

Section 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.

Section 01 객체 지향 프로그래밍의 이해

■ 객체와 클래스

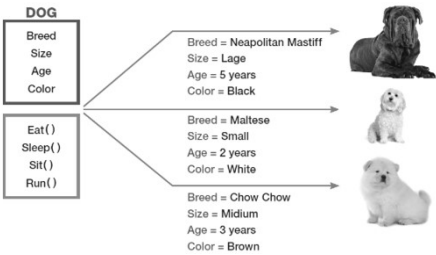
개념	설명	예시
객체(object)	실생활에 존재하는 실제적인 물건 또는 개념	심판, 선수, 팀
속성(attribute)	객체가 가지고 있는 변수	선수의 이름, 포지션, 소속팀
행동(action)	객체가 실제로 작동할 수 있는 함수, 메서드	공을 차다, 패스하다

[객체, 속성, 행동]

Section 01 객체 지향 프로그래밍의 이해

■ 객체와 클래스

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

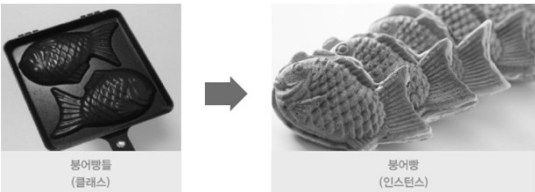


[Dog 인스턴스]

Section 01 객체 지향 프로그래밍의 이해

■ 객체와 클래스

- 잘 만든 붕어빵틀이 있다면 새로운 종류의 다양한 붕어빵을 만들 수 있다.
- 다시 말해, 잘 만든 클래스 코드가 있다면 이 코드로 다양한 종류의 인스턴스를 생성할 수 있다.



[클래스와 인스턴스의 관계]

Section 02 파이썬의 객체 지향 프로그래밍

클래스 구현하기

- 파이썬에서 클래스를 선언하기 위한 기본 코드 템플릿은 다음과 같다.

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



[파이썬에서의 클래스 선언]

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

Section02 클래스

클래스의 개념

- 클래스의 모양과 생성

```
class 클래스명 :  
    # 이 부분에 관련 코드 구현
```

- 현실 세계의 사물을 컴퓨터 안에서 구현하려고 고안된 개념

- 자동차를 클래스로 구현

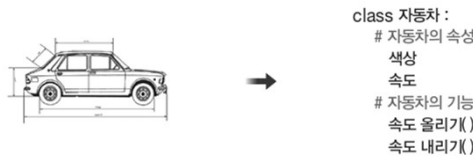


그림 12-1 자동차를 클래스로 구현

Section02 클래스

- Code12 - 01 . py 에 메서드 추가

```
def printMessage() :  
    print("시험 출력이다.")
```

- printMessage () 안에서는 필드를 사용하지 않으므로 이때는 self 생략이 가능

- 인스턴스의 생성(예 : 실제 생산되는 자동차)

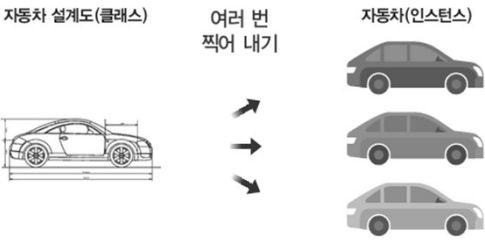


그림 12-2 클래스와 인스턴스의 개념

Section02 클래스

- 필드에 값 대입



그림 12-4 인스턴스의 필드에 값을 대입하는 개념

```
myCar1.color = "빨강"  
myCar1.speed = 0  
myCar2.color = "파랑"  
myCar2.speed = 0  
myCar3.color = "노랑"  
myCar3.speed = 0
```

Section 02 파이썬의 객체 지향 프로그래밍

파이썬에서 자주 사용하는 작명 기법

여기서 잠깐!

