# OBJECT ORIENTED PROGRAMMING
# CLASS 생성, 사용

서울대학교 전기정보공학부
명예교수 성원용

아래 자료와 인터넷 참고 편집
MIT 6.0001 Introduction to Computer Science and Programming in Python

# 절차지향(PROCEDURAL-ORIENTED)



- 명령어들을 위에서부터 아래로 순차적으로 처리하는 것.
- 함수 (function) 중심의 프로그래밍
  - Function 은 return 하고 나면 메모리에 아무것도 안 남는다.
    - Function은 원칙적으로 가지고 있는 state 가 없다.
    - Function 은 stack 을 이용해서 계산을 한다. Stack은 return 시 반납된다
  - 그 까닭으로 바뀐 결과물을 위에서 저장하고, 또 필요하다면 다음 function을 부를 때 그 것을 보내야 한다.
  - 소유물(state)의 중앙집중적 저장

비유: 일을 시키는데 매번 남대문시장에 가서 아무나 불러서 시킨다. 이 사람들에게 매번 manual 을 주고, 그 결과물을 그대로 받아 다가 중앙 창고에 저장해야 한다.
(공산주의 국가의 방식 – 모든 소유물이 국가의 것)

$y = f(x_1, x_2, x_3, \ldots)$, 내부에 state가 없다.

# 객체 지향(OBJECT-ORIENTED)

서울대학교 SEOUL NATIONAL UNIVERSITY

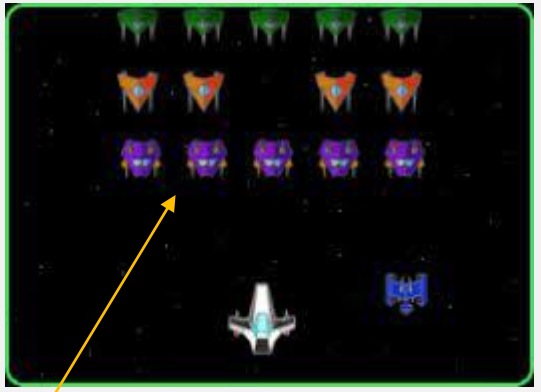
- 독립적인(내부 속성 보존) 객체를 이용하는 프로그래밍
  - 객체 – 자기(self)의 state(이름, 재산, 에너지 등) 소유
- 그런데 객체는 너무 개수가 많다. 어떻게 이들을 다 정의하지? 공통적인 틀(주형)을 이용하여 찍은 후 개별적인 성질을 심어주자. 공통적인 틀이 클래스(Class)
- 이 Class 내에는 어떤 객체가 가지는 state (재산)와 행동(class specific function – method 라 한다)을 정의하여 모아둔다. 이를 capsule화라 한다.
  - 사람 class 의 state (재산, 직장, 나이, 성별)
  - Method – 떠든다, 일한다, 술마신다.
  - 개class 의 state(품종, 나이, 성별)
  - Method – 짖는다, 꼬리친다, 사료먹는다.

재산(state)과
능력(function)
은 서로 대조적인 개념

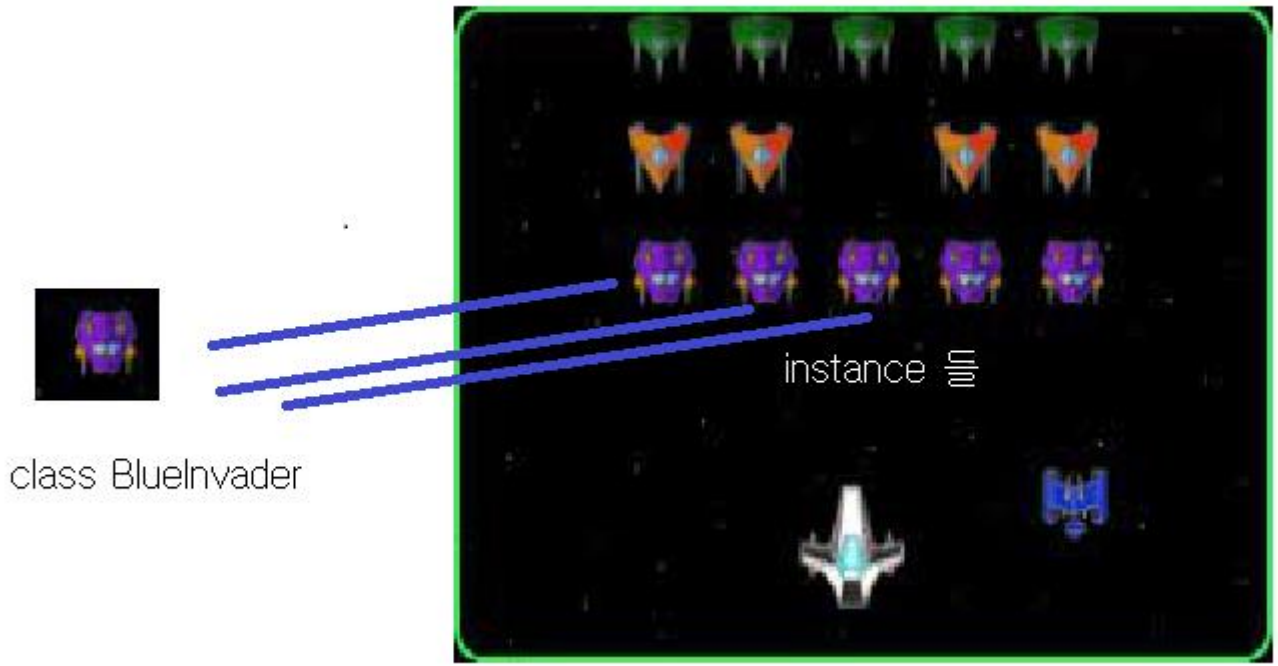
- 돈많은 맨날 놀며 술만 먹는 재벌 *세금수저
- 돈은 없지만 공부잘해 좋은 회사다니는 흑수저 -> 능력이 state로 되는데는 시간이 걸림

- 함수
- y = f(x1, x2, x3,…)
- 입력 x1, x2, … 에 따라 출력 y가 바로 결정이 된다.
- 내부에 state 가 없다. 즉 함수를 빠져 나와도 남아 있는 것이 없다.

- Class (그리고 이 것으로 만든 instance)
- y = f(x1, x2, x3, …, s1, s2)
  (s1, s2 는 내부에 저장된 것으로 instance 나 class를 쓰고 나와도 남아있다가 다음에 들어가서 찾아서 쓴다)



각각 위치, 속도 등의 state 가 있다.




