OOP 개념과 Python OO

High-Level Programming Paradigms

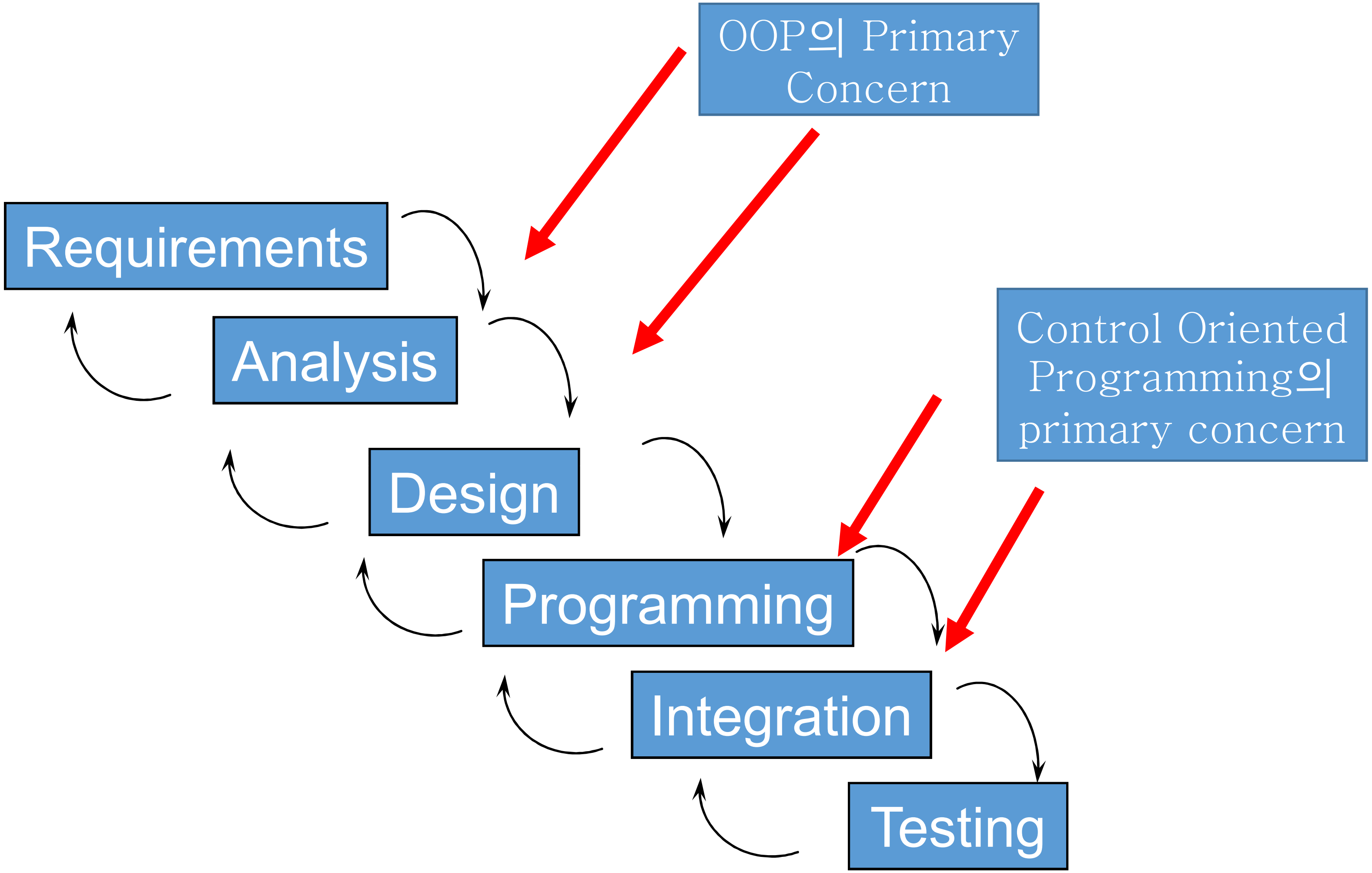
- Control-oriented Programming (before mid 80's)
  - Real world problem ➔ a set of functions
  - Data and functions are separately treated
  - Fortran, Pascal, C

- Object-oriented Programming (after mid 80's)
  - Real world problem ➔ a set of classes
  - Data and functions are encapsulated inside classes
  - C++, Java, and most Script Languages (Python, Ruby, PHP, R,…)

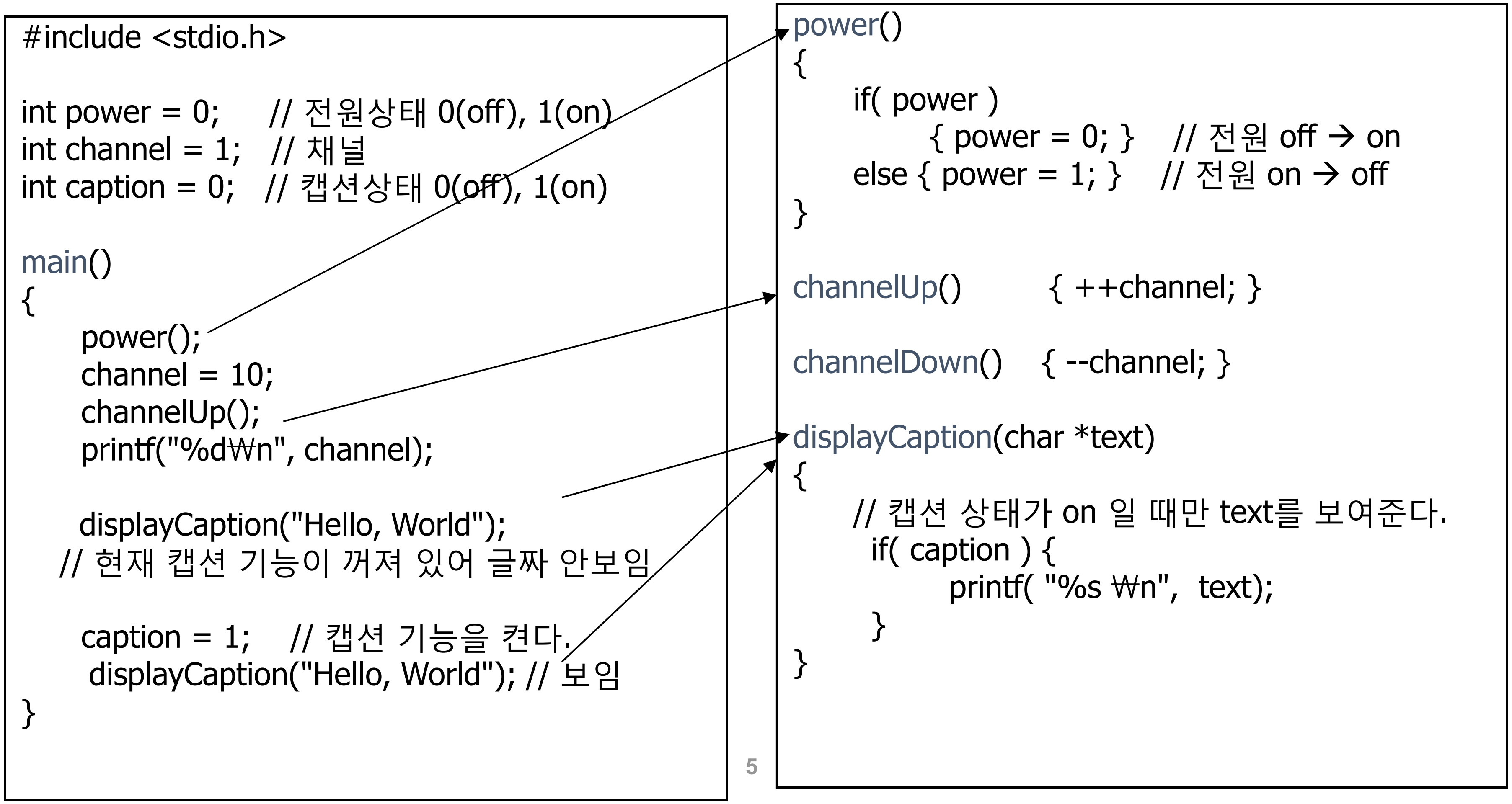
The Software Development Process: The WaterFall Model

- Analyze the Problem
  - Figure out exactly the problem to be solved.
- Determine Specifications
  - Describe exactly what your program will do. (not How, but What)
  - Includes describing the inputs, outputs, and how they relate to one another.
- Create a Design
  - Formulate the overall structure of the program. (*how* of the program gets worked out)
  - You choose or develop your own algorithm that meets the specifications.
- Implement the Design (coding!)
  - Translate the design into a computer language.
- Test/Debug the Program
  - Try out your program to see if it worked.
  - Errors (Bugs) need to be located and fixed. This process is called debugging.
  - Your goal is to find errors, so try everything that might “break” your program!
- Maintain the Program
  - Continue developing the program in response to the needs of your users.
  - In the real world, most programs are never completely finished – they evolve over time.

Waterfall SW Development Model



Typical Control-Oriented Programming:
C code for TV operations



```
#include <stdio.h>

int power = 0;      // 전원상태 0(off), 1(on)
int channel = 1;    // 채널
int caption = 0;    // 캡션상태 0(off), 1(on)

main()
{
      power();
      channel = 10;
      channelUp();
      printf("%d₩n", channel);

      displayCaption("Hello, World");
    // 현재 캡션 기능이 꺼져 있어 글짜 안보임

      caption = 1;    // 캡션 기능을 켠다.
       displayCaption("Hello, World"); // 보임
}
```

```
power()
{
      if( power )
              { power = 0; }    // 전원 off → on
      else { power = 1; }    // 전원 on → off
}

channelUp()         { ++channel; }

channelDown()    { --channel; }

displayCaption(char *text)
{
      // 캡션 상태가 on 일 때만 text를 보여준다.
        if( caption ) {
                printf( "%s ₩n",  text);
        }
}
```

Typical Object-Oriented Program:
JAVA code for TV operation

TV class

```
class Tv {
   boolean power = false;  // 전원상태(on/off)
   int channel;                      // 채널

    void power()            {power = !power; }
    void channelUp()        {++channel; }
    void channelDown()  {--channel;}
   }
```

CaptionTV class

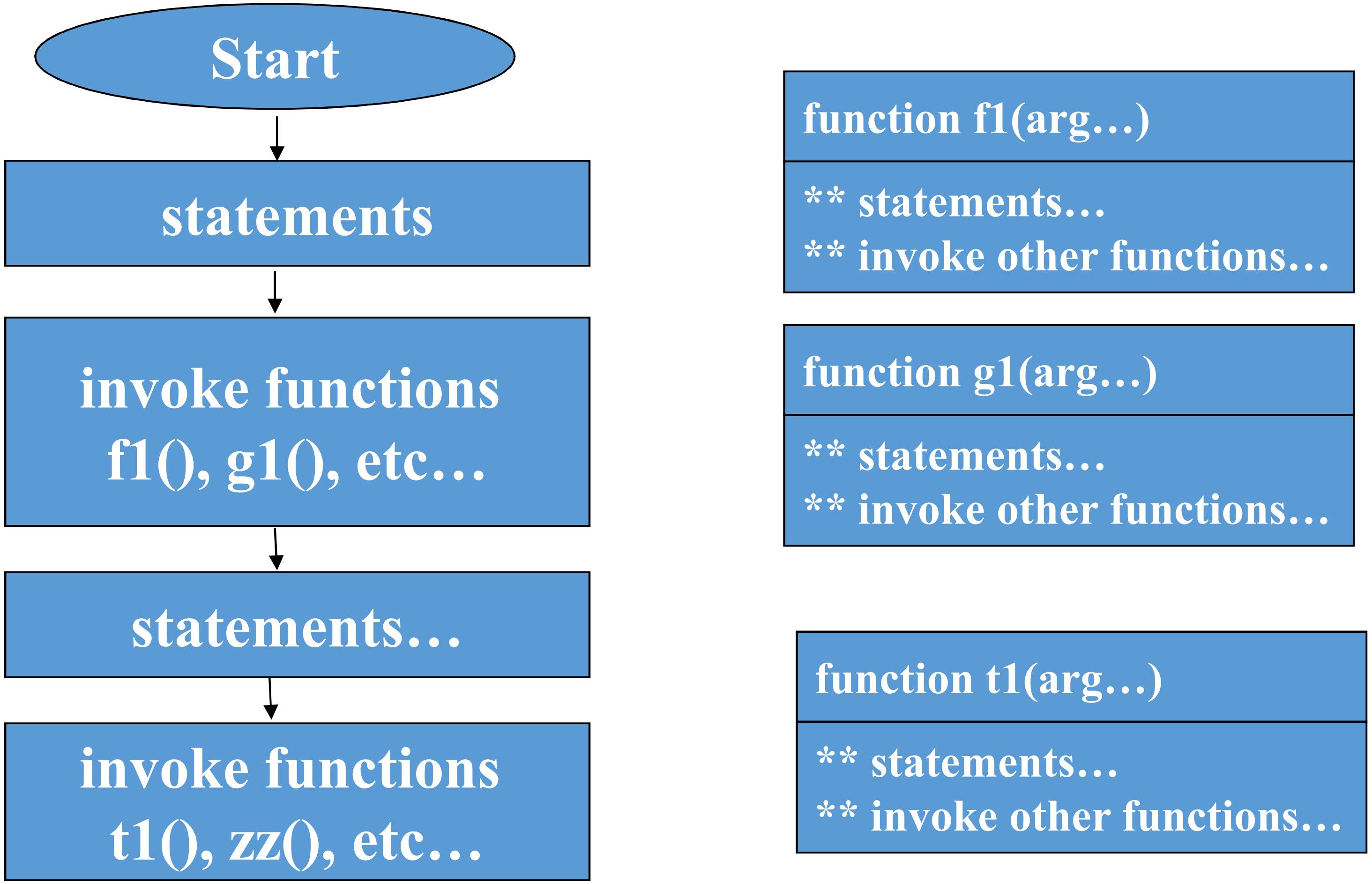
```
class CaptionTv extends Tv {
      boolean caption; // 캡션상태(on/off)

      void displayCaption(String text)
      {
    if (caption) {
        // 캡션  상태가 on(true)일 때만 text를 보임

            System.out.println(text);
      }
      }
}
```

CaptionTVTest class

```
class CaptionTvTest {

 public static void main(String args[])  {

      CaptionTv ctv = new CaptionTv();

      ctv.power();
      ctv.channel = 10;
       ctv.channelUp();

      System.out.println(ctv.channel);

     ctv.displayCaption("Hello, World");
   // 캡션 기능이 꺼져 있어 보여지지 않는다.

      ctv.caption = true;   // 캡션기능을 켠다.

      ctv.displayCaption("Hello,World");
      // 캡션을 화면에 보여 준다.
     }
}
```

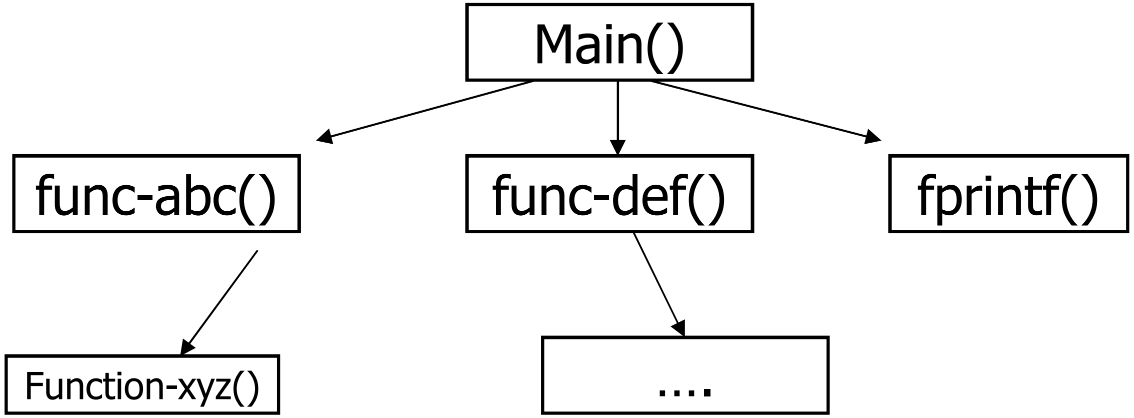
Control Oriented Programming Paradigm

- In traditional control-oriented programming, program describes what the machine will do sequentially, grouping commonly used parts into "functions".
- Fortran, Pascal, C, and many more old programming languages



Sample C program (Function-based structure)

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv) {

   int i, j, k, l;

for(i=0; i < argc; i++) {
        func-abc();
        func-def();
        fprintf();
}
}

