

파이썬 클래스

Seolyoung Jeong, Ph.D.

경북대학교 IT대학 컴퓨터학부

객체 지향 언어

파이썬(Python)이란?

- ◆ 1991년 귀도 반 로섬(Guido Van Rossum)이 발표
- ◆ 플랫폼 독립적인 인터프리터 언어이며,
- ◆ 객체 지향적, 동적 타이핑 대화형 언어
- ◆ 처음 C언어로 구현되었음



Guido Van Rossum



Python Logo

파이썬은 ‘피톤’이라는 이름으로 알려진, 고대 그리스 신화에 나오는 거대한 뱀의 이름. 피톤은 Python을 고대 그리스어로 읽은 것이며, 영어를 그대로 읽으면 ‘파이선’이 됨.

사실, 파이썬이라는 이름은 파이썬을 만든 귀도 반 로섬(Guido van Rossum)이 자신이 좋아하는 영국 코미디 프로인 ‘몬티 파이선의 날아다니는 서커스(Monty Python’s Flying Circus)’에서 따왔다고 함. 물론 여기서의 파이선도 피톤을 의미.

객체 지향적 언어

- ◆ 실행 순서가 아닌, 단위 **객체**를 중심으로 프로그램을 작성.
객체 지향 프로그래밍 (Object-Oriented Programming, OOP)
 - 객체 : 실생활에서의 모든 자료(물건)
 - 모든 객체는 **속성**(데이터, Attribute)과 **행동**(Method)을 가짐
 - 예) '공' 객체
 - 속성: 색상, 크기, 무게
 - 행동: 던지다, 차다, 튀기다
- ◆ 객체 지향 프로그래밍은 이러한 객체 개념을 프로그램으로 표현
속성 → **변수**, **행동** → **함수**로 표현
- ◆ 객체 지향 프로그래밍 언어
 - JAVA, C++, C#, **Python**, Objective-C, Swift, ...

객체 지향 프로그래밍

◆ 특징

- 추상화(Abstraction) : 객체들이 가진 공통적인 데이터들을 추상화시킴으로써, 대규모 개발에 유연성 있게 접근 가능
- 캡슐화(Encapsulation) : 추상화를 통해 내부에서 관리할 데이터와 외부에서 보여질 데이터를 관리함으로써 정보은닉 제공
- 상속성(Inheritance) : 기존의 클래스가 갖고 있는 속성과 메소드 정보를 모두 새로운 클래스에게 할당할 수 있는 기능. 프로그램의 재사용성 높임
- 다형성(Polymorphism) : 동일한 메소드 호출에 대해 호출하는 객체, 매개변수 유형 등에 따라 다르게 동작 가능

◆ 장점

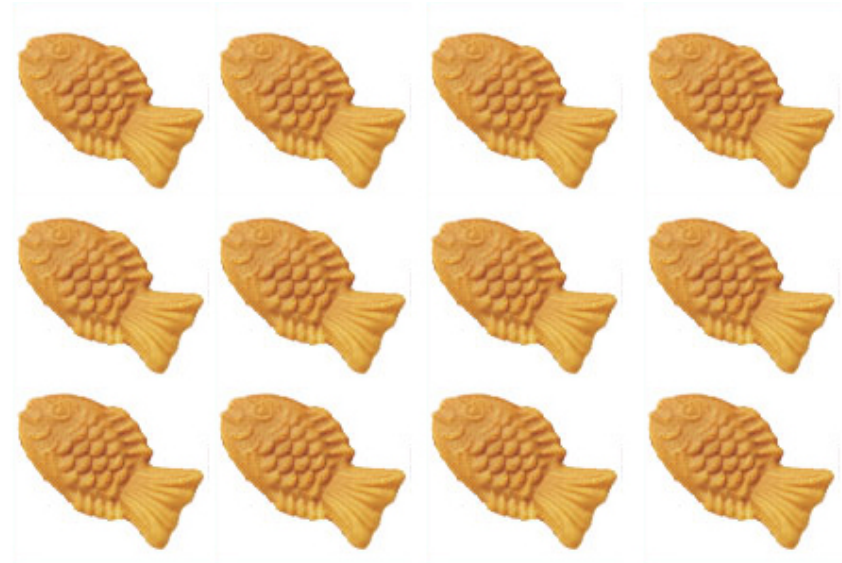
- 소프트웨어의 확장성(extensibility) 향상
- 재사용성(reusability) 향상
- 생산성(productivity) 향상
- 유지보수(maintainability) 비용 절감

객체 지향 프로그램

- ◆ 객체 템플릿 : 클래스(class)
- ◆ 실제 구현체 : 인스턴스(instance)



붕어빵 틀
(Class)



붕어빵
(Instance)

지금까지 사용한 클래스와 인스턴스 예)

```
>>>
>>> a = list(range(10))
>>> a
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>>
>>> b = dict(x=10, y=20)
>>> b
{'x': 10, 'y': 20}
>>>
>>> a.append(20)
>>> a
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 20]
>>>
>>> type(a)
<class 'list'>
>>> type(b)
<class 'dict'>
>>>
>>> c = 10
>>> type(c)
<class 'int'>
>>> c = int(10)
>>> type(c)
<class 'int'>
>>>
```

