Classese & Object Oriented Programming

해당 단원에서 다루는 내용들:

- 객체의 설계도 역할을 하는 클래스(class)를 이용하는 방법
- 상속(inheritance)를 이용해 클래스들을 계층적으로 표현하고, 다형성(polymorphism)을 가지도록 하는 방법

Introduction: Why Class?

클래스(class)는 데이터와 데이터에 대한 연산을 하나로 묶은 것입니다. 즉, 변수와 함수를 묶은 것입니다. 묶는 이유는, 묶으면 코드를 읽기 쉬워지며, 유지관리하기 좋아지기 때문입니다.

예를 들어, 자동차들의 정보를 관리하는 프로그램을 만든다고 해 봅시다. 각 자동차에 대해 색깔과 현재 속도라는 데이터를 관리하고, 각 자동차의 속도를 증가시키거나 감소시킬 수 있어야 합니다. 이를 파이썬 코드로 간단하게 나타내어보면 다음과 같습니다.

```
car_k5 = {
    "name": "k5",
    "color": "black",
    "speed": 100
}
car_ionic5 = {
    "name": "ionic 5",
    "color": "grey",
    "speed": 50
}

def speed_up(car, value):
    car["speed"] += value

def speed_down(car, value):
    car["speed"] -= value

def describe(car):
    print(f"차종: {car['name']}")
    print(f"색상: {car['color']}")
    print(f"속도: {car['speed']}")

speed_up(car_k5, 50)
```

위 프로그램에서는 데이터와 데이터에 대한 연산이 각각 변수(`car_k5` , `car_ionic5`)와 해당 변수를 인자로 받는 함수(`speed_up` , `speed_down` , `describe`)로 나타내어졌습니다. 클래스의 핵심 아이디어는 데이터와 이들에 대한 연산은 강한 관계를 가지므로, 이것들을 하나로 묶어주자는 것입니다.

위 코드를 클래스를 사용하도록 변경한 코드는 다음과 같습니다.

```
class Car:
    def __init__(self, name, color, speed):
        self.name = name
        self.color = color
        self.speed = speed
```

```
    def speed_up(self, value):
        self.speed += value

    def speed_down(self, value):
        self.speed -= value

    def describe(self):
        print(f"차종: {self.name}")
        print(f"색상: {self.color}")
        print(f"속도: {self.speed}")

car_k5 = Car("k5", "black", 100)
car_ionic5 = Car("ionic 5", "grey", 50)

car_k5.speed_up(50)
```

이와 같이 변수와 함수를 클래스로 묶어 클래스로 나타낼 수 있습니다.

클래스와 인스턴스

- 클래스(class)는 데이터와 데이터에 대한 연산을 하나로 묶은 것입니다.
- 클래스는 객체의 설계도 역할을 합니다.
  - 객체(object)란, 파이썬에서 만질(?) 수 있는 모든 것들을 의미합니다.
- 앞서 든 예시에서는 `Car` 이라는 클래스를 이용해서 `car_k5` 와 `car_ionic5` 라는 객체를 만들었습니다. 이들과 같이, 클래스로부터 만들어진 객체를 인스턴스(instance)라고 합니다.
- 인스턴스는 인스턴스 변수(instance variable)과 메소드(method)들 가집니다.
  - 인스턴스 변수: 인스턴스가 가지는 변수
  - 메소드: 인스턴스가 가지는 함수

클래스 정의(class definition)

```
class ClassName:
    <statement 1>
    <statement 2>
    ...
```

`<statement n>` 자리에는 보통 메소드의 정의, 그리고 클래스 변수의 정의가 옵니다.

메소드 정의

```
class Car:
    def speed_up(self, value):

car_k5.speed_up(50)
```

함수를 정의할때와 같은 방법으로 정의합니다. 단, 메소드는 첫번째 인자로 메소드를 사용한 인스턴스를 인자로 추가로 받습니다. 즉, `car_k5.speed_up(50)` 을 호출 시, `speed_up` 메소드는 첫번째 인자로 `car_k5` 인스턴스를 받고, 두 번째 인자로 `50` 을 받습니다. 일반적으로 메소드의 첫번째 인자에 `self` 라는 이름을 사용합니다.

클래스 객체(class object) & 클래스 변수(class variable)

클래스 그 자체도 별도의 객체입니다! 클래스 정의가 끝나면, 해당 클래스의 이름을 가지는 클래스 객체가 생성됩니다.

