

# 으뜸 파이썬



## 4강 함수와 클래스

- 본 강의노트는 으뜸 파이썬(박동규, 강영민 著) 1판의 강의자료를 활용하여 교양수업에 맞게 편집되었습니다.

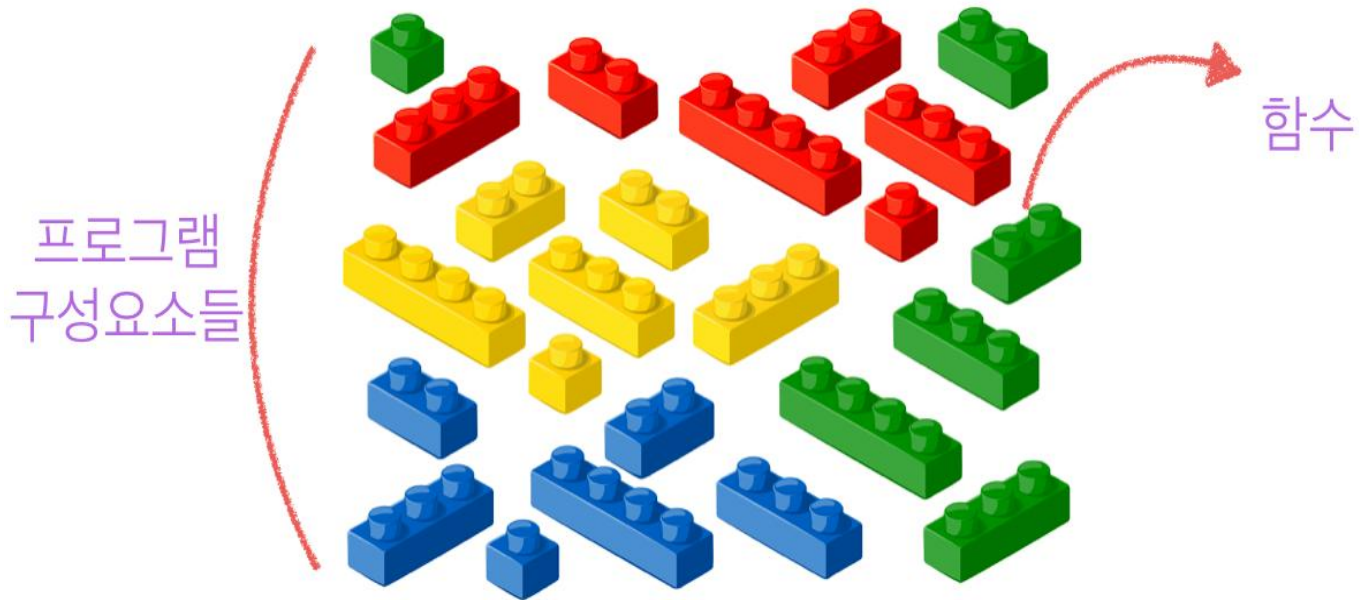
# 으뜸 파이썬



## 4-1강 함수 요약

- 본 강의노트는 으뜸 파이썬(박동규, 강영민 著) 1판의 강의자료를 활용하여 교양수업에 맞게 편집되었습니다.

# 함수의 역할



[그림 4-1] 레고 블록

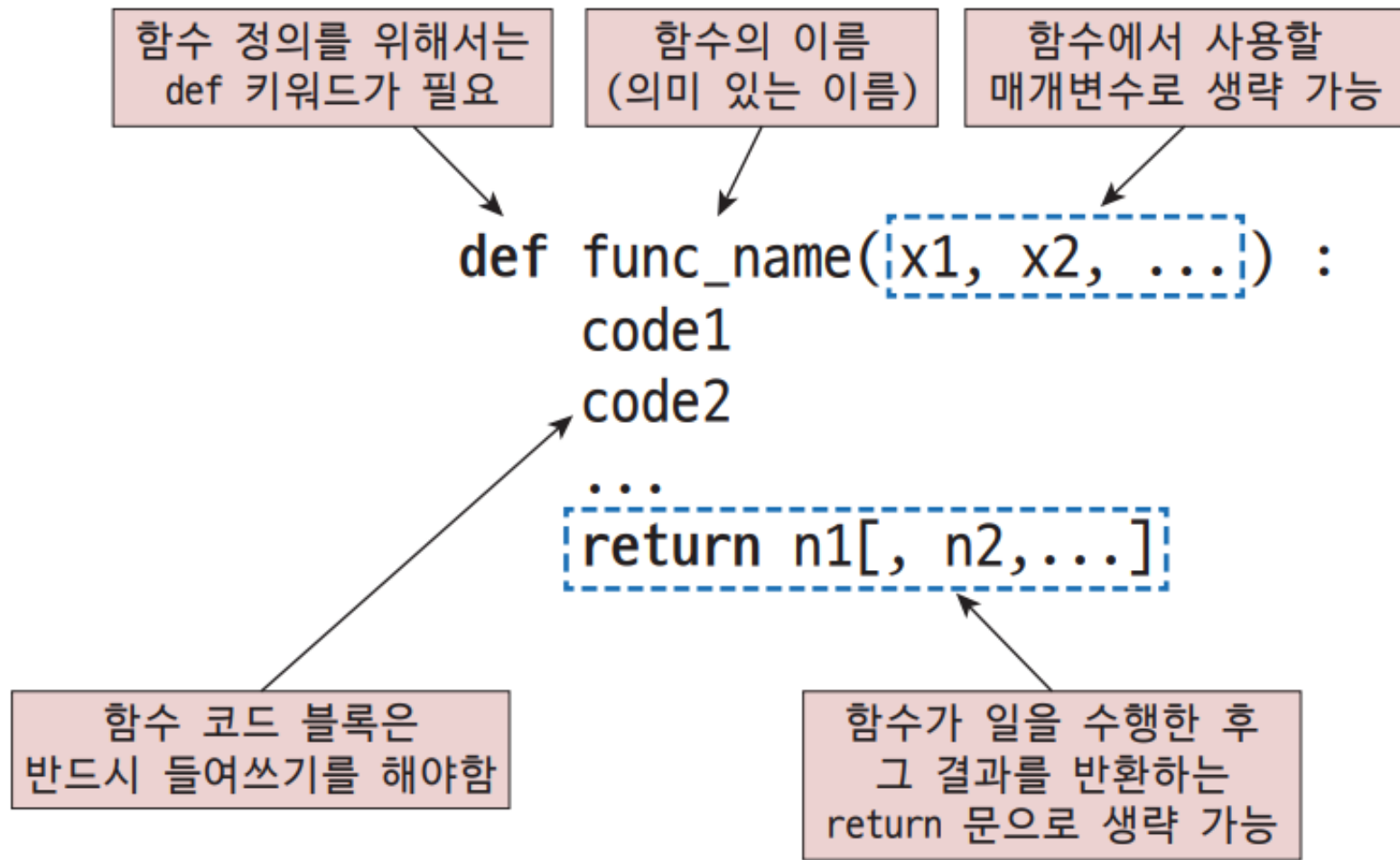


[그림 4-2] 레고 블록을 조립해서 만든 자동차

(출처: bricklink.com)

여러분이 사용하는 프로그램도 많은 부품(함수나 클래스)으로 이루어져 있습니다

- 반복적으로 사용되는 코드 - 덩어리(혹은 블록`block`) 이라고 함
- 기능에 따라 미리 만들어진 블록은 필요할 때 호출`function call`함
- 파이썬에서 미리 만들어서 제공하는 함수는 인터프리터에 포함되어 배포되는데 이러한 함수를 내장함수`built-in function` 라고 함
  - 대표적으로 `print()`가 있음
- 사용자가 직접 필요한 함수를 만들 수 있음
- 이러한 함수를 사용자 정의 함수`user defined function`라고 함
- `def` 키워드 사용 : `define`의 약자
  - `def`를 이용한 함수 정의 방법을 배워볼 예정



[그림 4-3] 파이썬에서 함수를 정의하는 문법

- return문이 없는 간단한 코드로 함수를 정의하고 호출하기

코드 4-1 : 별표 출력을 위한 함수 정의와 호출

print\_star\_func.py

```
def print_star():                # 별표 출력을 위한 함수 정의
    print('*****')

print_star()                     # 별표 출력을 위한 함수 호출
```

실행결과

```
*****
```

#### 코드 4-2 : 별표 출력을 위한 함수 정의와 반복 호출

print\_star\_4.py

```
def print_star():
```

```
    print('*****')
```

```
print_star()  # 별표 출력함수 호출 1
```

```
print_star()  # 별표 출력함수 호출 2
```

```
print_star()  # 별표 출력함수 호출 3
```

```
print_star()  # 별표 출력함수 호출 4
```

#### 실행결과

```
*****
```

```
*****
```

```
*****
```

```
*****
```

- **print\_star()**라는 함수는 어떤 일을 하도록 정의된 명령어들의 집합(혹은 블록)이며 이 집합은 외부에서 호출할 때마다 수행되는 것을 확인해 볼 수 있다.

# 함수와 매개변수

전달받을 값 3, 4를 가지는 변수 m, n : 매개변수

```
def foo(m, n) :  
    code  
    ...  
    return n1[, n2,...]
```

foo(3, 4)

foo 라는 함수에 넘겨줄 값 3, 4 : 인자

[그림 4-4] 매개변수와 인자의 개념과 사용방법



## NOTE : 인자와 매개변수

- **매개변수**Parameter : 함수나 메소드 헤더부에 정의된 변수로 함수가 호출될 때 실제 값을 전달받는 변수이다.

예: def foo(m, n): 의 m과 n

- **전달인자**Argument : 함수나 메소드가 호출될 때 전달되는 실제 값을 말하며, 간단하게 인자라고도 한다.

예: foo(3, 4)의 3과 4



#### 코드 4-5 : 매개변수를 가진 별표 출력 함수와 인자를 이용한 호출

print\_star\_param.py

# 별표 출력을 매개변수 n번만큼 반복하는 프로그램

```
def print_star(n):
```

```
    for _ in range(n):
```

```
        print('*****')
```

print\_star(4) # 별표 출력을 위해 4라는 인자 값을 준다.

#### 실행결과

```
*****
```

```
*****
```

```
*****
```

```
*****
```

# 매개변수를 활용한 2차 방정식의 근 구하기

$$ax^2 + bx + c = 0$$

[수식 4-1] x에 대한 2차 방정식

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

[수식 4-2] 2차 방정식의 근의 공식

#### 코드 4-8 : 2차 방정식의 근을 구하는 기능

root\_ex1.py

```
a = 1
```

```
b = 2
```

```
c = -8
```

```
# ( a * x^2 ) + ( b * x ) + c = 0
```

```
r1 = (-b + (b ** 2 - 4 * a * c) ** 0.5) / (2 * a)
```

```
r2 = (-b - (b ** 2 - 4 * a * c) ** 0.5) / (2 * a)
```

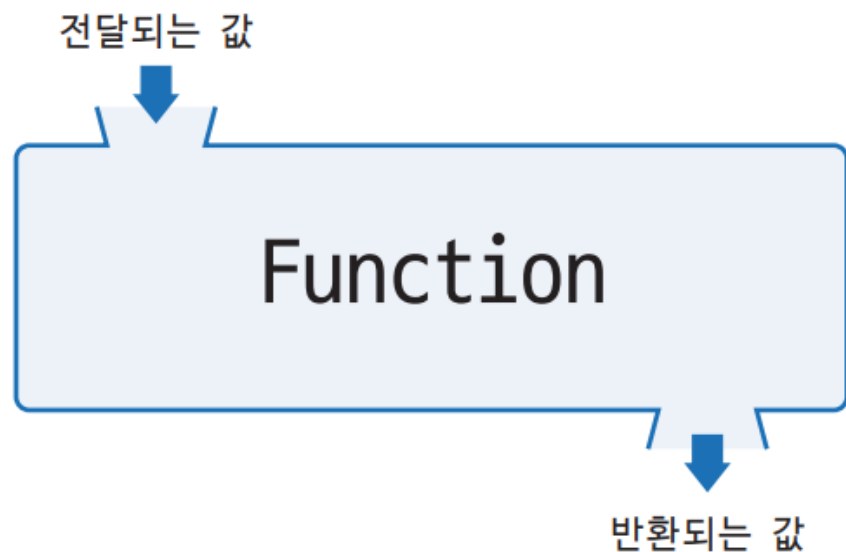
```
print('해는', r1, '또는', r2)
```

#### 실행결과

해는 2.0 또는 -4.0

- $a(a \neq 0)$ ,  $b$ ,  $c$ 를 해당하는 값을 방정식에 맞게 입력
- $a$ ,  $b$ ,  $c$ 에 해당하는 해를 변수  $r1$ ,  $r2$ 에 저장하여 출력

# 반환문 return



[그림 4-5] 값의 전달과 반환 : 함수는 값을 전달받아 처리하고 결과를 반환할 수 있다

```
def func_name([x1, x2, ...]) :  
    code1  
    code2  
    ...  
    return n1[, n2,...]
```

함수가 일을 수행한 후 그 결과를 반환하는 기능

[그림 4-6] 함수의 구성과 반환문 return

# 반환문 return

- 일반적으로 함수 내부는 블랙박스black box라고 가정
- 함수의 내부는 특정한 코드를 가지고 있으며 주어진 일을 수행하고 결과를 반환할 수 있음
- return 키워드를 사용하여 하나 이상의 값을 반환해 줄 수 있음

# 재귀함수

- 재귀함수 **recursion**란 함수 내부에서 자기 자신을 호출하는 함수를 말함
- 절차적 기법으로 해결하기 어려운 문제를 직관적이고 간단하게 해결 가능

- 함수 factorial()은  $n! = n * (n-1)!$  이라는 정의에 맞게 다음과 같이 다시 정의가 가능함

코드 4-29 : 재귀함수를 이용하여 정의한 팩토리얼

factorial\_recursion.py

```
def factorial(n):    # n!의 재귀적 구현
    if n <= 1 :      # 종료조건이 반드시 필요하다
        return 1
    else :
        return n * factorial(n-1)    # n * (n-1)! 정의에 따른 구현

n = 5
print('{}! = {}'.format(n, factorial(n)))
```

**실행결과**

5! = 120

# 으뜸 파이썬



## 4-2강 클래스 요약

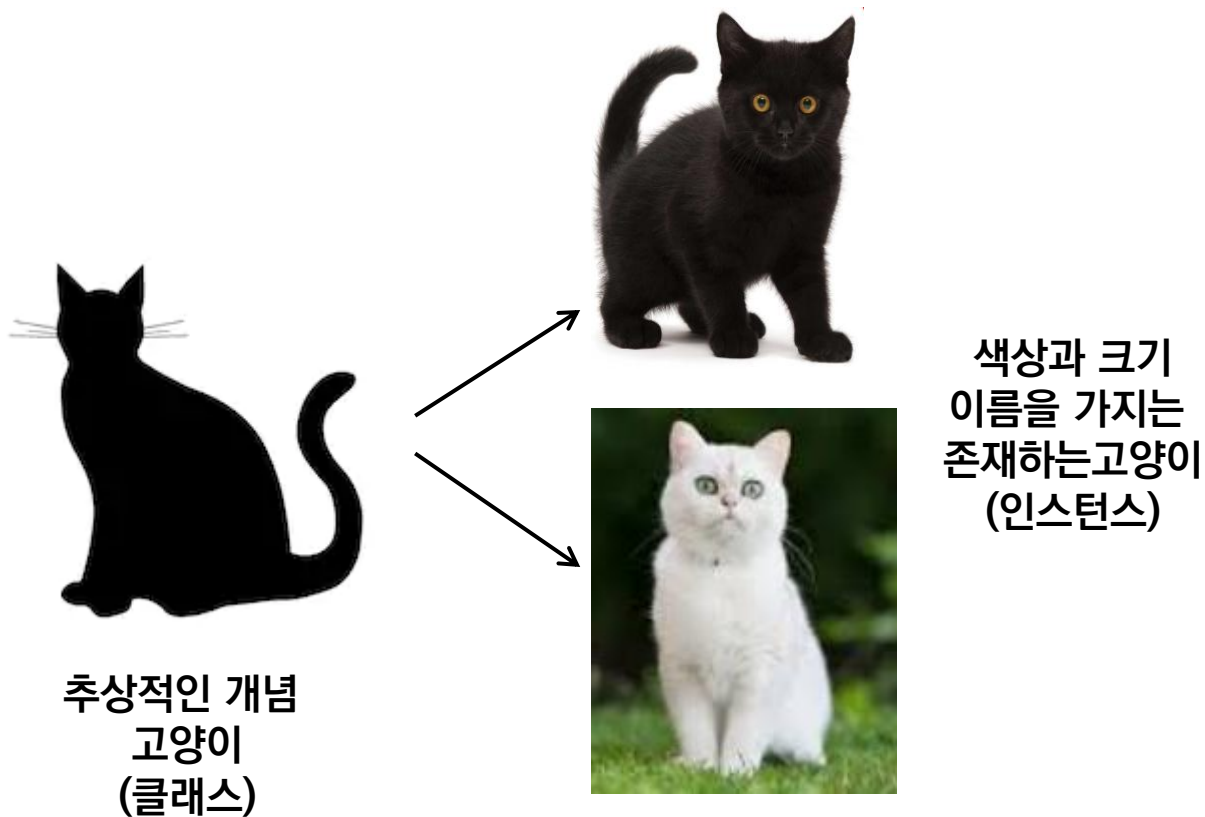
- 본 강의노트는 으뜸 파이썬(박동규, 강영민 著) 1판의 강의자료를 활용하여 교양수업에 맞게 편집되었습니다.



