(self, other)`



Bitwise Left Shift	`p1 << p2`	`p1.__lshift__(p2)`
Bitwise Right Shift	`p1 >> p2`	`p1.__rshift__(p2)`
Bitwise AND	`p1 & p2`	`p1.__and__(p2)`
Bitwise OR	`p1 | p2`	`p1.__or__(p2)`
Bitwise XOR	`p1 ^ p2`	`p1.__xor__(p2)`
Bitwise NOT	`~p1`	`p1.__invert__()`

Data Hiding

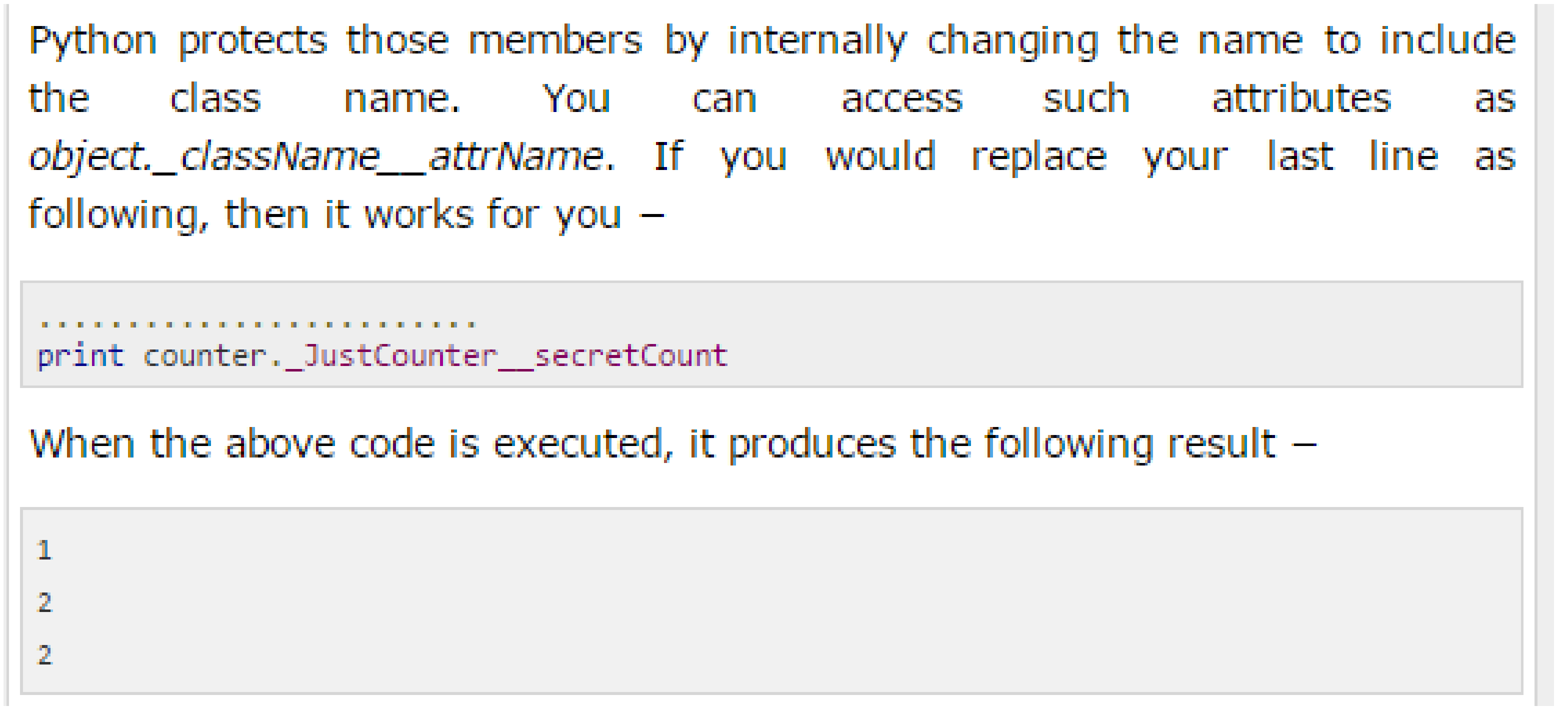
- An object's attributes may or may not be visible outside the class definition. You need to name attributes with a double underscore prefix, and those attributes then are not be directly visible to outsiders.



```python
class JustCounter:
   __secretCount = 0

   def count(self):
      self.__secretCount += 1
      print self.__secretCount

counter = JustCounter()
counter.count()
counter.count()
print counter.__secretCount
```

When the above code is executed, it produces the following result –

```
1
2
Traceback (most recent call last):
  File "test.py", line 12, in <module>
    print counter.__secretCount
AttributeError: JustCounter instance has no attribute '__secretCount'
```



Python protects those members by internally changing the name to include the class name. You can access such attributes as *object._className__attrName*. If you would replace your last line as following, then it works for you –

```
.........................
print counter._JustCounter__secretCount
```

When the above code is executed, it produces the following result –

```
1
2
2
```

Question(Account Class)

- Design the class name **account** that contains:
  - A private **id** for account
  - A private **balance** for account
  - A private **intrate** for annual interest rate
  - A **constructor** that creates an account with specified id, initial balance and interest rate.
  - A method name **getmonthlyinterestrate()** that return monthly interest rate.
  - A method name **checkbalance()** that return balance amount from the account.
  - A method name **withdraw()** that withdraws a specified amount from the account.
  - A method name **deposite()** that deposits a specified amount to the account.

  == Create 10 accounts with initial balance 100 and rate of interest 4%.

  ==The system prompts the user to enter account ID, if it is correct open a main menu having chaises

  1. Check balance, 2. withdraw, 3. deposit and 4. exit.