OOP 개념과 Python OOP

- History of Programming Languages
- Abstract Data Type
- Python OOP Tutorial
- Why do we need classes and OOP?
- More Formal Way of Python OOP

High-Level Programming Paradigms

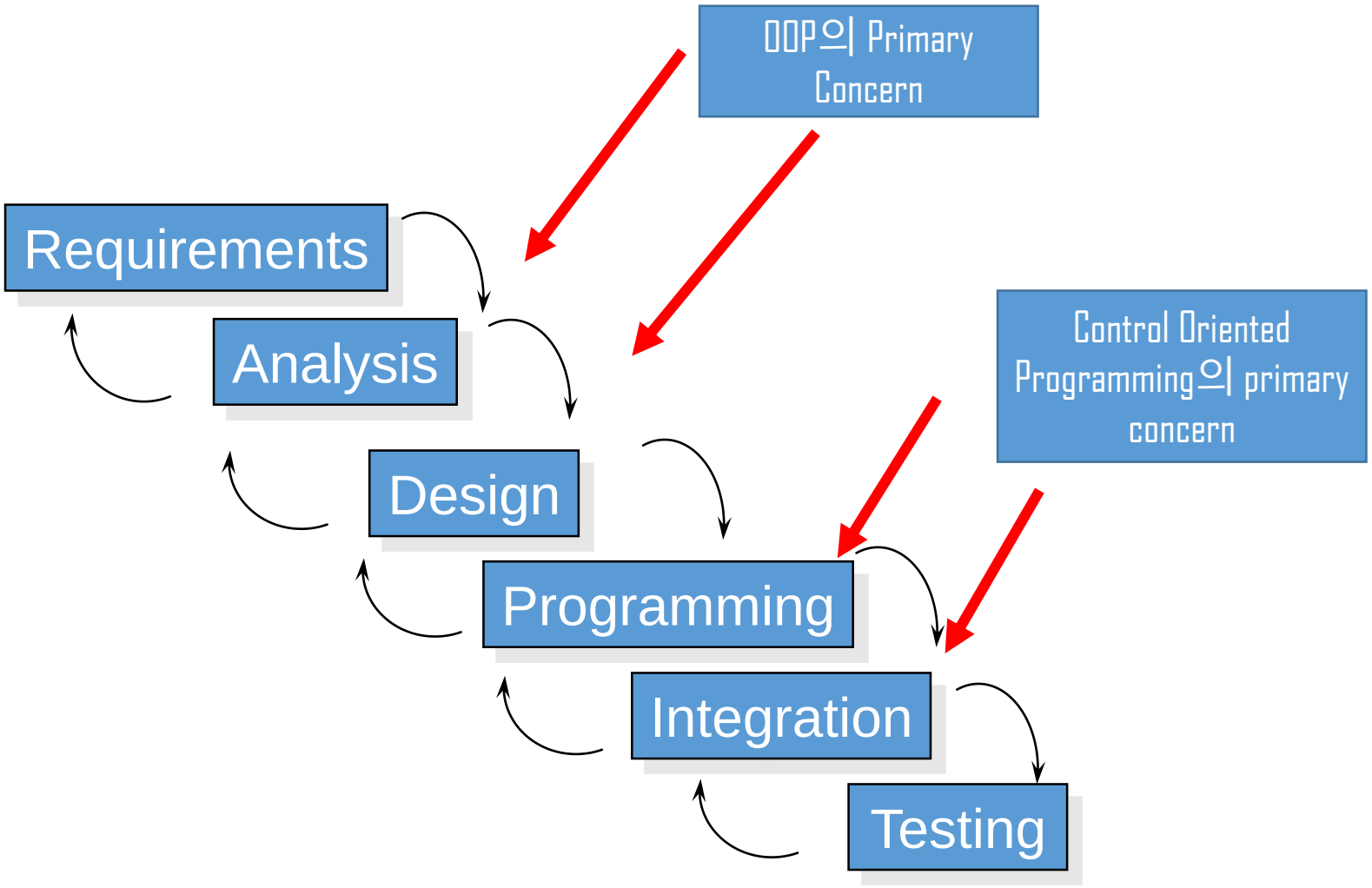
- Control-oriented Programming (before mid 80's)
  - Real world problem ➔ a set of functions
  - Data and functions are separately treated
  - Fortran, Cobol, PL/1, Pascal, C

- Object-oriented Programming (after mid 80's)
  - Real world problem ➔ a set of classes
  - Data and functions are encapsulated inside classes
  - C++ (1983)
  - Python (1991)
  - Java (1993)
  - and most Script Languages (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



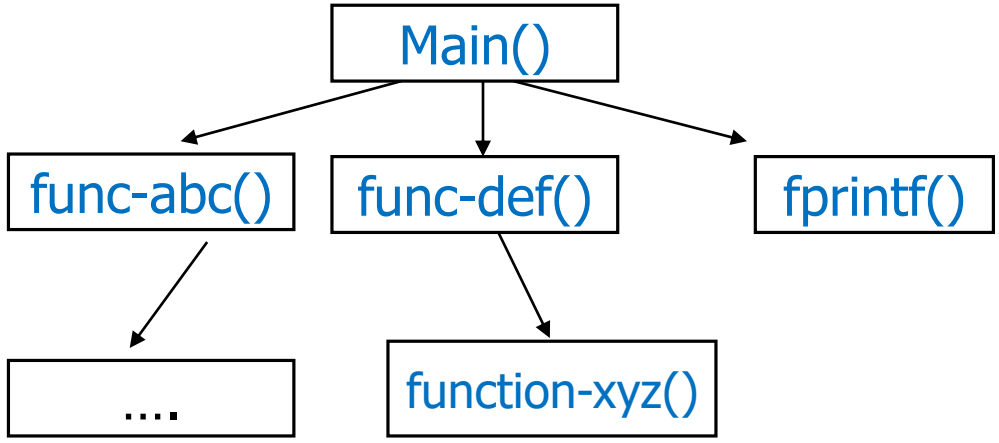
Typical Control-Oriented Programming:
C code for TV operations

```
#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);
       }
}
```

Sample C program (Function-based structure: Top-Down Function Design)

```
#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()    }
```



OOP 개념과 Python OOP

- History of Programming Languages
- Abstract Data Type
- Python OOP Tutorial
- Why do we need classes and OOP?
- More Formal Way of Python OOP

Concept of Abstract Data Types [1/5]

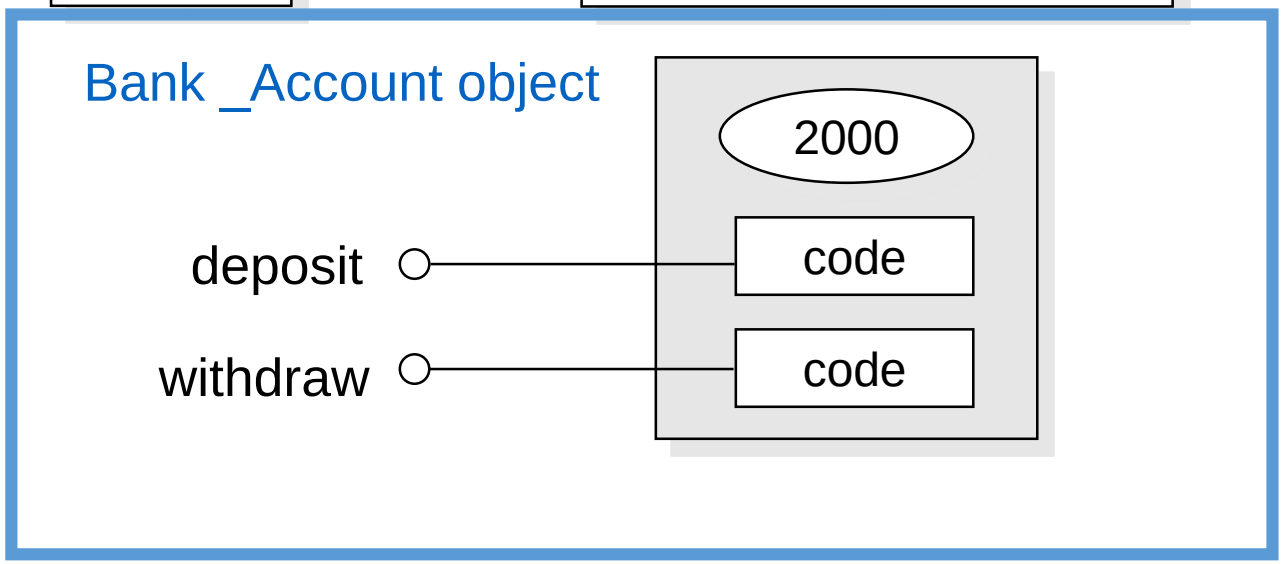
- Basic Data Types
  - Integer
  - Floating Number
  - Character
  - String
  - Boolean

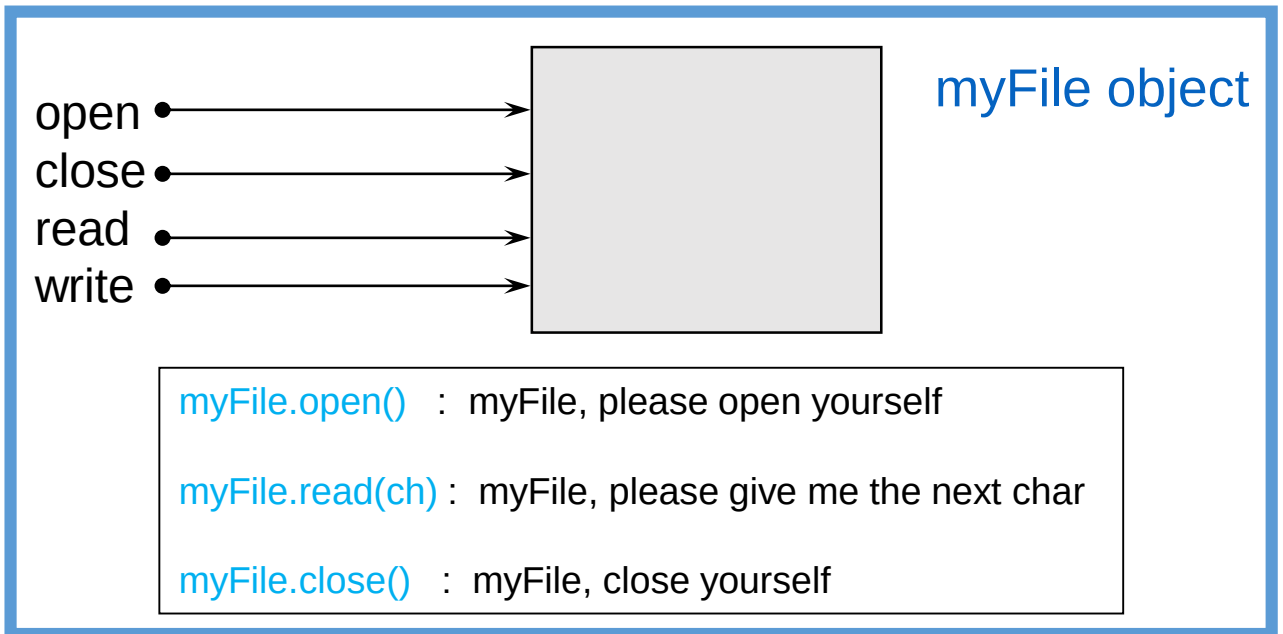
- Advanced Data Types
  - List (Array, Matrix)
  - Set
  - Dictionary
  - Tuple