func-abc ( )  {   …….  }

func-def ( )  { ….. funct-xyx()    }

func-xyz ( )  { ………}
```



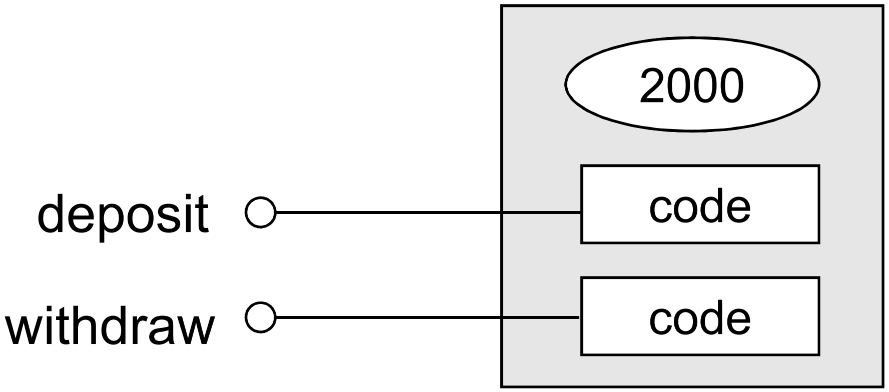
Object-Oriented Programming

- C++
- Java
- Most Script Languages
  - Python, JavaScript, PHP, ……

Object

An encapsulated software structure which usually models an application domain entity

Object = **Data + Operations**



Bank Account object

Related Terms

Instance variables

The variables(data) contained in an object are called *instance variables.*

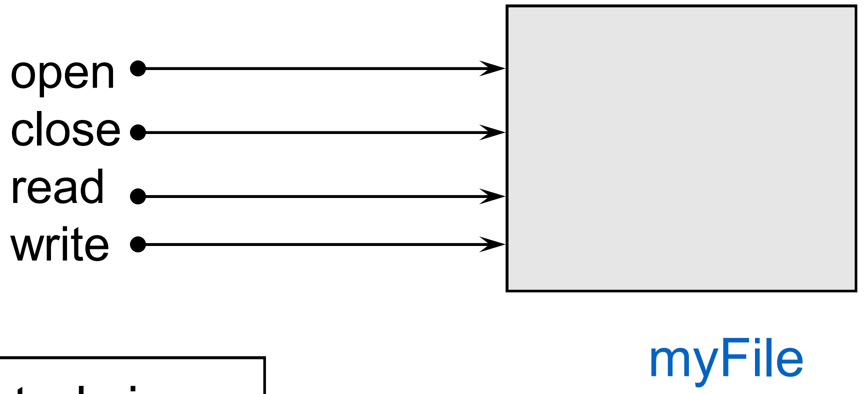
Method

An operations of an object is called *method*.

Message

A request to invoke an operation is called *message.*

Example: File Object



Object-oriented view

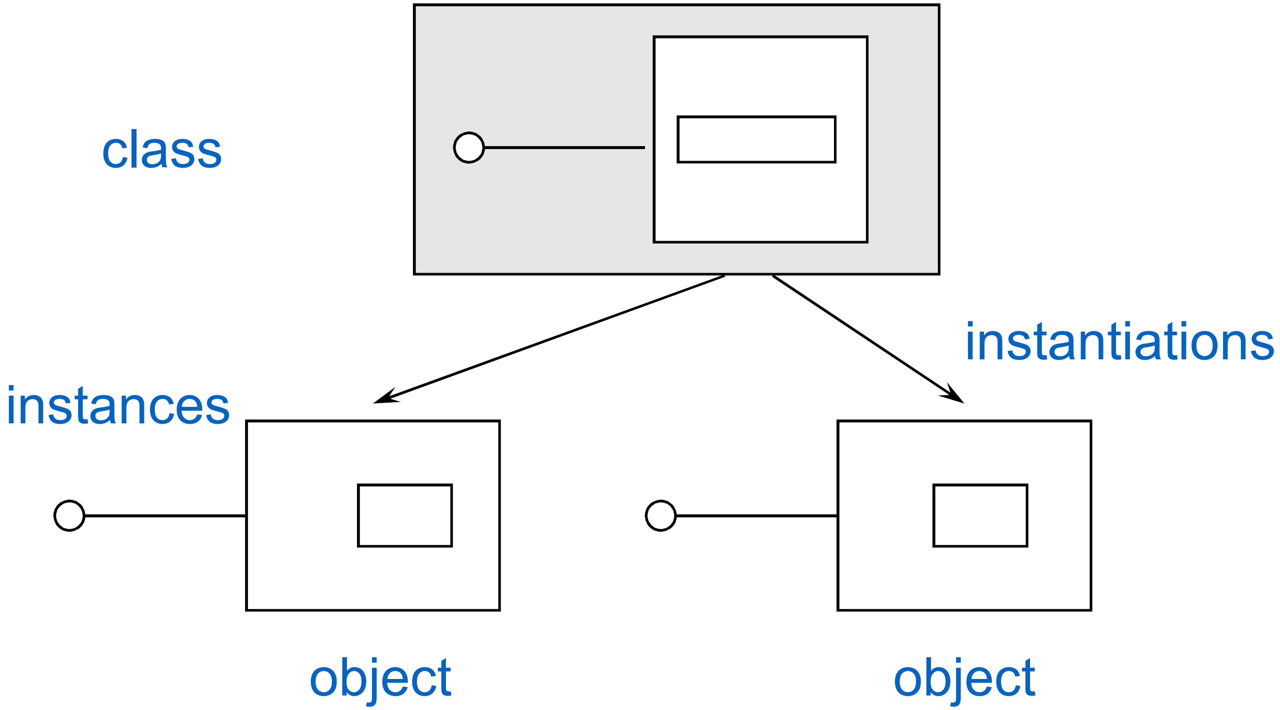
myFile.open() : myFile, please open yourself.

myFile.read(ch) : myFile, please give me the next char.

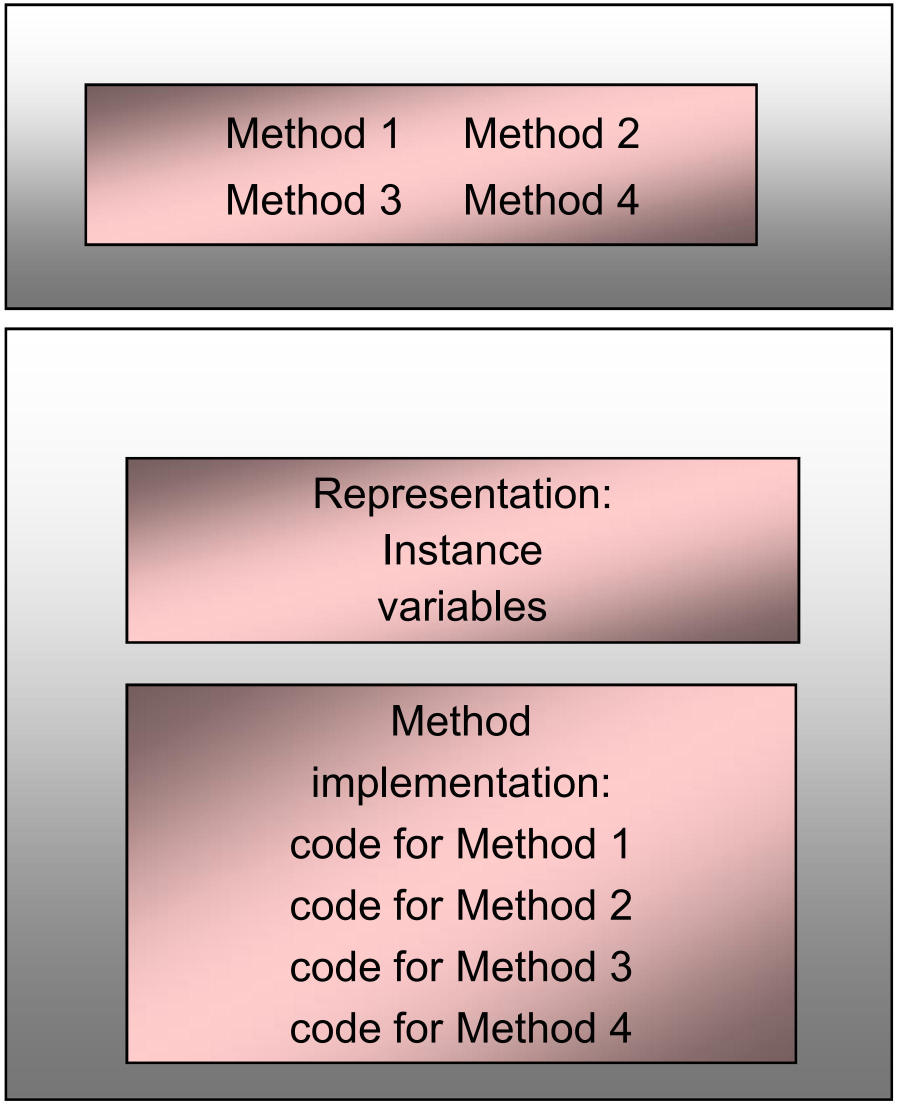
myFile.close() : myFile, close yourself.

Class

An abstract data type which define
the representation and behavior of objects.



The overall structure of an abstract data type (Class)

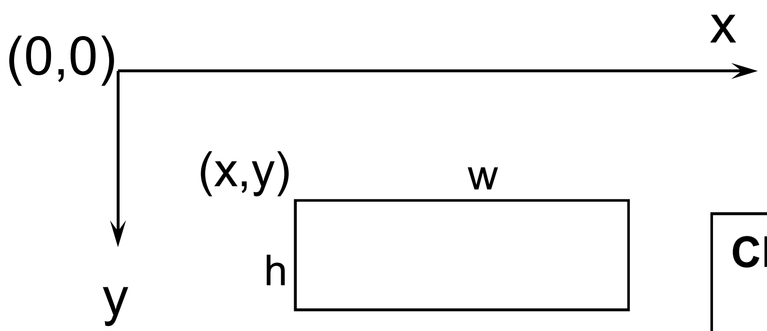


Interface
Public

Implementation
Private

Example: Rectangle

Define a Rectangle Class.

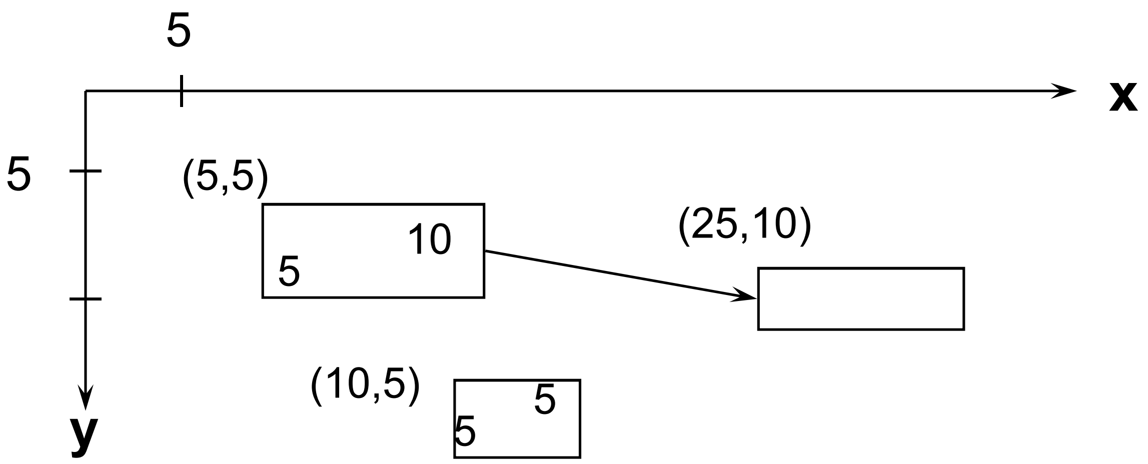


```
Class Rectangle
   data
            int x, y, h, w;
   method
            create (int x1, y1, h1, w1)
              { x=x1;   y=y1;   h=h1;   w=w1; }

            moveTo (int x1, y1)
              { x=x1;   y=y1;   self.display(); }

            display ()
              { drawRectangle(x, y, h, w); }
End Class
```

Interface of Rectangle

```
class Rectangle {
  operations
      create(int x1, y1, h1, w1);

      moveTo(int x1, y1);

      display();
}
```

Example: Using Rectangle object



```
#Import Rectangle

Rectangle  r1, r2;
r1.create(5, 5, 10, 5);
r1.display();
r1.moveTo(25,10);

r2.create(10, 15, 5, 5);
r2.display();
```

Rectangle object 를 직접 manipulat안했다면…

(5, 5, 5, 10) 에 (20, 5, 0, 0) 을 더하여 (25,10,5,10)를 만들고…

이런 rectangle 이 여러 개 있다면…

Why we need Objects?

- Box의 width, length, height 의 variable이 있다고 하자.

- Python에서 W, L, H 의 variable 을 써서 어떤 한 개 Box를 표현했다면….

```
W = 10
L  = 15
H  = 12
```

- (W, L, H) 가 그 Box를 표현하고 있는데... 이런 표현을 할 방법이 없나?
- Box structure ➔ (W, L, H) 뭐 이런거가 있으면 좋겠는데…
- 단순 Variable들을 묶어서 복잡한 구조를 표현하면 좋을텐데..

- 앞페이지에서도 rectangle 을 (x1, y1, width, height) 로 표현하고 있는데..
- Rectangle r1, r2 하는 식으로 표현을 하니까 r1 과 r2 를 직접 manipulate 할수 있네!!!!

Related Terms

Behavior

The set of methods exported by an object is called its *behavior* or *interface*.

Encapsulation

The data of an object can only be accessed via the methods of the object.

Data Abstraction

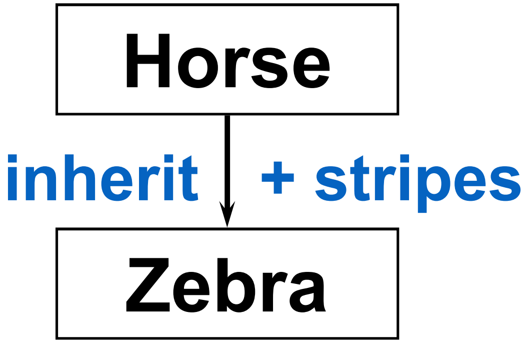
Definition of an abstract type.
Encapsulation is need.

Inheritance

A mechanism which allows a new class to be incrementally defined from an existing class.

Problem **What is a Zebra ?**

"A Zebra is like a horse but has stripes"



Horse

inherit **+ stripes**

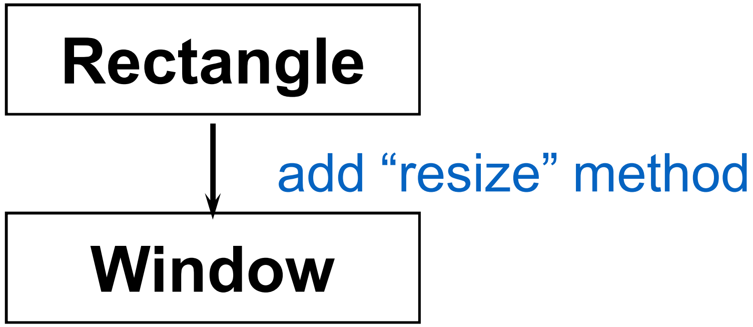
Zebra

Inheritance avoid repetition and confusion!

Example: Inheritance

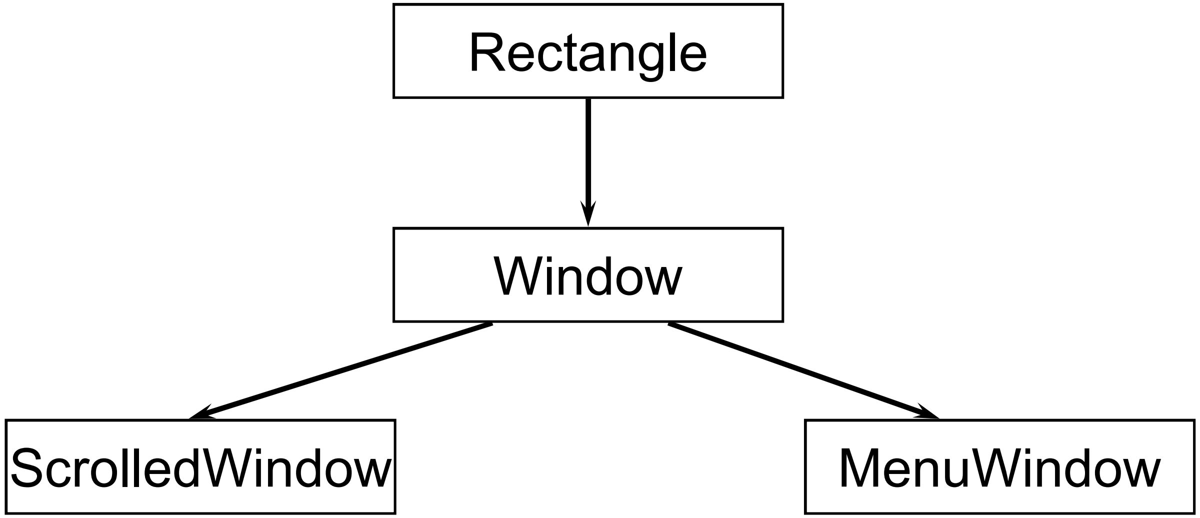
Problem

Define a new class called Window, which is a rectangle, but also resizable.



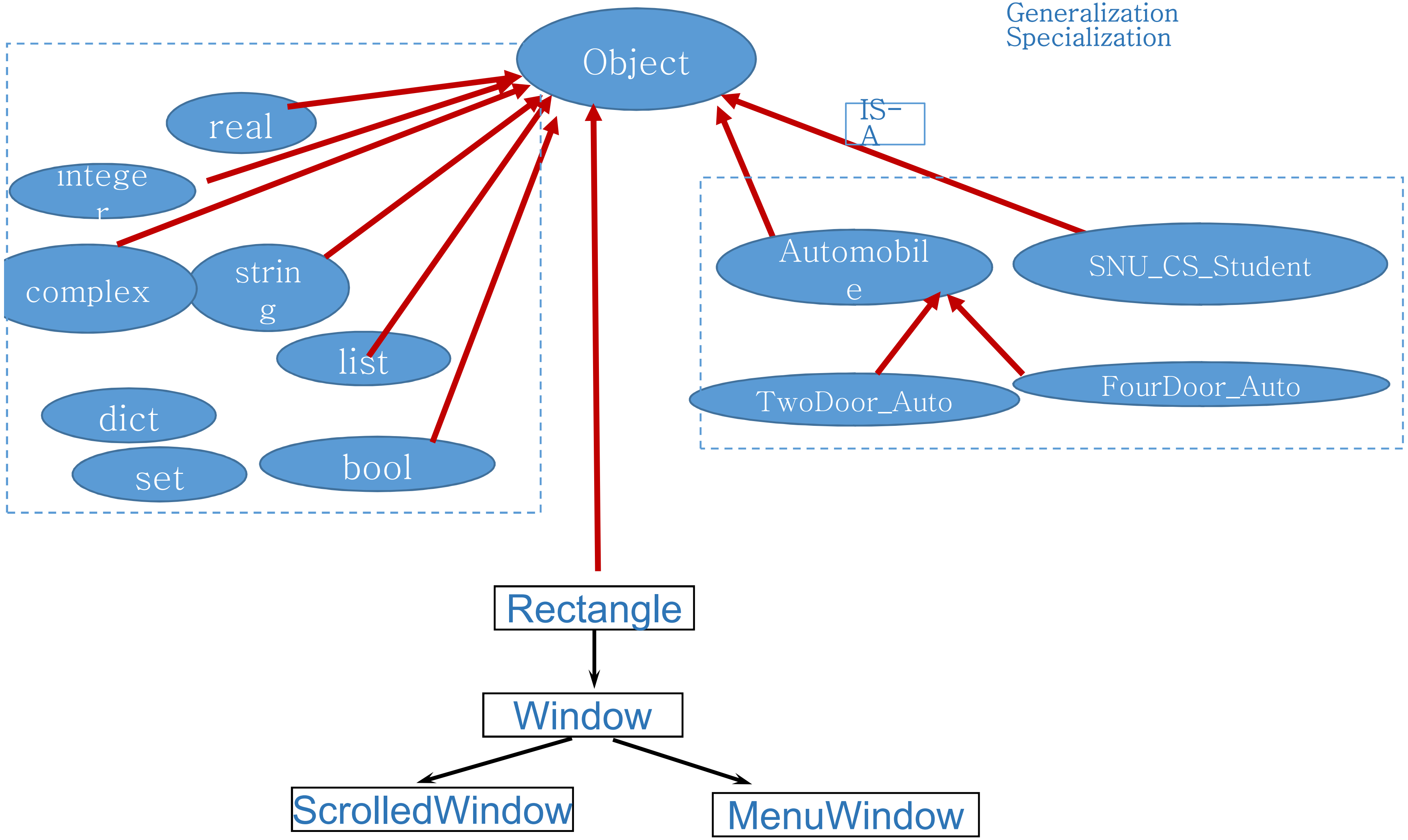
```
Class Window
   inherit Rectangle
   add operation
         resize(int h1, w1)
               { h=h1; w=w1; display(); }
```

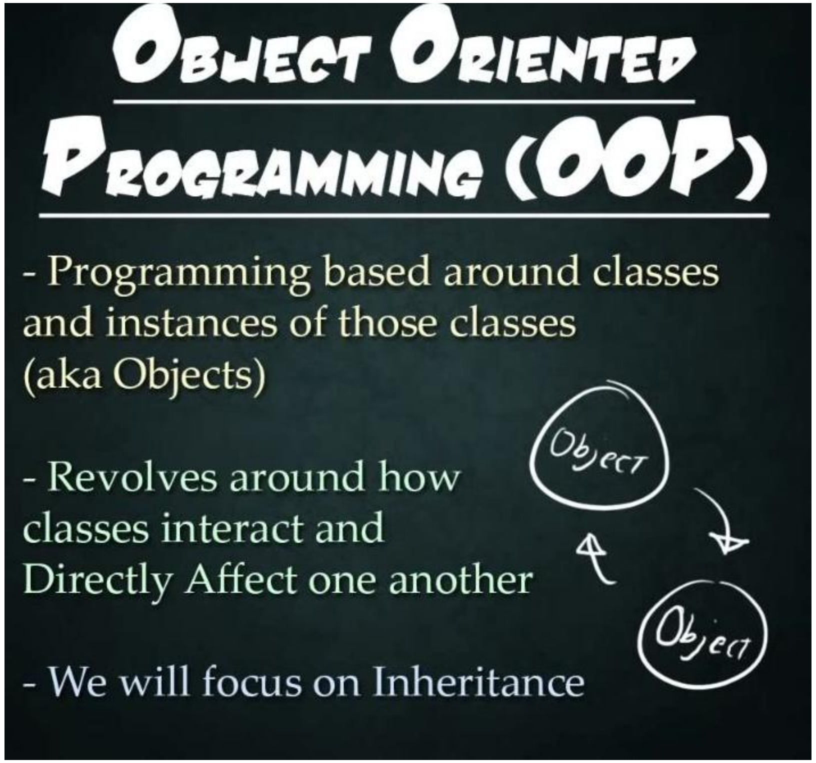
Class Hierarchy



Inheritance builds class hierarchies which are reusable and opens the possibility of application frameworks for domain reuse.

Everything is an Object in OOP





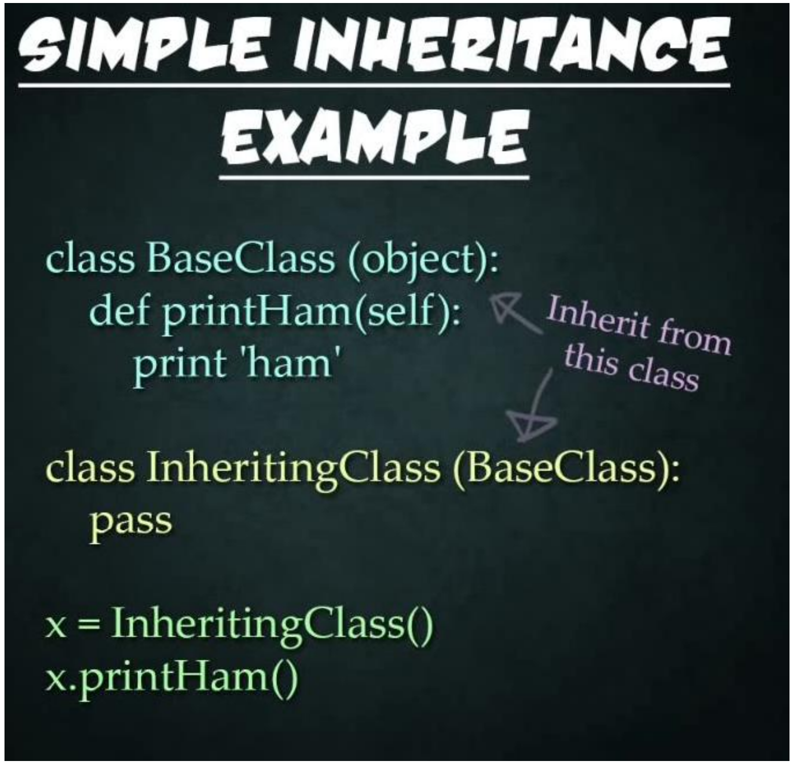
Object Oriented Programming (OOP)

- Programming based around classes and instances of those classes (aka Objects)

- Revolves around how classes interact and Directly Affect one another

- We will focus on Inheritance

Object
Object



Simple Inheritance Example

