Python
CH06: 클래스

전종준 교수

서울시립대학교 통계학과/통계데이터사이언스대학원

Contents

- 클래스 만들기

- 상속

클래스 만들기

데이터와 함수의 관리

- 한 학생의 이름과 점수를 딕셔너리로 저장할 수 있음
- 하지만 총점이나 평균을 계산하려면, 딕셔너리 바깥에서 따로 계산해야 함

예시 코드:

```
student = {
    "name": "김철수",
    "scores": [80, 90, 85]
}

total = sum(student["scores"])
print(f"{student['name']}의 총점은 {total}점입니다.")
```

클래스란 무엇인가요?

- 클래스는 '객체'를 만들기 위한 설계도
- 객체는 속성과 행동을 가진 실체입니다
- 예: 학생, 책, 은행 계좌, 캐릭터 등

정의: "클래스는 데이터와 행동을 함께 정의하는 틀."

왜 클래스를 써야 하나요?

- 함수로는 행동만 정의.
- 딕셔너리는 데이터를 담지만, 행동은 따로 처리
- 클래스는 **데이터 + 행동**을 하나의 구조로 묶을 수 있음

클래스의 장점:

- 코드가 명확해짐
- 재사용과 확장이 용이
- 유지보수가 편리

클래스 만들기와 객체 생성

다음 코드를 실행해보자. class Student 는 Student 라는 class (설계도)를 만든다는 뜻임. 아래 코드는 설계도에 대한 내용

```
1  class Student:
2      def __init__(self, name, major):
3          self.name = name
4          self.major = major
5
6      def introduce(self):
7          print(f"안녕하세요. {self.name}이고, 전공은 {self.major}입니다.")
```

- Student 클래스 내에 두개의 함수가 정의됨: __init__ 과 introduce

클래스 만들기와 객체 생성

Student(”김철수”, ”컴퓨터공학”) 는 설계도에 인자를 넣고 구체화된 제품 인스턴스를 만드는 작업임. 인스턴스의 이름은 s1. 다음으로 s1이 가진 기능 introduce 를 실행함.

객체 만들기:

```
1 s1 = Student("김철수", "컴퓨터공학")
2 s1.introduce()
```

- instance 가 붕어빵이라면 class 는 붕어빵 틀.
- 붕어빵 틀에 넣는 앙꼬와 반죽이 instance를 생성할 때 사용하는 인자가 됨.

간단한 클래스 예시: Person

- 이름을 저장하고 인사하는 사람(Person) 클래스를 해석해 보자.
- class 이름, class 내에 정의한 함수는 무엇인가?

```
1  class Person:
2      def __init__(self, name):
3          self.name = name
4
5      def greet(self):
6          print(f"안녕하세요, 저는 {self.name}입니다.")
```

간단한 클래스 예시: Person

- `Person` 붕어빵은 몇 개인가?
- 각 객체의 `greet()` 같은 붕어빵인가? 어떻게 다른가?

```
p1 = Person("김철수")
p2 = Person("이영희")

p1.greet()
p2.greet()
```

__init__ 이란?

- 객체가 생성될 때 자동으로 실행되는 **특별한 함수**
- 객체의 초기값(속성)을 설정하는 데 사용
- 생성자(Constructor)라고 부름

예시 구조:

```
1 class Student:
2     def __init__(self, name):
3         self.name = name
```

객체를 만들 때 `__init__()`이 자동 호출됨: `s = Student("김철수")`
객체가 가진 변수 self.name에 class 를 이용해 객체를 생성할때 상용한 인지 "김철수"를 전달함.
결국 s.name 에 '김철수'가 할당됨.

self란 무엇인가요?

- 클래스 안의 메서드는 항상 첫 번째 인자로 self를 받음.
- **self는 '자기 자신 객체'**를 뜻함
- 객체의 속성에 접근하거나, 다른 메서드를 부를 때 사용

예시:

```
1  class Dog:
2      def __init__(self, name):
3          self.name = name
4
5      def bark(self):
6          print(self.name + "가 멍멍 짖습니다!")
```

메서드란 무엇인가?

- 메서드는 **클래스 안에 정의된 함수**
- 객체가 가지고 있는 **동작(기능)**을 나타냄
- 항상 첫 번째 인자로 `self`를 받는다.

형식 예시:

```
1 class Sample:
2     def say_hello(self):
3         print("Hello!")
```

`say_hello()`는 **Sample 객체가 수행할 수 있는 동작**입니다. 참고로 위 클래스는 인스턴스가 생성될 때 어떤 동작이 수행되는 것을 정의하지 않았음

클래스 안에 여러 메서드 정의하기

- 클래스는 관련 있는 여러 기능을 함께 정의할 수 있음.
- 각 기능은 메서드(클래스 안의 함수)로 정의.

