

데이터과학을 위한

# 파이썬 프로그래밍



## 10. 객체 지향 프로그래밍

# 목차

1. 객체 지향 프로그래밍의 이해
2. 파이썬의 객체 지향 프로그래밍
3. Lab: 노트북 프로그램 만들기
4. 객체 지향 프로그래밍의 특징

01

객체 지향 프로그래밍의 이해

# 01. 객체 지향 프로그래밍의 이해

---

## ■ 객체 지향 프로그래밍을 배우는 이유

- 객체 지향 프로그래밍(Object Oriented Programming, OOP)은 함수처럼 어떤 기능을 함수 코드에 묶어 두는 것이 아니라, 그런 기능을 묶은 하나의 단일 프로그램을 객체라고 하는 코드에 넣어 다른 프로그래머가 재사용할 수 있도록 하는, 컴퓨터 공학의 오래된 프로그래밍 기법 중 하나이다.

# 절차지향 vs. 객체지향

- 절차지향- 명령들의 블록인 함수들의 조합으로 프로그램을 구성
- 객체지향- 데이터와 기능을 묶은 객체들의 조합으로 프로그램을 구성
  - 객체 지향 프로그래밍의 중요한 특징 => 클래스(class)와 객체(object)
  - 클래스는 새로운 타입(구조체 처럼)을 만들어낸다.
    - 붕어빵 틀을 제작하는 것과 유사한 동작
  - 클래스에 의해 예시화(instantiation)된 것이 객체이다. 클래스를 이용해 객체를 만들어낸다.
    - 붕어빵 틀(클래스)을 이용해 빵(객체)을 만들어 내는 것 => 예시화
    - 파이썬에서의 단순 사례: `int a, b, c;` # 3개의 변수(object)가 `int` 타입(class)로 생성되었다.

# 그림으로 살펴본 클래스와 객체

- 클래스(자동차)는 2가지 구성 성분으로 정의된다.
  - ▣ 데이터 : 변수
  - ▣ 동작 : 코드, 메서드(method)

데이터
메이커
모델
색상
연식
가격



동작
주행하기
방향바꾸기
주차하기

- 클래스 ⇨ 붕어빵 틀
- 객체 ⇨ 붕어빵
- 예시화 ⇨ 빵틀로 붕어빵을 찍어내기
- 클래스를 하나 정의해 놓으면 여러 이름의 객체를 생성할 수 있다.



# 01. 객체 지향 프로그래밍의 이해

## ■ 객체와 클래스

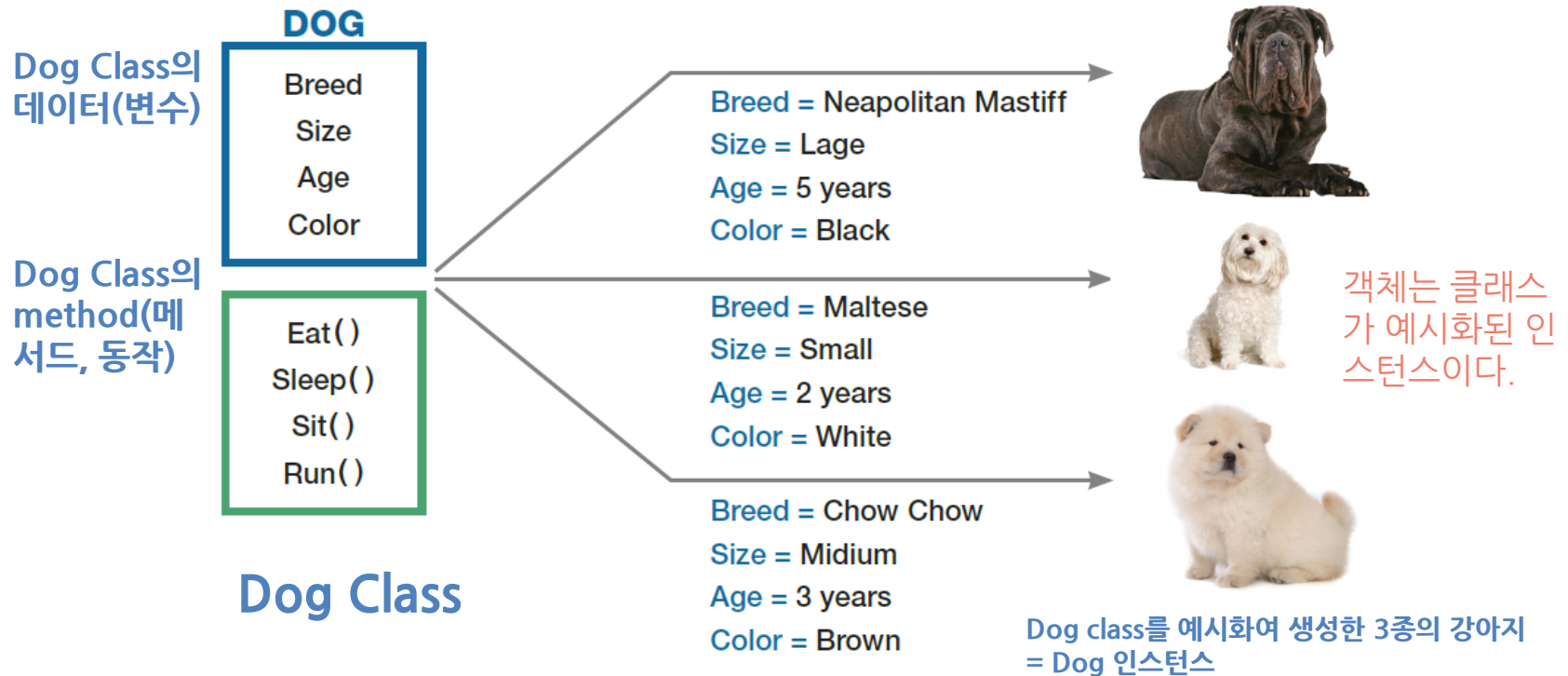
개념	설명	예시
객체(object)	실생활에 존재하는 실제적인 물건 또는 개념	심판, 선수, 팀
속성(attribute)	객체가 가지고 있는 변수	선수의 이름, 포지션, 소속팀
행동(action)	객체가 실제로 작동할 수 있는 함수, 메서드	공을 차다, 패스하다

[객체, 속성, 행동]

# 01. 객체 지향 프로그래밍의 이해

## ■ 객체와 클래스

- 클래스(class) : 객체가 가져야 할 기본 정보를 담은 코드이다.
- 클래스는 일종의 설계도 코드 주요 변수와 메소드를 담고 있다.
- 실제로 생성되는 객체를 인스턴스(instance)라고 한다.





# 02

## 파이썬의 객체 지향 프로그래밍

## 02. 파이썬의 객체 지향 프로그래밍

### ■ 클래스 구현하기

- 파이썬에서 클래스를 선언하기 위한 기본 코드 템플릿은 다음과 같다.

```
class SoccerPlayer(object):
```

클래스 예약어      클래스 이름      상속받는 객체명

[파이썬에서의 클래스 선언]

- 먼저 예약어인 class를 코드의 앞에 쓰고, 만들고자 하는 클래스 이름을 작성한다.
- 그 다음으로 상속받아야 하는 다른 클래스의 이름을 괄호 안에 넣는다.

## 02. 파이썬의 객체 지향 프로그래밍

### 여기서 잠깐! 파이썬에서 자주 사용하는 작명 기법

- 클래스의 이름을 선언할 때 한 가지 특이한 점은 기존과 다르게 첫 글자와 중간 글자가 대문자라는 것이다. 이것은 클래스를 선언할 때 사용하는 작명 기법에 의해 생성된다. 파이썬뿐 아니라 모든 컴퓨터 프로그래밍 언어에서는 변수, 클래스, 함수명을 짓는 작명 기법이 있다. 아래 표는 프로그래머가 흔히 사용하는 두 가지 작명 기법이다.

작명 기법	설명
snake_case	띄어쓰기 부분에 '_'를 추가하여 변수의 이름을 지정함, <u>파이썬 함수나 변수명에 사용됨</u>
CamelCase	띄어쓰기 부분에 대문자를 사용하여 변수의 이름을 지정함, 낙타의 혹처럼 생겼다하여 Camel 이라고 명명, <u>파이썬 클래스명에 사용됨</u>

[파이썬에서 자주 사용하는 작명 기법]

# 클래스 선언 및 객체 생성

A byte of Python: ch 12.2

A\_122\_oop\_simplestclass.py

```
2  # -----
3  # 몸체가 없는 간단한 클래스를 선언하고 그 클래스를 이용하여 객체를 생성해 본다.
4  # -----
5  class Person:
6      pass # An empty block
7
8
9  person = Person() # 인스턴스 person 생성
10 print("person=", person)
11 print("type(person)=", type(person))
```

```
person= <__main__.Person object at 0x000001B332FEAF08>
type(person)= <class '__main__.Person'>
```

# 모듈기반 클래스 선언 및 객체 생성

A byte of Python: ch 12.2

A\_122\_oop\_module.py + A\_122\_oop\_simplestclass.py

```
14 class Animal():
15     id = 'member variable of Animal'
16
17
18 if __name__ != "__main__": # main 프로그램의 자격이면 수행한다.
19     import os
20     _ = os.getcwd() # get Current Working Directory
21     print("\n'A_122_oop_module.py' is being loaded as a module from:\n" + _)
22 else: # import 되는 순간 수행된다..
23     Animal()
```

```
14 from A_122_oop_module import Animal
15
16 ani = Animal()
17 print("\nani=", ani)
18 print("type(ani)=", type(ani))
19 print("ani.id=", ani.id)
```

```
'A_122_oop_module.py' is being loaded as a module from:
D:\Work\@@Python\LectureMaterials\08_(ch10)_객체지향프로그래밍\Ch10

ani= <A_122_oop_module.Animal object at 0x000001ACA513AF08>
type(ani)= <class 'A_122_oop_module.Animal'>
ani.id= 'member variable of Animal'
```