```
class BaseClass (object):
    def printHam(self):
        print 'ham'

class InheritingClass (BaseClass):
    pass

x = InheritingClass()
x.printHam()
```

Inherit from this class

Why use OOP?

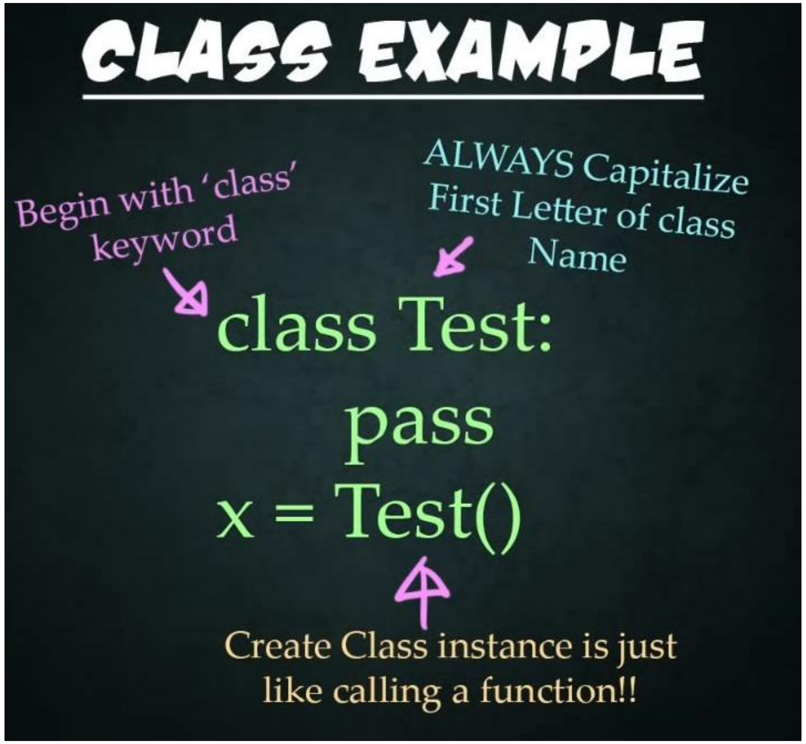
- Cleaner way of Programming!

CLASSES

- Way of packaging Variables and Functions together

Attributes

Actions



CLASS EXAMPLE

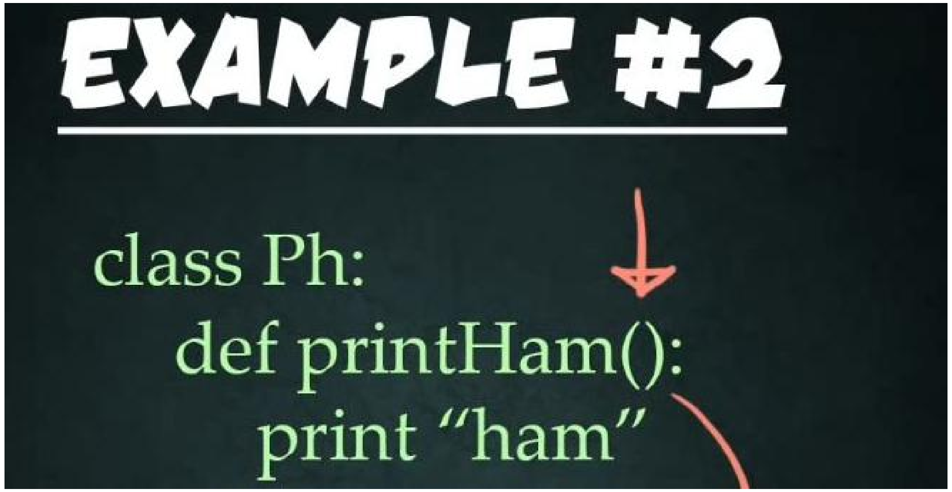
Begin with 'class' keyword

ALWAYS Capitalize First Letter of class Name

```
class Test:
    pass
x = Test()
```

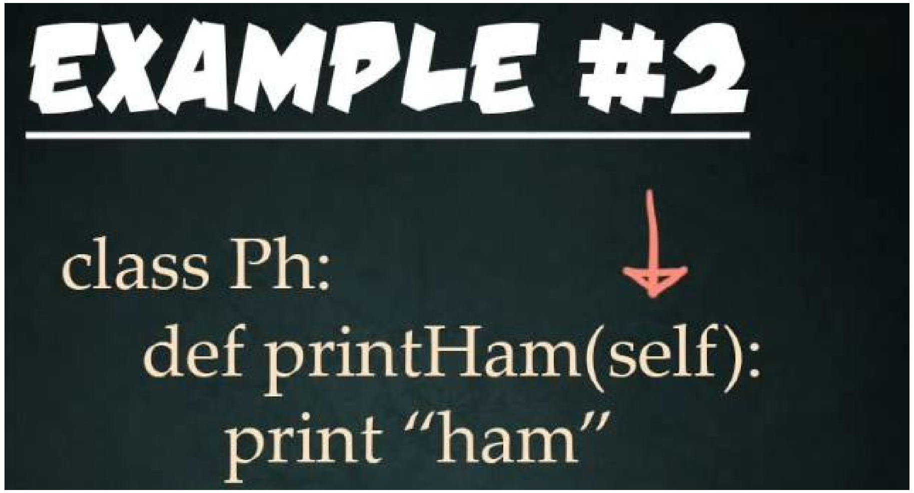
Create Class instance is just like calling a function!!

```
class TestClass:
    pass

X = TestClass()
```



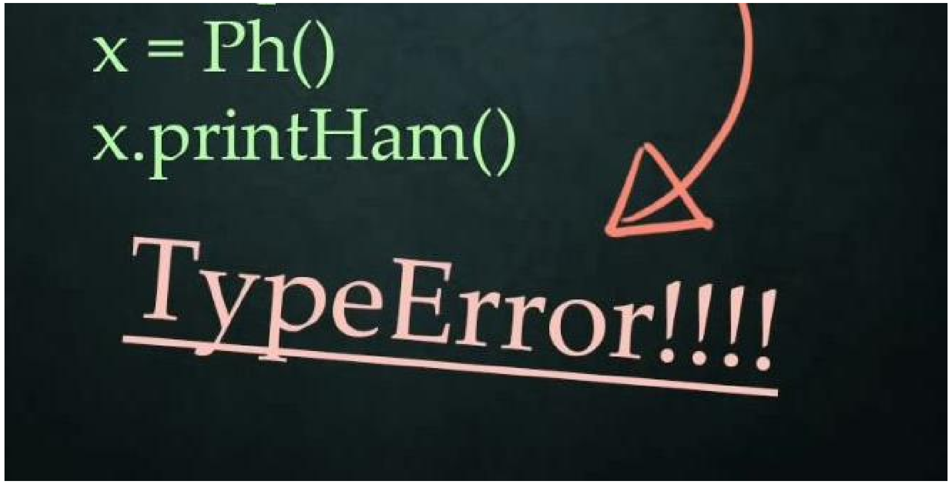
EXAMPLE #2

```
class Ph:
    def printHam():
        print "ham"
```



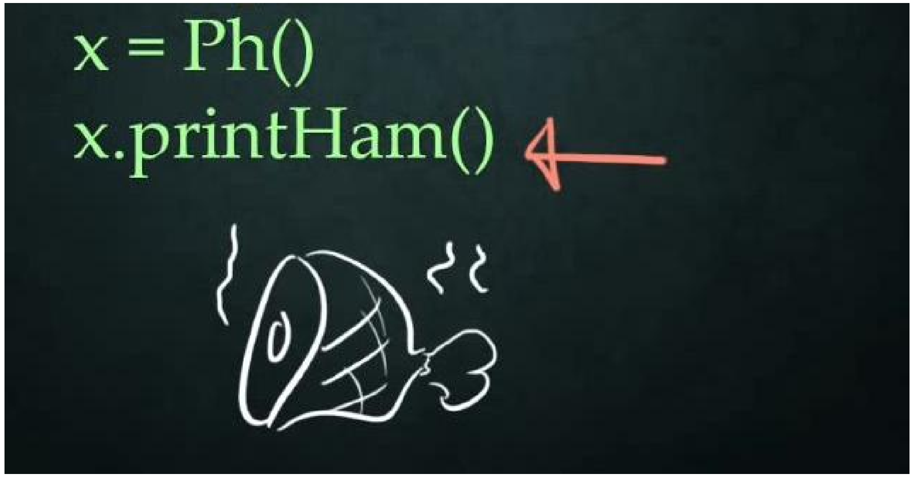
EXAMPLE #2

```
class Ph:
    def printHam(self):
        print "ham"
```



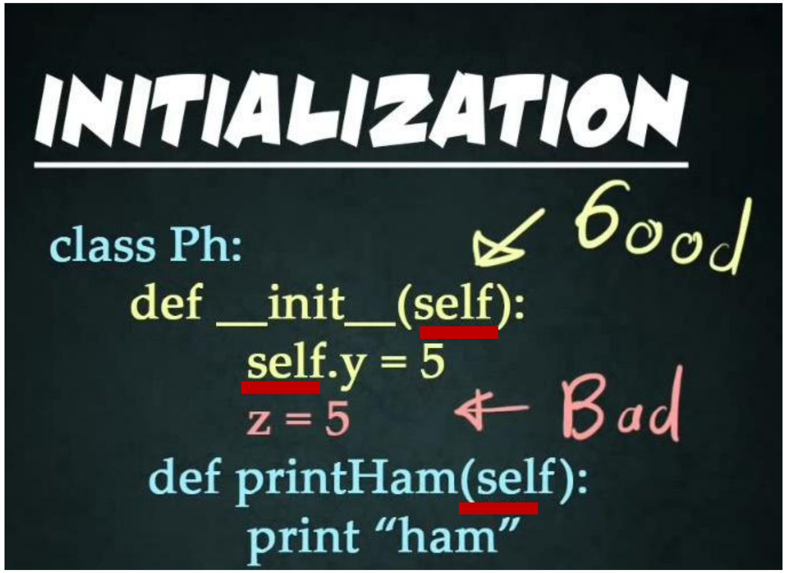
```
x = Ph()
x.printHam()
```

TypeError!!!!

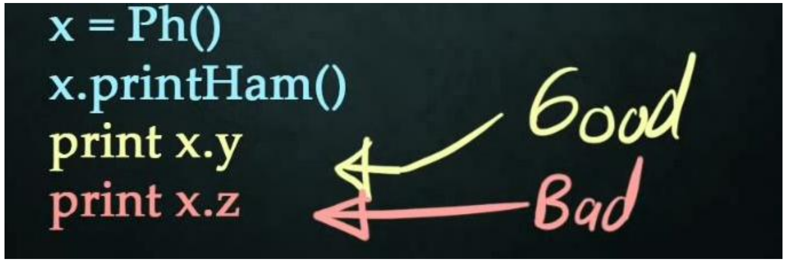


```
x = Ph()
x.printHam()
```

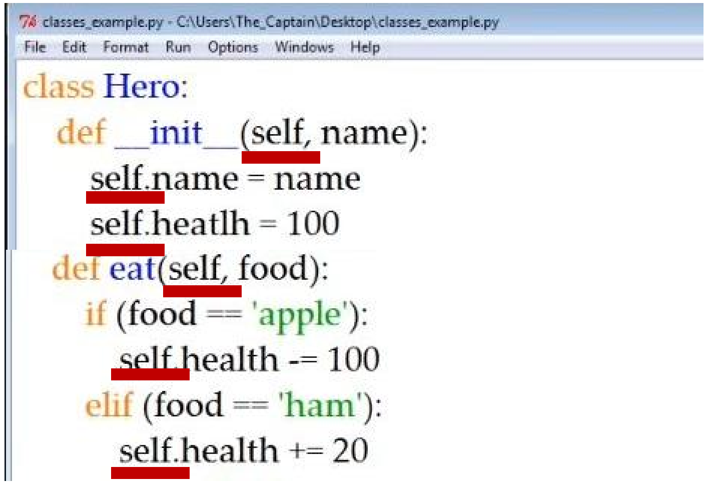
Object에 속한 instance variable들의 선언은 __init__(self) function의 속에서 이루어져야 함



```
INITIALIZATION

class Ph:
    def __init__(self):
        self.y = 5
        z = 5
    def printHam(self):
        print "ham"
```

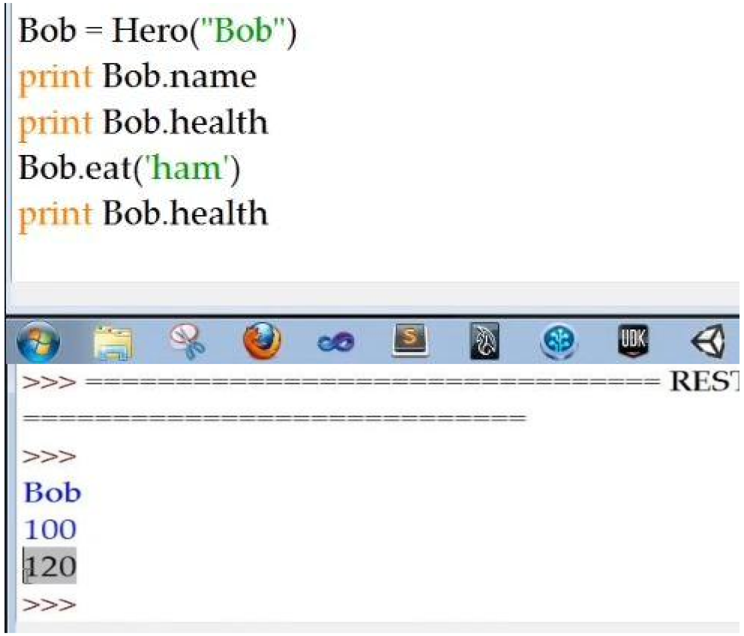


```
x = Ph()
x.printHam()
print x.y
print x.z
```



classes_example.py - C:\Users\The_Captain\Desktop\classes_example.py
File Edit Format Run Options Windows Help

```
class Hero:
  def __init__(self, name):
    self.name = name
    self.heatlh = 100
  def eat(self, food):
    if (food == 'apple'):
      self.health -= 100
    elif (food == 'ham'):
      self.health += 20
```



```
Bob = Hero("Bob")
print Bob.name
print Bob.health
Bob.eat('ham')
print Bob.health
```

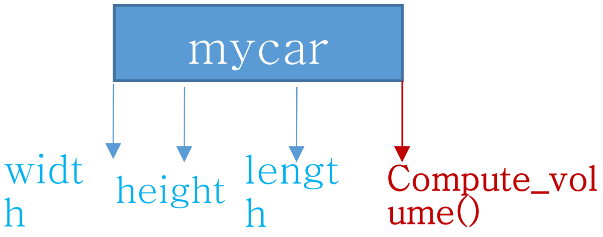
```
>>> ================================ REST
============================
>>>
Bob
100
120
>>>
```

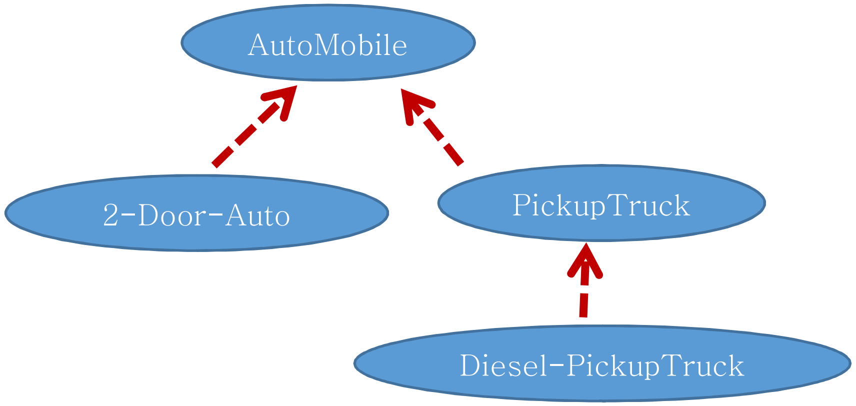
Function-Oriented Python Version of Auto Volume Computation

```
def volume_compute(x, y, z):
    return  x * y * z
def volume_compute1(x, y, z, l)
    return   x * y * z  +  l


def Test():
    print("My Automobile's volume is:",  volume_compute(10, 15, 25))
    print("Your PickupTruck's volume is:",  volume_compute1(10, 15, 25, 1000))
```

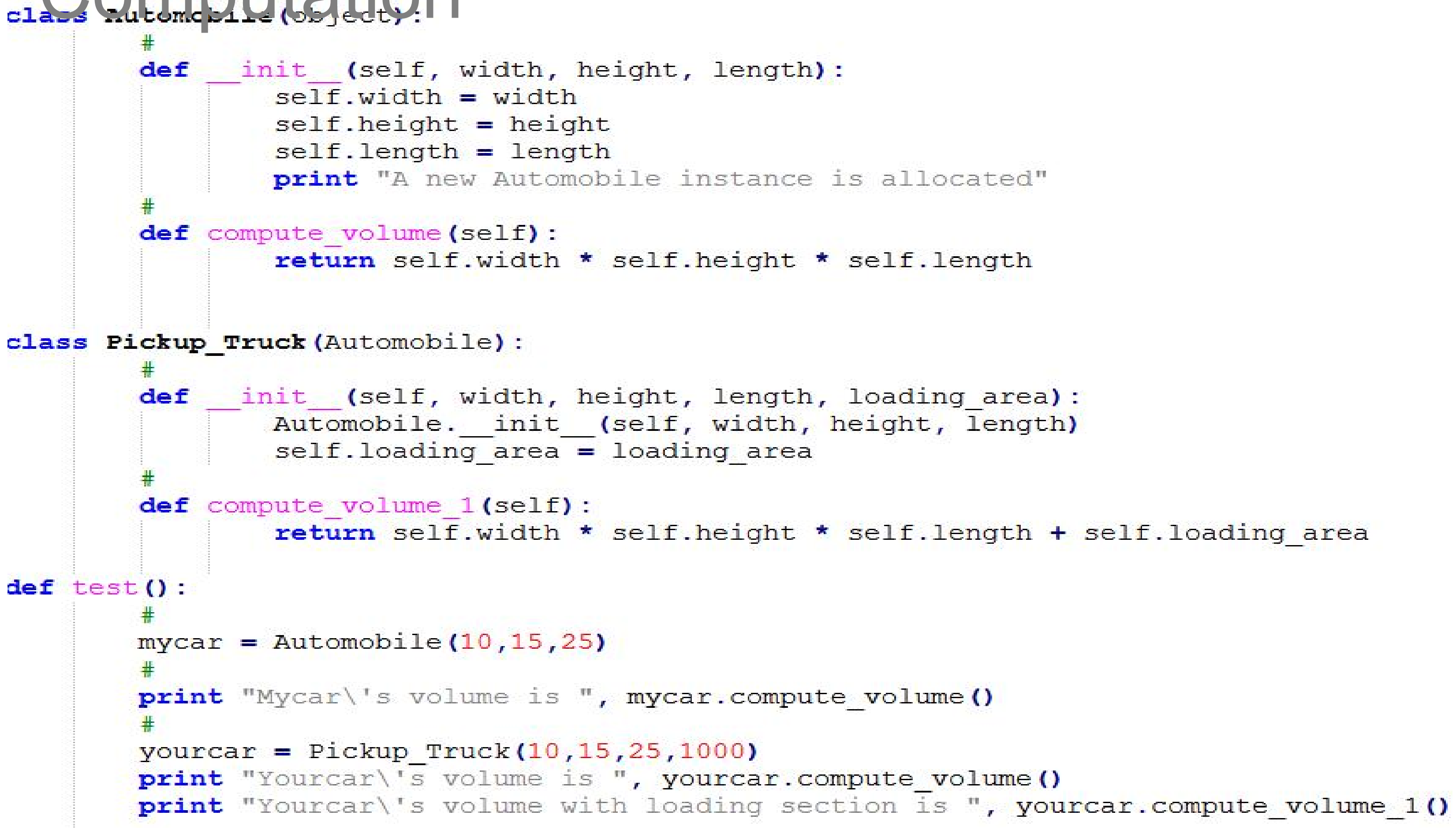
** My Automobile, Your PickupTruck 이라는 실체? (10, 15, 25), (10, 15, 25, 1000)?
2-Door-Auto 혹은 Diesel-PickupTruck 같은 비슷한 자동차에 대해서 무언가를 하고 싶을때?

** In OOP

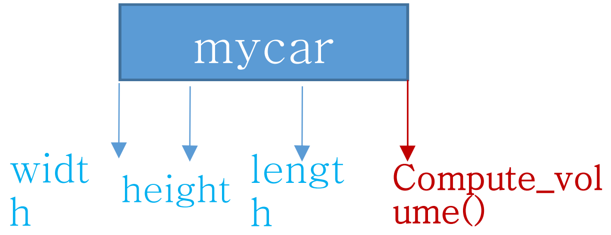


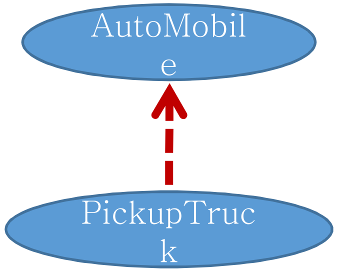


Object-Oriented Python Version of Auto Volume Computation

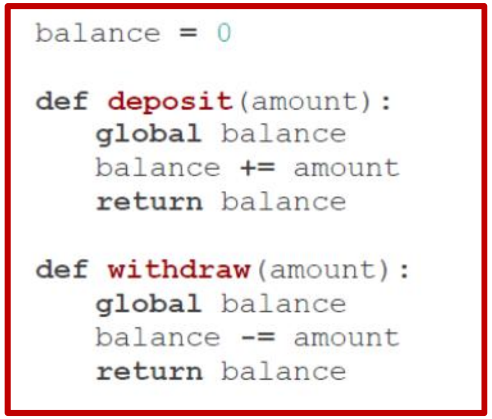