예: 계산기 클래스

```
1  class Calculator:
2      def __init__(self, x):
3          self.x = x
4
5      def square(self):
6          return self.x ** 2
7
8      def double(self):
9          return self.x * 2
```

메서드 호출과 인자 전달

- 객체를 만들면, 그 객체를 통해 메서드를 사용할 수 있음.
- self는 자동으로 전달되므로, .메서드() 형태로 호출.

예:

```
1  calc = Calculator(5)
2
3  print(calc.square())    # 25
4  print(calc.double())    # 10
```

- cals.x 에는 calc = Calculator(5) 를 실행할때 5 가 할당되어 있음
- square 메서드와 double 메서드 모두 cals.x를 이용하여 결과를 출력함.

클래스 안에서 메서드 간 호출하기

- 하나의 메서드에서 다른 메서드를 부를 때도 `self.메서드()`를 사용.

예:

```
class Student:
    def __init__(self, name, scores):
        self.name = name
        self.scores = scores

    def total(self):
        return sum(self.scores)

    def average(self):
        return self.total() / len(self.scores)
```

연습 1

- 생성자 `__init__()`는 기본 설정만 있음
- 실제 동작은 메서드에서 인자를 받아 수행

예: 계산기 클래스

```
1  class Calculator:
2      def __init__(self):
3          print("계산기를 시작합니다.")
4
5      def add(self, a, b):
6          return a + b
7
8      def multiply(self, a, b):
9          return a * b
```

연습 1

- 객체를 만든 뒤, 메서드에 인자를 전달해 사용
- `self`는 자동 전달되므로 따로 넣을 필요 없음

```
calc = Calculator()       # 생성자에 인자 없음
print(calc.add(3, 5))     # 8 출력
print(calc.multiply(4, 6))  # 24 출력
```

연습 2

- 생성자에서 `name`을 받아 저장
- 메서드 `greet()`는 인사할 횟수 `n`을 인자로 받음

예시 코드:

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

    def greet(self, n):
        for _ in range(n):
            print(f"안녕하세요, {self.name}입니다!")
```

연습2

- `name`은 객체를 만들 때 전달
- `greet(n)`은 실행할 때마다 다른 `n`을 넣을 수 있음

```
g1 = Greeter("김철수")
g1.greet(2)

g2 = Greeter("이영희")
g2.greet(3)
```

연습3: 게임 캐릭터 클래스 설계하기

- 이름, 직업, 체력(HP)을 가진 캐릭터를 만들어 보자
- `show_status()`로 상태를 출력
- `attack()`으로 공격 기능을 구현

```
1  class Character:
2      def __init__(self, name, job, hp):
3          self.name = name
4          self.job = job
5          self.hp = hp
6
7      def show_status(self):
8          print(f"{self.name} [{self.job}] - HP: {self.hp}")
9
10     def attack(self, target):
11         print(f"{self.name}이(가) {target}을 공격합니다!")
```

연습3: 캐릭터 객체 만들고 사용하기

- 캐릭터를 생성하고 메서드를 호출해보자.

```
c1 = Character("아리", "마법사", 120)
c2 = Character("브람", "전사", 150)

c1.show_status()
c2.show_status()

c1.attack("슬라임")
```

출력 예시:

아리 [마법사] - HP: 120
브람 [전사] - HP: 150
아리이(가) 슬라임을 공격합니다!

연습3: 공격력 속성을 추가한 캐릭터 클래스

- 각 캐릭터는 고유한 공격력을 가짐
- `attack()` 메서드에서 대상의 체력을 줄임

```
1  class Character:
2      def __init__(self, name, job, hp, power):
3          self.name = name
4          self.job = job
5          self.hp = hp
6          self.power = power
7
8      def show_status(self):
9          print(f"{self.name} [{self.job}] - HP: {self.hp}, 공격력: {self.power}")
10
11     def attack(self, target):
12         print(f"{self.name}이(가) {target.name}을 공격합니다!")
13         target.hp -= self.power
```

연습3: 전투 시뮬레이션 예시

- 공격 시 대상 캐릭터의 체력이 감소
- 전투 후 상태를 출력해 볼 수 있음

```
c1 = Character("아리", "마법사", 100, 25)
c2 = Character("고블린", "몬스터", 80, 15)

c1.attack(c2)
c2.show_status()
```

출력 결과 예시:

아리이(가) 고블린을 공격합니다!
고블린 [몬스터] - HP: 55, 공격력: 15

연습3: 전투를 함수나 클래스로 분리하기

- 전투 로직을 **별도의 함수나 클래스**로 관리하면 더 유연함
- 예: 한 턴씩 번갈아 공격하기, 죽었는지 판별하기 등

전투 함수 예시:

```
def battle(player, enemy):
    while player.hp > 0 and enemy.hp > 0:
        player.attack(enemy)
        if enemy.hp <= 0:
            print(f"{enemy.name}이 쓰러졌습니다!")
            break
        enemy.attack(player)
        if player.hp <= 0:
            print(f"{player.name}이 쓰러졌습니다!")
```

연습3: battle() 함수 실행 예시

- 캐릭터 두 명을 만들고 전투를 시켜보자