# method

A byte of Python: ch 12.3

A\_123\_oop\_method.py

```
6 class Person:
7     def say_hi(self): # instance method는 self가 첫 번째 인자이다.
8         print('Hello, how are you?')
9
10    def say_this(self, msg): # instance method는 self가 첫 번째 인자이다.
11        print(msg)
12
13
14    p = Person() # instance를 생성한다.
15    p.say_hi() # instance의 메서드를 호출한다.
16    Person().say_hi() # 클래스의 함수를 호출한다.
17    p.say_this('Good day to you!') # msg 파라미터 패싱
```

```
Hello, how are you?
Hello, how are you?
Good day to you!
```

# The self

예제는 A\_123\_oop\_method.py, A\_124\_oop\_init.py에서 볼 수 있으므로 생략함

- 클래스에서 사용하는 함수(method)가 C 언어와 같은 일반 함수와 다른 점
  - 매개 변수의 목록에 항상 추가로 한 개의 변수(self)가 맨 앞에 추가되어야 한다.
  - 메서드를 호출할 때 이 변수에는 우리가 직접 값을 넘겨주지 않으며 대신 파이썬이 자동으로 값을 할당한다.
  - 이 변수에는 현재 객체 자신의 참조가 할당되며 일반적으로 self 라 이름 짓는다.
- MyClass라는 클래스를 생성했고, 이 클래스의 객체를 myobject 라는 이름으로 생성했다고 가정해 보자.
  - 객체 생성 : `myobject = MyClass( )`
  - 이 객체 안의 메서드를 호출할 때 : `myobject.method(arg1, arg2)`
  - 이 호출은 파이썬에 의해 자동적으로 `MyClass.method(myobject, arg1, arg2)`의 형태로 바뀌게 됩니다.
    - 클래스 이름을 이용하여 내부의 함수를 호출할 때는 첫번째 인자로 그 클래스에 의해 예시화된 객체를 지정한다.

# \_\_init\_\_ method

A byte of Python: ch 12.4

A\_124\_oop\_init.py

```
12 class Person:
13     # self는 객체 자신을 참조한다.
14     def __init__(self, ame='JH'):
15         self.name = ame
16
17     def say_hi(self):
18         print('Hello, my name is', self.name)
19
20
21 p = Person('Swaroop')    # ame='Swaroop'로 입력.
22 # 자동 호출되는 __init__ 함수에서 self.name 변수가 초기화 된다.
23 p.say_hi()
24
25 Person('Tom').say_hi()  # 위의 2줄을 한줄로 표시한 것.
26 Person().say_hi()      # 역시 1줄로 표시한 것. 단, ame는 default 값 사용.
27
28 p = Person('홍길동')    # p 객체를 생성한다.
29 Person.say_hi(p)        # p 객체를 self 객체에 넘긴다.
```

- `__init__` 메서드는 클래스가 인스턴스화될 때 자동으로 호출되는 special method이다.
  - ▣ *class initializer*라고도 한다.
- 이 메서드는 객체가 생성될 때 여러 가지 초기화 작업을 수행할 때 유용하다.
- 다른 객체지향 언어의 **생성자(constructor)**와 유사한 역할을 수행한다.

```
Hello, my name is Swaroop
Hello, my name is Tom
Hello, my name is JH
Hello, my name is 홍길동
```



# 클래스(Class)-필드

## 데이터와 code(procedure, function)의 모음

- Field : 데이터는 2가지의 타입의 필드를 갖는다.
  - **인스턴스 변수** - 객체 안에 저장된 객체 전용 변수. 개별 객체마다 따로 변수 공간이 할당되고 별개 값으로 관리된다.
  - **클래스 변수** - 클래스 자체에 내장된 변수. 같은 클래스로 생성된 모든 객체들간에 공용으로 사용하는 공용 변수.
- Method: 클래스에서 정의된 함수는 method라고도 한다.
  - 뒷면 계속

# 클래스(Class)-메서드

## 데이터와 code(procedure, function)의 모음

- Method: 클래스에서 정의된 함수는 method라고도 한다.
- 클래스의 이름으로 호출하는 메서드. 객체가 없이 호출 가능하다.
  - **Class method** - 클래스 변수를 접근할 수 있음. @classmethod를 지정하고 선언함.
  - **Static method** - 클래스 변수를 접근할 필요가 없는 클래스 메서드. @staticmethod를 지정하고 선언함
- 객체의 이름으로 호출하는 메서드. 객체가 생성되어야 호출가능하다.
  - **Instance method** - 인스턴스 변수만을 접근함.
- 특별 메서드(**special method**)
  - Initializer, Destructor - 객체의 생성과 소멸에 관계된. \_\_init\_\_, \_\_del\_\_
  - \_\_str\_\_ ...

# 클래스 선언

A byte of Python: ch. 12

A\_120P\_attribute.py

- class 키워드에 의해 생성된다.
- 클래스명
  - PEP 8 Coding Convention에 안내된 대로 각 단어의 첫 문자를 대문자로 하는 CapWords 방식으로 명명

```
1 class MyClass:
2     pass

In[2]: class Person:
...:     pass # An empty block
```

```
1 class ExampleClass(object):
2     class_attr = 0
3
4     def __init__(self, instance_attr):
5         self.instance_attr = instance_attr
```

[링크](#)

- 클래스 멤버-클래스의 구성 요소
  - 클래스에 들어있는 속성과 메서드를 그 클래스의 멤버(member)가 된다.
  - 이러한 필드와 메서드들을 통틀어 클래스의 속성(attribute) 이라 한다.
    - Instance attribute는 특정한 객체에만 속하는 변수로서 \_\_init\_\_ 메서드 안에서 self.변수를 통해 정의된다.
    - Class attribute는 모든 객체에 공통적인 변수로서 \_\_init\_\_ 메서드 밖에서 정의된다.
  - 다른 OOP 언어와 달리 파이썬은 Dynamic Language로서 새로운 attribute를 동적으로 추가할 수 있고, 메서드도 일종의 메서드 객체로 취급하여 attribute에 포함한다.

attribute 참고: <https://dojang.io/mod/page/view.php?id=1072>

# 클래스 선언

A byte of Python: ch. 12

A\_120P\_attribute.py

- attribute에는 메소드와 클래스에서 선언한 변수가 모두 포함된다.
- 클래스 속성은 곧 인스턴스 속성이 된다.
- `hasattr(object, name)`: 파이썬 내장함수
  - `object`가 주어진 `name`의 속성(attribute)을 갖고 있는지의 여부를 반환한다.
  - 만약 `object`에 `name`의 속성이 존재하면 `True`, 아니면 `False`를 반환한다.
- 클래스 속성과 인스턴스 속성의 차이
  - 클래스 속성 - `class` 안에서 할당하는 public(공개) 변수들(`__init__`) 밖에서 )
  - 인스턴스 속성 - `'self'`에 할당하는 변수들, 클래스 내의 메소드들

# 클래스 선언

A byte of Python: ch. 12

A\_120P\_attribute.py

```
17 class Animal:
18     num_ani = 0      # 클래스 변수. public(공개), 클래스 속성은 곧 인스턴스 속성이 된다.
19     __num_ani = 0    # 클래스 변수. private(비공개), 클래스 속성(x), 인스턴스 속성(x)
20
21     # 이하 'self'에 소속된 변수들은 모두 해당 인스턴스의 속성이다.
22     def __init__(self, name1, legs, wing):
23         self.legs = legs
24         self.name = name1
25         self.wings = wing
26         Animal.num_ani += 1
27
28     # 메소드들은 클래스 속성이면서 인스턴스 속성이다.
29     # self 소속변수는 instance 속성
30     def run(self):
31         pass
32
33     def add(self, att):      # new_att는 인스턴스의 속성이다. 그러나 att는 속성이 아니다.
34         self.new_att = att  # 클래스 속성(x), 인스턴스 속성(x). public variable(0).
35         new_att2 = 'swim'   # 클래스 속성(x), 인스턴스 속성(x). public variable(x).
36
37
38 object_p= Animal('Cat', 4, 0)  # name, legs, wings
```

	Instance Attribute	Class Attribute
name	O	X
legs	O	X
wings	O	X
run	O	O
add	O	O
num_ani	O	O
__num_ani	X	X
new_att	O	X
att	X	X
new_att2	X	X

# 객체(Object)

- 데이터, 데이터 구조체, 함수, 메서드 등이 identifier에 의해 지정될 수 있는 메모리에 있는 어떤 값.
- 보통 클래스의 instance를 말한다. = 클래스에 의해 생성된다.
  - `a = MyClass()` # MyClass 클래스 속성을 갖는 객체 a 생성
  - `b = MyClass()` # MyClass 클래스 속성을 갖는 객체 b 생성
- 객체는 변수를 통해 데이터를 저장할 수 있다.
- 자신이 속한 클래스의 함수를 통해 동작을 수행할 수 있다. 이 함수를 **메서드(method)**라고 한다.