```
class Automobile(object):
    #
    def __init__(self, width, height, length):
        self.width = width
        self.height = height
        self.length = length
        print "A new Automobile instance is allocated"
    #
    def compute_volume(self):
        return self.width * self.height * self.length


class Pickup_Truck(Automobile):
    #
    def __init__(self, width, height, length, loading_area):
        Automobile.__init__(self, width, height, length)
        self.loading_area = loading_area
    #
    def compute_volume_1(self):
        return self.width * self.height * self.length + self.loading_area

def test():
    #
    mycar = Automobile(10,15,25)
    #
    print "Mycar\'s volume is ", mycar.compute_volume()
    #
    yourcar = Pickup_Truck(10,15,25,1000)
    print "Yourcar\'s volume is ", yourcar.compute_volume()
    print "Yourcar\'s volume with loading section is ", yourcar.compute_volume_1()
```





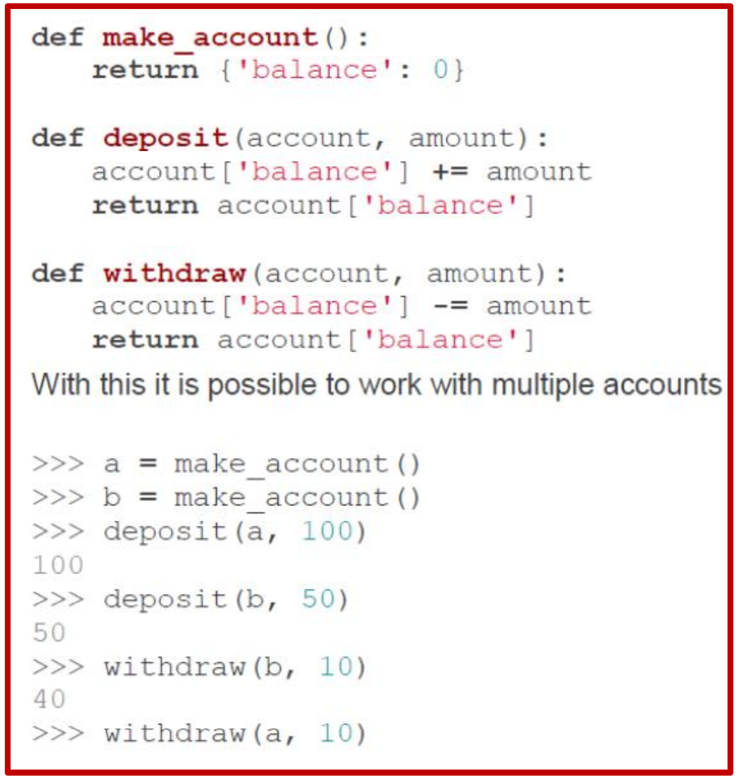
Another Motivational Example



```
balance = 0

def deposit(amount):
    global balance
    balance += amount
    return balance

def withdraw(amount):
    global balance
    balance -= amount
    return balance
```

The above example is good enough only if we want to have just a single account. It is getting complicated if want to model multiple accounts. We can solve the problem by making the state local, probably by using a dictionary to store the state.

Using Dictionary



```
def make_account():
    return {'balance': 0}

def deposit(account, amount):
    account['balance'] += amount
    return account['balance']

def withdraw(account, amount):
    account['balance'] -= amount
    return account['balance']
```

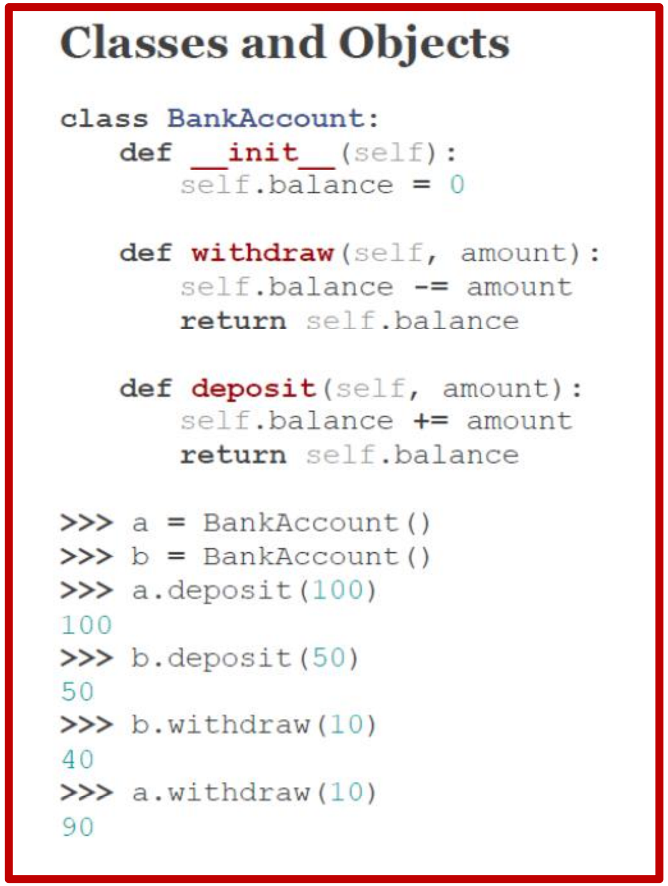
With this it is possible to work with multiple accounts

```
>>> a = make_account()
>>> b = make_account()
>>> deposit(a, 100)
100
>>> deposit(b, 50)
50
>>> withdraw(b, 10)
40
>>> withdraw(a, 10)
```

** 여러 개의 account를 만들려고 한다면?

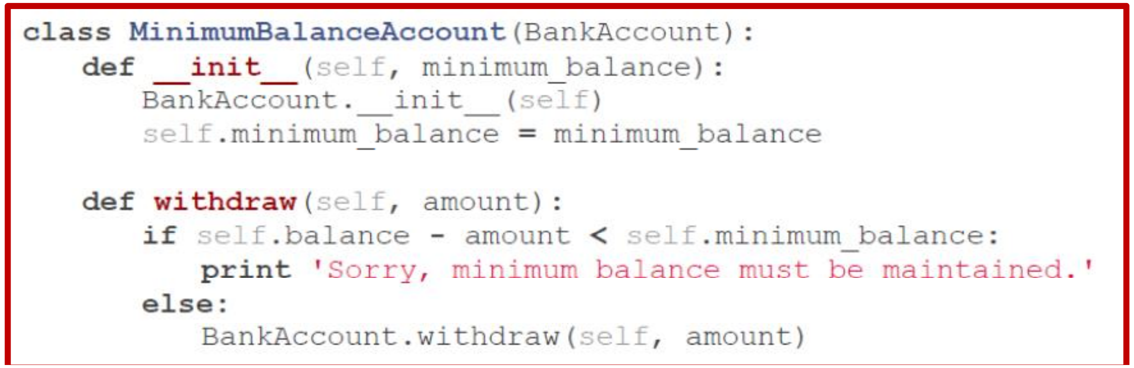
** 유사한 minimum balance account를 만들고 싶다면?

Another Motivational Example (contd)



Classes and Objects

```
class BankAccount:
    def __init__(self):
        self.balance = 0

    def withdraw(self, amount):
        self.balance -= amount
        return self.balance

    def deposit(self, amount):
        self.balance += amount
        return self.balance

>>> a = BankAccount()
>>> b = BankAccount()
>>> a.deposit(100)
100
>>> b.deposit(50)
50
>>> b.withdraw(10)
40
>>> a.withdraw(10)
90
```



```
class MinimumBalanceAccount(BankAccount):
    def __init__(self, minimum_balance):
        BankAccount.__init__(self)
        self.minimum_balance = minimum_balance

    def withdraw(self, amount):
        if self.balance - amount < self.minimum_balance:
            print 'Sorry, minimum balance must be maintained.'
        else:
            BankAccount.withdraw(self, amount)
```

Understanding Inheritance

```
class Foo(object):
   def __init__(self):
      self.health = 100


class SubFoo(Foo):
        pass


testobj = SubFoo()
testobj.health
```

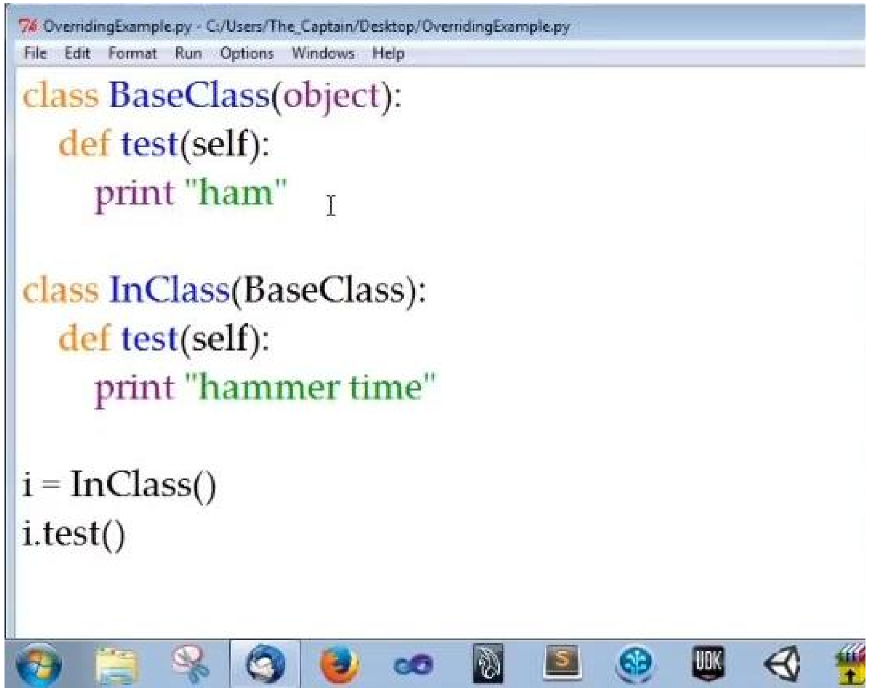
SubFoo class에서는 Foo class의 instance variable 을 inherit 받는다

```
class Foo(object):
   def __init__(self):
      self.health = 100


class SubFoo(Foo):
   def __init__(self):
      self.muscle = 200


testobj = SubFoo()
testobj.health
testobj.muscle
```

SubFoo class에서는 Foo class의 instance variable 을 inherit 받지못한다

```
class Foo(object):
   def   __init__(self):
       self.health = 100


class SubFoo(Foo):
   def   __init__(self):
      super(SubFoo, self)__init__()
      self.muscle = 200



testobj = SubFoo()
testobj.health
testobj.muscle
```

SubFoo class에서는 Foo class의 instance variable 을 inherit 받고 자체적인 instance variable도 선언을 하고 있다

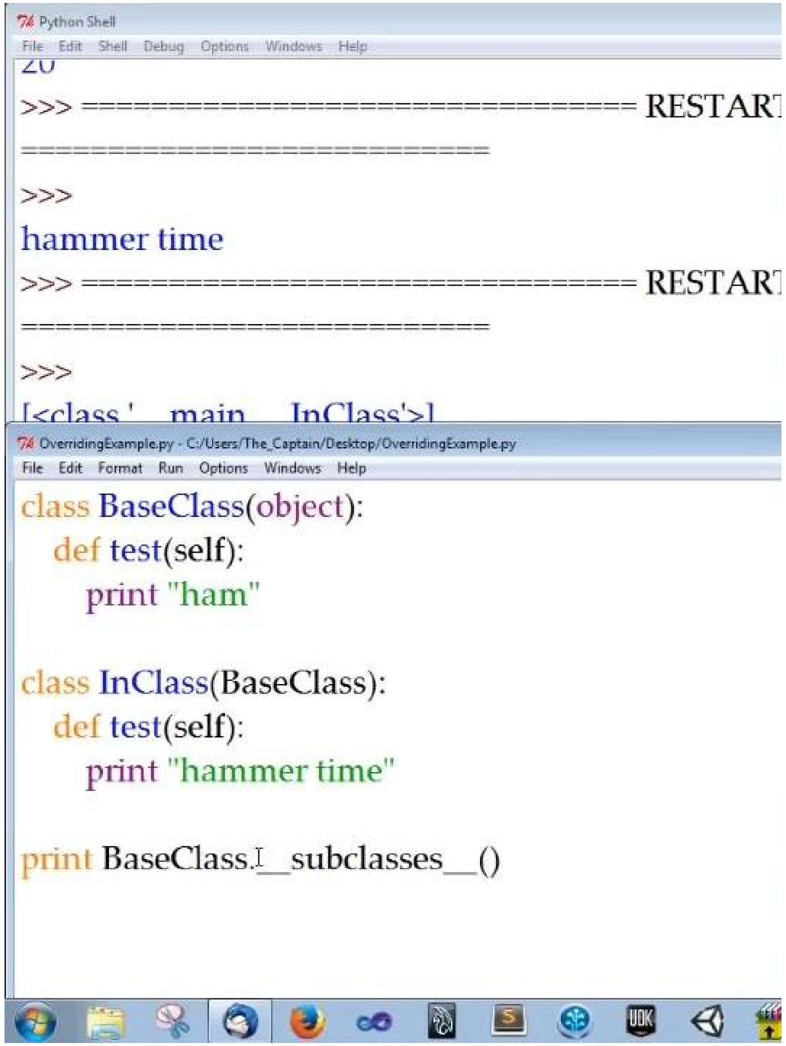

OverridingExample.py - C:/Users/The_Captain/Desktop/OverridingExample.py
File Edit Format Run Options Windows Help

```
class BaseClass(object):
    def test(self):
        print "ham"

class InClass(BaseClass):
    def test(self):
        print "hammer time"

i = InClass()
i.test()
```

```
class BaseClass(object):
    def  __init__(self):
        self.x = 100

class InClass(BaseClass):
    def  __init__(self):
        super(InClass, self)__init__()
        self.y = 200

i = InClass()
print(“Object i’s inherited variable:”,  i.x)
print(“Object i’s locally defined variable:”,  i.y)
```

InClass class에 test()가 있어서
BaseClass class의 test()를 override하므로
BaseClass의 test()는 수행이 안된다.

InClass class에 __init__()가 있어서
BaseClass class의 __init__()를 override하므로
BaseClass의 __init__()는 수행이 안된다.
그러나 super(InClass, self)__init__() 에 의해서
BaseClass의 instance variable을 inherit 받는다

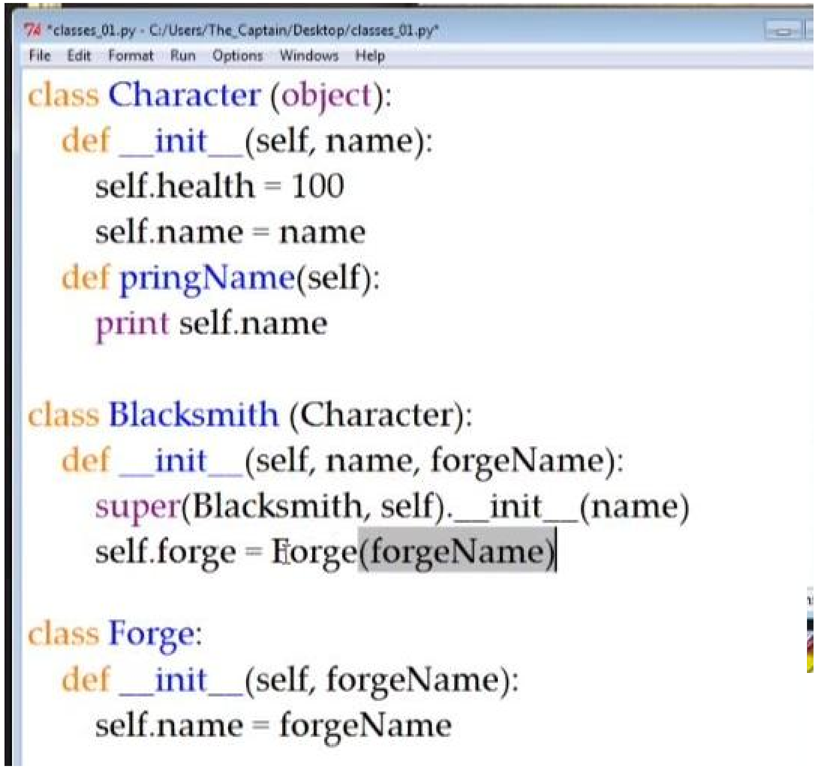

Python Shell
File Edit Shell Debug Options Windows Help

```
20
>>> ================================ RESTART
================================
>>>
hammer time
>>> ================================ RESTART
================================
>>>
[<class '__main__.InClass'>]
```

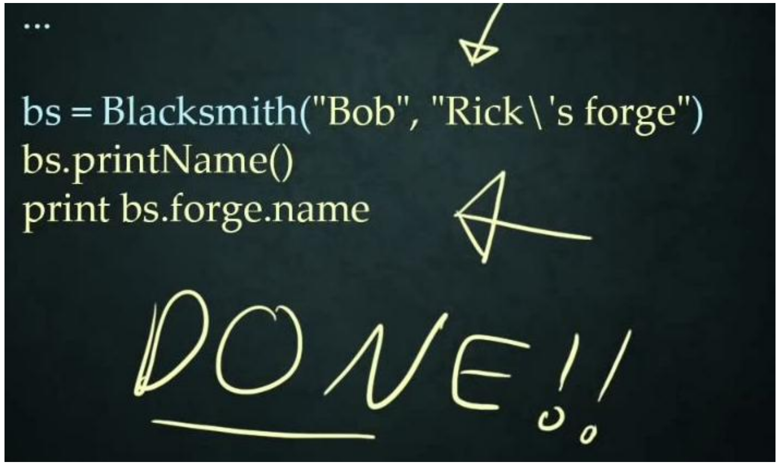
OverridingExample.py - C:/Users/The_Captain/Desktop/OverridingExample.py
File Edit Format Run Options Windows Help

```
class BaseClass(object):
    def test(self):
        print "ham"

class InClass(BaseClass):
    def test(self):
        print "hammer time"

print BaseClass.__subclasses__()
```

InClass class에서는 BaseClass class의 test()을 inherit 받지않고 같은이름의 test()를 locally define했다


classes_01.py - C:/Users/The_Captain/Desktop/classes_01.py
File Edit Format Run Options Windows Help

```
class Character (object):
    def __init__(self, name):
        self.health = 100
        self.name = name
    def pringName(self):
        print self.name

class Blacksmith (Character):
    def __init__(self, name, forgeName):
        super(Blacksmith, self).__init__(name)
        self.forge = Forge(forgeName)

class Forge:
    def __init__(self, forgeName):
        self.name = forgeName
```

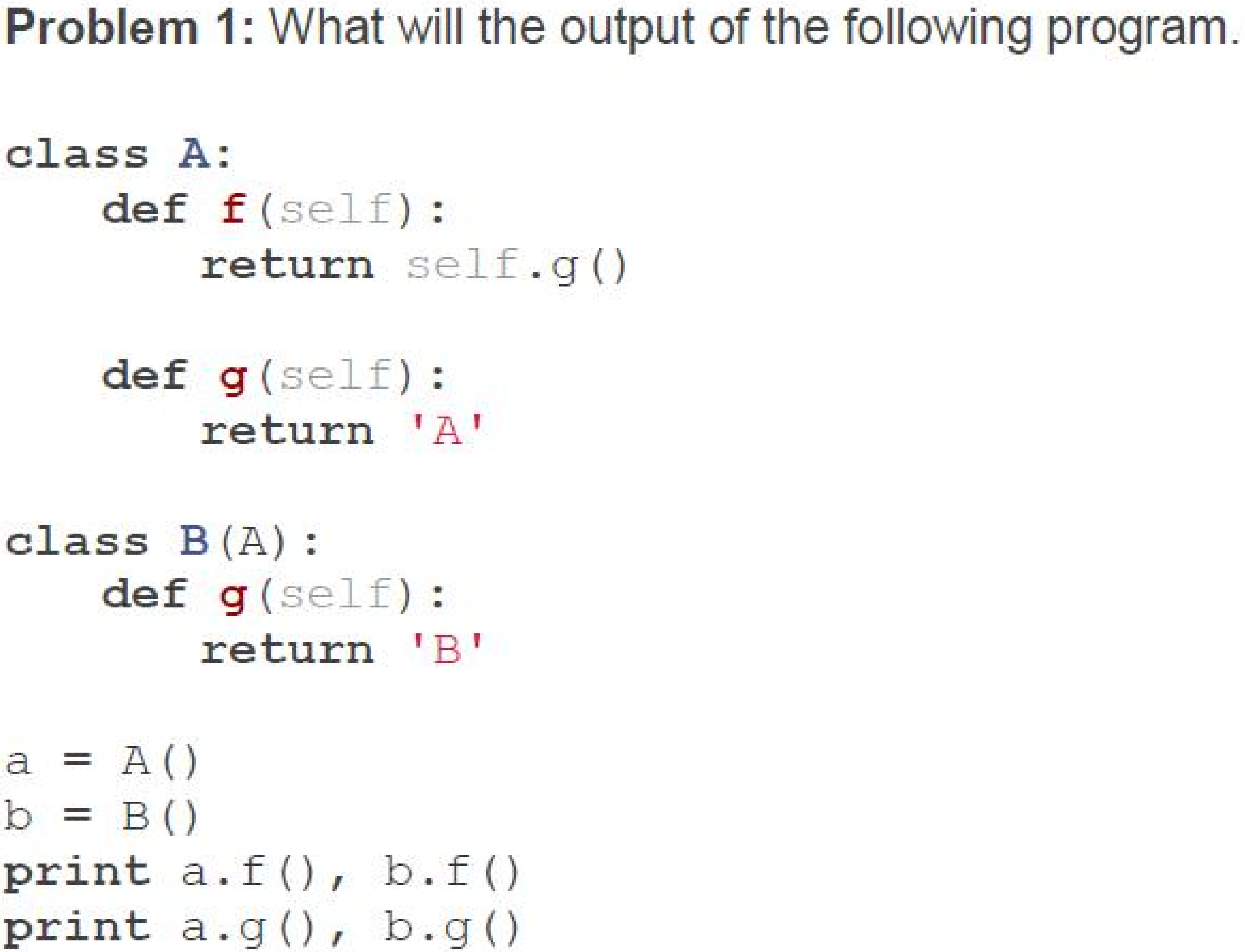


```
...
bs = Blacksmith("Bob", "Rick\'s forge")
bs.printName()
print bs.forge.name
```

Superclass로 “object”를 선언할때