```
hero = Character("아리", "마법사", 100, 30)
monster = Character("슬라임", "몬스터", 80, 15)

battle(hero, monster)
```

출력 예시:

```
아리이(가) 슬라임을 공격합니다!
슬라임이(가) 아리를 공격합니다!
...
슬라임이 쓰러졌습니다!
```

상속

상속이란 무엇인가요?

- 상속은 기존 클래스의 기능을 **물려받아** 새로운 클래스를 만드는 방법
- 코드의 중복을 줄이고, 공통 기능을 재사용할 수 있음
- 부모 클래스(기반 클래스) → 자식 클래스(파생 클래스)

상속 문법: 자식 클래스 만들기

- `class 자식클래스(부모클래스):` 형식으로 정의
- 부모 클래스의 속성과 메서드를 그대로 사용할 수 있음

```
1  class Animal:
2      def speak(self):
3          print("동물이 소리를 냅니다.")
4
5  class Dog(Animal):
6      pass # 문법적으로 써야함
```

Dog 클래스는 Animal을 상속받았기 때문에 speak 메소드를 사용할 수 있음

상속받은 메서드 사용해보기

```
1 d = Dog()
2 d.speak()
```

출력 결과: 동물이 소리를 냅니다.

- `Dog` 클래스에는 `speak()` 메서드가 없지만,
- 부모 클래스 `Animal`의 메서드를 물려받아 사용할 수 있음

자식 클래스에서 새로운 기능 추가하기

```
class Dog(Animal):
    def bark(self):
        print("멍멍!")
```

```
d = Dog()
d.speak()  # 부모 클래스 메서드
d.bark()   # 자식 클래스 메서드
```

부모 메서드를 자식 클래스에서 바꾸기: 오버라이딩

- 자식 클래스에서 같은 이름의 메서드를 다시 정의하면,
- 부모 클래스의 메서드를 덮어씀 (Override).

```
class Cat(Animal):
    def speak(self):
        print("야옹~")
```

```
c = Cat()
c.speak()  # 야옹~
```

super()란?

- super()는 부모 클래스의 메서드를 호출할 때 사용
- 주로 생성자 __init__()에서 부모 초기화를 그대로 사용하거나 확장할 때 사용함

형식:

```
1 super().메서드이름(인자)
```

예:

```
1 super().__init__(name)
```

super() 사용 예제

```python
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        print(f"{self.name}이(가) 소리를 냅니다.")

class Dog(Animal):
    def __init__(self, name, breed):
        super().__init__(name)  # 부모 생성자 호출
        self.breed = breed

    def show(self):
        print(f"{self.name} / 품종: {self.breed}")
```

super()로 부모 초기화 확장하기

```
d = Dog("초코", "푸들")
d.speak()
d.show()
```

출력 결과:

초코이(가) 소리를 냅니다.
초코 / 품종: 푸들

정리:

- `super()`를 사용하면 부모의 생성자/메서드를 재사용할 수 있음
- 자식 클래스는 필요한 부분만 추가해서 확장 가능

예시1: 기본 캐릭터 클래스 정의

- 이름과 체력(HP), 공격력을 가지는 기본 캐릭터 클래스를 만든다.

```python
class Character:
    def __init__(self, name, hp, power):
        self.name = name
        self.hp = hp
        self.power = power

    def attack(self, target):
        print(f"{self.name}이(가) {target.name}을 공격합니다!")
        target.hp -= self.power
```

예시1: 직업별 캐릭터 클래스 만들기

- 전사(Warrior)는 높은 체력과 중간 공격력
- 마법사(Mage)는 낮은 체력이지만 높은 공격력

```
class Warrior(Character):
    def __init__(self, name):
        super().__init__(name, hp=150, power=20)

class Mage(Character):
    def __init__(self, name):
        super().__init__(name, hp=80, power=40)
```

예시1: 직업별 캐릭터 생성 및 전투

```
w = Warrior("브람")
m = Mage("아리")

w.attack(m)
print(f"{m.name}의 남은 HP: {m.hp}")
```

출력 예시:

브람이(가) 아리를 공격합니다!
아리의 남은 HP: 60

정리하기

클래스 생성과 상속 요약 정리

- **클래스 생성**
  - `class 클래스이름:` 으로 정의
  - `__init__()` 메서드로 속성 초기화
  - `self`는 객체 자신을 가리키며 반드시 첫 번째 인자로 사용
- **상속(Inheritance)**
  - `class 자식클래스(부모클래스):`
  - 부모 클래스의 속성과 메서드를 물려받음
  - `super()`로 부모의 메서드를 호출 가능
  - 자식 클래스에서 기능 추가/수정 가능 (오버라이딩)

클래스 생성과 상속 요약 정리

예시 구조:

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

class Dog(Animal):
    def __init__(self, name, breed):
        super().__init__(name)
        self.breed = breed
```