

# 데이터 과학을 위한 파이썬 프로그래밍

2판



# Chapter 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

# 학습목표

- 객체 지향 프로그래밍을 배우는 이유에 대해 알아본다.
- 객체와 클래스의 개념에 대해 학습한다.
- 클래스를 구현하고 인스턴스를 사용하는 방법을 이해한다.
- 객체 지향 프로그래밍의 특징인 상속, 다형성, 가시성에 대해 학습한다.

**01**

# **객체 지향 프로그래밍의 이해**

# 1. 객체 지향 프로그래밍을 배우는 이유

- 객체 지향 프로그래밍(Object Oriented Programming, OOP)
  - 내가 아닌 남이 만든 코드를 재사용하고 싶을 때 사용하는 대표적인 방법
  - 함수처럼 어떤 기능을 함수 코드로 묶어 두는 것이 아니라, 어떤 기능을 수행하는 하나의 단일 프로그램을 객체라고 하는 코드로 만들어 다른 프로그래머가 재사용할 수 있도록 함.

## 2. 객체와 클래스

- **객체(object)**: 실생활에 존재하는 실제적인 물건 또는 개념
- **객체 지향 프로그래밍**: 객체의 개념을 활용하여 프로그램으로 표현하는 기법

표 10-1 객체, 속성, 행동

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 작동시키는 함수, 메서드	공을 차다, 패스하다

- 객체는 하나의 프로그램에서 여러 개가 사용될 수도 있으므로 객체들을 위한 설계도를 만들어야 함.
- ☞ **클래스(class)**: 객체가 가져야 할 기본 정보를 담은 코드, 일종의 설계도 코드

## 2. 객체와 클래스

- 'DOG' 클래스

- 실제 사용하는 종류별로 Dog 인스턴스를 만들 수 있음.

