OOP advanced

클래스 변수와 인스턴스 변수

클래스 변수

- 클래스의 속성입니다.
- 클래스 선언 블록 최상단에 위치합니다.
- 모든 인스턴스가 공유합니다.
- `Class.class_variable` 과 같이 접근/할당합니다.

```
class TestClass:

    class_variable = '클래스변수'
    ...

TestClass.class_variable  # '클래스변수'
TestClass.class_variable = 'class variable'
TestClass.class_variable  # 'class variable'

tc = TestClass()
tc.class_variable
# 인스턴스 ⇒ 클래스 ⇒ 전역 순서로 이름공간을 탐색하기 때문에, 접근하게 됩니다.
```

인스턴스 변수

- 인스턴스의 속성입니다.
- 각 인스턴스들의 고유한 변수입니다.
- 메서드 정의에서 `self.instance_variable` 로 접근/할당합니다.
- 인스턴스가 생성된 이후 `instance.instance_variable` 로 접근/할당합니다.

```
class TestClass:

    def __init__(self, arg1, arg2):
        self.instance_var1 = arg1    # 인스턴스 변수
        self.instance_var2 = arg2

    def status(self):
        return self.instance_var1, self.instance_var2


tc = TestClass(1, 2)
tc.instance_var1  # 1
tc.instance_var2  # 2
tc.status()  # (1, 2)
```

In [ ]:

```
# 확인해봅시다.
```

In [1]:

```
class TestClass:
    class_var = '클래스 변수'

    def __init__(self, arg1, arg2):
        self.instance_var1 = arg1
        self.instance_var2 = arg2

    def status(self):
        return self.instance_var1, self.instance_var2
```

In [ ]:

```
# 클래스 변수에 접근/재할당해 봅시다.
```

In [2]:

```
print(TestClass.class_var)
```

```
클래스 변수
```

In [3]:

```
TestClass.class_var = 'class variable'
print(TestClass.class_var)
```

```
class variable
```

In [ ]:

```
# 인스턴스를 생성하고 확인해봅시다.
```

In [5]:

```
test_instance = TestClass('인스턴스', '변수')
print(test_instance.instance_var1)
print(test_instance.instance_var2)
```

```
인스턴스
변수
```

In [ ]:

```
# 인스턴스 변수를 재할당 해봅시다.
```

In [6]:

```
test_instance.instance_var1 = '재할당 1'
test_instance.instance_var2 = '재할당 2'
print(test_instance.instance_var1)
print(test_instance.instance_var2)
```

```
재할당 1
재할당 2
```

인스턴스 메서드 / 클래스 메서드 / 스태틱(정적) 메서드

인스턴스 메서드

- 인스턴스가 사용할 메서드 입니다.
- 정의 위에 어떠한 데코레이터도 없으면, 자동으로 인스턴스 메서드가 됩니다.
- **첫 번째 인자로 `self` 를 받도록 정의합니다. 이 때, 자동으로 인스턴스 객체가 `self` 가 됩니다.**

```
class MyClass:
    def instance_method_name(self, arg1, arg2, ... ):
        ...

my_instance = MyClass()
my_instance.instance_method_name( .. , .. )  # 자동으로 첫 번째 인자로 인스턴스(my_instance)가 들어갑니다.
```

클래스 메서드

- 클래스가 사용할 메서드 입니다.
- 정의 위에 `@classmethod` 데코레이터를 사용합니다.
- **첫 번째 인자로 클래스(`cls`) 를 받도록 정의합니다. 이 때, 자동으로 클래스 객체가 `cls` 가 됩니다.**

```
class MyClass:
    @classmethod
    def class_method_name(cls, arg1, arg2, ... ):
        ...

MyClass.class_method_name( .. , .. )  # 자동으로 첫 번째 인자로 클래스(MyClass)가 들어갑니다.
```

스태틱(정적) 메서드

- 클래스가 사용할 메서드 입니다.
- 정의 위에 `@staticmethod` 데코레이터를 사용합니다.
- 묵시적인 첫 번째 인자를 받지 않습니다. 즉, 인자 정의는 자유롭게 합니다.
- **어떠한 인자도 자동으로 넘어가지 않습니다.**

```
class MyClass:
    @staticmethod
    def static_method_name(arg1, arg2, ... ):
        ...

MyClass.static_method_name( .. , .. )  # 아무일도 자동으로 일어나지 않습니다.
```

```
class MyClass:
    sample = 0

    def instance_method(self):
        return self


    @classmethod
    def class_method(cls): # 클래스와 그 안에 있는 어떤 것을 조작하는 데 사용
        return cls.sample


    @staticmethod # 완전 독자적인 함수. class, instance 속성을 사용하지 않음
    def static_method(a, b):
        return a + b
```

