

10

객체지향 프로그래밍

목차

1. 객체 지향 프로그래밍의 이해
2. 파이썬의 객체 지향 프로그래밍
3. 객체 지향 프로그래밍의 특징

01. 객체 지향 프로그래밍의 이해

■ 객체와 클래스

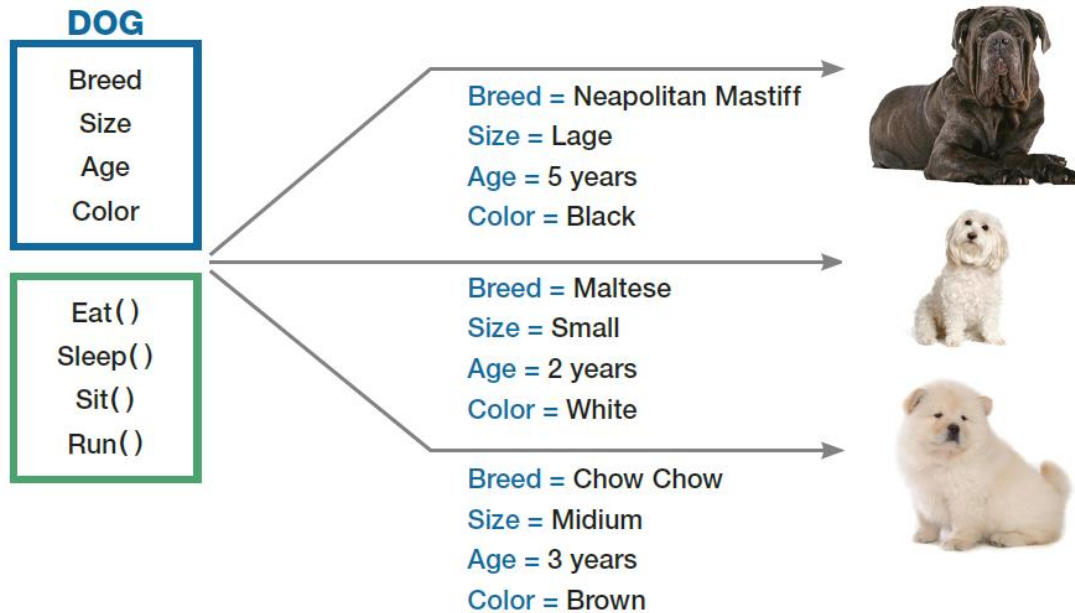
개념	설명	예시
객체(object)	실생활에 존재하는 실제적인 물건 또는 개념	심판, 선수, 팀
속성(attribute)	객체가 가지고 있는 변수	선수의 이름, 포지션, 소속팀
행동(action)	객체가 실제로 작동할 수 있는 함수, 메서드	공을 차다, 패스하다

[객체, 속성, 행동]

01. 객체 지향 프로그래밍의 이해

■ 객체와 클래스

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



[Dog 인스턴스]

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, 파이썬 함수나 변수명에 사용됨
CamelCase	띄어쓰기 부분에 대문자를 사용하여 변수의 이름을 지정함, 낙타의 혹처럼 생겼다하여 Camel 이라고 명명, 파이썬 클래스명에 사용됨

[파이썬에서 자주 사용하는 작명 기법]

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__()` 함수 : 이 class에서 사용할 변수를 정의하는 함수, `__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.

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.

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")
```

[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 사용하기

```
jinyun = SoccerPlayer("Jinyun", "MF", 10):
```

객체명

클래스 이름

__init__ 함수 Interface, 초기값

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

- 클래스에서 인스턴스를 호출하는 방법은 위와 같다. 먼저 클래스 이름을 사용하여 호출하고, 앞서 만든 `__init__()` 함수의 매개변수에 맞추어 값을 입력한다. 여기에서는 함수에서 배운 초기 값 지정 등도 사용할 수 있다. 여기서 `self` 변수는 아무런 값도 할당되지 않는다.
- `jinyun`이라는 인스턴스가 기존 `SoccerPlayer`의 클래스를 기반으로 생성되는 것을 확인할 수 있다. 이 `jinyun`이라는 인스턴스 자체가 `SoccerPlayer` 클래스에서 `self`에 할당된다.

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_
          number))
9          self.back_number = new_number
10     def __str__(self):
11         return "Hello, My name is %. I play in %s in center." % (self.name,
            self.position)
12
```