- 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, 파이썬 함수나 변수명에 사용됨
CamelCase	띄어쓰기 부분에 대문자를 사용하여 변수의 이름을 지정함, 닥터의 축처럼 생겼기하여 Camel 이라고 명명, 파이썬 클래스명에 사용됨

[파이썬에서 자주 사용하는 작명 기법]

Section 02 파이썬의 객체 지향 프로그래밍

클래스 구현하기 : 속성의 선언

- 속성에 대한 정보를 선언하기 위해서는 `__init__()`이라는 예약 함수를 사용한다.

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

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

Section 02 파이썬의 객체 지향 프로그래밍

클래스 구현하기 : 함수의 선언

- 함수는 이 클래스가 의미하는 어떤 객체가 하는 다양한 동작을 정의할 수 있다. 만약 축구 선수라면, 등번호 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.

Section 02 파이썬의 객체 지향 프로그래밍

클래스 구현하기 : `_`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")
```



Section 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

Section 02 파이썬의 객체 지향 프로그래밍

인스턴스 사용하기

- 클래스에서 인스턴스를 호출하는 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.
- jinhyun이라는 인스턴스가 기존 SoccerPlayer의 클래스를 기반으로 생성되는 것을 확인할 수 있다. 이 jinhyun이라는 인스턴스 자체가 SoccerPlayer 클래스에서 self에 할당된다.

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

객체명 클래스 이름 _init_ _ 함수 Interface, 초깃값
def _init_(self, name, position, back_number):

[파이썬에서 자주 사용하는 작명 기법]

Section 02 파이썬의 객체 지향 프로그래밍

인스턴스 사용하기

- 실제로 생성된 코드는 다음 [코드 10-2]와 같다.

코드 10-2 instance.py

```
1 # 전체 SoccerPlayer 코드
2 class SoccerPlayer(object):
3     def _init_(self, name, position, back_number):
4         self.name = name
5         self.position = position
6         self.back_number = back_number
7     def change_back_number(self, new_number):
8         print("선수의 등번호를 변경한다: From %d to %d" % (self.back_number, new_
9             number))
10         self.back_number = new_number
11     def _str_(self):
12         return "Hello, My name is %s. I play in %s in center." % (self.name,
13             self.position)
```

Section 02 파이썬의 객체 지향 프로그래밍

인스턴스 사용하기

```
13 # SoccerPlayer를 사용하는 instance 코드
14 jinhyun = SoccerPlayer("Jinhyun", "MF", 10)
15
16 print("현재 선수의 등번호는:", jinhyun.back_number)
17 jinhyun.change_back_number(5)
18 print("현재 선수의 등번호는:", jinhyun.back_number)
```



- ⇒ 14행에서 jinhyun = SoccerPlayer("Jinhyun", "MF", 10) 코드로 인스턴스를 새롭게 생성할 수 있다. 생성된 인스턴스인 jinhyun은 name, position, back_number에 각각 Jinhyun, MF, 10이 할당되었다. 해당 값을 생성된 인스턴스에서 사용하기 위해서는 jinhyun.back_number로 인스턴스 내의 값을 호출하여 사용할 수 있다.

Section 02 파이썬의 객체 지향 프로그래밍

인스턴스 사용하기

- 여기서 중요한 것은 인스턴스가 생성된 후에는 해당 인스턴스의 이름으로 값을 할당하거나 함수를 부르면 되지만, 클래스 내에서는 self로 호출된다. 즉, 생성된 인스턴스인 jinhyun과 클래스 내 self가 같은 역할을 하는 것이다.
- 함수를 호출할 때도 인스턴스의 이름과 함수명을 사용한다. 여기서는 17행에서 jinhyun.change_back_number(5)를 사용해 클래스 내의 함수를 사용하였다.

Section 02 파이썬의 객체 지향 프로그래밍

인스턴스 사용하기

- [코드 10-2]에 이어 19행에 print(jinhyun)을 입력하면 다음과 같은 결과가 출력된다.
- ```