In [8]:

```
class MyClass:
    def instance_method(self):
        return self


    @classmethod
    def class_method(cls): # 클래스와 그 안에 있는 어떤 것을 조작하는 데 사용
        return cls


    @staticmethod # 완전 독자적인 함수. class, instance 속성을 사용하지 않음
    def static_method(arg):
        return arg
```

In [9]:

```
mc = MyClass()
```

In [ ]:

```
# 인스턴스 입장에서 확인해 봅시다.
```

In [12]:

```
# instance는 instance method에 접근 가능함
mc.instance_method()
print(id(mc.instance_method()))
print(id(mc))
print(mc.instance_method() == mc)
```

```
1621811964232
1621811964232
True
```

In [15]:

```
# instance는 class method에도 접근 가능
mc.class_method()
print(id(mc.class_method()))
print(id(MyClass))
print(mc.class_method() == MyClass)
```

```
1621776022968
1621776022968
True
```

In [16]:

```
# instance는 static method에도 접근 가능
mc.static_method(1)
```

Out[16]:

```
1
```

정리 1 - 인스턴스와 메서드

- 인스턴스는, 3가지 메서드 모두에 접근할 수 있습니다.
- 하지만 인스턴스에서 클래스메서드와 스태틱메서드는 호출하지 않아야 합니다. **(가능하다 != 사용한다)**
  - 사용할 수 있으나, 사용해서는 안 된다!
- 인스턴스가 할 행동은 모두 인스턴스 메서드로 한정 지어서 설계합니다.

In [ ]:

```
# 클래스 입장에서 확인해 봅시다.
```

In [21]:

```
# class는 class method에 접근 가능
MyClass.class_method()
print(id(MyClass.class_method()))
print(id(MyClass))
print(MyClass.class_method() == MyClass)
```

```
1621776022968
1621776022968
True
```

In [23]:

```
# class는 static method에도 접근 가능
MyClass.static_method(1)
```

Out[23]:

```
1
```

In [26]:

```
# 결국 클래스에는 인스턴스 메서드에도 접근 가능
MyClass.instance_method() # 불가. 첫 번째 인자인 instance 객체가 존재하지 않음. TypeErro
r 발생
MyClass.instance_method(mc) # == mc.instance_method()
```

```
---------------------------------------------------------------------
---------
TypeError                                 Traceback (most recent cal
l last)
<ipython-input-26-fcfa57327ba2> in <module>
      1 # 결국 클래스에는 인스턴스 메서드에도 접근 가능
----> 2 MyClass.instance_method() # 불가. 첫 번째 인자인 instance 객체가
 존재하지 않음. TypeError 발생
      3 MyClass.instance_method(mc) # == mc.instance_method()

TypeError: instance_method() missing 1 required positional argument:
'self'
```

정리 2 - 클래스와 메서드

- 클래스는, 3가지 메서드 모두에 접근할 수 있습니다.
- 하지만 클래스에서 인스턴스메서드는 호출하지 않습니다. (가능하다 != 사용한다)
- 클래스가 할 행동은 다음 원칙에 따라 설계합니다.
  - 클래스 자체(`cls`)와 그 속성에 접근할 필요가 있다면 클래스메서드로 정의합니다.
  - 클래스와 클래스 속성에 접근할 필요가 없다면 스태틱메서드로 정의합니다.

인스턴스 / 클래스 / 스태틱 메서드 자세히 살펴보기

In [ ]:

```
# Puppy class를 만들어보겠습니다.
# 클래스 변수 num_of_dogs 통해 개가 생성될 때마다 증가시키도록 하겠습니다.
# 개들은 각자의 이름과 견종을 가지고 있습니다.
# 그리고 bark() 메서드를 통해 짖을 수 있습니다.
```

In [27]:

```
class Puppy:
    num_of_dogs = 0

    def __init__(self, name, breed):
        Puppy.num_of_dogs += 1
        self.name = name
        self.breed = breed


    def __del__(self):
        Puppy.num_of_dogs -= 1


    def bark(self):
        return 'bark bark'
```

In [ ]:

```
# 각각 이름과 견종이 다른 인스턴스를 3개 만들어봅시다.
```

In [28]:

```
puppy_1 = Puppy('가지', '말티즈')
puppy_2 = Puppy('나무', '푸들')
puppy_3 = Puppy('별이', '시츄')
```

