상속 = 상속을 이용하면 새로운 클래스는 기존 클래스를 복사하지 않고, 기존 클래스의 모든 코드를 쓸 수 있다. → 기존클래스에 일부를 추가하거나 변경하여 새 클래스를 생성한다.

기존 클래스 = 부모(parent) 클래스 / 새클래스 = 자식(child) 클래스.

```
>>> class Car():  ## 빈클래스 정의
...     pass
```

```
>>> class Yugo(Car):
...     pass
```
↳ Car의 서브클래스 Yugo 정의.

↳ 서브클래스 = 같은 class를 사용하지만, 괄호안에 부모클래스의 이름을 지정한다.

```
>>> give_me_a_car = Car()
>>> give_me_a_yugo = Yugo()
```
=> give_me_a_yugo 객체 = Yugo 클래스의 인스턴스지만, 또한 Car 클래스가 할수 있는 어떤것을 상속받는다.

```
>>> class Car():
        def exclaim(self):
            ① print("I'm a car!")
>>> class Yugo(Car):
        pass
```

①
```
>>> give_me_a_car.exclaim()
I'm a car!
>>> give_me_a_yugo.exclaim()
I'm a car!
```
↳ Yugo = Car로부터 exclaim() 메서드를 상속받았다.
자식 클래스 = 부모 클래스에 없는 메서드를 추가할수 있다.

• 오버라이드(over-ride) = 자식 클래스에서 부모클래스의 메서드를 동일한 이름으로 다시 정의하여 덮어쓰는 것. 「부모클래스에 정의된 메서드를 자식 클래스에서 재정의해서 다른 동작을 하도록 만드는것.」

```
>>> class Person():
        def __init__(self, name):
            self.name = name
```

```
class EmailPerson(Person):
    ② def __init__(self, name, email):
        ④ super().ⓔ__init__(name)
        ① self.email = email
```

① 서브클래스의 __init__() 메서드에 email 매개변수가 추가되었다.

④ super() = 자식클래스에서 부모클래스의 메서드를 호출할 때 사용하는 내장함수

ⓔ super().__init__(name) = Person(부모클래스).__init__() 메서드를 호출한다.

② 'super().__init__(name) = Person.__init(self, name)' = self는 EmailPerson 인스턴스지만, Person.__init__() 에 self를 전달하여 부모클래스인 Person의 초기화 작업을 현재 인스턴스에 적용

한다는 점.

super().__init__(name) = 부모클래스의 방식을 자식 클래스 인스턴스 그대로 따르되, 내 몸(self)에 적용시켜줘.

파이썬은 각각의 객체의 속성과 메서드를 찾기 위해 self 인자를 사용한다.

ex) ① >>> car = Car() → car 객체의 Car 클래스를 찾는다.
>>> car.exclaim() → car 객체를 Car 클래스의 exclaim() 메서드의 self 매개변수에 전달한다.

② class Dog:
    def __init__(self, name): #메서드
        self.name = name #속성: 인스턴스 변수

    def bark(self): #메서드
        print(f"{self.name}가 멍멍 짖습니다.") #self.name은 속성에 접근

dog1 = Dog("초코") #인스턴스
dog2 = Dog("바둑이")

L, class = 공통속성과 메서드를 정의한 설계도: ex) 「class Dog」
인스턴스 = 클래스로부터 생성된 실제 객체.

get/set 속성값과 프로퍼티

```
class Duck():
    def __init__(self, input_name):
        self.hidden_name = input_name

    def get_name(self):
        print('inside the getter')
        return self.hidden_name

    def set_name(self, input_name):
        print('inside the setter')
        self.hidden_name = input_name
    name = property(get_name, set_name)
```

③ 두 메서드(get_name, set_name)을 name 이라는 속성의 프로퍼티로 정의한다.
「set_name = setter 메서드
└get_name = getter 메서드

```
>>> fowl = Duck('Howard')
>>> fowl.name
inside the getter
'Howard'
>>> fowl.name = 'Daffy'
inside the setter
```

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

① @property

```
    def diameter(self):
        return 2 * self.radius
```

↳ 만약 속성의 정의를 바꾸려면 모든 호출자를 수정할 필요없이 클래스 정의에 있는 코드만 수정하면 된다. = 직접 속성을 접근하는 것보다 프로퍼티를 통해서 접근하면 큰 이점이 있다.

② getter 메서드 앞에 @property 데커레이터 / setter 메서드 앞에 @name.setter 데커레이터를 쓴다.

↳ 속성에 대한 setter 프로퍼티를 명시하지 않는다면 외부로부터 이 속성을 설정할수 없다.

• 메서드 타입

```
>>> class A():
    count = 0
    def __init__(self):
        A.count += 1
    def exclaim(self):
        print("I'm an A!")
```

① @classmethod

```
    def kids(cls):
        print("A has", cls.count, "little objects")
```

```
>>> easy_a = A()
>>> breezy_a = A()
>>> A.kids()
A has 2 little objects.
```

① '@classmethod'가 없으면

A.kids() => 오류발생

```
a1 = A()
a1.kids() #정상 작동
```

↳ kids() = 인스턴스 메서드가 되므로 반드시 인스턴스를 통해서만 호출가능.

```
>>> class CoyoteWeapon():
    @staticmethod => 정적 메서드 = @staticmethod 데커레이터가 붙어있고,
    def commercial(): 1번째 매개변수로 self나 cls가 없다.
        print('This CoyoteWeapon has been brought to you by Acme')
```

↳ 이 메서드를 접근하기 위해 CoyoteWeapon 클래스에서 객체를 생성할 필요가 없다.

• private 네임 맹글링

↳ 맹글링(name mangling) = 클래스 내부에서 정의된 변수나 메서드의 이름을 자동으로 변경하여 외부에서 직접 접근하기 어렵게 만드는 매커니즘입니다. => 파이썬은 클래스 정의 외부에서 볼수 없도록 하는 속성에 대한 네이밍 컨벤션(naming convention)이 있다.

```
>>> class Duck():
    def __init__(self, input_name):
        self.__name = input_name
    @property
    def name(self):
        print('inside the getter')
```

```
        return self.__name
    @name.setter
    def name(self, input_name):
        print('inside the setter')
        self.__name = input_name
```

```
>>> fowl = Duck('Howard')
>>> fowl.name
inside the getter
'Howard'

>>> fowl.name = 'Donald'
inside the setter
```

--name 속성을 바로 접근할수 없어.

ex)
```
class myclass:
    def __init__(self):
        self.__secret = "비밀값"
    def reveal(self):
        print(self.__secret)
obj = myclass()
obj.reveal()  # 출력 = '비밀값'
print(obj.__secret) => AttributeError:
'Myclass' object has no attribute
'__secret'
```

맹글링 발생.

⇒ sol) print(obj._Myclass__secret)

∴ 맹글링 = _변수명 형태의 이름을 '_클래스명__변수명'으로 내부 변환시켜 클래스 내부 속성을 외부에서 실수로 건드리는 것을 막기 위한 파이썬의 간접적 보호 장치.