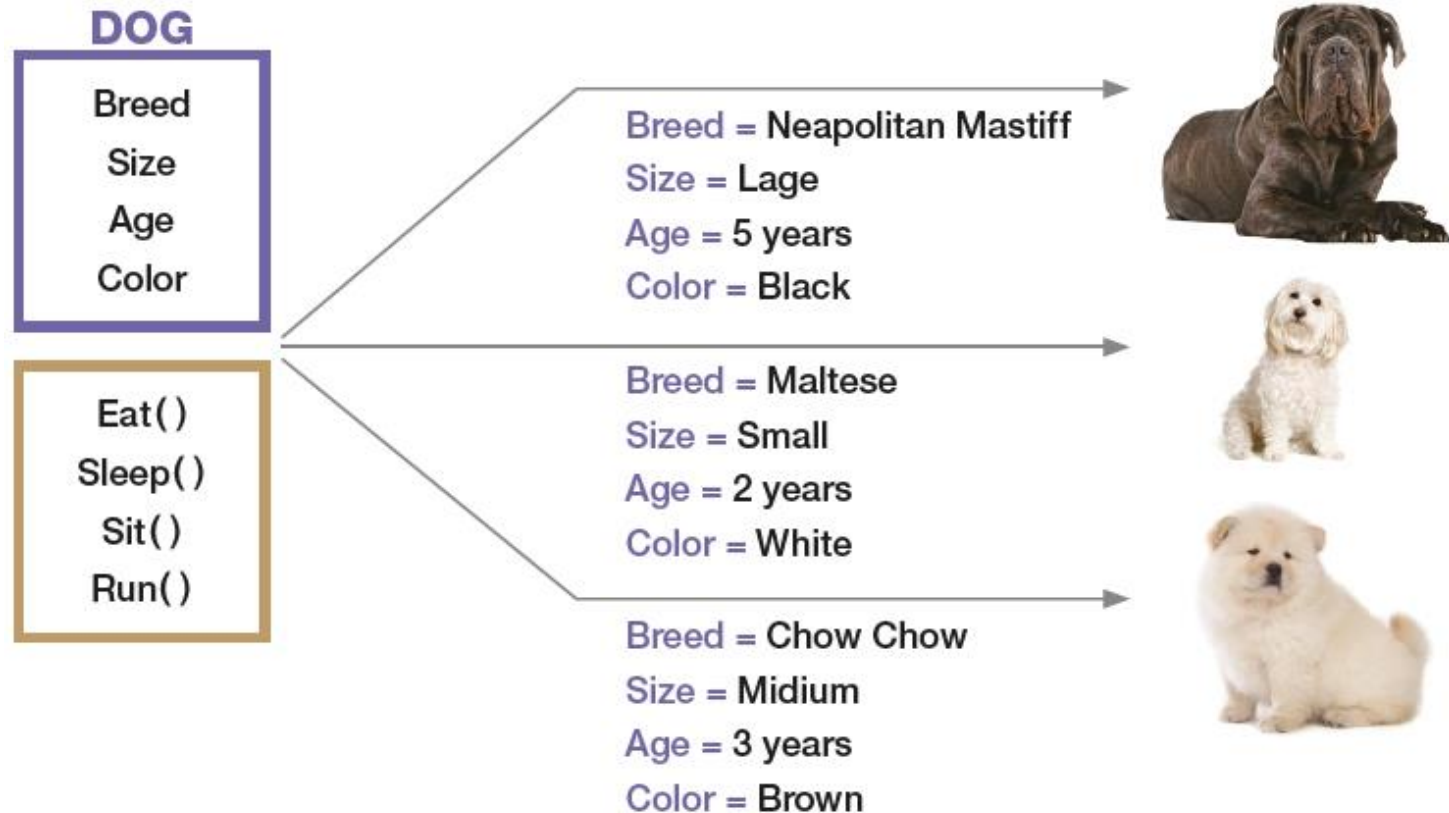


그림 10-1 Dog 인스턴스



## 2. 객체와 클래스

- 붕어빵 틀과 붕어빵
  - 붕어빵 가게의 붕어빵 틀을 이용해 여러 종류의 붕어빵을 만들 수 있음.



붕어빵틀  
(클래스)



붕어빵  
(인스턴스)

그림 10-2 클래스와 인스턴스의 관계

**02**

# **파이썬의 객체 지향 프로그래밍**

# 1. 클래스 구현하기

- 파이썬에 클래스를 선언하기 위한 기본 코드 템플릿

class SoccerPlayer(object):




클래스 예약어      클래스 이름      상속받는 객체명

그림 10-3 파이썬에서의 클래스 선언

- 예약어인 class를 코드의 맨 앞에 입력하고, 만들고자 하는 클래스 이름을 작성함.
- 그 다음으로 상속받아야 하는 다른 클래스의 이름을 괄호 안에 넣음.

# 1. 클래스 구현하기

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 의한 것이다. 파이썬뿐만 아니라 모든 컴퓨터 프로그래밍 언어에서 변수, 클래스, 함수명을 짓는 작명 기법이 있다. [표 10-2]는 프로그래머가 가장 많이 사용하는 작명 기법이다.

표 10-2 파이썬에서 자주 사용하는 작명 기법

작명 기법	설명
snake_case	띄어쓰기 부분에 '_'를 추가하여 변수의 이름을 지정한다. 파이썬 함수나 변수명에 사용된다.
CamelCase	띄어쓰기 부분에 대문자를 사용하여 변수의 이름을 지정한다. 낙타의 혹처럼 생겼다 하여 Camel이라고 부르고, 주로 파이썬 클래스명에 사용된다.

# 1. 클래스 구현하기

## 1.1 속성의 선언

- 축구 선수 클래스 구성
- 먼저 클래스의 속성을 추가하는 선언
  - 속성에 대한 정보를 선언하기 위해서는 `__init__()`라는 예약 함수 사용
  - 파이썬 클래스의 대표적 예약 함수: `__init__()`, `__str__`, `__add__` 등

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

- `__init__()` 함수: 이 클래스에서 사용할 변수를 정의하는 함수

# 1. 클래스 구현하기

## 1.2 함수의 선언

- 함수는 이 클래스가 할 수 있는 다양한 동작을 정의할 수 있음
- 축구 선수가 등번호 교체라는 행동을 함수 코드로 표현하기

```
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 함수의 이름을 쓰고 매개변수를 사용함
- 가장 큰 차이점은 바로 self를 매개변수에 반드시 넣어야 한다는 것!

# 1. 클래스 구현하기

## 1.3 \_의 쓰임

- 파이썬에서 \_의 쓰임은 개수에 따라 여러 가지로 나눌 수 있음.

### [코드 10-1]

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

### [실행결과]