중요한 개념 (instance) – 어떤 instance를 class를 이용하여 정의하면, 그 instance 에는 사유물이 허용이 된다 (self). 일을 시킬 때 어떤 사유물이 있는 (skill level 등) 사람을 불러다가 시킨다. 이 때 사람마다 instance를 정의하지 않고, 공통적 특징을 class로 정의하고 각 instance (개인)에는 self라는 것을 이용하여 사유물을 관리케한다.
이 class 나 instance variable은 일부러 없애기 전에는 없어지지 않는다 (function과 다름)

우리가 만든 class로 찍어낸 object 를 instance 라 한다. Python에서 object는 모든 것을 포함

# 객체 지향(OBJECT-ORIENTED) PROGRAMMING 장점



- 독립적으로 복수가 존재할 수 있는 객체를 이용한 프로그래밍
  - 객체 – 내부에 자기(self)만의 state(재산, 위치, 에너지 등 등)를 가질 수 있다. State 관리를 분산화 시켰다. 재산 관리를 개인에 맡긴 것에 비유.
- 캡슐화 : 클래스와 그에 속한 메소드(function) 를 묶어서 관리할 수 있다. 누구나 access 하는 function 에 비해서 관리가 편하다.
  - 성원용.논문쓴다().
  - 강아지.논문쓴다() <- 에러 남.
- 상속 : 이미 존재하는 클래스를 재활용하며 프로그램을 확장할 수 있다. 사람 class -> 학생 class -> 대학생 class
- 다형성 : 하나의 클래스 및 이름으로 다양한 상황에 대처할 수 있다. Battery -> Duracell, Energizer

# FUNCTION 과 CLASS 비교



- y = f1(x0, x1, x2, …)
  - 입력 (x0, x1, x2, …)이 같다면 언제나 같은 결과이다.
  - 내부에 따로 저장장치가 없다 가정한다 (일부 global 변수의 값을 바꾸면 달라지는데, 이 까닭으로 global 변수 사용을 가급적 하지 않도록 한다.)

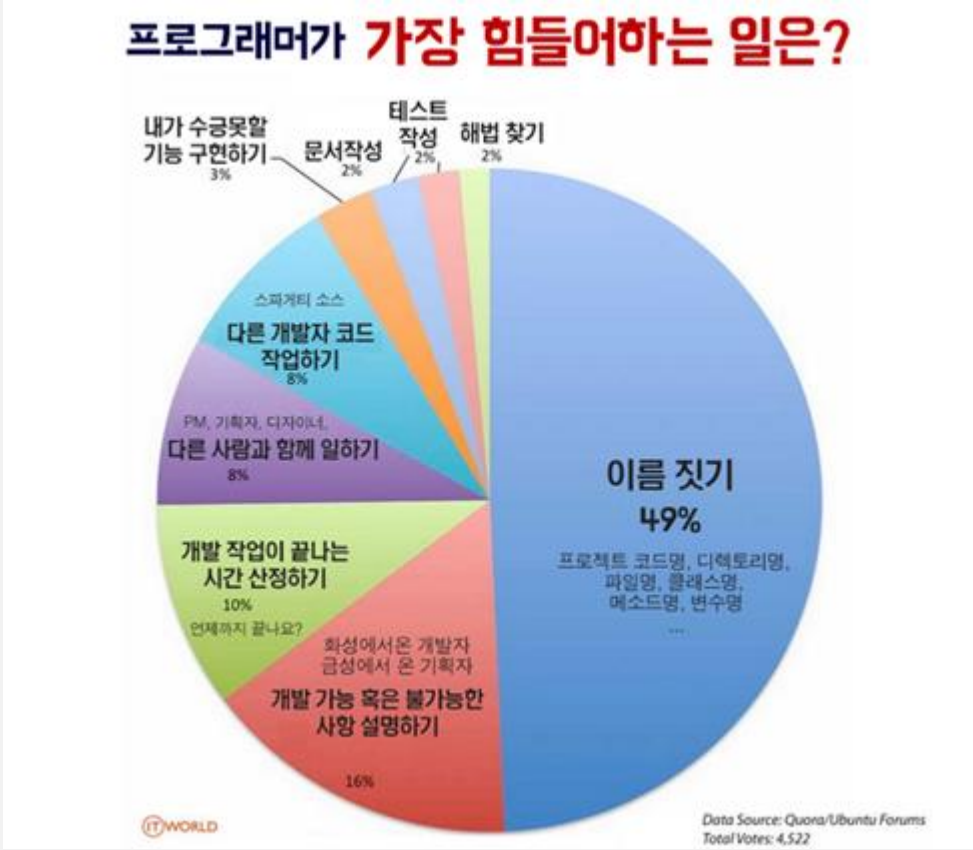
- Class C -> instance (O0, O1, O2…)를 생성
  O0, O1, O2 는 개별 state(s0, s1, s2,..)를 소유.
- y0 = C(x0, x1, x2, …, s0)
- y1 = C(x0, x1, x2, …, s1)

# PROCEDURAL VS OBJECT ORIENTED PROGRAMMING

- 절차지향적 programming (대표적으로 C 언어)
- 큰 task (program)를 함수(procedure) 단위로 나누어서 구조화한다.
- 상위에서 하위로 쪼개가기 때문에 top-down 방식
- 문제: 함수는 구조화를 하였지만, data 는 구조화하지 못해서 global name space 포화문제가 발생한다.
- Data 상태에 따라 동작이 다른 함수들이 많아진다. <- debugging 이 어렵다.

* Programmer 가 가장 힘들어하는 것 – 이름 찾기 (

- Object oriented programming (C++, Python)
- 캡슐화, 상속, 다형성
- 하위(object)에서 상위로 간다. Down-top.
- 외부에서 사용하는 interface와는 달리 내부의 information 이 숨겨질 수 있다.
- 객체들이 독립적 – 어떤 객체 수정 시 다른 것에 덜 영향 (class 별 method가 정의)
- Class의 상속 등의 특징 때문에 코드 재사용 쉽다.


프로그래머가 가장 힘들어하는 일은?
내가 수긍못할 기능 구현하기 3%
문서작성 2%
테스트 작성 2%
해법 찾기 2%
스파게티 소스
다른 개발자 코드 작업하기 8%
PM, 기획자, 디자이너,
다른 사람과 함께 일하기 8%
개발 작업이 끝나는 시간 산정하기 10%
언제까지 끝나요?
화성에서온 개발자
금성에서 온 기획자
개발 가능 혹은 불가능한 사항 설명하기 16%
이름 짓기 49%
프로젝트 코드명, 디렉토리명, 파일명, 클래스명, 메소드명, 변수명 ...
IT WORLD
Data Source: Quora/Ubuntu Forums
Total Votes: 4,522

# CLASS

서울대학교 SEOUL NATIONAL UNIVERSITY

- 정의하고자 하는 개념을 구체화시킨 틀
- 클래스는 자신에게 속하는 멤버 변수(state)와 함수를 지니고 있다.
- 규정된 클래스를 이용하여 해당 클래스의 형태를 가지는 객체(instance) 를 찍어낼 수 있다.
- 클래스와 함수를 정의하는 것으로 프로그래밍은 무한한 확장성과 활용성을 지니게 된다.

ex) 클래스는 키, 몸무게, 이름이라는 멤버 변수와 자기, 먹기, 놀기 등의 함수(method)를 가지는 사람이라는 개념으로 정의할 수 있다. 이 클래스를 이용하여 철수, 영희, 민수 등 객체를 찍어낼 수 있는 것.

