객체와 클래스

클래스 선언과 객체 생성

객체란 = 개념

- 객체는 속성(상태, 특징)과 행위(행동, 동작, 기능)로 구성된 대상을 의미
- 객체는 자동차나 로봇 같은 사물일 수도 있고 사람이나 동물일 수 있으며 어떤 개념일 수도 있음
- 프로그래밍 언어에서 객체를 만들 때는 주로 현실 세계를 반영
- 객체의 특징인 속성은 변수로, 객체가 할 수 있는 일인 행동은 함수로 구현
- 객체는 변수와 함수의 묶음

클래스 선언과 객체 생성

객체란

- 사람이라면 이름, 키, 몸무게 같은 속성은 변수로 구현하고 걷거나 뛰거나 앉는 행동은 함수로 구현
- 객체가 자전거라면 바퀴의 크기, 색깔 같은 속성은 변수로 구현하고 전진, 방향 전환, 정지 같은 동작은 함수로 구현
- 객체를 만들고 이용할 수 있는 기능을 제공하는 프로그래밍 언어를 객체지향 프로그래밍(Object- Oriented Programming, OOP) 언어 혹은 객체지향 언어라고 함

클래스 선언과 객체 생성

클래스 선언

- 객체를 만들려면 먼저 클래스를 선언해야 함
- 클래스는 객체의 공통된 속성과 행위를 변수와 함수로 정의한 것
- 클래스는 객체를 만들기 위한 기본 틀이고 객체는 기본 틀을 바탕으로 만들어진 결과
- 객체는 클래스에서 생성하므로 객체를 클래스의 인스턴스(Instance)라고 함

클래스 선언과 객체 생성

클래스 선언

- 클래스 선언을 위한 기본 구조

```
class 클래스명():
    [변수1] #클래스변수
    [변수2]
    def 함수1(self[, 인자1, 인자2, • • , 인자n]): # 클래스 함수
        〈코드 블록〉
    def 함수2(self[, 인자1, 인자2, • • , 인자n]):
        〈코드 블록〉
```

클래스 선언과 객체 생성

클래스 선언

- 클래스를 선언할 때 class 키워드 다음에 클래스명, 소괄호 , 콜론(:)을 순서대로 입력
- 클래스명 (클래스 이름)은 보통 로마자 알파벳 대문자로 시작하며 여러 단어가 연결된 클래스 이름은 가독성을 위해 대문자로 시작하는 단어를 연결해 클래스 이름을 만듬
- 클래스 내에서 변수를 선언하고 'def 함수():' 형태로 함수를 작성
- 클래스명 다음 줄에 오는 모든 코드는 들여쓰기 해야 함
- 클래스에서 정의한 함수의 첫 번째 인자는 self
- self는 객체 생성 후에 자신을 참조하는데 이용
- 대괄호([ ]) 안에 있는 인자는 필요한 만큼 사용할 수 있으며 , 필요 없으면 생략할 수 있음

클래스 선언과 객체 생성

객체 생성 및 활용

- 사용할 클래스는 자전거 클래스
- 자전거 클래스를 만들기 전에 우선 자전거가 갖는 속성과 동작을 정의
- 자전거의 속성: 바퀴 크기(wheel_size), 색상(color)
- 자전거의 동작 지정된 속도로 이동(move), 좌/우회전(turn), 정지(stop)
- 자전거의 속성과 동작을 바탕으로 자전거 클래스를 만듬
- 자전거 클래스를 선언하고 객 체를 생성한 후 클래스에 변수와 함수를 추가해서 클래스를 완성

클래스 선언과 객체 생성

객체 생성 및 활용

- 자전거 클래스를 선언
- 클래스를 단순화하기 위해 클래스명이 Bicycle인 자전거 클래스의 원형만 선언
- 클래스에는 클래스명(Bicycle)만 있고, 코드 부분에는 Pass만 있어서 실제로는 아무 일도 일어나지 않음
- Bicycle 클래스에는 변수도 함수도 없지만 이것도 엄연한 클래스

```
[ ] class Bicycle(): # 클래스 선언
        pass
```

클래스 선언과 객체 생성

객체 생성 및 활용

- 선언된 클래스로부터 클래스의 인스턴스인 객체를 생성하는 방법
- 객체명 = 클래스명()
- 클래스명()의 클래스는 앞에서 미리 선언돼 있어야 함
- 객체명은 변수명을 만들 때와 같은 규칙을 적용해서 만듬

클래스 선언과 객체 생성

객체 생성 및 활용

- 정의한 Bicycle 클래스의 객체는 생성할 수 있음

```
[ ] my_bicycle = Bicycle()
```

- 선언한 Bicycle 클래스에는 변수도 없고 함수도 없으므로 아직은 어떤 작업도 수행할 수 없지만 my_bicyde 객체는 Bicycle 클래스의 인스턴스
- 객체를 실행하면 객체의 클래스와 객체를 생성할 때 할당받은 메모리의 주소값을 출력함

```
[ ] my_bicycle
```