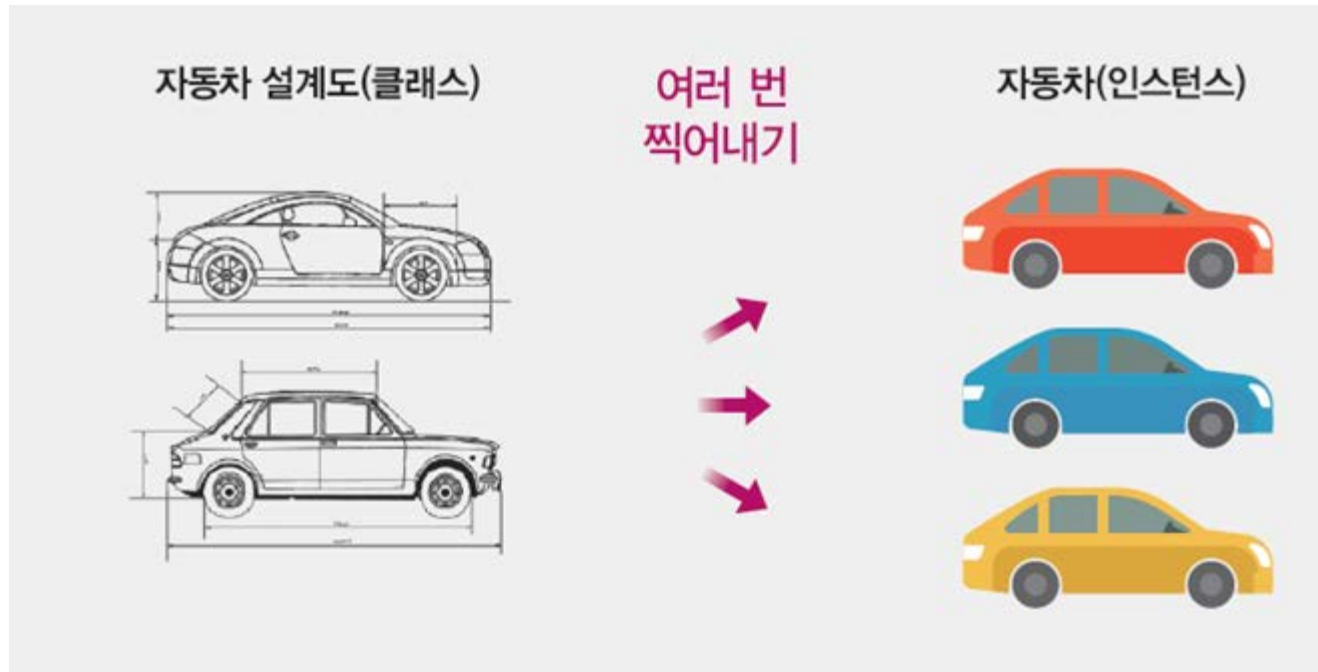
자동차(Car) 클래스 생성

- ◆ 자동차의 속성 : 멤버변수
- ◆ 자동차의 기능 : 함수 형태로 구현 (클래스 안에서 구현된 함수 : Method)

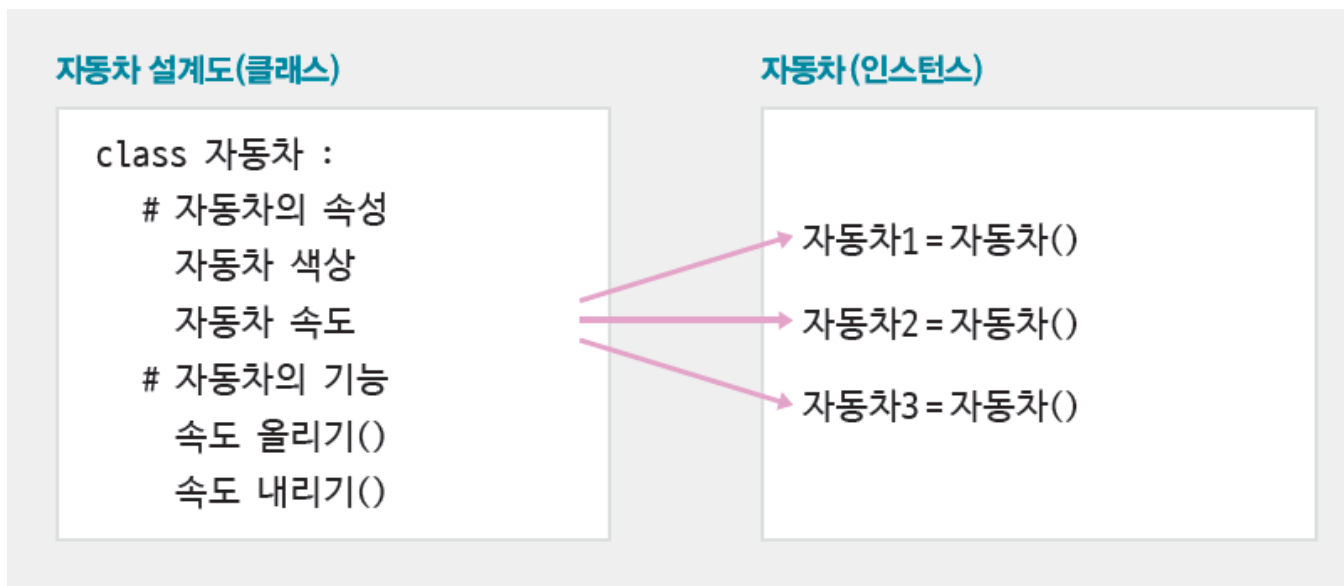
```
1  ## class define
2  class Car:
3      ## attribute of class
4      color = ""
5      speed = 0
6
7      ## method of class
8      def up_speed(self, value):
9          self.speed += value
10
11     def down_speed(self, value):
12         self.speed -= value
```

- **self**는 클래스 자기 자신을 가리킴
 - 즉, 8행 `self.speed`는 4행의 `speed`를 의미
 - 클래스 내 모든 method의 첫번째 인자는 반드시 **self** 포함되어야 함
 - 클래스로 정의된 변수, 함수 사용 시 클래스.변수, 클래스.함수로 접근
- ※ Method 안에서 Method 호출 : `self.함수명()`