# 클래스 변수와 객체 변수

A byte of Python: ch. 12.5

- Name space
  - 클래스 혹은 객체의 데이터, 즉 필드는 그 클래스 혹은 객체의 네임스페이스에 묶여 있다.
  - 이것은 필드의 이름은 그 클래스 혹은 객체 내부에서만 의미가 있음을 의미한다.
  - 그 이름이 통용되는 공간을 네임스페이스라고 한다.
- 필드에는 그것을 소유하고 있는 대상이 클래스인지 객체인지에 따라 클래스 변수와 객체 변수로 구분된다.

# 12.5. 클래스 변수와 객체 변수

## • 클래스 변수

- 그 클래스로부터 생성된 모든 인스턴스들이 접근할 수 있다. 모든 객체들간에 공유된다.
- 클래스 변수는 한 개만 존재하며 어떤 객체가 클래스 변수를 변경하면 모든 다른 인스턴스들에 변경 사항이 반영된다.
- 클래스 선언부 바로 밑에 선언한다. 메서드 바깥에...
- 객체 외부에서 액세스 가능하다. 즉 public이다. “클래스\_이름.클래스\_변수”

## • 객체 변수 혹은 인스턴스 변수

- 클래스로부터 생성된 각각의 객체/인스턴스에 속해 있는 변수이다.
- 이 경우에는 각각의 객체별로 객체 변수를 하나씩 따로 가지고 있으며, 서로 공유되지 않고 각 인스턴스에 존재하는 같은 이름의 필드끼리 서로 어떤 방식으로든 간섭되지 않는다.
- 객체 외부에서 액세스 가능하다. 즉 public이다. “객체\_이름.인스턴스\_변수\_이름”
  - 이것은 self.인스턴스\_변수\_이름 으로 선언했던 것이다.



# 클래스 변수(class variable)

실습 1: rectangle.py

```
rec = Rectangle(4, 5) # 4x5 사각형
Rectangle.count = 1 # 클래스 변수 변경
```

- 클래스를 정의할 때 메서드 밖에 존재하는 변수
- 해당 클래스를 사용하는 모두에게 공유로 사용되는 변수
- 클래스 변수는 클래스 내/외부에서 "**클래스명.변수명**" 으로 액세스 할 수 있다.
- 옆의 예제에서 count는 클래스변수로서 "**Rectangle.count**" 의 이름으로 액세스할 수 있다.

```
1 class Rectangle:
2     count = 0 # 클래스 변수
3
4     # 초기자(initializer) class initializer
5     def __init__(self, width, height):
6         # self.* : 인스턴스변수
7         self.width = width
8         self.height = height
9         Rectangle.count += 1
10
11     # 메서드 instance method
12     def calcArea(self):
13         area = self.width * self.height
14         return area
```

# Python 클래스는 기본적으로 모든 멤버가 public이다.

실습 1: private\_variable.py

- Python은 다른 언어에서 흔히 사용하는 public, protected, private 등의 접근 제한자 (Access Modifier)를 갖지 않는다.-> 하지만 개념은 지원한다.
- protected 개념은 없다.
  - Python 코딩 관례(Convention)상 내부적으로만 사용하는 변수 혹은 메서드는 그 이름 앞에 하나의 밑줄(\_) 을 붙인다(protected의 의미). 하지만 이는 코딩 관례에 따른 것일 뿐 실제 밑줄 하나를 사용한 멤버도 public 이므로 필요하면 외부에서 액세스할 수 있다.
- Private(비공개 클래스 속성) 개념
  - 만약 특정 변수 명이나 메서드를 private으로 만들어야 한다면 두 개의 밑줄(\_\_)을 이름 앞에 붙이면 된다.
- 클래스 변수는 클래스 외부에서 “클래스명.클래스변수”를 지정하여 접근할 수 있다.
- 인스턴스 변수도 클래스 외부에서 접근할 수 있다. 외부에서 읽고, 변경하고, 심지어 생성할 수도 있다. 인스턴스 변수는 클래스 밖에서는 "객체명.인스턴스\_변수명"와 같이 액세스 한다.
- 이와 별도로 @classmethod에서 정의한 클래스 메서드를 이용해서도 클래스 변수를 접근할 수 있다.

# Python 클래스는 기본적으로 모든 멤버가 public이다.

실습 1: private\_variable.py

```
5 class Something:
6     __prv = 'private' # private variable
7     _prt = 'protected' # public variable
8
9     def write_prv(self, value): # instance method
10         Something.__prv = value
11
12     def read_prv(self):
13         return Something.__prv
14
15
16 r = Something()
17 print(r.read_prv()) # private
18 r.write_prv(100)
19 print(r.read_prv()) # 100
20 r._prt = 'non protected'
21 print(r._prt) # non protected
22
23 r2 = Something()
24 r2._prt = 'new'
25 print(r2._prt) # new
26 print(r._prt) # non protected
27
28 # print(r.__prv) # 오류 발생. 외부에서 사적변수 접근 불가
29 # AttributeError: 'Something' object has no attribute '__prv'
```

```
private
100
non protected
new
non protected
```

# 인스턴스 변수(instance variable)

- 클래스 변수가 하나의 클래스에 하나만 존재하는 반면, 인스턴스 변수는 각 객체 인스턴스마다 별도로 존재한다.
- 클래스 정의에서 메서드 안에서 사용되면서 "self.변수"처럼 사용되는 변수를 인스턴스 변수라 한다.
- 이 변수는 각 객체 별로 서로 다른 값을 갖는다.
- 인스턴스 변수는 클래스 내부에서는 self.width 과 같이 "self." 을 사용하여 액세스하고, 클래스 밖에서는 "객체명.인스턴스변수"와 같이 액세스 한다.

```
1 class Rectangle:
2     count = 0 # 클래스 변수
3
4     # 초기자(class initializer)
5     def __init__(self, width, height):
6         # self.* : 인스턴스변수
7         self.width = width
8         self.height = height
9         Rectangle.count += 1
10
11    # 메서드(instance method)
12    def calcArea(self):
13        area = self.width * self.height
14        return area
```

```
# 4x5 사각형 instance 생성
rec = Rectangle(4, 5)
# 인스턴스 변수 변경
rec.width = 7
```

# 인스턴스 메서드(instance method)

- 인스턴스 이름을 통해 호출할 수 있는 메서드이다.
- 가장 흔히 쓰이는 인스턴스 메서드는 인스턴스 변수를 액세스할 수 있도록 메서드의 첫 번째 파라미터에 항상 객체 자신을 의미하는 "self"라는 파라미터를 갖는다.
- 옆의 예제에서 calcArea()가 인스턴스 메서드에 해당된다.
- 인스턴스 메서드는 인스턴스에 국한된 내부 변수만을 생성하거나 액세스 한다.

```
1  class Rectangle:
2      count = 0 # 클래스 변수
3
4      # 초기자(class initializer)
5      def __init__(self, width, height):
6          # self.* : 인스턴스 변수
7          self.width = width
8          self.height = height
9          Rectangle.count += 1
10
11     # 메서드(instance method)
12     def calcArea(self):
13         area = self.width * self.height
14         return area
```

# 클래스 메서드(class method)

실습 1: z\_class\_method.py

- 클래스 이름으로 호출하는 메서드 중 클래스 변수를 접근하는 메서드
- 객체를 생성하지 않고 클래스 변수를 접근할 때 활용
  - 참고: 클래스 변수를 액세스 할 필요가 없는 경우에는 정적 메서드를 활용

```
class Person:
    count = 0    # 클래스 속성의 변수

    def __init__(self):
        Person.count += 1

    @classmethod
    def print_count(cls):
        print('{0}명 등록되었습니다.'.format(cls.count))
```

```
class 클래스이름:
    @classmethod
    def 메서드(cls, 매개변수1, 매개변수2):
        코드
        cls = class
```

```
james = Person()
maria = Person()
Person.print_count()    # 2명 등록되었습니다.
Person.count = 4
Person.print_count()    # 4명 등록되었습니다.
```

출력 결과

2명 등록되었습니다.
4명 등록되었습니다.

# 클래스 메서드(class method)



실습 2: z\_class\_method.py

```
class Person:
    __count = 0    # 비공개 클래스 속성 변수

    def __init__(self):
        Person.__count += 1

    @classmethod
    def print_count(cls):    # cls.__count = Person.__count
        print('{0}명 등록되었습니다.'.format(cls.__count))
```

- 클래스 메서드와 비공개 클래스 변수의 활용 => **비공개 클래스 속성 변수(\_\_count)**를 사용하여 클래스 외부의 접근을 차단하기

정상수행 결과

2명 등록되었습니다.  
Person.\_\_count= 4  
2명 등록되었습니다.

```
james = Person()
maria = Person()
Person.print_count()    # 2명 등록되었습니다.
```

수행오류 발생

AttributeError: type object 'Person' has no attribute '\_\_count'

```
# 비공개 클래스 속성의 변수를 외부에서 액세스 할 수 없다.
Person.__count = 4    # 1) 새 변수를 선언한 것과 같음. 적용 안됨.
```

사실상 클래스내의 비공개 변수가 사용되는 것이 아니다.

```
# 1)의 문장의 수행 여부에 따라 2)번 문장의 수행여부가 달라진다.
```

```
# 1)을 수행하면 정상 수행. => 4를 출력한다.
```

```
# 1)을 주석 처리하면 오류 발생(type object 'Person' has no attribute '__count')
```

```
print('Person.__count=', Person.__count)    # 2) 클래스변수의 값을 출력
```

```
Person.print_count()    # 2명 등록되었습니다.
```