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)
```

현재 선수의 등번호는: 10	← 16행 실행 결과
선수의 등번호를 변경한다: From 10 to 5	← 17행 실행 결과
현재 선수의 등번호는: 5	← 18행 실행 결과

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

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.

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 `jinhyun`을 단순히 `print()` 함수로 썼을 때 나오는 결과이다. 이는 [코드10-2]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

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.

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,
    positions, numbers)]
8 print(players)
9 print(players[0])
10
```

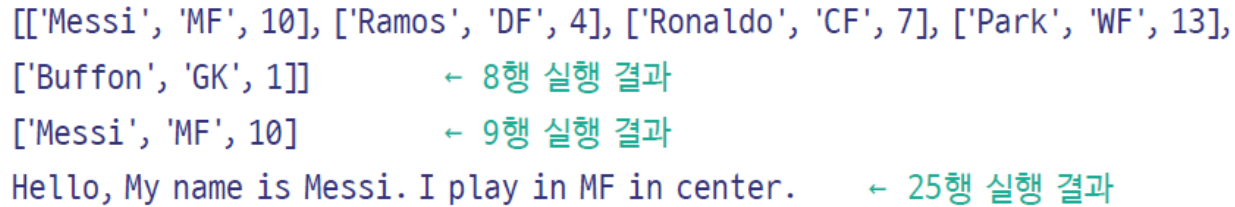

02. 파이썬의 객체 지향 프로그래밍

■ 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))
20         self.back_number = new_number
21     def __str__(self):
22         return "Hello, My name is %. I play in %s in center." % (self.name,
23             self.position)
24 # 클래스-인스턴스
25 player_objects = [SoccerPlayer(name, position, number) for name, position,
26     number in zip(names, positions, numbers)]
27 print(player_objects[0])
```

02. 파이썬의 객체 지향 프로그래밍

■ 클래스를 사용하는 이유



```
[[ 'Messi', 'MF', 10], [ 'Ramos', 'DF', 4], [ 'Ronaldo', 'CF', 7], [ 'Park', 'WF', 13],  
[ 'Buffon', 'GK', 1]]          ← 8행 실행 결과  
[ 'Messi', 'MF', 10]          ← 9행 실행 결과  
Hello, My name is Messi. I play in MF in center.    ← 25행 실행 결과
```

03. 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'>
```

03. 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
```

03. 객체 지향 프로그래밍의 특징

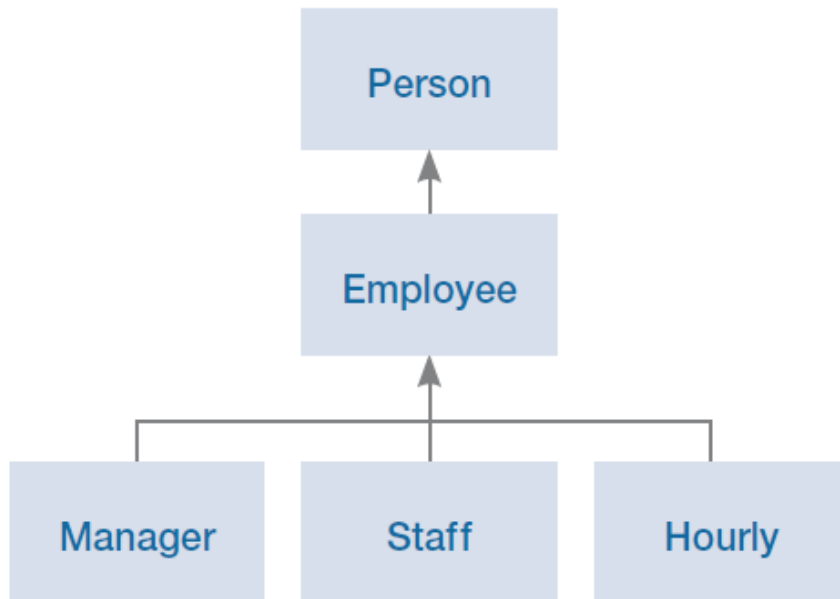
■ 상속

- ➔ 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.

03. 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]

03. 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.")
```

03. 객체 지향 프로그래밍의 특징

■ 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 들어가는 클래스이지만 임의의 about_me 클래스를 생성하였다.