- User-Defined Abstract Data Types (= Classes)
  - Student Data Type
  - Professor Data Type
  - Automobile Data Type
  - Bank-Account Data Type

Concept of Abstract Data Types [2/5]

Object = **Data + Operations**

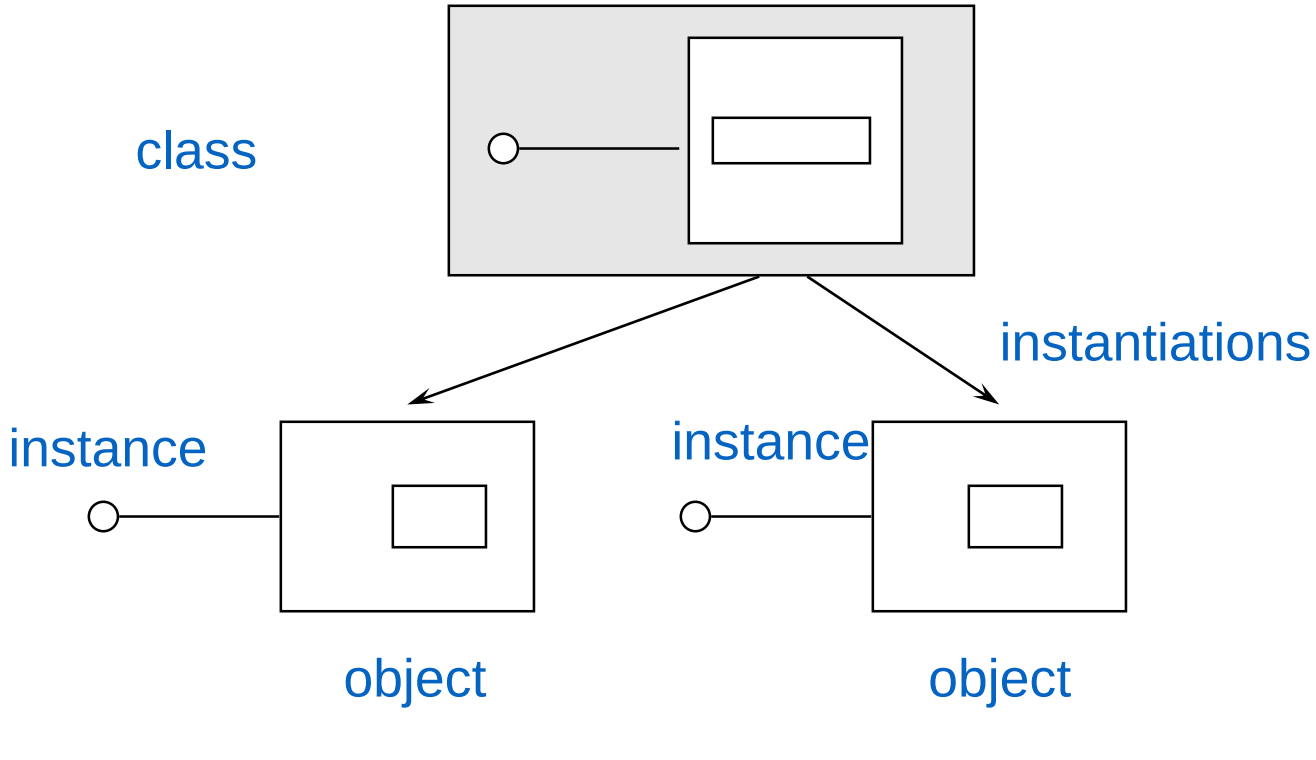


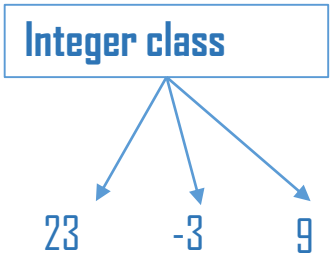


Concept of Abstract Data Types [3/5]

Class

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

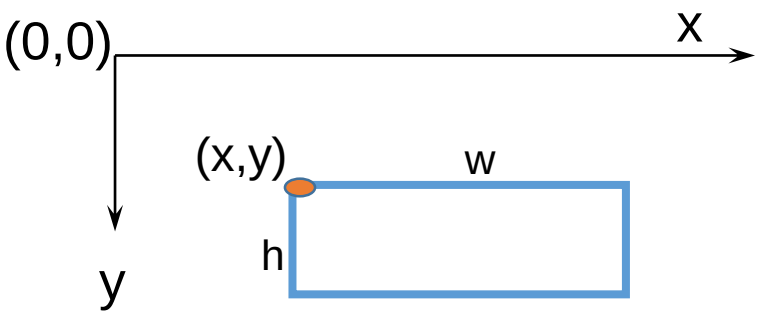


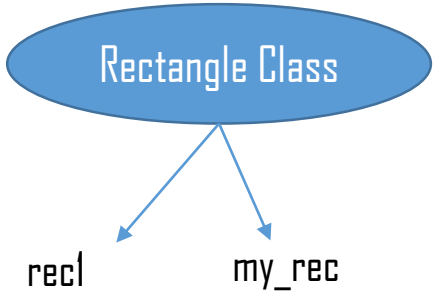


X = 23
Y = int(-3)

Concept of Abstract Data Types [4/5]

Define a Rectangle Class



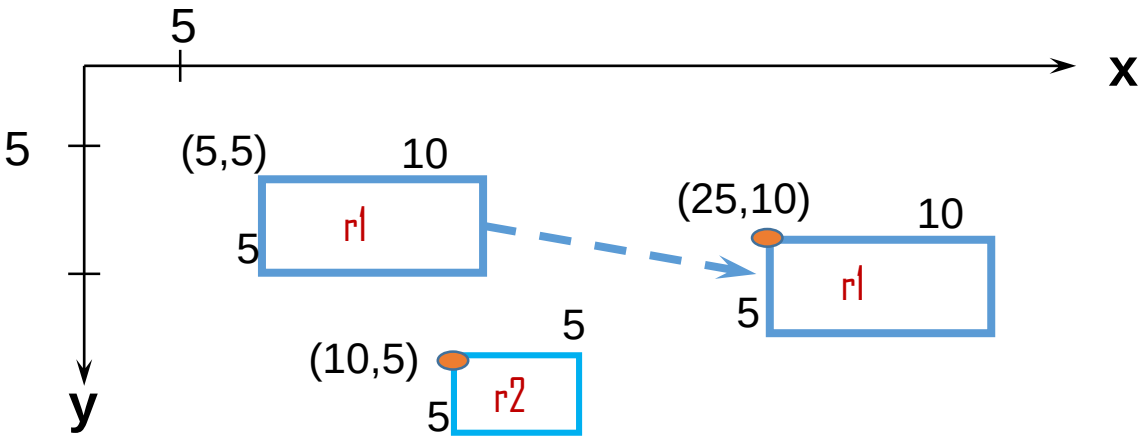



```
Rectangle rec1, my_rec
rec1.create(5,5,10,5)
my_rec.create(10,10,10,5)
```

```
Class Rectangle
   variables
            int x, y, h, w;
   methods
            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
```

Concept of Abstract Data Types [5/5]

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 manipulate 안했다면...

(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 Classes and 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

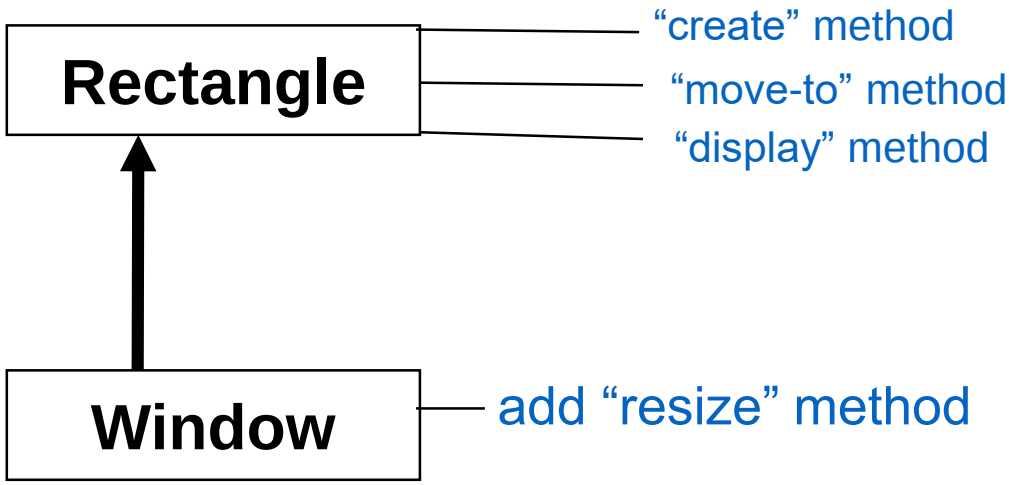
- (W, L, H) 가 그 Box를 표현하고 있는데, Basic Data Type을 가지고는 이런 표현을 할 방법이 없네!
  - Box structure ➔ (W, L, H) 뭐 이런거가 있으면 좋겠는데…
  - 단순 Variable들을 묶어서 복잡한 구조를 표현하면 좋을텐데...