```
Hello, World  
Hello, World  
Hello, World  
Hello, World  
Hello, World  
Hello, World  
Hello, World  
Hello, World  
Hello, World  
Hello, World
```

## 2. 인스턴스 사용하기

- 생성된 클래스를 인스턴스로 호출해 사용하는 방법
- 인스턴스: 클래스에서 실제적인 데이터가 입력되어 사용할 수 있는 형태의 객체
- 클래스에서 인스턴스를 호출하는 방법

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

객체명

클래스 이름

\_\_init\_\_ 함수 인터페이스 초깃값

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

그림 10-4 클래스에서 인스턴스 호출



## 2. 인스턴스 사용하기

### [코드 10-2]

```
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" %
9               (self.back_number, new_number))
10        self.back_number = new_number
11    def __str__(self):
12        return "Hello, My name is %s. I play in %s in center."
13        % (self.name, self.position)
14
15 # SoccerPlayer를 사용하는 instance 코드
16 jinhyun = SoccerPlayer("Jinhyun", "MF", 10)
17
18 print("현재 선수의 등번호는:", jinhyun.back_number)
19 jinhyun.change_back_number(5)
20 print("현재 선수의 등번호는:", jinhyun.back_number)
```

## 2. 인스턴스 사용하기

### [실행결과]

현재 선수의 등번호는: 10

선수의 등번호를 변경한다: From 10 to 5

현재 선수의 등번호는: 5

← 16행 실행 결과

← 17행 실행 결과

← 18행 실행 결과

- [코드 10-2]에 이어서 19행으로 `print(jinhyun)`을 입력했을때의 출력 결과

### [실행결과]

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

- 생성된 인스턴스인 `jinhyun`을 `print( )` 함수에서 사용했을 때 나타나는 결과

☞ [코드 10-2]의 10 · 11행에서 클래스 내 함수로 선언되었기 때문

### 3. 클래스를 사용하는 이유

#### ■ 클래스를 사용하는 이유

- 자신의 코드를 다른 사람이 손쉽게 사용할 수 있도록 설계하기 위함.
- 코드를 좀 더 손쉽게 선언할 수 있음.
  - [코드 10-3]을 보면 단순히 이차원 리스트로 선언할 수 있는 것을 객체 지향 프로그래밍의 개념을 적용시킴으로 좀 더 명확하게 저장된 데이터를 확인할 수 있음.

### 3. 클래스를 사용하는 이유

#### [코드 10-3]

```
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
8              zip(names, positions, numbers)]
9  print(players)
10 print(players[0])
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" %
19               (self.back_number, new_number))
```

### 3. 클래스를 사용하는 이유

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

#### [실행결과]

```
[['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**

**Lab: 노트북 프로그램 만들기**

- 객체 지향 프로그래밍을 이용하여 프로그램을 만들기 위해 가장 먼저 해야 할 일은 프로그램 설계!

- 이 프로그램은 어떤 객체가 필요할까?
- 어떤 기능을 어떻게 정의해 사용해야 할까?
- 어떤 데이터를 저장해야 할까?

☞ 이런 내용을 정리하여 프로그램을 설계해야 한다

- 사용자 요구사항

- 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 더이상 삽입하지 못한다.

- 객체 - 사용자 · 노트 · 노트북 등
- 저장하는 데이터(=변수) 정의
  - 가장 기본적인 것은 입력해야 할 콘텐츠가 있고, 노트마다 페이지가 생성되어야 함.
  - 노트북은 노트라고 하는 객체와 타이틀을 포함해야 함.
- 기능(function) 정의: 노트북/노트의 입장, 각각의 객체별로 기능을 정의
  - 노트: 내용을 입력하는 함수와 지우는 함수가 필요
  - 노트북: 노트를 추가하거나 지우는 등의 함수가 필요
  - 그 외: 300페이지 등에 대한 로직을 따로 정의하여 함수 안에 넣어야 함.