인스턴스 생성



클래스와 인스턴스 코드 구성



◆ 세 대의 자동차 인스턴스 생성

```
myCar1 = Car()  
myCar2 = Car()  
myCar3 = Car()
```

- 생성된 3개의 인스턴스는 각각의 자동차 색상(color), 속도(speed) 필드를 가짐

클래스 사용

- ◆ 각각의 인스턴스에는 별도의 필드가 존재하며, 각각에 별도의 값 대입이 가능함



클래스 사용

◆ 필드에 값 대입 : `object.attribute = “값”`

```
myCar1.color = “빨강”  
myCar1.speed = 0  
  
myCar2.color = “파랑”  
myCar2.speed = 0  
  
myCar3.color = “노랑”  
myCar3.speed = 0
```



◆ 메소드 호출 : `object.method(인자값)`

```
myCar1.up_speed(30)  
  
myCar2.down_speed(10)  
  
myCar3.up_speed(50)
```

클래스 사용

```
1  ## class define
2  class Car:
3      ## attribute of class
4      color = ""
5      speed = 0
6
7      ## method of class
8      def up_speed(self, value):
9          self.speed += value
10
11     def down_speed(self, value):
12         self.speed -= value
13
14
15     ## main code
16     myCar1 = Car()
17     myCar1.color = "red"
18     myCar1.speed = 0
19
20     myCar2 = Car()
21     myCar2.color = "blue"
22     myCar2.speed = 0
23
24     myCar3 = Car()
25     myCar3.color = "yellow"
26     myCar3.speed = 0
27
28
29     myCar1.up_speed(30)
30     print("CAR_1's color is %s. speed is %dkm" % (myCar1.color, myCar1.speed))
31
32     myCar2.up_speed(60)
33     print("CAR_2's color is %s. speed is %dkm" % (myCar2.color, myCar2.speed))
34
35     myCar3.up_speed(0)
36     print("CAR_3's color is %s. speed is %dkm" % (myCar3.color, myCar3.speed))
```

클래스 정의 및 생성 단계

단계	작업	형식	예
1단계	클래스 생성	<pre>class 클래스_이름: // 멤버변수 선언 // 메소드 선언</pre>	<pre>class Car: color = "" def up_speed(self, value): ...</pre>
			
2단계	인스턴스 생성	<pre>인스턴스 = 클래스_이름()</pre>	<pre>myCar1 = Car()</pre>
			
3단계	필드 및 메소드 사용	<pre>인스턴스.멤버변수 = 값 인스턴스.메소드()</pre>	<pre>myCar1.color = "red" myCar1.up_speed(30)</pre>

클래스 생성자

생성자

◆ 생성자 : 인스턴스를 생성하면 최초로 호출되는 메소드

```
16 myCar1 = Car()  
17 myCar1.color = "red"  
18 myCar1.speed = 0
```

- 16행: 인스턴스 생성
- 17, 18행: 인스턴스 초기화

◆ 생성과 초기화를 동시에 할 수 있는 함수 → 생성자

- 생성자 함수: `__init__()`

```
class 클래스이름:  
    def __init__(self):  
        // 초기화 코드
```


생성자 구현

◆ Car 클래스 생성자

```
1  ## class define
2  class Car:
3      ## attribute of class
4      color = ""
5      speed = 0
6
7      ## method of class
8      def __init__(self):
9          self.color = "red"
10         self.speed = 0
11
12     def up_speed(self, value):
13         self.speed += value
14
15     def down_speed(self, value):
16         self.speed -= value
```

- 20행: 인스턴스 생성 시 자동으로 생성자가 호출되어 초기화

```
19  ## main code
20  myCar1 = Car()
21  #myCar1.color = "red"
22  #myCar1.speed = 0
```

기본 생성자

◆ 매개변수 self만 있는 기본 생성자

```
1  ## class define
2  class Car:
3      ## attribute of class
4      color = ""
5      speed = 0
6
7      ## method of class
8      def __init__(self):
9          self.color = "red"
10         self.speed = 0
11
12     def up_speed(self, value):
13         self.speed += value
14
15     def down_speed(self, value):
16         self.speed -= value
17
18
19 ## main code
20 myCar1 = Car()
21 myCar2 = Car()
```

매개변수가 있는 생성자

◆ 인스턴스 생성 시 초기값을 매개변수로 넘겨줌

```
1  ## class define
2  class Car:
3      ## attribute of class
4      color = ""
5      speed = 0
6
7      ## method of class
8      def __init__(self, val_col, val_spd):
9          self.color = val_col
10         self.speed = val_spd
11
12         def up_speed(self, value):
13             self.speed += value
14
15         def down_speed(self, value):
16             self.speed -= value
17
18
19  ## main code
20  myCar1 = Car("red", 0)
21  myCar2 = Car("blue", 0)
22  myCar3 = Car("yellow", 0)
23
24  myCar1.up_speed(30)
25  print("CAR_1's color is %. speed is %dkm" % (myCar1.color, myCar1.speed))
26
27  myCar2.up_speed(60)
28  print("CAR_2's color is %. speed is %dkm" % (myCar2.color, myCar2.speed))
29
30  myCar3.up_speed(0)
31  print("CAR_3's color is %. speed is %dkm" % (myCar3.color, myCar3.speed))
```

정적 변수 vs. 인스턴스 변수

인스턴스 변수

- ◆ 일반적으로 인스턴스 변수는 생성자 안에 정의

```
1  ## class define
2  class Car:
3      ## constructor
4      def __init__(self, name, speed):
5          ## instance attributes
6          self.name = name
7          self.speed = speed
8
9      ## method of class
10     def get_name(self):
11         return self.name
12
13     def get_speed(self):
14         return self.speed
```