클래스 선언과 객체 생성

객체 생성 및 활용

- 객체에 속성을 설정하려면 '객체명.변수명'에 '속성값'을 할당

  객체명.변수명 = 속성값

- 생성한 my_bicycle 객체에 속성값을 설정

```
[ ]  my_bicycle.wheel_size = 26
     my_bicycle.color = 'black'
```

클래스 선언과 객체 생성

객체 생성 및 활용

- 객체의 변수에 접근해서 객체의 속성을 가져오는 방법

  객체명.변수명

- 객체의 속성값을 가져와서 출력

```
[ ] print("바퀴 크기:", my_bicycle.wheel_size)  # 객체의 속성 출력
    print("색상:", my_bicycle.color)
```

클래스 선언과 객체 생성

객체 생성 및 활용

- 선언한 Bicycle 클래스에 함수를 추가

```
[ ] class Bicycle():

        def move(self, speed):
            print("자전거: 시속 {0}킬로미터로 전진".format(speed))

        def turn(self, direction):
            print("자전거: {0}회전".format(direction))

        def stop(self):
            print("자전거({0}, {1}): 정지 ".format(self.wheel_size, self.color))
```

클래스 선언과 객체 생성

객체 생성 및 활용

- Bicycle 클래스에 '지정된 속도로 이동' , '좌/우회전' , '정지' 동작을 나타내는 move(), turn(), stop() 함 수를 각각 추가
- 객체를 생성한 후에는 '객체명.변수명 = 속성값'으로 속성값을 설정하고
- '객체명.변수명'으로 속성값을 가져왔지만, 클래스의 함수 안에서는 'self. 변수명 = 속성값'으로 속성값을 설정하고 , self. 변수명'으로 속성값을 가져옴
- stop()함수에서는 self .wheel_size 와 self .color 로 객체의 속성값을 가져와서 출력

클래스 선언과 객체 생성

객체 생성 및 활용

- 객체의 메서드를 호출 할 때
- 객체명.메서드명([인자1, 인자2, • • • , 인자n])
- 메서드명은 클래스에서 정의한 함수명
- 객체에서 메서드를 호출할 때 인자는 클래스에서 정의한 함수의 인자만큼 필요
- 클래스를 선언할 때 추가했던 함수의 인자 self는 필요하지 않음
- 클래스에서 self만 인자로 갖는 함수를 객체에서 이용할 때는 소괄호 안에 인자를 지정하지 않음

클래스 선언과 객체 생성

객체 생성 및 활용

- 구현한 Bicycle 클래스에서 객체를 생성한 후에 속성을 설정하고 객체의 메서드를 호출하는 방법

```
[ ] my_bicycle = Bicycle() # Bicycle 클래스의 인스턴스인 my_bicycle 객체 생성

    my_bicycle.wheel_size = 26 # 객체의 속성 설정
    my_bicycle.color = 'black'

    my_bicycle.move(30)  # 객체의 메서드 호출
    my_bicycle.turn('좌')
    my_bicycle.stop()
```

- Bicycle 클래스에서 my_bicycle 객체를 생성한 후 속성을 설정하고 메서드를 호출
- Bicycle 클래스에서 함수를 정의할 때 self 외의 인자가 있는 move(), turn() 함수의 경우에는 객체의 메서드를 호출할 때 인자를 입력했고
- self 인자만 있는 stop() 함수의 경우에는 인자 없이 객체의 메서드를 호출

클래스 선언과 객체 생성

객체 생성 및 활용

- 클래스의 선언, 객체의 생성 및 활용 방법
- 자전거 (Bicycle) 클래스를 선언한 후 두 개의 객체(bicycle1, bicycle2)를 생성하고 활용

```
[ ] bicycle1 = Bicycle() # Bicycle 클래스의 인스턴스인 bicycle1 객체 생성

    bicycle1.wheel_size = 27 # 객체의 속성 설정
    bicycle1.color = 'red'

    bicycle1.move(20)
    bicycle1.turn('좌')
    bicycle1.stop()
```

클래스 선언과 객체 생성

객체 생성 및 활용

- 클래스의 선언, 객체의 생성 및 활용 방법

```
[ ] bicycle1 = Bicycle() # Bicycle 클래스의 인스턴스인 bicycle1 객체 생성

    bicycle1.wheel_size = 27 # 객체의 속성 설정
    bicycle1.color = 'red'

    bicycle1.move(20)
    bicycle1.turn('좌')
    bicycle1.stop()
```

```
[ ] bicycle2 = Bicycle() # Bicycle 클래스의 인스턴스인 bicycle2 객체 생성

    bicycle2.wheel_size = 24 # 객체의 속성 설정
    bicycle2.color = 'blue'

    bicycle2.move(15)
    bicycle2.turn('우')
    bicycle2.stop()
```