Hello, My name is Jinhyun. I play in MF in center.
```
- 생성된 인스턴스인 jinhyun을 단순히 print() 함수로 썼을 때 나오는 결과이다. 이는 [코드 10-2]의 2 · 10 · 11행에 클래스 내 함수가 선언되었기 때문이다.
  - 9행에서 \_str\_ 함수로 선언된 부분이 print() 함수를 사용하면 반환되는 함수이다. 인스턴스의 정보를 표시하거나 구분할 때는 \_str\_ 문을 사용하면 된다. 이처럼 예약 함수는 특정 조건에서 작동하는 함수로 유용하다

Section 02 파이썬의 객체 지향 프로그래밍

Hydrogen 패키지

여기서 잠깐!

- 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.

Section 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 배포한다면 훨씬 더 손쉽게 사용할 수 있을 것이다.
- 또한, 코드를 좀 더 손쉽게 선언할 수 있다는 장점도 있다.

코드 10-3 class.py

```
1 # 데이터
2 names = ["Messi", "Ramos", "Ronaldo", "Park", "Buffon"]
3 positions = ["MF", "DF", "CF", "WF", "GK"]
4 numbers = [10, 4, 7, 13, 1]
5
6 # 이차원 리스트
7 players = [[name, position, number] for name, position, number in zip(names,
8 positions, numbers)]
9 print(players)
10 print(players[0])
11
```

Section 02 파이썬의 객체 지향 프로그래밍

클래스를 사용하는 이유

```
11 # 전체 SoccerPlayer 코드
12 class SoccerPlayer(object):
13 def __init__(self, name, position, back_number):
14 self.name = name
15 self.position = position
16 self.back_number = back_number
17 def change_back_number(self, new_number):
18 print("선수의 등번호를 변경한다: From %d to %d" % (self.back_number, new_
19 number))
19 self.back_number = new_number
20 def __str__(self):
21 return "Hello, My name is %. I play in %s in center." % (self.name,
22 self.position)
23 # 클래스-인스턴스
24 player_objects = [SoccerPlayer(name, position, number) for name, position,
25 number in zip(names, positions, numbers)]
26 print(player_objects[0])
```

Section 02 파이썬의 객체 지향 프로그래밍

클래스를 사용하는 이유

```
[['Messi', 'MF', 10], ['Ramos', 'DF', 4], ['Ronaldo', 'CF', 7], ['Park', 'WF', 13],
['Buffon', 'GK', 1]]
['Messi', 'MF', 10]
Hello, My name is Messi. I play in MF in center.
```

Section02 인스턴스 변수와 클래스 변수

인스턴스 변수의 개념

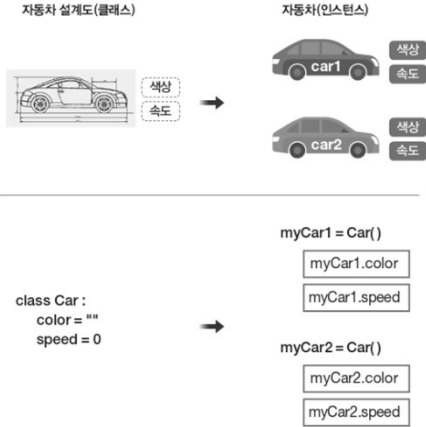


그림 12-5 인스턴스 변수의 개념

Section02 인스턴스 변수와 클래스 변수

클래스 변수

클래스 안에 공간이 할당된 변수, 여러 인스턴스가 클래스 변수 공간 함께 사용

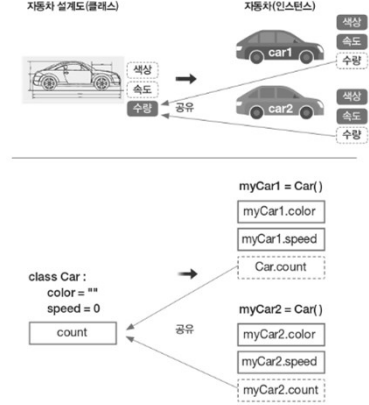


그림 12-6 클래스 변수의 개념

Section 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.

Section 03 Lab: 노트북 프로그램 만들기

■ 노트북 프로그램 만들기 설계

| 구분  | Notebook                                       | Note                        |
|-----|------------------------------------------------|-----------------------------|
| 메서드 | add_note<br>remove_note<br>get_number_of_pages | write_content<br>remove_all |
| 변수  | title<br>page_number<br>notes                  | contents                    |

[노트북 프로그램의 객체 설계]

Section 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
```