```
class Foo(object):
    def __init__(self):
            self.health = 100

Class SubFoo(Foo):
        pass

testobj = SubFoo()
testobj.health
```

```
class Foo:
    def __init__(self):
            self.health = 100

Class SubFoo(Foo):
        pass

testobj = SubFoo()
testobj.health
```

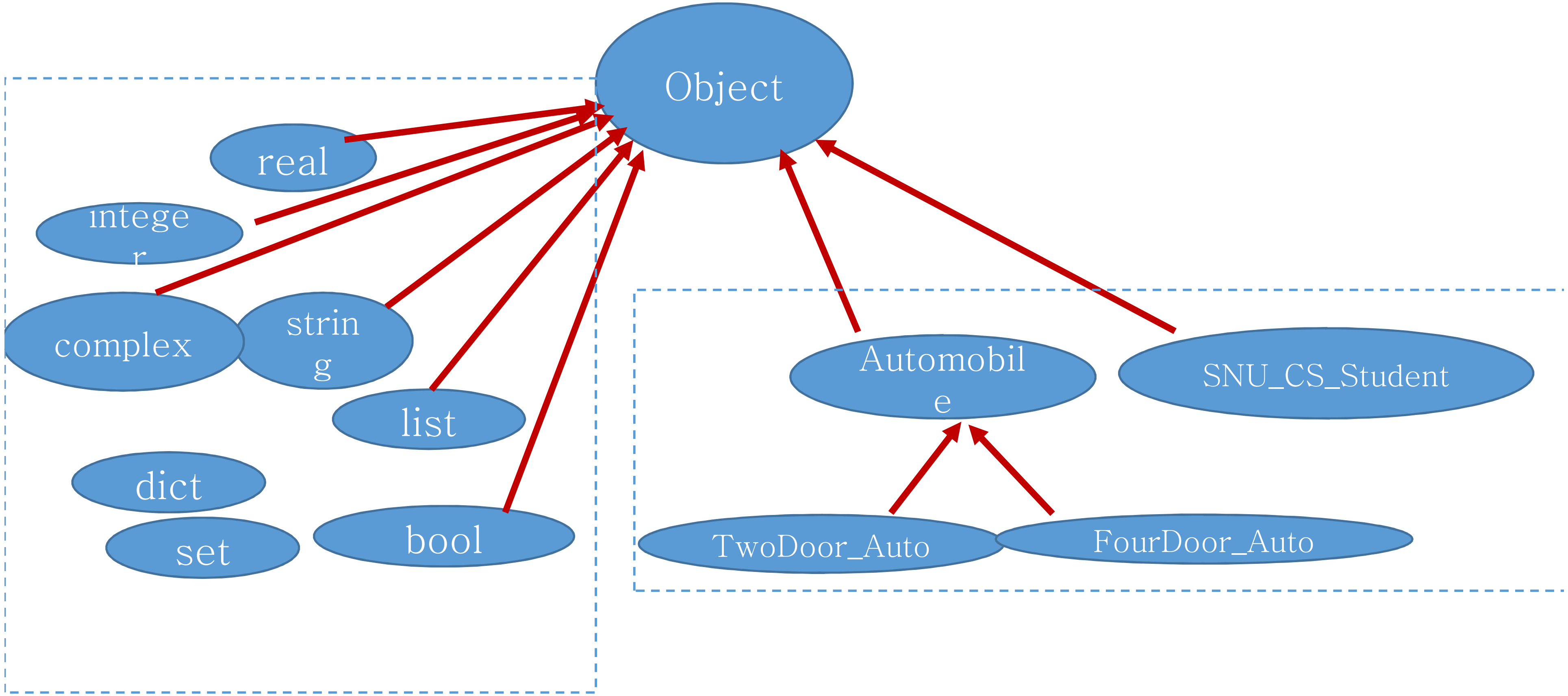


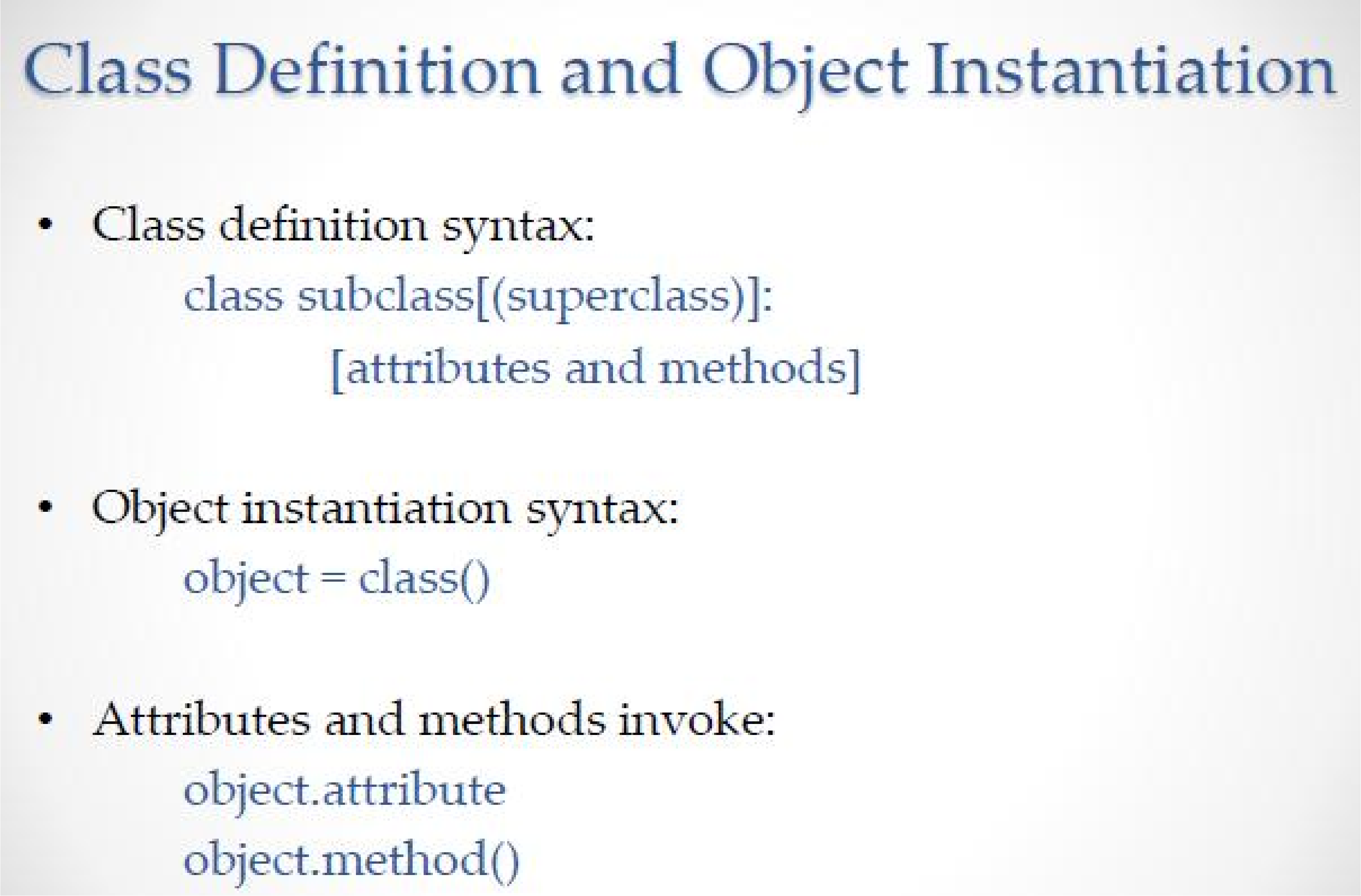
Problem 1: What will the output of the following program.

```python
class A:
    def f(self):
        return self.g()

    def g(self):
        return 'A'

class B(A):
    def g(self):
        return 'B'

a = A()
b = B()
print a.f(), b.f()
print a.g(), b.g()
```

A More Formal Way of Python OO Programming

Python의 type system
(class structure)





Class Definition and Object Instantiation

- Class definition syntax:

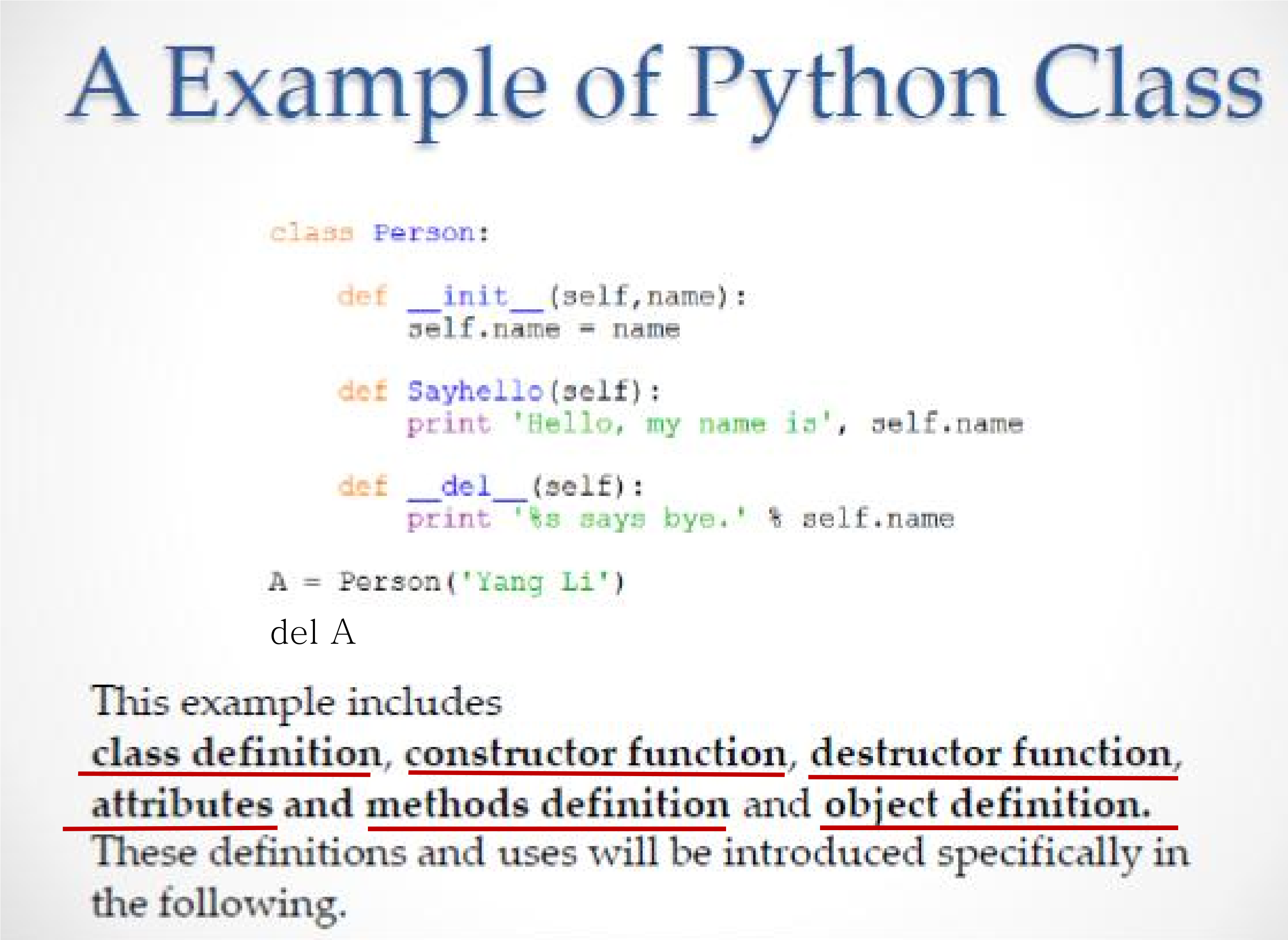
```
class subclass[(superclass)]:
        [attributes and methods]
```

- Object instantiation syntax:

```
object = class()
```

- Attributes and methods invoke:

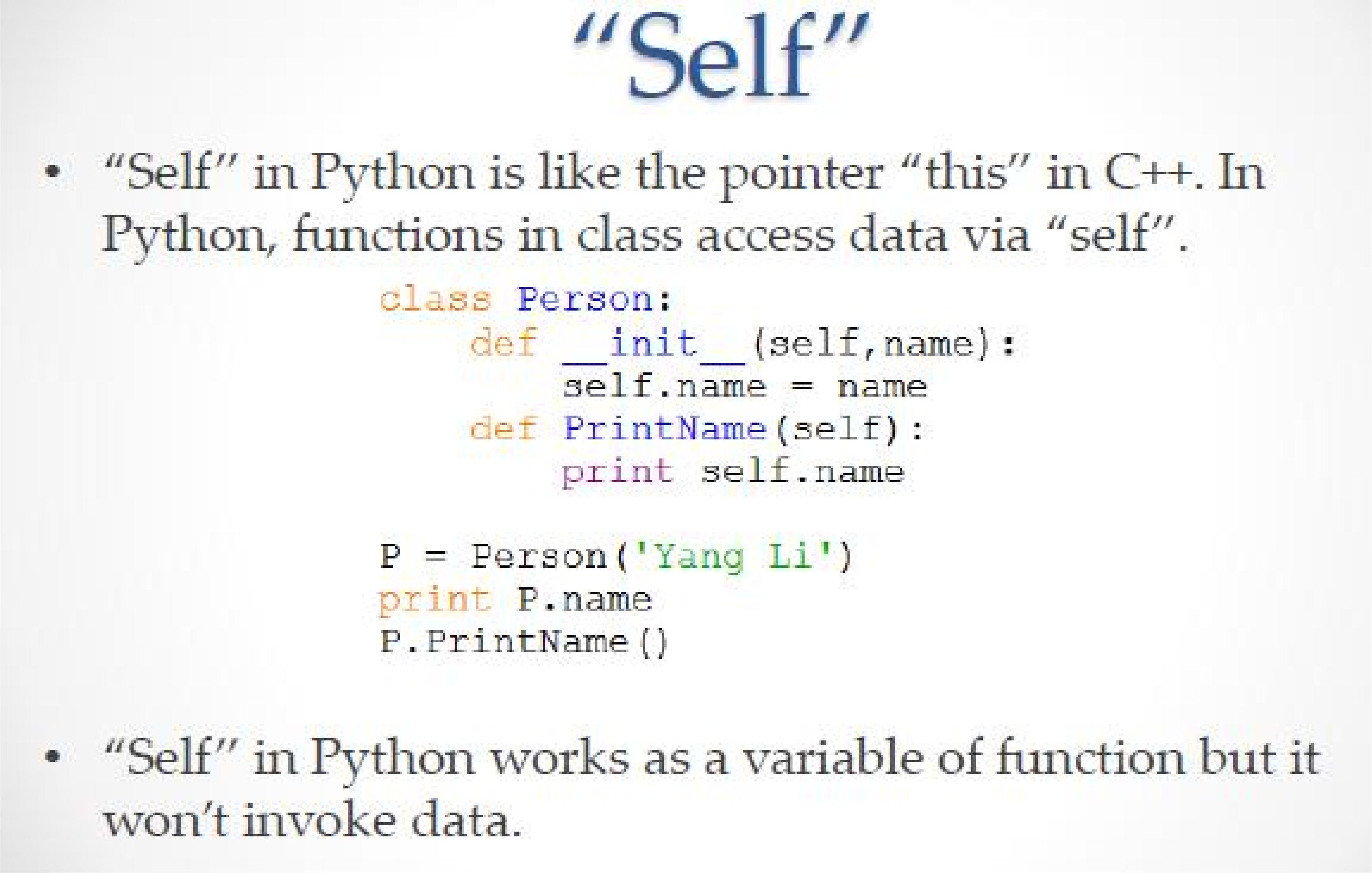
```
object.attribute
object.method()
```



A Example of Python Class

```
class Person:

    def __init__(self,name):
        self.name = name

    def Sayhello(self):
        print 'Hello, my name is', self.name

    def __del__(self):
        print '%s says bye.' % self.name

A = Person('Yang Li')
del A
```

This example includes
class definition, **constructor function**, **destructor function**, **attributes and methods definition** and **object definition.** These definitions and uses will be introduced specifically in the following.

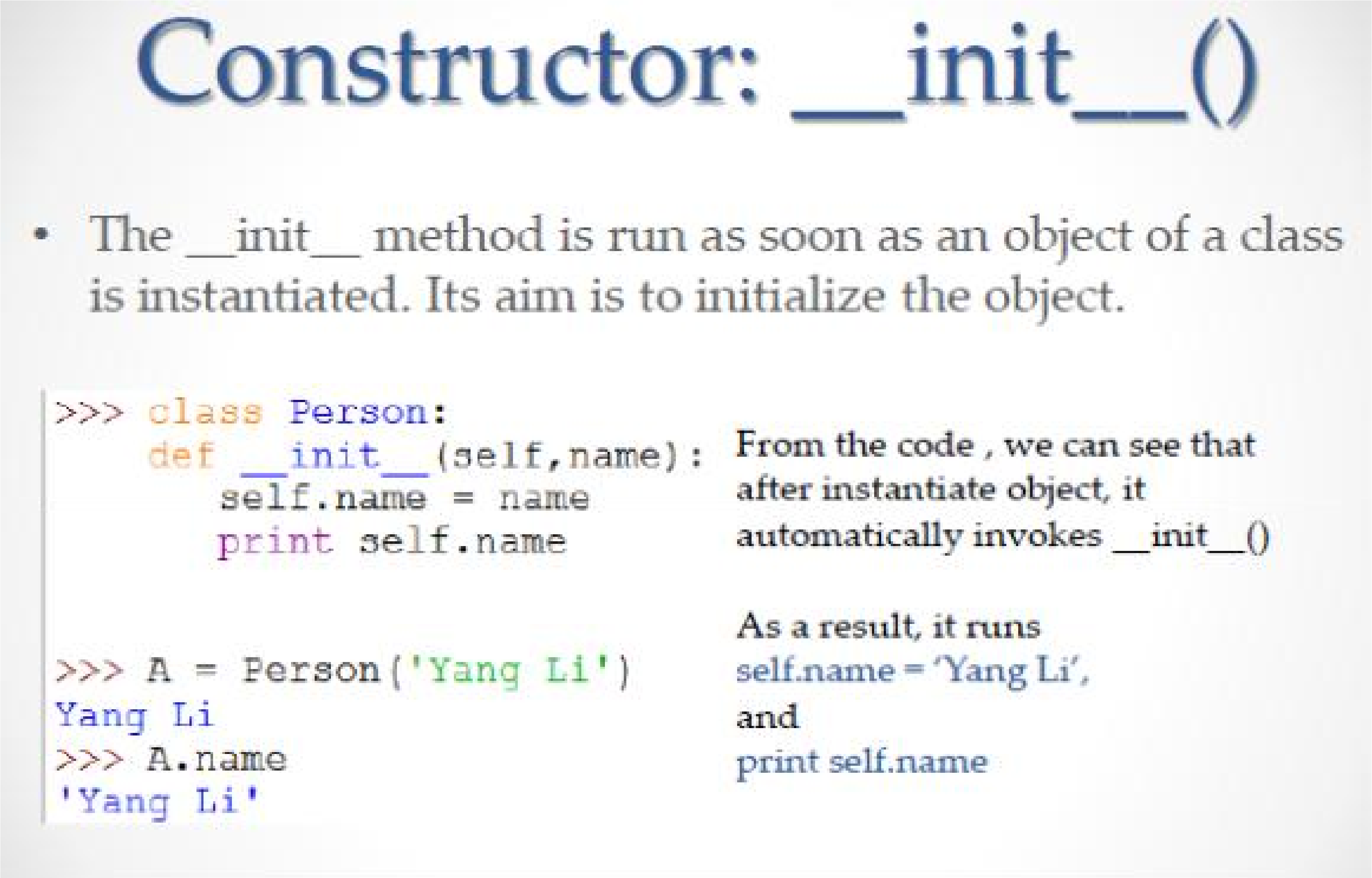


“Self”

- “Self” in Python is like the pointer “this” in C++. In Python, functions in class access data via “self”.

```
class Person:
    def __init__(self,name):
        self.name = name
    def PrintName(self):
        print self.name

P = Person('Yang Li')
print P.name
P.PrintName()
```

- “Self” in Python works as a variable of function but it won't invoke data.



Constructor: __init__()

- The __init__ method is run as soon as an object of a class is instantiated. Its aim is to initialize the object.