표 10-3 노트북 프로그램의 객체 설계

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



## 2.1 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
```

## 2.2 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.")
    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 클래스에 필요한 3가지
  - ❶ 타이틀(title)
  - ❷ 페이지 수(page\_number)
  - ❸ 저장 공간
- 이 세 가지 정보를 저장하기 위해 `__init__()` 함수 안에 모든 정보를 입력함
- 중요한 변수는 `self.notes` - 위 코드에서는 `notes` 변수를 딕셔너리형으로 선언
  - `notes` 변수 안에는 `Note`의 인스턴스가 값(value)으로 들어가게 되고, 각 `Note`의 `page_number`가 키(key)로 들어감

- **add\_note( ) 함수:** 새로운 Note를 Notebook에 삽입하는 함수
  - 몇 가지 요구 조건에 대한 로직이 들어감
  - 맨 마지막 페이지는 늘 `page_number`에 저장되어 있음
- **remove\_note( ) 함수:** 특정 페이지 번호에 있는 Note를 제거하는 함수
  - 앞에서 Note가 저장되는 객체를 딕셔너리형으로 만들었고 페이지 번호를 키로 저장했으므로, 딕셔너리형인 `self.notes`에 해당 페이지 번호의 키가 있는지 확인하면 됨.
  - 이는 `page_number in self.notes.keys()`로 쉽게 확인할 수 있음.

- 이 장에서는 따로 모듈을 설명하지 않고 소스파일에서 제공하는 'notebook.py'와 'notebook\_client.py' 파일을 활용

notebook.py	Note 클래스와 NoteBook 클래스에 모듈을 적용하여 저장한 파일
notebook_client.py	지금 코딩할 내용을 저장한 파일

- 해당 클래스를 불러 사용할 수 있는 클라이언트 코드 만들기
  - 지금부터 작성하는 코드는 'notebook\_client.py'에 있는 것
  - 먼저 2개의 클래스 호출

```
from notebook import Note
from notebook import NoteBook
```

- 다음으로 몇 장의 Note를 생성함.

```
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)
```

- 모두 4개의 Note가 만들어짐. 이 Note를 파이썬 셸에서 확인하기 위해 다음과 같이 코드를 입력하면 Note의 인스턴스 생성을 확인할 수 있음

```
>>> 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 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.
```

- 다음은 새로운 Notebook을 생성하는 코드
  - 새로운 노트북을 생성한 후 기존의 Note들을 add\_note( ) 함수로 추가.

```
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)
```



- 다음 코드에서 실제 추가된 것을 확인할 수 있음.

```
>>> 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
```

- 노트의 삭제나 추가도 여러 가지 명령어로 가능
- 특정 Note를 지우는 `remove_note( )`를 사용하거나 새로운 빈 노트를 임의 추가도 가능.

```
>>> wise_saying_notebook.remove_note(3)
>>> print(wise_saying_notebook.get_number_of_all_pages())
3
>>>
>>> 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)
...
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**

# 1. 상속

- **상속(inheritance):** 그대로 무엇인가를 내려 받음

☞ 부모 클래스에 정의된 속성과 메서드를 자식 클래스가 물려받아 사용

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

- 예약어 class 다음에 클래스명 Person을 쓰고 ( ) 안에 object를 입력함.
- **Object:** 바로 Person 클래스의 부모 클래스.

# 1. 상속

- 파이썬의 문자열형 변수에 대해 다음과 같이 객체 이름을 확인할 수 있음

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

- 객체 지향 프로그래밍에서의 상속: 부모 클래스로부터 값과 메서드를 물려받아 자식 클래스를 생성함. 🖱️ 자식이 부모의 특성을 그대로 포함한 채 생성

```
>>> 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
```

# 1. 상속

- 다음으로 Korean 클래스를 만들면서 Person 클래스를 상속받음.
  - `class Korean(Person)`과 같이 작성하면 간단히 상속받을 수 있음
  - `pass`: 별도의 내용 없이 클래스만 존재한다는 뜻. 즉, 별도의 내용이 없는 Korean 클래스 인스턴스를 만든 것.