클래스 선언과 객체 생성

객체 초기화

- Bicycle 클래스를 선언하고 객체를 생성한 후에 객체의 속성을 설정
- 클래스를 선언할 때 초기화 함수 __init__()를 구현하면 객체를 생성하는 것과 동시에 속성값을 지정할 수 있음
- __init__() 함수는 클래스의 인스턴스가 생성될 때(즉, 객체가 생성될 때) 자동으로 실행
- __init__() 함수에 초기화하려는 인자를 정의하면 객체를 생성할 때 속성을 초기화 할 수 있음

클래스 선언과 객체 생성

객체 초기화

- Bicycle 클래스에 __init__() 함수를 추가한 코드

```
[ ] class Bicycle():

        def __init__(self, wheel_size, color):
            self.wheel_size = wheel_size
            self.color = color

        def move(self, speed):
            print("자전거: 시속 {0}킬로미터로 전진".format(speed))

        def turn(self, direction):
            print("자전거: {0}회전".format(direction))

        def stop(self):
            print("자전거({0}, {1}): 정지 ".format(self.wheel_size, self.color))
```

클래스 선언과 객체 생성

객체 초기화

- __init__(self, wheel_size, color) 함수는 wheel_size와 color를 인자로 입력받아 함수 내에서 'self.변수명 = 인자'로 객체의 속성을 초기화
- 클래스에 __init__()함수가 정의돼 있으면 객체를 생성할 때 __init__() 함수의 인자를 입력(self는 제외)함
- 객체명 = 클래스명(인자1, 인자2, 인자3, • • • , 인자n)

클래스 선언과 객체 생성

객체 초기화

- Bicycle 클래스에서 객체를 생성할 때 속성값을 지정해서 초기화하는 방법

```
[ ] my_bicycle = Bicycle(26, 'black') # 객체 생성과 동시에 속성값을 지정.

    my_bicycle.move(30)  # 객체 메서드 호출
    my_bicycle.turn('좌')
    my_bicycle.stop()
```

  - 클래스에서 초기화 함수 __init__(self, wheel_size, color)를 구현하지 않았을 때는 객체를 생성한 후에 속성을 지정
  - 초기화 함수를 정의한 후로는 객체를 생성하면서 객체의 속성을 지정할 수 있음

클래스를 구성하는 변수와 함수

클래스에서 사용하는 변수

- 클래스에서 사용하는 변수는 위치에 따라 클래스 변수(class variable)와 인스턴스 변수(instance variable)로 구분
- 클래스 변수는 클래스 내에 있지만 함수 밖에서 '변수명 = 데이터'형식으로 정의한 변수로서 클래스에서 생성한 모든 객체가 공통으로 사용
- 클래스 변수는 '클래스명.변수' 형식으로 접근할 수 있음

클래스를 구성하는 변수와 함수

클래스에서 사용하는 변수

- 클래스 변수와 인스턴스 변수를 사용한 자동차 클래스

```
[ ] class Car():
        instance_count = 0 # 클래스 변수 생성 및 초기화

        def __init__(self, size, color):
            self.size = size    # 인스턴스 변수 생성 및 초기화
            self.color = color  # 인스턴스 변수 생성 및 초기화
            Car.instance_count = Car.instance_count + 1 # 클래스 변수 이용
            print("자동차 객체의 수: {0}".format(Car.instance_count))

        def move(self):
            print("자동차({0} & {1})가 움직입니다.".format(self.size, self.color))
```

  - 클래스 변수인 instance_count를 초기화 함수 __init__()에서 Car.instance_count의 형식으로 이용

클래스를 구성하는 변수와 함수

클래스에서 사용하는 변수

- 선언한 클래스를 이용해 클래스 변수와 인스턴스 변수를 각각 어떻게 사용하는 두개의 객체(carI과 car2)를 생성

```
[ ]  car1 = Car('small', 'white')
     car2 = Car('big', 'black')
```

클래스를 구성하는 변수와 함수

클래스에서 사용하는 변수

- 선언한 클래스를 이용해 클래스 변수와 인스턴스 변수를 각각 어떻게 사용하는 두개의 객체(carI과 car2)를 생성

```
[ ]  car1 = Car('small', 'white')
     car2 = Car('big', 'black')
```

- 클래스 Car를 이용해 객체 carI과 car2를 생성
- 출력된 결과를 보면 객체를 생성할 때마다 클래스 변수 instance_count가 1씩 증가해서 Car 클래스의 객체가 몇 개 생성됐는지 알 수 있음

클래스를 구성하는 변수와 함수

클래스에서 사용하는 변수

- 클래스 변수는 '클래스명.변수명' 형식으로 언제든지 호출

```
[ ] print("Car 클래스의 총 인스턴스 개수:{}".format(Car.instance_count))
```