```
>>> class Person:
    def __init__(self,name):
        self.name = name
        print self.name


>>> A = Person('Yang Li')
Yang Li
>>> A.name
'Yang Li'
```

From the code , we can see that after instantiate object, it automatically invokes __init__()

As a result, it runs self.name = 'Yang Li', and print self.name

Instance Variable vs Class Variable Instance Method vs Class Method

- class SNU_Student

- Instance Variable
  - Instance에 해당하는 variable
  - Name, Student_ID, Courses, GPA

- Class Variable
  - Class의 모든 Instance에 해당하는 variable
  - University_name

- Instance Method
  - Instance에 적용되는 method
  - 학생이름을 input으로 학점을 retur하는 method ➔ gpa(name)
  - 학생이름을 input으로 이수한 과목들을 retur하는 method ➔ taken_course(name)

- Class Method의 예
  - Class에 적용되는 method
  - SNU_Student class에 있는 전체학생수를 return하는 num_students()
  - SNU_Student class에 있는 전체학생수들인 평균GPA를 return하는 avg_gpa()

Instance Variable vs Class Variable

```
Class AAA_Club;
    club_name = “American Auto Association”

    def __init__ (self, name, num):
        self.name = name
        self.member_id = num
```

```
John = AAA_Club(“John”, 123)
Bob  = AAA_Club(“Bob”, 124)

print(AAA_Club.club_name)
print(John.name)
print(Bob.member_id)
```

Instance Method vs Class Method

- Example

```
class A(object):
    def foo(self):
        print ('executing foo')

    @classmethod
    def class_foo(cls):
         print ('executing class_foo‘)
```

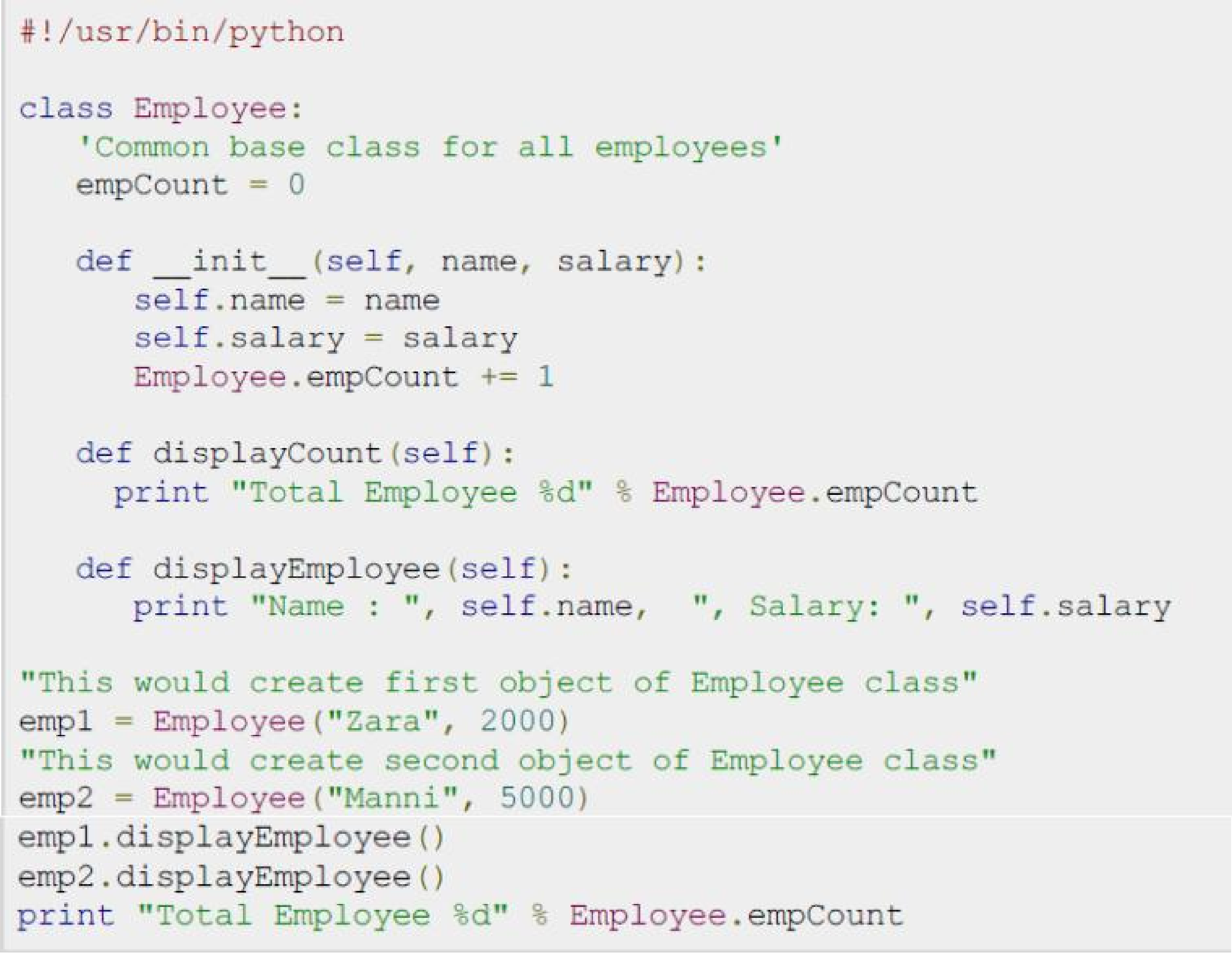
- Example

```
a = A()
#A.foo()        # Error
A.class_foo()   # class-method class_foo()를 class A에서 call
a.foo()         # instance-method foo()를 instance a에서 call
a.class_foo()   # class-method class_foo()를 instance a에서 call
```

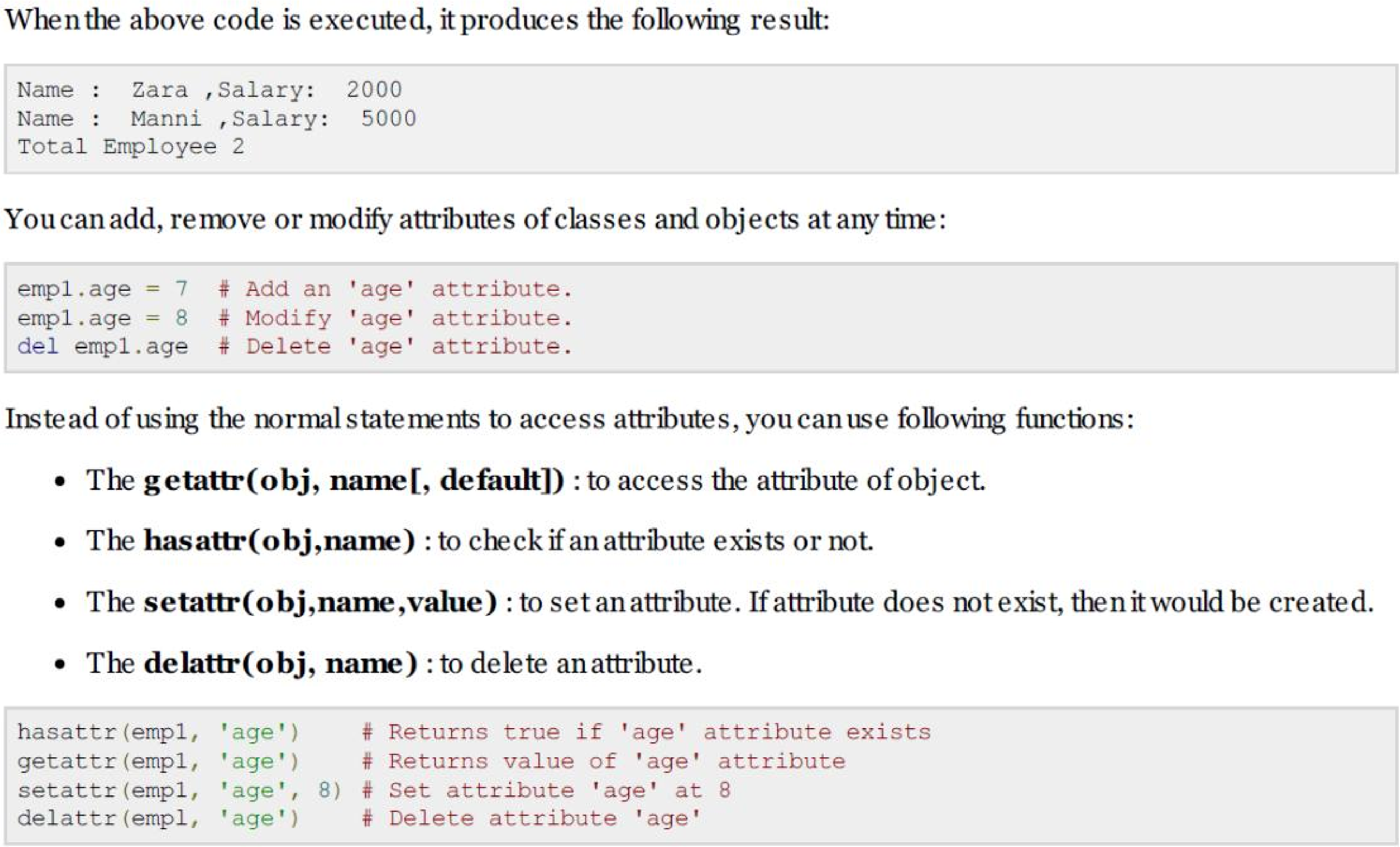
- Result

```
executing class_foo
executing foo
executing class_foo
```

Employee Class



```
#!/usr/bin/python

class Employee:
   'Common base class for all employees'
   empCount = 0

   def __init__(self, name, salary):
      self.name = name
      self.salary = salary
      Employee.empCount += 1

   def displayCount(self):
     print "Total Employee %d" % Employee.empCount

   def displayEmployee(self):
      print "Name : ", self.name,  ", Salary: ", self.salary

"This would create first object of Employee class"
emp1 = Employee("Zara", 2000)
"This would create second object of Employee class"
emp2 = Employee("Manni", 5000)
emp1.displayEmployee()
emp2.displayEmployee()
print "Total Employee %d" % Employee.empCount
```



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

```
Name :  Zara ,Salary:  2000
Name :  Manni ,Salary:  5000
Total Employee 2
```

You can add, remove or modify attributes of classes and objects at any time:

```
emp1.age = 7  # Add an 'age' attribute.
emp1.age = 8  # Modify 'age' attribute.
del emp1.age  # Delete 'age' attribute.
```

Instead of using the normal statements to access attributes, you can use following functions:

- The **getattr(obj, name[, default])** : to access the attribute of object.
- The **hasattr(obj,name)** : to check if an attribute exists or not.
- The **setattr(obj,name,value)** : to set an attribute. If attribute does not exist, then it would be created.
- The **delattr(obj, name)** : to delete an attribute.

```
hasattr(emp1, 'age')    # Returns true if 'age' attribute exists
getattr(emp1, 'age')    # Returns value of 'age' attribute
setattr(emp1, 'age', 8) # Set attribute 'age' at 8
delattr(empl, 'age')    # Delete attribute 'age'
```

hasattr(), getattr(), setattr(), delattr() 들은 Python Built-in functions

Built-In Class Attributes:

Every Python class keeps following built-in attributes and they can be accessed using dot operator like any other attribute:

- **__dict__** : Dictionary containing the class's namespace.
- **__doc__** : Class documentation string or None if undefined.
- **__name__**: Class name.
- **__module__**: Module name in which the class is defined. This attribute is "__main__" in interactive mode.
- **__bases__** : A possibly empty tuple containing the base classes, in the order of their occurrence in the base class list.

For the above class let's try to access all these attributes:

```
#!/usr/bin/python

class Employee:
   'Common base class for all employees'
   empCount = 0

   def __init__(self, name, salary):
      self.name = name
      self.salary = salary
      Employee.empCount += 1

   def displayCount(self):
     print "Total Employee %d" % Employee.empCount

   def displayEmployee(self):
      print "Name : ", self.name,  ", Salary: ", self.salary
print "Employee.__doc__:", Employee.__doc__
print "Employee.__name__:", Employee.__name__
print "Employee.__module__:", Employee.__module__
print "Employee.__bases__:", Employee.__bases__
print "Employee.__dict__:", Employee.__dict__
```

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

```
Employee.__doc__: Common base class for all employees
Employee.__name__: Employee
Employee.__module__: __main__
Employee.__bases__: ()
Employee.__dict__: {'__module__': '__main__', 'displayCount':
<function displayCount at 0xb7c84994>, 'empCount': 2,
'displayEmployee': <function displayEmployee at 0xb7c8441c>,
'__doc__': 'Common base class for all employees',
'__init__': <function __init__ at 0xb7c846bc>}
```

Destroying Objects (Garbage Collection):

Python deletes unneeded objects (built-in types or class instances) automatically to free memory space. The process by which Python periodically reclaims blocks of memory that no longer are in use is termed garbage collection.

Python's garbage collector runs during program execution and is triggered when an object's reference count reaches zero. An object's reference count changes as the number of aliases that point to it changes.

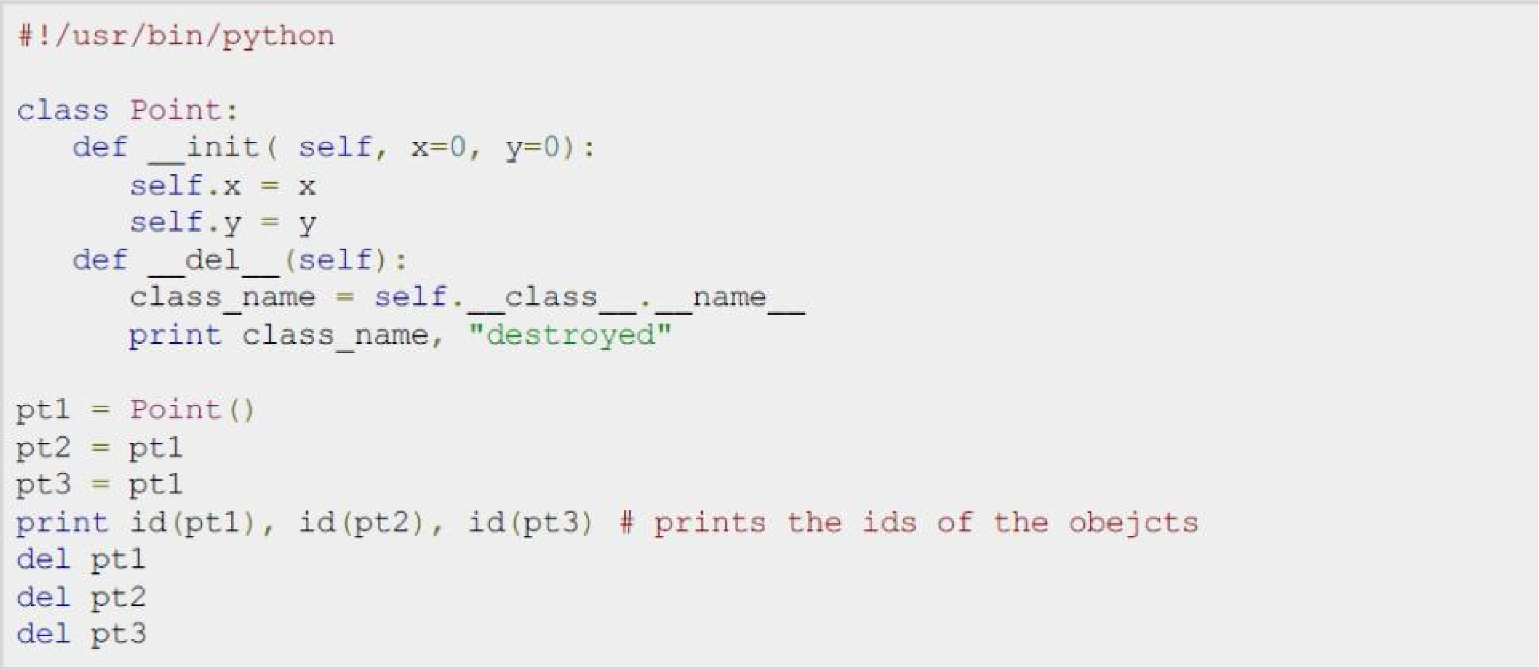
An object's reference count increases when it's assigned a new name or placed in a container (list, tuple or dictionary). The object's reference count decreases when it's deleted with *del*, its reference is reassigned, or its reference goes out of scope. When an object's reference count reaches zero, Python collects it automatically.

```
a = 40      # Create object <40>
b = a       # Increase ref. count  of <40>
c = [b]     # Increase ref. count  of <40>

del a       # Decrease ref. count  of <40>
b = 100     # Decrease ref. count  of <40>
c[0] = -1   # Decrease ref. count  of <40>
```

You normally won't notice when the garbage collector destroys an orphaned instance and reclaims its space. But a class can implement the special method __del__(), called a destructor, that is invoked when the instance is about to be destroyed. This method might be used to clean up any nonmemory resources used by an instance.

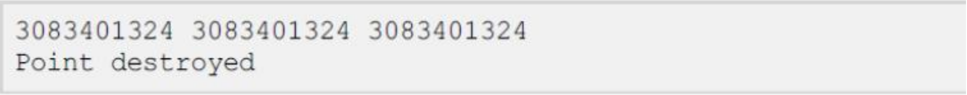
Example:

This __del__() destructor prints the class name of an instance that is about to be destroyed:



```
#!/usr/bin/python

class Point:
   def __init( self, x=0, y=0):
      self.x = x
      self.y = y
   def __del__(self):
      class_name = self.__class__.__name__
      print class_name, "destroyed"

pt1 = Point()
pt2 = pt1
pt3 = pt1
print id(pt1), id(pt2), id(pt3) # prints the ids of the obejcts
del pt1
del pt2
del pt3
```

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



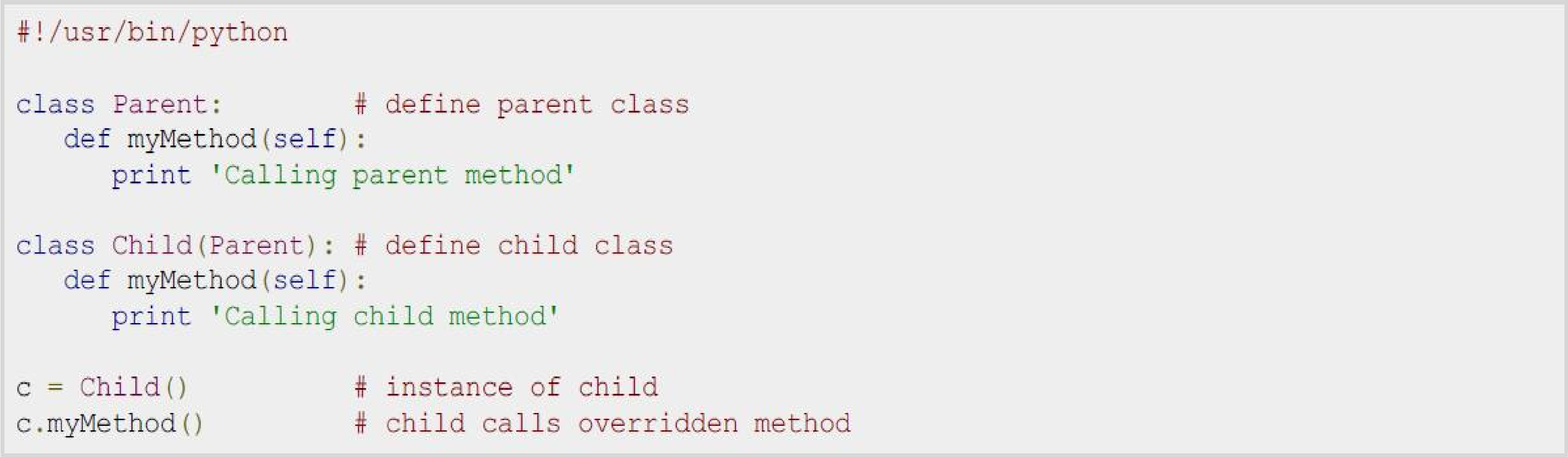
```
3083401324 3083401324 3083401324
Point destroyed
```

원래 object class에 있는 del 이 있는데 Point class에서 __del__()를 locally define했으므로 locally define한 __del__()이 call된다

Reusing Methods and Overriding Methods

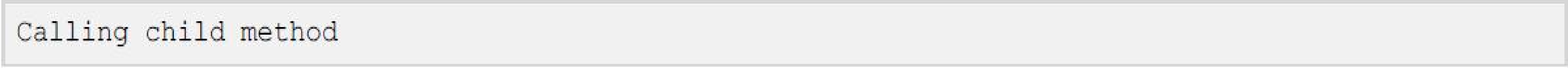
You can always override your parent class methods. One reason for overriding parent's methods is because you may want special or different functionality in your subclass.

Example:



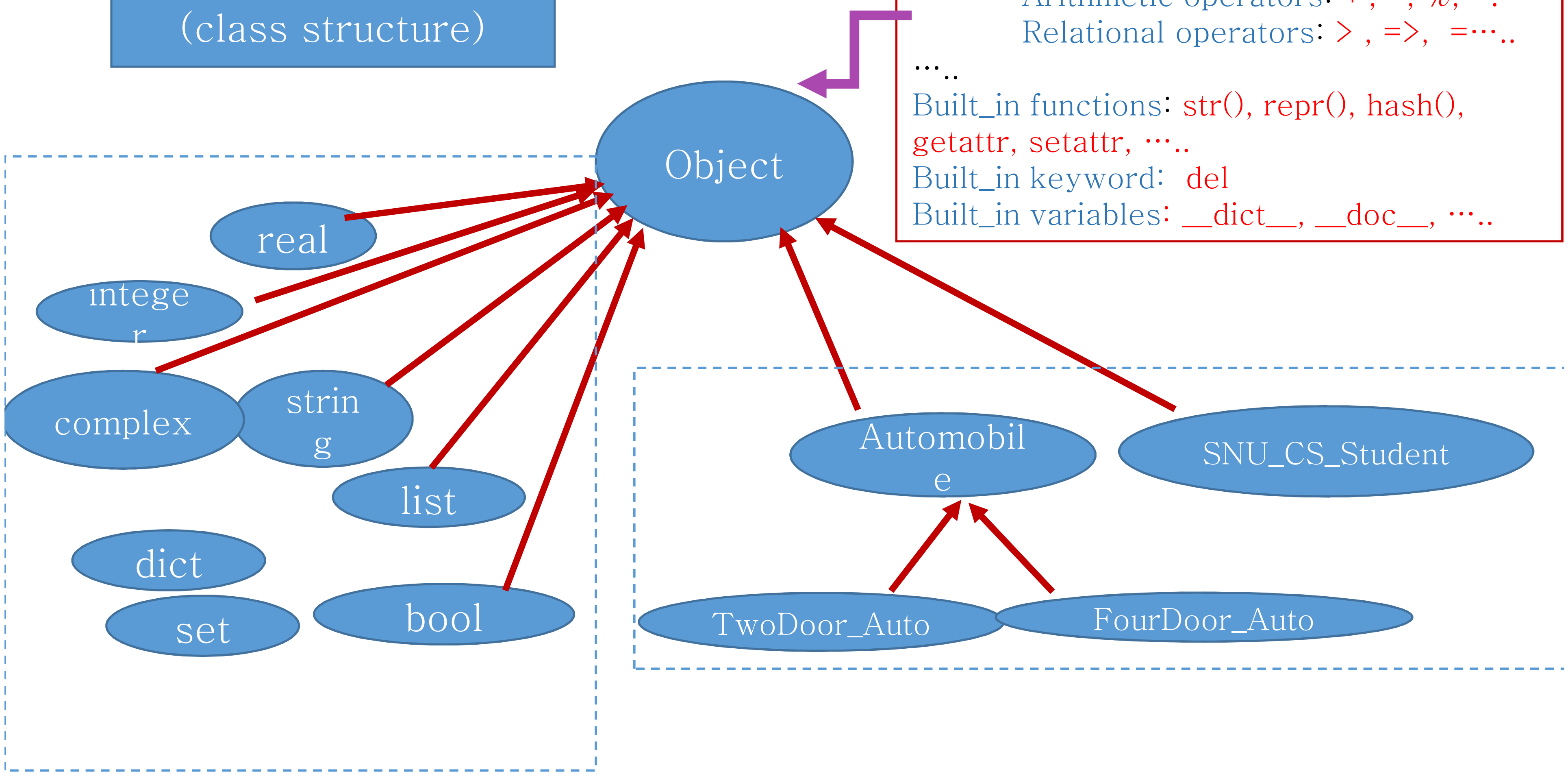
```
#!/usr/bin/python