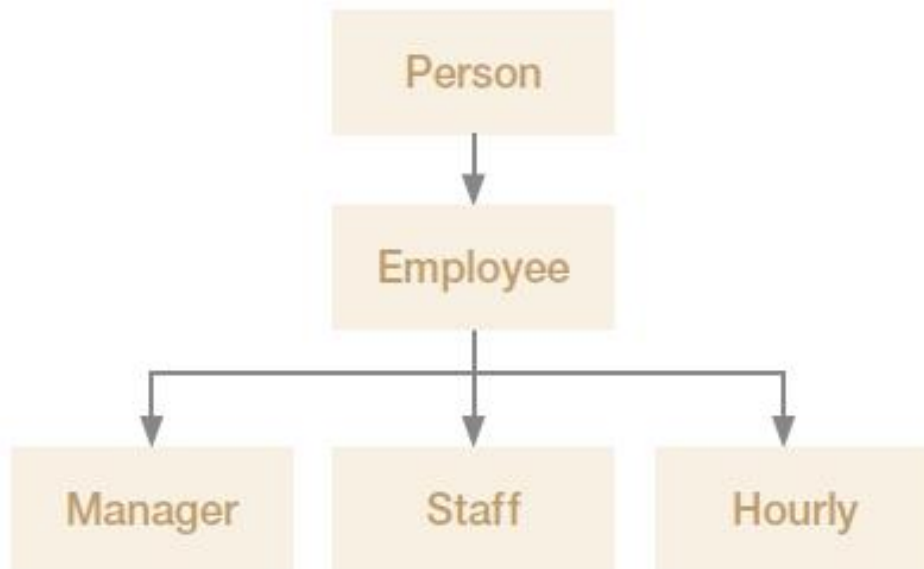


그림 10-5 상속 구조

# 1. 상속

[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.")
```

- 부모 클래스가 Person.
- name, age, gender에 대해 변수 선언
- about\_me 함수를 사용하여 생성된 인스턴스가 자신을 설명할 수 있도록 함.
- str() 함수에 들어가도 되는 클래스이지만 임의의 about\_me 클래스를 생성함.

# 1. 상속

[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.")
```



# 1. 상속

- Person 클래스가 단순히 사람에 대한 정보를 정의했다면, Employee 클래스는 사람에 대한 정의와 함께 일하는 시간과 월급에 대한 변수를 추가함.

☞ `__init__()` 함수 재정의.

- 이러한 함수의 재정의를 오버라이딩(overriding)이라고 함.
  - **오버라이딩**: 상속 시 함수 이름과 필요한 매개변수는 그대로 유지하면서 함수의 수행 코드를 변경하는 것

## 2. 다형성

- **다형성(polymorphism):** 같은 이름의 메서드가 다른 기능을 하는 것.
- 상속에서 about\_me라는 함수를 부모 클래스와 자식 클래스가 서로 다르게 구현했는데, 이것도 일종의 다형성임.

### ➤ 뉴스를 모으는 크롤러 만들기

- **크롤러(crawler):** 인터넷에서 데이터를 모으는 프로그램

#### [코드 10-6]

```
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())
```

## 2. 다형성

### [코드 10-7]

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

## 2. 다형성

### [실행결과]

Missy: Meow!

Mr. Mistoffelees: Meow!

Lassie: Woof! Woof!

↑ 15~17행 실행 결과

### 3. 가시성

- **가시성(visibility)**: 객체의 정보를 볼 수 있는 레벨을 조절하여 객체의 정보 접근을 숨기는 것
- **캡슐화(encapsulation)**: 객체의 세부 내용은 모른 채 객체의 사용법만 알고 사용한다는 뜻
- **정보 은닉(information hiding)**: 외부에서 코드 내부를 볼 수 없게 하기 위해 내부의 정보를 숨기는 개념

☞ 캡슐화와 정보 은닉으로 표현은 다르지만 둘 다 코드의 내부 구현을 잘 해서 외부에서 쉽게 사용하게 하고, 코드의 세부적인 내용은 모르게 한다는 측면에서 비슷한 의미로 사용됨.

### 3. 가시성

- 캡슐화를 사용해야 하는 이유는?

☞ 클래스를 설계할 때 클래스 간 간섭 및 정보 공유를 최소화하여 개별 클래스가 단독으로도 잘 동작할 수 있도록 해야 하기 때문

- 코드를 작성해야 하는 상황

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

### 3. 가시성

#### [코드 10-8]

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

### 3. 가시성

#### [실행결과]

```
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'
```



### 3. 가시성

#### [코드 10-9]

```
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
```

- 이렇게 코드를 추가하면 다음과 같이 외부에서도 해당 메서드를 사용할 수 있음.

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

# Thank you!