- 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!!!!
  - Retangle Class (abstract data type)를 만들고 필요한 object instance를 생성하여 coding에 이용

Another Benefit of Abstract Data Types:
Class Inheritance

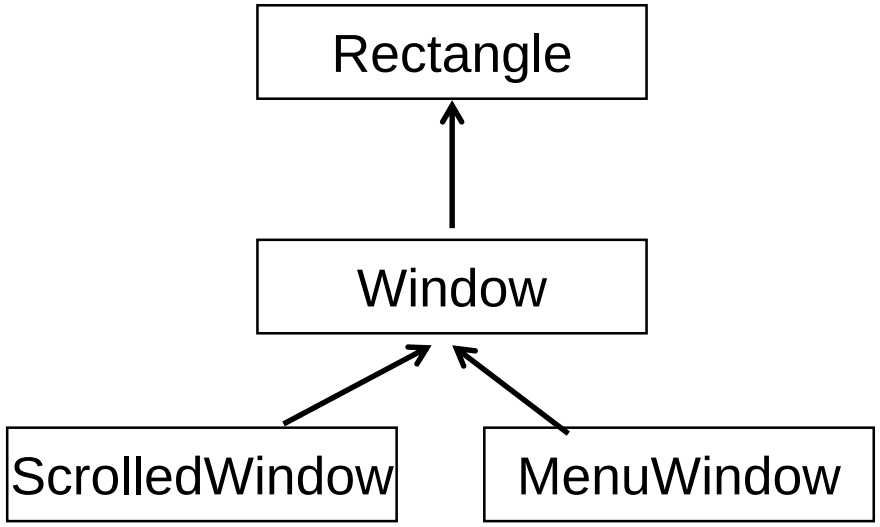
Problem

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



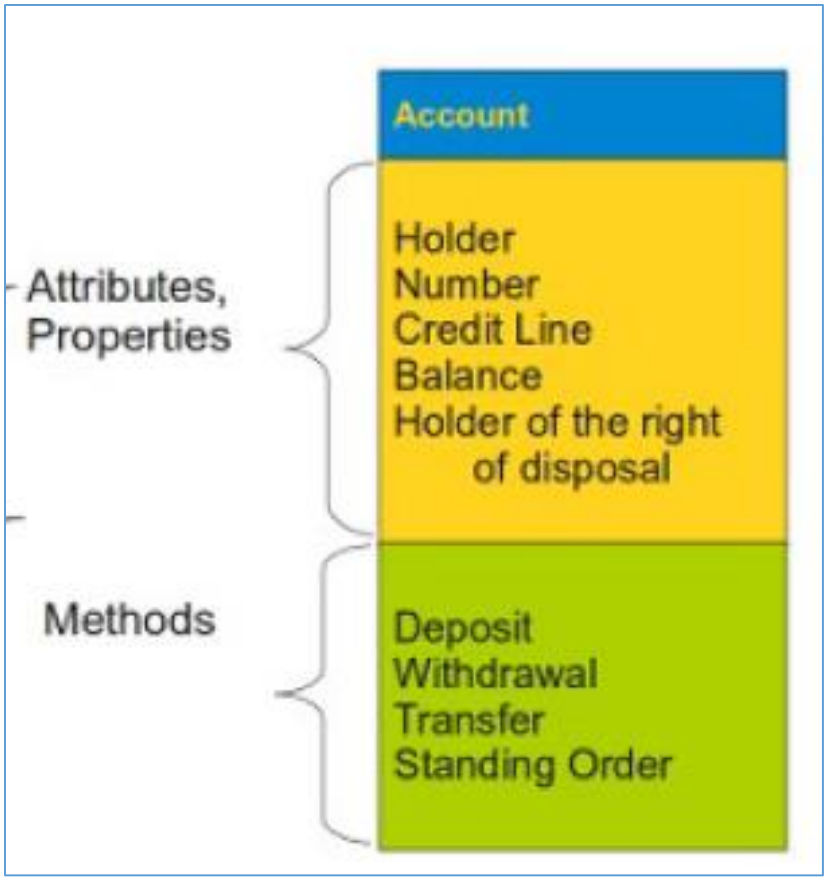

```
Class Window
    inherit Rectangle
    add operation
            resize(int h1, w1)
                    { h=h1; w=w1; display(); }
```

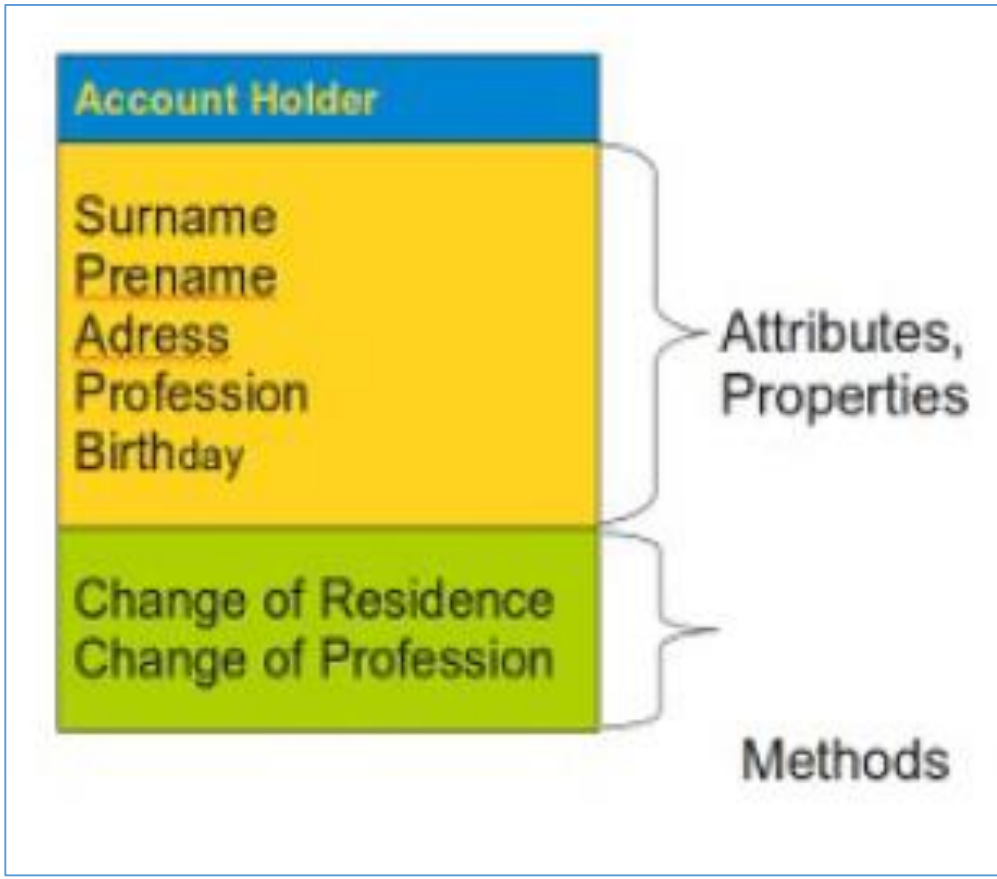
Class Hierarchy provides Code Reusability and Cleaner Way of Programming



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

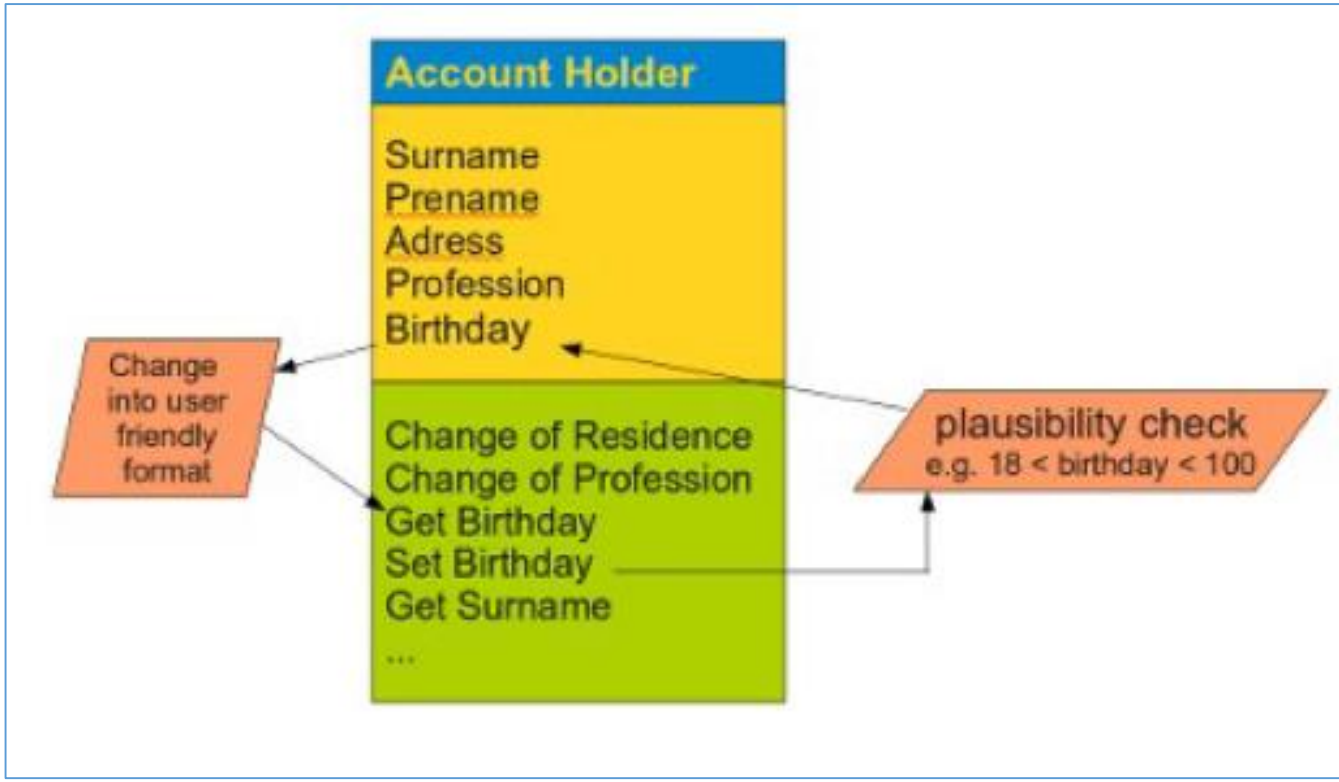
OOP of Banking Account [1/3]