# 객체 지향 프로그래밍과 절차적 프로그래밍

- 객체 지향 프로그래밍 **OOP:object oriented programming**
  - 프로그램을 짤 때, 프로그램을 실제 세상에 가깝게 모델링 하는 기법
  - 컴퓨터가 수행하는 작업을 **객체들 사이의 상호작용**으로 표현
  - 클래스 **class**나 객체 **object** 들의 집합으로 소프트웨어를 개발하자는 개념
  - Java, Python, C++, C#, Swift 등 현재 사용중인 많은 프로그래밍 언어에서 채택
- 절차적 프로그래밍 언어 **procedural programming language**
  - 함수나 모듈을 만들어두고 이것들을 문제해결 순서에 맞게 호출하여 수행하는 방식
  - C, Fortran, Basic 등의 고전적인 프로그래밍 언어에서 사용함
  - 그래픽 사용자 인터페이스 **graphic user interface** 시스템과 같이 다양한 그래픽 요소들이 있을 경우 효과적으로 문제해결을 하기 힘들다.

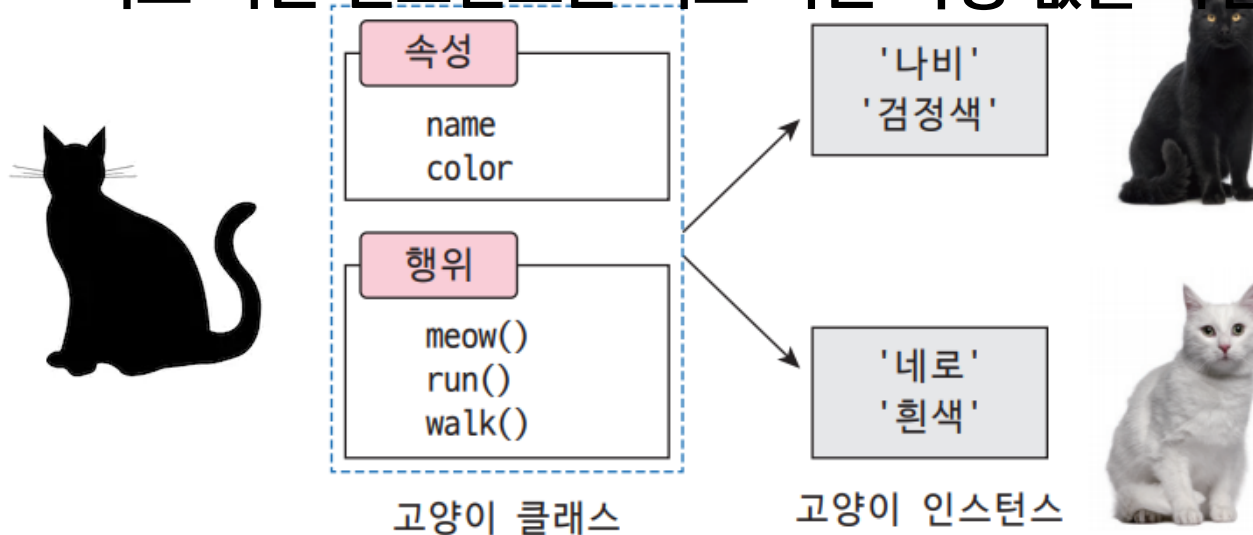
# 클래스와 객체, 인스턴스



- 클래스 **class**
  - 프로그램 상에서 사용되는 속성과 행위를 모아놓은 집합체
  - 객체의 설계도 혹은 템플릿(형틀 **template**), 청사진 **blueprint**
- 인스턴스 **instance**
  - 클래스로부터 만들어지는 각각의 개별적인 객체
  - 서로 다른 인스턴스는 서로 다른 속성 값을 가질 수 있음.

# 클래스와 객체, 인스턴스

- 클래스 **class**
  - 프로그램 상에서 사용되는 속성과 행위를 모아놓은 집합체
  - 객체의 설계도 혹은 템플릿(틀) **template**, 청사진 **blueprint**
- 인스턴스 **instance**
  - 클래스로부터 만들어지는 각각의 개별적인 객체
  - 서로 다른 인스턴스는 서로 다른 속성 값을 가질 수 있음.



[그림 9-5] 고양이 클래스와 인스턴스 개념도

## 9.4 클래스 정의와 인스턴스

- 고양이 클래스는 아주 추상적인 개념



[그림 9-6] 고양이 클래스와 인스턴스의 관계

- 클래스의 정의 방법
  - class라는 키워드를 써 준 후 class의 이름을 써 준다. 그 후 필요한 속성과 메소드를 파이썬 문법에 맞게 써 준다
- pass 문은 아무런 역할을 하지 않는 파이썬 명령문

코드 9-1 : Cat 클래스 정의와 인스턴스 생성 문법

cat\_pass.py

```
class Cat:                # Cat 클래스의 정의
    pass

nabi = Cat()              # Cat 인스턴스 생성
print(nabi)
```

```
class ClassName :
```

```
    <statement-1>
```

```
    ...
```

```
    <statement-n>
```

실행결과

⟨\_\_main\_\_.Cat object at 0x7f78399e0eb8⟩



## NOTE : 객체와 인스턴스

많은 책에서 객체와 인스턴스를 비슷한 개념으로 섞어서 사용한다. 두 용어는 매우 비슷해서 구분하지 않는 경우도 많지만 조금 더 엄밀하게 정의하자면 객체는 하나의 사물로 정의할 수 있으며 인스턴스는 클래스에 의해 만들어진 사물로 정의해서 사용한다.

예를 들어 이 책에서 정의한 Cat이라는 틀은 클래스이며 이 Cat이라는 클래스에 의해서 만들어진 사물 nabi는 "객체"이면서, 동시에 "Cat 클래스의 인스턴스"라고 이야기 할 수 있다.

정리하자면 다음과 같은 표현은 올바른 표현이다.

- 1) 파이썬의 모든 객체는 자료형을 가진다.
- 2) 파이썬의 인스턴스는 클래스로부터 만들어 진다.
- 3) 파이썬의 객체는 클래스로부터 만들어 진다.
- 4) 파이썬의 클래스는 객체이다.
- 5) nabi는 객체이다.
- 6) Cat 클래스의 인스턴스는 nabi이다.
- 7) 100은 int 형 객체이다.

한편 다음은 **올바르지 않거나 부자연스러운 표현**이다.

- 1) nabi는 인스턴스이다.(?)
- 2) Cat 클래스는 인스턴스이다.(X)

- 클래스 내부에서 정의되어 클래스나 클래스 인스턴스가 사용하는 함수를 메소드 **method** 혹은 멤버 함수 **member function**라 한다.
- **meow()** 메소드의 매개변수인 **self**는 자기 자신을 참조하는 변수이며 메소드의 첫 번째 매개변수로 반드시 들어가야 한다

코드 9-2 : Cat 클래스 정의와 meow() 메소드

cat\_class.py

```
class Cat:  
    def meow(self):  
        print('야옹 야옹~~~')
```

Cat 클래스내의 함수로 메소드라고 함

```
nabi = Cat()  
nabi.meow()
```

Cat()을 통해 Cat 클래스의 객체를 생성함. 이제 nabi.meow()를 통해 메소드 호출이 가능함

실행결과

야옹 야옹~~~

- 인스턴스의 메소드 호출 문법

인스턴스이름.메소드([인자])

- nabi라는 인스턴스는 Cat이라는 클래스가 가진 meow()라는 메소드를 사용할 수 있음

코드 9-3 : Cat 클래스 정의와 여러 개의 객체 생성

many\_cats.py

```
class Cat:
```

```
    def meow(self):
```

```
        print('야옹 야옹~~~')
```

```
nabi = Cat()
```

```
nabi.meow()
```

```
nero = Cat()
```

```
nero.meow()
```

```
mimi = Cat()
```

```
mimi.meow()
```

nabi 객체가 meow()실행

nero 객체가 meow()실행

mini 객체가 meow()실행

실행결과

야옹 야옹~~~

야옹 야옹~~~

야옹 야옹~~~



#### 코드 9-4 : 생성자를 가진 Cat 클래스의 정의와 인스턴스 생성

init\_cats.py

```
class Cat:
    # 생성자 혹은 초기화 메소드라 한다
    def __init__(self, name, color='흰색'):
        self.name = name                # name이라는 인스턴스 변수를 생성
        self.color = color              # color라는 인스턴스 변수를 생성
    # 고양이의 정보를 출력하는 메소드
    def meow(self):
        print('내이름은 {}, 색깔은 {}, 야옹 야옹~~'.format(self.name, self.color))

nabi = Cat('나비', '검정색')           # nabi 인스턴스 생성
nero = Cat('네로', '흰색')             # nero 인스턴스 생성
mimi = Cat('미미', '갈색')            # mimi 인스턴스 생성

nabi.meow()
nero.meow()
mimi.meow()
```

#### 실행결과

내이름은 나비, 색깔은 검정색, 야옹 야옹~~

내이름은 네로, 색깔은 흰색, 야옹 야옹~~

내이름은 미미, 색깔은 갈색, 야옹 야옹~~

- 두 번째 매개변수 name과 세 번째 매개변수 color는 인스턴스의 속성에 해당하는 이름과 색상을 할당하기 위한 변수

```
class Cat:  
    def __init__(self, name, color):  
        ....
```

```
nabi = Cat('나비', '검정색')
```

```
nero = Cat('네로', '흰색')
```

```
mimi = Cat('미미', '갈색')
```

[그림 9-7] self 매개변수의 의미와 인스턴스와의 관계

- 인스턴스 변수 `instance variable`, 멤버 변수 `member variable`, 혹은 필드 `field`
  - 각각의 인스턴스들이 개별적으로 가지는 속성을 저장하는 변수



[그림 9-8] `self.name`, `self.color`의 의미와 인스턴스와의 관계

# 캡슐화

- Cat 클래스에 나이를 의미하는 age라는 속성을 부여
  - nabi.age = -5를 넣게 된다면 코드상의 문법적인 문제는 없으나 고양이의 나이가 음수가 되는 논리적인 오류 발생
- 캡슐화encapsulation
  - 클래스의 속성을 외부에서 접근할 때 오류를 줄일 수 있음



**[그림 9-9]** 캡슐화의 개념도: 클래스의 메소드와 변수를 외부에서 함부로 조작하지 못하도록 감싸고 제한하는 기능

## 코드 9-6 : nabi.age에 직접 값을 할당하기

cat\_age\_change.py

```
class Cat:
    def __init__(self, name, age):
        self.name = name
        self.age = age

        # Cat 객체의 문자열 표현방식

    def __str__(self):
        return 'Cat(name='+self.name+', age='+str(self.age)+')'

nabi = Cat('나비', 3)          # nabi 인스턴스 생성
print(nabi)
nabi.age = 4
nabi.age = -5
print(nabi)
```

age가 음수가 되는  
비정상적 상황

### 실행결과

Cat(name=나비, age=3)  
Cat(name=나비, age=-5)

문법 오류는 아니지만 값이  
보호받지 못한 상태의 논리적  
문제

- 캡슐화(encapsulation)
  - 메소드와 변수를 외부에서 함부로 조작하는 것을 제한
  - 데이터를 보호
  - 우연히 값이 변경되는 것을 방지

## 코드 9-7 : set\_age() 메소드를 통해서 age 값을 할당하기

cat\_age\_with\_setter\_getter.py

```
class Cat:
    def __init__(self, name, age):
        self.__name = name
        self.__age = age
    # Cat 객체의 문자열 표현방식
    def __str__(self):
        return 'Cat(name='+self.__name+', age='+str(self.__age)+')'
    # self.__age를 외부에서 자유롭게 접근하는 것을 제한하고 음수가 되지 않도록 함
    def set_age(self, age):
        if age > 0:
            self.__age = age
    def get_age(self):
        return self.__age
```

```
nabi = Cat('나비', 3) # nabi 인스턴스 생성
print(nabi)
nabi.set_age(4)      # set_age() 메소드를 통해서 age에 접근
nabi.set_age(-5)     # set_age() 메소드를 통해서 age가 음수가 되지 않도록 함
print(nabi)
```

- if 조건식을 넣어서 age 값이 음수일때는 할당이 되지 않도록 해보기
- setXXX와 같이 시작하는 메소드를 세터setter라고 함
- 반대로 getXXX와 같이 시작하는 게터getter를 통해서 멤버 값을 읽어오는 것도 가능

### 실행결과

```
Cat(name=나비, age=3)
Cat(name=나비, age=4)
```

나이가 음수가 되는 비논리적  
상황을 해결하는 방법

- 캡슐화를 통해 보다 안전하게 멤버 내부의 변수를 보호

Cat 클래스의 속성 :  
외부에 공개하고 싶지 않은 속성

```
class Cat:
    def __init__(self, name, age):
        self.__name = name
        self.__age = age
        ...
    def set_age(self, age):
        if age > 0:
            self.__age = age
```

세터 : set\_age() 라는 메소드를  
통해서 속성에 접근



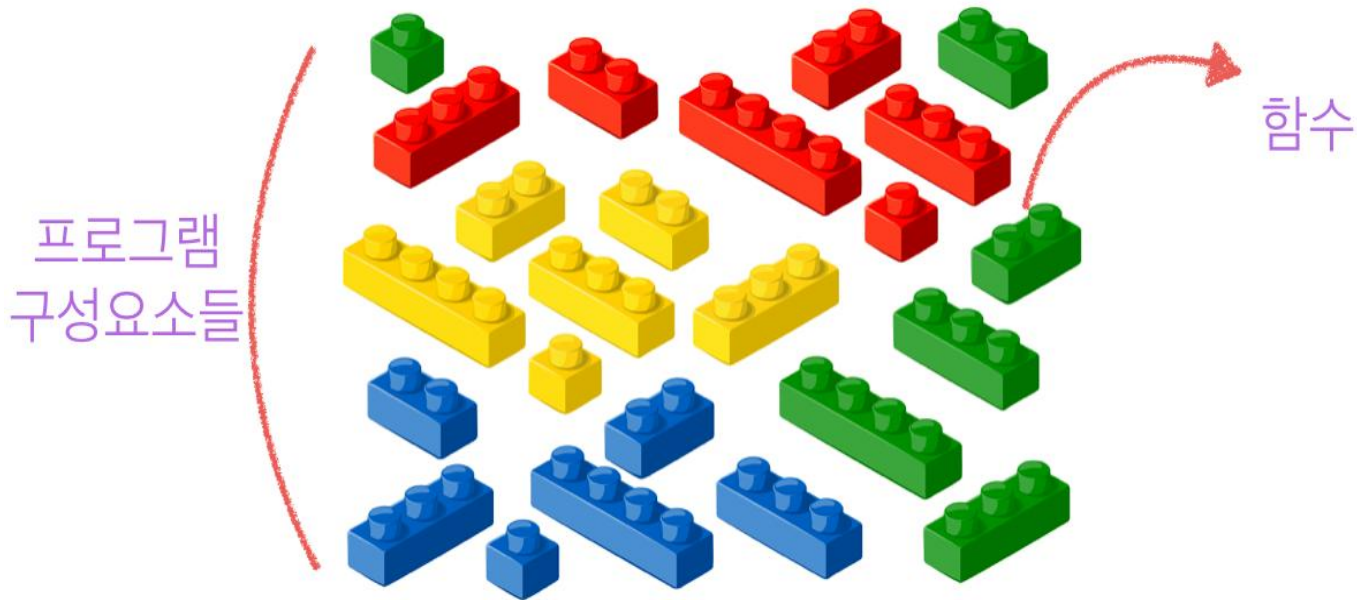
# 으뜸 파이썬



## 4-1강 함수 확장

- 본 강의노트는 으뜸 파이썬(박동규, 강영민 著) 1판의 강의자료를 활용하여 교양수업에 맞게 편집되었습니다.