인스턴스 생성 실습

- ◆ 브랜드명과 속도로 자동차 인스턴스를 생성하고, 함수를 사용하여 값 출력

```
1  ## class define
2  class Car:
3      ## constructor
4      def __init__(self, name, speed):
5          ## instance attributes
6          self.name = name
7          self.speed = speed
8
9      ## method of class
10     def get_name(self):
11         return self.name
12
13     def get_speed(self):
14         return self.speed
15
16     ## variables
17     car1, car2 = None, None
18
19     ## main code
20     car1 = Car("BENZ", 30)
21     print("%s's speed is %dkm" % (car1.get_name(), car1.get_speed()))
22
23     car2 = Car("AUDI", 10)
24     print("%s's speed is %dkm" % (car2.get_name(), car2.get_speed()))
```

인스턴스 변수

- ◆ 인스턴스 변수는 인스턴스 생성 후에도 속성 추가 가능
- ◆ 해당 인스턴스에만 생성

```
1  ## class define
2  class Car:
3      ## constructor
4      def __init__(self, name, speed):
5          ## instance attributes
6          self.name = name
7          self.speed = speed
8
9      ## method of class
10     def get_name(self):
11         return self.name
12
13     def get_speed(self):
14         return self.speed
15
16     ## variables
17     car1, car2 = None, None
18
19     ## main code
20     car1 = Car("BENZ", 30)
21     car1.color = 'red'
22     print("%s %s's speed is %dkm" % (car1.color, car1.get_name(), car1.get_speed()))
23
24     car2 = Car("AUDI", 10)
25     print("%s's speed is %dkm" % (car2.get_name(), car2.get_speed()))
```

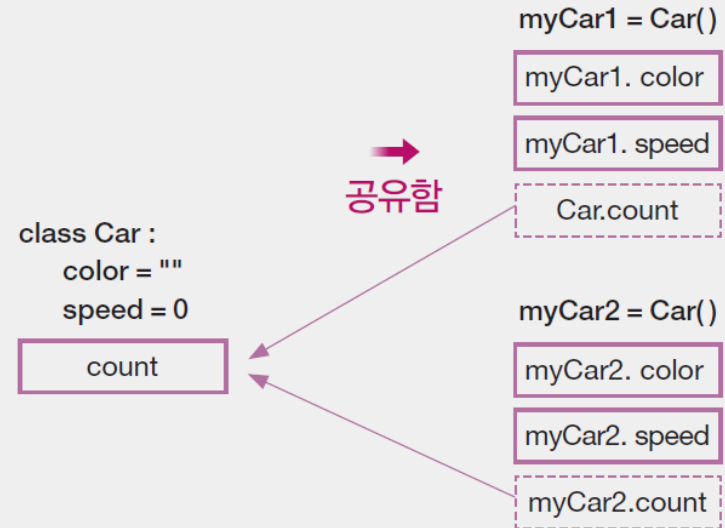
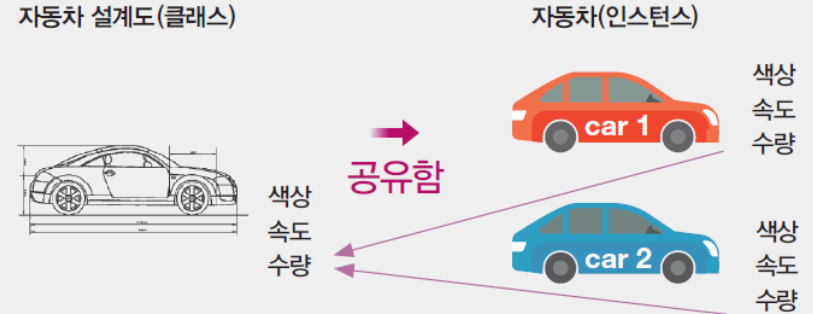
정적 변수 (클래스 변수)

- ◆ 클래스 내부에 공간이 할당되어 여러 인스턴스들이 하나의 자료를 공유 가능 (예: 생산된 자동차 총 대수)



인스턴스 변수

vs.



정적 변수 (클래스 변수)

정적 변수

◆ 정적 변수는 클래스에 정의

- 사용법 : 클래스이름.변수 or 인스턴스이름.변수

```
1  ## class define
2  class Car:
3      count = 0
4
5      ## constructor
6      def __init__(self, name, speed):
7          self.name = name
8          self.speed = speed
9          Car.count += 1
10
11     ## method of class
12     def get_name(self):
13         return self.name
14
15     def get_speed(self):
16         return self.speed
17
18     ## variables
19     car1, car2 = None, None
20
21     ## main code
22     car1 = Car("BENZ", 30)
23     print("%s's speed is %dkm. producted car: %d" % (car1.get_name(), car1.get_speed(), Car.count))
24
25     car2 = Car("AUDI", 10)
26     print("%s's speed is %dkm. producted car: %d" % (car2.get_name(), car2.get_speed(), car2.count))
```

◆ 이름 찾는 순서 : 인스턴스 → 클래스

정적 메소드 vs. 클래스 메소드

정적 메소드

◆ 인스턴스를 통하지 않고, 클래스에서 바로 호출

- 키워드 '@staticmethod' 사용
- self 키워드 없이 정의
- 클래스에 직접 접근 가능, 객체별로 달라지는 것이 아니라 함께 공유
- 주로 인스턴스 속성 및 메서드가 필요 없는 유틸리티성 클래스 만들 때 많이 사용

```
1  # class define
2  class Test:
3
4      @staticmethod
5      def plus(a, b):
6          return a + b
7
8  # main method
9  print("class direct:", Test.plus(1, 2))
```

정적 메소드

- self를 통한 접근 불가능

```
1  # class define
2  class Test:
3      num = 10
4
5      def ins_plus(self, a, b):
6          return a + b + self.num
7
8      @staticmethod
9      def plus(a, b):
10         return a + b + self.num
11
12     # main method
13     print("class direct:", Test.plus(1, 2))
14
15     my_t = Test()
16     print("using instance:", my_t.ins_plus(1, 2))
17
```

```
    return a + b + self.num
NameError: name 'self' is not defined
```

