OOP 개념과 Python OO

High-Level Programming Paradigms

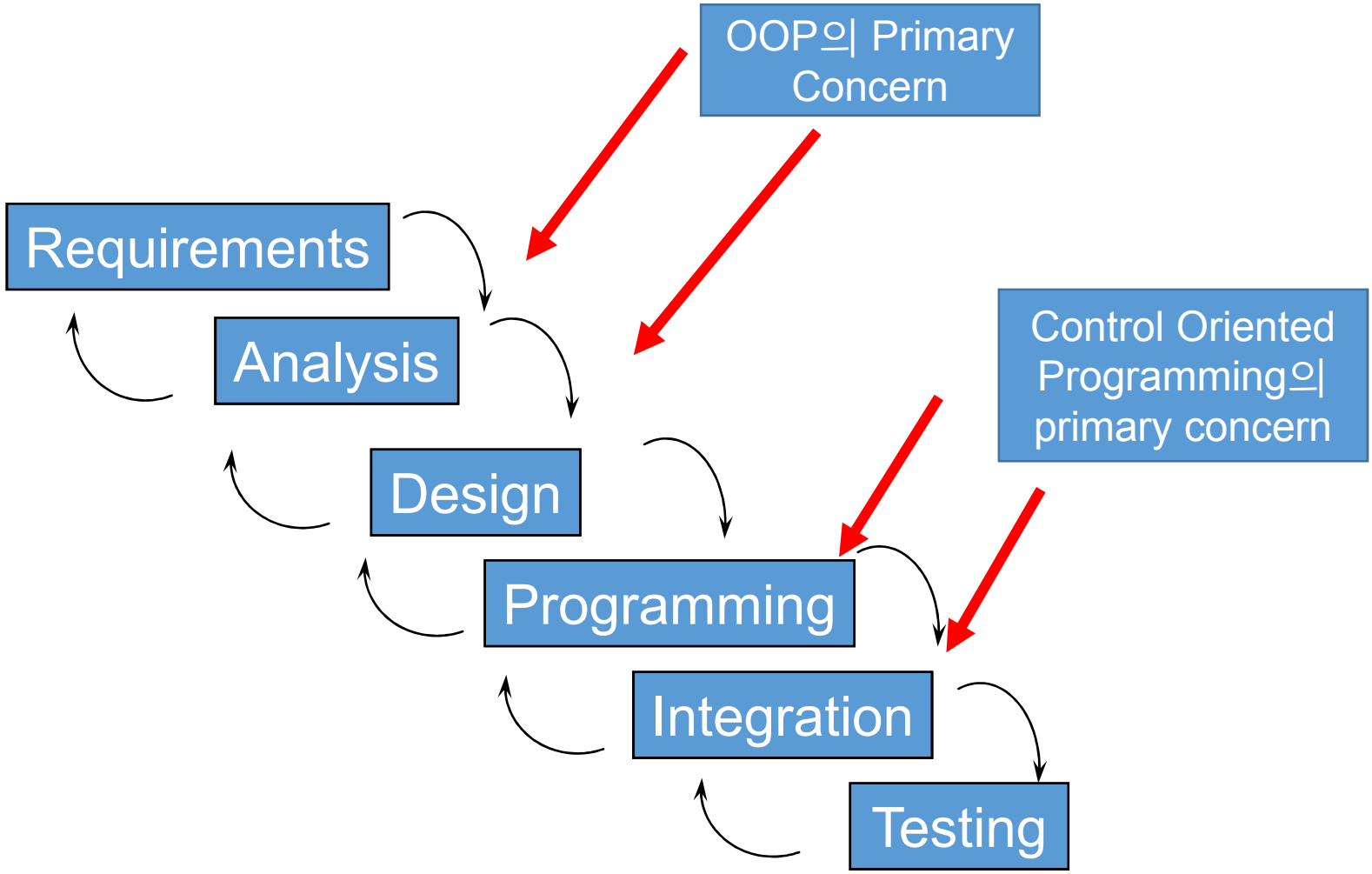
- Control-oriented Programming (before mid 80's)
  - Real world problem ➔ a set of functions
  - Data and functions are separately treated
  - Fortran, Pascal, C

- 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

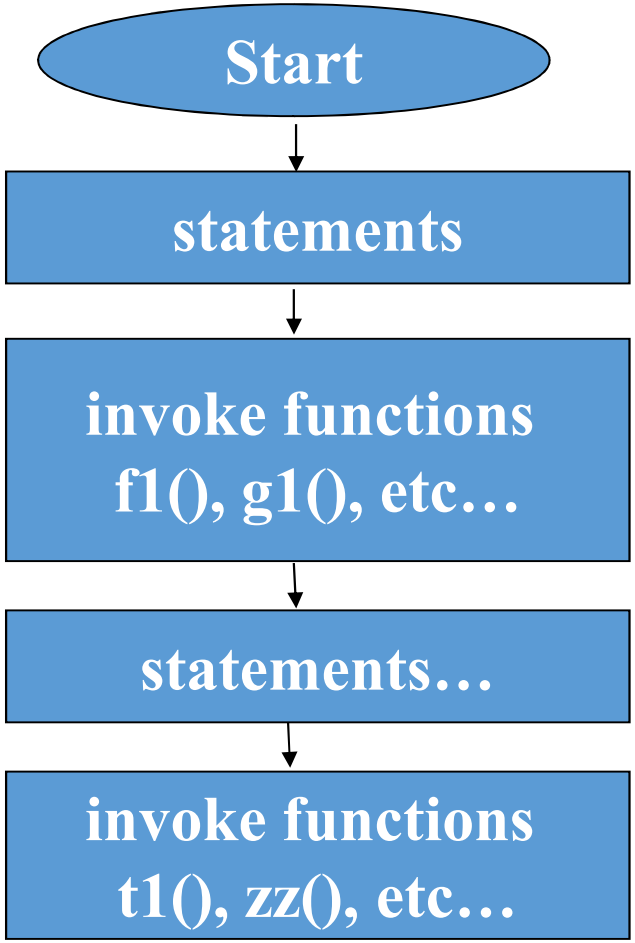
```
class CaptionTv extends Tv {
      boolean caption; // 캡션상태(on/off)

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

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

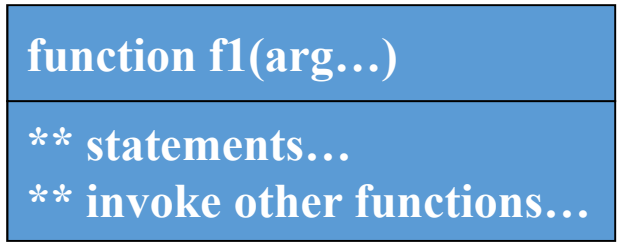
CaptionTVTest class

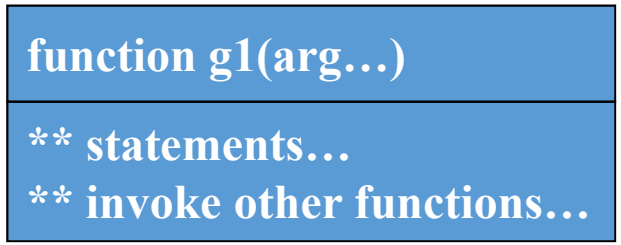
```
class CaptionTvTest {

