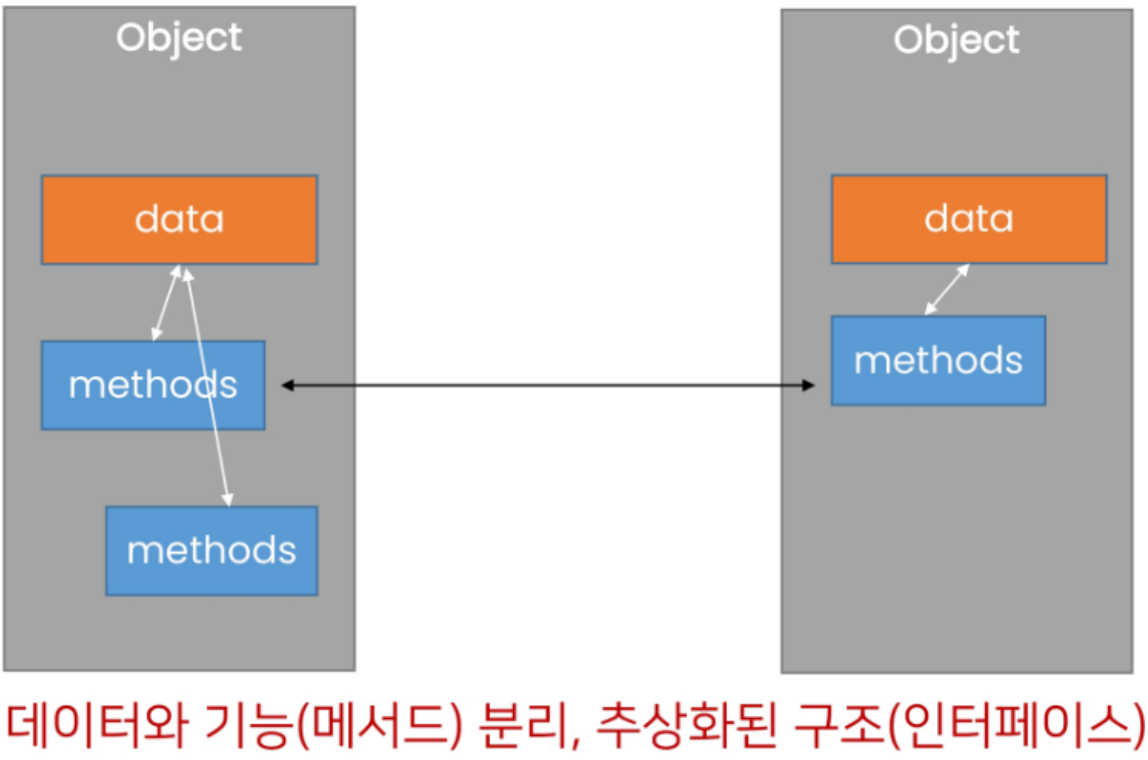
객체지향 프로그래밍(OOP)

작성일시	@2022년 7월 27일 오전 9:04
내용	OOP 파이썬
주차	2주차
자료	

객체 지향 프로그래밍

객체 지향 프로그래밍은 컴퓨터 프로그래밍의 패러다임 중 하나이다.

객체 지향 프로그래밍은 컴퓨터 프로그램을 명령어의 목록으로 보는 시각에서 벗어나 **여러 개의 독립된 단위, 즉 '객체'들의 모임으로 파악**하고자 하는 것이다. 각각의 객체는 메세지를 주고 받고, 데이터를 처리할 수 있다.



객체지향의 장점 / 단점

장점

- 클래스 단위로 모듈화시켜 개발할 수 있으므로 많은 인원이 참여하는 대규모 소프트웨어 개발에 적합
- 필요한 부분만 수정하기 쉽기 때문에 프로그램의 유지보수가 쉬움

단점

- 설계 시 많은 노력과 시간이 필요함
  - 다양한 객체들의 상호 작용 구조를 만들기 위해 많은 시간과 노력이 필요
- 실행 속도가 상대적으로 느림
  - 절차 지향 프로그래밍이 컴퓨터의 처리구조와 비슷해서 실행 속도가 빠름