# 클래스 메서드(class method)



실습 3: z\_class\_method.py

```
81 class Person:
82     __count = 0    # 비공개 클래스 속성
83
84     def __init__(self):
85         Person.__count += 1
86         #print('Person.__count=', Person.__count)
87
88     @classmethod
89     def print_count(cls):    # cls로 클래스 속성에 접근
90         print('{0}명 등록되었습니다.'.format(cls.__count))
91         #print('{0}명 등록되었습니다.'.format(Person.__count))
92
93     @classmethod
94     def create(cls):
95         print('create method is called.')
96         p = cls()    # cls()로 인스턴스 생성
97         return p
98
99     def pr_count(self):
100         print('Person.__count=', Person.__count)
101
102
103 james = Person()    # __count = 1
104 maria = Person()    # __count = 2
105 jh = Person.create()    # instance jh 생성
106 Person.print_count()    # 3명 등록되었습니다.
107 jh.pr_count()          # Person.__count= 3
```

인스턴스를 생성해주는 함수 cls()  
# class method에서 호출하면 인스턴스를  
생성해 반환해 줄 수 있다.

create method is called.  
3명 등록되었습니다.  
Person.\_\_count= 3



# 클래스 메서드 활용 사례

A\_125\_oop\_objvar.py

- 현재 코딩은 같은 이름(name)의 객체를 생성해도 로봇의 수를 1 증가시킨다. 또한, 다른 이름을 가진 객체의 반환 값이 같아도 로봇의 수를 1 증가시킨다.
- `droid1 = Robot("R2-D2"); droid1 = Robot("R2-D2")`  
=> 이렇게 하면 로봇이 2대가 되는 문제가 있다. 이것을 해결해 보자.
- `droid1 = Robot("R2-D2"); droid2 = Robot("R2-D2")`  
=> 이렇게 해서 로봇이 2대가 되는 것은 괜찮다. R2-D2가 2대로 출력되게 하면 된다.
- 이를 다음 방식으로 개선하는 코드를 작성해 보자.
- 기능 1) 클래스 변수를 전체 로봇의 생성 수량을 파악하는 `population` 외에 생성하는 로봇의 모델을 클래스 변수에 기록하게 한다. 객체를 생성할 때마다 같은 모델의 수량을 클래스 변수에 기록한다. 메인 루틴에서 여러 종류의 로봇을 생성할 때마다 기종과 해당 기종의 수량 및 총 수량을 화면에 출력하게 한다.
- 기능 2) 로봇을 폐기시킬 때 해당 기종의 모델의 수를 클래스 변수에서 삭제할 수 있는 방안을 수립한다. (힌트: special method `__del__`) 메인 루틴에서 여러 종류의 로봇을 생성, 폐기하면서 기종과 수량이 올바르게 관리되는지 점검한다.

# 스태틱 메서드(static method)

z\_static\_method.py

- 클래스 이름으로 호출하는 메서드 중 객체 데이터를 액세스할 필요가 없는 사용하는 메서드
  - 클래스 메서드: 클래스 변수를 액세스할 필요가 있는 경우
  - 정적 메서드: 클래스 변수를 액세스 할 필요가 없는 경우

```
class Calc:  
    @staticmethod  
    def add(a, b):  
        print(a + b)  
  
    @staticmethod  
    def mul(a, b):  
        print(a * b)
```

```
class 클래스이름:  
    @staticmethod  
    def 메서드(매개변수1, 매개변수2):  
        코드
```

정적 메서드는 매개변수에 self를 지정하지 않는다.

```
Calc.add(10, 20) # 클래스에서 바로 메서드 호출  
Calc.mul(10, 20) # 클래스에서 바로 메서드 호출
```

출력결과

30 200
-----------

# 스태틱 메서드(static method)

- 정적 메서드는 self 파라미터를 갖지 않아서 인스턴스 변수에 액세스할 수 없다.
- 메서드 선언 위에 @staticmethod 라는 Decorator를 표시하여 정적 메서드임을 표시한다.
- 일반적으로 인스턴스 데이터를 액세스 할 필요가 없는 경우 클래스 메서드나 정적 메서드를 사용한다.
  - 이때 클래스 변수를 액세스할 필요가 있을 때는 클래스 메서드를, 이를 액세스할 필요가 없을 때는 정적 메서드를 사용한다.

```
@staticmethod          # static method
def isSquare(rectWidth, rectHeight):
    return rectWidth == rectHeight
```

```
if Rectangle.isSquare(5, 5):    # static method 호출
    print("Rect is square.")    # True
```

출력결과 Rect is square.

## 02. 파이썬의 객체 지향 프로그래밍

\* variable= {class variable, instance variable}

\* attribute(속성)={variable, method}

### ■ 클래스 구현하기 : 속성의 선언 instance variable의 선언

- 속성에 대한 정보를 선언하기 위해서는 `__init__()`이라는 예약 함수(special method)를 사용한다. 이 함수는 객체를 선언할 때 자동으로 한번 수행되는 constructor의 역할을 수행한다.

```
class SoccerPlayer(object):  
    def __init__(self, name, position, back_number):  
        self.name = name  
        self.position = position  
        self.back_number = back_number
```

`__init__`에서 선언한 변수는 instance 변수 속성을 갖는다. 각 객체마다 다른 값을 가진다.

- ➔ `__init__()` 함수는 이 class에서 사용할 변수를 정의하는 함수이다. `__init__()` 함수의 첫 번째 매개변수는 반드시 `self` 변수를 사용해야 한다. `self` 변수는 클래스에서 생성된 인스턴스에 접근하는 예약어이다.
- ➔ `self` 뒤의 매개변수들은 실제로 클래스가 가진 속성으로, 축구 선수의 이름, 포지션, 등 번호 등이다. 이 값들은 실제 생성된 인스턴스에 할당된다. 할당되는 코드는 `self.name = name` 이다.

## 02. 파이썬의 객체 지향 프로그래밍

### ■ 클래스 구현하기 : 함수의 선언

- 함수는 이 클래스가 의미하는 어떤 객체가 하는 다양한 동작을 정의할 수 있다. 만약 축구 선수라면, 등 번호 교체라는 행동을 할 수 있고, 이를 다음과 같은 코드로 표현할 수 있다.

```
class SoccerPlayer(object):  
    def change_back_number(self, new_number):  
        print("선수의 등번호를 변경한다: From %d to %d" % (self.back_number, new_number))  
        self.back_number = new_number
```

- 클래스 내에서의 함수도 기존 함수와 크게 다르지 않다. 함수의 이름을 쓰고 매개변수를 사용하면 된다. 여기서 가장 큰 차이점은 바로 `self`를 매개변수에 반드시 넣어야 한다는 것이다. `self`가 있어야만 실제로 인스턴스가 사용할 수 있는 함수로 선언된다.

## 02. 파이썬의 객체 지향 프로그래밍

## ■ 클래스 구현하기 : underscore \_의 쓰임

- 일반적으로 파이썬에서 \_의 쓰임은 개수에 따라 여러 가지로 나눌 수 있다. 예를 들어, \_ 1개는 이후로 쓰이지 않을 변수에 특별한 이름을 부여하고 싶지 않을 때 사용한다

**코드 10-1** underscore.py

```
1 for _ in range(10):
2     print("Hello, World")
```

[illegible]

## 02. 파이썬의 객체 지향 프로그래밍

### ■ 클래스 구현하기 : \_의 쓰임

- ➡ 위 코드는 'Hello, World'를 화면에 10번 출력하는 함수이다. 횟수를 세는 `_` 변수는 특별한 용도가 없으므로 뒤에서 사용되지 않는다. 따라서 `_`를 임의의 변수명 대신에 사용한다.
- 다른 용도로는 `_` 2개를 사용하여 특수한 예약 함수나 변수에 사용하기도 한다. 대표적으로 `__str__`이나 `__init__()` 같은 함수이다. `__str__()` 함수는 클래스로 인스턴스를 생성했을 때, 그 인스턴스 자체를 `print()` 함수로 화면에 출력하면 나오는 값을 뜻한다. 다양한 용도가 있으니 `_`의 특수한 용도에 대해서는 인지해 두는 것이 좋다

## 02. 파이썬의 객체 지향 프로그래밍

### ■ 인스턴스 생성하기 사용하기

- 클래스를 이용해 인스턴스를 호출 선언(혹은 생성)하는 방법은 아래 그림과 같다. 먼저 클래스 이름을 사용하여 호출하고, 앞서 만든 `__init__()` 함수의 매개변수에 맞추어 값을 입력한다. 여기에서는 함수에서 배운 초깃값 지정 등도 사용할 수 있다. 여기서 `self` 변수는 아무런 값도 할당되지 않는다.
- `jinhyun`이라는 인스턴스가 기존 `SoccerPlayer`의 클래스를 기반으로 생성되는 것을 확인할 수 있다. 이 `jinhyun`이라는 인스턴스 자체가 `SoccerPlayer` 클래스에서 `self`에 할당된다.
- 아래 할당문으로 한 객체가 새로 예시화(instantiation)되었다.