# 함수의 역할



[그림 4-1] 레고 블록

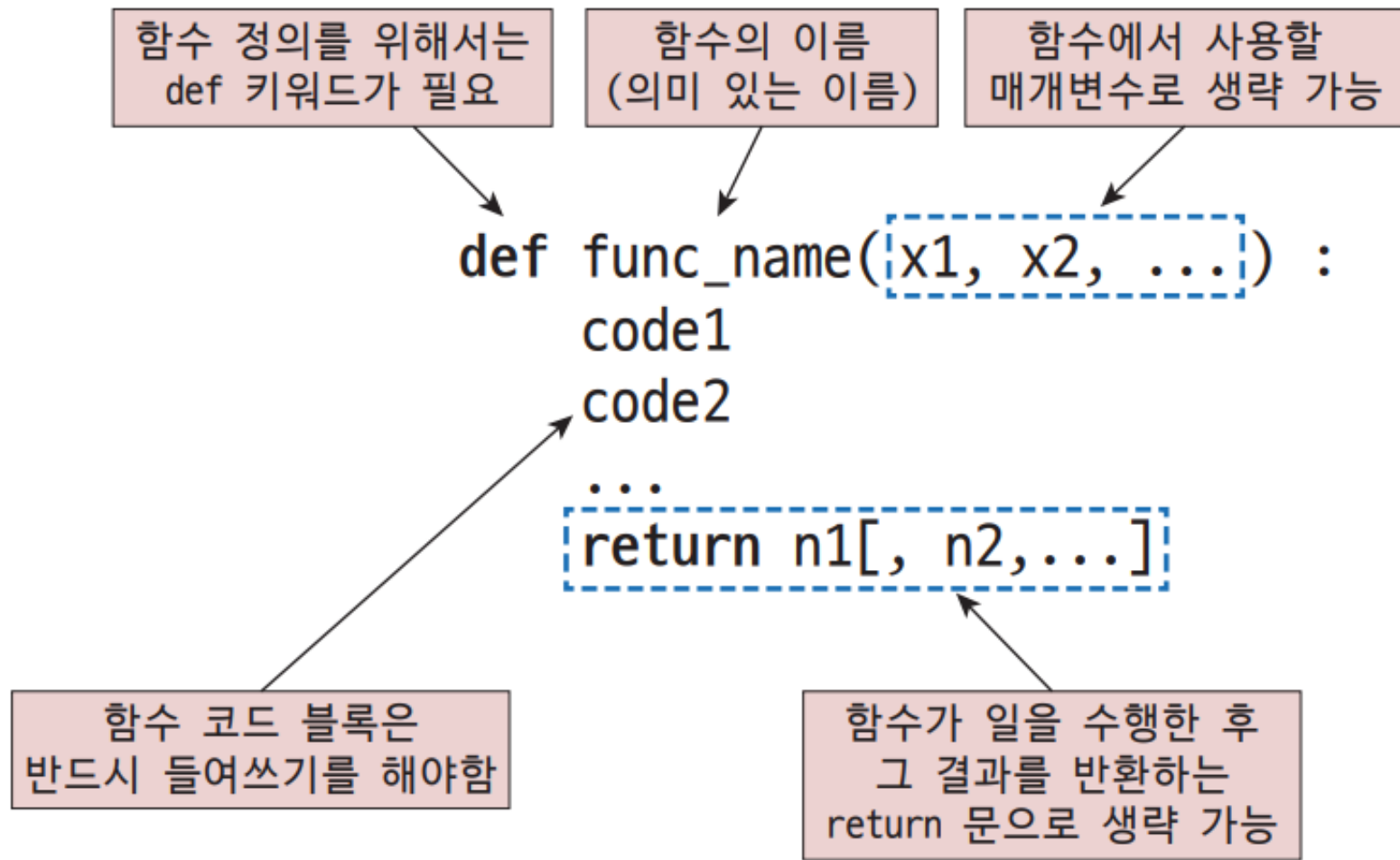


[그림 4-2] 레고 블록을 조립해서 만든 자동차

(출처: bricklink.com)

여러분이 사용하는 프로그램도 많은 부품(함수나 클래스)으로 이루어져 있습니다

- 반복적으로 사용되는 코드 - 덩어리(혹은 블록`block`) 이라고 함
- 기능에 따라 미리 만들어진 블록은 필요할 때 호출`function call`함
- 파이썬에서 미리 만들어서 제공하는 함수는 인터프리터에 포함되어 배포되는데 이러한 함수를 내장함수`built-in function` 라고 함
  - 대표적으로 `print()`가 있음
- 사용자가 직접 필요한 함수를 만들 수 있음
- 이러한 함수를 사용자 정의 함수`user defined function`라고 함
- `def` 키워드 사용 : `define`의 약자
  - `def`를 이용한 함수 정의 방법을 배워볼 예정



[그림 4-3] 파이썬에서 함수를 정의하는 문법

- **return문이 없는 간단한 코드로 함수를 정의하고 호출하기**

코드 4-1 : 별표 출력을 위한 함수 정의와 호출

print\_star\_func.py

```
def print_star():                # 별표 출력을 위한 함수 정의
    print('*****')

print_star()                     # 별표 출력을 위한 함수 호출
```

**실행결과**

```
*****
```

#### 코드 4-2 : 별표 출력을 위한 함수 정의와 반복 호출

print\_star\_4.py

```
def print_star():
```

```
    print('*****')
```

```
print_star()  # 별표 출력함수 호출 1
```

```
print_star()  # 별표 출력함수 호출 2
```

```
print_star()  # 별표 출력함수 호출 3
```

```
print_star()  # 별표 출력함수 호출 4
```

#### 실행결과

```
*****
```

```
*****
```

```
*****
```

```
*****
```

- **print\_star()**라는 함수는 어떤 일을 하도록 정의된 명령어들의 집합(혹은 블록)이며 이 집합은 외부에서 호출할 때마다 수행되는 것을 확인해 볼 수 있다.



## LAB 4-1 : 함수 정의와 호출

1. [코드 4-1]의 함수 호출문을 삭제하면 어떻게 되는가?
2. [코드 4-2]를 수정하여 6줄의 별표를 출력해 보시오. 이때 함수 호출을 6회 하시오.

```
*****
```

```
*****
```

```
*****
```

```
*****
```

```
*****
```

```
*****
```

#### 코드 4-3 : 3줄 별표 출력을 위한 함수 정의와 호출 방법

print\_star3.py

```
def print_star3():
```

```
    print('*****')
```

```
    print('*****')
```

```
    print('*****')
```

```
print_star3() # 3줄의 별표가 출력됨
```

```
print_star3() # 3줄의 별표가 출력됨
```

```
print_star3() # 3줄의 별표가 출력됨
```

#### 실행결과

```
*****
```

```
*****
```

```
*****
```

```
*****
```

```
*****
```

```
*****
```

```
*****
```

```
*****
```

```
*****
```





## LAB 4-2 : 함수 정의와 호출

1. [코드 4-3]을 수정하여 함수 호출 두 번으로 10줄의 별표를 출력해 보시오.

```
*****  
*****  
*****  
*****  
*****  
*****  
*****  
*****  
*****  
*****
```

#### 코드 4-4 : 별표 출력을 위한 함수 정의와 호출 방법의 수정

print\_star\_plus.py

```
def print_star(): # 별표 기호를 한 줄 출력함
    print('*****')

def print_plus(): # 더하기 기호를 한 줄 출력함
    print('+++++')

print_star()      # 별표 기호 출력
print_plus()      # 더하기 기호 출력
print_star()
print_plus()
```

#### 실행결과

```
*****
+++++
*****
+++++
```

- 한번 만들어진 함수는 다른 프로그램에서 재사용이 가능
- 프로그램 개발의 시간과 비용을 절약할 수 있다



## LAB 4-3 : 함수 정의와 호출

1. [코드 4-4]를 수정하여 해시마크(#)를 한 줄 출력하는 `print_hash()` 함수를 추가로 구현하십시오.
2. `print_star()`, `print_plus()`, `print_hash()` 함수를 모두 이용하여 다음과 같은 출력이 나타나도록 함수를 호출하십시오.

```
#####  
*****  
+++++  
+++++  
*****  
#####
```

# 함수와 매개변수

전달받을 값 3, 4를 가지는 변수 m, n : 매개변수

```
def foo(m, n) :  
    code  
    ...  
    return n1[, n2,...]
```

foo(3, 4)

foo 라는 함수에 넘겨줄 값 3, 4 : 인자

[그림 4-4] 매개변수와 인자의 개념과 사용방법



## NOTE : 인자와 매개변수

- **매개변수**Parameter : 함수나 메소드 헤더부에 정의된 변수로 함수가 호출될 때 실제 값을 전달받는 변수이다.

예: def foo(m, n): 의 m과 n

- **전달인자**Argument : 함수나 메소드가 호출될 때 전달되는 실제 값을 말하며, 간단하게 인자라고도 한다.

예: foo(3, 4)의 3과 4

#### 코드 4-5 : 매개변수를 가진 별표 출력 함수와 인자를 이용한 호출

print\_star\_param.py

# 별표 출력을 매개변수 n번만큼 반복하는 프로그램

```
def print_star(n):
```

```
    for _ in range(n):
```

```
        print('*****')
```

print\_star(4) # 별표 출력을 위해 4라는 인자 값을 준다.

#### 실행결과

\*\*\*\*\*

\*\*\*\*\*

\*\*\*\*\*

\*\*\*\*\*

# 매개변수를 활용한 2차 방정식의 근 구하기

$$ax^2 + bx + c = 0$$

[수식 4-1] x에 대한 2차 방정식

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

[수식 4-2] 2차 방정식의 근의 공식

#### 코드 4-8 : 2차 방정식의 근을 구하는 기능

root\_ex1.py

```
a = 1
```

```
b = 2
```

```
c = -8
```

```
# ( a * x^2 ) + ( b * x ) + c = 0
```

```
r1 = (-b + (b ** 2 - 4 * a * c) ** 0.5) / (2 * a)
```

```
r2 = (-b - (b ** 2 - 4 * a * c) ** 0.5) / (2 * a)
```

```
print('해는', r1, '또는', r2)
```

#### 실행결과

해는 2.0 또는 -4.0

- $a(a \neq 0)$ ,  $b$ ,  $c$ 를 해당하는 값을 방정식에 맞게 입력
- $a$ ,  $b$ ,  $c$ 에 해당하는 해를 변수  $r1$ ,  $r2$ 에 저장하여 출력

- 변수 a, b, c의 값을 2, -6, -8로 바꾼 방정식의 해를 구하고 싶을 때

코드 4-9 : 2차 방정식의 근을 구하는 기능의 반복 사용

root\_ex2.py

```
a = 1
b = 2
c = -8
# 근의 공식으로 해를 한 번 더 구한다.(반복되는 코드)
r1 = (-b + (b ** 2 - 4 * a * c) ** 0.5) / (2 * a)
r2 = (-b - (b ** 2 - 4 * a * c) ** 0.5) / (2 * a)
print('해는', r1, '또는', r2)

a = 2
b = -6
c = -8
# 근의 공식으로 해를 한 번 더 구한다.(반복되는 코드)
r1 = (-b + (b ** 2 - 4 * a * c) ** 0.5) / (2 * a)
r2 = (-b - (b ** 2 - 4 * a * c) ** 0.5) / (2 * a)
print('해는', r1, '또는', r2)
```

실행결과

해는 2.0 또는 -4.0

해는 4.0 또는 -1.0



- 문제점

- 변수  $a$ ,  $b$ ,  $c$ 에 원하는 계수를 입력하고, 다시  $r1$ ,  $r2$ 의 수식을 구해줘야 함
- 복사, 붙여 넣기를 한다 해도 코드가 중복되는 부분이 많고 불필요하게 긴 것을 한눈에 알 수 있다.

#### 코드 4-10 : 2차 방정식의 근을 구하는 기능을 함수로 만들기

root\_ex3.py

```
def print_root(a, b, c):  
    r1 = (-b + (b ** 2 - 4 * a * c) ** 0.5) / (2 * a)  
    r2 = (-b - (b ** 2 - 4 * a * c) ** 0.5) / (2 * a)  
    print('해는', r1, '또는', r2)
```

# 계수 값이 다른 2차 방정식의 해를 구함

```
print_root(1, 2, -8)
```

```
print_root(2, -6, -8)
```

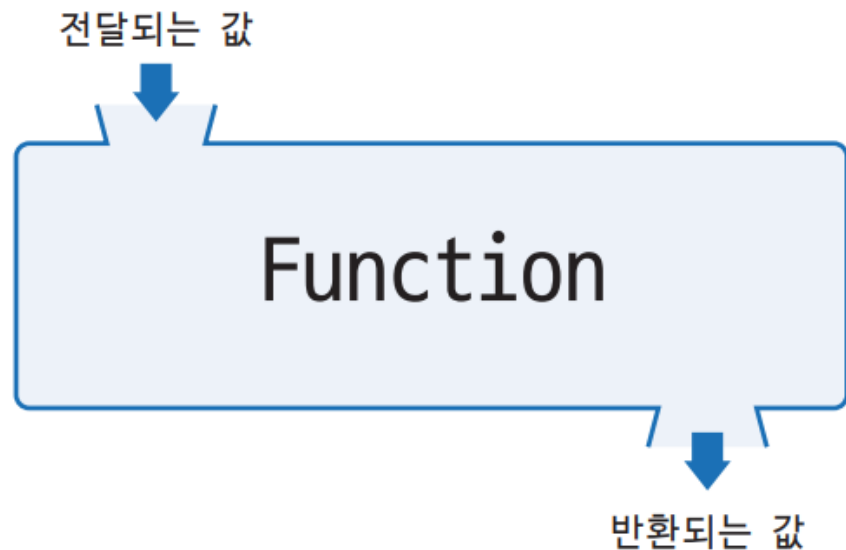
- 밖에서 넘겨준 계수 3개 a, b, c를 매개변수로 받고, 함수 몸체에 근의 공식 연산을 한 후, 결과 r1, r2를 출력하는 코드
- 코드가 훨씬 간결해지고, 사용하기 편리함

#### 실행결과

해는 2.0 또는 -4.0

해는 4.0 또는 -1.0

# 반환문 return



[그림 4-5] 값의 전달과 반환 : 함수는 값을 전달받아 처리하고 결과를 반환할 수 있다

```
def func_name([x1, x2, ...]) :  
    code1  
    code2  
    ...  
    return n1[, n2,...]
```

함수가 일을 수행한 후 그 결과를 반환하는 기능

[그림 4-6] 함수의 구성과 반환문 return

# 반환문 return

- 일반적으로 함수 내부는 블랙박스black box라고 가정
- 함수의 내부는 특정한 코드를 가지고 있으며 주어진 일을 수행하고 결과를 반환할 수 있음
- return 키워드를 사용하여 하나 이상의 값을 반환해 줄 수 있음

#### 코드 4-11 : 두 값의 합을 반환하는 get\_sum() 함수와 return 문의 사용

sum\_with\_return1.py

```
def get_sum(a, b):                # 두 수의 합을 반환하는 함수
    result = a + b
    return result                 # return 문을 사용하여 result를 반환

n1 = get_sum(10, 20)
print('10과 20의 합 =', n1)

n2 = get_sum(100, 200)
print('100과 200의 합 =', n2)
```

#### 실행결과

10과 20의 합 = 30

100과 200의 합 = 300

# 전역 변수

- 전역변수 `global variable`
  - 함수 바깥에서 선언되거나 전체 영역에서 사용 가능한 변수

코드 4-14 : 매개변수를 사용하지 않고 외부 변수를 사용하는 경우

sum\_func\_global1.py

```
def print_sum():  
    result = a + b  
    print('print_sum() 내부 :, a, '과', b, '의 합은', result, '입니다.')
```

```
a = 10    # 전역변수 a  
b = 20    # 전역변수 b  
  
print_sum()  
  
result = a + b  
  
print('print_sum() 외부 :, a, '과', b, '의 합은', result, '입니다.')
```

## 실행결과

print\_sum() 내부 : 10 과 20 의 합은 30 입니다.

print\_sum() 외부 : 10 과 20 의 합은 30 입니다.