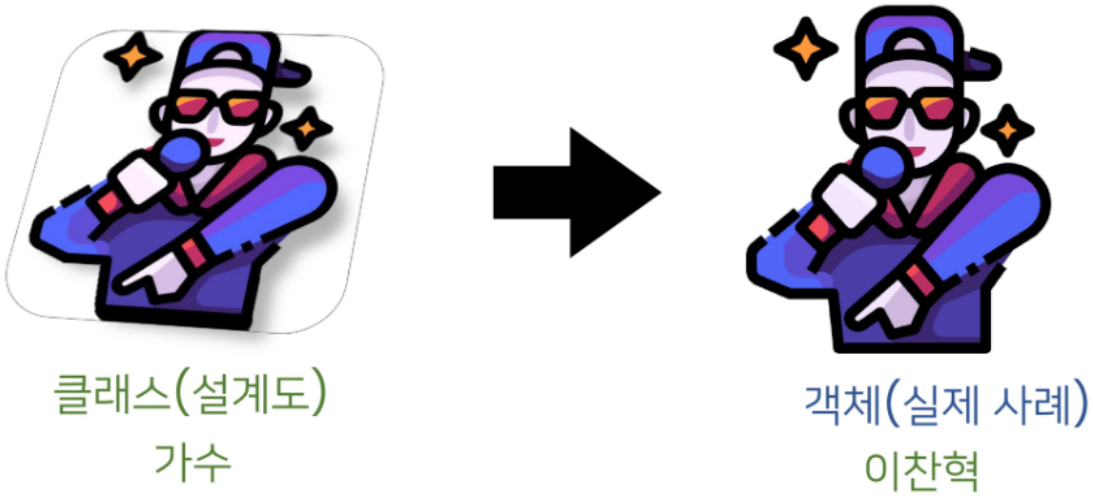
OOP 기초

객체

컴퓨터 과학에서 객체 또는 object **클래스에서 정의한 것을 토대로 메모(실제 저장공간)에 할당된 것**으로 프로그램에서 사용되는 데이터 또는 식별자에 의해 참조되는 공간을 의미하며, 변수, 자료구소, 함수 , 또는 메서드가 될 수 있다.

→ **속성(변수)과 행동(함수-메서드)으로 구성된 모든 것**

클래스와 객체



- 객체와 인스턴스
  - 클래스로 만든 객체를 인스턴스라고도 함
    - xx(타입/클래스)의 인스턴스
  - 객체(object)는 특정 타입의 인스턴스(instance)

파이썬과 객체

- 파이썬은 모든 것이 객체
  - 파이썬의 모든 것에는 속성과 행동이 존재
    - [3, 2, 1].sort()
    - 리스트.정렬()
    - 객체.행동()
    - ‘banana’.upper()
    - 문자열.대문자로()
    - 객체.행동()
- 객체(object)의 특징
  - 타입(type): 어떤 연산자(operator)와 조작(method)이 가능한가?
  - 속성(attribute): 어떤 상태(데이터)를 가지는가?
  - 조작법(method): 어떤 행위(함수)를 할 수 있는가?

객체와 클래스 문법

기본 문법

```
# 클래스 정의
class MyClass:
    pass

# 인스턴스 생성
my_instance = MyClass()

# 메서드 호출
my_instance.my_method()

# 속성
my_instance.my_attribute
```

클래스와 인스턴스

- 객체의 설계도(클래스)를 가지고, 객체(인스턴스)를 생성한다

```
class Person:
    pass

print(type(Person)) # <class 'type'>

person1 = Person()

print(isinstance(person1, Person)) # True
print(type(person1)) # <class '__main__.Person'>
```

객체 비교하기

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

```
a = [1, 2, 3]
b = [1, 2, 3]

print(a == b, a is b) # True False

a = [1, 2, 3]
b = a

print(a == b, a is b) # True True
```

속성

- 특정 데이터 타입/클래스의 객체들이 가지게 될 상태/데이터를 의미
- 클래스 변수/인스턴스 변수가 존재

```
class Person:
    blood_color = 'red' # 클래스 변수
    population = 100 # 클래스 변수

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

person1 = Person('지민')
print(person1.name) # 지민
```

인스턴스 변수

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

```
class Person:

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

john = Person('john') # 인스턴스 변수 접근 및 할당
print(john.name) # john
john.name = 'John.Kim' # 인스턴스 변수 접근 및 할당
print(john.name) # John.Kim
```

클래스 변수

- 클래스 선언 내부에서 정의
- `<classname>.<name>`으로 접근 및 할당

```
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 # 클래스.클래스 변수
```

```
print(c1.pi) # 3.14     # 인스턴스.클래스 변수
print(c2.pi) # 3.14

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

- 사용자 수 계산하기
  - 인스턴스가 생성될 때 마다 클래스 변수가 늘어나도록 설정

```
class Person:
    count = 0 # 클래스 변수
    # 인스턴스 변수 설정
    def __init__(self, name): # 생성자
        self.name = name
        Person.count += 1

person1 = Person('John')
person2 = Person('Sam')

print(Person.count) # 2
```

클래스 변수와 인스턴스 변수

- 클래스 변수를 변경할 때는 항상 **클래스.클래스변수** 형식으로 변경

OOP 메서드

메서드

- 특정 데이터 타입/클래스의 객체에 공통적으로 적용 가능한 행위(함수)

```
class Person:

    def talk(self):
        print('안녕')

    def eat(self, food):
        print(f'{food}를 냠냠')

person1 = Person()
person1.talk() # 안녕
person1.eat('피자') # 피자를 냠냠
person1.eat('치킨') # 치킨을 냠냠
```

인스턴스 메서드

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