# 객체와 인스턴스



- 구체화된 모든 클래스를 객체(instance, object)라고 칭한다.
  - 사람은 class
  - 김철수, 성원용, 최석현은 instance
- 인스턴스는 일반적으로 특정 클래스로부터 유래된 객체를 칭한다.(관계적인 의미) 철수는 Human 의 인스턴스이자 객체다.
- 클래스의 내부를 수정하는 것은 틀을 수정하는 것으로, 생성되는 모든 인스턴스에 영향을 준다.
- 인스턴스 내부의 값을 바꾸는 것 (김철수가 가진 돈을 늘린다) 은 해당 인스턴스에게만 해당되며, 틀인 클래스나 다른 인스턴스에는 영향을 주지 않는다.

# CLASS 선언



- Class의 이름은 첫글자를 대문자로 쓴다. 그리고 (object)를 넣을 수도 있고, 안 넣어도 된다.  나중에 hierarchical class 생성에서는 이 괄호안에 parent class 이름을 넣는다.
- Class내에서 그 class를 지금 이용하는 어떤 'instance' 를 지칭하기 위해 self 를 사용한다. 자기.
- Class는 여러가지의 function (method 라 한다) 을 포함하는데, 처음에 생성시 한번만 구동되는 것이 def __init__(self, …) 이다.
- 어떤 class에 처음 생성자에 필요한 argument 를 주고 부르면 instance 가 찍어진다.

```python
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def normsq(self):
        nsq = self.x**2 + self.y**2
        return nsq

    def move(self, xinc, yinc):
        self.x += xinc
        self.y += yinc

p1 = Point(0.0, 1.0)
p2 = Point(1.0, 2.0)
p1.move(1, 2)
print(p1.x, p1.y, p1.normsq())  #1.0  3.0  10.0
```

<- instance 를 생성하는 부분

여기에서 self 는 instance의 이름

Class에 정의된 함수 (method 라 한다)

각 instance의 값을 이용할 때 self.를 사용

p1 과 p2 라는 instance를 Point 라는 class를 이용하여 찍어낸다.

def __init__() 은 instance 를 만들 때 한번만 사용된다.

Instance이름.함수명(), instanc이름.변수명 으로 이용할 수 있다.

# 생성자



- class에 __init__ 이라는 이름을 가진 함수가 클래스의 생성자가 된다.
- 생성자는 인스턴스가 만들어 질 때 불러와지며, 새로 생성되는 인스턴스를 구체적으로 어떻게 생성할지를 다룬다.
- 생성자의 self를 이용해 인스턴스 내부 변수와 함수를 스스로 호출하거나 수정할 수 있다.
- **실제로 인스턴스를 호출할 때 인스탄스의 이름이 self** 에 매핑된다.

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
…
p1 = Point(0.0, 1.0)
```

# FUNCTION VS CLASS

```
def func_bark(name_energy):
    print(name_energy[0]+ ' is barking. mung! mung!')
    name_energy[1] = name_energy[1]-1
    print('The energy level is ' + str(name_energy[1]))

john_dog = ['John', 10]    # 0th name, 1st energy
mary_dog =['Mary', 10]
chul_man = ['Chulsoo', 10]

func_bark(john_dog)
func_bark(chul_man)
print(john_dog[1])
-----
John is barking. mung! mung!
The energy level is 9
Chulsoo is barking. mung! mung!
The energy level is 9
9
```

```
class Animal:
    def __init__(self, name, energy):
        self.name = name
        self.energy = energy

    def bark(self):
        print(self.name+ '(dog) is barking. mung! mung!')
        self.energy += -1
        print(self.energy)

john_dog = Animal('John', 10)
mary_dog = Animal('Mary', 10)
john_dog.bark()
john_dog.bark()
#chul_man.bark()    ??
-------
John(dog) is barking. mung! mung!
9
John(dog) is barking. mung! mung!
8
```

함수(function)을 이용하는 방식에서는 데이터를 부르는 곳에서 중앙관리안나.
그리고 함수의 입력으로 다른 종류의 입력이 들어가도 눈치채지 못한다.
Class의 경우에는 그 class에 허용된 함수(method)만 사용을 해야 한다.
(encapsulation) 따라서 data를 객체단위로 관리하고 또 error의 가능성 적다.

# OBJECT ORIENTED PROGRAMMING (OOP)



- Class에 의해서 생성된 instance가 object 의 예
- 사실 Python 에서는 int, string 부터 복잡한 instance까지 모든 것을 object 로 부름
- `1234` `3.14159` `"Hello"` `[1, 5, 7, 11, 13]`

```
{"CA": "California", "MA": "Massachusetts"}
```

- Each is an **object**, and every object has:
  - a **type**
  - an internal **data representation** (primitive or composite)
  - a set of procedures for **interaction** with the object (methods)

# ADVANTAGES OF OBJECT ORIENTED PROGRAMMING (OOP)

서 울 대 학 교
SEOUL NATIONAL UNIVERSITY

- **bundle data into packages** together with procedures (methods) that work on them through well-defined interfaces
- **divide-and-conquer** development
  - implement and test behavior of each class separately
  - increased modularity reduces complexity
- classes make it easy to **reuse** code
  - many Python modules define new classes
  - each class has a separate environment (no collision on function names)
  - inheritance allows subclasses to redefine or extend a selected subset of a superclass' behavior

# CLASS 만들기 (CLASS는 형틀, INSTANCE 는 그것으로 만든 객체)

- Class는 자기 소유물이 있다. 이를 통해서 instance 를 여러 개 만드는데, 이 객체의 자기를 나타내기 위해 self 라는 변수를 이용
- Class 에는 instance 가 만들어질 때 딱 한번만 만들어지는 함수( __init__) 가 있다.
- Class 에서 찍혀진 모든 instance 들이 공유하는 변수가 class variable 이다. 맨 처음에 나오고 self 라는 이름이 없다.
- 각 instance 들이 각자 독립적으로 가지는 변수가 instance variable 이고 self.* 로 표시된다.
- Class variable 은 동일 이름의 instance variable 이 따로 없으면 instance variable 로 사용된다.