코드 4-15 : 함수 외부에서 정의된 값을 함수 내부에서 변경하는 경우

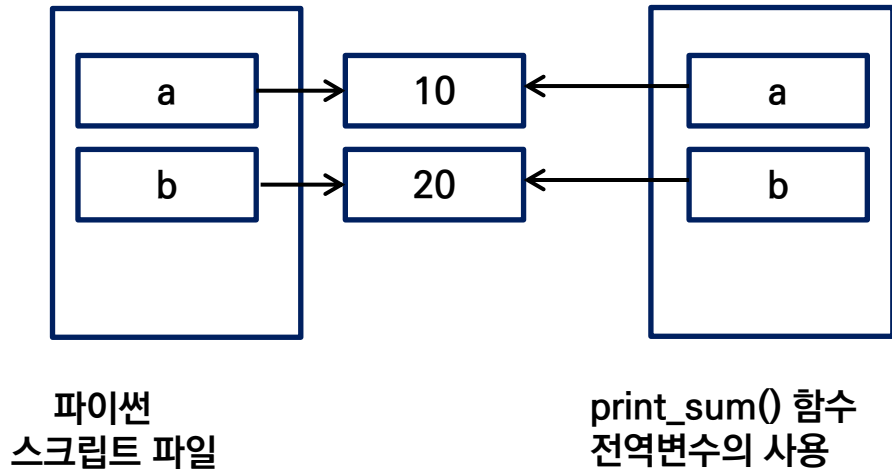
sum\_func\_global2.py

```
def print_sum():  
    a = 100  
    b = 200  
    result = a + b  
    print('print_sum() 내부 :, a, '과', b, '의 합은', result, '입니다.')
```

```
a = 10  
b = 20  
print_sum()
```

**실행결과**

print\_sum() 내부 : 100 과 200 의 합은 300 입니다.



[그림 4-8] 파이썬 스크립트 파일과 전역변수,  
그리고 이 전역변수를 사용하는  
print\_sum() 함수

코드 4-16 : 함수 내부에서 값을 변경하고, 그 값을 외부에서 확인하기

sum\_func\_global3.py

```
def print_sum():  
    a = 100  
    b = 200  
    result = a + b  
    print('print_sum() 내부 :, a, '과', b, '의 합은', result, '입니다.')  
  
a = 10  
b = 20  
print_sum()  
result = a + b  
print('print_sum() 외부 :, a, '과', b, '의 합은', result, '입니다.')
```

100, 200을 참조하는 새로운  
a, b 변수 생성

10, 20을 참조하는 a, b 변수 생성

### 실행결과

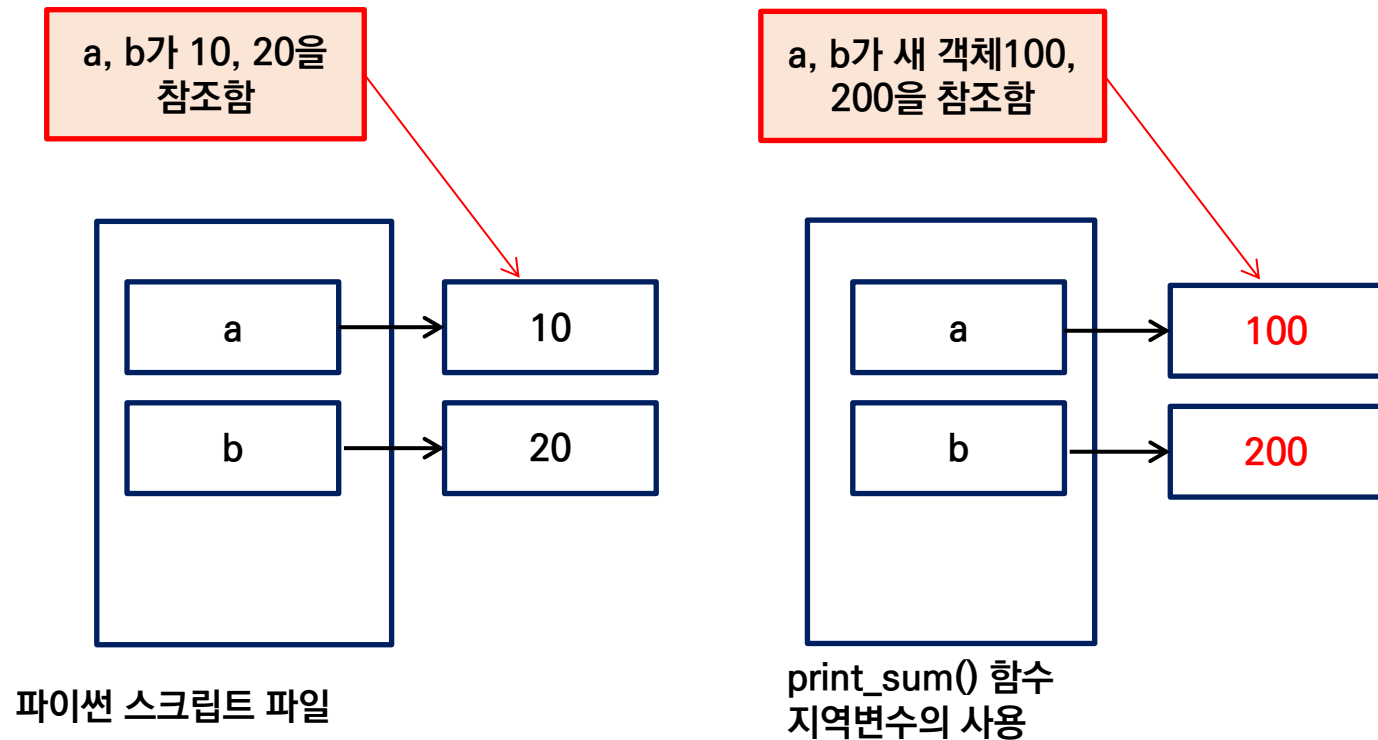
print\_sum() 내부 : 100 과 200 의 합은 300 입니다.

print\_sum() 외부 : 10 과 20 의 합은 30 입니다.

- print\_sum()을 수행한 다음 함수 외부에서 다시 한번 a와 b를 합하여 result에 대입하고 그 결과를 출력



- 할당 **assign**
  - `a = 100, b = 200`
- 지역 변수 **local variable**
- 참조 **reference**



[그림 4-9] 파이썬 스크립트 파일과 전역변수, 그리고 지역변수를 사용하는 print\_sum() 함수. 이 함수 내부의 a, b는 지역변수가 참조하는 객체가 아닌 별개의 객체를 참조함

## 코드 4-17 : global 키워드를 사용한 전역변수의 참조 방법

sum\_func\_global4.py

```
def print_sum():
```

```
    global a, b
```

# a, b는 함수외부에서 선언된 a, b를 사용한다.

```
    a = 100
```

```
    b = 200
```

```
    result = a + b
```

```
    print('print_sum() 내부 :, a, '과', b, '의 합은', result, '입니다.')
```

```
a = 10
```

```
b = 20
```

```
print_sum()
```

```
result = a + b
```

```
print('print_sum() 외부 :, a, '과', b, '의 합은', result, '입니다.')
```

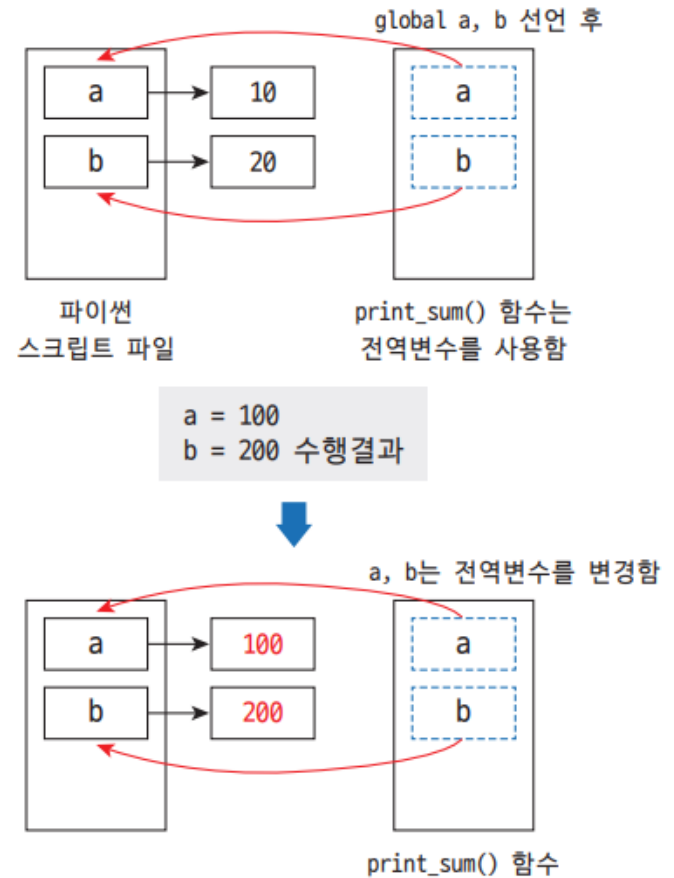
전역변수 a, b가 100, 200을 참조함

### 실행결과

print\_sum() 내부 : 100 과 200 의 합은 300 입니다.

print\_sum() 외부 : 100 과 200 의 합은 300 입니다.

global a = 100 # 문법 오류 발생



[그림 4-9] 명시적 global 선언을 통하여 함수 내에서 전역변수 a, b를 사용하는 과정



### 주의 : 전역변수와 전역상수

전역변수를 사용하는 것은 파이썬뿐만 아니라 모든 프로그래밍 언어에서 매우 나쁜 습관이다. 특히 코드의 길이가 길어질 경우 전역변수는 예외의 주요 원인이 된다.

그러나 **전역상수** `global constant`의 경우는 반드시 나쁘다고 볼 수 없다. 전역상수는 다음과 같이 `global`이라는 키워드로 선언하는데 함수의 외부에서 선언해서 모듈 전체에서 참조할 수 있다. 전역 상수값은 일반적으로 **대문자**를 사용한다.

아래의 코드를 살펴보면, `GLOBAL_VALUE`라는 이름의 변수에 1024라는 값을 할당한 후 `foo()` 함수에서 이 변수 값을 불러서 사용하기 위해 `global GLOBAL_VALUE`라는 이름으로 선언했다.

전역상수의 예 :

```
GLOBAL_VALUE = 1024
...
def foo():
    global GLOBAL_VALUE
    a = GLOBAL_VALUE * 100
```

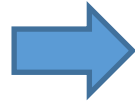
수학 연산을 위해 사용되는 `math` 모듈의 경우 원주율 `pi`와 오일러 상수 `e`를 프로그램 전체에서 참조하여 사용하는데, 이러한 상수 값의 경우는 예외적으로 소문자로 표기한다.

## 4.6 함수의 인자 전달 방식

코드 4-18 : 인자를 빠뜨린 호출

```
print_star_param_error.py
def print_star(n): # 인자를 필요로 함
    for _ in range(n):
        print('*****')

print_star() # 인자가 없으므로 에러 발생
```



코드 4-19 : 디폴트 값을 가지는 print\_star() 함수

```
print_default_param.py
def print_star(n = 1): # 매개변수 n은 디폴트 값 1을 가짐
    for _ in range(n):
        print('*****')

print_star() # 인자가 없더라도 에러 없이 수행됨
```

실행결과

```
*****
```

**TypeError:** print\_star() missing 1 required positional argument: 'n'

- 함수에 특정한 작업을 위임하기 위하여 정확한 인자를 넣어주는 것도 필요하지만 가끔씩은 위와 같은 에러를 예방하고, 좀 더 유연성 있는 작업을 위해서 디폴트 값을 사용하는 것이 편리할 때가 있음
- 이때 사용하는 것이 디폴트 매개변수 `default parameter`
- [코드 4-19]와 같이 매개변수에 `= 1`과 같이 디폴트 값을 할당

- 인자가 없이 호출해도 디폴트 값 1을 매개변수 n에 전달하므로 한 줄의 별표 라인이 정상적으로 출력됨

코드 4-19 : 디폴트 값을 가지는 print\_star() 함수

print\_default\_param.py

```
def print_star(n = 1):  # 매개변수 n은 디폴트 값 1을 가짐
    for _ in range(n):
        print('*****')
```

print\_star() # 인자가 없더라도 에러 없이 수행됨

실행결과

```
*****
```

# 키워드 인자 keyword argument

코드 4-22 : 2차 방정식의 근을 구하는 함수와 함수 호출문

root\_func.py

```
def get_root(a, b, c):  
    r1 = (-b + (b ** 2 - 4 * a * c) ** 0.5) / (2 * a)  
    r2 = (-b - (b ** 2 - 4 * a * c) ** 0.5) / (2 * a)  
    return r1, r2
```

# 함수 호출시 인자를 1, 2, -8 인자를 사용함.

# result1, result2를 이용해서 결과 값을 반환 받는다.

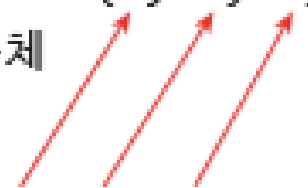
```
result1, result2 = get_root(1, 2, -8)  
print('해는', result1, '또는', result2)
```

실행결과

해는 2.0 또는 -4.0

- 함수를 호출할 때 인자의 값만을 전달하는 것이 아니라 그 인자의 이름을 함께 명시하여 전달하는 방식
- 파이썬의 기본 인자 전달 방식을 위치 인자 positional argument 방식이라고 함

```
def get_root(a, b, c):  
    함수 몸체  
  
get_root(1, 2, -8)  #함수 호출문
```



**[그림 4-10]** 위치 인자의 전달 방식 : 매개변수에 전달할 값을 전달할 때 a, b, c 순서에 따라 전달하므로 순서가 중요함

```
result1, result2 = get_root(-8, 2, 1) # 1, 2, -8을 인자로 줄 때와 그 결과가 다름
```

실행결과

해는 -0.25 또는 0.5



위치와 상관없이 키워드에 의해서 인자 값이 결정됨

```
result1, result2 = get_root(a = 1, b = 2, c = -8)
```

실행결과

해는 2.0 또는 -4.0

```
result1, result2 = get_root(a = 1, c = -8, b = 2)
```

위의 코드와 아래 코드는 그 결과가 동일하다,  
키워드 인자를 사용하면 인자의 위치는 중요하지 않다

```
result1, result2 = get_root(c = -8, b = 2, a = 1)
```

실행결과

해는 2.0 또는 -4.0

```
def get_root(a, b, c):  
    함수 몸체  
  
get_root(a=1, c=-8, b=2)
```

**[그림 4-11]** 키워드 인자의 전달 방식 : a, b, c의 키워드를 통해서 매개변수에 전달할 값을 명시해 주므로 순서는 중요하지 않음

```
result1, result2 = get_root(c = -8, b = 2, 1)
```

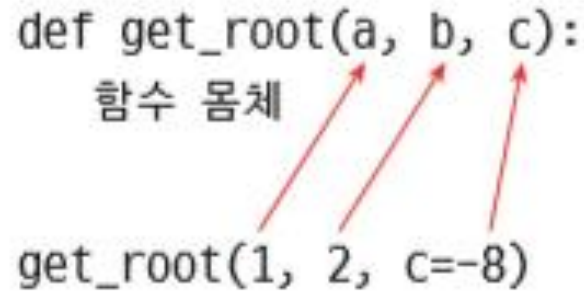
키워드 인자와 위치 인자를 섞어서 사용할 적에는 반드시 위치인자가 먼저 나타나야 한다(위의 경우는 오류)

**SyntaxError:** positional argument follows keyword argument

```
result1, result2 = get_root(1, 2, c = -8)
```

키워드 인자와 위치 인자를 섞어서 사용할 적에 위치인자를 먼저 적어주면 된다

```
def get_root(a, b, c):  
    함수 몸체  
  
get_root(1, 2, c=-8)
```



[그림 4-12] 위치 인자와 키워드 인자의 혼용 : 1, 2는 a, b에 전달되고 -8은 키워드를 통해 명시해준 c에 전달됨



#### NOTE : 위치 인자와 키워드 인자로 인자를 전달하기

파이썬의 함수에서는 인자를 전달할 때 위치 인자로 전달하는 방식과 키워드 인자로 전달하는 방식이 있다. 그리고 두 가지 방식을 혼합하는 방식도 있다. 그러나 두 가지 방식을 혼합하는 경우 **반드시** 위치 인자 뒤에 키워드 인자가 와야 한다.

```
result1, result2 = get_root(1, -8, b = 2)
```

**TypeError: get\_root() got multiple values for argument 'b'**

```
>>> def func(a, b, c) :
```

```
...     print(a, b, c)
```

```
...
```

```
>>> func(1, 2, 3)
```

```
1 2 3
```

```
>>> func(1, c=2, b=3)
```

```
1 3 2
```

```
>>> func(1, b=2, 3)
```

**SyntaxError: positional argument follows keyword argument**



## LAB 4-9 : 키워드 인자

1. 다음과 같이 성(last name)과 이름(first name), 존칭(honorifics)을 매개변수로 받아서 출력하는 함수 print\_name이 있다.

```
def print_name(honorifics, first_name, last_name):  
    ''' 키워드 인자를 이용한 출력용 프로그램 '''  
    print(honorifics, first_name, last_name)
```

a) 다음과 같은 함수 호출의 결과는 무엇인가?

```
print_name(first_name='Gildong', last_name='Hong', honorifics='Dr.')
```

b) 다음과 같은 함수 호출의 결과는 무엇인가?

```
print_name('Gildong', 'Hong', 'Dr.')
```

코드 4-24 : 인자를 하나 가지는 함수

arg\_greet1.py

```
def greet1(name):  
    print('안녕하세요', name, '씨')  
  
greet1('홍길동')
```

**실행결과**

안녕하세요 홍길동 씨

코드 4-25 : 인자를 2개 가지는 함수

arg\_greet2.py

```
def greet2(name1, name2):  
    print('안녕하세요', name1, '씨')  
    print('안녕하세요', name2, '씨')  
  
greet2('홍길동', '홍길순')
```

**실행결과**

안녕하세요 홍길동 씨  
안녕하세요 홍길순 씨

인자의 개수를 미리 알 수 없을 경우에는 어떻게 해야만 할까?