 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.
      }
}
```

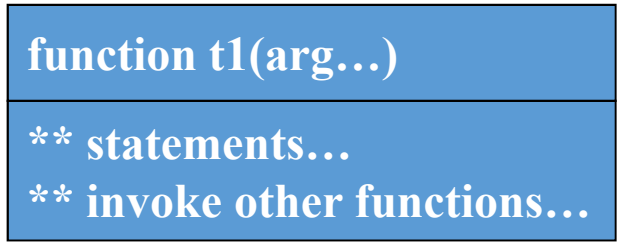
Control Oriented Programming Paradigm

- 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



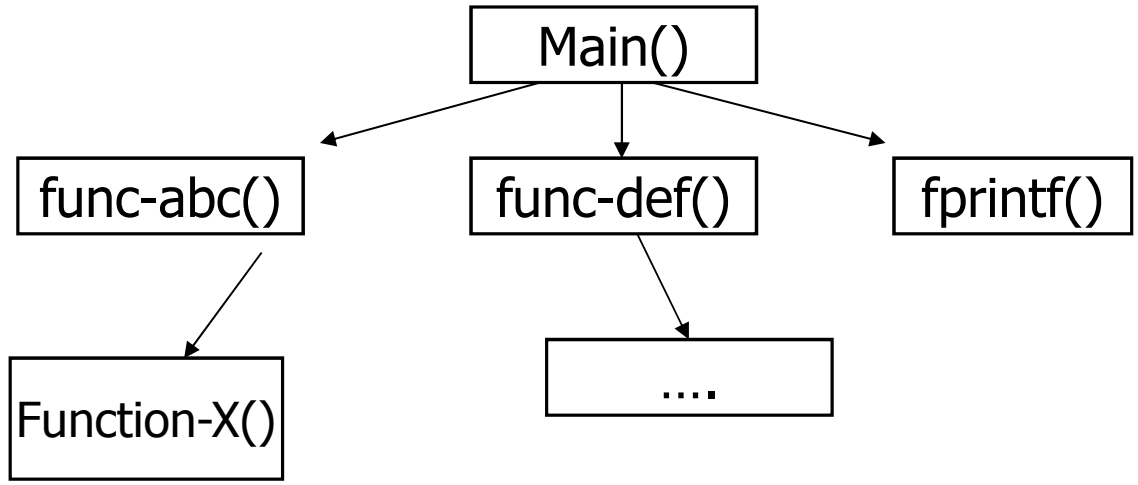






Sample C program (Function-based structure)