Section 03 Lab: 노트북 프로그램 만들기

■ 노트북 프로그램 구현 : Note 클래스

⇒ Note 클래스에서 가장 중요한 변수는 contents, 즉 내용을 적는 변수이다. 이를 위해 Note객체를 생성할 때 contents의 내용을 넣을 수 있도록 \_\_init\_\_() 함수 안에 self.contents를 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 = None으로 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.

Section 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.")
```

29/46

Section 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.

30/46

Section 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 값(value)으로 들어가게 하고, 키로 각 `Note`의 `page_number`를 사용할 예정이다. 이는 `page_number`로 `Note`를 쉽게 찾기 위해서다.
- ⇒ 다음은 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.

31/46

Section 03 Lab: 노트북 프로그램 만들기

■ 노트북 프로그램 구현 : Notebook 클래스

- ⇒ 두 번째 함수는 `remove_note()` 함수이다. `remove_note()` 함수는 특정 페이지 번호에 있는 `Note`를 제거하는 함수이다. 앞에서 `Note`가 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 알려줄 수 있다.
- ⇒ `page_number`를 넣고 그 페이지가 기존의 노트에 있다면 해당 페이지를 팝하여 돌려주고, 없다면 '해당 페이지는 존재하지 않는다.'라고 출력한다

32/46



Section 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\_clinet.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
```

Section 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)
```

Section 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, 위로되는 명언, 좋은 명언 모음 100가지. 자주 보면 좋을 것 같아 선별했습니다."""
>>> note_1 = Note(good_sentence)
>>>
>>> note_1
<notebook.Note object at 0x000022278C06DD8>
>>> 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.
```

Section 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)
```

Section 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
```

Section 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)
...
```

Section 03 Lab: 노트북 프로그램 만들기

■ 노트북 프로그램 사용

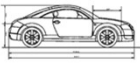
```
해당 페이지에는 이미 노트가 존재합니다.
해당 페이지에는 이미 노트가 존재합니다.
해당 페이지에는 이미 노트가 존재합니다.
해당 페이지에는 이미 노트가 존재합니다.
>>> print(wise_saying_notebook.get_number_of_all_pages())
300
```

Section04 클래스의 상속

■ 상속의 개념

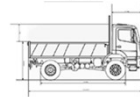
- 클래스의 상속( Inheritance ) : 기존 클래스에 있는 필드와 메서드를 그대로 물려받는 새로운 클래스를 만드는 것

승용차 클래스



```
class 승용차 :
 필드 - 색상, 속도, 좌석 수
 메서드 - 속도 올리기(),
 속도 내리기(),
 좌석 수 알아보기()
```

트럭 클래스



```
class 트럭 :
 필드 - 색상, 속도, 적재량
 메서드 - 속도 올리기(),
 속도 내리기(),
 적재량 알아보기()
```

그림 12-7 승용차와 트럭 클래스의 개념

Section04 클래스의 상속

- 상속의 개념
  - 공통된 내용을 자동차 클래스에 두고 상속을 받음으로써 일관되고 효율적인 프로그래밍 가능
  - 상위 클래스인 자동차 클래스를 슈퍼 클래스 또는 부모 클래스, 하위의 승용차와 트럭 클래스는 서브 클래스 또는 자식 클래스

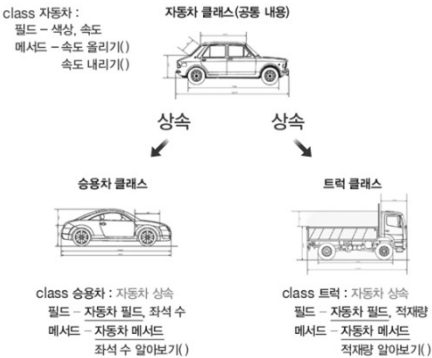
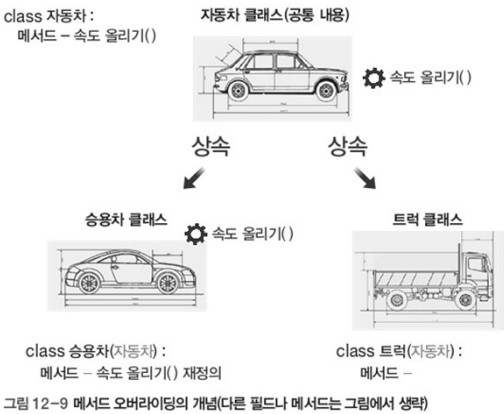


그림 12-8 상속의 개념

Section04 클래스의 상속

- 메서드 오버라이딩
  - 상위 클래스의 메서드를 서브 클래스에서 재정의



Section 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'>
```