```
class Employee:
    no_emps = 0
    raise_amount = 1.04
    def __init__(self,first,last,pay):
        self.first = first
        self.last = last
        self.pay = pay
        self.email = first + '.' +last+ '@company.com'
        Employee.no_emps +=1
    def fullname(self):
        return self.first + ' ' + self.last
    def apply_raise(self):
        self.pay = int(self.pay * self.raise_amount )
```

class variable

instance variable

instance가 없으면

class variable 사용

```
print(Employee.no_emps)                    0
emp_1 = Employee('Chulsoo','Kim',50000)
emp_2 = Employee('Soonhee','Park',80000)
print(Employee.no_emps)                    2
print (emp_1.fullname())                   Chulsoo Kim
print (emp_1.email)                        Chulsoo.Kim@company.com
print (emp_1.pay)
print (emp_2.pay)                          50000
emp_1.apply_raise()                        80000
emp_2.apply_raise()                        52000
print (emp_1.pay)                          83200
print (emp_2.pay)
```

# CLASS DESCRIPTION (개념적)

서울대학교 SEOUL NATIONAL UNIVERSITY

- data and procedures that "**belong**" to the class
- **data attributes**
  - think of data as other objects that make up the class
  - *for example, a coordinate is made up of two numbers*
  - States (What it is, 소유)
- **methods** (procedural attributes)
  - think of methods as functions that only work with this class
  - how to interact with the object
  - What it does (행동, 능력)
  - *for example you can define a distance between two coordinate objects but there is no meaning to a distance between two list objects*

# METHOD



- Procedural attribute, like a **function that works only with this class**
- Python always passes the object as the first argument
  - convention is to use **`self`** as the name of the first argument of all methods
- the "**.**" **operator** is used to access any attribute
  - a data attribute of an object
  - a method of an object

# 이미 우리가 METHOD 는 많이 사용해 왔다. LI[illegible] 경우 (변수 A)

- 값 제거 : a.remove(값)
- 맨 끝에 값 추가 : a.append(값)
- 중간에 값 삽입 : a.insert(위치, 값)
- Pop : a.pop()  //마지막 값을 반환하면서 list에서 삭제함
- 정렬 : a.sort()
- 역전 : a.reverse()
- 확장 : a.extend(b) //a의 뒤에 list b를 추가함

# METHOD를 사용하는 두가지 방법

```
def distance(self, other):
    # code here
```

method def

Using the class:

- conventional way

```
c = Coordinate(3,4)
zero = Coordinate(0,0)
print(c.distance(zero))
```

object to call method on

name of method

parameters not including `self` (`self` is implied to be `c`)

- equivalent to

```
c = Coordinate(3,4)
zero = Coordinate(0,0)
print(Coordinate.distance(c, zero))
```

name of class

name of method

parameters, including an object to call the method on, representing `self`

# INSTANCE의 PRINT (__STR__ METHOD) (1/2)



- `>>> c = Coordinate(3,4)`
- `>>> print(c)`
- `<_main_.Coordinate object at 0x7fa918510488>`

- **uninformative** print representation by default
- define a **`__str__` method** for a class
- Python calls the `__str__` method when used with
- `print` on your class object
- you choose what it does! Say that when we print a
- `Coordinate` object, want to show

- `>>> print(c)`
- `<3,4>`

# __STR__ FOR PRINT (2/2)

```
class Coordinate(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def distance(self, other):
        x_diff_sq = (self.x-other.x)**2
        y_diff_sq = (self.y-other.y)**2
        return (x_diff_sq + y_diff_sq)**0.5
    def __str__(self):
        return "<"+str(self.x)+","+str(self.y)+">"
```

name of special method

must return a string

# SPECIAL METHODS FOR A CLASS (OVERLOADING)



전에 string 두개를 + 로 연결하면 서로 연결이 되었다. print('how ' + 'are you')

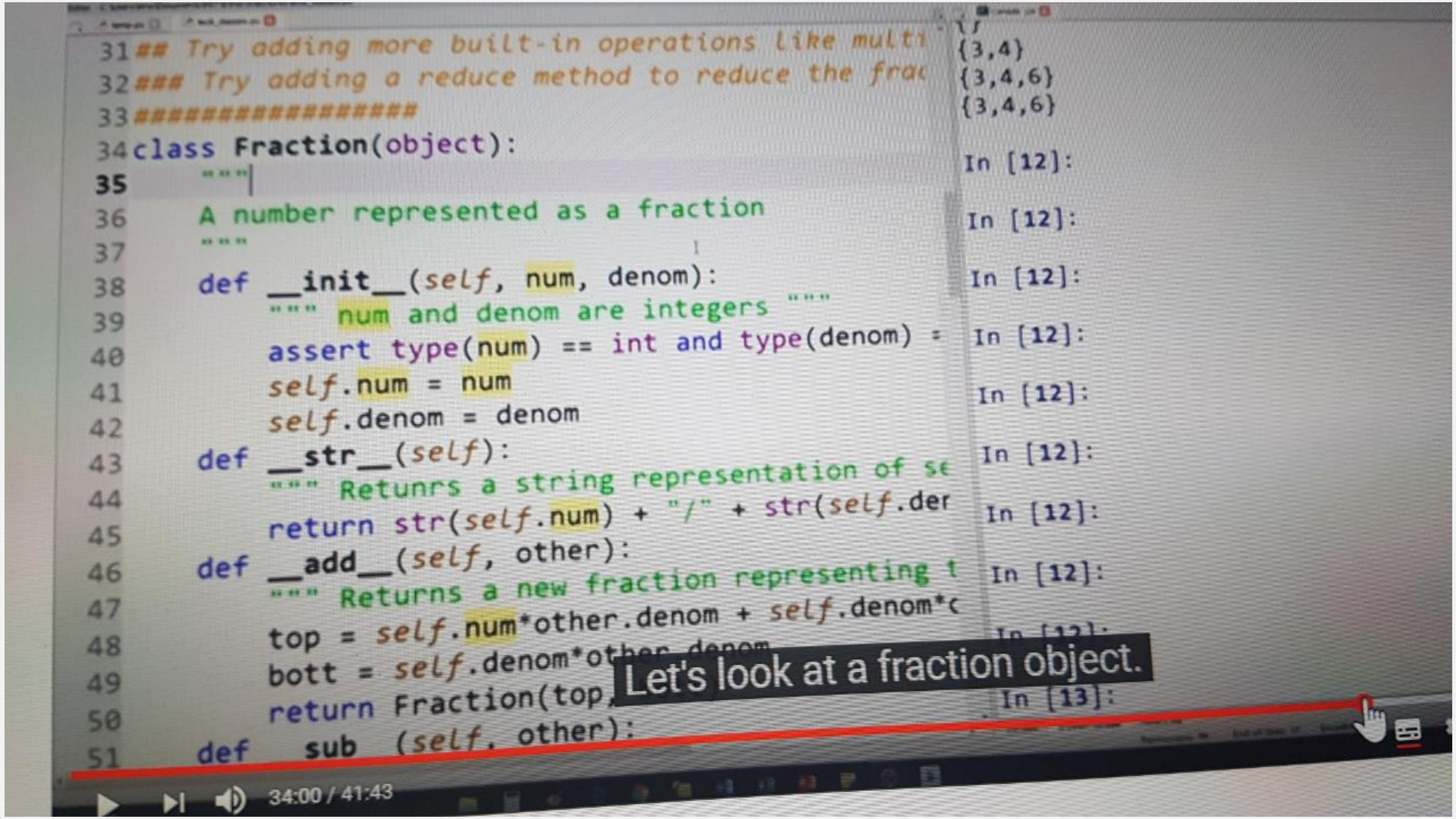
- +, -, ==, <, >, len(), print, and many others
- https://docs.python.org/3/reference/datamodel.html#basic-customization
- like `print`, can override these to work with your class
- define them with double underscores before/after