- 클래스 변수도 객체를 생성한 후 '객체명.변수명'형식으로 접근

```
[ ] print("Car 클래스의 총 인스턴스 개수:{}".format(car1.instance_count))
    print("Car 클래스의 총 인스턴스 개수:{}".format(car2.instance_count))
```

  - 출력 결과를 보면 carl과 car2 객체에서 사용한 클래스 변수 instance_count는 값이 같은 것을 볼 수 있음
  - 모든 객체에서 클래스 변수가 공통으로 사용되기 때문

클래스를 구성하는 변수와 함수

클래스에서 사용하는 변수

- 클래스 변수는 '클래스명.변수명' 형식으로 언제든지 호출

```
[ ]  car1.move()
     car2.move()
```

- 출력 결과에서 볼 수 있듯이 인스턴스 변수(여기서는 self .size와 self .color)는 각 객체에서 별도로 관리됨

클래스를 구성하는 변수와 함수

클래스에서 사용하는 변수

- 이름이 같은 클래스 변수와 인스턴스 변수가 있는 클래스를 정의해서 객체에서 각 변수가 동작

```
[ ] class Car2():
        count = 0; # 클래스 변수 생성 및 초기화

        def __init__(self, size, num):
            self.size = size    # 인스턴스 변수 생성 및 초기화
            self.count = num  # 인스턴스 변수 생성 및 초기화
            Car2.count = Car2.count + 1 # 클래스 변수 이용
            print("자동차 객체의 수: Car2.count = {0}".format(Car2.count))
            print("인스턴스 변수 초기화: self.count = {0}".format(self.count))

        def move(self):
            print("자동차({0} & {1})가 움직입니다.".format(self.size, self.count))
```

클래스를 구성하는 변수와 함수

클래스에서 사용하는 변수

이름은 동일, 동작하는 것은 다름

- 클래스의 초기화 함수 __init__()에서 클래스 변수 count(함수 내에서 Car2.count로 이용)와 인스턴스 변수 count(함수 내에서 self.count로 이용)를 이용
- 변수 이름은 같지만 이 둘은 별개로 동작
- 객체를 생성해서 각 객체에서 두 변수가 어떻게 동작하는지 확인

```
[ ]  car1 = Car2("big", 20)
     car2 = Car2("small", 30)
```

- 클래스 변수 count와 인스턴스 변수 count가 별개로 동작하는 것

클래스를 구성하는 변수와 함수

클래스에서 사용하는 함수

- 클래스에서 정의할 수 있는 함수(메서드)에는 그 기능과 사용법에 따라
- 인스턴스 메서드(instance method)
- 정적 메서드(static method)
- 클래스 메서드(class method)가 있음

클래스를 구성하는 변수와 함수

인스턴스 메서드

- 인스턴스 메서드는 각 객체에서 개별적으로 동작하는 함수를 만들고자 할 때 사용하는 함수
- 인스턴스 메서드는 함수를 정의할 때 첫 인자로 self가 필요
- self는 클래스의 인스턴스(객체) 자신을 가리킴
- 인스턴스 메서드에서는 self를 이용해 인스턴스 변수를 만들고 사용
- 인스턴스 메서드 안에서는 'self.함수명()'형식으로 클래스 내의 다른 함수를 호출할 수 있음

클래스를 구성하는 변수와 함수

인스턴스 메서드

- 인스턴스 메서드의 구조

```
class 클래스명():

    def 함수명(self[, 인자1, 인자2, • • , 인자n]):

        self. 변수명1 = 인자1

        self. 변수명2 = 인자2

        self. 변수명3 = 데이터

        〈코드 블록〉
```

클래스를 구성하는 변수와 함수

인스턴스 메서드

- 인스턴스 메서드는 객체를 생성한 후에 호출

객체명 = 클래스명()

객체명.메서드명([인자1, 인자2, • • , 인자n])

파일 객체 (클래스) 안에는 변수와 함수 (read, close, write,read)가 있음
아래는 모두 함수 중 인스턴스 함수에 해당함, 데코레이션이 없음

```
f1=open('test1.txt')
f1.read( )
f1.close ( )
f2=open('test2.txt)
f1.write ()
f2.close ( )
```

클래스를 구성하는 변수와 함수

인스턴스 메서드

-
```
[ ] # Car 클래스 선언
    class Car():
        instance_count = 0 # 클래스 변수 생성 및 초기화

        # 초기화 함수(인스턴스 메서드)
        def __init__(self, size, color):
            self.size = size    # 인스턴스 변수 생성 및 초기화
            self.color = color  # 인스턴스 변수 생성 및 초기화
            Car.instance_count = Car.instance_count + 1 # 클래스 변수 이용
            print("자동차 객체의 수: {0}".format(Car.instance_count))

        # 인스턴스 메서드
        def move(self, speed):
            self.speed = speed  # 인스턴스 변수 생성
            print("자동차({0} & {1})가 ".format(self.size, self.color), end='')
            print("시속 {0}킬로미터로 전진".format(self.speed))

        # 인스턴스 메서드
        def auto_cruise(self):
            print("자율 주행 모드")
            self.move(self.speed) # move() 함수의 인자로 인스턴스 변수를 입력
```