```
#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 ( )  { ……..}
```



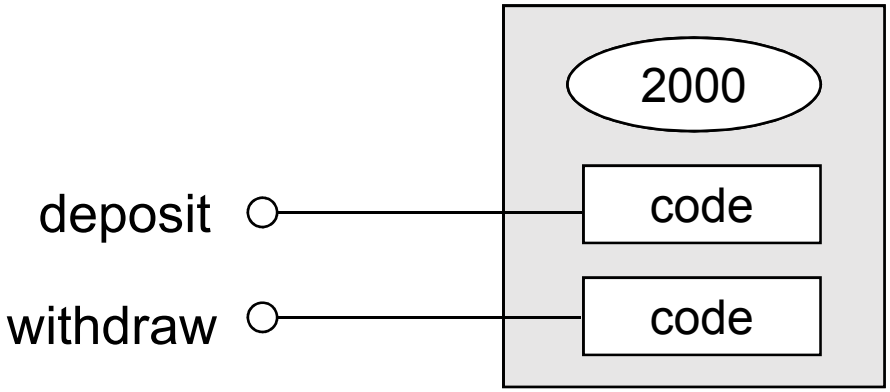
Object-Oriented Programming

- 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.*

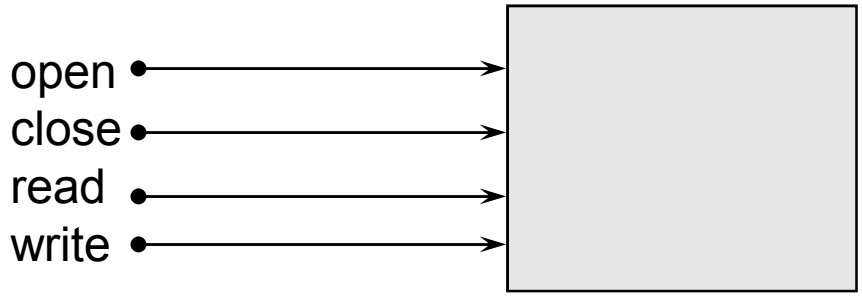
Method

An operations of an object is called *method*.

Message

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

Example: File Object



Object-oriented view

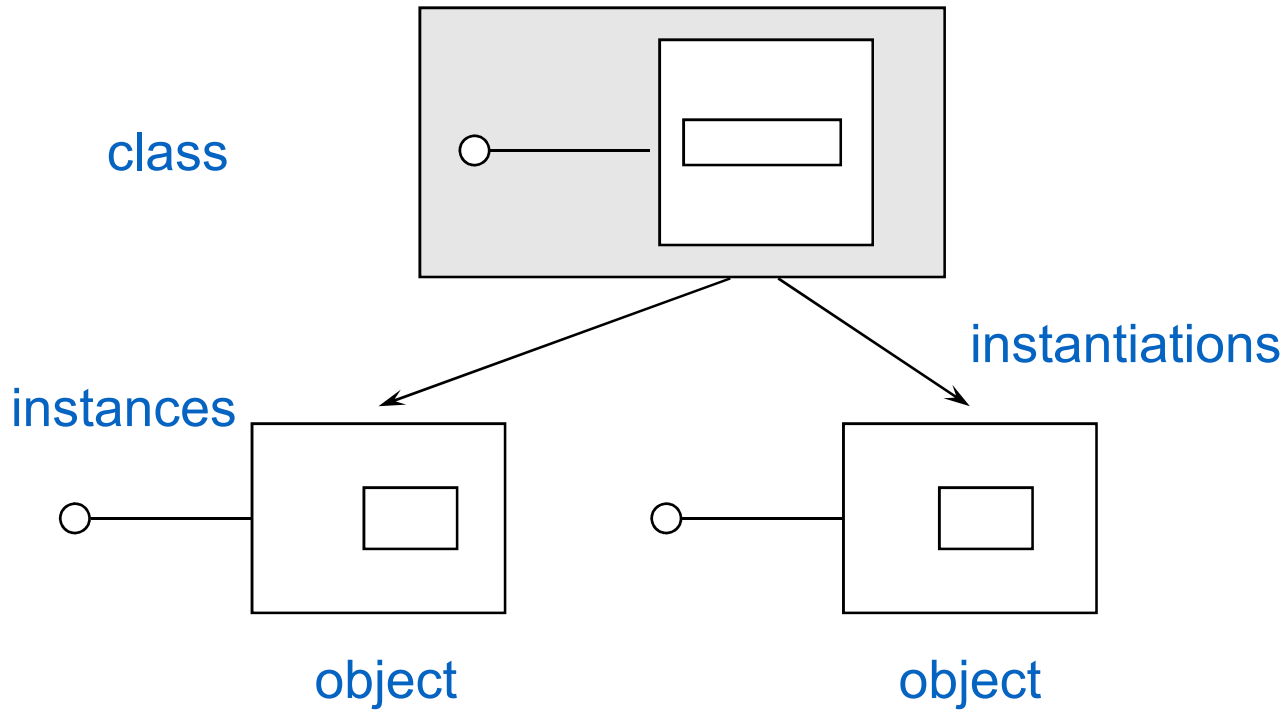
myFile open() : myFile, please open yourself.

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

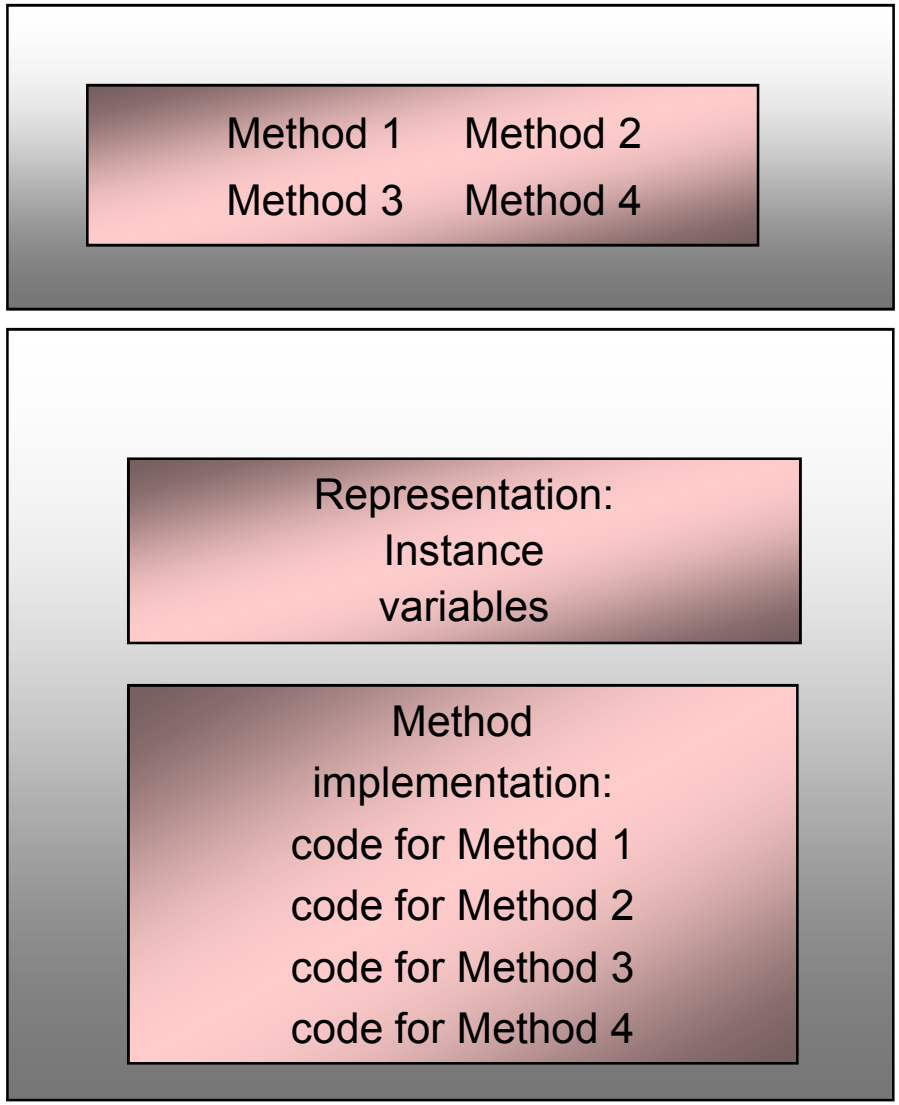
myFile close() : myFile, close yourself.

Class

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



The overall structure of an abstract data type (Class)

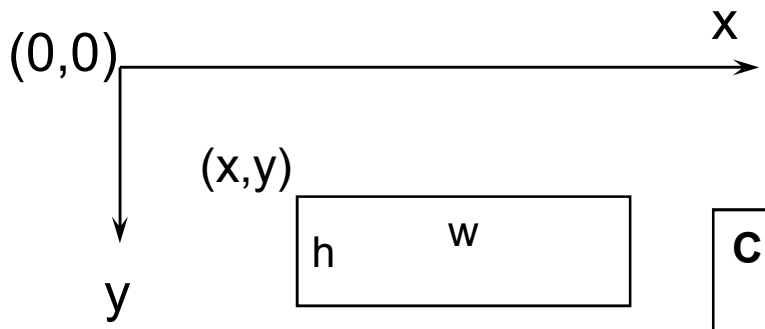


Interface
Public

Implementation
Private

Example: Rectangle

Define a Rectangle Class.

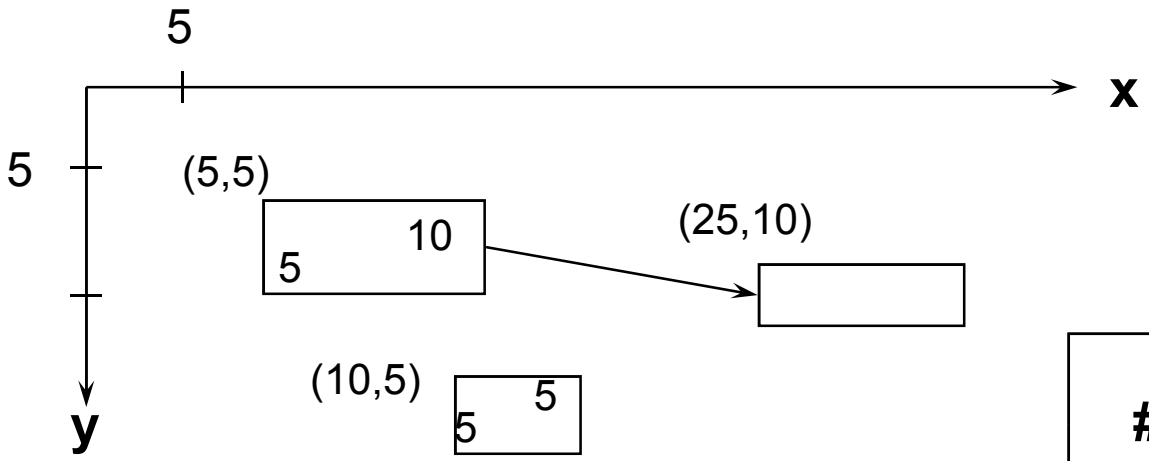