클래스 메소드

- 키워드 '@classmethod' 사용
- cls 매개변수 사용
- 클래스 자체를 객체로 취급

```
1  # class define
2  class Test:
3      num = 10
4
5      @classmethod
6      def plus(cls, a, b):
7          return a + b
8
9  # main method
10 print(Test.plus(1, 2))
```

- cls를 이용하여 메서드 안에서 클래스 속성, 클래스 메서드에 접근 가능
- 메서드 안에서 현재 클래스의 인스턴스 생성 가능 : cls()

클래스 메소드의 필요성

◆ 정적 메소드 now() 선언

```
1  # class define
2  class Date:
3      # static variable
4      word = "date : "
5
6      # construct
7      def __init__(self, date):
8          self.date = self.word + date
9
10     # method
11     @staticmethod
12     def now():
13         return Date("today")
14
15     def show(self):
16         print(self.date)
17
```

클래스 메소드의 필요성

- ◆ Date 클래스 상속 → KoreanDate 자식클래스

➔ 정적 메소드 now()
함수를 통해 KoreanDate
객체가 생성되는게
아니라, Date 객체가
생성됨

```
1  # class define
2  class Date:
3      # static variable
4      word = "date : "
5
6      # construct
7      def __init__(self, date):
8          self.date = self.word + date
9
10     # method
11     @staticmethod
12     def now():
13         return Date("today")
14
15     def show(self):
16         print(self.date)
17
18     # sub class
19     class KoreanDate(Date):
20         # static variable
21         word = "날짜 : "
22
23     # main code
24     my_day = Date("2019-10-04")
25     my_day.show()
26
27     your_day = Date.now()
28     your_day.show()
29
30     his_day = KoreanDate.now()
31     his_day.show()
```

클래스 메소드의 필요성

- ◆ 정적 메소드 → 클래스 메소드로 변경

```
1  # class define
2  class Date:
3      # static variable
4      word = "date : "
5
6      # construct
7      def __init__(self, date):
8          self.date = self.word + date
9
10     # method
11     @classmethod
12     def now(cls):
13         return cls("today")
14
15     def show(self):
16         print(self.date)
17
18     # sub class
19     class KoreanDate(Date):
20         # static variable
21         word = "날짜 : "
22
23     # main code
24     my_day = Date("2019-10-04")
25     my_day.show()
26
27     your_day = Date.now()
28     your_day.show()
29
30     his_day = KoreanDate.now()
31     his_day.show()
```


캡슐화(Encapsulation)

캡슐화(Encapsulation)

- ◆ 클래스에서는 보호가 필요한 일부 데이터를 내부에 숨김
- ◆ 객체 내부가 어떻게 생겼는지 모른 채, 특정 메소드를 사용해서 필요한 정보를 얻어감
- ◆ 캡슐화를 사용하면 개발하기가 용이하고, 정보은닉 가능
- ◆ 일반적인 캡슐화 모드
 - **class의 모드 3가지**
 - **public** : 모든 패키지에서 클래스 참조 가능
 - **private** : 자신을 포함한 클래스에서만 참조 가능
 - **protected** : 자신을 포함하는 클래스에서 상속받은 클래스에서도 참조 가능
 - **method의 모드 3가지**
 - **public** : 클래스 밖에서도 메소드 참조 가능
 - **private** : 해당 클래스 내에서만 참조
 - **protected** : 상속받은 클래스와 해당 클래스 내에서만 참조 가능
 - **field(attribute) 모드 3가지**
 - 메소드 모드와 동일

파이썬 클래스 특징

- ◆ 기본적으로 모든 클래스 멤버들은 **public**
- ◆ 함수들은 **동적 바인딩 (실행 시 요소의 성격 결정)**
- ◆ 인스턴스 메소드에서 **object**를 접근하기 위한 **shorthand** 없음
(C++, Java의 **this** 같은...)
→ 모든 인스턴스 메소드의 첫번째 인자가 **self**
- ◆ **private** 모드 : 기본적으로 파이썬은 **private** 함수나 변수를 지원하지 않음
- ◆ 관습적으로 **'_'**로 시작하는 이름은 **non-public**으로 취급됨
- ◆ **private** 멤버의 필요성으로 인해 **naming decoration**을 사용하여 **private variable** 지원함 (**'__'**)

캡슐화 in 파이썬

◆ Naming Decoration (이름을 변형)을 이용하여 캡슐화 구현

◆ public mode

◆ non-public mode

- protected mode : '_' 사용
- private mode : '__' 사용

→ __ 변수 접근 시 에러 발생

```
1  ## class
2  class Car:
3      ## class attributes
4      count = 0                # public
5      _date = "2019-10-04"    # protected
6      __company = "GM"        # private
7
8      __com1_ = "General Motors one" # private
9      __com2__ = "General Motors two" # public
10
11     ## construct
12     def __init__(self, name):
13         self.brand = name
14         Car.count += 1
15
16     ## class methods
17     # public
18     def get_brand(self):
19         return self.brand
20
21     # protected
22     def _get_date(self):
23         return self._date
24
25     # private
26     def __get_company(self):
27         return self.__company
28
```

접두사 사용

◆ non-public mode에서

- 접미사는 밑줄 한 개('_')까지만 허용 (예) `_num_`, `__num_`
- 접미사는 밑줄이 두 개('__') 이상이면 public으로 간주 (예) `__num__`

```
1  ## class
2  class Car:
3      ## class attributes
4      count = 0                # public
5      _date = "2019-10-04"    # protected
6      __company = "GM"        # private
7
8      __com1_ = "General Motors one" # private
9      __com2__ = "General Motors two" # public
```