- 클래스메서드는 다음과 같이 정의됩니다.

```
@classmethod
def methodname(cls):
    codeblock
```

In [ ]:

```
# Doggy 클래스의 속성에 접근하는 클래스메서드를 생성해 보겠습니다.
```

In [29]:

```
class Doggy:
    num_of_dogs = 0
    birth_of_dogs = 0

    def __init__(self, name, breed):
        Doggy.num_of_dogs += 1
        Doggy.birth_of_dogs += 1
        self.name = name
        self.breed = breed


    def __del__(self):
        Doggy.num_of_dogs -= 1


    def bark(self):
        return 'bark bark'


    @classmethod
    def get_status(cls):
        return f'Birth: {cls.birth_of_dogs} / Current: {cls.num_of_dogs}'
```

In [ ]:

```
# Dog 3 마리를 만들어보고,
```

In [30]:

```
doggy_1 = Doggy('가지', '말티즈')
doggy_2 = Doggy('나무', '푸들')
doggy_3 = Doggy('별이', '시츄')
```

In [ ]:

```
# 함수를 호출해봅시다.
```

In [33]:

```
doggy_1.name
```

Out[33]:

'가지'

In [35]:

```
Doggy.num_of_dogs
```

Out[35]:

3

In [36]:

```
Doggy.get_status()
```

Out[36]:

```
'Birth: 3 / Current: 3'
```

- 스태틱메서드는 다음과 같이 정의됩니다.

```
@staticmethod
def methodname():
    codeblock
```

In [ ]:

```
# Dog 에 어떠한 속성에도 접근하지 않는 스태틱메서드를 만들어보겠습니다.
```

In [37]:

```
class Dog:
    num_of_dogs = 0
    birth_of_dogs = 0

    def __init__(self, name, breed):
        Dog.num_of_dogs += 1
        Dog.birth_of_dogs += 1
        self.name = name
        self.breed = breed


    def __del__(self):
        Dog.num_of_dogs -= 1


    def bark(self):
        return 'bark bark'


    @classmethod
    def get_status(cls):
        return f'Birth: {cls.birth_of_dogs} / Current: {cls.num_of_dogs}'


    @staticmethod
    def info():
        return '이거슨 댕댕이 입니다!'

dog = Dog('초코', '푸들')

# 인스턴스 method
dog.bark()

# class method
Dog.get_status()

# static method
Dog.info()
```

Out[37]:

```
'이거슨 댕댕이 입니다!'
```

In [ ]:

```
# Dog 3 마리를 만들어보고,
```

In [38]:

```
dog_1 = Dog('가지', '말티즈')
dog_2 = Dog('나무', '푸들')
dog_3 = Dog('별이', '시츄')
```

In [ ]:

```
# 함수를 호출해봅시다.
```

In [39]:

```
print(Dog.info())
print(dog_1.name)
dog_2.bark()
```

이거슨 댕댕이 입니다!
가지

Out[39]:

```
'bark bark'
```

실습 - 스태틱(정적) 메서드

> 계산기 class인 `Calculator` 를 만들어봅시다.

- 다음과 같이 정적 메서드를 구성한다.
- 모든 정적 메서드는, 두 수를 받아서 각각의 연산을 한 결과를 리턴한다.
- `a` 연산자 `b` 의 순서로 연산한다. (`a - b` , `a / b`)

1. `add(a, b)` : 덧셈
2. `sub(a, b)` : 뺄셈
3. `mul(a, b)` : 곱셈
4. `div(a, b)` : 나눗셈

In [ ]:

```
# 아래에 코드를 작성해주세요.
```

In [40]:

```
class Calculator:
    @staticmethod
    def add(a, b):
        return a + b


    @staticmethod
    def sub(a, b):
        return a - b


    @staticmethod
    def mul(a, b):
        return a * b


    @staticmethod
    def div(a, b):
        return a / b
```

In [ ]:

```
# 정적 메서드를 호출해보세요.
```

In [41]:

```
Calculator.add(1, 2)
```

Out[41]:

3

연산자 오버로딩 (중복 정의)

> operator overloading

- 파이썬에 기본적으로 정의된 연산자를 직접적으로 정의하여 활용할 수 있습니다.
- 몇 가지만 소개하고 활용해봅시다.

```
+  __add__
-  __sub__
*  __mul__
<  __lt__
⩽  __le__
=  __eq__
≠  __ne__
⩾  __ge__
>  __gt__
```