`jinhyun` = `SoccerPlayer`(`"Jinhyun"`, `"MF"`, `10`):

객체명

클래스 이름

`__init__` 함수 Interface, 초깃값

```
def __init__(self, name, position, back_number);
```

[파이썬에서 자주 사용하는 작명 기법]



## 02. 파이썬의 객체 지향 프로그래밍

실습1: instance.py

### ■ 인스턴스 생성하기 사용하기

현재 선수의 등번호는: 10

선수의 등번호를 변경한다: From 10 to 5

현재 선수의 등번호는: 5

Hello, My name is Jinhyun. I play in MF in center.

```
9 class SoccerPlayer(object):
10     def __init__(self, name, position, back_number):
11         self.name = name
12         self.position = position
13         self.back_number = back_number
14
15     def change_back_number(self, new_number):
16         print("선수의 등번호를 변경한다: From %d to %d" % (self.back_number, new_number))
17         self.back_number = new_number
18
19     def __str__(self): # 인스턴스 자체를 출력 할 때의 형식을 지정해주는 함수
20         return "Hello, My name is %. I play in %s in center." % (self.name, self.position)
21
22 # SoccerPlayer를 클래스를 이용하여 instance(객체)를 생성한다.
23 jinhyun = SoccerPlayer("Jinhyun", "MF", 10)
24
25 print("현재 선수의 등번호는:", jinhyun.back_number)
26 jinhyun.change_back_number(5) # 객체의 메소드 호출.
27 print("현재 선수의 등번호는:", jinhyun.back_number)
28 print(jinhyun)
```

## 02. 파이썬의 객체 지향 프로그래밍

실습2: instance.py

```
40 class SoccerPlayer(object):
41     def __init__(self, name='You', position='out', back_number=30): 1) default keyword 사용
42         self.name = name
43         self.position = position
44         self.back_number = back_number
45
46     def change_back_number(self, new_number):
47         print("선수의 등번호를 변경한다: From %d to %d" % (self.back_number, new_number))
48         self.back_number = new_number
49
50     def change_back_number2(self, new_number):
51         print("선수의 등번호를 변경한다: From %d to %d" % (self.back_number, new_number))
52         self.back_number = new_number
53         return self.back_number # 2) 반환 값이 있는 함수로 변경하였다.
54
55     2) 반환 값이 있는 함수 추가
56
57 # SoccerPlayer를 클래스를 이용하여 instance(객체)를 생성한다.
58 you = SoccerPlayer(back_number=8) # 1) 입력 파라미터 일부 미제공
59 print(you.name, you.position, you.back_number)
60
61 jinhyun = SoccerPlayer("Jinhyun", "MF")
62 print("현재 선수의 등번호는:", jinhyun.back_number)
63
64 print(jinhyun.change_back_number(11)) # None. 반환 값이 없는 메서드를 출력하였다.
65 print(jinhyun.change_back_number2(12)) # 12. 반환 값이 있다.
66 print("현재 선수의 등번호는:", jinhyun.back_number)
```

You out 8  
현재 선수의 등번호는: 30  
선수의 등번호를 변경한다: From 30 to 11  
None  
선수의 등번호를 변경한다: From 11 to 12  
12  
현재 선수의 등번호는: 12

## 02. 파이썬의 객체 지향 프로그래밍

설명 생략

### 여기서 잠깐! Hydrogen 패키지

- Atom에서 [코드 10-2]의 실행 결과를 확인하려면 Hydrogen 패키지를 사용하면 된다. Hydrogen 패키지를 사용하면 코드를 작성한 후 결과를 확인하기 위해 일일이 cmd 창으로 가지 않아도 되어 편리하고, 출력 결과가 어떤 코드로 인해 발생한 것인지 직접 확인할 수 있어 매우 유용하다.

[Atom Editor](#)

[Atom - hydrogen 설치 및 사용법](#)

Atom의 package인 hydrogen은 atom 내부에서 Jupyter kernel을 인식하여 python 코드 실행을 가능케 한다.

## 02. 파이썬의 객체 지향 프로그래밍

### ■ 클래스를 사용하는 이유

- 자신이 만든 코드가 데이터 저장뿐 아니라 데이터를 변환하거나 데이터베이스에 저장하는 등의 역할이 필요할 때가 있다. 이것을 리스트와 함수로 각각 만들어 공유하는 것보다 하나의 객체로 생성해 다른 사람들에게 배포한 다면 훨씬 더 손쉽게 사용할 수 있을 것이다.
- 또한, 코드를 좀 더 손쉽게 선언할 수 있다는 장점도 있다.

## 02. 파이썬의 객체 지향 프로그래밍

z\_class\_example.py

```
1  names = ["Messi", "Ramos", "Kim"]
2  positions = ["MF", "DF", "CF"]; numbers = [10, 4, 7]
3  players = [[name, position, number] # 이차원 리스트
4             for name, position, number in zip(names, positions, numbers)]
5  print(players) # [['Messi', 'MF', 10], ['Ramos', 'DF', 4], ['Kim', 'CF', 7]]
6  print(players[0]) # ['Messi', 'MF', 10]
7  class SoccerPlayer(object):
8      def __init__(self, name, position, numbers):
9          self.name = name
10         self.position = position
11         self.b_num = numbers # back number
12     def change_b_num(self, new_number):
13         local_var = self.b_num # 로컬 변수.
14         self.b_num = new_number
15         return f'Back number of {self.name} is changed from {local_var} to {self.b_num}.'
16     def __str__(self):
17         return "%s: I play in %s with back number %d." % (self.name, self.position, self.b_num)
18
19  p_obj = [SoccerPlayer(name, position, number)
20           for name, position, number in zip(names, positions, numbers)]
21  print(p_obj[0]) # Messi: I play in MF with back number 10.
22  print(p_obj[1]) # Ramos: I play in DF with back number 4.
23
24  print(p_obj[0].b_num) # 1) messi의 등번호를 출력한다. # 10
25  print(p_obj[0].change_b_num(13)) # 2) 현재 messi의 등번호를 13으로 바꾼다.
26  # Back number of Messi is changed from 10 to 13.
27  # 3) Ramos의 position을 WF로 바꾸고 그 결과를 출력한다.
28  p_obj[1].position = 'WF'; print(p_obj[1].position) # WF
```

# 정리: 클래스/객체

- class 정리 - 클래스 기본적인 사용
- 클래스와 메서드 만들기
- 객체는 내부의 일반적 변수를 사용하여 데이터를 저장할 수 있다.
- 객체 혹은 클래스에 소속된 변수는 필드(field)라고 부른다.
  - 인스턴스 변수: 객체에 내장된 변수
  - 클래스 변수: 클래스 자체에 내장된 변수
  - 비공개 속성(private attribute) : \_\_(2개)로 시작하는 변수 혹은 메소드. 클래스 바깥에서는 접근할 수 없음.
    - 클래스 바깥에서 접근할 수 있는 속성을 공개 속성(public attribute)이라 부르고, 클래스 안에서만 접근할 수 있는 속성을 비공개 속성(private attribute)이라 부릅니다.
- 객체는 안에 함수를 내장할 수 있는데 클래스의 이를 메소드(method)라 한다.
- 필드와 메소드를 통틀어 그 클래스의 속성이라 한다.

03

Lab: 노트북 프로그램 만들기

## 03. Lab: 노트북 프로그램 만들기

### ■ 노트북 프로그램 만들기 설계

- 이번 Lab에서는 지금까지 배운 객체 지향 프로그래밍의 개념을 이용하여 실제 구현 가능한 노트북(Notebook) 프로그램을 만들고자 한다.

- 노트(note)를 정리하는 프로그램이다.
- 사용자는 노트에 콘텐츠를 적을 수 있다.
- 노트는 노트북(notebook)에 삽입된다.
- 노트북은 타이틀(title)이 있다.
- 노트북은 노트가 삽입될 때 페이지를 생성하며, 최대 300페이지까지 저장할 수 있다.
- 300페이지를 넘기면 노트를 더는 삽입하지 못한다.



## 03. Lab: 노트북 프로그램 만들기

### ■ 노트북 프로그램 만들기 설계

구분	Notebook	Note
메서드	add_note remove_note get_number_of_pages	write_content remove_all
변수	title page_number notes	contents

[노트북 프로그램의 객체 설계]

## 03. Lab: 노트북 프로그램 만들기

### ■ 노트북 프로그램 구현 : Note 클래스

```
class Note(object):
    def __init__(self, contents = None):
        self.contents = contents

    def write_contents(self, contents):
        self.contents = contents

    def remove_all(self):
        self.contents = ""

    def __str__(self):
        return self.contents
```

## 03. Lab: 노트북 프로그램 만들기

### ■ 노트북 프로그램 구현 : Note 클래스

➔ Note 클래스에서 가장 중요한 변수는 contents, 즉 내용을 적는 변수이다. 이를 위해 Note 객체를 생성할 때 contents의 내용을 넣을 수 있도록 `__init__()` 함수 안에 `self.contents`를 초기값으로 만든다. Note는 Notebook에 있는 여러 장의 노트 중 한 장이라고 생각 할 수 있다. 당연히 새로운 Note를 만들고 아무런 내용도 넣지 않을 수 있으므로, 초기값을 `contents = None`으로 한다.

다음으로 Note에 새로운 내용을 쓰는 `write_contents()`와 Note의 모든 내용을 지우는 `remove_all()` 함수를 만든다. 각각은 문자열형을 입력받은 후, `self.contents` 변수에 할당하거나 내용을 삭제하는 `self.contents = ""`를 호출한다.

마지막으로 `print()` 함수를 유용하게 사용하기 위해 `__str__`을 선언하여 contents를 반환한다.