# 가변적인 인자전달

- 인자의 수가 정해지지 않은  
가변 인자 `arbitrary argument`

→ 별표(\*)를 매개변수의  
앞에 넣어 사용

- 가변적 인자는 튜플이나  
리스트와 비슷하게 `for - in`문에서 사용가능

코드 4-26 : 가변 인자를 가지는 함수의 정의와 호출

arg\_greet.py

```
def greet(*names):
```

```
    for name in names:
```

```
        print('안녕하세요', name, '씨')
```

```
greet('홍길동', '양만춘', '이순신') # 인자가 3개
```

```
greet('James', 'Thomas') # 인자가 2개
```

실행결과

안녕하세요 홍길동 씨

안녕하세요 양만춘 씨

안녕하세요 이순신 씨

안녕하세요 James 씨

안녕하세요 Thomas 씨

코드 4-27 : 가변 인자를 가지는 함수에서 len() 함수 활용

arg\_foo.py

```
def foo(*args):  
    print('인자의 개수:', len(args))  
    print('인자들 :', args)
```

```
foo(10, 20, 30)
```

#### 실행결과

인자의 개수: 3

인자들 : (10, 20, 30)

- len() 함수를 이용하여 다음과 같이 가변적으로 전달된 인자의 개수를 출력하는 것도 가능



- 숫자의 합을 구하는 프로그램
- `sum_num()` 함수에 전달될 인자의 개수를 미리 알 수 없는 경우, 가변인자를 받는 `*numbers`라는 매개변수를 사용하여 전체 인자를 튜플 형식으로 받을 수 있음

코드 4-28 : 가변 인자를 가지는 함수를 이용한 합계 구하기

arg\_sum\_nums.py

```
def sum_nums(*numbers):  
    result = 0  
    for n in numbers:  
        result += n  
    return result  
  
print(sum_nums(10, 20, 30))           # 10, 20, 30 인자들의 합을 출력  
print(sum_nums(10, 20, 30, 40, 50))  # 10, 20, 30, 40, 50 인자들의 합을 출력
```

실행결과

60

150

# 재귀함수

- 재귀함수 **recursion**란 함수 내부에서 자기 자신을 호출하는 함수를 말함
- 절차적 기법으로 해결하기 어려운 문제를 직관적이고 간단하게 해결 가능

- 함수 factorial()은  $n! = n * (n-1)!$  이라는 정의에 맞게 다음과 같이 다시 정의가 가능함

코드 4-29 : 재귀함수를 이용하여 정의한 팩토리얼

factorial\_recursion.py

```
def factorial(n):    # n!의 재귀적 구현
    if n <= 1 :      # 종료조건이 반드시 필요하다
        return 1
    else :
        return n * factorial(n-1)    # n * (n-1)! 정의에 따른 구현

n = 5
print('{}! = {}'.format(n, factorial(n)))
```

**실행결과**

5! = 120

# 으뜸 파이썬



## 4-2강 클래스와 객체 확장

- 본 강의노트는 으뜸 파이썬(박동규, 강영민 著) 1판의 강의자료를 활용하여 교양수업에 맞게 편집되었습니다.

# 객체 지향 프로그래밍과 객체

대화창 실습 : animals 리스트 객체와 다양한 메소드

```
>>> animals = ['lion', 'tiger', 'cat', 'dog']
```

```
>>> animals.sort()
```

```
>>> animals
```

```
['cat', 'dog', 'lion', 'tiger']
```

```
>>> animals.append('rabbit')
```

```
>>> animals
```

```
['cat', 'dog', 'lion', 'tiger', 'rabbit']
```

```
>>> animals.reverse()
```

```
>>> animals
```

```
['rabbit', 'tiger', 'lion', 'dog', 'cat']
```

- 파이썬의 리스트 **list**는 'lion', 'tiger', 'cat', 'dog' 등의 항목을 원소(**속성**)로 가질 수 있음.
- 또한 `sort()`, `append()`, `remove()`, `reverse()`, `pop()` 라는 함수(**메소드**)를 가지고 있음(**. 표기로 메소드 호출**)
- 객체 **object**
  - 컴퓨터 시스템에서 다양한 기능을 수행하도록 **속성**과 **메소드**를 가진 요소를 객체라고 함

- 편리한 메소드를 쉽게 이용할 수 있는 것이 객체 지향 프로그래밍 **objected oriented programming**의 큰 장점
- 또한 문자열 **str**이라는 클래스 자료형에는 문자와 관련된 다양한 처리기능이 미리 잘 정의되어 있기 때문에 객체 지향 프로그래밍에서 이렇게 많은 메소드를 사용할 수 있음

대화창 실습 : 리스트 객체의 pop() 메소드와 스트링 객체의 upper() 메소드

```
>>> s = animals.pop()
```

```
>>> s
```

```
'cat'
```

```
>>> s.upper()    # 문자열을 대문자로 치환하여 반환
```

```
'CAT'
```

```
>>> s.find('a')  # 문자열내의 특정 문자를 찾아서 인덱스를 반환
```

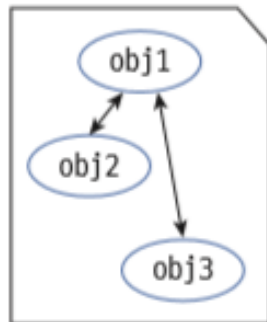
# 예: str 클래스의 다양한 메소드

- upper()
- lower()
- capitalize()
- startswith()
- strip()
- find()
- split()
- join()
- casefold()
- center()
- count()
- endswith()
- format()
- index()
- isalnum()
- isalpha()
- isdecimal()
- islower()
- ....

str 클래스 하나만 해도 이와 같은 많은 메소드가 제공되고 있다. 파이썬은 기본적으로 int, list, tuple, dic, date, time 등 많은 클래스가 있으며, 이들이 가진 메소드(기능)이 무궁무진하다.

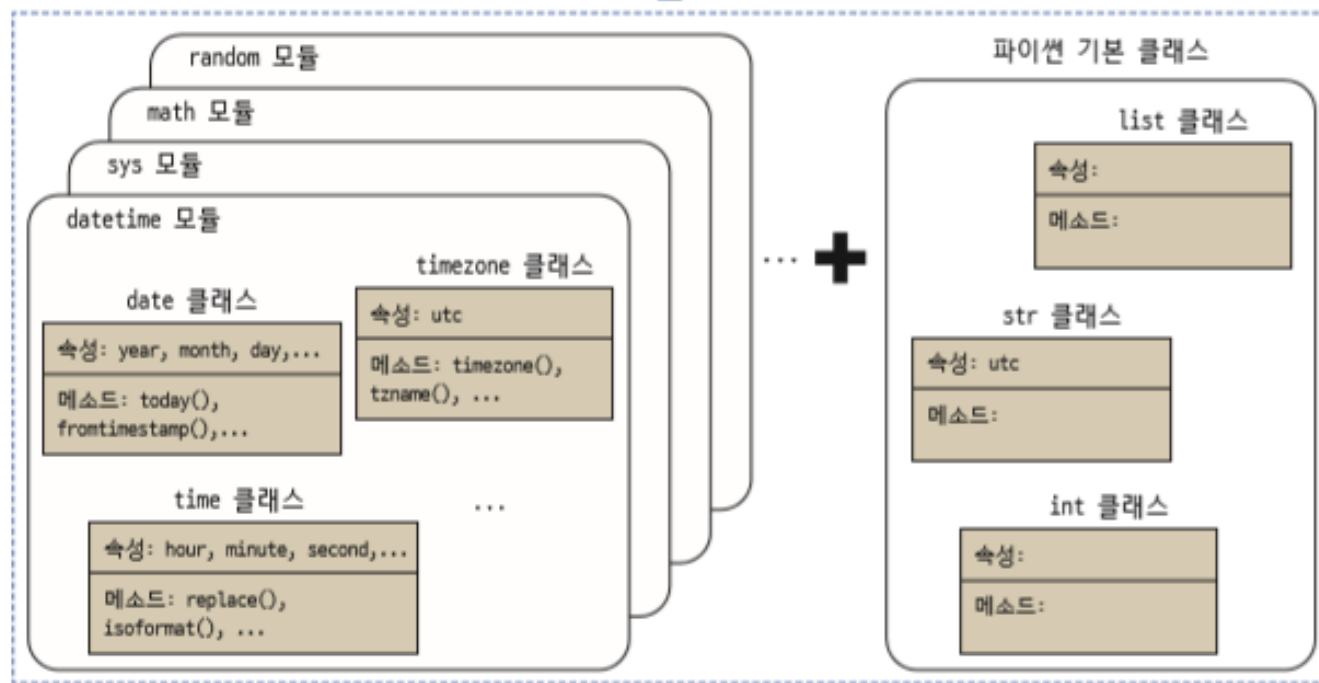
사용자가 코딩하는 부분  
prog.py 라 하자

객체간의  
상호작용



prog.py

수많은 클래스는 객체를 생성하기 위한 틀(타입)이다. 실제로 프로그램을 하기 위해서는 객체를 생성해야 하고 객체간의 상호작용이 필요하다



[그림 9-1] 파이썬이 제공하는 다양한 모듈과 클래스, 그리고 기본 클래스를 조합하여 프로그램을 개발하는 객체 지향 프로그래밍의 과정



- `type()`은 객체의 자료형을 알려주는 함수
- 객체들은 생성될 때 서로 다른 고유의 아이디 값을 가짐

대화창 실습 : `type()` 과 `id()` 함수

```
>>> animals = ['lion', 'tiger', 'cat', 'dog']
```

```
>>> type(animals)                # animals 객체의 자료형을 반환함
```

```
<class 'list'>
```

```
>>> id(animals)                  # animals 객체의 고유한 id를 반환함
```

```
24990662
```

```
>>> s = 'tiger'
```

```
>>> type(s)                      # s 객체의 자료형을 반환함
```

```
<class 'str'>
```

```
>>> id(s)                        # s 객체의 고유한 id를 반환함
```

```
83596600
```

## · 정수형의 type과 id 그리고 메소드

대화창 실습 : 정수형의 type과 id, 연산자와 메소드

객체지향 언어인 파이썬에서  $n + 100$ 과 같은 연산은  $n$ 이라는 객체의 `__add__(100)` 메소드 호출과 내부적으로는 동일하게 동작함

```
>>> n = 200
```

```
>>> type(n)
```

# n 객체의 자료형을 반환함

```
<class 'int'>
```

```
>>> id(n)
```

# n 객체의 고유한 id를 반환함

```
24990565
```

```
>>> n + 100
```

# n 객체와 정수 100의 합을 연산함

```
300
```

```
>>> n.__add__(100) # n 객체와 정수 100의 합을 연산함(위의 +와 동일한 일을 함)
```

```
300
```

```
>>> 200 + 100 # 200 과 100의 합을 구하는 연산자
```

```
300
```

```
>>> (200).__add__(100) # 위의 200 + 100 연산과 동일한 일을 함
```

```
300
```

## • int 클래스의 메소드

- `__add__()`
- `__sub__()`
- `__mult__()`
- `__truediv__()`
- `__mod__()`
- `__pow__()`
- `__lshift__()`
- `__rshift__()`, ...

이러한 다양한 메소드들을 통해서 산술연산( +, -, \*, @, /, //, %, \*\*, <<, >>, &, ^, | ) 을 수행한다. 책 2장의 다양한 연산자는 내부적으로 이 메소드의 호출로 동작한다.



### NOTE : `__div__()`와 `__truediv__()` 메소드

파이썬의 나누기를 수행하는 연산자 /는 파이썬 2 버전에서는 `__div__()` 메소드로 해석되었으나 파이썬 3 버전부터는 `__truediv__()` 메소드로 해석된다. 따라서 파이썬 3에서 `200/100`은 `(200).__truediv__(100)`로 해석되어 2.0이라는 float 값을 반환한다. 반면 `(200).__div__(100)`은 오류를 발생시킨다.



## LAB 9-1 : 객체와 메소드 호출

1. 다음 메소드 호출의 결과는 무엇인가?

```
>>> (200).__sub__(100)
```

a) \_\_\_\_\_

```
>>> (200).__mul__(100)
```

b) \_\_\_\_\_

```
>>> (200).__truediv__(100)
```

c) \_\_\_\_\_

2. 다음 메소드 호출의 결과는 무엇인가?

```
>>> [10, 20, 30, 40].pop()
```

a) \_\_\_\_\_

3. 다음 중 리스트 객체가 호출할 수 없는 메소드는 무엇인가?