```
__add__(self, other)    →    self + other
__sub__(self, other)    →    self - other
__eq__(self, other)     →    self == other
__lt__(self, other)     →    self < other
__len__(self)           →    len(self)
__str__(self)           →    Print(self)
```

... and others

# EXAMPLE: FRACTIONS

- create a **new type** to represent a number as a fraction
- **internal representation** is two integers
  - numerator
  - denominator
- **interface** a.k.a. **methods** a.k.a **how to interact** with `Fraction` objects
  - add, subtract
  - print representation, convert to a float
  - invert the fraction
- the code for this is in the handout, check it out!


## Try adding more built-in operations like multi
### Try adding a reduce method to reduce the frac
class Fraction(object):
"""
A number represented as a fraction
"""
def __init__(self, num, denom):
""" num and denom are integers """
assert type(num) == int and type(denom) =
self.num = num
self.denom = denom
def __str__(self):
""" Retunrs a string representation of se
return str(self.num) + "/" + str(self.der
def __add__(self, other):
""" Returns a new fraction representing t
top = self.num*other.denom + self.denom*c
bott = self.denom*other.denom
return Fraction(top,
def __sub__(self, other):
{3,4}
{3,4,6}
{3,4,6}
In [12]:
In [12]:
In [12]:
In [12]:
In [12]:
In [12]:
In [12]:
In [12]:
In [13]:
Let's look at a fraction object.
34:00 / 41:43

# NO GOOD FOR INFORMATION HIDING

- allows you to **access data** from outside class definition

```
print(a.age)
```

- allows you to **write to data** from outside class definition

```
a.age = 'infinite'
```

- allows you to **create data attributes** for an instance from outside class definition

```
a.size = "tiny"
```

- it's **not good style** to do any of these!

# DEFAULT ARGUMENTS

- **default arguments** for formal parameters are used if no actual argument is given

```
def set_name(self, newname=""):
    self.name = newname
```

- default argument used here

```
a = Animal(3)
a.set_name()
print(a.get_name())
```

prints ""

- argument passed in is used here

```
a = Animal(3)
a.set_name("fluffy")
print(a.get_name())
```

prints "fluffy"

# GETTER AND SETTER METHODS

```
class Animal(object):
    def __init__(self, age):
        self.age = age
        self.name = None
    def get_age(self):
        return self.age
    def get_name(self):
        return self.name
    def set_age(self, newage):
        self.age = newage
    def set_name(self, newname=""):
        self.name = newname
    def __str__(self):
        return "animal:"+str(self.name)+":"+str(self.age)
```

getter

setter

- **getters and setters** should be used outside of class to access data attributes

# 범위(scope) – public 위주

Class 와 instance 변수를 외부에서 볼 수 있고, 또 바꿀 수도 있다.
기본적으로 public
<- encapsulation에 나쁨.

더 좋은 방법은 method를 통해서 내부를 access

- good style
- easy to maintain code
- prevents bugs

```
class Person(object):
    def __init__(self, name):
        self.name = name
        print(self.name + 'is initialized')

    def set_work(self, company):
        self.work=company

    def set_home(self, gooname):
        self.home=gooname

    def get_name(self):
        return self.name

    def get_work(self):
        return self.work

    def get_home(self):
        return self.home


obj1 = Person('Sung')
obj1.set_work('SNU')
obj1.set_home('Kangnamego')

obj2 = Person ('Kim')
obj2.set_work('Samsung')
obj2.set_home('Suwon')

print(obj1.get_name(), obj1.get_work(), obj1.get_home())

print(obj1.name, obj1.work, obj1.home)

obj1.home = 'Ilwondon'
print(obj1.name, obj1.work, obj1.home)
```

```
Sungis initialized
Kimis initialized
Sung SNU Kangnamego
Sung SNU Kangnamego
Sung SNU Ilwondon
```

# PRIVATE __MEMBER변수, __멤버METHOD

```
class Person(object):
    ID_number = 0

    def __init__(self, name):
        self.name = name
        print(self.name + 'is initialized')
        Person.ID_number += 1
        self.id = Person.ID_number  #Instance 변수

    def set_work(self, company):
        self.work=company

    def get_name(self):
        return self.name

    def set_wife_name(self, wife_name):
        self.__wife_name = wife_name

    def get_work(self):
        return self.work

    def get_wife_name(self):
        self.__print_wife_name()

    def __print_wife_name(self):
        print(self.__wife_name)

obj1 = Person('Sung')
obj1.set_work('SNU')
obj1.set_wife_name('Kweon')

obj2 = Person ('Kim')
obj2.set_work('Samsung')

obj1.get_wife_name()
obj1.__print_wife_name()
print(obj1.__wife_name)
```

```
Sungis initialized
Kimis initialized
Kweon

---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-37-255bb802ad44> in <module>
     35 obj1.get_wife_name()
     36 #obj1.__print_wife_name()
---> 37 print(obj1.__wife_name)

AttributeError: 'Person' object has no attribute '__wife_name'
```

# PUBLIC, PROTECTED, PRIVATE

서 울 대 학 교
SEOUL NATIONAL UNIVERSITY

- Public – all member variables and methods are public (accessible including modifying from outside) by default in Python (no good sometimes)
- Private - means that nobody should be able to access it from outside the class, i.e. strong you can't touch this policy. Python supports a technique called name mangling. This feature turns every member name prefixed with at least two underscores and suffixed with at most one underscore
- Protected – When you prefix the name of your member with *a single underscore*, you're telling others "don't touch this, unless you're a subclass".

```
class myClass:
  __privateVar = 27;
  def __privMeth(self):
    print("I'm inside class myClass")
  def hello(self):
    print("Private Variable value: ",myClass.__privateVar)
foo = myClass()
foo.hello()
foo.__privateMeth
```

```
Private Variable value: 27
Traceback (most recent call last):
  File "C:/Python/Python361/privateVar1.py", line 12, in
<module>
  foo.__privateMeth
AttributeError: 'myClass' object has no attribute
'__privateMeth'
```