캡슐화 in 파이썬

- ◆ 하지만, 파이썬은 white box encapsulation
- ◆ ‘**object._클래스이름__멤버이름**’을 사용하면 protected mode일 때에도 값을 알 수 있음

```
38 my_car = Car("Cadillac")
39
40 print(my_car.count, my_car._date, my_car._Car__company)
41 print(my_car.get_brand(), my_car._get_date(), my_car._Car__get_company())
```

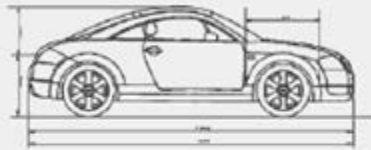
- 파이썬에서는 일반적으로 클래스의 인스턴스나 변수에 값을 직접 쓰지 않음
- 자바, C++ 등에서도 만약 정말로 그런 일을 하고 싶다면 막을 방법 없음 (클래스 소스 코드 자체 편집)
- 파이썬에서는 이런 보안에 대한 속임수를 없애고, **프로그래머로 하여금 책임감을 갖도록** 장려
- ‘_’ 접미사는 원칙적으로 ‘해당 변수로의 접근이 막혀있지 않더라도 그러지 말라’는 의미

상속(Inheritance)

상속(Inheritance)

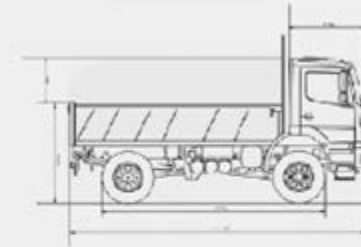
- ◆ 기존 클래스의 필드와 메소드를 그대로 물려받는 **새로운 클래스**를 만드는 것

승용차 클래스



class 승용차 :
필드 - 색상, 속도, 좌석수
메소드 - 속도 올리기()
 속도 내리기()
 좌석수 알아보기()

트럭 클래스



class 트럭 :
필드 - 색상, 속도, 적재량
메소드 - 속도 올리기()
 속도 내리기()
 적재량 알아보기()

상속(Inheritance)

- ◆ 기존의 두 클래스가 공통되는 것이 많음
- ◆ 공통된 특징 ‘자동차’ 클래스
 - 공통된 특징을 물려받은 후, 각각 필요한 필드와 메소드만 추가한 ‘승용차’ 클래스와 ‘트럭’ 클래스

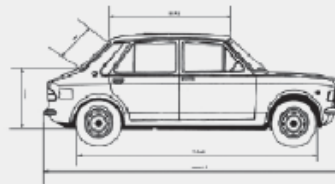
class 자동차:

필드 - 색상, 속도

메소드 - 속도 올리기()

속도 내리기()

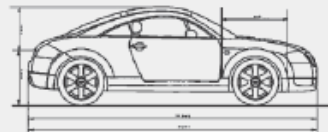
자동차 클래스(공통 내용)



상속



승용차 클래스



class 승용차 : 자동차를 상속

필드 - 자동차 필드, 좌석수

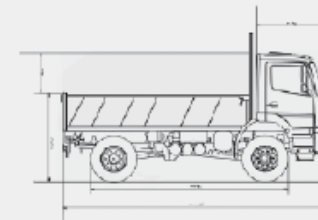
메소드 - 자동차 메소드

좌석수 알아보기()

상속



트럭 클래스



class 트럭 : 자동차를 상속

필드 - 자동차 필드, 적재량

메소드 - 자동차 메소드

적재량 알아보기()

상속(Inheritance)

- ◆ 상위클래스 (자동차 클래스) : 슈퍼클래스 or 부모클래스
- ◆ 하위클래스 (승용차 클래스, 트럭 클래스) : 서브클래스 or 자식클래스
- ◆ 서브클래스 모양

```
class 서브 클래스(슈퍼 클래스) :  
    // 이곳에 서브 클래스의 내용을 코딩
```

상속(Inheritance)

◆ 부모 클래스 정의 : Car

```
1  ## super class : Car
2  class Car:
3      ## constructor
4      def __init__(self, name):
5          self.name = name
6          self.speed = 0
7
8      ## method of class
9      def up_speed(self, value):
10         self.speed += value
11
12     def down_speed(self, value):
13         self.speed -= value
```

상속(Inheritance)

◆ 자식 클래스 정의 : Sedan, Truck

```
15  ## sub class : Sedan
16  class Sedan(Car):
17      ## static attribute
18      seat_num = 0
19
20      ## method of class
21      def get_seatnum(self):
22          return self.seat_num
23
24  ## sub class : Truck
25  class Truck(Car):
26      ## static attribute
27      capacity = 0
28
29      ## method of class
30      def get_capa(self):
31          return self.capacity
```

상속(Inheritance)

◆ 각 클래스의 인스턴스 생성 및 사용

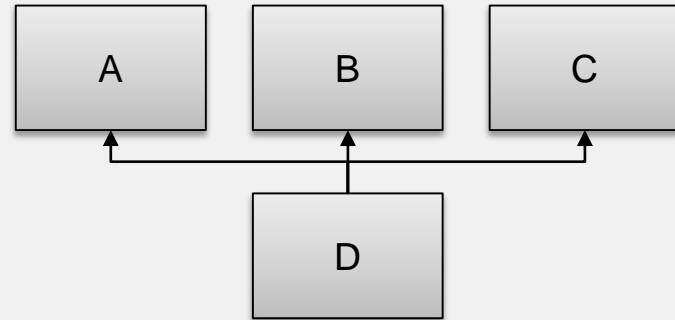
```
32
33  ## global variables
34  my_sedan, my_truck = None, None
35
36  ## main code
37  my_sedan = Sedan("Sonata")
38  my_truck = Truck("Turbo")
39
40  my_sedan.up_speed(100)
41  my_truck.up_speed(70)
42
43  my_sedan.seat_num = 5
44  my_truck.capacity = 50
45
46  print("my car's speed is %dkm, seat : %d " % (my_sedan.speed, my_sedan.get_seatnum()))
47  print("my car's speed is %dkm, capacity : %d " % (my_truck.speed, my_truck.get_capa()))
```