클래스 객체는 클래스 변수(class variable)이라는 변수를 가질 수 있습니다. 클래스 변수는 변수 정의를 클래스 안에 써주면 정의할 수 있습니다.

```
class MyClass:
    magic = 42

print(MyClass.magic) # 42
```

클래스 객체를 호출하면 클래스의 인스턴스가 생성됩니다.

```
myClass = MyClass()
```

클래스로부터 만들어진 인스턴스도 클래스 변수에 접근할 수 있습니다. 이때, 모든 인스턴스와 클래스 객체는 클래스 변수를 공유한다는 것에 유의해야 합니다.

```
print(myClass.magic)

myClass.magic = 100
print(MyClass.magic)
print(myClass.magic)
```

```
42
100
100
```

`__init__` 메소드

```
class Car:
    def __init__(self, name, color, speed):
        self.name = name
        self.color = color
        self.speed = speed
    ...
```

파이썬에서 특수한 역할을 하는 메소드들은 언더바 두개(`__`)를 앞/뒤에 붙여 표시합니다. 그중에서 `__init__` 메소드는 인스터스가 생성되면 자동으로 호출되는 특수 메소드입니다. `__init__` 메소드의 역할은 인스턴스의 상태를 초기화하는 것으로, 보통 인스턴스 변수들을 정의하기 위해서 사용합니다.

연산자 관련 메소드

`__add__()` 메소드를 정의하면 더하기(`+`) 연산자를, `__sub__()` 메소드를 정의하면 빼기(`-`) 연산자를 클래스의 인스턴스에 대해서 사용할 수 있습니다.

```
class Account:
    def __init__(self, balance):
        self.balance = balance

    def __add__(self, other):
        return Account(balance + other.balance)

acc1 = Account(100)
acc2 = Account(300)
acc3 = acc1 + acc2
print(acc3.balance) # 400
```

연산	호출되는 함수
a + b	a.__add__(b)
a - b	a.__sub__(b)
a * b	a.__mul__(b)
a / b	a.__truediv__(b)
a // b	a.__floordiv__(b)
a % b	a.__mod__(b)
a ** b	a.__pow__(b)
a & b	a.__and__(b)
a | b	a.__or__(b)
a ^ b	a.__xor__(b)
-a	a.__neg__()
~a	a.__invert__()

연산	호출되는 함수
a += b	a.__iadd__(b)
a -= b	a.__isub__(b)
a *= b	a.__imul__(b)
a /= b	a.__itruediv__(b)
a //= b	a.__ifloordiv__(b)
a %= b	a.__mod__(b)
a **= b	a.__ipow__(b)
a &= b	a.__iand__(b)
a |= b	a.__ior__(b)
a ^= b	a.__ixor__(b)

연산	호출되는 함수
a < b	a.__lt__(b)
a <= b	a.__le__(b)
a == b	a.__eq__(b)
a != b	a.__ne__(b)
a > b	a.__gt__(b)
a >= b	a.__ge__(b)

연산	호출되는 함수
a[key]	a.__getitem__(key)
a[key]	a.__setitem__(key)
len(a)	a.__len__()
print(a)	a.__str__()

체크

다음 코드의 빈칸을 채워보세요.

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

    def deposit(self, amount):
        self.balance += amount

    def withdraw(self, amount):
        if self.balance - amount >= 0:
            _____
        else:
            print("잔액 부족!")
            return False

    def print_info(self):
        print("이름: %s, 잔고: %d" % (self.name, self.balance))

    def transfer(self, other, amount):
        _____

acc1 = BankAccount("철수", 1000)
acc2 = BankAccount("영희", 500)
acc2.transfer(acc1, 600)
acc2.transfer(acc1, 300)

acc1.print_info()
acc2.print_info()
```

출력:

```
잔액 부족!
이름: 철수, 잔고: 1300
이름: 영희, 잔고: 200
```

2

1. 다음 코드의 출력을 예상해보세요.
2. 다음 코드가 작성자가 의도한대로 동작할까요? 그렇지 않다고 생각하면 잘못된 부분을 고쳐보세요.