## 03. Lab: 노트북 프로그램 만들기

### ■ 노트북 프로그램 구현 : Notebook 클래스

```
class NoteBook(object):
    def __init__(self, title):
        self.title = title
        self.page_number = 1
        self.notes = {}

    def add_note(self, note, page = 0):
        if self.page_number < 300:
            if page == 0:
                self.notes[self.page_number] = note
                self.page_number += 1
            else:
                self.notes = {page : note}
                self.page_number += 1
        else:
            print("페이지가 모두 채워졌다.")
```

## 03. Lab: 노트북 프로그램 만들기

### ■ 노트북 프로그램 구현 : Notebook 클래스

```
def remove_note(self, page_number):  
    if page_number in self.notes.keys():  
        return self.notes.pop(page_number)  
    else:  
        print("해당 페이지는 존재하지 않는다.")  
  
def get_number_of_pages(self):  
    return len(self.notes.keys())
```

➡ Notebook 클래스에는 다음 세 가지가 필요하다.

- ① 타이틀(title): Notebook의 제목이 필요하다.
- ② 페이지 수(page\_number): 현재 Notebook에 총 몇 장의 노트가 있는지 기록하는 page\_number가 필요하며, 1페이지부터 시작한다.
- ③ 저장 공간: 노트 자체를 저장하기 위한 공간이 필요하다.

## 03. Lab: 노트북 프로그램 만들기

### ■ 노트북 프로그램 구현 : Notebook 클래스

- ➡ 이 세 가지 정보를 저장하기 위해 `__init__()` 함수 안에 정보를 모두 입력하였다. 여기서 관심을 두어야 할 변수는 `self.notes`이다. 위 코드에서는 `notes` 변수를 딕셔너리형으로 선언하였다. 이는 클래스를 설계하는 사람이 자신의 기호에 맞게, 또는 개발 목적에 따라 적절히 지정할 수 있다. 여기서 `notes` 변수 안에는 `Note`의 인스턴스가 값value으로 들어가게 하고, 키로 각 `Note`의 `page_number`를 사용할 예정이다. 이는 `page_number`로 `Note`를 쉽게 찾기 위해서다.
- ➡ 다음으로는 함수 부분이다. 먼저 `add_note()` 함수는 새로운 `Note`를 `Notebook`에 삽입하는 함수이다. 몇 가지 요구 조건에 대한 로직이 들어간다. 예를 들어, `page_number`가 300이하이면 새로운 `Note`를 계속 추가할 수 있지만, 그 이상일 때는 `Note`를 더 추가하지 못하도록 한다. 새로운 `Note`가 들어갈 때는 임의의 페이지 번호를 넣을 수 있다. 하지만 사용자가 입력하지 않을 때는 맨 마지막에 입력된 `Note` 다음 장에 `Note`를 추가한다. 맨 마지막 페이지는 늘 `page_number`에 저장되어 있다.

## 03. Lab: 노트북 프로그램 만들기

### ■ 노트북 프로그램 구현 : Notebook 클래스

- ➡ 두 번째 함수는 `remove_note()` 함수이다. `remove_note()` 함수는 특정 페이지 번호에 있는 Note를 제거하는 함수이다. 앞에서 Note가 저장된 객체를 딕셔너리형으로 저장하고 페이지 번호를 키로 저장했으므로, 딕셔너리형인 `self.notes`에 해당 페이지 번호의 키가 있는지 확인하면 된다. 이는 `page_number in self.notes.keys()`로 쉽게 확인할 수 있다. 만약 페이지가 있다면 해당 페이지를 삭제하면서 반환하고, 없다면 `print()` 함수를 사용하여 없다고 사용자에게 알려 줄 수 있다.
- ➡ `page_number`를 넣고 그 페이지가 기존의 노트에 있다면 해당 페이지를 팝하여 돌려주고, 없다면 '해당 페이지는 존재하지 않는다.'라고 출력한다

## 03. Lab: 노트북 프로그램 만들기

### ■ 노트북 프로그램 사용

- 앞에서 구현한 Note 클래스와 Notebook 클래스를 활용하여 실제 프로그램을 작성해 보자. 소스 파일에서 제공하는 'notebook.py'와 'notebook\_client.py' 파일을 활용한다.
- 먼저 해당 클래스를 불러 사용할 수 있는 클라이언트 코드를 만들어 보자. 지금부터 작성하는 코드는 'notebook\_client.py'에 있는 코드이다. 먼저 2개의 클래스를 호출해야 하는데, 호출하는 코드는 내장 모듈을 사용한 것처럼 from과 import를 사용한다. 'from 파일명 import 클래스명'의 형태로 생각하면 된다. 이 코드가 실행되면 두 클래스는 메모리에 로딩된다

```
from notebook import Note
from notebook import Notebook
```



## 03. Lab: 노트북 프로그램 만들기

### ■ 노트북 프로그램 사용

- 다음으로 몇 장의 Note를 생성한다. 넣고 싶은 Note의 내용과 함께 문자열형을 사용하여 생성자로 만들면 된다.

```
good_sentence = """"세상 사는 데 도움이 되는 명언, 힘이 되는 명언, 용기를 주는 명언, 위로되는 명언, 좋은 명언 모음 100가지. 자주 보면 좋을 것 같아 선별했습니다."""
```

```
note_1 = Note(good_sentence)
```

```
good_sentence = """"삶이 있는 한 희망은 있다. - 키케로 """"
```

```
note_2 = Note(good_sentence)
```

```
good_sentence = """"하루에 3시간을 걸으면 7년 후에 지구를 한 바퀴 돌 수 있다. - 새뮤얼 존슨""""
```

```
note_3 = Note(good_sentence)
```

```
good_sentence = """"행복의 문이 하나 닫히면 다른 문이 열린다. 그러나 우리는 종종 닫힌 문을 멍하니 바라보다가 우리를 향해 열린 문을 보지 못하게 된다. - 헬렌 켈러""""
```

```
note_4 = Note(good_sentence)
```

## 03. Lab: 노트북 프로그램 만들기

### ■ 노트북 프로그램 사용

- 모두 4개의 Note이다. 이 Note를 파이썬 셸에서 확인하기 위해 다음과 같이 코드를 입력하면, Note의 인스턴스 생성을 확인할 수 있다. Note를 생성한 후 `print()` 함수를 사용하면 해당 Note에 있는 텍스트를 확인할 수 있다. 또한, `remove()` 함수도 사용할 수 있다.

```
>>> from notebook import Note
>>> from notebook import Notebook
>>> good_sentence = """세상 사는 데 도움이 되는 명언, 힘이 되는 명언, 용기를 주는 명언, 위로
>>> 되는 명언, 좋은 명언 모음 100가지. 자주 보면 좋을 것 같아 선별했습니다."""
>>> note_1 = Note(good_sentence)
>>>
>>> note_1
<notebook.Note object at 0x0000022278C06DD8>
>>> print(note_1)
세상 사는 데 도움이 되는 명언, 힘이 되는 명언, 용기를 주는 명언, 위로되는 명언, 좋은 명언 모음
100가지. 자주 보면 좋을 것 같아 선별했습니다.
>>> note_1.remove()
>>> print(note_1)
삭제된 노트입니다.
```

## 03. Lab: 노트북 프로그램 만들기

### ■ 노트북 프로그램 사용

- 다음은 새로운 Notebook을 생성하는 코드이다. 새로운 노트를 생성한 후 기존의 Note들을 add\_note( ) 함수로 추가하였다.

```
wise_saying_notebook = Notebook("명언 노트")
wise_saying_notebook.add_note(note_1)
wise_saying_notebook.add_note(note_2)
wise_saying_notebook.add_note(note_3)
wise_saying_notebook.add_note(note_4)
```

## 03. Lab: 노트북 프로그램 만들기

### ■ 노트북 프로그램 사용

- 다음 코드처럼 실제 추가된 것을 확인할 수 있다. Note 4장을 추가하였으므로 `get_number_of_all_pages()`를 사용하면 총 페이지 수가 출력되고, `get_number_of_all_characters()`로 총 글자 수를 확인할 수 있다.

```
>>> wise_saying_notebook = Notebook("명언 노트")
>>> wise_saying_notebook.add_note(note_1)
>>> wise_saying_notebook.add_note(note_2)
>>> wise_saying_notebook.add_note(note_3)
>>> wise_saying_notebook.add_note(note_4)
>>> print(wise_saying_notebook.get_number_of_all_pages())
4
>>> print(wise_saying_notebook.get_number_of_all_characters())
159
```

## 03. Lab: 노트북 프로그램 만들기

### ■ 노트북 프로그램 사용

- 또한, 노트의 삭제나 추가도 여러 가지 명령어로 가능하다. 다음과 같이 특정 Note를 지우는 `remove_note()`를 사용할 수 있고, 객체 지향 프로그래밍을 사용하여 새로운 빈 노트를 임의로 추가할 수도 있다. 기존 페이지에 노트를 추가하려고 하면 오류 메시지도 출력된다

```
>>> wise_saying_notebook.remove_note(3)
>>> print(wise_saying_notebook.get_number_of_all_pages())
3
>>>
>>> wise_saying_notebook.add_note(note_1, 100)
>>> wise_saying_notebook.add_note(note_1, 100)
해당 페이지에는 이미 노트가 존재합니다.
>>>
>>> for i in range(300):
...     wise_saying_notebook.add_note(note_1, i)
...
...
```

## 03. Lab: 노트북 프로그램 만들기

### ■ 노트북 프로그램 사용

해당 페이지에는 이미 노트가 존재합니다.

해당 페이지에는 이미 노트가 존재합니다.