클래스를 구성하는 변수와 함수

인스턴스 메서드

- 함수 auto_cruise()는 ‘self.함수명()'을 이용해 인스턴스 메서드(move())를 호출
- 클래스 내의 함수에서 인스턴스 메서드를 호출할 때는 인자에 self는 전달하지 않음

클래스를 구성하는 변수와 함수

인스턴스 메서드

- 인스턴스 메서드를 실행하기 위해 객체를 생성하고 move()와 auto_cruise() 메서드를 호출

```
[ ]  car1 = Car("small", "red") # 객체 생성 (car1)
     car2 = Car("big", "green") # 객체 생성 (car2)

     car1.move(80) #객체(car1)의 move() 메서드 호출
     car2.move(100) #객체(car2)의 move() 메서드 호출

     car1.auto_cruise() #객체(car1)의 auto_cruise() 메서드 호출
     car2.auto_cruise() #객체(car2)의 auto_cruise() 메서드 호출
```

- 인스턴스 메서드인 move()와 auto_cruise()는 두 개의 객체(car1, car2)에서 개별적으로 동작

클래스를 구성하는 변수와 함수

정적 메서드

- 정적 메서드는 클래스와 관련이 있어서 클래스 안에 두기는 하지만
- 클래스나 클래스의 인스턴스(객체) 와는 무관하게 독립적으로 동작하는 함수를 만들고 싶을 때 이용하는 함수
- 함수를 정의할 때 인자로 self를 사용하지 않으며 정적 메서드 안에서는 인스턴스 메서드나 인스턴스 변수에 접근할 수 없음

클래스를 구성하는 변수와 함수

정적 메서드

- 함수 앞에 데코레이터 (Decorator) 인 @staticmethod를 선언해 정적 메서드임을 표시
- 정적 매서드의 구조

```
class 클래스명():

    @staticmethod

    def 함수명([인자1, 인자2, • • , 인자n]):

        <코드 블록>
```

클래스를 구성하는 변수와 함수

정적 메서드

- 객체를 생성한 후에 정적 메서드를 호출할 수도 있지만 정적 메서드는 보통 객체를 생성하지 않고 클래스명을 이용해 바로 메서드를 호출
- 클래스명.메서드명([인자1, 인자2, • • • , 인자n]):

클래스를 구성하는 변수와 함수

정적 메서드

- 객체를 생성한 후에 정적 메서드를 호출할 수도 있지만 정적 메서드는 보통 객체를 생성하지 않고 클래스명을 이용해 바로 메서드를 호출
- 클래스명.메서드명([인자1, 인자2, • • • , 인자n]):
- 정적 메서드는 날짜 및 시간 정보 제공, 환율 정보 제공, 단위 변환과 같이 객체와 관계없이 독립적으로 동작하는 함수를 만들 때 주로 이용

클래스를 구성하는 변수와 함수

정적 메서드

- 정적 메서드를 사용한 예로, 앞에서 만든 Car() 클래스에 정적 메서드인 check_type()을 추가

```
[ ] # Car 클래스 선언
    class Car():

        # def __init__(self, size, color): => 앞의 코드 활용
        # def move(self, speed): => 앞의 코드 활용
        # def auto_cruise(self): => 앞의 코드 활용

        # 정적 메서드
        @staticmethod
        def check_type(model_code):
            if(model_code >= 20):
                print("이 자동차는 전기차입니다.")
            elif(10 <= model_code < 20):
                print("이 자동차는 가솔린차입니다.")
            else:
                print("이 자동차는 디젤차입니다.")
```

클래스를 구성하는 변수와 함수

정적 메서드

- 정적 메서드 check_type()을 살펴보면 self 인자 없이 일반 함수처럼 필요한 인자만 사용
- '클래스명.정적메서드명()'형식으로 정적 메서드를 호출

```
[ ]  Car.check_type(25)
     Car.check_type(2)
```

클래스를 구성하는 변수와 함수

클래스 메서드

- 클래스 메서드는 클래스 변수를 사용하기 위한 함수
- 클래스 메서드는 함수를 정의할 때 첫 번째 인자로 클래스를 넘겨받는 cls가 필요하며 이를 이용해 클래스 변수에 접근
- 클래스 메서드를 사용하기 위해서는 함수 앞에 데코레이터인 @classmethod를 지정