```
Class Rectangle
   data
            int x, y, h, w;
   method
            create (int x1, y1, h1, w1)
              { x=x1;   y=y1;   h=h1;   w=w1; }

            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




```
#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();
```

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.

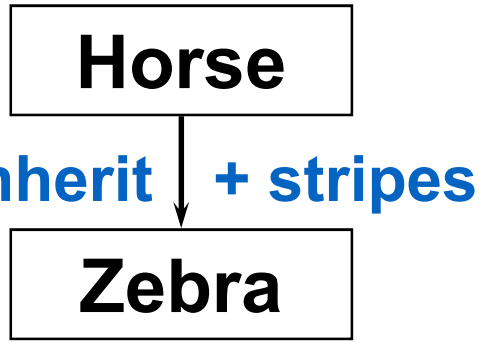
참고slide

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"



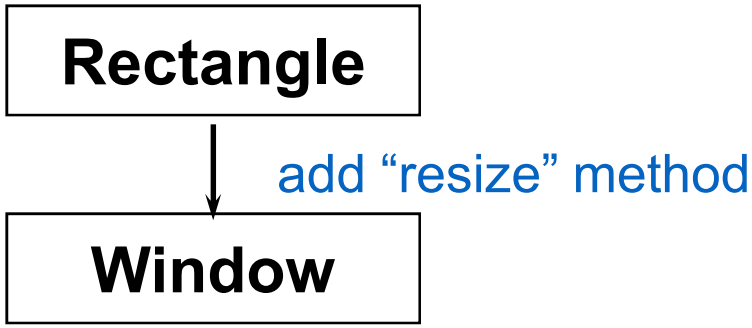
Inheritance avoid repetition and confusion!

참고slide

Example: Inheritance

Problem

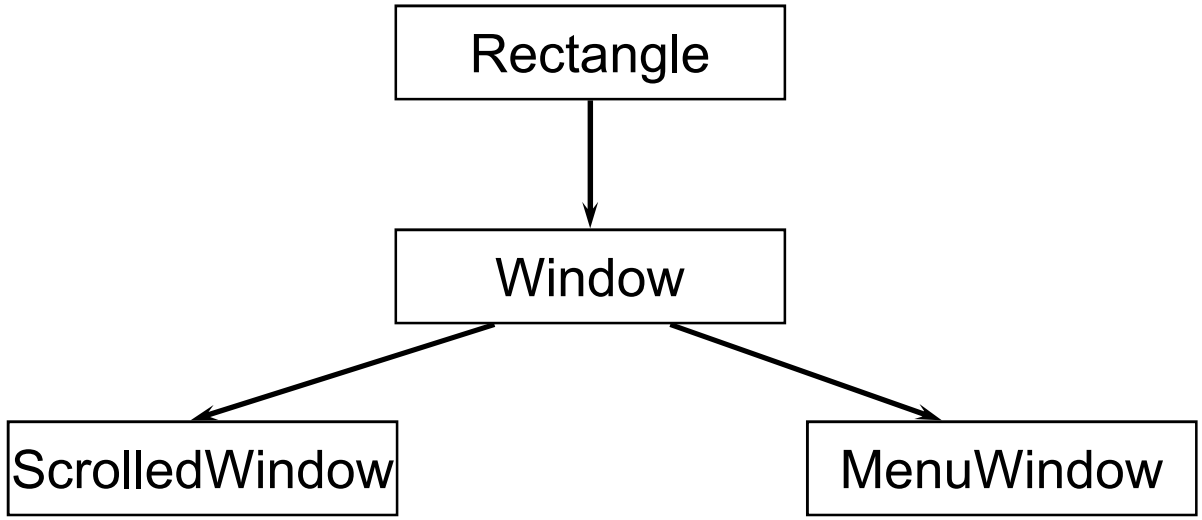
Define a new class called Window, which is a rectangle, but also resizable.