다중 상속

다중 상속

- ◆ 하나의 자식 클래스가 여러 부모로부터 상속 받는 것
- ◆ 부모 클래스의 이름을 콤마(,)로 구분하여 적어 줌

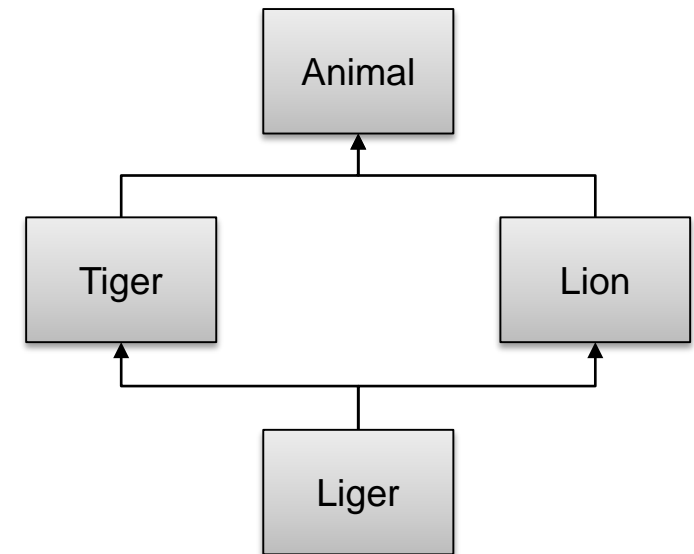
```
class A:  
    ...  
  
class B:  
    ...  
  
class C:  
    ...  
  
class D(A, B, C):  
    ...
```



다중상속 구현

- ◆ level-1 : Animal
- ◆ level-2 : Tiger, Lion
- ◆ level-3 : Liger

```
1 class Animal:
2     def __init__(self, name):
3         print("Class Animal")
4         self.name = name
5
6     def get_name(self):
7         return self.name
8
9 class Tiger(Animal):
10    def __init__(self, name):
11        print("Class Tiger")
12        Animal.__init__(self, name)
13
14 class Lion(Animal):
15    def __init__(self, name):
16        print("Class Lion")
17        Animal.__init__(self, name)
18
19 class Liger(Tiger, Lion):
20    def __init__(self, name):
21        print("Class Liger")
22        Tiger.__init__(self, name)
23        Lion.__init__(self, name)
24
25
26
27 liger = Liger("John")
28 print("\nAnimal's name is %s" % liger.get_name())
```



```
Class Liger
Class Tiger
Class Animal
Class Lion
Class Animal

Animal's name is John

Process finished with exit code 0
```


다중 상속에서 발생할 수 있는 문제점

- ◆ 최상위 클래스의 생성자가 중복 호출
Tiger 클래스 → Animal 클래스 생성자 호출
Lion 클래스 → Animal 클래스 생성자 호출
- ◆ `super()` 함수 → 부모클래스의 객체 반환
(Java의 `super`, C#의 `base` 와 유사)

```
super(). 함수이름 (값)
```

- 인스턴스 인자인 `self`는 전달하지 않아도 됨

super() 함수 사용

```
1 class Animal:
2     def __init__(self, name):
3         print("Class Animal")
4         self.name = name
5
6     def get_name(self):
7         return self.name
8
9 class Tiger(Animal):
10    def __init__(self, name):
11        print("Class Tiger")
12        super().__init__(name)
13
14 class Lion(Animal):
15    def __init__(self, name):
16        print("Class Lion")
17        super().__init__(name)
18
19
20 class Liger(Tiger, Lion):
21    def __init__(self, name):
22        print("Class Liger")
23        super().__init__(name)
24
25
26 liger = Liger("John")
27 print("\nAnimal's name is %s" % liger.get_name())
```

```
Class Liger
Class Tiger
Class Lion
Class Animal
```

```
Animal's name is John
```

```
Process finished with exit code 0
```

super() 함수

- ◆ 명시적으로 부모클래스의 이름을 직접 쓰는 것보다 코드의 유지보수 쉬움
- ◆ 동적 실행 환경에서 클래스 간 상호 동작으로 다중상속 문제 해결 가능
→ 인터프리터 수행 중 Tiger 클래스 생성 시 Animal 클래스를 생성했다면, Lion 클래스 생성 시에는 Animal 클래스의 생성자가 두 번 호출되는 것을 피함
- ◆ 각 클래스에서는 부모의 생성자를 호출하기 위하여 단일상속인지 다중상속인지 고려하지 않고 단지, 'super().__init__()'만 호출하면 됨
- ◆ 생성자뿐만 아니라, 일반 메소드에도 적용 가능
- ◆ super() 함수 호출 시, 명시적으로 클래스 이름과 인스턴스 객체를 인자로 전달하여 호출 가능 (2.x 버전에서 사용 방법, 3.x 버전부터 파라미터 없이도 동작)