```
class Cup:
    def __init__(self):
        self.color = None
        self._content = None # protected variable

    def fill(self, beverage):
        self._content = beverage
```

```
redCup = Cup()
redCup.color = "red"
redCup.content = "tea"
```

```
class Cup:
    def __init__(self, color):
        self._color = color     # protected variable
        self.__content = None   # private variable

    def fill(self, beverage):
        self.__content = beverage

    def empty(self):
        self.__content = None
```

# OBJECT ORIENTED PROGRAMMING의 장점



- **Bundle together objects** that share
  - common attributes and (내부 states)
  - Procedures that operate on those attributes (methods)
- Use **abstraction** to make a distinction between how to implement an object vs how to use the object
- Build **layers** of object abstractions that inherit behaviors from other classes of objects
- Create our **own classes of objects** on top of Python's basic classes

# CLASS IMPLEMENTATION VS USE

- write code from two different perspectives

**implementing** a new object type with a class

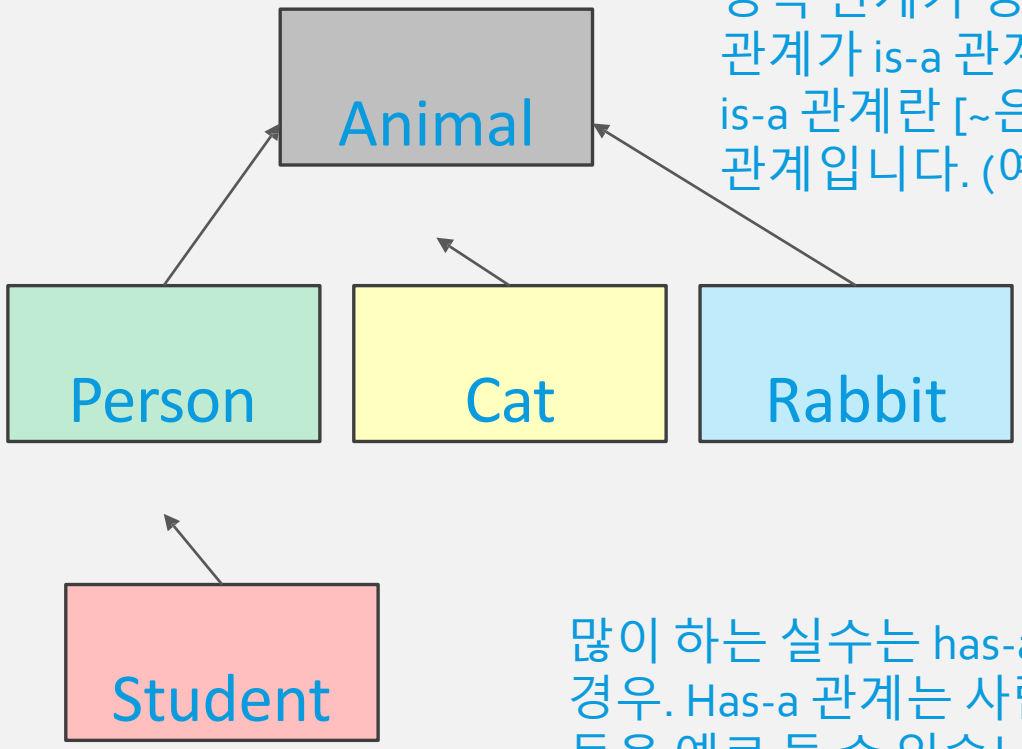
- **define** the class
- define **data attributes** (WHAT IS the object)
- define **methods** (HOW TO use the object)

**using** the new object type in code

- create **instances** of the object type
- do **operations** with them

# HIERARCHIES AND INHERITANCE (상속)

- **parent class** (superclass)
- **child class** (subclass)
  - **inherits** all data and behaviors of parent class
  - **add** more **info**
  - **add** more **behavior**
  - **override** behavior




상속 관계가 성립하려면 두 클래스 간의 관계가 is-a 관계여야 합니다.
is-a 관계란 [~은 ~이다] 라고 부를 수 있는 관계입니다. (예를들면 학생은 사람이다)

많이 하는 실수는 has-a 관계인 클래스를 상속하는 경우. Has-a 관계는 사람-팔, 자동차-엔진, 새-부리 등을 예로 들 수 있습니다.
하나의 클래스가 다른 클래스의 일부로 속할 때 has-a 관계가 성립됩니다.

# POLYMORPHISM (다형성) AND INHERITANCE

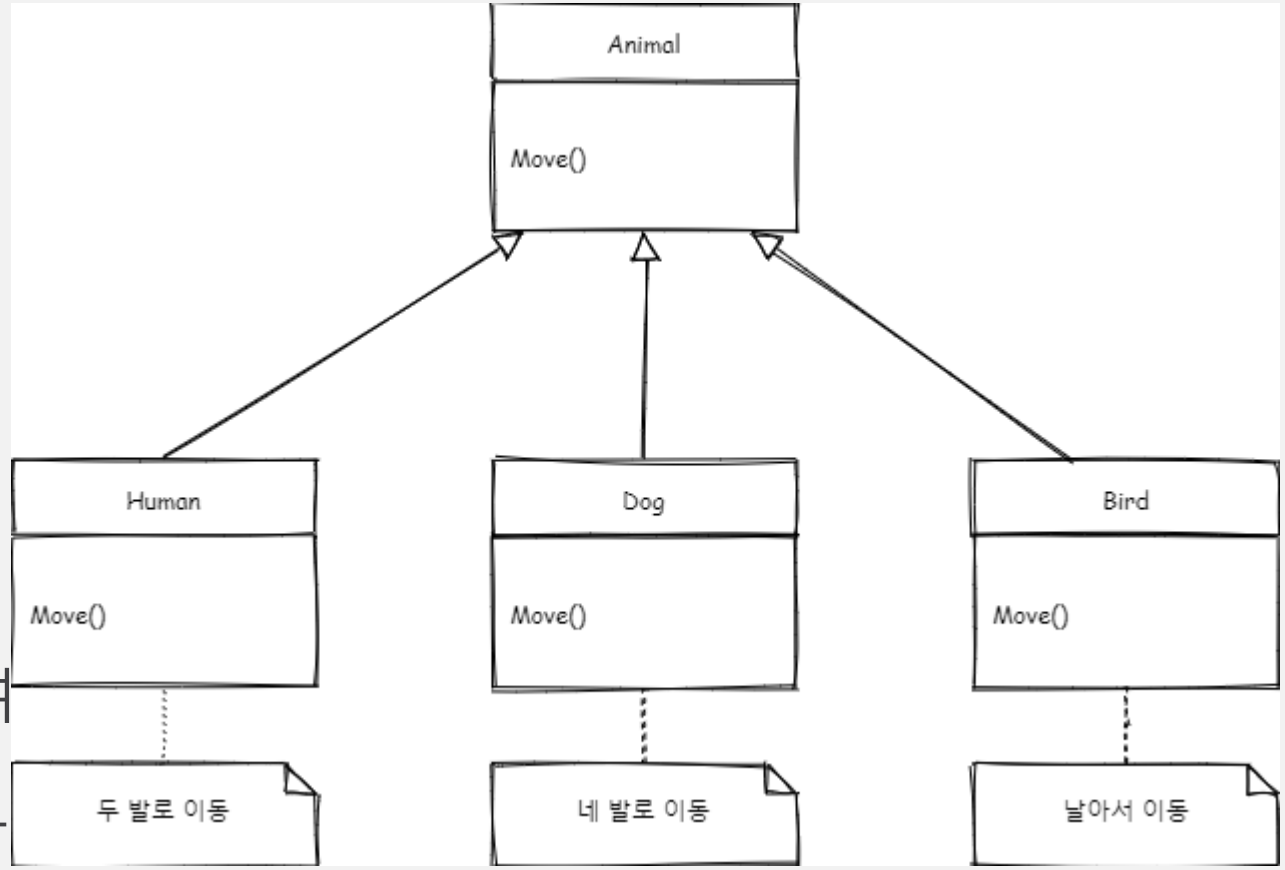


- Battery 라도 Rocket, Duracell, Energizer 등의 약간씩 다른 제품들이 있다. 이 들을 Battery 라는 super class 를 이용하고 조금씩 변형해서 만드는 것을 다형성이라 한다.
- 다형성의 효과로 각 개체를 부품화 할 수 있다.

- Overriding – superclass 를 상속받은 subclass에서 superclass 의 method 를 다시 정의해서 사용하도록 한다.
- Overloading – 하나의 class에서 같은 이름의 method 를 여러 개 가지는 것을 허용하는 것. 이 때 인자들의 type 이나 개수로 구별이 된다.

# 다형성(POLYMORPHISM)

- 예를들어 이동(move)라는 행위는 여러 가지 형태를 가질 수 있습니다.
- "현재 위치에서 특정 위치로 옮겨간다" 라는 목적은 같지만 실제로 이동하는 대상에 따라서 이동하는 방식이 달라질 수 있다.
- 직립보행을 하는 사람은 두 발로 걷거나 뛸 수 있으며 개나 고양이 같은 동물들은 네 발로 이동을 하고 새나 날개가 있는 곤충들은 날아서 이동 할 수도 있습니다.
- 이처럼 이동(move)이라는 행위에 대해 여러 가지 형태가 존재하기 때문에 전통적인 프로그래밍 방법으로는 if 구문이 반복적으로 사용되는 복잡한 코드가 생겨날 가능성이 높습니다.



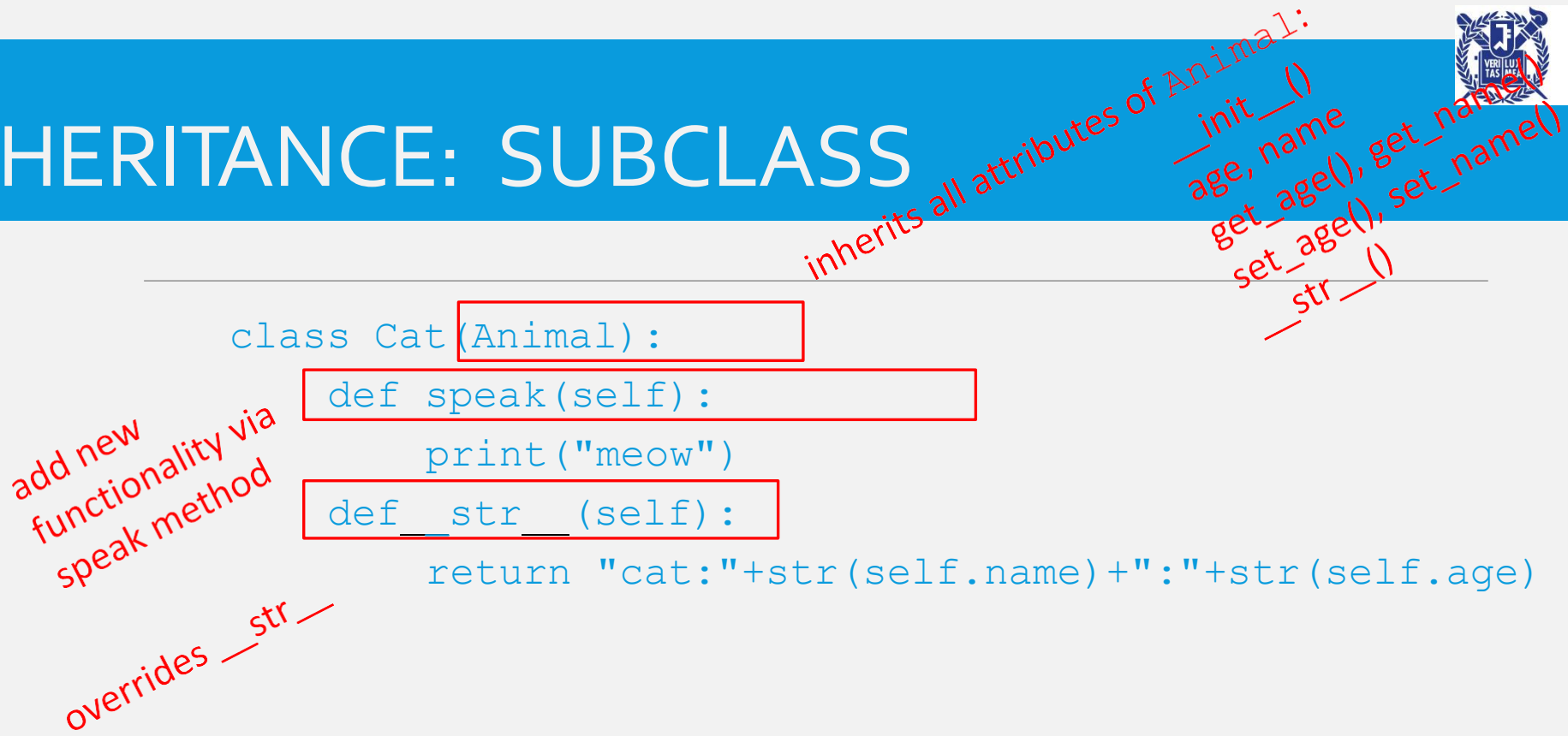

출처: https://gracefulprograming.tistory.com/130 [Peter의 우아한 프로그래밍]

# INHERITANCE: PARENT CLASS