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

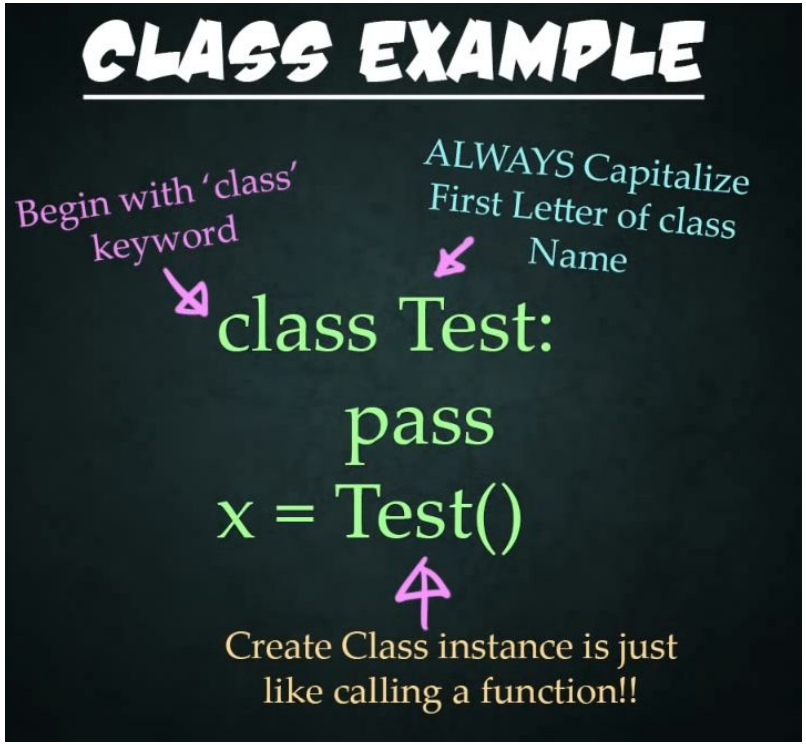
참고slide

Class Hierarchy



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

PYTHON OOP Tutorial




```
class TestClass:
    pass

X = TestClass()
```

EXAMPLE #2

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

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

TypeError!!!!

EXAMPLE #2

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

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

INITIALIZATION

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

Good

Bad

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

Good

Bad

```
Python Shell
File Edit Shell Debug Options Windows Help
py", line 10, in <module>
    print x.z
AttributeError: Ph instance has no attribute 'z'
>>> ================================ REST
==========
>>>
ham
>>>
```

```
classes_example.py - C:\Users\The_Captain\Desktop\classes_example.py
File Edit Format Run Options Windows Help
class Ph:
    def __init__(self):
        self.y = 5
        self.printHam()
    def printHam(self):
        print 'ham'
x = Ph()
```

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

```
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()
```

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
>>>
```

classes_example.py - C:\Users\The_Captain\Desktop\classes_example.py

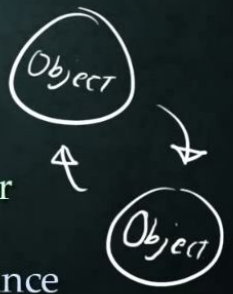
File Edit Format Run Options Windows Help

```
class Hero:
    '''
    A hero who is allegic to Apples.
    '''
    def __init__(self, name):
        '''
        whatever we want...
        '''
        self.name = name
        self.health = 100
    def eat(self, food):
```

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

Simple Inheritance Example

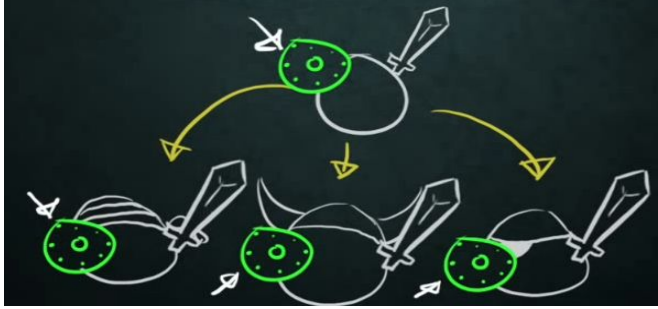
```
class BaseClass (object):
    def printHam(self):
        print 'ham'

class InheritingClass (BaseClass):
    pass

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

Inherit from this class

Why use OOP?

- Cleaner way of Programming!

- Changes to BaseClass affect all classes with Inherit FROM it!