Section 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
```

Section 04 객체 지향 프로그래밍의 특징

■ 상속

⇒ 위 코드에서는 먼저 Person 클래스를 생성하였다. Person 클래스에는 생성자 `_init_()` 함수를 만들어 `name`과 `age`에 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 `pass`로 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.

Section 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.



Section 04 객체 지향 프로그래밍의 특징

■ 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.")
```

Section 04 객체 지향 프로그래밍의 특징

■ 상속

⇒ 먼저 부모 클래스 Person이다. `name`, `age`, `gender`에 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.

Section 04 객체 지향 프로그래밍의 특징

■ 상속

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

```
코드 10-5 inheritance2.py
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.")
```

Section 04 객체 지향 프로그래밍의 특징

■ 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 재정의의 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.

Section 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 crawlers = [n_crawler, d_crawler]
4 news = []
5 for crawler in crawlers:
6 news.append(crawler.do_crawling())
```

- ⇒ [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.

Section 04 객체 지향 프로그래밍의 특징

■ 다형성

```
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')]
```

Section 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.

Section 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

Section 04 객체 지향 프로그래밍의 특징

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
```

Section 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 + 17행 실행 결과
new item added + 18행 실행 결과
Traceback (most recent call last): + 20행 실행 결과
 File "visibility1.py", line 20, in <module>
 my_inventory._items
AttributeError: 'Inventory' object has no attribute '_items'
```

Section 04 객체 지향 프로그래밍의 특징

가시성

- ⇒ [코드 10-8]에서는 Inventory 객체에 add\_new\_item( ) 함수를 사용하여 새롭게 생성된 Product 객체를 넣어 준다. \_items는 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.\_items에서 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 @property를 사용한다.

Section 04 객체 지향 프로그래밍의 특징

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.

Section 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 items로 호출할 수 있다는 것이다. 바로 @property를 붙인 함수 이름으로 실제 \_items를 사용할 수 있는 것이다. 이는 기존 private 변수를 누구나 사용할 수 있는 public 변수로 바꾸는 방법 중 하나이다.

Section06 객체지향 프로그래밍의 심화 내용

추상 메서드

- 서브 클래스에서 메서드를 오버라이딩 : 슈퍼 클래스에서는 빈 껍질의 메서드만 만들어 놓고 내용은 pass 로 채움

```
Code12-10.py
1 ## 클래스 선언 부분 ## 2 ~ 11행 : SuperClass 상속받은 SubClass1 과 SubClass2 만듦
2 class SuperClass : 14 ~ 15행 : 각 인스턴스 sub1 과 sub2 생성
3 def method(self) : 17 ~ 18행 : 오버라이딩한 method () 호출
4 pass
5
6 class SubClass1(SuperClass) :
7 def method(self) : # 메서드 오버라이딩
8 print('SubClass1에서 method()를 오버라이딩함')
9
10 class SubClass2(SuperClass) :
11 pass
12
13 ## 메인 코드 부분 ##
14 sub1 = SubClass1()
15 sub2 = SubClass2()
16
17 sub1.method()
18 sub2.method()

출력 결과
SubClass1에서 method()를 오버라이딩함
```