```
class Pair:
    c = 0

    def __init__(self, x, y):
        self.x = x
        self.y = y
        Pair.c += 1

    def __add__(self, other):
        self.x += other.x
        self.y += other.y

    def __str__(self):
        return f"({self.x}, {self.y})"

p1 = Pair(10, 20)

p2 = Pair(30, 40)

p3 = p1 + p2
print(p1)
print(p2)
print(p3)
print(p3.c)
```

상속(inheritance)

클래스를 이용해 프로그램을 작성하다 보면 여러 클래스가 공통적으로 가지는 변수나 메소드가 나타나기 시작합니다. 어떻게 하면 이러한 중복을 해결할 수 있을까요? 이를 해결하는 한 방법은 상속을 이용하는 것입니다.

상속(Inheritance)는 여러 클래스의 공통된 부분을 새로운 클래스로 분리하고, 기존 클래스들이 새로운 클래스로부터 변수와 메소드들을 상속받도록 해 중복을 제거합니다. 이때 분리된 클래스를 부모(parent) 또는 슈퍼클래스(superclass)라고 하며, 부모로부터 메소드들과 변수를 상속받는 클래스들을 자식(child) 또는 서브클래스(subclass)라고 합니다. 따라서 슈퍼클래스와 서브클래스는 부모-자식과 같은 관계를 가지게 되며, 그림으로 나타낼때는 마치 가계도와 같이 나타냅니다.

예를 들어, 앞서 예시로 든 `Car` 클래스와 동일한 가속/감속 기능을 가지는 두 클래스 `Sedan`과 `Truck`을 만들었다고 해 봅시다.

```
class Sedan:
    def __init__(self, name, color, speed, seat_num):
        self.name = name
        self.color = color
        self.speed = speed
        self.seat_num = seat_num

    def speed_up(self, value):
        self.speed += value

    def speed_down(self, value):
```

```
        self.speed -= value

    def describe(self):
        print(f"차종: {self.name}")
        print(f"색상: {self.color}")
        print(f"속도: {self.speed}")
        print(f"좌석 수: {self.seat_num}")

    def get_seat_num(self):
        return self.seat_num

class Truck:
    def __init__(self, name, color, speed, capacity):
        self.name = name
        self.color = color
        self.speed = speed
        self.capacity = capacity

    def speed_up(self, value):
        self.speed += value

    def speed_down(self, value):
        self.speed -= value

    def describe(self):
        print(f"차종: {self.name}")
        print(f"색상: {self.color}")
        print(f"속도: {self.speed}")
        print(f"총중량: {self.capacity}")
```

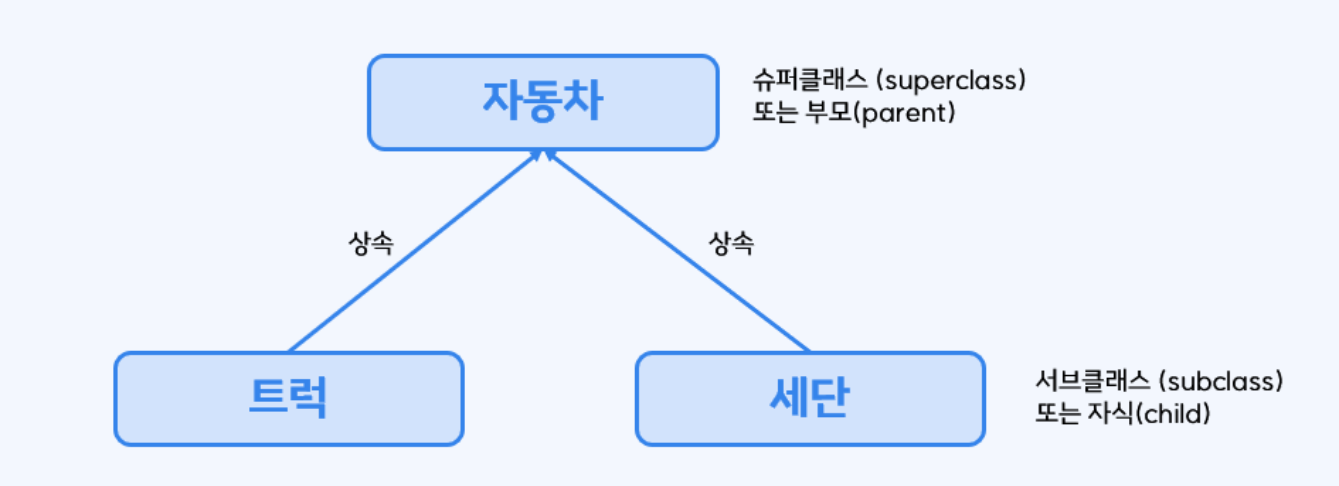
위 두 클래스는 거의 대부분의 부분이 중복되는 걸 볼 수 있습니다. 앞서 만든 `Car` 클래스로부터 상속을 받도록 해 중복을 제거해봅시다.