1) pop()          2) keys()          3) remove()          5) get()

4. 다음과 같은 방법으로 int 클래스의 메소드와 속성을 조사하여라.

```
>>> dir(int)
```

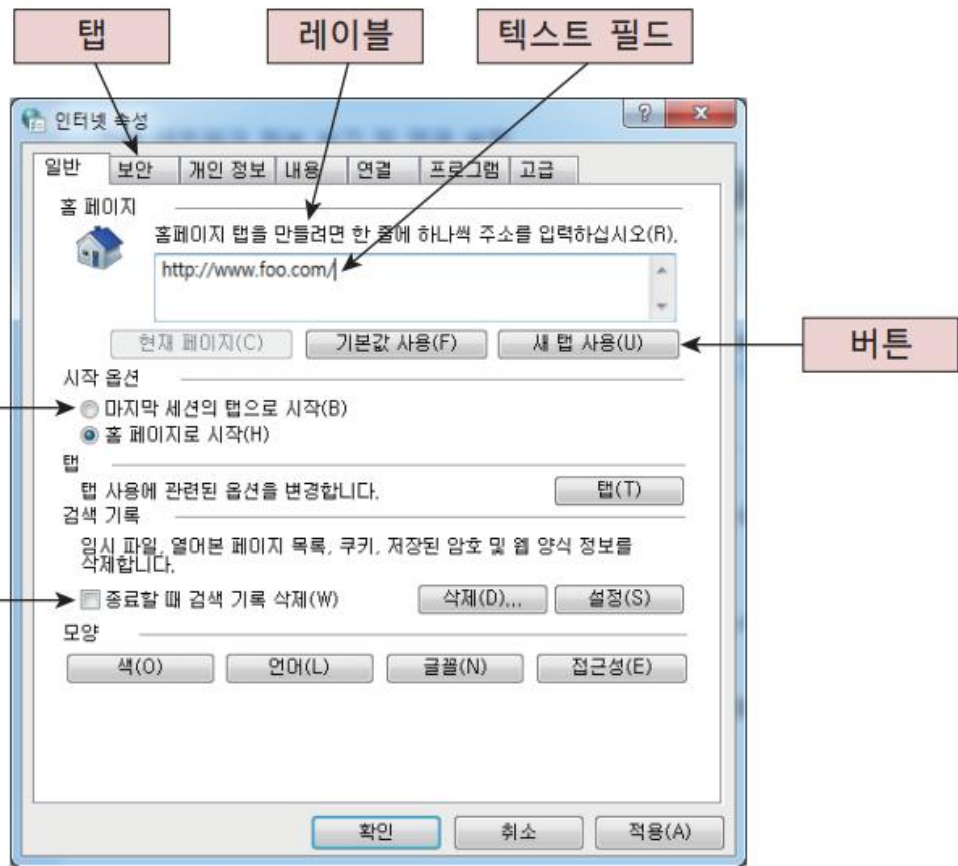
5. 다음과 같은 방법으로 list 클래스의 메소드와 속성을 조사하여라.

```
>>> dir(list)
```

# 객체 지향 프로그래밍과 절차적 프로그래밍

- 객체 지향 프로그래밍 **OOP:object oriented programming**
  - 프로그램을 짤 때, 프로그램을 실제 세상에 가깝게 모델링 하는 기법
  - 컴퓨터가 수행하는 작업을 **객체들 사이의 상호작용**으로 표현
  - 클래스 **class**나 객체 **object** 들의 집합으로 소프트웨어를 개발하자는 개념
  - Java, Python, C++, C#, Swift 등 현재 사용중인 많은 프로그래밍 언어에서 채택
- 절차적 프로그래밍 언어 **procedural programming language**
  - 함수나 모듈을 만들어두고 이것들을 문제해결 순서에 맞게 호출하여 수행하는 방식
  - C, Fortran, Basic 등의 고전적인 프로그래밍 언어에서 사용함
  - 그래픽 사용자 인터페이스 **graphic user interface** 시스템과 같이 다양한 그래픽 요소들이 있을 경우 효과적으로 문제해결을 하기 힘들다.

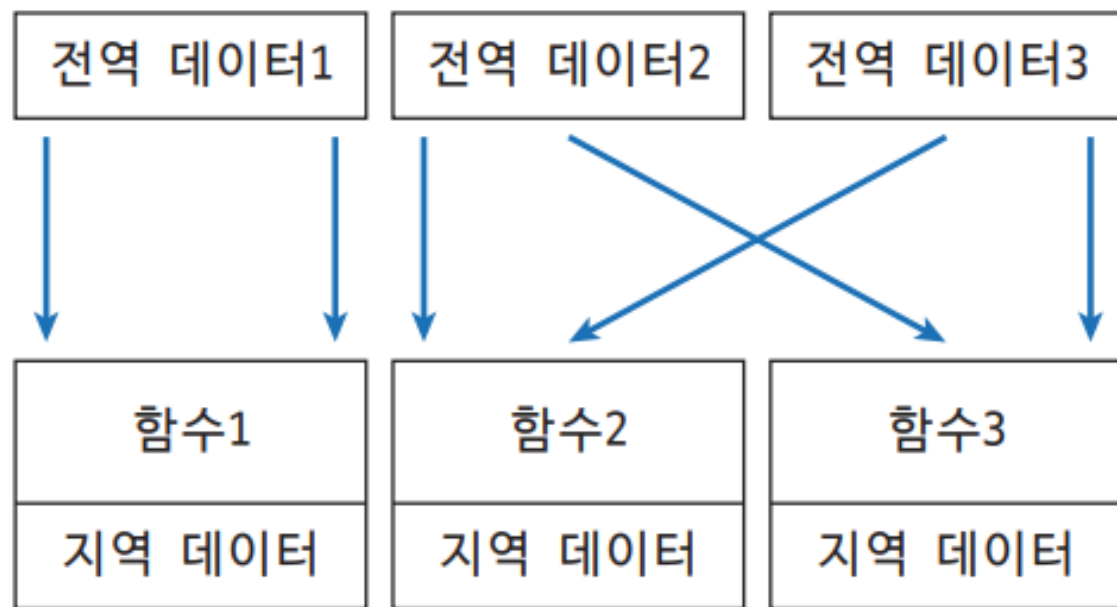
아래의 다양한 구성요소(객체)들이 상호작용하도록 하자는 것이 객체지향 프로그래밍의 핵심



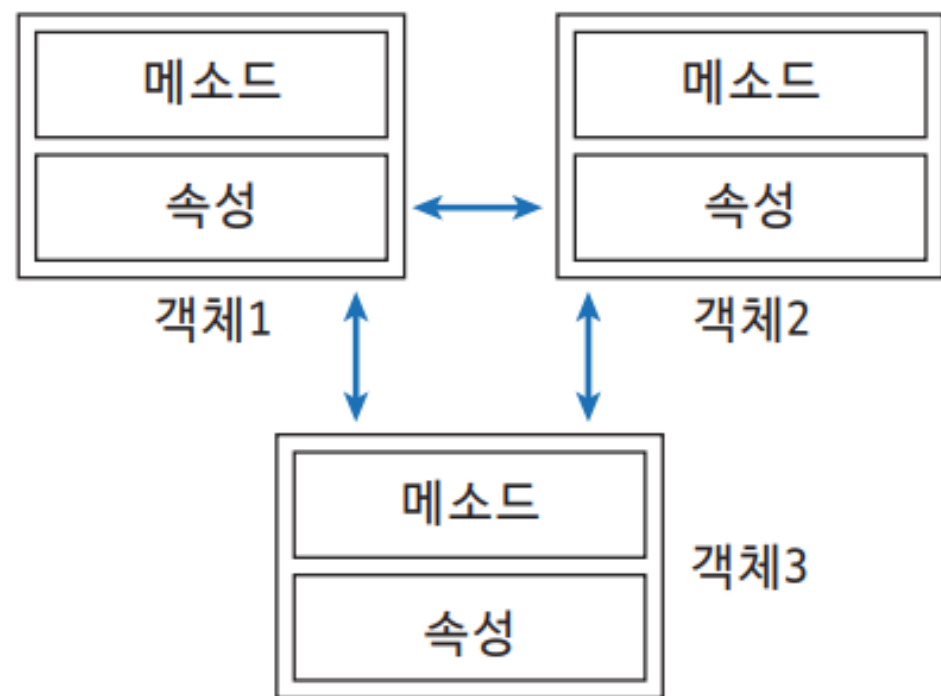
[그림 9-3] 게임 화면과 객체들의 상호 작용

[그림 9-2] 윈도 운영체제 인터넷 속성창의 나타난 여러 가지 그래픽 객체들

- 절차적 프로그래밍 방식procedural programming
  - 데이터들이 많아지고 함수가 많아진다면 매우 많은 화살표와 함수 호출이 필요
  - 대규모 프로젝트에서는 큰 어려움
- 객체 지향 프로그래밍OOP:objected oriented programming
  - 잘 설계된 클래스를 이용하여 객체를 생성
  - 클래스는 속성과 행위를 가지도록 설계하고 이 클래스를 이용하여 실제로 상호작용하는 객체를 만들어서 프로그램에 적용시키는 방법을 사용
- 객체 지향 프로그래밍 방식이 개발이나 소프트웨어 업데이트시의 유지보수 비용이 매우 적게 들기 때문에 최근 프로그래밍 경향은 대부분 객체 지향 방식을 선호



1) 절차적 프로그래밍



2) 객체 지향 프로그래밍

[그림 9-4] 1) 절차적 프로그래밍과 2) 객체 지향 프로그래밍의 비교





NOTE : Everything in Python is an object.

우선 다음과 같은 코드를 다시 한 번 살펴보자.

```
'tiger'.upper()  
animals.append('rabbit')
```

문자열은 upper()라는 메소드로  
대문자 변환을 수행

리스트는 append()라는 메소드로 새  
요소를 추가함

이 코드에서 사용한 'tiger'라는 문자열과 animals라는 리스트는 upper(), append()와 같은 메소드를 통해서 지정된 일을 수행할 수 있는데 이러한 프로그래밍의 구성 요소를 **객체object**라고 한다. 파이썬은 객체 지향 프로그래밍 언어로 여기에서 사용되는 데이터는 모두 객체이다.

그렇다면 다음과 같은 수식과 연산에 사용되는 n이나 100, 200은 객체일까?

```
n = 100 + 200
```

수식 연산 역시 아래의 코드와 같이 메소드  
호출로 동작함

그렇다. 위의 코드는 다음 코드와 동일하다.

```
n = (100).__add__(200)
```

즉, 위의 코드에서 100이라는 정수 객체는 \_\_add\_\_() 메소드를 이용해서 200이라는 객체를 더하는 일을 한다. 그리고 이 객체는 n이라는 변수가 참조하여 접근하게 된다. 이런 점에서 더하기(+) 연산자 역시 본질적으로 animals.append('rabbit')이라는 메소드 호출 코드와 수행 절차가 동일하다.



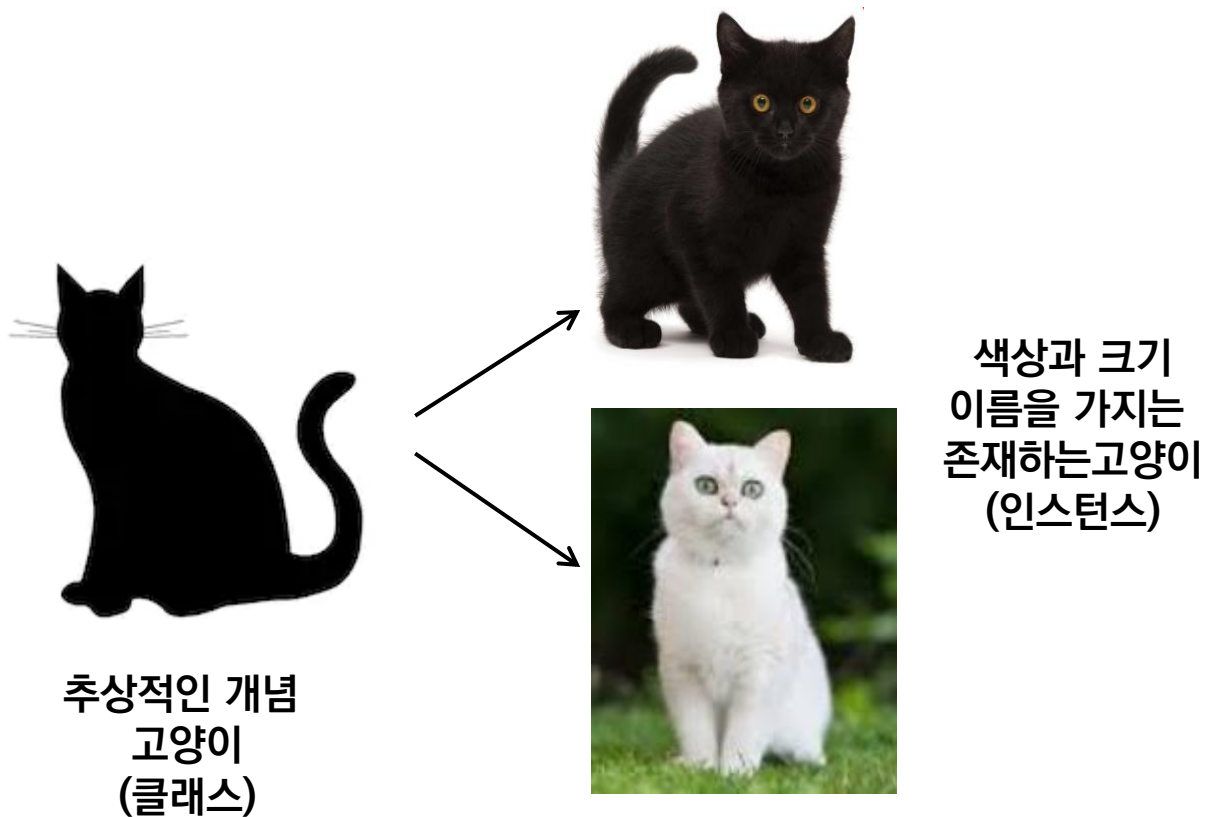
## LAB 9-2 : 용어 정리

1. 다음 용어를 정의하여라.

- a) 객체 지향 프로그래밍
- b) 절차적 프로그래밍
- c) 그래픽 사용자 인터페이스

2. 객체 지향 프로그래밍 기법과 절차적 프로그래밍 기법의 차이점을 기술하여라.

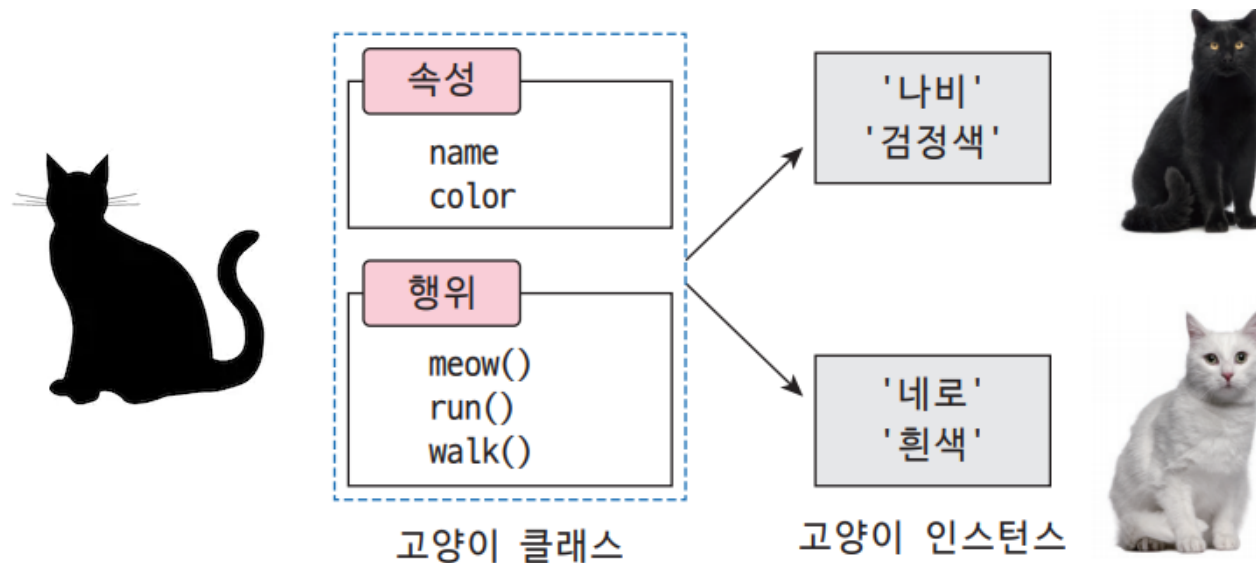
# 클래스와 객체, 인스턴스



- 클래스 **class**
  - 프로그램 상에서 사용되는 속성과 행위를 모아놓은 집합체
  - 객체의 설계도 혹은 템플릿(형틀 **template**), 청사진 **blueprint**
- 인스턴스 **instance**
  - 클래스로부터 만들어지는 각각의 개별적인 객체
  - 서로 다른 인스턴스는 서로 다른 속성 값을 가질 수 있음.

## 9.3 클래스와 객체, 인스턴스

- 클래스 **class**
  - 프로그램 상에서 사용되는 속성과 행위를 모아놓은 집합체
  - 객체의 설계도 혹은 템플릿(틀) **template**, 청사진 **blueprint**
- 인스턴스 **instance**
  - 클래스로부터 만들어지는 각각의 개별적인 객체
  - 서로 다른 인스턴스는 서로 다른 속성 값을 가질 수 있음.



[그림 9-5] 고양이 클래스와 인스턴스 개념도



### LAB 9-3 : 용어 정리

1. 다음 용어를 정의하여라.

