Python_07

객체 지향 프로그래밍

객체지향 프로그래밍이란?

객체 지향 프로그래밍은 컴퓨터 프로그래밍의 패러다임 중 하나

컴퓨터 프로그램을 명령어의 목록으로 보는 시각에서 벗어나 여러 개의 독립 단위

절차 지향 프로그래밍	객체 지향 프로그래밍
- 프로그램 전체가 유기적인 흐름으로 연결 - 기능 중심의 프로그램 - 순서가 정해져 있으므로 실행이 빠름	- 프로그램을 여러개의 독립된 객체들과 그 객체 간의 상호작용으로 파악하는 프로그래밍 방법 - 객체는 잘 만들어놓으면 계속해서 재사용이 가능 - 객체는 그 자체로 데이터와 행동이 정의됨 - 객체 단위로 모듈화시켜 개발할 수 있으므로 많은 인원이 참여하는 대규모 소프트웨어 개발 가능 - 개발 용이성, 유지 보수 편의성, 신뢰성을 바탕으로 생산성이 대폭 증가
소프트웨어 위기(Software Crisis) 하드웨어가 발전함에 따라 소프트웨어도 점점 커지고 복잡한 설계가 요구됨 하드웨어의 발전 속도를 소프트웨어의 발전 속도가 따라가지 못함	- 설계 시 많은 노력과 시간이 필요 - 실행 속도가 상대적으로 느림

OOP 기초

객체[속성(변수) , 행동(함수-메서드)]

클래스에서 정의한 것을 토대로 메모리에 할당된 것

프로그램에서 사용되는 데이터 또는 식별자에 의해 참조되는 공간을 의미

변수, 자료구조, 함수 또는 메서드가 될 수 있다.

하나의 객체는 특정 타입의 인스턴스

- 객체(object)의 특징
  - 타입(type) : 어떤 연산자와 메서드가 가능한가?
  - 속성(attribute) : 어떤 데이터를 가지는가?
  - 조작법(method) : 어떤 함스를 할 수 있는가?

```
객체(object) = 속성(attribute) + 기능(method)
```

Class 와 Instance

클래스로 만든 객체

객체의 설계도(클래스)를 가지고, 객체(인스턴스)를 생성한다.

OOP 문법

```
클래스 정의    class MyClass: pass
인스턴스 생성  my_instance = MyClass()
```

```
메서드 호출    my_instance.my_method()
속성 접근      my_instance.my_attribute
```

> **==**
> - 동등한(equal)
> - 변수가 참조하는 객체가 동등한 경우 True
> - 두 객체가 같아 보이지만 실제로 동일한 대상을 가리키고 있다고 확인해준 것은 아님
>
> **is**
> - 동일한(identical)
> - 두 변수가 동일한 객체를 가리키는 경우 True

속성

- 특정 데이터 타입/클래스의 객체들이 가지게 된 상태/데이터
- 클래스 변수 / 인스턴스 변수가 존재

인스턴수와 클래스 간의 이름 공간(namespace)

- 클래스를 정의하면, 클래스와 해당하는 이름 공간 생성
- 인스턴스를 만들면, 인스턴스 객체가 생성되고 이름 공간 생성
- 인스턴스에서 특정 속성에 접근하면, 인스턴스-클래스 순으로 탐색

OOP 변수

인스턴스 변수

- 인스턴스 변수란?
  - 인스턴스가 개인적으로 가지고 있는 속성(atrribute)
  - 각 인스턴스들의 고유한 변수
- 생성자 메서드(`__init__`)에서 `self.<name>`으로 정의
- 인스턴스가 생성된 이후 `<instance>.<name>`으로 접근 및 할당

클래스 변수

- 클래스 변수
  - 한 클래스의 모든 인스턴스가 공유하는 값을 의미
  - 같은 클래스의 인스턴스들은 같은 값을 갖게 됨
  - 클래스 선언 내부에서 정의
  - <classname>.<name>으로 접근 및 할당

```
class Person:
    count = 0
    def __init__(self, name):
        self.name = name
        Person.count += 1

person1 = Person('아이유')
person2 = Person('이찬혁')

print(Person.count) # 2
```

클래스 변수와 인스턴스 변수

- 클래스 변수를 변경할 때는 항상 클래스.클래스변수 형식으로 변경

```
class Circle():
    pi = 3.14 # 클래스 변수 정의

    def __init__(self, r):
        self.r = r # 인스턴스 변수

c1 = Circle(5)
c2 = Circle(10)

print(Circle.pi)  # 3.14
print(c1.pi)  # 3.14
print(c2.pi)  # 3.14

Circle.pi = 5

print(Circle.pi)  # 5
print(c1.pi)  # 5
print(c2.pi)  # 5

c2.pi = 3.14
print(Circle.pi)  # 5
print(c1.pi)  # 5
print(c2.pi)  # 3.14
```