```
class Animal(object):
    def __init__(self, age):
        self.age = age
        self.name = None
    def get_age(self):
        return self.age
    def get_name(self):
        return self.name
    def set_age(self, newage):
        self.age = newage
    def set_name(self, newname=""):
        self.name = newname
    def __str__(self):
        return "animal:"+str(self.name)+":"+str(self.age)
```

- everything is an object
- class `object` implements basic operations in Python, like binding variables, etc

# INHERITANCE: SUBCLASS




- add new functionality with `speak()`
  - instance of type `Cat` can be called with new methods
  - instance of type `Animal` throws error if called with `Cat`'s new method
- `__init__` is not missing, uses the `Animal` version

# WHICH METHOD TO USE?

- subclass can have **methods with the same name** as superclass
- for an instance of a class, look for a method name in **current class definition**
- if not found, look for method name **up the hierarchy** (in parent, then grandparent, and so on)
- Use the first method up the hierarchy that you found with that method name

# INHERITANCE: PARENT CLASS

```
class Animal(object):
    def __init__(self, age):
        self.age = age
        self.name = None
    def get_age(self):
        return self.age
    def get_name(self):
        return self.name
    def set_age(self, newage):
        self.age = newage
    def set_name(self, newname=""):
        self.name = newname
    def __str__(self):
        return "animal:"+str(self.name)+":"+str(self.age)
```

- everything is an object
- class `object` implements basic operations in Python, like binding variables, etc