클래스를 구성하는 변수와 함수

클래스 메서드

- 클래스 메서드의 구조

```
class 클래스명():

    @classmethod

    def 함수명(cls[, 인자1, 인자2, • • • , 인자n]):

        〈코드 블록〉
```

클래스를 구성하는 변수와 함수

클래스 메서드

- 클래스 메서드도 객체를 생성하지 않고 클래스명을 이용해 바로 호출
- 클래스명.메서드명([인자1, 인자2, • • • , 인자n]):
- 클래스 메서드는 생성된 객체의 개수를 반환하는 등 클래스 전체에서 관리해야 할 기능이 있을 때 주로 이용

클래스를 구성하는 변수와 함수

클래스 메서드

- 클래스 메서드를 사용한 Car() 클래스에 클래스 메서드인 count_instance()를 추가

```
[ ] # Car 클래스 선언
    class Car():
        instance_count = 0 # 클래스 변수

        # 초기화 함수(인스턴스 메서드)
        def __init__(self, size, color):
            self.size = size    # 인스턴스 변수
            self.color = color  # 인스턴스 변수
            Car.instance_count = Car.instance_count + 1

        # def move(self, speed): => 앞의 코드 활용
        # def auto_cruise(self): => 앞의 코드 활용
        ##@staticmethod
        # def check_type(model_code): => 앞의 코드 활용

        # 클래스 메서드
        @classmethod
        def count_instance(cls):
            print("자동차 객체의 개수: {0}".format(cls.instance_count))
```

클래스를 구성하는 변수와 함수

클래스 메서드

- 클래스 변수 instance_count는 초기화 함수 __init()__에서 1씩 증가하므로 객체가 생성될 때마다 값이 1씩 증가
- 클래스 변수 instance_count를 출력하는 클래스 메서드 count_instance()를 호출하면
- 현재까지 생성된 객체의 개수를 알 수 있음

클래스를 구성하는 변수와 함수

클래스 메서드

- '클래스명.클래스메서드명()'형식으로 클래스 메서드를 호출

```
[ ] Car.count_instance() # 객체 생성 전에 클래스 메서드 호출

    car1 = Car("small", "red") # 첫 번째 객체 생성
    Car.count_instance() # 클래스 메서드 호출

    car2 = Car("big", "green") # 두 번째 객체 생성
    Car.count_instance() # 클래스 메서드 호출
```

- 객체를 생성할 때마다 클래스 변수 instance_count의 값이 1씩 증가하는 것

개체와 클래스를 사용하는 이유

객체와 클래스를 사용하는 이유

- 작은 규모의 프로그램을 만들 때는 클래스와 객체를 사용하지 않고 코드를 작성하기도 하지만 규모가 큰 프로그램을 만들 때는 클래스와 객체를 많이 이용
- 게임의 캐릭터와 같이 유사한 객체가 많은 프로그램을 만들 때도 주로 클래스와 객체를 이용해 코드를 작성

개체와 클래스를 사용하는 이유

객체와 클래스를 사용하는 이유

- 컴퓨터 게임의 로봇은 위로만 이동할 수 있다고 가정하고 로봇의 속성과 동작을 정의
- 로봇의 속성: 이름, 위치
- 로봇의 동작:한 칸 이동

개체와 클래스를 사용하는 이유

객체와 클래스를 사용하는 이유

- 클래스와 객체를 사용하지 않고 코드를 작성

```
[ ] robot_name = 'R1' # 로봇 이름
    robot_pos = 0 # 로봇의 초기 위치

    def robot_move():
        global robot_pos
        robot_pos = robot_pos + 1
        print("{0} position: {1}".format(robot_name, robot_pos))
```

- robot_name과 robot_pos 변수에 각각 로봇의 속성을 지정했고 함수 robot_move()는 로봇을 한 칸 이동한 후에 로봇의 이름과 위치를 출력

개체와 클래스를 사용하는 이유

객체와 클래스를 사용하는 이유

- 함수 robot_move()를 호출

```
[ ]  robot_move()
```

- 한 대의 로봇을 구현하기 위해 두 개의 변수(robot_name과 robot_pos)와 하나의 함수(robot_move()) 를 만들었음

개체와 클래스를 사용하는 이유

객체와 클래스를 사용하는 이유

- 로봇을 하나 더 추가해 두 대의 로봇을 구현한 코드

```
[ ] robot1_name = 'R1' # 로봇 이름
    robot1_pos = 0 # 로봇의 초기 위치

    def robot1_move():
        global robot1_pos
        robot1_pos = robot1_pos + 1
        print("{0} position: {1}".format(robot1_name, robot1_pos))

    robot2_name = 'R2' # 로봇 이름
    robot2_pos = 10 # 로봇의 초기 위치

    def robot2_move():
        global robot2_pos
        robot2_pos = robot2_pos + 1
        print("{0} position: {1}".format(robot2_name, robot2_pos))
```