해당 페이지에는 이미 노트가 존재합니다.

해당 페이지에는 이미 노트가 존재합니다.

```
>>> print(wise_saying_notebook.get_number_of_all_pages())
```

```
300
```

# 04

## 객체 지향 프로그래밍의 특징

## 04. 객체 지향 프로그래밍의 특징

### ■ 상속

- 상속(inheritance) 은 이름 그대로 무엇인가를 내려받는 것을 뜻한다. 부모 클래스에 정의된 속성과 메서드를 자식 클래스가 물려받아 사용하는 것이다.

```
class Person(object):  
    pass
```

- ➡ class라는 예약어 다음에 클래스명으로 Person을 쓰고 object를 입력하였다. 여기서 object가 바로 Person 클래스의 부모 클래스이다. 사실 object는 파이썬에서 사용하는 가장 기본 객체(base object) 이며, 파이썬 언어가 객체 지향 프로그래밍이므로 모든 변수는 객체이다. 예를 들어, 파이썬의 문자열형은 다음과 같이 객체 이름을 확인할 수 있다.

```
>>> a = "abc"  
>>> type(a)  
<class 'str'>
```



## 04. 객체 지향 프로그래밍의 특징

### ■ 상속

```
>>> class Person(object):
...     def __init__(self, name, age):
...         self.name = name
...         self.age = age
...
>>> class Korean(Person):
...     pass
...
>>> first_korean = Korean("Sungchul", 35)
>>> print(first_korean.name)
Sungchul
```

## 04. 객체 지향 프로그래밍의 특징

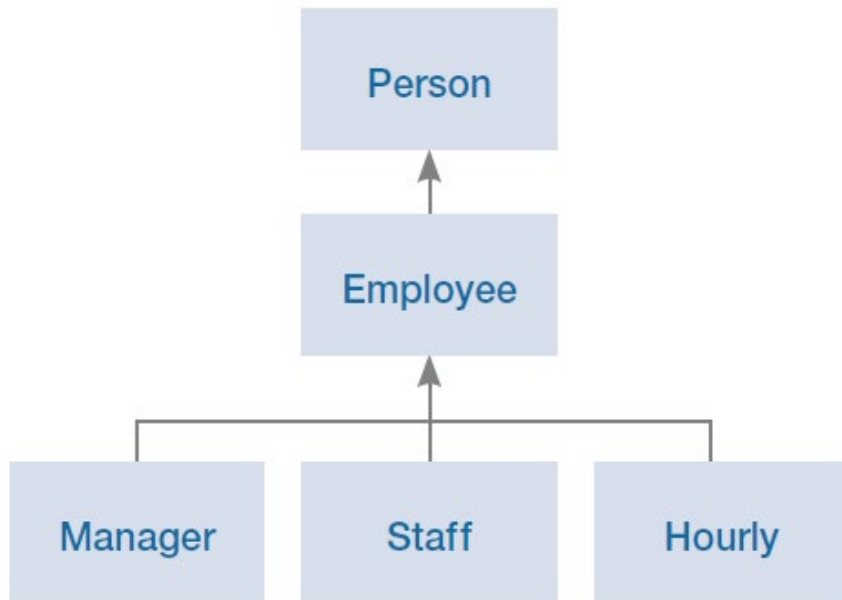
### ■ 상속

- ➔ 위 코드에서는 먼저 Person 클래스를 생성하였다. Person 클래스에는 생성자 `__init__()` 함수를 만들어 name과 age에 관련된 정보를 입력할 수 있도록 하였다. 다음으로 Korean 클래스를 만들면서 Person 클래스를 상속받게 한다. 상속은 `class Korean(Person)` 코드처럼 매우 간단하다. 그리고 별도의 구현 없이 pass로 클래스만 존재하게 만들고, Korean 클래스의 인스턴스를 생성해 준다. Korean 클래스는 별도의 생성자가 없지만, Person 클래스가 가진 생성자를 그대로 사용하여 인스턴스를 생성할 수 있다. 그리고 Person 클래스에서 생성할 수 있는 변수를 그대로 사용할 수 있다. 이러한 객체 지향 프로그래밍의 특징을 상속이라고 한다.

## 04. 객체 지향 프로그래밍의 특징

### ■ 상속

- 사각형이 클래스이고, 화살표는 각 클래스의 상속 관계이다. Person 클래스를 Employee가 상속하고, 이 클래스를 다시 한번 Manager, Staff, Hourly 등이 상속하는 것이다.



[상속 구조]

## 04. 객체 지향 프로그래밍의 특징

inheritance\_main.py

### ■ 상속

- 상속이 진행될수록 부모 클래스에 대해 각 클래스의 기능이 구체화되도록 부모 객체에는 일반적인 기능을, 자식 객체에는 상세한 기능을 넣어야 한다. 그리고 같은 일을 하는 메서드이지만 부모 객체보다 자식 객체가 좀 더 많은 정보를 줄 수도 있다. 이를 '부모 클래스의 메서드를 재정의한다'라고 한다.

코드 10-4 inheritance1.py

```
1 class Person(object):                                # 부모 클래스 Person 선언
2     def __init__(self, name, age, gender):
3         self.name = name
4         self.age = age
5         self.gender = gender
6
7     def about_me(self):                                # 메서드 선언
8         print("저의 이름은", self.name, "이고요, 제 나이는", str(self.age), "살입니다.")
```

### ■ 상속

- ➡ 먼저 부모 클래스 Person이다. name, age, gender에 대해 변수를 받을 수 있도록 선언하였고, about\_me 함수를 사용하여 생성된 인스턴스가 자신을 설명할 수 있도록 하였다. 사실 str( )에 들어가도 되는 클래스이지만 임의의 about\_me 클래스를 생성하였다.

## 04. 객체 지향 프로그래밍의 특징

inheritance\_main.py

### ■ 상속

- 다음으로 상속받는 Employee 클래스는 [코드 10-5]와 같다.

코드 10-5 inheritance2.py

Import 해 놓아야 외부 모듈에서 활용할 수 있다.

```
from inheritance1 import Person
1 class Employee(Person):                                # 부모 클래스 Person으로부터 상속
2     def __init__(self, name, age, gender, salary, hire_date):
3         super().__init__(name, age, gender)             # 부모 객체 사용
4         self.salary = salary
5         self.hire_date = hire_date                       # 속성값 추가
6
7     def do_work(self):                                   # 새로운 메서드 추가
8         print("열심히 일을 한다.")
9
10    def about_me(self):                                  # 부모 클래스 함수 재정의
11        super().about_me()                               # 부모 클래스 함수 사용
12        print("제 급여는", self.salary, "원이고, 제 입사일은", self.hire_date, "입니다.")
```

### ■ 상속

- ➔ Person 클래스가 단순히 사람에 대한 정보를 정의했다면, Employee 클래스는 사람에 대한 정의와 함께 일하는 시간과 월급에 대한 변수를 추가한다. 즉 `__init__()` 함수를 재정의한다. 이때 부모 클래스의 `__init__()` 함수를 그대로 사용하려면 별도의 `__init__()` 함수를 만들지 않아도 된다. 하지만 기존 함수를 사용하면서 새로운 내용을 추가하기 위해서는 자식 클래스에 `__init__()` 함수를 생성하면서 `super().__init__(매개변수)`를 사용해야 한다. 여기서 `super()`는 부모 클래스를 가리킨다. 즉, 부모 클래스의 `__init__()` 함수를 그대로 사용한다는 뜻이다.
- ➔ 그 아래에는 필요한 자식 클래스의 새로운 변수를 추가하면 된다. 이러한 함수의 재정의를 오버라이딩(overriding)이라고 한다. 오버라이딩은 상속 시 함수 이름과 필요한 매개변수는 그대로 유지하면서 함수의 수행 코드를 변경하는 것이다. 같은 방식으로 `about_me()` 함수가 오버라이딩된 것을 확인할 수 있다. Person과 Employee에 대한 설명을 추가한 것이다.

# 외부 모듈에 있는 클래스의 활용

실습 1: inheritance\_main.py

```
2  # -----
3  # 실습 1: 외부 모듈의 class를 import 하여 활용한다.
4  # -----
5  from inheritance1 import Person
6  #from inheritance2 import Employee # 이것도 OK
7  from inheritance2 import *
8
9  taylor = Person('Taylor', 24, 'male')
10
11  e1 = Employee('Jane', 20, 'female', 300, '20190104')
12  e1.do_work()
13  e1.about_me()
```

열심히 일을 한다.

저의 이름은 Jane 이고요, 제 나이는 20 살입니다.

제 급여는 300 원이고, 제 입사일은 20190104 입니다.



## 04. 객체 지향 프로그래밍의 특징

### ■ 다형성

- 다형성(polymorphism)은 같은 이름의 메서드가 다른 기능을 할 수 있도록 하는 것을 말한다.

코드 10-6 polymorphism1.py

```
1 n_crawler = NaverCrawler()
2 d_crawler = DaumCrawler()
3 cralwers = [n_crawler, d_crawler]
4 news = []
5 for cralwer in cralwers:
6     news.append(cralwer.do_crawling())
```

수행 안 되는 의사 코드임!!

- ➡ [코드 10-6]을 보면 두 Crawler 클래스가 같은 do\_crawling() 함수를 가지고 이에 대한 역할이 같으므로 news 변수에 결과를 저장하는 데 문제가 없다. [코드 10-6]은 의사 코드(pseudo code)로, **실제 실행되는 코드는 아니지만** 작동의 예시를 보기에는 적당하다. 이렇게 클래스의 다형성을 사용하여 다양한 프로그램을 작성할 수 있다.