OOP 메서드

메서드

- 특정 데이터 타입/클래스의 객체에 공통적으로 적용 가능한 행위
- 인스턴스 메서드

> 인스턴스 변수를 사용하거나, 인스턴스 변수에 값을 설정하는 메서드
> 클래스 내부에 정의되는 메서드의 기본
> 호출 시, 첫번째 인자로 인스턴스 자기자신(self)이 자동으로 전달됨

```
class MyClass:

    def instance_method(self, arg1. ...):

my_instance = MyClass()
my_instance.instance_method(...)
```

- self

> 인스턴스 자기자신
> 파이썬에서 인스턴스 메서드는 메서드 호출 시 첫번째 인자로 인스턴스 자신이 전달되게 설계
> - 매개변수 이름으로 self를 첫번째 인자로 정의
> - 다른 단어로 써도 작동하지만, 파이썬의 암묵적인 규칙

- 매직 메서드

> Double underscore(__)가 있는 메서드는 특수한 동작을 위해 만들어진 메서드로, 스페셜 메서드 혹은 매직 메서드라고 불림
> 객체의 특수 조작 행위를 지정(함수, 연산자)
> * 특정 상황에 자동으로 불리는 메서드

```
__str__(self), __len__(self), __it__(self,other), __le__(self,other), __eq__(self,other)
__gt__(self,other), __ge__(self,other), __ne__(self,other)
```

`__str__(self)` : 이 객체를 문자열로 표현하면 어떻게 표현할지를 지정

print 함수 등에서 객체를 출력하면 자동으로 호출되는 메서드

`__gt__(>,grater than)` : 부등호 연산자

생성자 메서드

- 인스턴스 객체가 생성될 때 자동으로 호출되는 메서드
- 인스턴수 변수들의 초기값을 설정
  - 인스턴스 생성
  - __init__메서드 자동 호출

```
class Circle:
    def __init__(self, r):
        self.r = r

    def area(self):
        return 3.14 * self.r * self.r

    def __str__(self):
        return f'[원] radius : {self.r}'

    def __gt__(self, other):
        return self.r > other.r

c1 = Circle(10)
c2 = Circle(1)

print(c1) # [원] radius : 10
print(c2) # [원] radius : 1
print(c1 > c2) # True
print(c1 < c2) # False
```

- 클래스 메서드
  - 클래스가 사용할 메서드
  - @classmethod 데코레이터를 사용하여 정의
  - 호출 시, 첫번째 인자로 클래스가 전달됨

```
class Circle:
    count = 0
    def __init__(self, r):
        self.r = r
        Circle.count += 1

    @classmethod
    def count_circle(cls):
        print(f'원의 수는 {cls.count}입니다.')

c1 = Circle(10)
c2 = Circle(1)

Circle.count_circle()
c1.count_circle()
c2.count_circle()
```

데코레이터

- 함수를 어떤 함수로 꾸며서 새로운 기능을 부여
- @데코레이터(함수명) 형태로 함수 위에 작성
- 순서대로 적용 되기 때문에 작성 순서가 중요

- 클래스 메서드와 인스턴스 메서드
  - 클래스 메서드 → 클래스 변수 O , 인스턴스 변수 X

- 인스턴스 메서드 → 클래스 변수 O , 인스턴스 변수 O

- 정적 메서드
  - 인스턴스 변수, 클래스 변수를 전혀 다루지 않는 메서드
  - 속성을 다루지 않고 단지 기능만을 하는 메서드를 정의할 때 사용
  - 인스턴스 변수, 클래스 변수 아무것도 사용하지 않을 경우 사용
    - 즉 객체 상태나 클래스 상태를 수정할 수 없음
  - @staticmethod 데코레이터를 사용하여 정의
  - 일반 함수처럼 동작하지만, 클래스의 이름공간에 귀속됨
  - 주로 해당 클래스로 한정하는 용도로 사용
  - static은 cls, self 사용 X

```
class Person:
    count = 0
    def __init__(self, name):
        self.name = name
        Person.count += 1

    @staticmethod
    def check_rich(money): #  static은 cls, self 사용 X
        return money>10000

person1 = Person('아이유')
person2 = Person('이찬혁')
print(Person.check_rich(100000)) # True 스태틱은 클래스로 접근 가능
print(person1.check_rich(100000)) # True 스태틱은 인스턴스로 접근 가능
```

💡 메서드 정리

- 인스턴스 메서드
  - 메서드를 호출한 인스턴스를 의미하는 self 매개변수를 통해 인스턴스를 조작
- 클래스 메서드
  - 클래스를 의미하는 cls 매개변수를 통해 클래스를 조작
- 스태틱 메서드
  - 클래스 변수나 인스턴스 변수를 사용하지 않는 경우에 사용
    - 객체 상태나 클래스 상태를 수정할 수 없음

```
class MyClass:
    def method(self):
        return 'instance method', self

    @classmethod
    def classmethod(cls):
        return 'class method', cls

    @staticmethod
    def staticmethod():
        return 'static method'
```