```
class Sedan(Car):
    def __init__(self, name, color, speed, seat_num):
        super().__init__(name, color, speed)
        self.seat_num = seat_num

    def describe(self):
        super().describe()
        print(f"좌석 수: {self.seat_num}")

    def get_seat_num(self):
        return self.seat_num

class Truck(Car):
    def __init__(self, name, color, speed, capacity):
        super().__init__(name, color, speed)
        self.capacity = capacity

    def describe(self):
        super().describe()
        print(f"총중량: {self.capacity}")
```

`Car` 로부터 상속받은 클래스들은 `Car` 의 메소드들을 모두 상속받습니다. 따라서 `Sedan` 과 `Truck` 클래스의 인스턴스는 `speed_up()` , `speed_down()` 등 `Car` 클래스에서 정의된 메소드들을 그대로 사용할 수 있습니다.

한편, 서브클래스에서 슈퍼클래스와 동일한 메소드를 정의하면 그 메소드는 새로운 정의로 덮어씌어집니다. 이를 메소드 오버라이딩(method overriding)이라고 합니다. 그리고 덮어쓰는 메소드 안에서 `super(),<메소드 이름>`을 호출해 슈퍼클래스에서 정의된 메소드에 접근할 수 있습니다. `Sedan`과 `Truck` 클래스의 `__init__()`과 `describe()`는 메소드 오버라이딩을 이용해 해당 메소드가 작동하는 방식을 변경합니다.

`Car`, `Sedan`과 `Truck` 클래스의 관계를 그림으로 나타내면 다음과 같습니다.



체크

다음 코드는 정상적으로 작동하지 않습니다. 오류의 원인을 찾고, 올바르게 코드를 수정해보세요.

```
class Track:
    def __init__(self, name, artist, duration):
        self.name = name
        self.artist = artist
        self.duration = duration

    def __str__(self):
        return f"{self.artist} - {self.name} | {self.duration}s"

class OnlineTrack:
    def __init__(self, name, artist, duration, likes):
        self.likes = likes

    def __str__(self):
        return super().__str__()

track = OnlineTrack("비밀번호 486", "윤하", 316, 194721)
print(track)
```

다형성(polymorphism)

다형성(polymorphism)이란, 서로 다른 타입들을 동일한 타입인것처럼 사용할 수 있는 성질을 의미합니다. 파이썬에서는 클래스의 상속을 이용해서 객체들이 다형성을 가지도록 할 수 있습니다.

is-a 관계

- '사람'이라는 클래스가 있고, 해당 클래스의 서브클래스로 '남자'와 '여자'가 있다고 해 봅시다.
- 이때 `'남자'는 '사람'이다`라는 명제는 옳은 명제입니다. 왜냐하면 남자는 사람의 정의를 모두 만족하기 때문입니다.

- 따라서 `'사람'은 말을 할 수 있다`라는 명제에서 '사람'대힌 '남자'가 와도, '여자'가 와도 모두 옳습니다.
- 이러한 관계를 'is-a'(~는 ~이다) 관계라고 합니다. 서브클래스와 슈퍼클래스는 'is-a' 관계를 가지는 것입니다.

상속과 다형성

서브클래스와 슈퍼클래스가 'is-a' 관계를 가진다는 것은, 서브클래스의 인스턴스를 슈퍼클래스의 인스턴스처럼 사용할 수 있다는 것입니다. 따라서 서브클래스의 인스턴스들은 다형성을 지닌다고 말할 수 있습니다.

예를 들어서, 주차장을 나타내는 클래스 `ParkingLot`을 만든다고 해 봅시다. 이 클래스는 `Car` 클래스의 인스턴스를 주차하고, 주차된 모든 자동차에 대한 설명을 출력하는 기능을 가지고 있습니다.

```
class ParkingLot:
    def __init__(self):
        self.cars = []

    def park(self, car):
        self.cars.append(car)

    def describe(self):
        for c in cars:
            c.describe()
            print() # 빈 줄 출력
```

이 주차장 클래스는 `Car` 클래스의 인스턴스를 주차시킬 수 있습니다. 그런데 `Car` 클래스의 서브클래스들인 `Sedan`과 `Truck`도 `Car`이기 때문에 이들 클래스의 인스턴스 또한 주차시킬 수 있습니다.