- a) 클래스
- b) 객체
- c) 인스턴스
- d) 클래스의 속성
- e) 클래스의 동작

## 9.4 클래스 정의와 인스턴스

- 고양이 클래스는 아주 추상적인 개념



[그림 9-6] 고양이 클래스와 인스턴스의 관계

- 클래스의 정의 방법
  - class라는 키워드를 써 준 후 class의 이름을 써 준다. 그 후 필요한 속성과 메소드를 파이썬 문법에 맞게 써 준다
- pass 문은 아무런 역할을 하지 않는 파이썬 명령문

코드 9-1 : Cat 클래스 정의와 인스턴스 생성 문법

cat\_pass.py

```
class Cat:                # Cat 클래스의 정의
    pass

nabi = Cat()              # Cat 인스턴스 생성
print(nabi)
```

```
class ClassName :
```

```
    <statement-1>
```

```
    ...
```

```
    <statement-n>
```

실행결과

⟨\_\_main\_\_.Cat object at 0x7f78399e0eb8⟩



## NOTE : 객체와 인스턴스

많은 책에서 객체와 인스턴스를 비슷한 개념으로 섞어서 사용한다. 두 용어는 매우 비슷해서 구분하지 않는 경우도 많지만 조금 더 엄밀하게 정의하자면 객체는 하나의 사물로 정의할 수 있으며 인스턴스는 클래스에 의해 만들어진 사물로 정의해서 사용한다.

예를 들어 이 책에서 정의한 Cat이라는 틀은 클래스이며 이 Cat이라는 클래스에 의해서 만들어진 사물 nabi는 "객체"이면서, 동시에 "Cat 클래스의 인스턴스"라고 이야기 할 수 있다.

정리하자면 다음과 같은 표현은 올바른 표현이다.

- 1) 파이썬의 모든 객체는 자료형을 가진다.
- 2) 파이썬의 인스턴스는 클래스로부터 만들어 진다.
- 3) 파이썬의 객체는 클래스로부터 만들어 진다.
- 4) 파이썬의 클래스는 객체이다.
- 5) nabi는 객체이다.
- 6) Cat 클래스의 인스턴스는 nabi이다.
- 7) 100은 int 형 객체이다.

한편 다음은 **올바르지 않거나 부자연스러운 표현**이다.

- 1) nabi는 인스턴스이다.(?)
- 2) Cat 클래스는 인스턴스이다.(X)



- 클래스 내부에서 정의되어 클래스나 클래스 인스턴스가 사용하는 함수를 메소드 **method** 혹은 멤버 함수 **member function**라 한다.
- **meow()** 메소드의 매개변수인 **self**는 자기 자신을 참조하는 변수이며 메소드의 첫 번째 매개변수로 반드시 들어가야 한다

코드 9-2 : Cat 클래스 정의와 meow() 메소드

cat\_class.py

class Cat:

def meow(**self**):

print('야옹 야옹~~~')

nabi = Cat()

nabi.meow()

Cat 클래스내의 함수로 메소드라고 함

Cat()을 통해 Cat 클래스의 객체를 생성함. 이제 nabi.meow()를 통해 메소드 호출이 가능함

실행결과

야옹 야옹~~~

- 인스턴스의 메소드 호출 문법

인스턴스이름.메소드([인자])

- nabi라는 인스턴스는 Cat이라는 클래스가 가진 meow()라는 메소드를 사용할 수 있음

코드 9-3 : Cat 클래스 정의와 여러 개의 객체 생성

many\_cats.py

```
class Cat:
```

```
    def meow(self):
```

```
        print('야옹 야옹~~~')
```

```
nabi = Cat()
```

```
nabi.meow()
```

```
nero = Cat()
```

```
nero.meow()
```

```
mimi = Cat()
```

```
mimi.meow()
```

nabi 객체가 meow()실행

nero 객체가 meow()실행

mini 객체가 meow()실행

실행결과

야옹 야옹~~~

야옹 야옹~~~

야옹 야옹~~~



#### LAB 9-4 : Dog 클래스와 인스턴스 생성

1. 다음 기능을 가진 Dog 클래스와 객체를 생성하라.

a) `def bark(self):` 라는 메소드를 가진다. 이 메소드를 통해 짖는 소리를 출력한다.

d) `my_dog = Dog()` 라는 명령어로 Dog 인스턴스를 생성하고 `my_dog`로 참조한다.

e) `my_dog.bark()` 라는 메소드로 개짖는 소리를 다음과 같이 출력한다.

"멍멍~~"

## 코드 9-4 : 생성자를 가진 Cat 클래스의 정의와 인스턴스 생성

init\_cats.py

```
class Cat:
```

```
    # 생성자 혹은 초기화 메소드라 한다
```

```
    def __init__(self, name, color='흰색'):
```

```
        self.name = name                # name이라는 인스턴스 변수를 생성
```

```
        self.color = color              # color라는 인스턴스 변수를 생성
```

```
    # 고양이의 정보를 출력하는 메소드
```

```
    def meow(self):
```

```
        print('내이름은 {}, 색깔은 {}, 야옹 야옹~~'.format(self.name, self.color))
```

```
nabi = Cat('나비', '검정색')           # nabi 인스턴스 생성
```

```
nero = Cat('네로', '흰색')             # nero 인스턴스 생성
```

```
mimi = Cat('미미', '갈색')            # mimi 인스턴스 생성
```

```
nabi.meow()
```

```
nero.meow()
```

```
mimi.meow()
```

### 실행결과

내이름은 나비, 색깔은 검정색, 야옹 야옹~~

내이름은 네로, 색깔은 흰색, 야옹 야옹~~

내이름은 미미, 색깔은 갈색, 야옹 야옹~~

- 두 번째 매개변수 name과 세 번째 매개변수 color는 인스턴스의 속성에 해당하는 이름과 색상을 할당하기 위한 변수

```
class Cat:  
    def __init__(self, name, color):  
        ....
```

```
nabi = Cat('나비', '검정색')
```

```
nero = Cat('네로', '흰색')
```

```
mimi = Cat('미미', '갈색')
```

[그림 9-7] self 매개변수의 의미와 인스턴스와의 관계

- 인스턴스 변수 `instance variable`, 멤버 변수 `member variable`, 혹은 필드 `field`
  - 각각의 인스턴스들이 개별적으로 가지는 속성을 저장하는 변수

```
self.name = name
self.color = color
```

에서 `self`는 각각 다음과 같이 해석된다

nabi  
인스턴스

```
nabi.name = '나비'
nabi.color = '검정색'
```

nero  
인스턴스

```
nero.name = '네로'
nero.color = '흰색'
```

mimi  
인스턴스

```
mimi.name = '미미'
mimi.color = '갈색'
```

[그림 9-8] `self.name`, `self.color`의 의미와 인스턴스와의 관계



#### NOTE : 클래스와 생성자의 디폴트 매개변수 값

파이썬은 클래스에 대해 하나의 생성자만을 허용한다. C++나 Java와 같은 프로그래밍 언어는 여러 가지 형태의 생성자를 허용하고 이 때는 클래스의 이름을 중복 정의하여 사용할 수 있다. 이 때문에 위에서 만든 `__init__()` 메소드의 생성자 매개변수에 `(self, name, color = '흰색')`와 같이 `color = '흰색'` 형식의 디폴트 매개변수를 넣어주면, `Cat('네로')`와 `Cat('네로', '흰색')`을 모두 사용할 수 있다.



### LAB 9-5 : Dog 클래스와 인스턴스 생성

1. 다음 기능을 가진 Dog 클래스를 생성하고 인스턴스와 메소드를 호출하여라.

- a) name 이라는 속성을 가진다.
- b) def \_\_init\_\_(self, name): 이라는 초기화 메소드를 가진다. 이 메소드를 통해서 이름을 초기화 한다.
- c) def bark(self): 라는 메소드를 가진다. 이 메소드를 통해 짖는 소리를 출력한다.
- d) my\_dog = Dog('Jindo') 라는 명령어로 my\_dog 인스턴스를 생성한다.
- e) my\_dog.bark() 라는 메소드로 개짖는 소리를 다음과 같이 출력한다.  
"멍멍~~"



# 문자열화 메소드

- 생성자
  - 객체를 만들 때 인스턴스 내부의 변수가 기본값을 가지도록 하는 역할을 하는 메소드
  - `__init__`이라는 이름을 가진다
  - 객체가 생성될 때 자동으로 실행
- `__str__()` 메소드
  - 객체가 어떤 이름과 색상정보를 가지는지 알 수 있는 메소드
- `__main__`은 현재 파이썬 인터프리터에 의해 수행되는 메인 프로그램을 의미하는데 현재 수행중인 프로그램을 지칭
- 16진수는 이 객체의 아이디`id`

<\_\_main\_\_.Cat object at 0x0000000004CDD2E8>

코드 9-5 : \_\_str\_\_ 메소드와 print() 함수에서 적용하기

str\_cats.py

```
class Cat:
    def __init__(self, name, color):
        self.name = name
        self.color = color

        # Cat 객체의 문자열 표현방식

    def __str__(self):
        return 'Cat(name='+self.name+', color='+self.color+)'

nabi = Cat('나비', '검정색')          # nabi 인스턴스 생성
nero = Cat('네로', '흰색')           # nero 인스턴스 생성
print(nabi)
print(nero)
```

#### 실행결과

Cat(name=나비, color=검정색)

Cat(name=네로, color=흰색)

- Cat 클래스 내에 다음과 같이 \_\_str\_\_() 메소드를 추가해보기
- \_\_str\_\_() 은 어떤 객체의 문자열 표현 방식을 정의하는데 반환 값은 문자열이 됨

- 위의 코드 아래에 다음과 같이 {} 플레이스홀더를 두고 format() 메소드로 출력을 하게되어도 \_\_str\_\_() 메소드가 수행됨을 알 수 있음

```
print('nabi의 정보 : {}'.format(nabi))
```

```
nabi의 정보 : Cat(name=나비, color=검정색)
```



## LAB 9-6 : Dog 클래스와 문자열화 메소드

1. 다음 기능을 가진 Dog 클래스를 생성하고 인스턴스와 메소드를 호출하여라.

- a) name 이라는 속성을 가진다.
- b) def \_\_init\_\_(self, name): 이라는 초기화 메소드를 가진다. 이 메소드를 통해서 이름을 초기화 한다.
- c) my\_dog = Dog('Jindo') 라는 명령어로 my\_dog 인스턴스를 생성한다.
- d) Dog 클래스의 \_\_str\_\_() 메소드를 정의하여 print('my\_dog의 정보 :',my\_dog)와 같은 명령문에 대해 다음과 같은 출력이 나타나도록 하여라.

my\_dog의 정보 : Dog(name = Jindo)

# 캡슐화

- Cat 클래스에 나이를 의미하는 age라는 속성을 부여
  - nabi.age = -5를 넣게 된다면 코드상의 문법적인 문제는 없으나 고양이의 나이가 음수가 되는 논리적인 오류 발생
- 캡슐화encapsulation
  - 클래스의 속성을 외부에서 접근할 때 오류를 줄일 수 있음

## 코드 9-6 : nabi.age에 직접 값을 할당하기

cat\_age\_change.py

```
class Cat:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # Cat 객체의 문자열 표현방식
    def __str__(self):
        return 'Cat(name='+self.name+', age='+str(self.age)+')'

nabi = Cat('나비', 3) # nabi 인스턴스 생성
print(nabi)
nabi.age = 4
nabi.age = -5
print(nabi)
```

age가 음수가 되는  
비정상적 상황

### 실행결과

Cat(name=나비, age=3)  
Cat(name=나비, age=-5)

문법 오류는 아니지만 값이  
보호받지 못한 상태의 논리적  
문제

- 캡슐화(encapsulation)
  - 메소드와 변수를 외부에서 함부로 조작하는 것을 제한
  - 데이터를 보호
  - 우연히 값이 변경되는 것을 방지



**[그림 9-9]** 캡슐화의 개념도: 클래스의 메소드와 변수를 외부에서 함부로 조작하지 못하도록 감싸고 제한하는 기능

## 코드 9-7 : set\_age() 메소드를 통해서 age 값을 할당하기

cat\_age\_with\_setter\_getter.py

```
class Cat:
    def __init__(self, name, age):
        self.__name = name
        self.__age = age
    # Cat 객체의 문자열 표현방식
    def __str__(self):
        return 'Cat(name='+self.__name+', age='+str(self.__age)+')'
    # self.__age를 외부에서 자유롭게 접근하는 것을 제한하고 음수가 되지 않도록 함
    def set_age(self, age):
        if age > 0:
            self.__age = age
    def get_age(self):
        return self.__age
```

```
nabi = Cat('나비', 3) # nabi 인스턴스 생성
print(nabi)
nabi.set_age(4)      # set_age() 메소드를 통해서 age에 접근
nabi.set_age(-5)     # set_age() 메소드를 통해서 age가 음수가 되지 않도록 함
print(nabi)
```

- if 조건식을 넣어서 age 값이 음수일때는 할당이 되지 않도록 해보기
- setXXX와 같이 시작하는 메소드를 세터setter라고 함
- 반대로 getXXX와 같이 시작하는 게터getter를 통해서 멤버 값을 읽어오는 것도 가능

### 실행결과

```
Cat(name=나비, age=3)
Cat(name=나비, age=4)
```

나이가 음수가 되는 비논리적  
상황을 해결하는 방법



- 캡슐화를 통해 보다 안전하게 멤버 내부의 변수를 보호

Cat 클래스의 속성 :  
외부에 공개하고 싶지 않은 속성

```
class Cat:
    def __init__(self, name, age):
        self.__name = name
        self.__age = age
        ...
    def set_age(self, age):
        if age > 0:
            self.__age = age
```

세터 : set\_age() 라는 메소드를  
통해서 속성에 접근

# 객체의 아이덴티티 연산 identity operator : is, is not

- is 연산자는 두 인스턴스가 같으면 True를 반환하며, 그렇지 않으면 False를 반환. is not은 반대 역할을 함

코드 9-8 : 객체의 아이덴티티 연산자 : is, is not 연산자

is\_test.py

```
list_a = [10, 20, 30]          # 리스트 객체를 참조하는 list_a
list_b = [10, 20, 30]          # 리스트 객체를 참조하는 list_b

if list_a is list_b:           # 두 리스트 객체가 같은지 검사함
    print('list_a is list_b')
else:
    print('list_a is not list_b')
```

실행결과

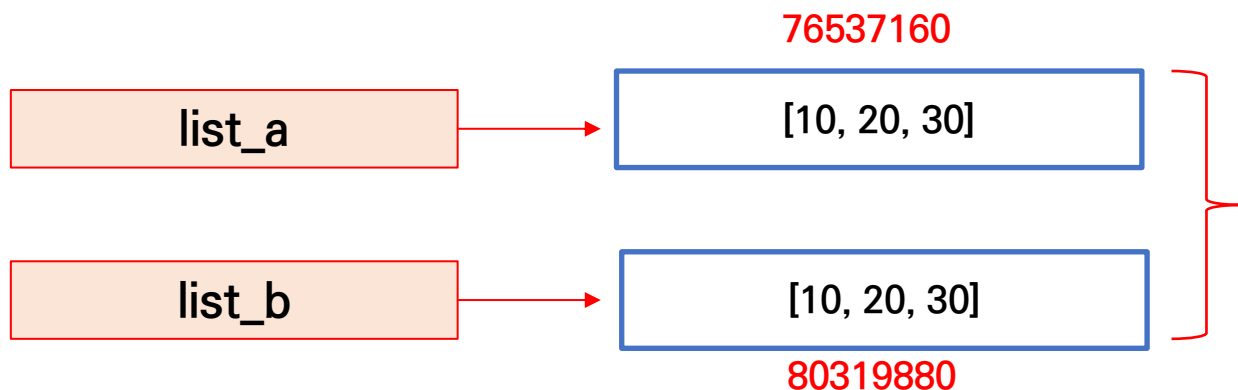
list\_a is not list\_b

- `id(list_a)`, `id(list_b)`를 출력하는 방법으로 결과 확인

```
print('id(list_a) =', id(list_a), ', id(list_b) = ', id(list_b))
```

```
id(list_a) = 76537160 , id(list_b) = 80319880
```

- 출력 결과 `list_a`와 `list_b`는 각기 다른 아이디 값을 가짐



두 객체의 내용(속성값)은 `[10, 20, 30]`이지만 서로 다른 메모리에 저장되는 다른 객체이므로 아이디가 다르게 출력됨(**is**는 **아이디 값을 비교함**)