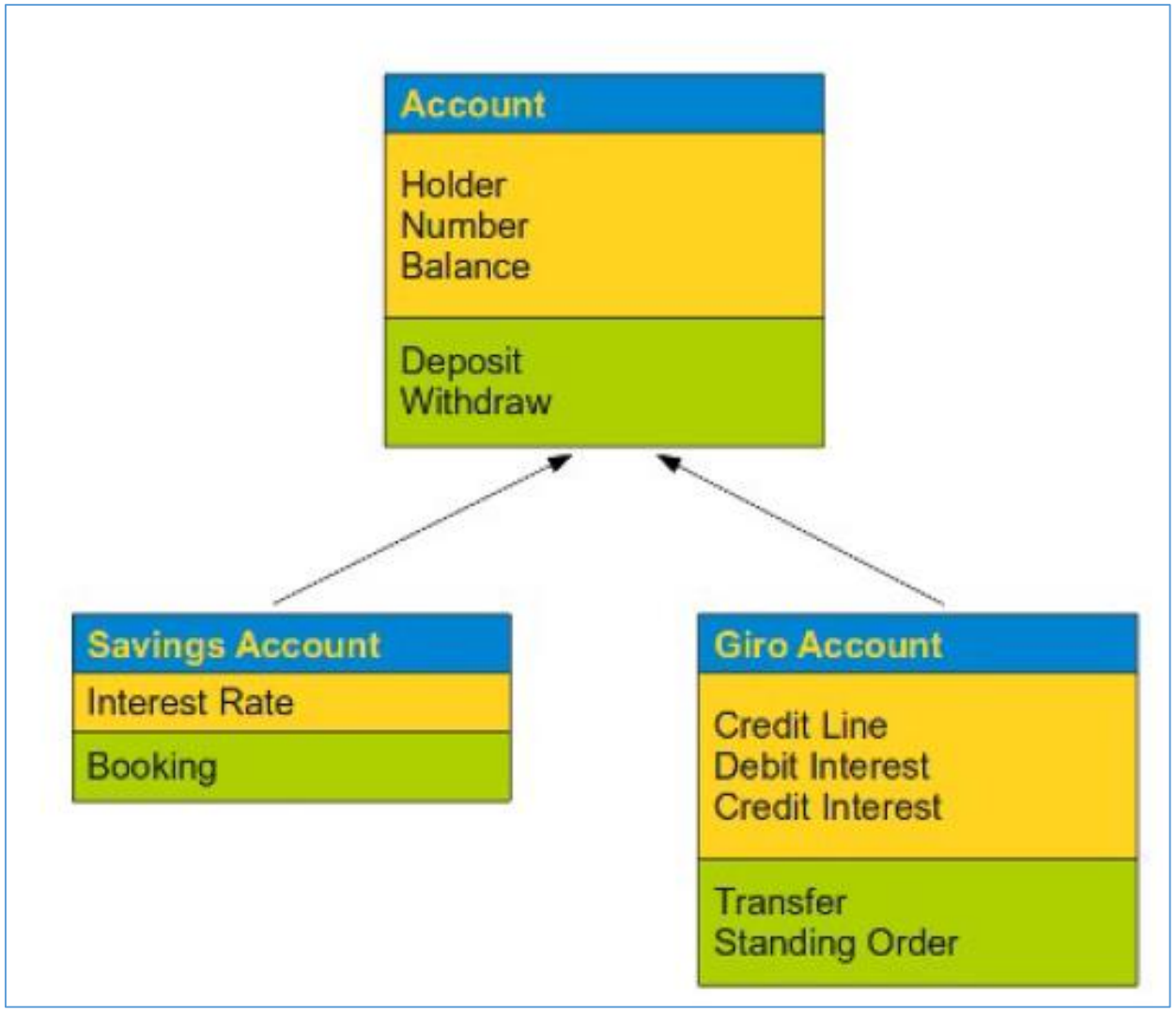
OOP of Banking Account [2/3]

Encapsulation of Data = Data Hiding

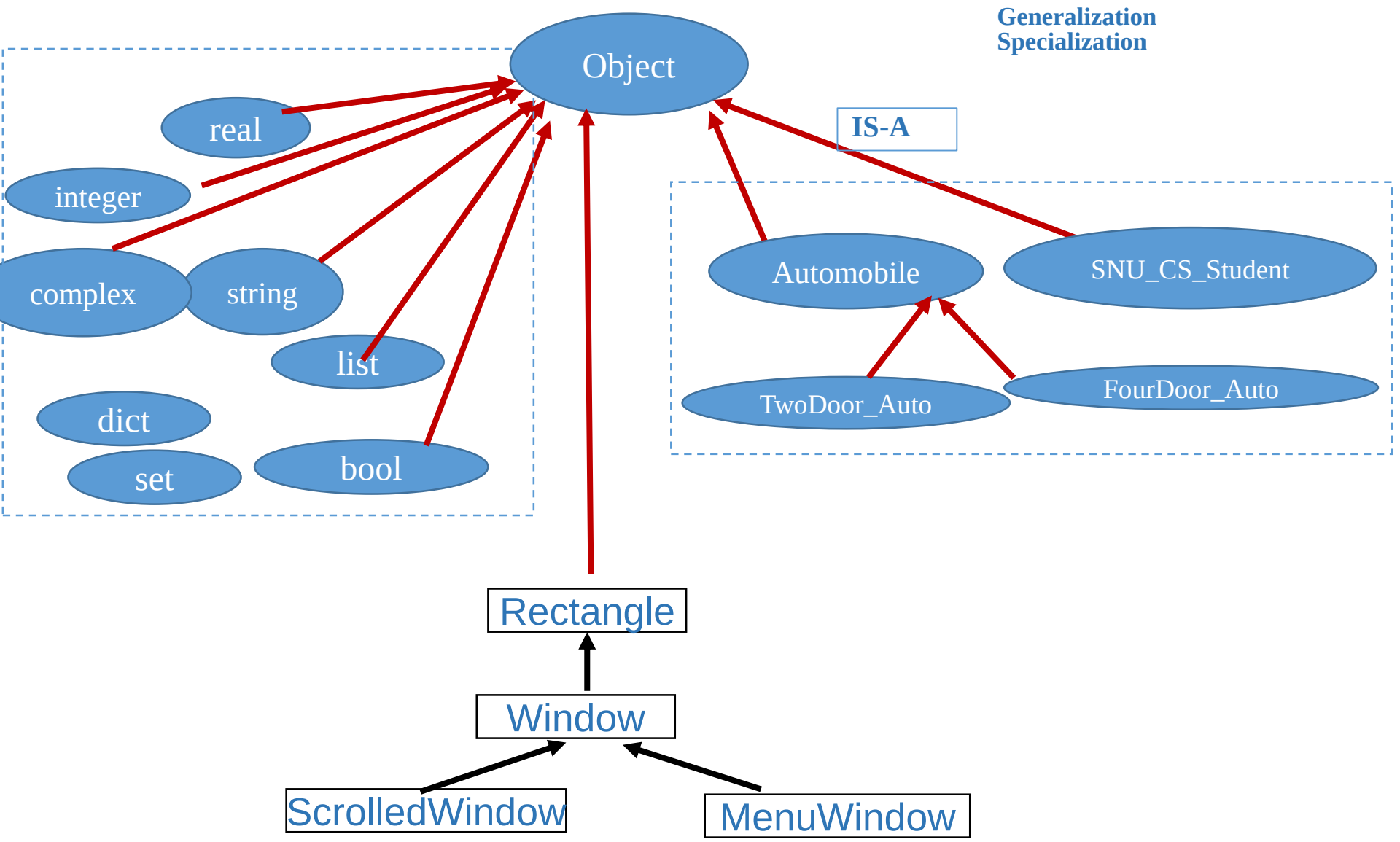


OOP of Banking Account [3/3]

Inheritance ➔ Code Reusability ➔ More Cleaner Way of Coding



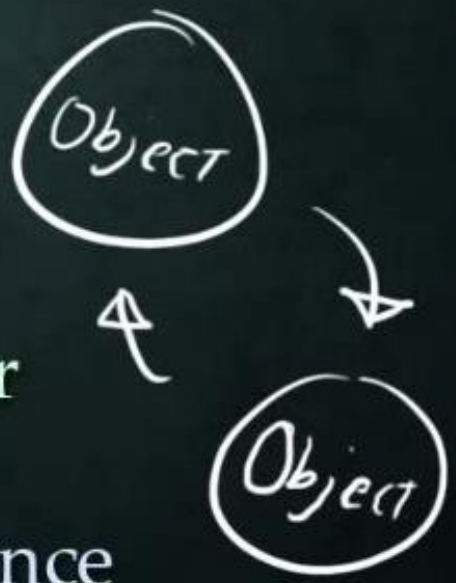
Everything is an Object in OOP



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

OOP 개념과 Python OOP

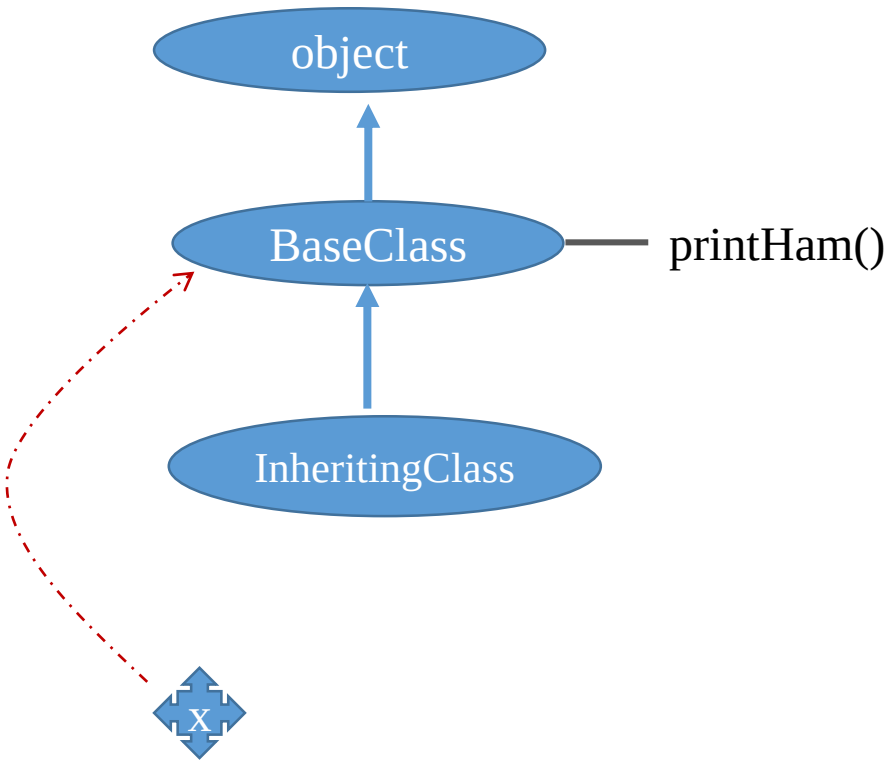
- History of Programming Languages
- Abstract Data Type
- Python OOP Tutorial
- Why do we need classes and OOP?
- More Formal Way of Python OOP

SIMPLE INHERITANCE EXAMPLE

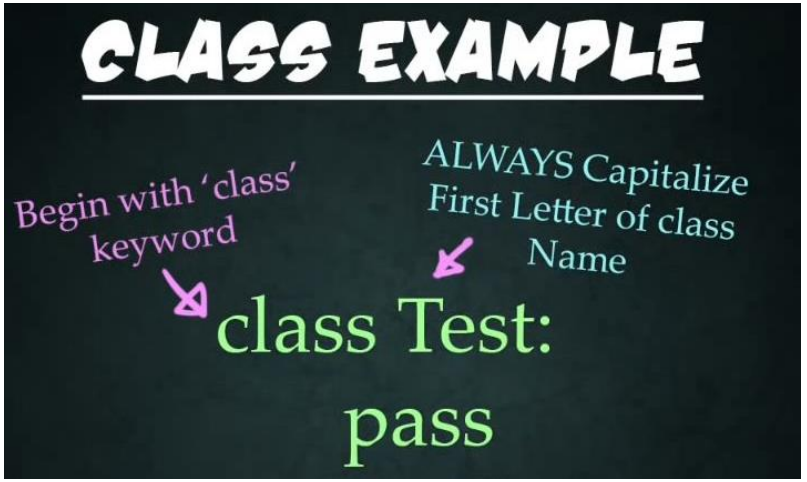
```
class BaseClass (object):
    def printHam(self):
        print 'ham'

class InheritingClass (BaseClass):
    pass
```