```
parking_lot = ParkingLot()

sedan = Sedan("k5", "green", 100, 4)
truck = Truck("porter", "blue", 50, 1000)

parking_lot.park(sedan)
parking_lot.park(truck)
parking_lot.describe()
```

```
차종: k5
색상: green
속도: 100
좌석 수: 4

차종: porter
색상: blue
속도: 50
총중량: 1000
```

위 코드에서는 `Sedan`과 `Truck`의 인스턴스를 `Car` 클래스의 인스턴스로 여기고, `describe()` 메소드를 호출하는 것을 볼 수 있습니다. 이때, 실제 출력이 서로 다르다는 것에 주목해 봅시다. 이는 `Sedan`과 `Truck` 클래스가 메소드 오버라이딩을 통해서 `describe()` 메소드의 동작을 바꿔주었기 때문입니다. 이처럼 클래스의 인스턴스를 '사용하는' 쪽에서는 동일한 클래스인것처럼 사용할 수 있지만, 실제 동작은 다를 수도 있다는 것이 다형성이란 성질의 핵심입니다. 쉽게 말하면 겉은 같고 속은 다르다는 것이죠. 그렇기 때문에, 다형성을 **"상속 관계 내의 서로 다른 클래스들의 인스턴스들이 같은 멤버 호출에 대해 각각 다르게 반응하도록 하는 기능"**이라고도 할 수 있습니다.

Why Polymorphism?

- 다형성을 활용하면 코드의 중복을 줄일 수 있습니다. 앞에서 주차장 클래스를 만들었을 때 세단을 위한 주차장, 트럭을 위한 주차장을 따로 만들어줄 필요가 없었습니다.
- 또한, 코드의 가독성이 높아지고 유지관리성이 좋아진다는 장점도 있습니다.

체크

다음 코드는 `Shape` 클래스와 해당 클래스의 서브클래스들인 `Rectangle`, `Triangle`들을 구현한 것입니다. 빈칸에 들어갈 코드를 출력을 보고 채워보세요.

```
class Shape:
    def area(self):
        pass
    def draw(self):
        pass

class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height
    def area(self):
        return self.width * self.height
    def draw(self):
        _____

class Triangle(Shape):
    def __init__(self, base, height):
        self.base = base
        self.height = height
    def area(self):
        return 0.5 * self.base * self.height
    def draw(self):
        print(f"Drawing a triangle with base {self.base} and height {self.height}.")

shapes = [Rectangle(2, 3), Triangle(4, 5)]

for shape in shapes:
    _____
```

출력:

```
6
Drawing a 2x3 rectangle.
10
Drawing a triangle with base 4 and height 5.
```

Additional Notes

인스턴스 변수와 클래스 변수를 구분합시다

수업에 사용되는 PPT를 보면 종종 인스턴스 변수가 아닌 클래스 변수를 사용하는 예시 코드를 볼 수 있습니다. 하지만, **합당한 이유가 없다면 인스턴스 변수를 사용합시다**. 클래스 변수는 모든 클래스 인스턴스 사이에 공유되기 때문에, 예상치 못한 오류를 발생시킬 수 있습니다:

```
class Room:
    students = []
    def add_student(self, name):
        students.append(name)

roomA = Room()
roomB = Room()

roomA.add_student("alice")
roomB.add_student("bob")
print(roomA.students) # ["alice", "bob"]
print(roomB.students) # ["alice", "bob"]
```

클래스 변수는 상수를 선언하거나, 인스턴스 사이에 공유해야 할 값이 있을때만 사용하고, 나머지 상황에서는 인스턴스 변수를 사용하는 것이 적절합니다.

정답

클래스와 인스턴스

1

첫번째 빈칸:

```
    self.balance -= amount
    return True
```

두번째 빈칸:

```
    if self.withdraw(amount):
        other.deposit(amount)
```

2

출력:

```
    (10, 20)
    (30, 40)
    None
```

`__add__()` 메소드는 피연산자들의 인스턴스 변수를 변경하지 않고, 합을 표현하는 새로운 인스턴스를 반환해야 합니다. 따라서 해당 메소드를

```
    def __add__(self, other):
        return Pair(self.x + other.x, self.y + other.y)
```

로 고쳐야 합니다. ### 상속

1. `OnlineTrack` 클래스가 `Track` 클래스를 상속하지 않습니다.
2. `OnlineTrack.__init__()` 메소드가 슈퍼클래스의 `__init__()` 메소드를 호출하지 않습니다.

다형성
첫번째 빈칸:

```
    print(f"Drawing a {self.width}x{self.height} rectangle.")
```

두번째 빈칸:

```
    print(shape.area())
    shape.draw()
```