- 로봇이 한 대에서 두 대로 늘어남에 따라 변수와 함수가 두 배로 늘어났음

개체와 클래스를 사용하는 이유

객체와 클래스를 사용하는 이유

- 로봇용으로 정의한 함수를 호출

```
[ ]  robot1_move()
     robot2_move()
```

개체와 클래스를 사용하는 이유

객체와 클래스를 사용하는 이유

- 만약 더 많은 로봇을 구현해야 한다면 그만큼 변수와 함수도 더 늘어날 것
- 로봇이 늘어남에 따라 같은 비율로 변수와 함수가 증가하고 코드 작성과 관리는 상당히 힘들어질 것
- 로봇별로 변수와 함수의 역할은 같다는 사실을 알 수 있음
- 클래스와 객체를 이용하면 편리하게 코드를 작성

개체와 클래스를 사용하는 이유

객체와 클래스를 사용하는 이유

- 변수와 함수로만 구현한 로봇 코드를 클래스와 객체를 이용해 구현
- 로봇 클래스를 선언

```
[ ] class Robot():
        def __init__(self, name, pos):
            self.name = name # 로봇 객체의 이름
            self.pos = pos # 로봇 객체의 위치

        def move(self):
            self.pos = self.pos + 1
            print("{0} position: {1}".format(self.name, self.pos))
```

- Robot 클래스에서 속성값(self .name와 self .pos)은 __init__() 함수에서 초기화하고 move()함수에 한 칸 이동하는 기능을 구현

개체와 클래스를 사용하는 이유

객체와 클래스를 사용하는 이유

- 객체를 생성

```
[ ]  robot1 = Robot('R1', 0)
     robot2 = Robot('R2', 10)
```

- Robot 클래스의 인스턴스 robot1과 robot2 객체를 생성
- 클래스와 객체를 이용하지 않은 코드 에서 로봇의 개수에 비례해 변수와 함수가 늘어났던 것에 비교하면 코드가 간단해졌음

개체와 클래스를 사용하는 이유

객체와 클래스를 사용하는 이유

- 생성 된 각 로봇 객체의 메서드를 실행

```
[ ]  robot1.move()
     robot2.move()
```

- robot1과 robot2 객체의 메서드가 잘 실행되어 각 로봇 객체를 한 칸씩 움직였음

개체와 클래스를 사용하는 이유

객체와 클래스를 사용하는 이유

- 더 많은 로 봇을 만들어야 한다면 얼마든지 로봇 객체를 손쉽게 생성하고 움직일 수 있음

```
[ ]  myRobot3 = Robot('R3', 30)
     myRobot4 = Robot('R4', 40)

     myRobot3.move()
     myRobot4.move()
```

- 클래스를 선언한 이후에는 로봇이 필요할 때마다 로봇 객체만 생성하면 됨
- 객체가 아무리 늘어나도 변수나 함수를 추가로 구현할 필요가 없음
- 객체와 클래스 없이 로봇을 구현할 때보다 코드의 양도 줄고 관리도 편리해짐

클래스 상속

클래스 상속

- 객체를 생성할 때 먼저 객체의 공통된 속성과 행위를 정의하는 클래스를 선언
- 처음부터 클래스를 만들 수도 있지만 이미 만들어진 클래스의 변수와 함수를 그대로 이어받고 새로운 내용만 추가해 서 클래스를 선언할 수도 있음
- 객체지향 프로그래밍에서는 이어받기를 상속
- 상속 관계에 있는 두 클래스는 자식이 부모의 유전적 형질을 이어받는 관계와 유사하기 때문에 흔히 부모 자식과의 관계로 표현해서 부모 클래스와 자식 클래스

클래스 상속

클래스 상속

- 부모 클래스는 상위 클래스 혹은 슈퍼 클래스라고도 하며 , 자식 클래스는 하위 클래스 혹은 서브 클래스
- 자식 클래스가 부모 클래스로부터 상속을 받으면 자식 클래스는 부모 클래스의 속성(변수)과 행위(함수)를 그대로 이용할 수 있음
- 상속 후에는 자식 클래스만 갖는 속성과 행위를 추가할 수 있음

클래스 상속

클래스 상속

- 부모 클래스에서 상속받는 자식 클래스를 선언하는 형식

  class 자식 클래스 이름(부모 클래스 이름):

    〈코드 블록〉

클래스 상속

클래스 상속

- 부모 클래스로부터 상속을 받으려면 클래스를 선언할 때 클래스 이름 소괄호 안에 부모 클래스의 이름을 넣음
- 부모 클래스는 미리 선언되어 있어야 함
- 부모 클래스를 상속한 후에는 자식 클래스에서 부모 클래스의 변수나 함수를 자식 클래스에서 정의한 것처럼 사용할 수 있음
- 부모 클래스에서 정의한 함수와 자식 클래스에서 정의한 함수 이름이 같은 경우 부모 클래스의 함수를 호출하려면 명시적으로 '부모 클래스 이름.함수명()' 으로 호출하거나 , 'super().함수명()'을 사용
- 초기화 함수 __init__()에서 많이 이용