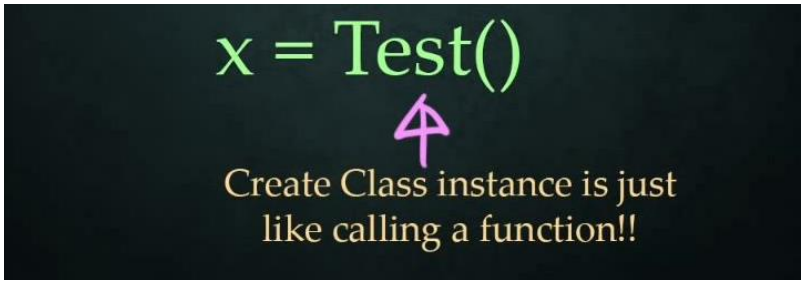
Inherit from this class

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



Object x is created





Attribute ➔ Instance Variable

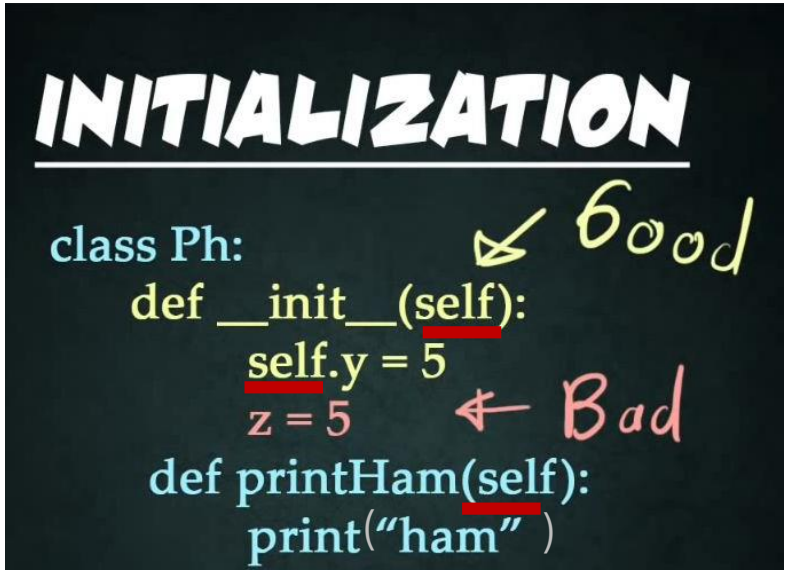
Action ➔ Method (Function)

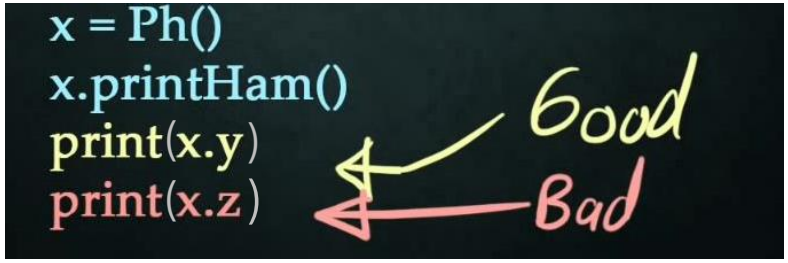
Object에 속한 function들의 선언에는 “self “가 1st parameter로 있어야 한다

```
Python 3.5.1 (v3.5.1:37a07cee5969, Dec  6 2015, 01:38:48) [MSC v.1900
tel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> class Ph():
        def printHam():
                print("ham")


>>> x = Ph()
>>> x.printHam()
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    x.printHam()
TypeError: printHam() takes 0 positional arguments but 1 was given
>>>
```

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

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

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






```
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
>>>
```

Python Class Syntax [1/2]

```
class 클래스이름[(상속 클래스명)]:
    <클래스 변수 1>
    <클래스 변수 2>
    ...
    def 인스탄스 함수1(self[, 인수1, 인수2,,,]):
        <수행할 문장 1>
        <수행할 문장 2>
        ...
    def 인스탄스 함수2(self[, 인수1, 인수2,,,]):
        <수행할 문장1>
        <수행할 문장2>
        ...
    ...
```

```
>>> class Programmer:
>>>     pass
```

```
>>> kim = Programmer()
>>> park = Programmer()
```

Instance Variable
Instance Function
Class Variable
Class Function

Python Class Syntax [2/2]

```
>>> class Service:
...     secret = "영구는 배꼽이 두 개다"     # 유용한 정보
...     def sum(self, a, b):                  # 더하기 서비스
...         result = a + b
...         print("%s + %s = %s입니다." % (a, b, result))
...
>>>
```

```
>>> pey = Service()
```

```
>>> pey.sum(1,1)
1 + 1 = 2입니다.
```

```
>>> class Service:
...     secret = "영구는 배꼽이 두 개다"
...     def __init__(self, name):
...         self.name = name
...     def sum(self, a, b):
...         result = a + b
...         print("%s님 %s + %s = %s입니다." % (self.name, a, b, result))
...
>>>
```

Class variable

Instance variable

Instance method

```
>>> pey = Service("홍길동")
>>> pey.sum(1, 1)
```

```
>>> pey.secret
>>> pey.name
>>> Service.secret
```

OOP 개념과 Python OOP

- History of Programming Languages
- Abstract Data Type
- Python OOP Tutorial
- Why do we need classes and OOP?
- More Formal Way of Python OOP

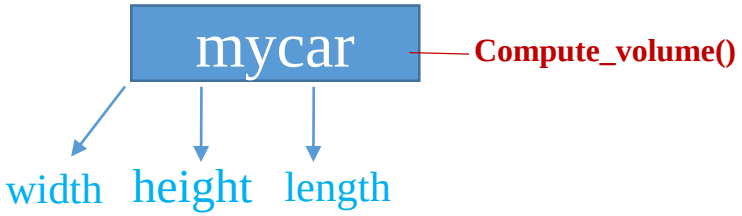
OOP Motivational Example: Auto Volume Computation [1/2]

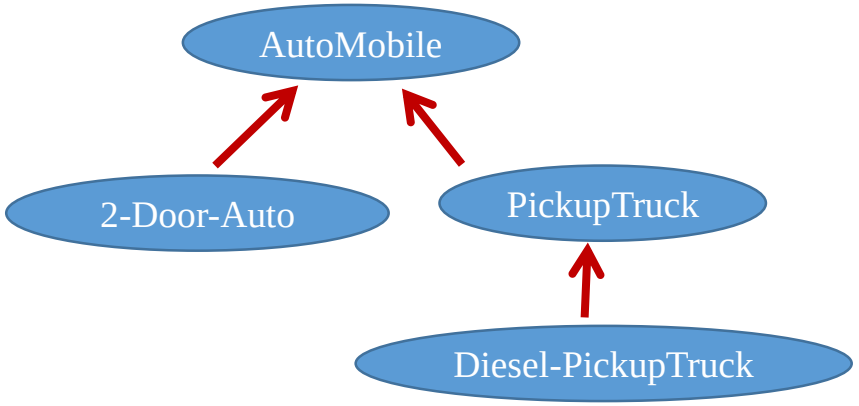
Function-Oriented Python Version

```
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

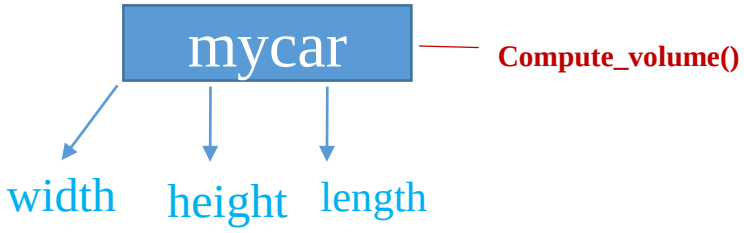


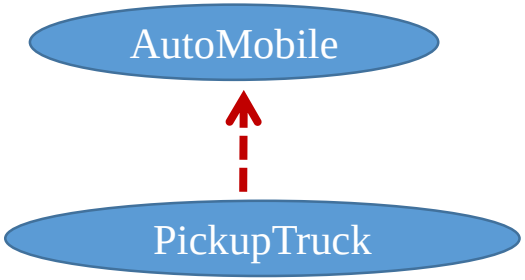


OOP Motivational Example: Auto Volume Computation [2/2]

Object-Oriented Python Version

```
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()
```





OOP Motivational Example: Bank Account [1/2]

```
balance = 0

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

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

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?

OOP Motivational Example: Bank Account [2/2]

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)
```

OOP Motivational Example: Calculator [1/3]

만약 한 프로그램에서 2개의 계산기가 필요하다면 어떻게 해야 할까?

각각의 계산기는 각각의 결과값을 유지해야 하므로 adder function 1개로는 결과값을 따로 유지할수 없다

```
result = 0

def adder(num):
    global result
    result += num
    return result

print(adder(3))
print(adder(4))
```

```
3
7
```

```
result1 = 0
result2 = 0

def adder1(num):
    global result1
    result1 += num
    return result1

def adder2(num):
    global result2
    result2 += num
    return result2

print(adder1(3))
print(adder1(4))
print(adder2(3))
print(adder2(7))
```

```
3
7
3
10
```

OOP Motivational Example: Calculator [2/3]

만약 10개의 계산기가 필요하다면 어떻게 해야 할까?

각각의 계산기는 각각의 결과값을 유지해야 하므로 10개의 adder function을 각각 만들어야 하나?