03. 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.")
```

03. 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.

03. 객체 지향 프로그래밍의 특징

■ 다형성

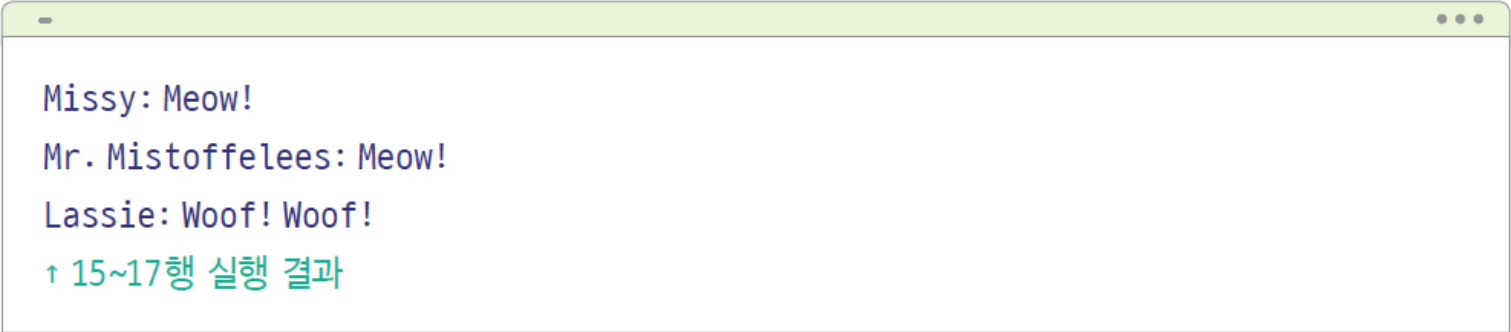
코드 10-7 polymorphism2.py

```
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')]
```

03. 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.

03. 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

03. 객체 지향 프로그래밍의 특징

■ 가시성

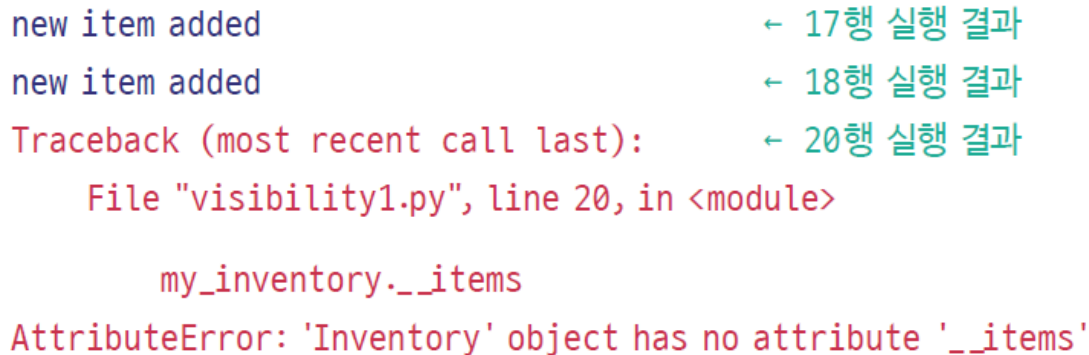
코드 10-8 visibility1.py

```
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
```

03. 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
```



A terminal window with a light green title bar and three dots in the top right corner. It displays the output of a Python program. The first two lines are 'new item added' in blue. The third line is 'Traceback (most recent call last):' in red. The fourth line is 'File "visibility1.py", line 20, in <module>' in red. The fifth line is 'my_inventory._items' in red. The sixth line is 'AttributeError: 'Inventory' object has no attribute '_items'' in red. To the right of the first three lines are green arrows pointing to the corresponding code lines in the previous block: '← 17행 실행 결과', '← 18행 실행 결과', and '← 20행 실행 결과'.

```
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'
```

03. 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.

03. 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.