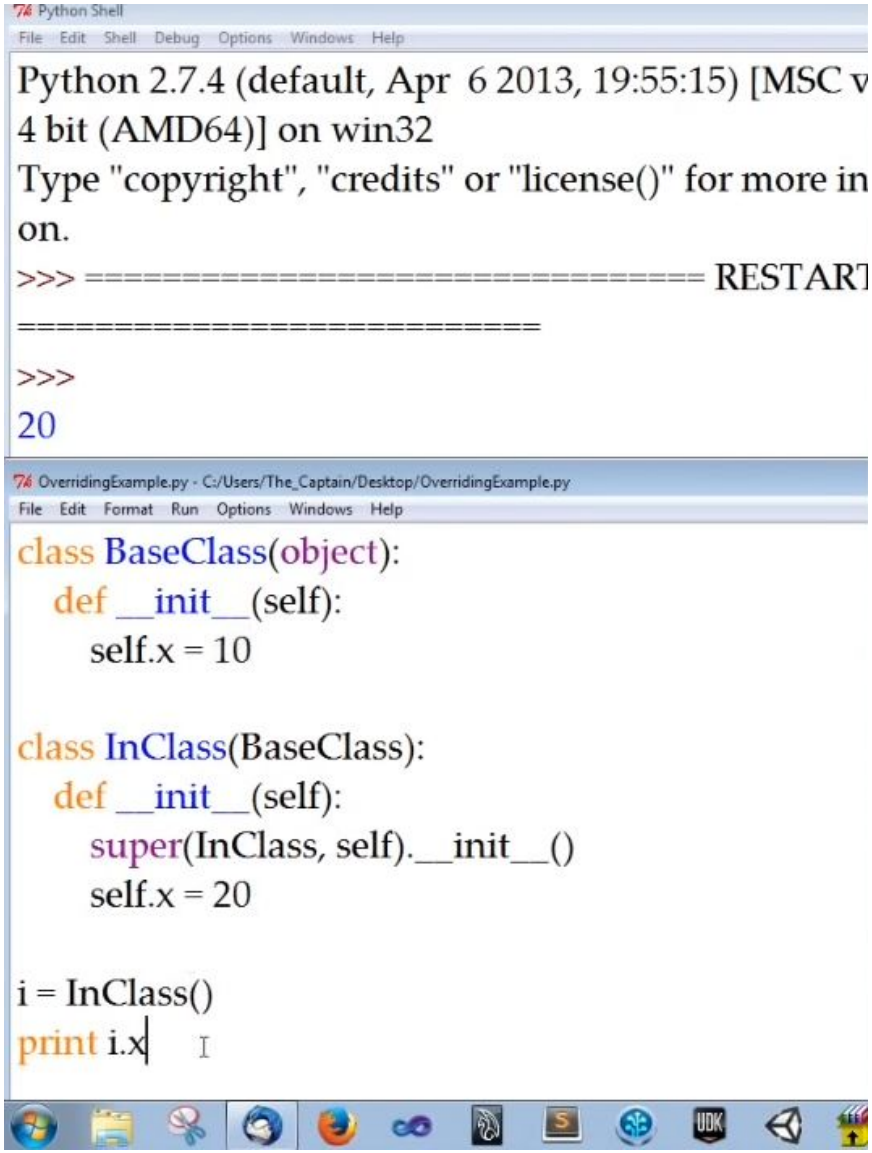


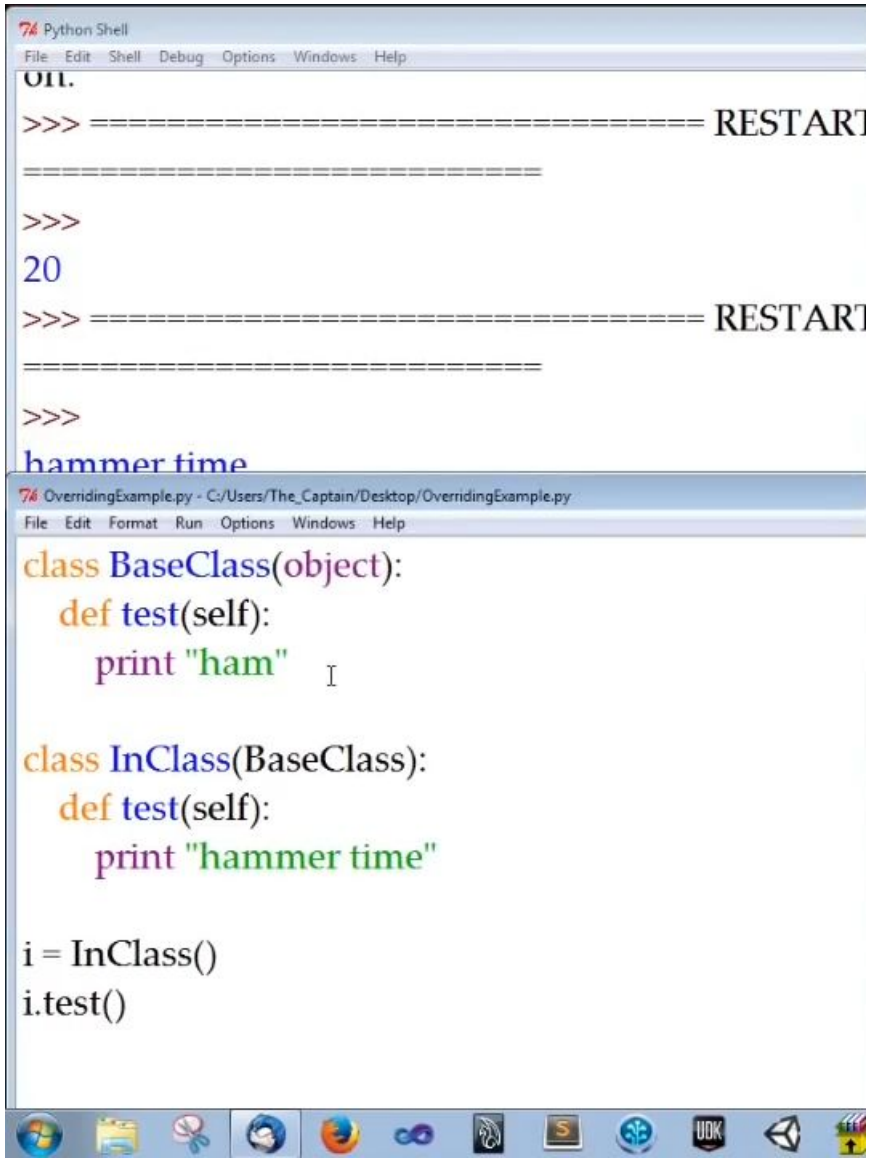
EXAMPLE #2

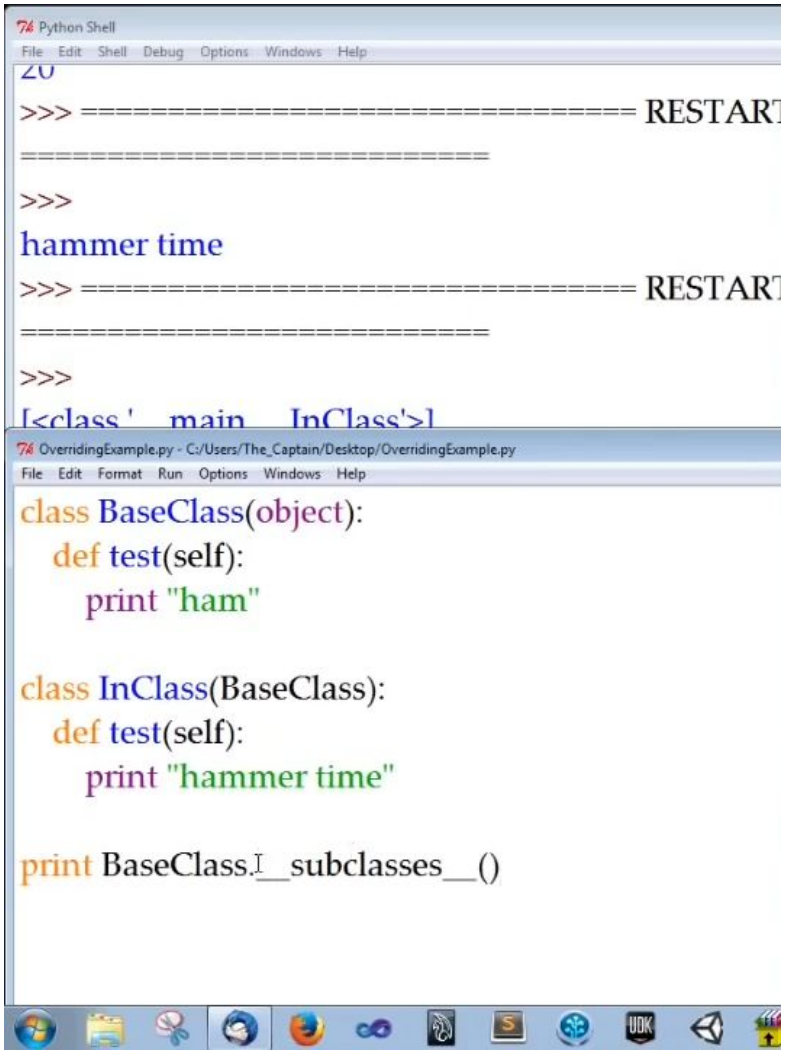
```
class Character (object):
    def __init__(self):
        self.health = 100

class Blacksmith (Character):
    def __init__(self):
        super(Blacksmith, self).__init__()

bs = Blacksmith()
print bs.health
```

calls the 'init' funciotn of BaseClass

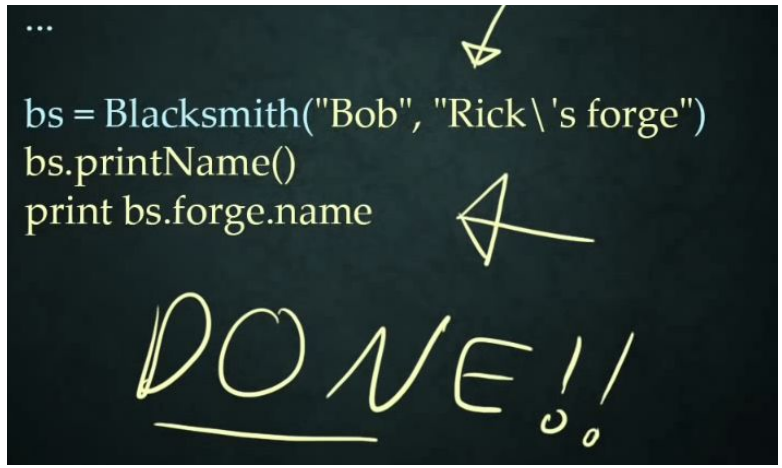
VISUALIZING THE CODE


Python Shell
File Edit Shell Debug Options Windows Help
Python 2.7.4 (default, Apr 6 2013, 19:55:15) [MSC v
4 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more in
on.
>>> ========== RESTART
==========
>>>
20
OverridingExample.py - C:/Users/The_Captain/Desktop/OverridingExample.py
File Edit Format Run Options Windows Help
class BaseClass(object):
def __init__(self):
self.x = 10
class InClass(BaseClass):
def __init__(self):
super(InClass, self).__init__()
self.x = 20
i = InClass()
print i.x


Python Shell
File Edit Shell Debug Options Windows Help
>>> ========== RESTART
==========
>>>
20
>>> ========== RESTART
==========
>>>
hammer time
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()


Python Shell
File Edit Shell Debug Options Windows Help
>>> ========== 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__()