`list_a is list_b` 는 **False**

- == 연산자는 두 인스턴스의 속성 값 즉 인스턴스 변수 값이 서로 일치하는지 확인하는데 사용

코드 9-9 : 두 개의 리스트와 == 연산

equal\_test.py

```
list_a = [10, 20, 30]           # 리스트 객체를 참조하는 list_a
list_b = [10, 20, 30]           # 리스트 객체를 참조하는 list_b

if list_a == list_b:             # 리스트 객체의 속성 값이 같은지 비교함
    print('list_a == list_b')
else:
    print('list_a != not list_b')
```

실행결과

list\_a == list\_b

76537160

[10, 20, 30]

==

[10, 20, 30]

80319880

list\_a = list\_b 는 True

## 코드 9-10 : 두 개의 문자열 참조변수와 is, is not 연산

equal\_string\_test.py

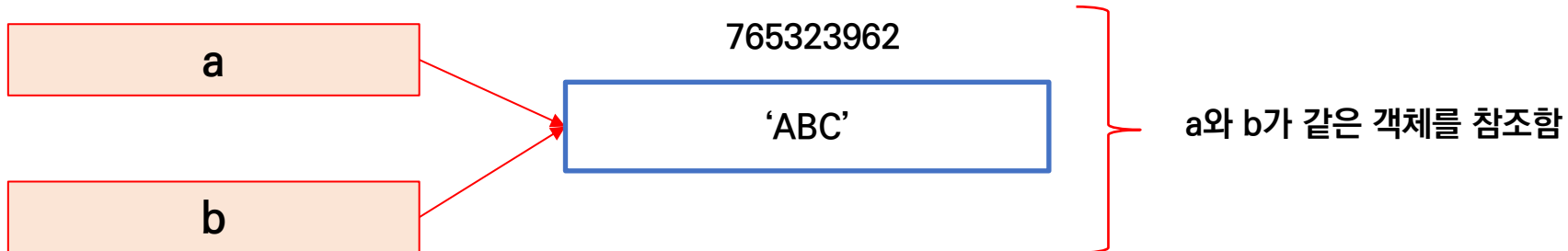
```
a = 'ABC'                # 문자열 객체를 참조하는 변수 a
b = 'ABC'                # 문자열 객체를 참조하는 변수 b
if a is b:               # 문자열 객체 a, b가 같은가 비교
    print('a is b')      # 문자열 객체 a, b는 같은 객체를 참조함
else:
    print('a is not b')
```

문자열은 불변객체(immutable) 이므로  
이 방법을 사용해도 아무런 문제가  
발생하지 않는다

### 실행결과

a is b

파이썬의 str 객체는 문자 객체를 저장하는 표에 넣어  
놓은 후 a, b에 같은 문자를 할당할 경우 **메모리를  
절약**하기 위해서 같은 저장위치를 참조한다





### LAB 9-7 : 정수 객체의 is 연산

1. 다음 코드의 수행 결과는 무엇인가? 결과를 적고 그 이유를 설명하여라.

```
n = 100
m = 100
if n is m:
    print('n is m')
else:
    print('n is not m')
```

# 클래스와 특수 메소드

- 2차원 벡터를 표현하는 Vector2D라는 클래스를 구현하고 이 클래스를 통해서 특수 메소드의 필요성에 대해 살펴보자.

코드 9-11 : 사용자 정의 Vector2D 클래스와 + 연산

```
vector2d_error.py
class Vector2D:
    def __init__(self, x, y):
        self.x = x
        self.y = y

v1 = Vector2D(30, 40)
v2 = Vector2D(10, 20)
v3 = v1 + v2                                     # Vector2D의 + 연산이 정의되지 않았다:오류 출력
print('v1 + v2 = ',v3)
```

실행결과

**TypeError: unsupported operand type(s) for +: 'Vector2D' and 'Vector2D'**

- 에러가 나타나는 이유는 클래스 내부에 어떤 방식으로 덧셈을 할지 그 방법을 서술해야 하기 때문
- `add()`라는 메소드를 정의하고 두 벡터의 x 성분과 y 성분을 더하고 이 성분값을 초기값으로 가지는 `Vector2D`를 반환하도록 해보기



코드 9-12 : add() 메소드를 이용한 벡터의 덧셈

vector\_add.py

```
class Vector2D:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def __str__(self):
        return "({}, {})".format(self.x, self.y)
    def add(self, other):
        return Vector2D(self.x + other.x, self.y + other.y)

v1 = Vector2D(30, 40)
v2 = Vector2D(10, 20)
v3 = v1.add(v2)          # Vector2D의 add() 메소드 사용
print('v1.add(v2) =', v3)
```

**실행결과**

v1.add(v2) = (40, 60)

- **v1.add(v2)는 두 벡터의 합을 출력**
- **add() 메소드 대신에 +, -와 같은 연산자를 사용하면 보다 편리**

vector2d\_add\_sub.py

```
class Vector2D:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def __add__(self, other):
        return Vector2D(self.x + other.x, self.y + other.y)
    def __sub__(self, other):
        return Vector2D(self.x - other.x, self.y - other.y)
    def __str__(self):
        return "({}, {})".format(self.x, self.y)

v1 = Vector2D(30, 40)
v2 = Vector2D(10, 20)
v3 = v1 + v2
print('v1 + v2 =', v3)
v4 = v1 - v2
print('v1 - v2 =', v4)
```

실행결과

$v1 + v2 = (40, 60)$

$v1 - v2 = (20, 20)$

- `__add__()`에는 두 벡터의 합을 구하는 기능을 구현하고, `__sub__()`에는 두 벡터의 차를 구하는 기능을 구현
- $v1 + v2$ 는  $v1.__add__(v2)$ 와 같은 메소드 호출과 동일한 기능

연산자	특수 메소드	하는 일
$x + y$	<code>__add__(self, other)</code>	x와 y의 합을 구한다.
$x - y$	<code>__sub__(self, other)</code>	x와 y의 차를 구한다.
$x * y$	<code>__mul__(self, other)</code>	x와 y의 곱을 구한다.
$x ** y$	<code>__pow__(self, other)</code>	x의 y 거듭제곱을 구한다.
$x / y$	<code>__truediv__(self, other)</code>	x를 y로 나눈 값을 구한다.
$x // y$	<code>__floordiv__(self, other)</code>	x를 y로 나눈 몫을 구한다.
$x \% y$	<code>__mod__(self, other)</code>	x를 y로 나눈 나머지를 구한다.
$+x$	<code>__pos__(self)</code>	x를 구한다.
$-x$	<code>__neg__(self)</code>	x의 음수를 구한다.

참고 : 파이썬 2에서는 `__truediv__` 대신 `__div__` 을 사용하였음

파이썬의 대표적인 연산자와 해당하는 특수 메소드



## LAB 9-8 : 특수 메소드의 응용

1. 앞서 배운 `__mul__()`과 `__truediv__()` 메소드를 이용하여 두 벡터의 곱셈과 나눗셈 기능을 구현하여라. `v1`이 (30, 40)이고 `v2`가 (10, 20)이라고 가정하고 다음과 같은 결과가 나타나도록 출력문을 작성하여라.

```
v1 * v2 = (300, 800)
```

```
v1 / v2 = (3.0, 2.0)
```

2. 앞서 배운 `__neg__()` 메소드 이용하여 벡터의 음의 벡터를 구하시오. `v1`이 (10, 20)일 경우 출력 값은 다음과 같다.

```
-v1 = (-10, -20)
```

연산자	특수 메소드	하는 일
<b>x &lt; y</b>	<b>__lt__(self, other)</b>	x가 y보다 작은가?
<b>x &lt;= y</b>	<b>__le__(self, other)</b>	x가 y보다 작거나 같은가?
<b>x &gt;= y</b>	<b>__ge__(self, other)</b>	x가 y보다 크거나 같은가?
<b>x &gt; y</b>	<b>__gt__(self, other)</b>	x가 y보다 큰가?
<b>x == y</b>	<b>__eq__(self, other)</b>	x와 y가 같은가?
<b>x != y</b>	<b>__ne__(self, other)</b>	x와 y가 서로 다른가?

파이썬의 비교 연산자와 해당하는 특수 메소드



### LAB 9-9 : 벡터의 크기 비교하기

1. 앞서 배운 비교 연산자에 해당하는 특수 메소드를 이용하여 두 벡터의 크기를 비교하는 프로그램을 작성하시오.  $v1$ 이 (30, 40)이고  $v2$ 가 (10, 20)이라고 가정하면 다음과 같은 결과가 나타나도록 출력문을 작성하여라. (벡터의 크기는 각 성분값을 제곱하여 더한 후 제곱근을 취하여 구한다.)

$v1 > v2 = \text{True}$

$v1 \geq v2 = \text{True}$

$v1 < v2 = \text{False}$

$v1 \leq v2 = \text{False}$

- 이외에도 파이썬에서는 len()이라던지 float(), int(), str(), abs(), hash(), iter()과 같이 많은 내장 함수들이 존재함
- 이 내장함수들은 \_\_len\_\_(), \_\_float\_\_(), \_\_int\_\_(), \_\_str\_\_(), \_\_abs\_\_(), \_\_hash\_\_(), \_\_iter\_\_()과 같은 특수 메소드로 구현이 가능

# 객체와 참조, 할당연산의 의미

대화창 실습 : n = 100의 할당 연산과 객체의 id 값

```
>>> n = 100
```

```
>>> id(100)
```

```
4509016272
```

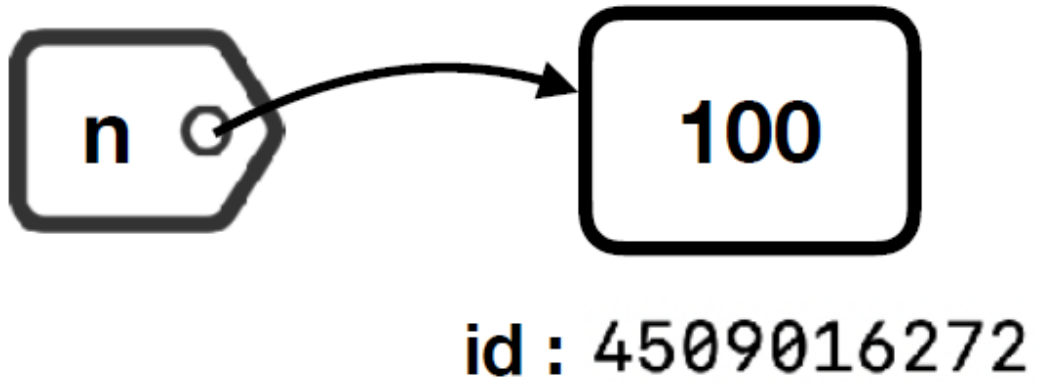
```
>>> id(n)
```

```
4509016272
```

- 변수명이란 객체에 대한 참조명
- 100이라는 객체를 n이라는 이름의 변수를 통해서 접근
- 각 객체의 아이디id를 출력해 보기



- 100이라는 값을 가지는 객체가 있을 때 이 객체에 대한 참조변수 n이 동일한 객체 100을 참조
- 변수 n을 통해 객체 100에 접근하는 것이 가능
- 다른 변수 m을 할당 연산자를 사용하여 100이라는 값을 가지는 객체에 접근시킬 수 있음



- 할당 연산자 =는 객체에 대한 참조와 재참조를 수행
- 두 변수가 동일한 객체를 참조하는 관계

대화창 실습 : 할당 연산자를 통한 참조와  $m = n$  연산자를 이용한 재참조

```
>>> n = 100
```

```
>>> m = n
```

```
>>> id(n)
```

```
4509016272
```

```
>>> id(m)
```

```
4509016272
```



#### NOTE : 불변 속성을 가진 객체의 참조

정수, 실수, 문자열, 부울, 튜플 객체는 불변형 객체이다. 따라서 다음과 같이 할 경우에 m, n은 동일한 객체 100을 참조한다.

```
>>> n = 100
```

```
>>> m = 100    # m = n과 동일한 결과
```

이 때문에 이 식의 수행 결과는 [그림 9-13]의 결과와 동일하다. 이와 같이 하는 이유는 메모리를 효율적으로 사용하기 위해서이다.

- 객체를 참조하는 변수 n과 m은 새로운 객체를 참조할 수도 있다
- 최초상태의 n은 100을 참조하고 있으며, m = n을 통해 m도 역시 동일한 객체를 참조하고 있음
- n = 200을 통해 m과 n이 다른 객체를 참조함

대화창 실습 : 할당 연산자를 통한 참조와 n = 200 연산자를 이용한 재참조

```
>>> n = 100
```

```
>>> m = n
```

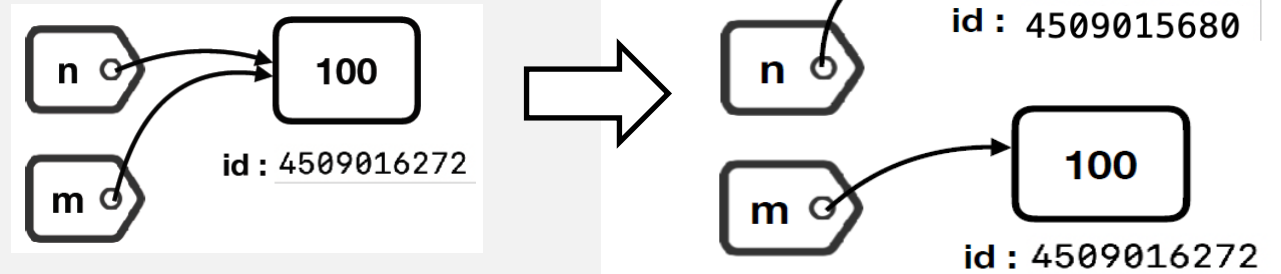
```
>>> n = 200
```

```
>>> id(n)
```

```
4509015680
```

```
>>> id(m)
```

```
4509016272
```



- $n = n + 1$ 의 연산자의 연산과 객체 할당 방법
- $n = n + 1$ 을 통해 새롭게 할당된  $n$ 과 이전의  $n$ 은 서로 다른 객체를 참조한다는것을 확인함

대화창 실습 :  $n = n + 1$  연산과 객체의 id 값의 변화

```
>>> n = 100
```

```
>>> id(n)
```

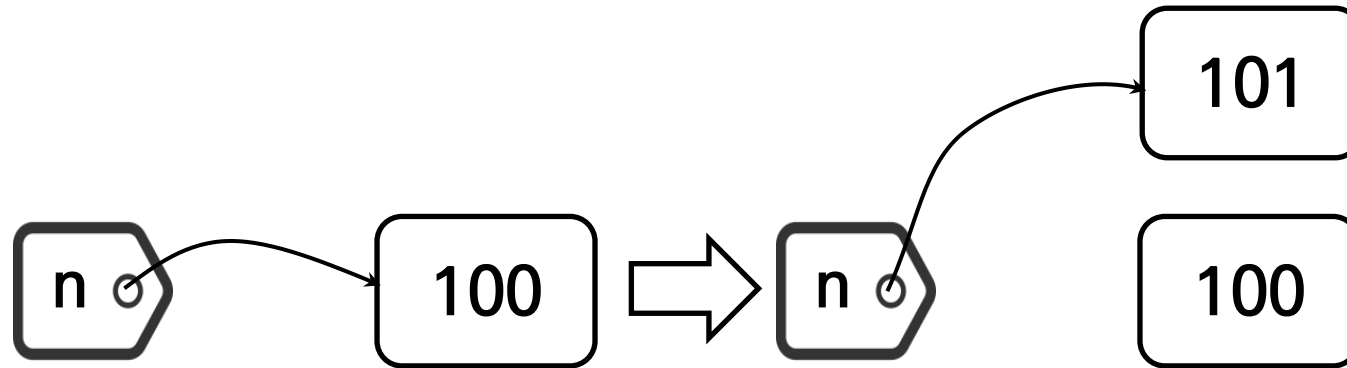
```
4509016272
```

```
>>> n = n + 1
```

```
>>> id(n)    # 할당 연산에 의해 n이 새로운 객체를 참조하고 있음
```

```
4509016346
```

- n이 참조하고 있는 정수형 객체가 변경 불가능 **immutable** 속성으로 정의되어 있음



- 더 이상 이 객체를 참조하는 변수가 존재하지 않으므로 메모리 낭비를 초래
- 메모리상에 존재하는 객체 중에서 참조가 사라져서 더 이상 참조할 방법이 없는 객체를 가비지 **garbage** 라고 부르며
- 주기적으로 정리하는 메모리 관리 절차를 가비지 수집 **garbage collection** 이라고 한다.



Questions?