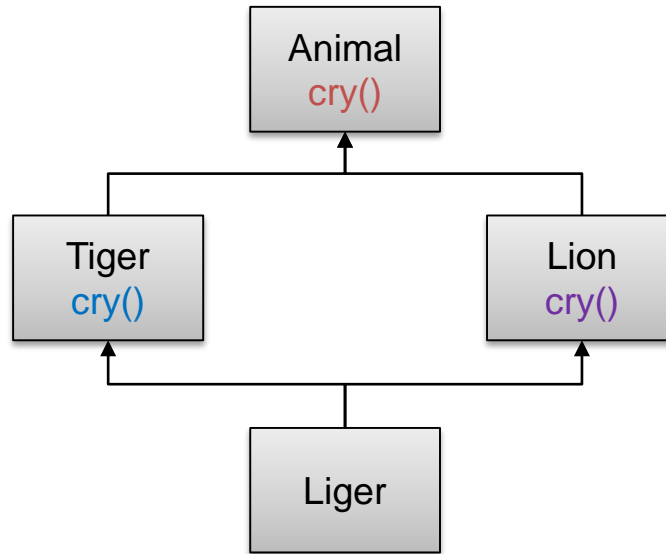
```
super(클래스명, self).__init__(인자)
```

```
class Liger(Tiger, Lion):  
    def __init__(self, name):  
        print("Class Liger")  
        super(Liger, self).__init__(name)
```

다형성 (Polymorphism)

다형성

- ◆ 서로 다른 객체가 동일한 함수에 대해 다르게 수행



- ◆ 함수 이름, 인수, 반환값 모두 동일하나 수행 코드는 다름
→ **Method Overriding**
- ◆ 상위 클래스의 method를 하위 클래스에서 재정의

오버라이딩된 메소드 호출

```
1 class Animal:
2     def __init__(self, name):
3         # print("Class Animal")
4         self.name = name
5
6     def get_name(self):
7         return self.name
8
9     def cry(self):
10        print("Animal %s: ~" % self.name)
11
12
13 class Tiger(Antel):
14     def __init__(self, name):
15         # print("Class Tiger")
16         super().__init__(name)
17
18     def cry(self):
19        print("Tiger %s: 어흥" % self.name)
20
21
22 class Lion(Antel):
23     def __init__(self, name):
24         # print("Class Lion")
25         super().__init__(name)
26
27     def cry(self):
28        print("Lion %s: 으르렁" % self.name)
29
30
31 class Liger(Tiger, Lion):
32     def __init__(self, name):
33         # print("Class Liger")
34         super(Liger, self).__init__(name)
35
36
37 #liger = Liger("John")
38 #print("\nAnimal's name is %s" % liger.get_name())
39
40 Animal("A").cry()
41 Tiger("B").cry()
42 Lion("C").cry()
```

다중 상속에서의 메소드

- ◆ 동일한 이름의 함수가 각 클래스마다 있는 경우 어느 부모 클래스의 함수가 수행될까?

```
"C:\Program Files (x86)\Anaconda3\python.exe"
Class Animal
Class Lion
Class Tiger
Class Liger

Animal's name is John
Method Tiger

Process finished with exit code 0
```

→ 메소드 검색 시, 상속하는 부모 클래스의 나열 순서대로 검색

```
1 class Animal:
2     def __init__(self, name):
3         # print("Class Animal")
4         self.name = name
5
6     def get_name(self):
7         return self.name
8
9     def cry(self):
10        print("Animal %s: ~" % self.name)
11
12
13 class Tiger(Animal):
14     def __init__(self, name):
15         # print("Class Tiger")
16         super().__init__(name)
17
18     def cry(self):
19         print("Tiger %s: 어흥" % self.name)
20
21
22 class Lion(Animal):
23     def __init__(self, name):
24         # print("Class Lion")
25         super().__init__(name)
26
27     def cry(self):
28         print("Lion %s: 으르렁" % self.name)
29
30
31 class Liger(Tiger, Lion):
32     def __init__(self, name):
33         # print("Class Liger")
34         super(Liger, self).__init__(name)
35
36
37 Liger("John").cry()
38 print(Liger.__mro__)
```

함수 및 멤버 검색 순서

- ◆ 각각의 클래스는 독립적인 영역을 가짐
- ◆ 함수 및 멤버 검색 순서
인스턴스 객체 영역 → 클래스 객체 영역 → 전역 영역
- ◆ 클래스 상속 받은 경우
인스턴스 객체 영역 → 자식 클래스 영역 → 부모 클래스 영역
→ 전역 영역
- ◆ 자식 클래스에 정의되어 있지 않은 경우 부모 클래스 참조
➔ 중복된 데이터와 메소드를 최소화. 메모리 사용 효율성 높임
- ◆ 다중 부모클래스 상속 순서대로 호출됨...
- ◆ 호출 순서 확인 :
 - 클래스.__mro__ 또는 클래스.mro()

리스트 [클래스 인스턴스]

◆ 클래스 인스턴스 객체를 리스트 요소로 사용

- 랜덤한 속도를 가지는 10대의 자동차 (이름 A~J) 리스트 생성

```
1  import random
2
3  ## class define
4  class Car:
5      count = 0
6
7      ## constructor
8      def __init__(self, name, speed):
9          self.name = name
10         self.speed = speed
11         Car.count += 1
12
13     ## method of class
14     def get_name(self):
15         return self.name
16
17     def get_speed(self):
18         return self.speed
19
20     ## main code
21     car_list = []
22
23     for i in range(10):
24         car = Car(chr(ord("A")+i), random.randint(10,100))
25         car_list.append(car)
26
27     for car in car_list:
28         print(car.get_name(), car.get_speed())
29
30     print(car_list)
```

비교) 2개의 리스트 : 자동차 이름, 속도

```
1  import random
2
3  ## class define
4  class Car:
5      count = 0
6
7      ## constructor
8      def __init__(self, name, speed):
9          self.name = name
10         self.speed = speed
11         Car.count += 1
12
13         ## method of class
14         def get_name(self):
15             return self.name
16
17         def get_speed(self):
18             return self.speed
19
20     ## main code
21
22     car_name = []
23     car_speed = []
24
25     for i in range(10):
26         car_name.append(chr(ord("A")+i))
27         car_speed.append(random.randint(10,100))
28
29     for j in range(len(car_name)):
30         print(car_name[j], car_speed[j])
```

Any Questions...
Just Ask!