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
```

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



```
Calling child method
```

Python의 type system (class structure)

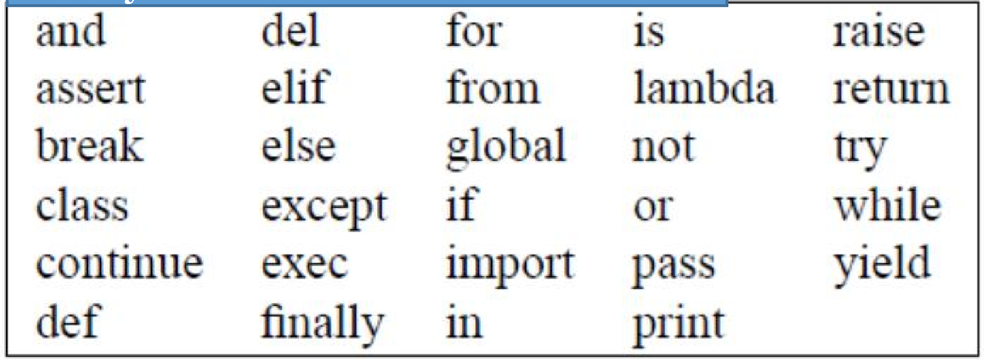


Primitive Operators such as
Arithmetic operators: + , −, %,….
Relational operators: > , =>, =…..
…..
Built_in functions: str(), repr(), hash(), getattr, setattr, …..
Built_in keyword: del
Built_in variables: __dict__, __doc__, …..

Python의 Primitive operators, Built_in functions, Predefined variables들이 user_defined classes들에 내려와서는 해당 class에 맞추어 작동을 하기위해 user_defined class의 내부에서 __wyx__로 redefine 된다!

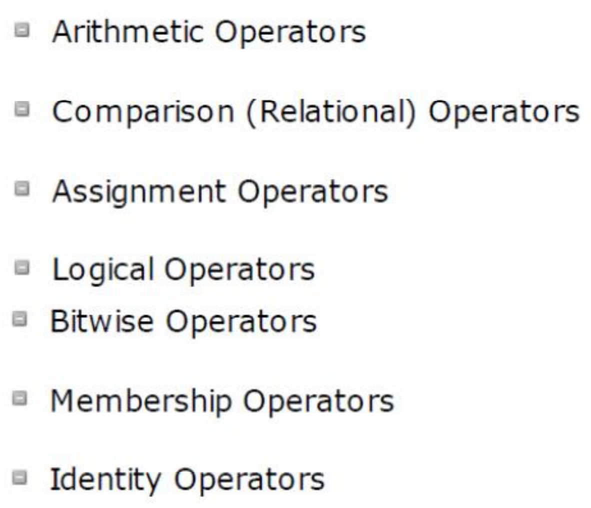
참고자료

Python Reserved Words

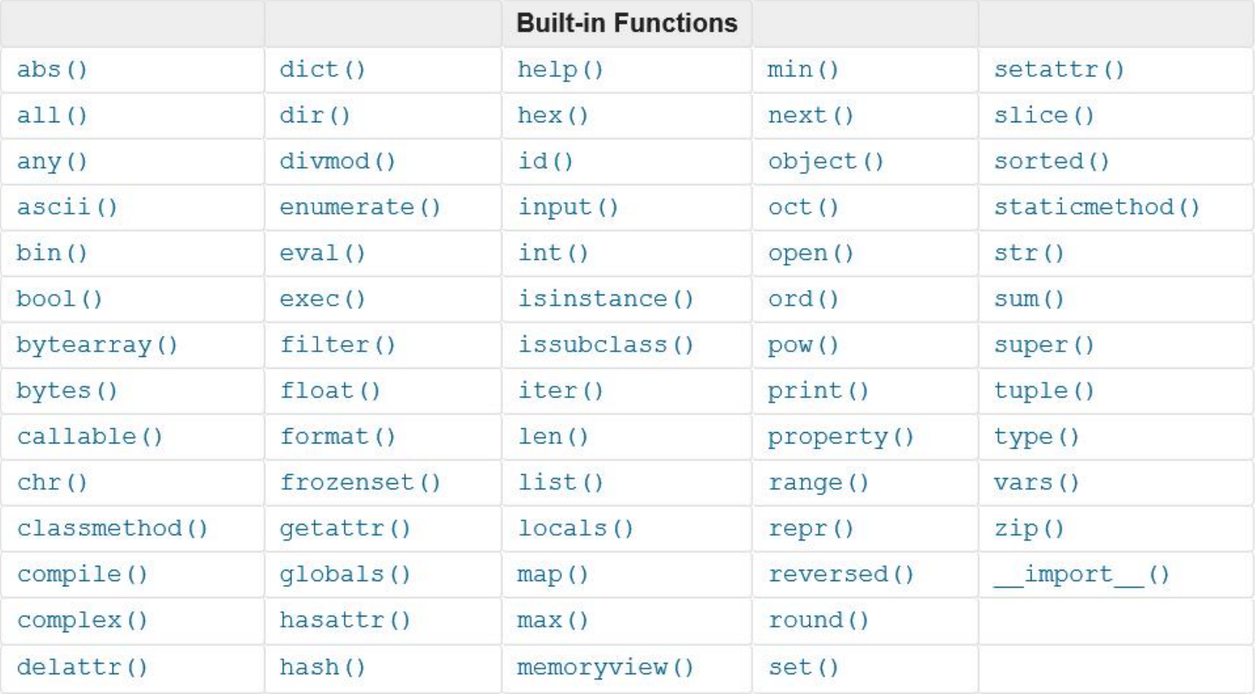


and	del	for	is	raise
assert	elif	from	lambda	return
break	else	global	not	try
class	except	if	or	while
continue	exec	import	pass	yield
def	finally	in	print	

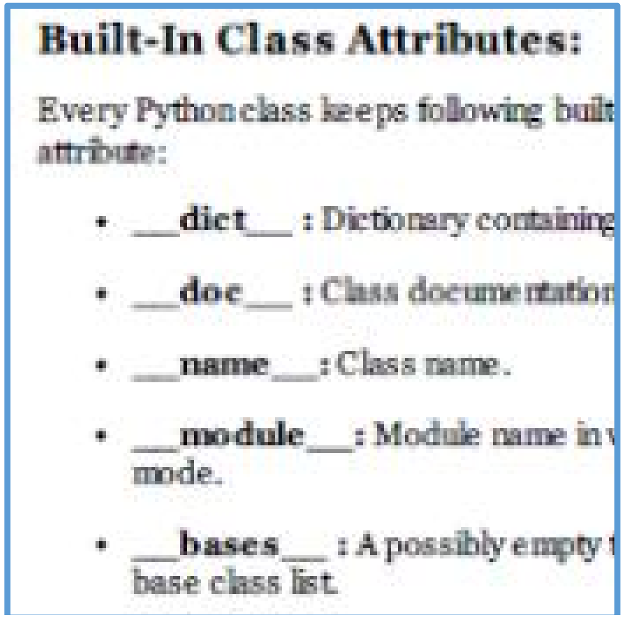
Python Operators



- Arithmetic Operators
- Comparison (Relational) Operators
- Assignment Operators
- Logical Operators
- Bitwise Operators
- Membership Operators
- Identity Operators



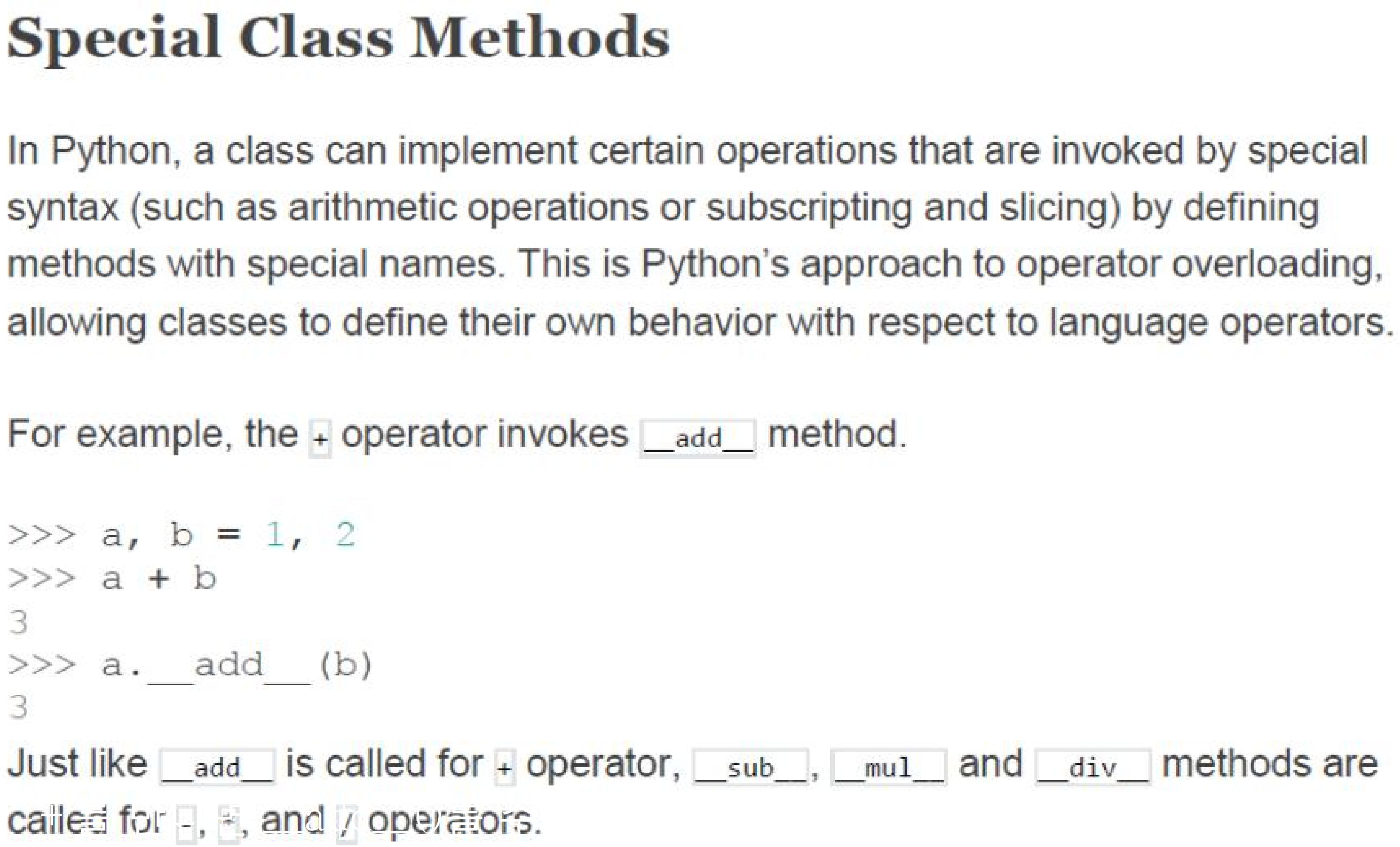
		Built-in Functions		
abs()	dict()	help()	min()	setattr()
all()	dir()	hex()	next()	slice()
any()	divmod()	id()	object()	sorted()
ascii()	enumerate()	input()	oct()	staticmethod()
bin()	eval()	int()	open()	str()
bool()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	
delattr()	hash()	memoryview()	set()	



Built-In Class Attributes:

Every Python class keeps following buil
attribute:

- **__dict__** : Dictionary containing
- **__doc__** : Class documentation
- **__name__**: Class name.
- **__module__**: Module name in
mode.
- **__bases__** : A possibly empty
base class list.



Special Class Methods

In Python, 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. This is Python's approach to operator overloading, allowing classes to define their own behavior with respect to language operators.

For example, the `+` operator invokes `__add__` method.

```
>>> a, b = 1, 2
>>> a + b
3
>>> a.__add__(b)
3
```

Just like `__add__` is called for `+` operator, `__sub__`, `__mul__` and `__div__` methods are called for `-`, `*`, and `/` operators.

-를 만나면 __sub__()를 수행 ……

*를 만나면 __mul__()를 수행 ……

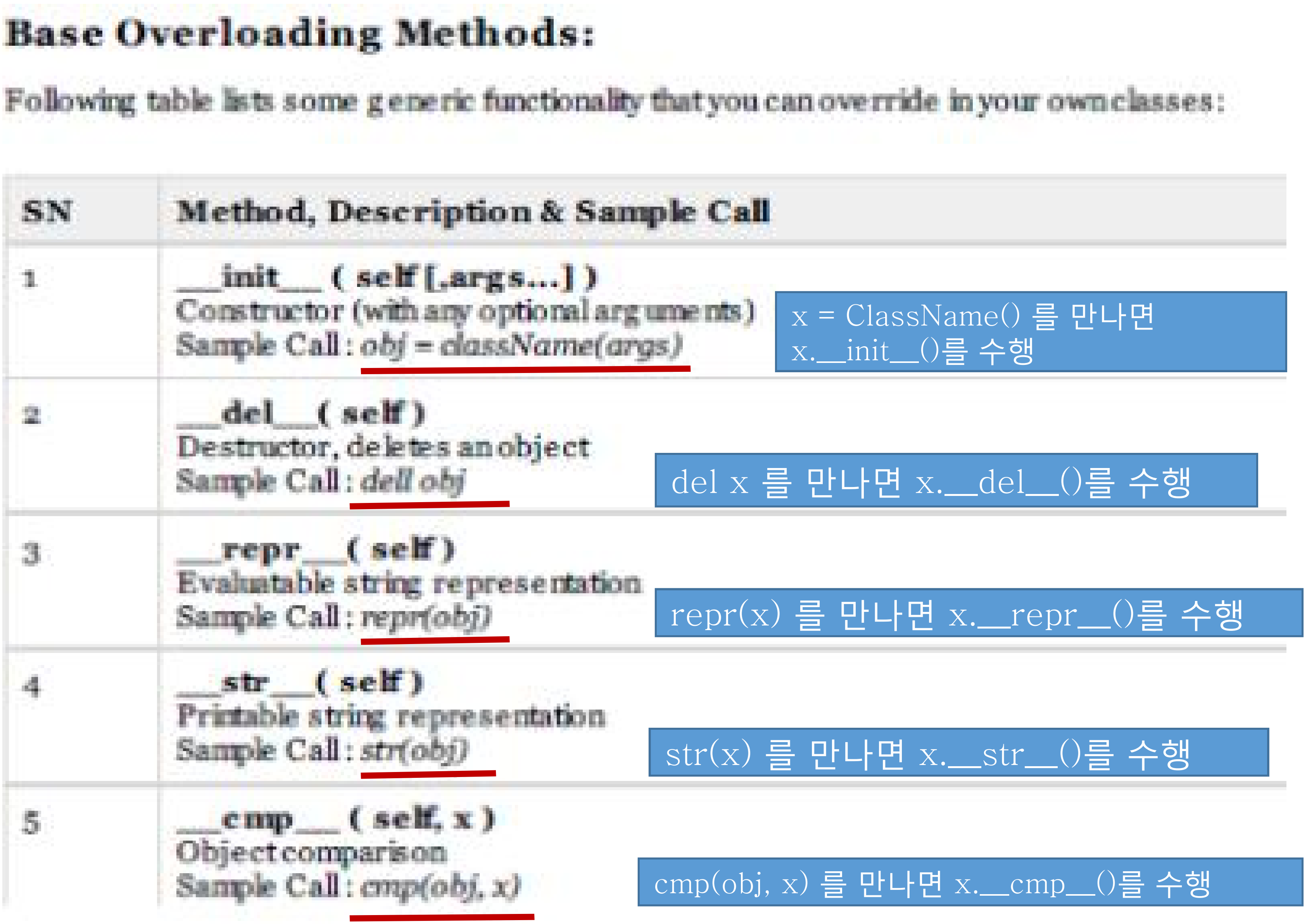
/를 만나면 __div__()를 수행

수행…

>를 만나면 __gt__()를 수행…

=>를 만나면 __ge__()를 수행…

<를 만나면 __lt__()를



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 : *dell 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)*

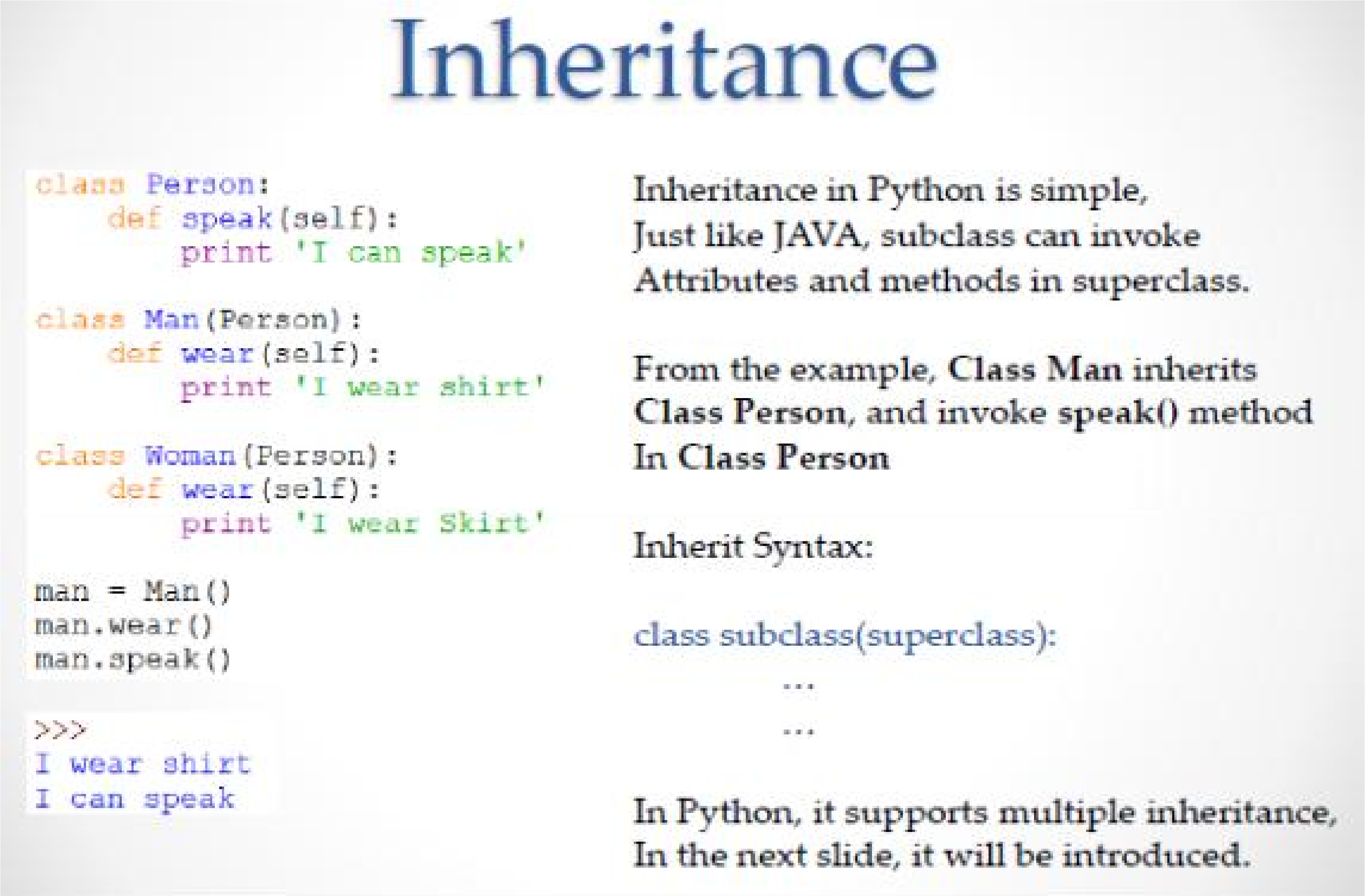
x = ClassName() 를 만나면
x.__init__()를 수행

del x 를 만나면 x.__del__()를 수행

repr(x) 를 만나면 x.__repr__()를 수행

str(x) 를 만나면 x.__str__()를 수행

cmp(obj, x) 를 만나면 x.__cmp__()를 수행



Inheritance