```
class MyClass:

    def instance_method(self, arg1, _):


my_instanve = MyClass()
my_instnce.insatance_method(_)
```

- **self**
  - 인스턴스 자기 자신
  - 파이썬에서 인스턴스 메서드는 호출 시 첫번째 인자로 인스턴스 자신이 전달되도록 설계
    - 매개변수 이름으로 self를 첫 번째 인자로 정의
    - 파이썬의 암묵적인 규칙
- **생성자(constructor) 메서드**
  - 인스턴스 객체가 생성될 때 자동으로 호출되는 메서드
  - 인스턴스 변수들의 초기값을 설정
    - 인스턴스 생성
    - `__init__`메서드 자동 호출

```
class Person:

    def __init__(self):
        print('인스턴스가 생성되었습니다.')

person1 = Person() # 인스턴스가 생성되었습니다.
```

```
class Person:

    def __init__(self, name):
        print(f'인스턴스가 생성되었습니다. {name}')
```

```
person1 = Person('John') # 인스턴스가 생성되었습니다. John
```

- **매직 매서드**
  - Double underscore(__)가 있는 메서드는 특수한 동작을 위해 만들어진 메서드로, 스페셜 메서드 혹은 매직 메서드라 불림
  - 특정 상황에 자동으로 불리는 메서드

```
__str__(self) # 해당 객체의 출력 형태를 지정
__len__(self)
__repr__(self)
__lt__(self, other)
__le__(self, other)
__eq__(self, other)
__gt__(self, other) # 부등호 연산자(>, greater than)
__ge__(self, other)
__ne__(self, other)
```

- 소멸자(destructor) 메서드
  - 인스턴스 객체가 소멸(파괴)되기 전에 호출되는 메서드

```
class Person:

    def __del__(self):
        print('인스턴스가 사라졌습니다.')

person1 = Person()
del person1 # 인스턴스가 사라졌습니다.
```

클래스 메서드

- 클래스가 사용할 메서드
- `@classmethod` 데코레이터를 사용하여 정의
- 호출 시, 첫번째 인자로 클래스(cls)가 정의됨

```
class MyClass:

    @classmethod
    def classmethod(cls, arg1, _):

MyClass.class_method(_)
```

```
class Person:
    count = 0 # 클래스 변수
    # 인스턴스 변수 설정
    def __init__(self, name):
        self.name = name
        Person.count += 1

    @classmethod #데코레이터
    def number_of_population(cls):
        print(f'인구수는 {cls.count}입니다.')

person1 = Person('John')
person2 = Person('Sam')
Person.number_of_population() # 인구수는 2입니다.
```

- **데코레이터**
  - 함수를 어떤 함수로 꾸며서 새로운 기능을 부여
  - `@<데코레이터함수명>` 형태로 함수 위에 적음
  - 순서대로 적용되기 때문에 작성 순서 중요
  - 데코레이터 없이 함수 꾸미기

```
 def hello():
    print('hello')

# 데코레이팅 함수
def add_print(original): # 파라미터로 함수를 받는다.
    def wrapper(): # 함수 내에서 새로운 함수 선언
        print('함수 시작') # 부가기능 -> original 함수를 꾸민다.
        original()
        print('함수 끝') # 부가기능 -> original 함수를 꼬민다.
    return wrapper

add_print(hello)()
# 함수 시작
# hello
# 함수 끝

print_hello = add_print(hello)
print_hello()
# 함수 시작
# hello
# 함수 끝
```

  - 데코레이터를 활용하면 쉽게 여러 함수를 원하는대로 변경할 수 있음

```
def add_print(original): # 파라미터로 함수를 받는다.
    def wrapper(): # 함수 내에서 새로운 함수 선언
        print('함수 시작') # 부가기능 -> original 함수를 꾸민다.
        original()
        print('함수 끝') # 부가기능 -> original 함수를 꾸민다.
    return wrapper

@add_print #add_print를 사용해서 print_hello()함수를 꾸며주도록 하는 명령어
def print_hello():
    print('hello')

print_hello()
# 함수 시작
# hello
# 함수 끝
```

- 클래스 메서드와 인스턴스 메서드
  - 클래스 메서드 → 클래스 변수 사용(cls)
  - 인스턴스 메서드 → 인스턴스 변수 사용(self)
  - 둘 다 사용하고 싶다면?
    - 클래스는 인트선스 변수 사용이 불가능
    - 인스턴스 메서드는 클래스 변수, 인스턴스 변수 둘 다 사용이 가능
- 인스턴스와 클래스 간의 이 름 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

스태틱 메서드

- 인스턴스 변수, 클래스 변수를 전혀 다루지 않는 메서드
- 속성을 다루지 않고 단지 기능(행동)만을 하는 메서드를 정의할 때 사용
- `@staticmethod` 데코레이터를 사용하여 정의
- 일반 함수처럼 동작하지만, 클래스의 이름공간에 귀속됨
  - 주로 해당 클래스로 한정하는 용도로 사용

```
class Person:
    count = 0 # 클래스 변수
    # 인스턴스 변수 설정
```

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

    @staticmethod #데코레이터
    def check_rich(money): # 스태틱은 cls, self 사용 x
        return money > 10000

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

메서드 정리

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