```
class Calculator:
    def __init__(self):
        self.result = 0

    def adder(self, num):
        self.result += num
        return self.result

cal1 = Calculator()
cal2 = Calculator()
```

```
print(cal1.adder(3))
print(cal1.adder(4))
print(cal2.adder(3))
print(cal2.adder(7))
```

실행하면 함수 2개를 사용했을 때와 동일한 결과가 출력

```
3
7
3
10
```

OOP Motivational Example: Calculator [3/3]

Python FourCal Class

```
>>> class FourCal:
...     def setdata(self, first, second):
...         self.first = first
...         self.second = second
...     def sum(self):
...         result = self.first + self.second
...         return result
...     def mul(self):
...         result = self.first * self.second
...         return result
...     def sub(self):
...         result = self.first - self.second
...         return result
...     def div(self):
...         result = self.first / self.second
...         return result
```

```
>>> a = FourCal()
>>> b = FourCal()
>>> a.setdata(4, 2)
>>> b.setdata(3, 7)
>>> a.sum()
6
>>> a.mul()
8
>>> a.sub()
2
>>> a.div()
2
>>> b.sum()
10
>>> b.mul()
21
>>> b.sub()
-4
>>> b.div()
0
```

Understanding Inheritance [1/3]

```
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

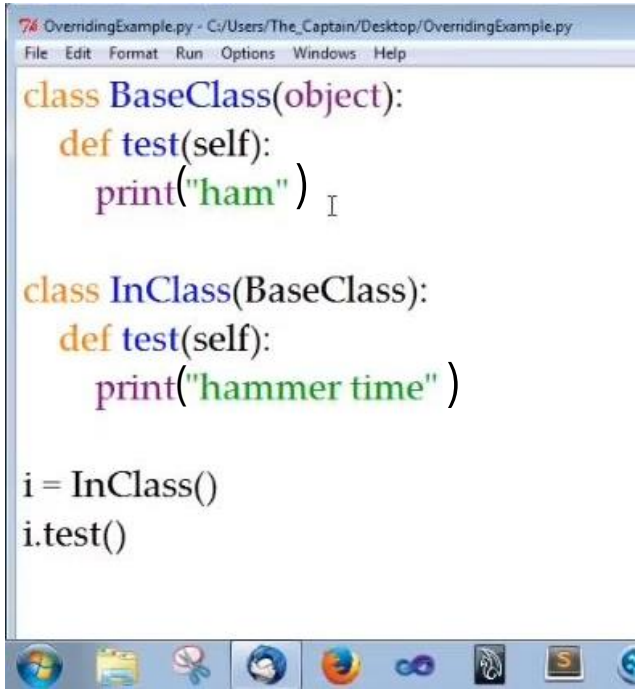
```
class Foo(object):
   def __init__(self):
      self.health = 100

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

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

```
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 받지 않는다

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

Understanding Inheritance [2/3]

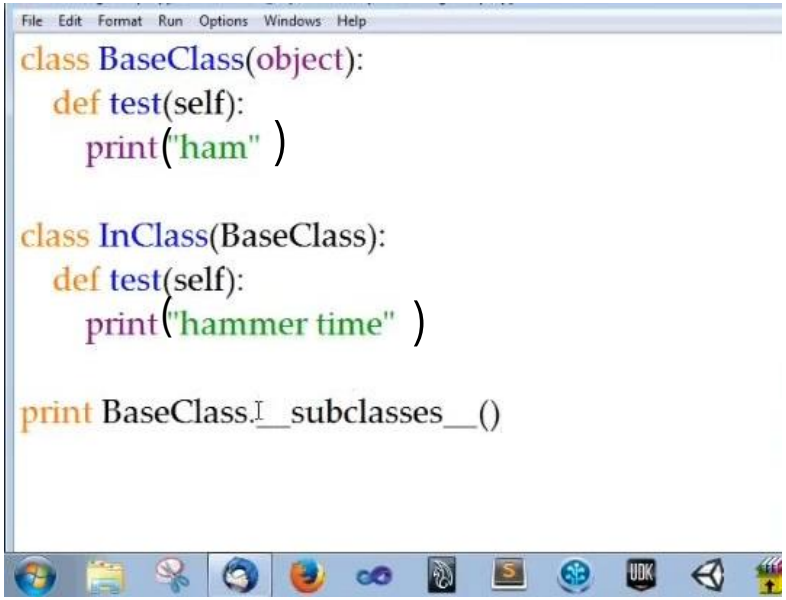



```
class BaseClass(object):
    def __init__(self):
        self.x = 100
class InClass(BaseClass):
    def __init__(self):
        super(InClass, self)__init__()
        self.y = 200

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

Understanding Inheritance [3/3]




```
>>> A = InClass()
>>> A.test()
>>> BaseClass.__subclasses__()
>>> InClass.__superclasses__()
```

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

** __subclasses__() 는 subclass들을 return하는 함수
** __superclasses__()는 없음

Practice of Class Inheritance [1/5]

```
>>> class HousePark:
...     lastname = "박"
...     def __init__(self, name):
...         self.fullname = self.lastname + name
...     def travel(self, where):
...         print("%s, %s여행을 가다." % (self.fullname, where))
```

```
>>> pey = HousePark()
TypeError: __init__() takes exactly 2 arguments (1 given)
```

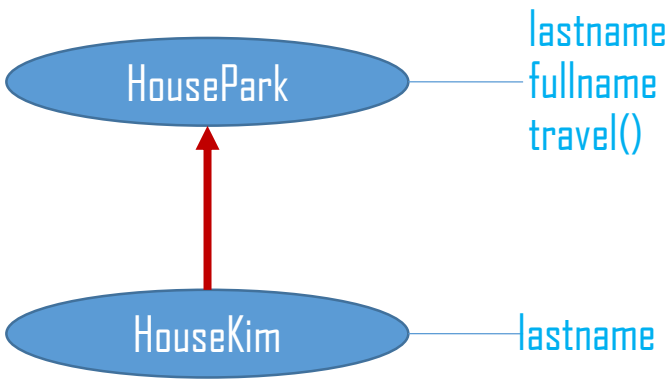
```
>>> pey = HousePark("응용")
>>> pey.travel("태국")
박응용, 태국여행을 가다.
```

Practice of Class Inheritance [2/5]

```
>>> class HousePark:
...     lastname = "박"
...     def __init__(self, name):
...         self.fullname = self.lastname + name
...     def travel(self, where):
...         print("%s, %s여행을 가다." % (self.fullname, where))
```

```
>>> class HouseKim(HousePark):
...     lastname = "김"
```

```
>>> juliet = HouseKim("줄리엣")
>>> juliet.travel("독도")
김줄리엣, 독도여행을 가다.
```

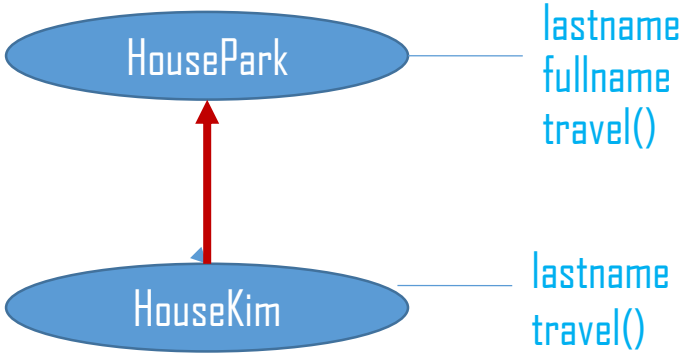


Practice of Class Inheritance [3/5]

```
>>> class HousePark:
...     lastname = "박"
...     def __init__(self, name):
...         self.fullname = self.lastname + name
...     def travel(self, where):
...         print("%s, %s여행을 가다." % (self.fullname, where))
```

```
>>> class HouseKim(HousePark):
...     lastname = "김"
...     def travel(self, where, day):
...         print("%s, %s여행 %d일 가네." % (self.fullname, where, day))
```

```
>>> juliet = HouseKim("줄리엣")
>>> juliet.travel("독도", 3)
김줄리엣, 독도여행 3일 가네.
```



Practice of Class Inheritance [4/5]

```
class HousePark:
    lastname = "박"
    def __init__(self, name):
        self.fullname = self.lastname + name
    def travel(self, where):
        print("%s, %s여행을 가다." % (self.fullname, where))
    def love(self, other):
        print("%s, %s 사랑에 빠졌네" % (self.fullname, other.fullname))
    def __add__(self, other):
        print("%s, %s 결혼했네" % (self.fullname, other.fullname))


class HouseKim(HousePark):
    lastname = "김"
    def travel(self, where, day):
        print("%s, %s여행 %d일 가네." % (self.fullname, where, day))
```

```
>>> pey = HousePark("응용")
>>> juliet = HouseKim("줄리엣")
>>> pey + juliet
```

```
박응용, 김줄리엣 결혼했네
>>>
```

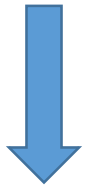
```
pey = HousePark("응용")
juliet = HouseKim("줄리엣")
pey.love(juliet)
pey + juliet
```

```
박응용, 김줄리엣 사랑에 빠졌네
박응용, 김줄리엣 결혼했네
```

Practice of Class Inheritance [5/5]

```
class HousePark:
    lastname = "박"
    def __init__(self, name):
        self.fullname = self.lastname + name
    def travel(self, where):
        print("%s, %s여행을 가다." % (self.fullname, where))
    def love(self, other):
        print("%s, %s 사랑에 빠졌네" % (self.fullname, other.fullname))
    def fight(self, other):
        print("%s, %s 싸우네" % (self.fullname, other.fullname))
    def __add__(self, other):
        print("%s, %s 결혼했네" % (self.fullname, other.fullname))
    def __sub__(self, other):
        print("%s, %s 이혼했네" % (self.fullname, other.fullname))

class HouseKim(HousePark):
    lastname = "김"
    def travel(self, where, day):
        print("%s, %s여행 %d일 가네." % (self.fullname, where, day))
```

```
pey = HousePark("응용")
juliet = HouseKim("줄리엣")
pey.travel("부산")
juliet.travel("부산", 3)
pey.love(juliet)
pey + juliet
pey.fight(juliet)
pey - juliet
```



```
박응용 부산여행을 가다.
김줄리엣 부산여행 3일 가네.
박응용, 김줄리엣 사랑에 빠졌네
박응용, 김줄리엣 결혼했네
박응용, 김줄리엣 싸우네
박응용, 김줄리엣 이혼했네
```

Code Inheritance의 좋은 예제이나 ISA Semantics는 성립안함

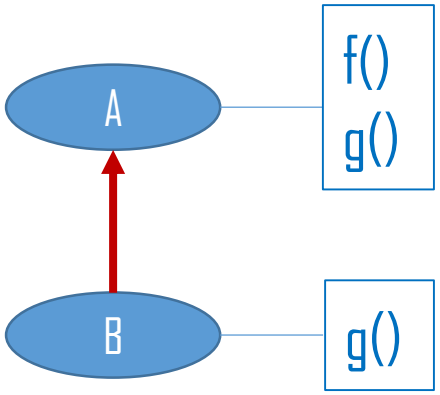
Superclass로 "object"를 선언할때