```
class Person:
    def speak(self):
        print 'I can speak'

class Man(Person):
    def wear(self):
        print 'I wear shirt'

class Woman(Person):
    def wear(self):
        print 'I wear Skirt'

man = Man()
man.wear()
man.speak()
```

```
>>>
I wear shirt
I can speak
```

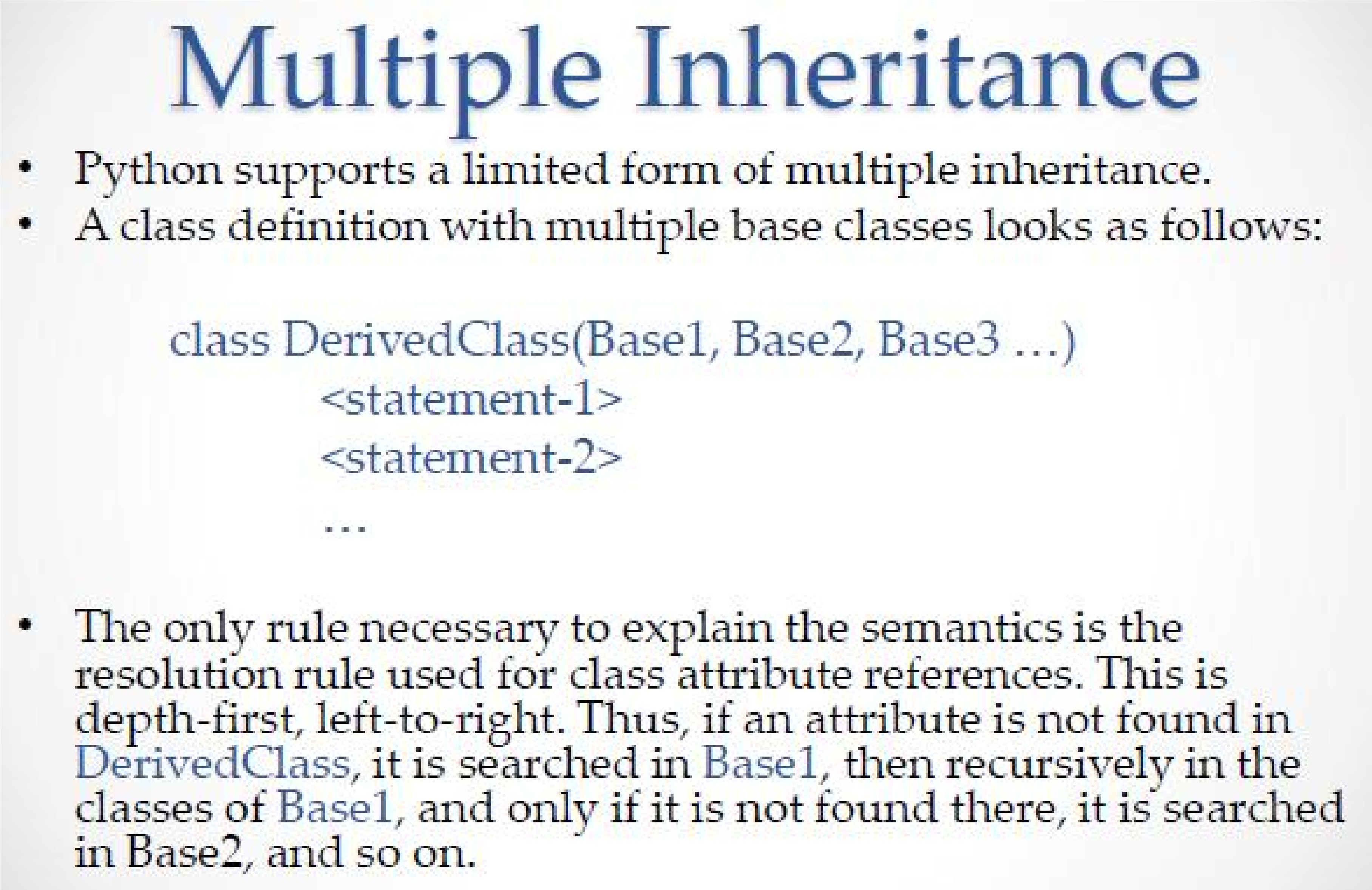
Inheritance in Python is simple,
Just like JAVA, subclass can invoke
Attributes and methods in superclass.

From the example, **Class Man** inherits
Class Person, and invoke **speak()** method
In **Class Person**

Inherit Syntax:

```
class subclass(superclass):
        ...
        ...
```

In Python, it supports multiple inheritance,
In the next slide, it will be introduced.

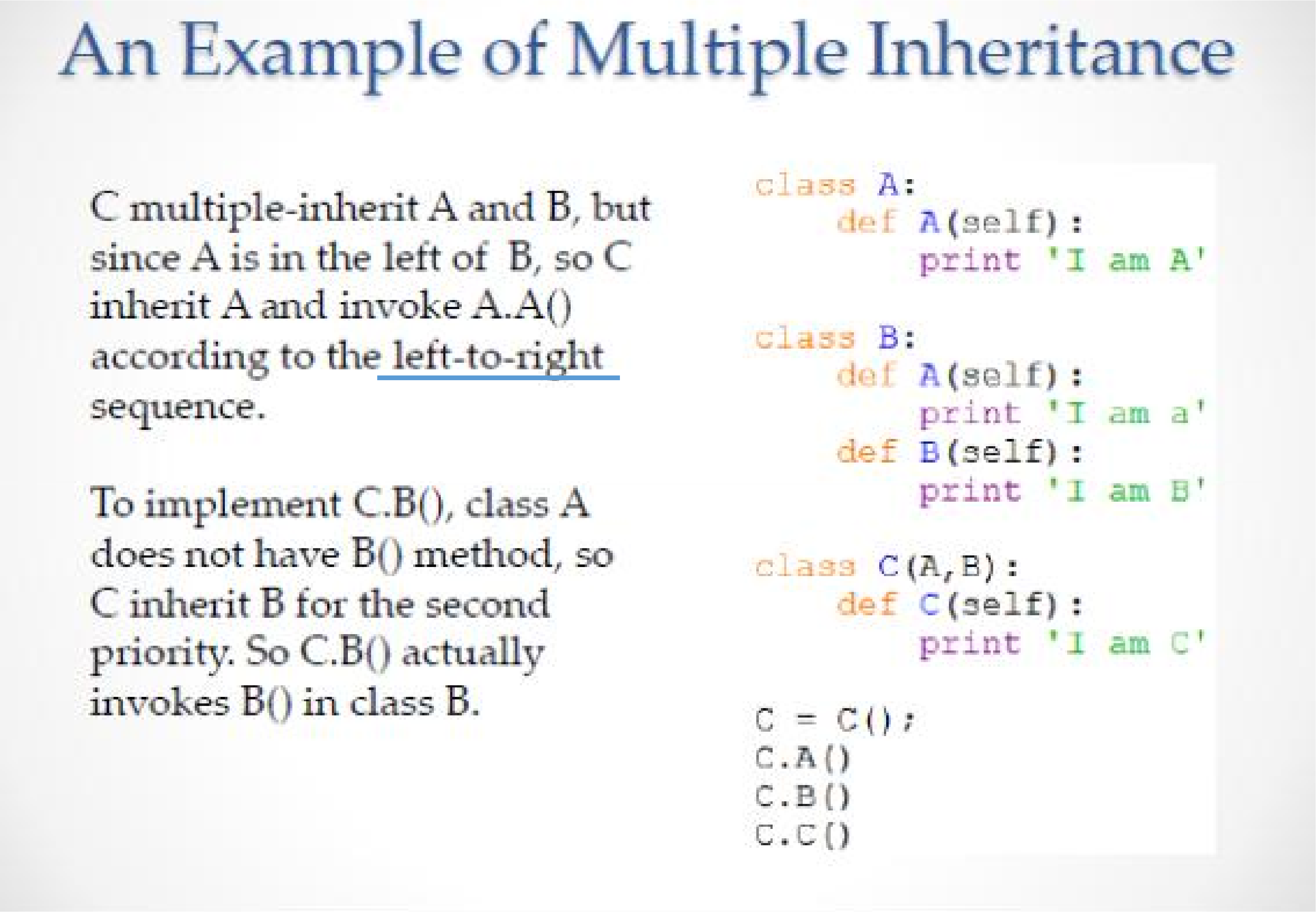


Multiple Inheritance

- Python supports a limited form of multiple inheritance.
- A class definition with multiple base classes looks as follows:

```
class DerivedClass(Base1, Base2, Base3 …)
        <statement-1>
        <statement-2>
        …
```

- The only rule necessary to explain the semantics is the resolution rule used for class attribute references. This is depth-first, left-to-right. Thus, if an attribute is not found in DerivedClass, it is searched in Base1, then recursively in the classes of Base1, and only if it is not found there, it is searched in Base2, and so on.



An Example of Multiple Inheritance

C multiple-inherit A and B, but since A is in the left of B, so C inherit A and invoke A.A() according to the left-to-right sequence.

To implement C.B(), class A does not have B() method, so C inherit B for the second priority. So C.B() actually invokes B() in class B.

```
class A:
    def A(self):
        print 'I am A'

class B:
    def A(self):
        print 'I am a'
    def B(self):
        print 'I am B'

class C(A,B):
    def C(self):
        print 'I am C'

C = C();
C.A()
C.B()
C.C()
```

Class Inheritance:

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.

Syntax:

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

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

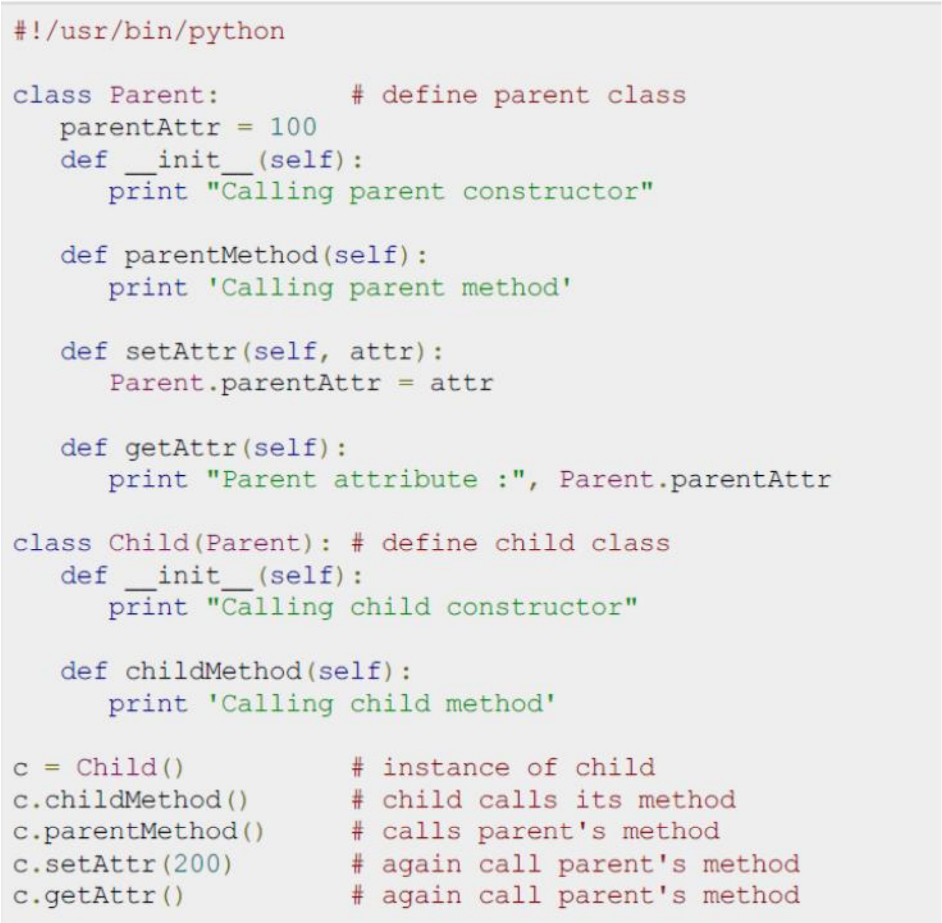
Similar way, you can drive a class from multiple parent classes as follows:

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

```
class B:         # define your calss B
.....

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

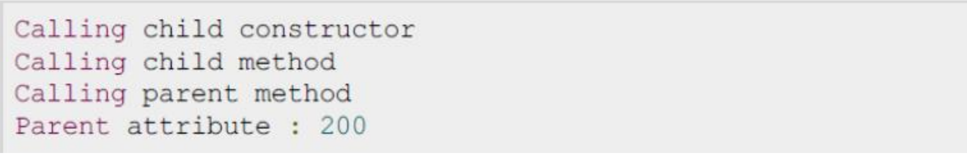
You can use issubclass() or isinstance() functions to check a relationships of two classes and instances.

- 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



```
#!/usr/bin/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
```

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



```
Calling child constructor
Calling child method
Calling parent method
Parent attribute : 200
```

원래 object class에 있는 setAttr() 이나 getAttr() 이 있는데 Parent class에서 setAttr() 이나 getAttr() 을 locally define

Overloading Operators:

Suppose you've 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.

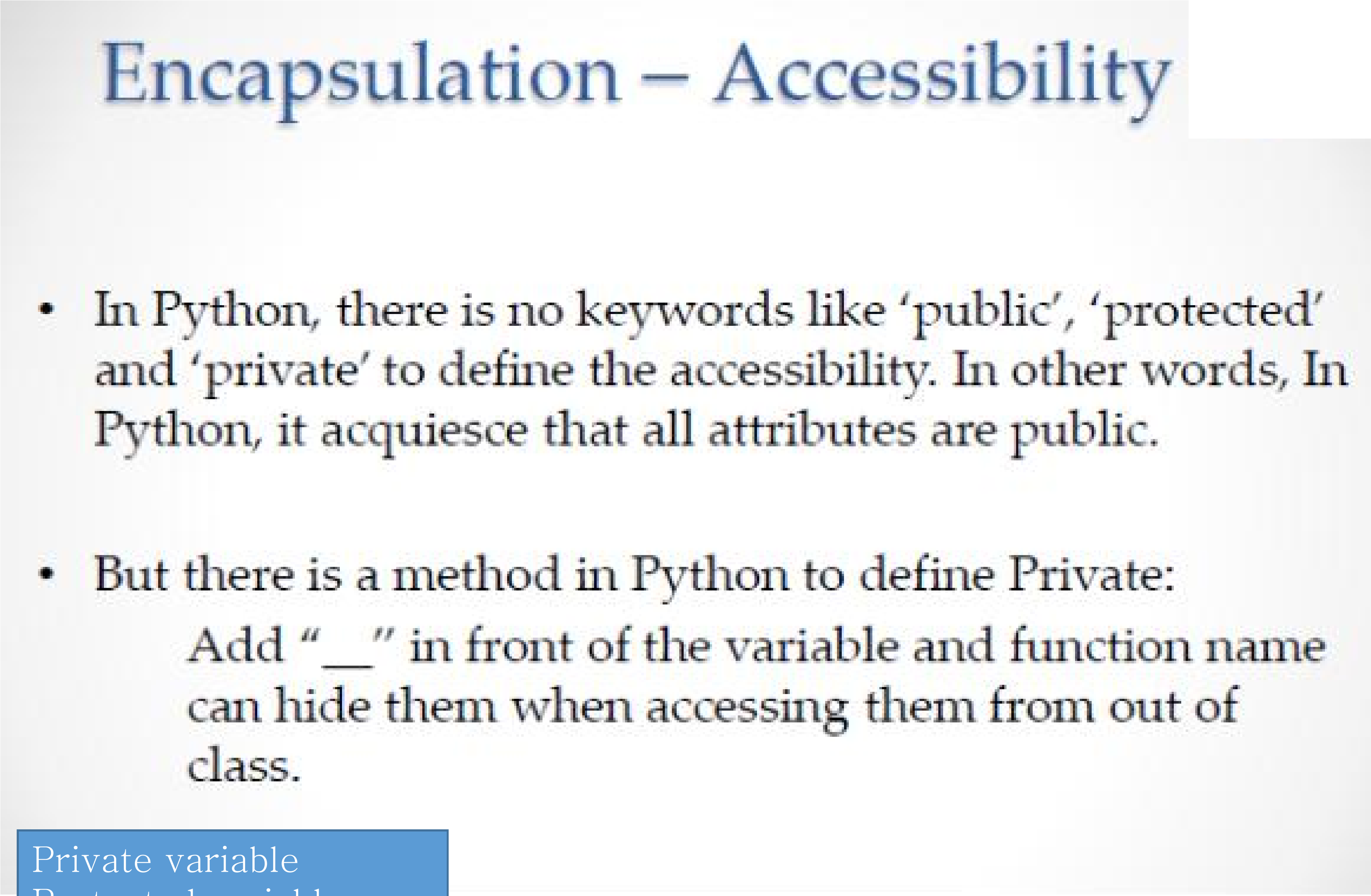
You could, however, define the __*add*__ method in your class to perform vector addition and then the plus operator would behave as per expectation:

Example:

```
#!/usr/bin/python

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)
```



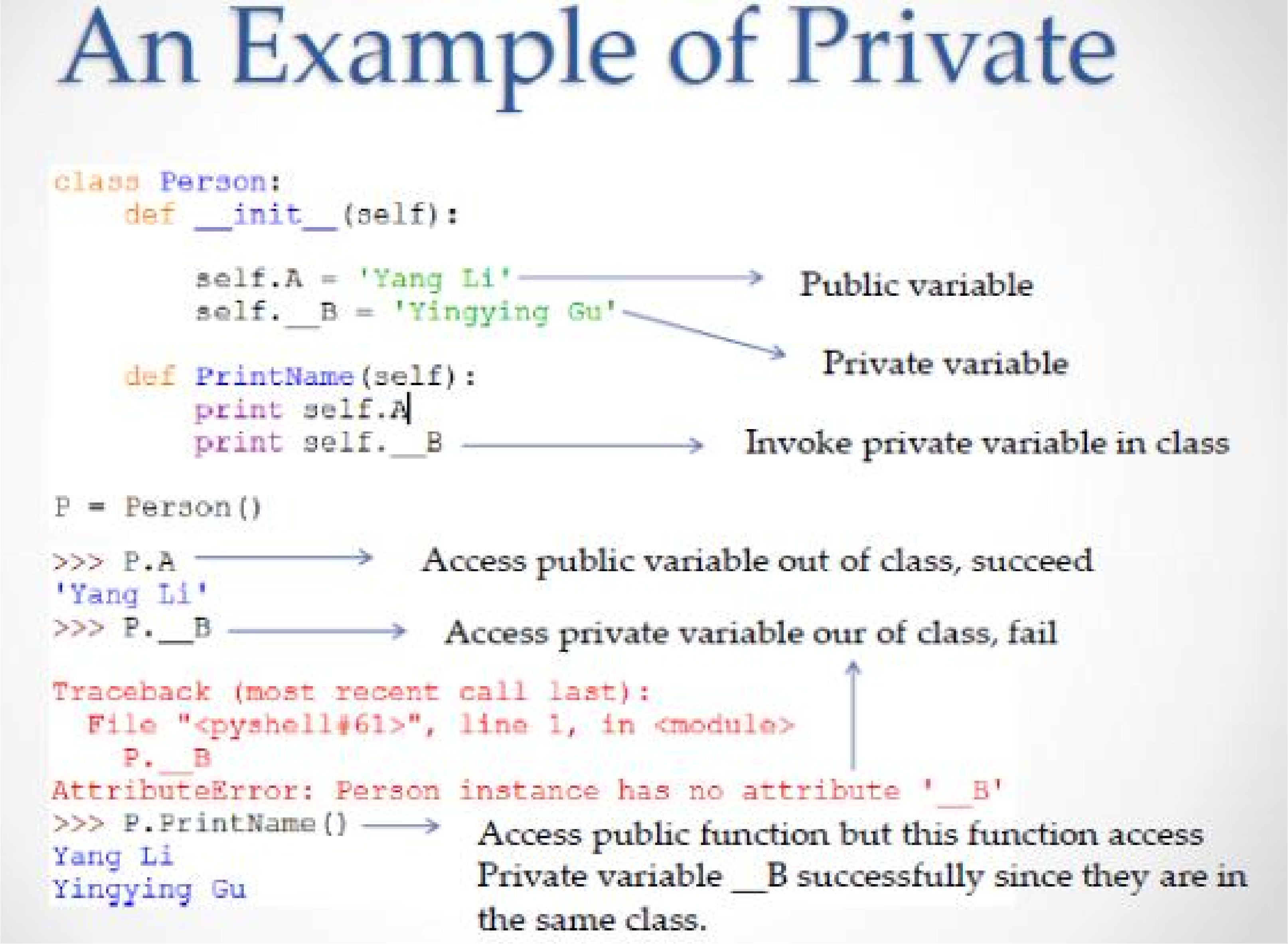
Encapsulation – Accessibility

- In Python, there is no keywords like ‘public’, ‘protected’ and ‘private’ to define the accessibility. In other words, In Python, it acquiesce that all attributes are public.

- But there is a method in Python to define Private:

  Add “__” in front of the variable and function name can hide them when accessing them from out of class.

Private variable
Protected variable
Pubic variable



An Example of Private

```
class Person:
    def __init__(self):

        self.A = 'Yang Li'
        self.__B = 'Yingying Gu'

    def PrintName(self):
        print self.A
        print self.__B

P = Person()

>>> P.A
'Yang Li'
>>> P.__B

Traceback (most recent call last):
  File "<pyshell#61>", line 1, in <module>
    P.__B
AttributeError: Person instance has no attribute '__B'
>>> P.PrintName()
Yang Li
Yingying Gu
```

Public variable

Private variable

Invoke private variable in class

Access public variable out of class, succeed

Access private variable our of class, fail

Access public function but this function access Private variable __B successfully since they are in the same class.

Variable __B는 외부에서 직접 access 불가능하고, PrintName()을 수행할때만 외부에서 볼수 있다 ➔ Information Hiding

Data Hiding:

An object's attributes may or may not be visible outside the class definition. For these cases, you can name attributes with a double underscore prefix, and those attributes will not be directly visible to outsiders.

Example:

```
#!/usr/bin/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 would work for you:

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

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

```
1
2
2
```