클래스 상속

클래스 상속

- 부모 클래스에서 상속받아서 자식 클래스를 만드는 예
- 부모 클래스인 자전거 클래스를 상속받아서 자식 클래스인 접는 자전거 클래스를 만듬
- 접는 자전거는 일반 자전거의 속성과 동작을 그대로 갖고 있기 때문
- 상속한 후에는 접는 자전거의 속성과 동작만 추가하면 됨

클래스 상속

클래스 상속

- 선언한 자전거 클래스

```
[ ] class Bicycle():

        def __init__(self, wheel_size, color):
            self.wheel_size = wheel_size
            self.color = color

        def move(self, speed):
            print("자전거: 시속 {0}킬로미터로 전진".format(speed))

        def turn(self, direction):
            print("자전거: {0}회전".format(direction))

        def stop(self):
            print("자전거({0}, {1}): 정지 ".format(self.wheel_size, self.color))
```

클래스 상속

클래스 상속

- 자전거 클래스 Bicycle을 상속해 접는 자전거 클래스인 FoldingBicycle을 만듬

```
[ ] class FoldingBicycle(Bicycle):

        def __init__(self, wheel_size, color, state): # FoldingBicycle 초기화
            Bicycle.__init__(self, wheel_size, color) # Bicycle의 초기화 재사용
            #super().__init__(wheel_size, color) # super()도 사용 가능
            self.state = state  # 자식 클래스에서 새로 추가한 변수

        def fold(self):
            self.state = 'folding'
            print("자전거: 접기, state = {0}".format(self.state))

        def unfold(self):
            self.state = 'unfolding'
            print("자전거: 펴기, state = {0}".format(self.state))
```

클래스 상속

클래스 상속

- FoldingBicycle 클래스는 Bicycle 클래스를 상속받은 후에 self.state 변수를 추가
- 자전거를 접는 기능을 수행하는 fold() 함수와 펴는 기능을 수행하는 unfold() 함수를 추가로 구현
- FoldingBicycle 클래스의 초기화 함수인 __init__()에서 인자 wheel_size와 color를 초기화
- 상속받은 Bicycle 클래스의 초기화 함수인 'Bicycle.__init__(self, wheel_size, color)'를 이용
- Bicycle 클래스에는 없는 self.state 변수를 초기화하기 위해 'self.state = state'를 추가

클래스 상속

클래스 상속

- 초기화할 때 부모 클래스의 이름을 이용한 'Bicycle.__init__(self, wheel_size, color)'대신 'super().__ init__(wheel_size, color)' 를 이용할 수도 있음
- super()를 이용할 때는 인자에서 self를 빼야함

클래스 상속

클래스 상속

- FoldingBicycle 클래스의 인스턴스(객체)를 생성한 후에 메서드를 호출

```
[ ] folding_bicycle = FoldingBicycle(27, 'white', 'unfolding') # 객체 생성

    folding_bicycle.move(20) # 부모 클래스의 함수(메서드) 호출
    folding_bicycle.fold()   # 자식 클래스에서 정의한 함수 호출
    folding_bicycle.unfold()
```

- FoldingBicycle 클래스의 인스턴스인 folding_bicycle 객체를 생성한 후 객체의 메서드를 호출
- FoldingBicycle 클래스에서 move() 함수를 구현하지 않음
- Bicycle 클래스에서 상속받았으므로 FoldingBicycle 클래스에서 생성된 객체에서도 Bicycle 클래스의 함수를 이용
- fold() 함수와 unfold() 함수는 FoldingBicycle 클래스에서 추가로 구현했으므로 folding_bicycle 객체에서 호출

클래스 상속

클래스 상속

- 클래스에서 상속을 이용하면 이미 만들어진 클래스의 변수와 함수를 그대로 이 용할 수 있으므로 코드의 재사용성이 좋아짐
- 유사한 클래스를 여러 개 만들어야 할 경우 공통 부분은 부모 클래스로 구현하고 부모 클래스를 상속하는 자식 클래스를 각각 구현한다면 좀 더 간편 하게 코드를 작성할 수 있음

정리

정리

- 객체와 클래스의 개념과 클래스를 선언하고 객체를 생성해 다루는 방법
- 클래스에서 사용하는 클래스 변수와 인스턴스 변수
- 인스턴스 메서드, 정적 메서드, 클래스 메서드의 차이점
- 객체와 클래스를 사용하는 이유와 이미 작성해 놓은 클래스를 상속해서 새로운 클래스를 선언하는 방법