```
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
```

What will be the output of the following program?

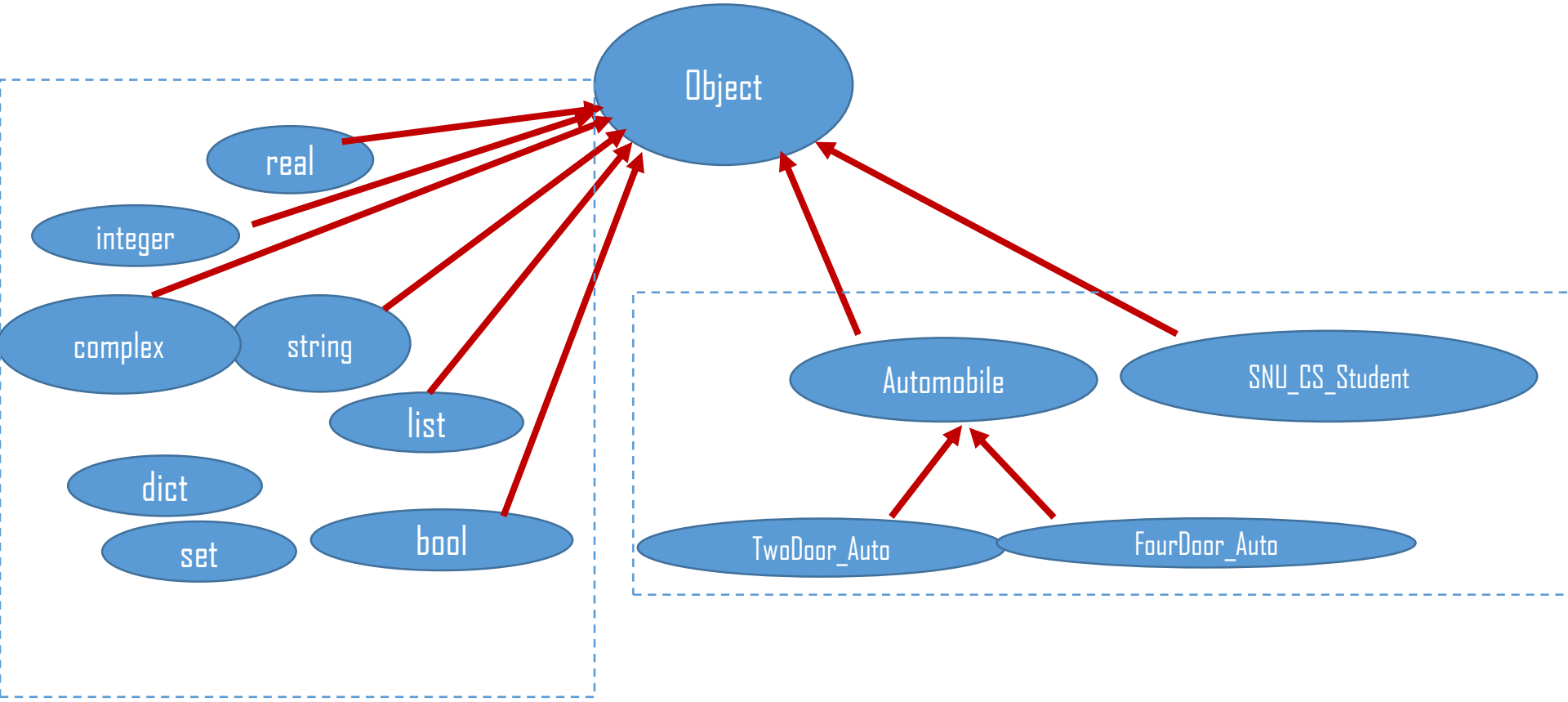
```
class A:
    def f(self):
        return self.g()

    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())
```



OOP 개념과 Python OOP

- History of Programming Languages
- Abstract Data Type
- Python OOP Tutorial
- Why do we need classes and OOP?
- More Formal Way of Python OOP
  - Class Definition
  - Class Inheritance
  - Class Accessibility
  - Class Variable and Class Method
  - Static Method
  - Special Class Methods (double underline methods __foo__( ))

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()
```

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" in Python

- "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)

OOP 개념과 Python OOP

- History of Programming Languages
- Abstract Data Type
- Python OOP Tutorial
- Why do we need classes and OOP?
- More Formal Way of Python OOP
  - Class Definition
  - Class Inheritance
  - Class Accessibility
  - Class Variable and Class Method
  - Static Method
  - Special Class Methods (double underline methods __foo__( ))

Class Inheritance [1/2]

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
```

Class Inheritance [2/2]

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
.....
```

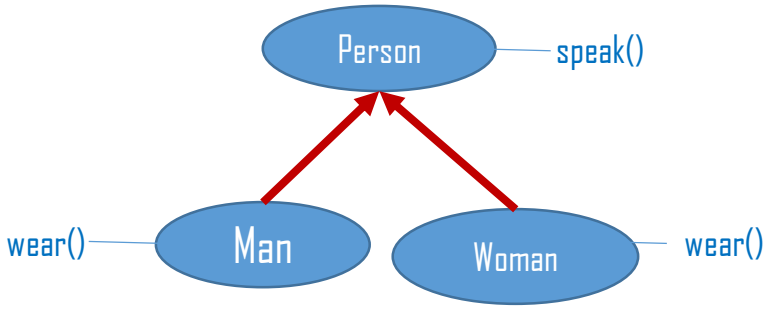
You can use issubclass() or isinstance() functions to check a relationships of two classes and instances.

- 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

Inheritance in Python

```
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
```

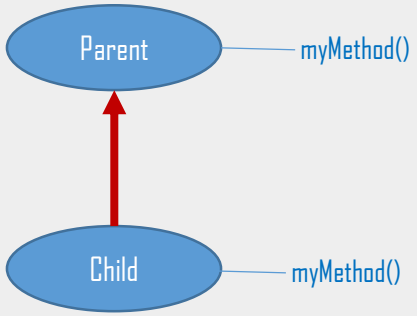


Reusing Methods and Overriding Methods

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
```

Overrriding Built-in Functions

```
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')
```

원래 setAttr() 이나 getAttr()는 built_in function으로 있는데 필요에 따라서 setAttr() 이나 getAttr()를 locally define

```
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
```

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

User-defined Vector class

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:

```
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)
```

Multiple Inheritance in Python

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

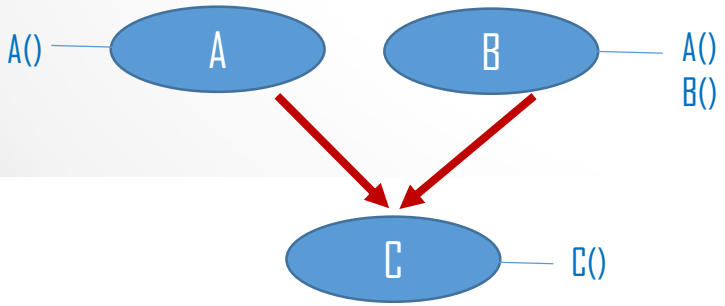
```
class DerivedClass(Base1, Base2, Base3 …)
        <statement-1>
        <statement-2>
        …
```

- 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.

A 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.

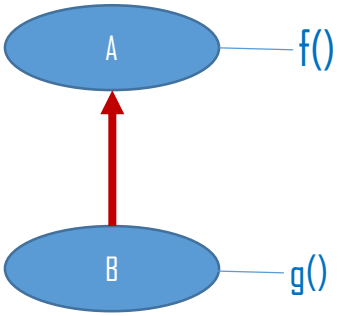



```
class A:
    def A(self):
        print('I am A')

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()
```

Inheritance Example [1/3]

Specifying a Superclass video

```
class A(object):
    def __init__(self, x):
        self.x = x
    def f(self):
        return 10*self.x

class B(A):
    def g(self):
        return 1000*self.x
```



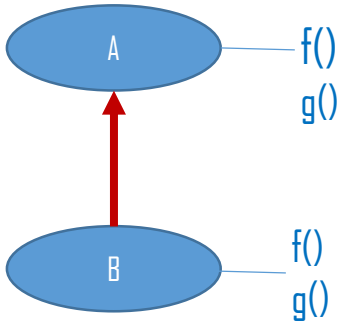

```
print(A(5).f()) # 50
print(B(7).g()) # 7000
print(B(7).f()) # 70 (class B inherits the method f from class A)
print(A(5).g()) # crashes (class A does not have a method g)
```

Inheritance Example [2/3]

Overriding methods video

```
class A(object):
    def __init__(self, x):
        self.x = x
    def f(self):
        return 10*self.x
    def g(self):
        return 100*self.x

class B(A):
    def __init__(self, x=42, y=99):
        super().__init__(x) # call overridden init!
        self.y = y
    def f(self):
        return 1000*self.x
    def g(self):
        return (super().g(), self.y)
```

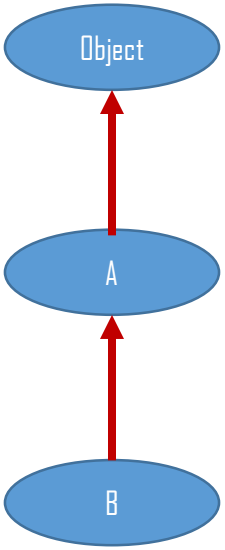
```
a = A(5)
b = B(7)
print(a.f()) # 50
print(a.g()) # 500
print(b.f()) # 7000
print(b.g()) # (700, 99)
```



Inheritance Example [3/3]

isinstance vs type in inherited classes

```
class A(object): pass
class B(A): pass
```




```
a = A()
b = B()
print(type(a) == A) # True
print(type(b) == A) # False
print(type(a) == B) # False
print(type(b) == B) # True
print()
print(isinstance(a, A)) # True
print(isinstance(b, A)) # True (surprised?)
print(isinstance(a, B)) # False
print(isinstance(b, B)) # True
```

OOP 개념과 Python OOP

- History of Programming Languages
- Abstract Data Type
- Python OOP Tutorial
- Why do we need classes and OOP?
- More Formal Way of Python OOP
  - Class Definition
  - Class Inheritance
  - Class Accessibility
  - Class Variable and Class Method
  - Static Method
  - Special Class Methods (double underline methods __foo__( ))

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.

In Java!
Private variable
Protected variable
Pubic variable

Acquiesce: 묵인하다, connive
Condon: 용서하다

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 [1/2]

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.

```
class JustCounter:
   __secretCount = 0

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

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

count()를 통해서 실행할때는 __secretCount의 값이 access

counter object를 통해서 __secretCount는 access 금지

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'
```

Data Hiding [2/2]

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
```

object.__attrName # access is not allowed

object._className__attrName # access is allowed

OOP 개념과 Python OOP

- History of Programming Languages
- Abstract Data Type
- Python OOP Tutorial
- Why do we need classes and OOP?
- More Formal Way of Python OOP
  - Class Definition
  - Class Inheritance
  - Class Accessibility
  - Class Variable and Class Method
  - Static Method
  - Special Class Methods (double underline methods __foo__( ))
  - Double Underline Functions in Python OOP

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
```

class variable

instance variable

```
John = AAA_Club("John", 123)
Bob  = AAA_Club("Bob", 124)

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

className.classVar

object.instanceVar

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 with a class variable empCount

```
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)
```

class variable

```
"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)
```

className.classVar

Result

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

Built-In Functions: hasattr(), getattr(), setattr(), delattr()

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(emp1, 'age')    # Delete attribute 'age'
```

Built-in Class Variables

- **__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.