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

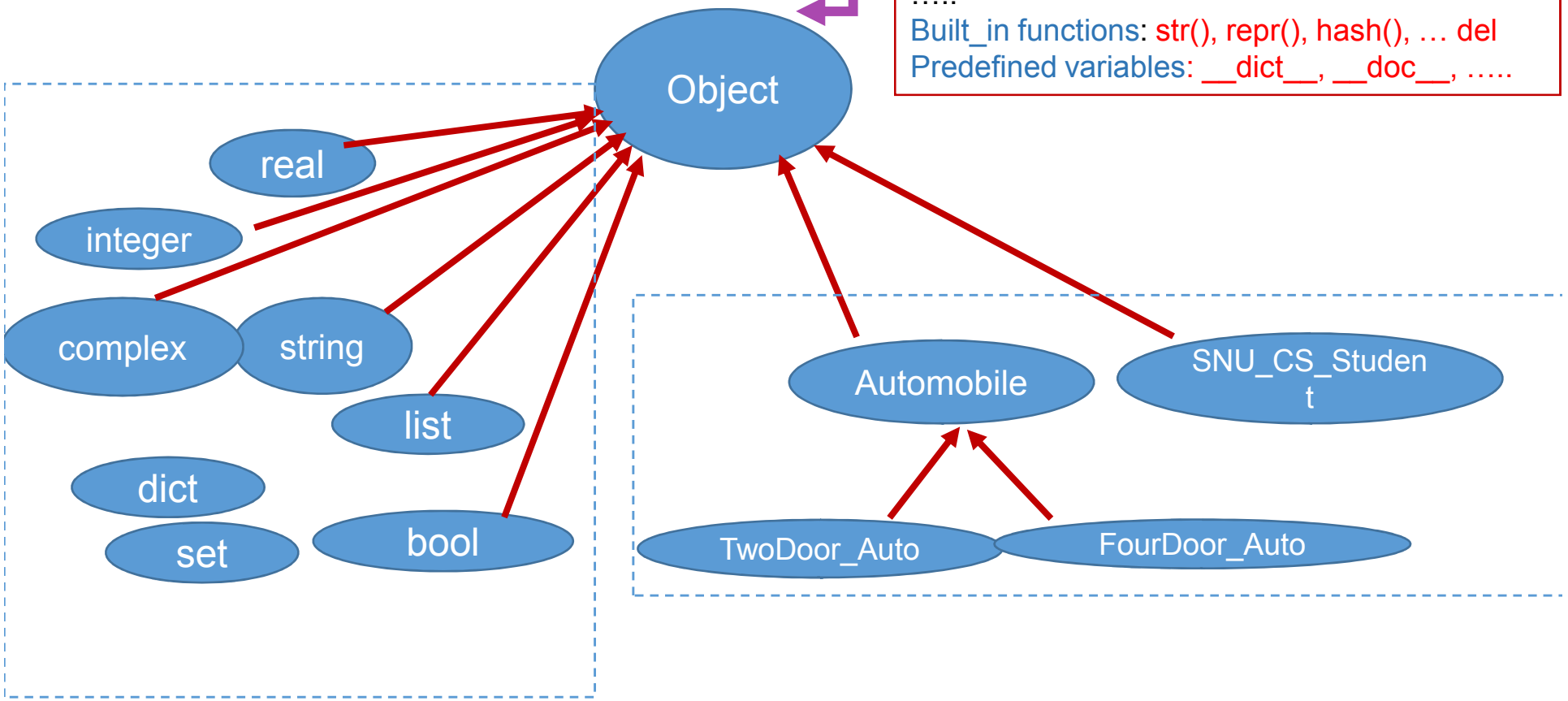

...
bs = Blacksmith("Bob", "Rick\'s forge")
bs.printName()
print bs.forge.name
DONE!!

A More Formal Way

of Python OO Programming

Python의 type system
(class structure)

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



Python의 Primitive operators, Built_in functions, Predefined variables들이 user_defined classes들에 내려와서는 작동이 안할수 있으므로 user_defined class의 내부에서 __wyx__로 redefine이 될수 있다!

Overview of OOP Terminology

- **Class:** A user-defined prototype for an object that defines a set of attributes that characterize any object of the class. The attributes are data members (class variables and instance variables) and methods, accessed via dot notation.
- **Class variable:** A variable that is shared by all instances of a class. Class variables are defined within a class but outside any of the class's methods. Class variables aren't used as frequently as instance variables are.
- **Data member:** A class variable or instance variable that holds data associated with a class and its objects.
- **Function overloading:** The assignment of more than one behavior to a particular function. The operation performed varies by the types of objects (arguments) involved.
- **Instance variable:** A variable that is defined inside a method and belongs only to the current instance of a class.
- **Inheritance :** The transfer of the characteristics of a class to other classes that are derived from it.
- **Instance:** An individual object of a certain class. An object obj that belongs to a class Circle, for example, is an instance of the class Circle.
- **Instantiation :** The creation of an instance of a class.
- **Method :** A special kind of function that is defined in a class definition.
- **Object :** A unique instance of a data structure that's defined by its class. An object comprises both data members (class variables and instance variables) and methods.
- **Operator overloading:** The assignment of more than one function to a particular operator.

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')
```

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.

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()
```

Special Class Attributes in Python

- Except for self-defined class attributes in Python, class has some special attributes. They are provided by object module.

Attributes Name	Description
__dict__	Dict variable of class name space
__doc__	Document reference string of class
__name__	Class name
__module__	Module name consisting of class
__bases__	The tuple including all the superclasses

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

Form and Object for Class

- Class includes two members: form and object.
- The example in the following can reflect what is the difference between object and form for class.

```
class A:
    i = 123
    def __init__(self):
        self.i = 12345

print A.i
print A().i
```

```
>>>
123
12345
```

Invoke form: just invoke data or method in the class, so i=123

Invoke object: instantialize object Firstly, and then invoke data or Methods.
Here it experienced __init__(), i=12345

class variable vs instance variable

```
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
```

- The variable *empCount* is a class variable whose value would be shared among all instances of a this class. This can be accessed as *Employee.empCount* from inside the class or outside the class.
- The first method *__init__()* is a special method, which is called class constructor or initialization method that Python calls when you create a new instance of this class.
- You declare other class methods like normal functions with the exception that the first argument to each method is *self*. Python adds the *self* argument to the list for you; you don't need to include it when you call the methods.

Creating instance objects:

To create instances of a class, you call the class using class name and pass in whatever arguments its __init__ method accepts.

```
"This would create first object of Employee class"
emp1 = Employee("Zara", 2000)
"This would create second object of Employee class"
emp2 = Employee("Manni", 5000)
```

Accessing attributes:

You access the object's attributes using the dot operator with object. Class variable would be accessed using class name as follows:

```
emp1.displayEmployee()
emp2.displayEmployee()
print "Total Employee %d" % Employee.empCount
```

Now, putting all the concepts together:

```
#!/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:

```
empl.age = 7  # Add an 'age' attribute.
empl.age = 8  # Modify 'age' attribute.
del empl.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(empl, 'age')    # Returns true if 'age' attribute exists
getattr(empl, 'age')    # Returns value of 'age' attribute
setattr(empl, 'age', 8) # Set attribute 'age' at 8
delattr(empl, 'age')    # Delete attribute 'age'
```

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 objects
del pt1
del pt2
del pt3
```

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

```
3083401324 3083401324 3083401324
Point destroyed
```

Note: Ideally, you should define your classes in separate file, then you should import them in your main program file using *import* statement. Kindly check Python - Modules chapter for more details on importing modules and classes.

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
```

Example:

```
#!/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
```

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
```

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)*

del x 를 만나면 x.__del__()를 수행

repr(x) 를 만나면 x.__repr__()를 수행

str(x) 를 만나면 x.__str__()를 수행

==를 만나면 __eq__()를 수행, >를 만나면 __gt__()를 수행,
=>를 만나면 __ge__()를 수행, <를 만나면 __lt__()를 수행,
+를 만나면 __add__()를 수행 ……

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)
```

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
```

Private variable
Protected variable
Pubic variable

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
```

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()
```

Encapsulation – Accessibility (1)

- 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.

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.

Encapsulation – Accessibility (2)

- Actually, the private accessibility method is just a rule, not the limitation of compiler.
- Its fact is to change name of private name like __variable or __function() to _ClassName__variable or _ClassName__function(). So we can't access them because of wrong names.
- We even can use the special syntax to access the private data or methods. The syntax is actually its changed name. Refer to the following example in the next slide.