## 04. 객체 지향 프로그래밍의 특징

polymorphism2.py

### ■ 다형성

Method 'talk'가 어떤 클래스의 객체를 사용하는가에 따라 다른 역할을 수행한다.

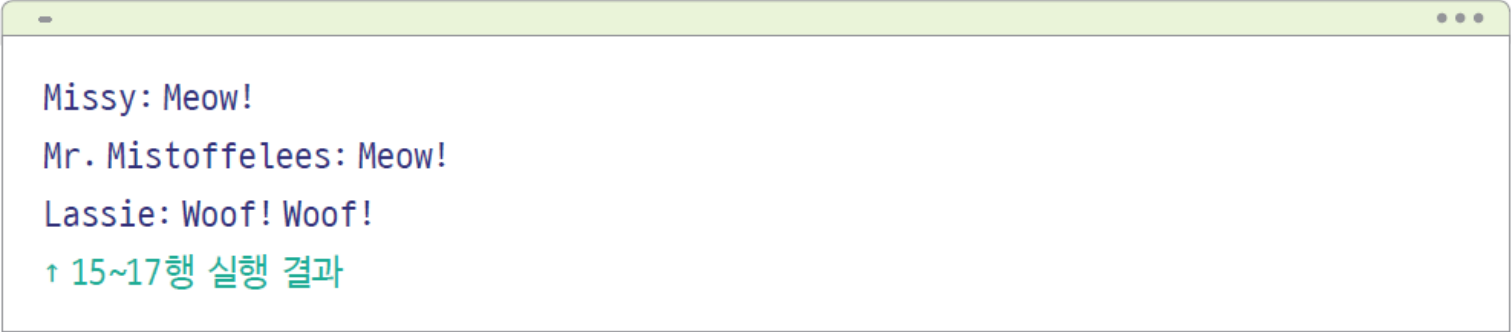
코드 10-7 polymorphism2.py

```
1 class Animal:
2     def __init__(self, name):
3         self.name = name
4     def talk(self):
5         raise NotImplementedError("Subclass must implement abstract method")
6
7 class Cat(Animal):
8     def talk(self):
9         return 'Meow!'
10
11 class Dog(Animal):
12     def talk(self):
13         return 'Woof! Woof!'
14
15 animals = [Cat('Missy'), Cat('Mr. Mistoffelees'), Dog('Lassie')]
```

## 04. 객체 지향 프로그래밍의 특징

### ■ 다형성

```
16 for animal in animals:  
17     print(animal.name + ': ' + animal.talk())
```



```
Missy: Meow!  
Mr. Mistoffelees: Meow!  
Lassie: Woof! Woof!  
↑ 15~17행 실행 결과
```

- ➡ [코드 10-7]에서 부모 클래스는 Animal이며, Cat과 Dog는 Animal 클래스를 상속받는다. 핵심 함수는 talk로, 각각 두 동물 클래스의 역할이 다른 것을 확인할 수 있다. Animal 클래스는 NotImplementedError라는 클래스를 호출한다. 이 클래스는 자식 클래스에만 해당 함수를 사용할 수 있도록 한다. 따라서 두 클래스가 내부 로직에서 같은 이름의 함수를 사용하여 결과를 출력하도록 한다. 실제로는 15~17행과 같이 사용할 수 있다.

## 04. 객체 지향 프로그래밍의 특징

### ■ 가시성

- 가시성visibility 은 객체의 정보를 볼 수 있는 레벨을 조절하여 객체의 정보 접근을 숨기는 것을 말하며, 다양한 이름으로 불린다. 파이썬에서는 가시성이라고 하지만, 좀 더 중요한 핵심 개념은 캡슐화(encapsulation)와 정보 은닉(information hiding)이다.]
- 파이썬의 가시성 사용 방법에 대한 예시 코드를 작성해야 하는 상황은 다음과 같다.

- Product 객체를 Inventory 객체에 추가
- Inventory에는 오직 Product 객체만 들어감
- Inventory에 Product가 몇 개인지 확인이 필요
- Inventory에 Product items는 직접 접근이 불가

## 04. 객체 지향 프로그래밍의 특징

visibility1.py

### ■ 가시성

코드 10-8 visibility1.py

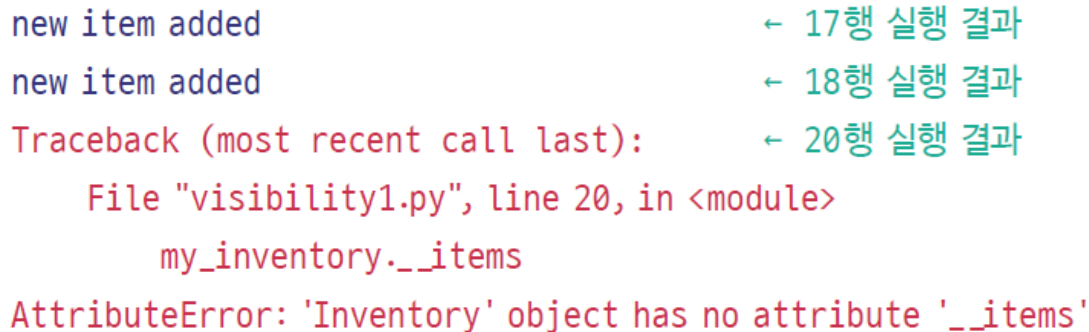
```
1 class Product(object):
2     pass
3
4 class Inventory(object):
5     def __init__(self):
6         self.__items = []
7     def add_new_item(self, product):
8         if type(product) == Product:
9             self.__items.append(product)
10            print("new item added")
11        else:
12            raise ValueError("Invalid Item")
13    def get_number_of_items(self):
14        return len(self.__items)
15
```

```
In[2]: a = 5
In[3]: type(a)
Out[3]: int
In[4]: type(a) == int
Out[4]: True
In[5]: type(type(a))
Out[5]: type
```

## 04. 객체 지향 프로그래밍의 특징

### ■ 가시성

```
16 my_inventory = Inventory()  
17 my_inventory.add_new_item(Product())  
18 my_inventory.add_new_item(Product())  
19  
20 my_inventory.__items
```



```
new item added  
new item added  
Traceback (most recent call last):  
  File "visibility1.py", line 20, in <module>  
    my_inventory.__items  
AttributeError: 'Inventory' object has no attribute '__items'
```

← 17행 실행 결과  
← 18행 실행 결과  
← 20행 실행 결과

## 04. 객체 지향 프로그래밍의 특징

### ■ 가시성

- ➡ [코드 10-8]에서는 Inventory 객체에 `add_new_item()` 함수를 사용하여 새롭게 생성된 Product 객체를 넣어 준다. `__items`는 Product 객체가 들어가는 공간으로, `get_number_of_items()`를 사용하여 총 객체의 개수를 반환한다.
- ➡ 여기서 핵심은 `__items` 변수이다. Inventory를 저장하는 공간으로, `add_new_item()`을 통해 Product 객체를 넣을 수 있다. 하지만 다른 프로그램이 `add_new_item()`이 아니라 직접 해당 객체에 접근해 새로운 값을 추가하려고 한다면 어떻게 할까? 다른 코드에서는 잘 실행되다가 20행의 `my_inventory.__items`에서 오류가 발생한다. 왜냐하면 `_`가 특수 역할을 하는 예약 문자로 클래스에서 변수로 두 개 붙어, 사용될 클래스 내부에서만 접근할 수 있고, 외부에는 호출하여 사용하지 못하기 때문이다. 즉, 클래스 내부용으로만 변수를 사용하고 싶다면 `'__변수명'` 형태로 변수를 선언한다. 가시성을 클래스 내로 한정하면서 값이 다르게 들어가는 것을 막을 수 있다. 이를 정보 은닉이라고 한다.
- ➡ 이러한 정보를 클래스 외부에서 사용하기 위해서는 어떻게 해야 할까? 데코레이터 (decorator)라고 불리는 `@property`를 사용한다.

### ■ 가시성

- [코드 10-8]의 14행 뒷부분에 [코드 10-9]를 추가하여 @property를 사용하면 해당 변수를 외부에서 사용할 수 있다

코드 10-9 visibility2.py

```
1 class Inventory(object):
2     def __init__(self):
3         self.__items = []           # private 변수로 선언(타인이 접근 못 함)
4
5     @property                       # property 데코레이터(숨겨진 변수 반환)
6     def items(self):
7         return self.__items
```

- ➡ 다른 코드는 그대로 유지하고 마지막에 items라는 이름으로 메서드를 만들면서 @property를 메서드 상단에 입력한다. 그리고 외부에서 사용할 변수인 \_\_items를 반환한다.



## 04. 객체 지향 프로그래밍의 특징

### ■ 가시성

- 코드를 추가하면 다음과 같이 외부에서도 해당 메서드를 사용할 수 있다.

```
>>> my_inventory = Inventory()  
>>> items = my_inventory.items  
>>> items.append(Product())
```

- ➡ 이번 코드에서는 오류가 발생하지 않았다. 여기서 주목할 부분은 `_items` 변수의 원래 이름이 아닌 `items`로 호출할 수 있다는 것이다. 바로 `@property`를 붙인 함수 이름으로 실제 `_items`를 사용할 수 있는 것이다. 이는 기존 `private` 변수를 누구나 사용할 수 있는 `public` 변수로 바꾸는 방법 중 하나이다.

# Thank You !