In [ ]:

```
# 사람과 사람을 같은지 비교하면, 이는 나이가 같은지 비교한 결과를 반환하도록 만들어봅시다.
```

In [1]:

```
class Person:
    population = 0

    def __init__(self, name, age):
        Person.population += 1
        self.name = name
        self.age = age


    def greeting(self):
        print(f'{self.name} 입니다. 반갑습니다!')


    def __gt__(self, other):
        if self.age > other.age:
            return f'내가 {other.name} 보다 나이가 많아용'
        else:
            return f'내가 {other.name} 보다 어려용'
```

In [2]:

```
# 연산자를 호출해 봅시다.
```

In [3]:

```
young = Person('젊은이', 40)
old = Person('노인', 100)
```

In [4]:

```
print(old > young)
```

내가 젊은이 보다 나이가 많아용

In [5]:

```
print(young > old)
```

내가 노인 보다 어려용

상속

기초

- 클래스에서 가장 큰 특징은 '상속' 기능을 가지고 있다는 것입니다.
- 부모 클래스의 모든 속성이 자식 클래스에게 상속 되므로 코드 재사용성이 높아집니다.

```
class DerivedClassName(BaseClassName):
    code block
```

In [ ]:

```
# 인사만 할 수 있는 간단한 Person 클래스를 만들어봅시다.
```

In [ ]:

```
# 학생을 만들어봅시다.
```

In [ ]:

```
# 부모 클래스에 정의된 메서드를 호출 할 수 있습니다.
```

- 이처럼 상속은 공통된 속성이나 메서드를 부모 클래스에 정의하고, 이를 상속받아 다양한 형태의 사람들을 만들 수 있습니다.

In [ ]:

```
# 진짜 상속관계인지 확인해봅시다.
```

super()

- 자식 클래스에 메서드를 추가로 구현할 수 있습니다.
- 부모 클래스의 내용을 사용하고자 할 때, `super()` 를 사용할 수 있습니다.

```
class BabyClass(ParentClass):
    def method(self, arg):
        super().method(arg)
```

- 위의 코드를 보면, 상속을 했음에도 불구하고 동일한 코드가 반복됩니다.

In [ ]:

```
# 이를 수정해봅시다.
```

메서드 오버라이딩

> method overriding

- 메서드를 재정의할 수도 있습니다.
- 상속 받은 클래스에서 메서드를 덮어씁니다.

In [ ]:

```
# Person 클래스의 상속을 받아 군인처럼 인사하는 Soldier 클래스를 만들어봅시다.
```

상속관계에서의 이름공간

- 기존에 인스턴스 -> 클래스 순으로 이름 공간을 탐색해나가는 과정에서 상속관계에 있으면 아래와 같이 확장됩니다.
- 인스턴스 -> 클래스 -> 전역
- 인스턴스 -> 자식 클래스 -> 부모 클래스 -> 전역

실습 1

> 사실 사람은 포유류입니다.
>
> Animal Class를 만들고, Person클래스가 상속받도록 구성해봅시다.
>
> 변수나, 메서드는 자유롭게 만들어봅시다.

In [ ]:

```
# 아래에 코드를 작성해주세요.
```

다중 상속

두개 이상의 클래스를 상속받는 경우, 다중 상속이 됩니다.

In [ ]:

```
# Person 클래스를 정의합니다.
```

In [ ]:

```
# Mom 클래스를 정의합니다.
```

In [ ]:

```
# Dad 클래스를 정의합니다.
```

In [ ]:

```
# FirstChild 클래스를 정의합니다.
```

In [ ]:

```
# FirstChild 의 인스턴스 객체를 확인합니다.
```

In [ ]:

```
# cry 메서드를 실행합니다.
```

In [ ]:

```
# swim 메서드를 실행합니다.
```

In [ ]:

```
# walk 메서드를 실행합니다.
```

In [ ]:

```
# gene 은 누구의 속성을 참조할까요?
```

In [ ]:

```
# 그렇다면 상속 순서를 바꿔봅시다.
```

In [ ]:

```
# SecondChild 의 인스턴스 객체를 확인합니다.
```

In [ ]:

```
# cry메서드를 실행합니다.
```

In [ ]:

```
# walk 메서드를 실행합니다.
```

In [ ]:

```
# swim 메서드를 실행합니다.
```

In [ ]:

```
# gene 은 누구의 속성을 참조할까요?
```