```
class Person(Animal):
    def __init__(self, name, age):
        Animal.__init__(self, age)
        self.set_name(name)
        self.friends = []
    def get_friends(self):
        return self.friends
    def add_friend(self, fname):
        if fname not in self.friends:
            self.friends.append(fname)
    def speak(self):
        print("hello")
    def age_diff(self, other):
        diff = self.age - other.age
        print(abs(diff), "year difference")
    def __str__(self):
        return "person:"+str(self.name)+":"+str(self.age)
```

parent class is Animal

call Animal constructor

call Animal's method

add a new data attribute

new methods

override Animal's __str__ method

```
import random

class Student(Person):
    def __init__(self, name, age, major=None):
        Person.__init__(self, name, age)
        self.major = major
    def change_major(self, major):
        self.major = major
    def speak(self):
        r = random.random()
        if r < 0.25:
            print("i have homework")
        elif 0.25 <= r < 0.5:
            print("i need sleep")
        elif 0.5 <= r < 0.75:
            print("i should eat")
        else:
            print("i am watching tv")
    def __str__(self):
        return "student:"+str(self.name)+":"+str(self.age)+":"+str(self.major)
```

bring in methods from `random` class

inherits `Person` and `Animal` attributes

adds new data

- I looked up how to use the `random` class in the python docs
- `random()` method gives back float in [0, 1)

# CLASS VARIABLES

- **class variables** and their values are shared between all instances of a class

```
class Rabbit(Animal):
    tag = 1
    def __init__(self, age, parent1=None, parent2=None):
        Animal.__init__(self, age)
        self.parent1 = parent1
        self.parent2 = parent2
        self.rid = Rabbit.tag
        Rabbit.tag += 1
```

parent class

class variable

instance variable

access class variable

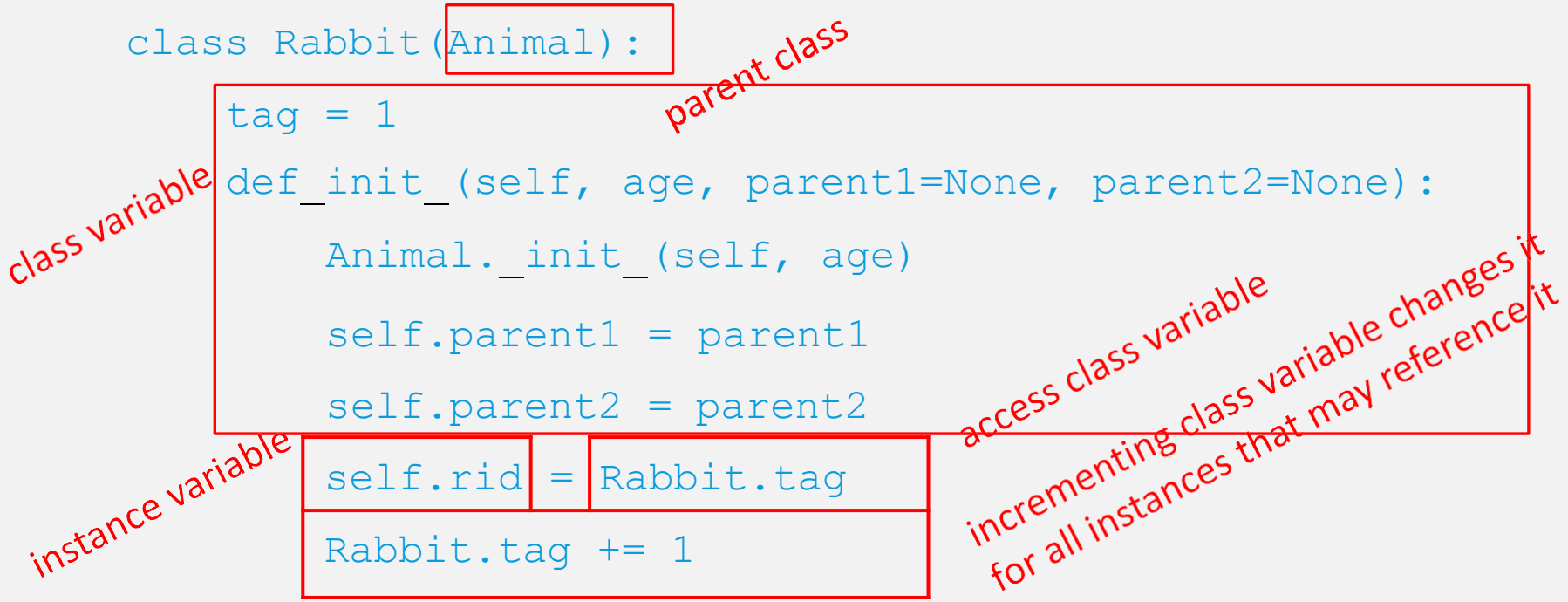
incrementing class variable changes it for all instances that may reference it

- `tag` used to give **unique id** to each new rabbit instance

# CLASS VARIABLES AND THE RABBIT SUBCLASS

- **class variables** and their values are shared between all instances of a class

```
class Rabbit(Animal):
    tag = 1
    def __init__(self, age, parent1=None, parent2=None):
        Animal.__init__(self, age)
        self.parent1 = parent1
        self.parent2 = parent2
        self.rid = Rabbit.tag
        Rabbit.tag += 1
```




- `tag` used to give **unique id** to each new rabbit instance

# SPECIAL METHODS TO COMPARE TWO RABBITS

- decide that two rabbits are equal if they have the **same two parents**

```
def __eq__(self, other):
    parents_same = self.parent1.rid == other.parent1.rid \
                   and self.parent2.rid == other.parent2.rid
    parents_opposite = self.parent2.rid == other.parent1.rid \
                       and self.parent1.rid == other.parent2.rid
    return parents_same or parents_opposite
```

booleans

- compare ids of parents since **ids are unique** (due to class var)
- note you can't compare objects directly
  - for ex. with `self.parent1 == other.parent1`
  - this calls the `__eq__` method over and over until call it on `None` and gives an `AttributeError` when it tries to do `None.parent1`

# OBJECT ORIENTED PROGRAMMING

- create your own **collections of data**
- **organize** information
- **division** of work
- access information in a **consistent** manner
- add **layers** of complexity
- like functions, classes are a mechanism for **decomposition** and **abstraction** in programming

# 결론 – 파이썬의 특징

서울대학교 SEOUL NATIONAL UNIVERSITY

- 파이썬은 인터프리터 언어로 한줄 한줄 바로바로 실행하며 디버깅이 용이하다.
- 변수가 동적 type을 가진다.  편리하지만 mutable, immutable 고려를 해야 한다.
- 기본으로 제공하는 data 구조가 있다 (list, tuple, dic), 특히 자료구조 작성에 좋다.
- 파이썬은 가독성과 신뢰성이 높은 객체 지향 프로그래밍 언어이다 (encapsulation - 기능, 데이타)
- 큰 장점은 생산성인데 검증된 모듈이 굉장히 많으며 표준 라이브러리도 많기 때문에 원하는 기능은 대부분 구현된 경우가 많다.
- 많은 응용을 지원하는 빨리 실행되는 모듈이 있다.  Glue 언어라 할 수 있다.