```
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__ )
```

```
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>}
```

OOP 개념과 Python OOP

- History of Programming Languages
- Abstract Data Type
- Python OOP Tutorial
- Why do we need classes and OOP?
- More Formal Way of Python OOP
  - Class Definition
  - Class Inheritance
  - Class Accessibility
  - Class Variable and Class Method
  - Static Method
  - Special Class Methods (double underline methods __foo__( ))

Static Methods inside Class [1/4]

```
class Fraction(object):
    def __init__(self, num, den):
        self.num = num
        self.den = den
        self.simplify()

    def simplify():
        g = gcd(self.num, self.den)
        self.num = self.num//g
        self.den = self.den//g

    …

def gcd(a, b):
    while (b != 0):
        (a, b) = (b, a%b)
    return a
```

You might decide that you'll only use gcd inside the Fraction class.

You might decide it *belongs* inside the Fraction class.

Yet, it can't really be a method.

Static Methods inside Class [2/4]

```
class Person(object):
    def __init__(self, name, age, gender):
        self.name = name
        self.age = age
        self.gender = gender

    def changeName(self, newName):
        if (isValidName(newName)):
            self.name = newName

    def changeAge(self, newAge):
        ...

def isValidName(name):
    ...
```

isValidName is a helper function (and not a method).

We won't really use it outside of Person class.

And we shouldn't pollute the global space with it.

Static Methods inside Class [3/4]

```
class Fraction(object):
    def __init__(self, num, den):
        self.num = num
        self.den = den
        self.simplify()

    def simplify():
        g = Fraction.gcd(self.num, self.den)
        self.num = self.num//g
        self.den = self.den//g

    @staticmethod
    def gcd(a, b):
        while (b != 0):
            (a, b) = (b, a%b)
        return a
```

Static Methods inside Class [4/4]

```
class Person(object):
    def __init__(self, name, age, gender):
        self.name = name
        self.age = age
        self.gender = gender

    def changeName(self, newName):
        if (Person.isValidName(newName)):
            self.name = newName

    def changeAge(self, newAge):
        ...

    @staticmethod
    def isValidName(name):
        ...
```

OOP 개념과 Python OOP

- History of Programming Languages
- Abstract Data Type
- Python OOP Tutorial
- Why do we need classes and OOP?
- More Formal Way of Python OOP
  - Class Definition
  - Class Inheritance
  - Class Accessibility
  - Class Variable and Class Method
  - Static Method
  - Special Class Methods (double underline methods __foo__( ))

Python의 type system (class structure)

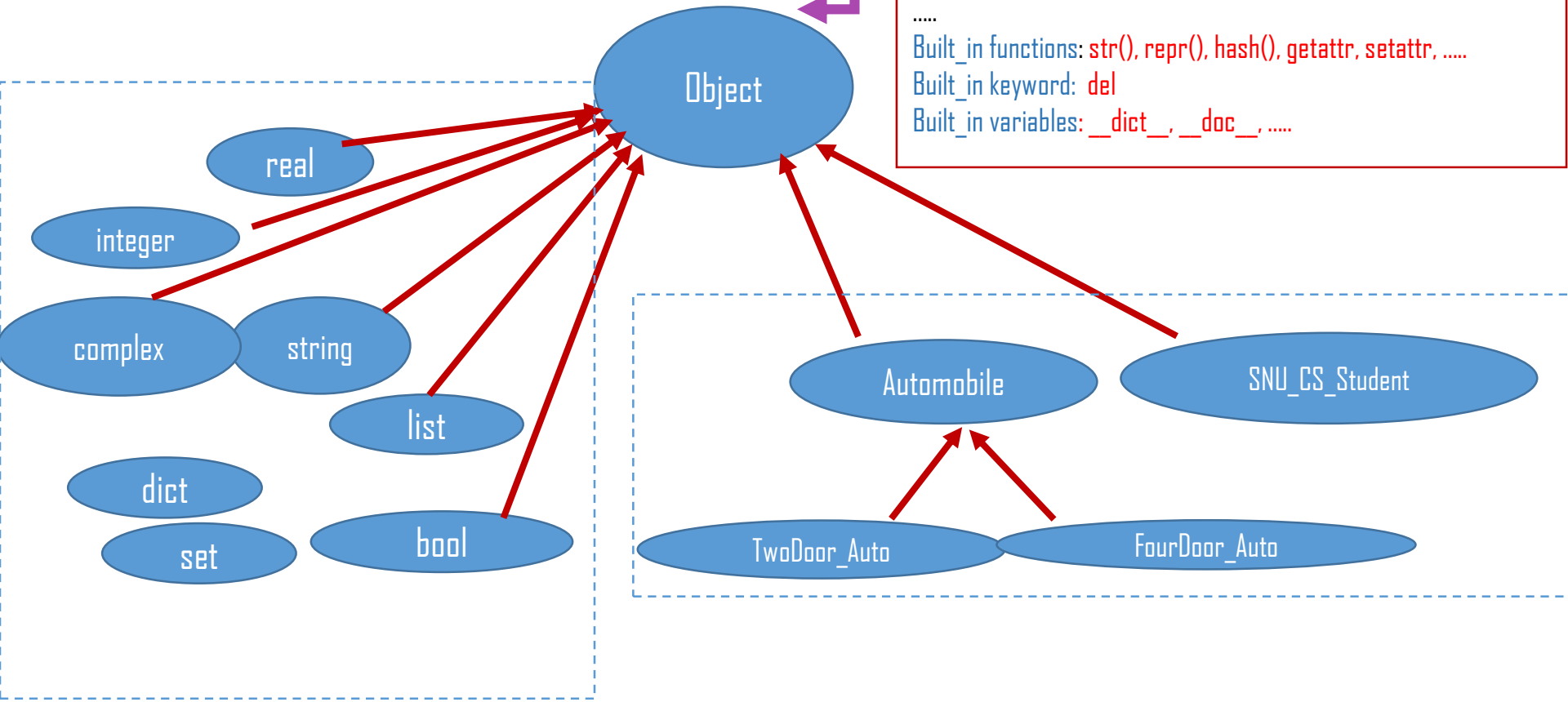
Primitive Operators such as
- Arithmetic operators: +, -, %,....
- Relational operators: > , =>, =.....

.....

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!

object: mother of all classes

```
class Person(object):
    ...
```

object is actually a built-in data type (i.e. class).

When we define a class, we always make it a subclass of object.

What does object contain?

```
>>> dir(object)
['__class__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__',
'__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__le__',
'__lt__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__setattr__', '__sizeof__', '__str__', '__subclasshook__']
```

Python Built-In 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 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()	

참고자료

Python Built-In Operators

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

Python Built-In Class Attributes

Every Python class keeps following built-
attribute:

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

Python Built-In Special Class Methods __foo__()

__str__

__repr__

__hash__

__float__

__lt__

__le__

__gt__

__ge__

__eq__

Accessing Built-in Class Attributes

Direct access to __dict__ (for instances and classes) video

```
class A(object):
    x = 42
    def __init__(self, y):
        self.y = y

a = A(99)
print(a.__dict__) # {'y': 99}
print(A.__dict__) # {'x': 42, ... }

a.__dict__['y'] = 100
print(a.y) # 100
```

```
print(a.__name__) # A
```

Fraction Class를 user가 define했다면 [1/6]

```
class Fraction(object):
    def __init__(self, num, den):
        self.num = num
        self.den = den
        self.simplify()

    def toString(self):
        return str(self.num) + "/" + str(self.den)

    def add(self, other):
        …
    def mul(self, other):
        …
    def toFloat(self):
        …
    def simplify(self):
        …
```

Integer, Float에서 썼던 print()을 Fraction class에서도 쓰고 싶다면!

```
f1 = Fraction(4, 6)
f2 = Fraction(5, 9)
print(f1) <__main__.Fraction object at 0x1010349b0>
print(f1.toString())    2/3
print(f1.add(f2).toString())  11/9

print(f1.__str__())
            <__main__.Fraction object at 0x1010349b0>
```

print implicitly calls object's __str__ method

Fraction Class를 user가 define했다면 [2/6]

```
class Fraction(object):
    def __init__(self, num, den):
        self.num = num
        self.den = den
        self.simplify()

    def __str__(self):
        return str(self.num) + "/" + str(self.den)

    def add(self, other):
        …
    def mul(self, other):
        …
    def toFloat(self):
        …
    def simplify(self):
        …
```

Integer, Float에서 썼던 print()을 Fraction class에서도 쓰고 싶다면!

print()가 __str__()를 call한다는 지식을 알고 있어야!

```
f1 = Fraction(4, 6)
f2 = Fraction(5, 9)
print(f1)    2/3

print(f1.add(f2))    11/9

print(f1.__str__())    2/3
```

print implicitly calls object's __str__ method

Fraction Class를 user가 define했다면 [3/6]

```
class Fraction(object):
    def __init__(self, num, den):
        self.num = num
        self.den = den
        self.simplify()

    def __str__(self):
        return str(self.num) + "/" + str(self.den)

    def __add__(self, other):
        …
    def mul(self, other):
        …
    def toFloat(self):
        …
    def simplify(self):
        …
```

Integer, Float에서 썼던 +을 Fraction class에서도 쓰고 싶다면!

+ 가 __add__()를 call한다는 지식을 알고 있어야!

```
f1 = Fraction(4, 6)
f2 = Fraction(5, 9)
print(f1)    2/3

print(f1 + f2)        11/9
```

+ implicitly calls object's __add__ method

Fraction Class를 user가 define했다면 [4/6]

Be careful implementing these methods!

```
def __eq__(self, other):
    return ((self.num == other.num) and (self.den == other.den))
```

```
f1 = Fraction(4, 6)
f2 = Fraction(2, 3)
f3 = Fraction(2, 4)
print(f1 == f2)        True
print(f1 == f3)        False

print(f1 == 5)         Crash
```

Integer, Float에서 썼던 ==을 Fraction class에서도 쓰고 싶다면!

== 가 __eq__()를 call한다는 지식을 알고 있어야!

```
def __eq__(self, other):
    return (isinstance(other, Fraction) and
            (self.num == other.num) and (self.den == other.den))
```

Fraction Class를 user가 define했다면 [5/6]

What if we try to put our objects in a set?

```
f1 = Fraction(4, 6)
s = set()
s.add(f1)
```

Either crashes, or doesn't work the way you want.

Built-in hash function calls the object's __hash__ method

You need to override __hash__ inherited from object

```
def __hash__(self):
    hashables = (self.num, self.den)
    return hash(hashables)
```

```
def getHashables(self):
    return (self.num, self.den)

def __hash__(self):
    return hash(self.getHashables())
```

Integer, Float class의 instance들이 set에 add할수 있었던 것을 Fraction class에서도 하고 싶다면!

set.add() 가 __hash__()를 call한다는 지식을 알고 있어야!

Fraction Class를 user가 define했다면 [6/6]

One annoying problem:

```
f1 = Fraction(4, 6)
L = [f1]
print(L)        [<__main__.Fraction object at 0x101e34a20>]
```

print actually calls __repr__ for each element of the list.

So you should rewrite __repr__.

print() 문장이 ( )안에 string을 가지고 있으면 _str_()를 call하고,
( )안에 있는 expression을 evaluation을 하고나서 string으로 변화을 해야 하는
상황에서는 _repr_()를 call 한다

Special Class Method __ foo() __ 의 Mechanism

Summary

__str__	Used by built-in str function
__repr__	To create computer readable form
__hash__	Used by built-in hash function
__float__	Used by built-in float function
__lt__	<
__le__	<=
__gt__	>
__ge__	>=
__eq__	==

Overrriding __del__() function

```
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
```

del 은 object를 소멸시키고 __del__()을 수행한다. Point class에서 __del__()를 locally define했으므로 locally define한 __del__()이 call된다

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.