Python implementation of OOP II (Inheritance, Override)

ESM2014-41

객체지향프로그래밍 및 실습

SKKU 시스템경영공학과

조영일

Class Inheritance

- 상속(Inheritance)기능을 사용하면 부모 클래스(Parent Class, Super Class)의 Attribute와 Method를 그대로 물려받는 자식 클래스(Child Class)를 만들 수 있다.

```
# Parent Class : CustomerClass
class CustomerClass():
    def __init__(self, name, email, mobile_num, date_of_birth, grade):
        self.name = name
        self.email = email
        self.mobile_num = mobile_num
        self.date_of_birth = date_of_birth
        self.grade = grade

    def get_benefit(self):
        if self.grade > 5:
            return "10%"
        else:
            return "5%"

# Child Class : CustomerClass 를 상속 받는 EmployeeCustomerClass
class EmployeeCustomerClass(CustomerClass):
    employee_num = "" # Child Class 만이 가지는 독자적인 Attribute

    # Child Class 만이 가지는 독자적인 Class Method
    def get_employee_num(self):
        return self.employee_num
```

Class Inheritance

```
# 부모 클래스의 __init__을 그대로 사용하여 initialize
>>> new_employee_customer_1 = EmployeeCustomerClass(name='나성균', email='me@skku.edu',
mobile_num='010-1111-2222', date_of_birth="1992-04-06", grade=5)

# 부모 클래스의 메서드를 그대로 사용 할수있다.
>>> new_employee_customer_1.get_benefit()
"5%“

# 자식 클래스에만 존재하는 Attribute
>>> new_employee_customer_1.employee_num = "20091234"

# 자식 클래스에만 존재하는 Method
>>> new_employee_customer_1.get_employee_num()
"20091234"
```

Class Inheritance

- 여러 계층간 상속 : 부모 클래스 역시 부모 클래스를 가질 수 있다.

```python
# Grand Parent Class : Person
class PersonClass():
    age = 0

    def get_age(self):
        return self.age


# Parent Class : CustomerClass
class CustomerClass(PersonClass):
    def __init__(self, name, email, mobile_num, grade):
        self.name = name
        self.email = email
        self.mobile_num = mobile_num
        self.grade = grade

    def get_benefit(self):
        if self.grade > 5:
            return "10%"
        else:
            return "5%"

# Child Class : CustomerClass 를 상속 받는 EmployeeCustomerClass
class EmployeeCustomerClass(CustomerClass):
    employee_num = "" # Child Class 만이 가지는 독자적인 Attribute

    # Child Class 만이 가지는 독자적인 Class Method
    def get_employee_num(self):
        return self.employee_num
```

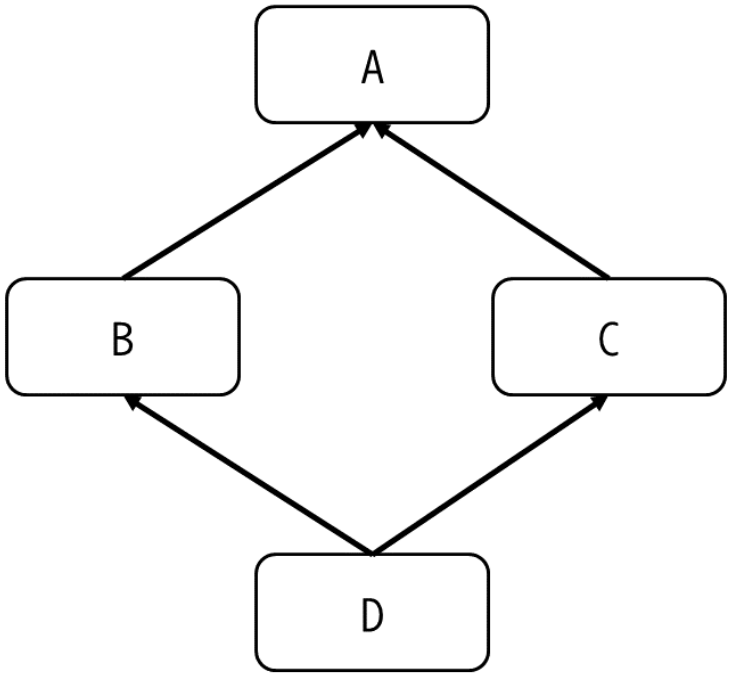
Class Inheritance

- 다중 상속(Multiple Inheritance) : 여러 개의 부모 클래스로 부터 하나의 자식 클래스를 만드는 방법

```
class Person:
    def greeting(self):
        print('안녕하세요.')

class University:
    def manage_gpa(self):
        print('학점 관리')

class Undergraduate(Person, University):
    def study(self):
        print('공부하기')

jeffrey = Undergraduate()
jeffrey.greeting()          # 안녕하세요.: Person의 Method
jeffrey.manage_gpa()     # 학점 관리: University의 Method
jeffrey.study()             # 공부하기: Undergraduate에 추가한 study Method
```

Class Inheritance

- The Deadly Diamond of Death(DDD) : 일명 '죽음의 다이아몬드'. 다중 상속 기능을 이용해서 아래와 같은 상속 관계를 만든다면..?



Class Inheritance

- The Deadly Diamond of Death(DDD) : 프로그래밍 언어에서 가장 중요한 요구사항 중 하나는 같은 구문이 두 가지 이상의 의미로 해석될 여지가 있어서는 안 된다는 점이다.(이런 코드가 달 착륙선을 위한 코드였다면 어떤 결과를 초래했을까?)

```
class A:
    def greeting(self):
        print('안녕하세요. A입니다.')


class B(A):
    def greeting(self):
        print('안녕하세요. B입니다.')


class C(A):
    def greeting(self):
        print('안녕하세요. C입니다.')


class D(B, C):
    pass


x = D()
x.greeting()  # Which Greeting() ??
```

Class Inheritance

- Solution of The Deadly Diamond of Death(DDD)
  1. 명확하지 않은 형태의 다중 상속 설계는 가능한 피한다.(프로그래머의 역할)
  2. 다중 상속 관계에서 탐색의 우선순위를 정한다.(Python이 정한 해결책)

- .mro() method : 메서드 탐색 순서(Method Resolution Order, MRO). Python이 미리 정해 놓은 규칙에 따라 자식 클래스에서 특정 메서드를 호출 했을 때 메서드의 위치를 탐색하는 순서를 반환함 (자식->부모 중 왼쪽->부모 중 왼쪽->상위의 부모..)

```
>>> D.mro()
[<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class
'__main__.A'>, <class 'object'>]
```

Method Overloading vs Overriding

- Feature of OOP – Polymorphism(다형성) : Same name, different function

- OOP 에서 다형성을 구현하는 두가지 방식 : Overloading vs Overriding

- Overloading : 이름은 같지만 전달 받는 Arguments의 개수, 형태가 다른 메서드를 중복하여 선언 하는 것
  (Python에선 지원하지 않는다. / 필요할 경우엔 Arbitrary Argument Lists 를 사용하면 된다.)

- Overriding : 부모/자식 클래스로 이루어진 상속 관계에서 자식 클래스의 메서드를 재정의 하여 같은 이름을 가진 메서드가 다른 역할을 하도록 선언하는 것

Method Overriding

- Overriding : 자식 클래스에서 부모 클래스의 메서드와 동일한 이름의 메서드를 재정의

```
# Parent Class : CustomerClass
class CustomerClass():
    def __init__(self, name, email, mobile_num, grade):
        self.name = name
        self.email = email
        self.mobile_num = mobile_num
        self.grade = grade

    def get_benefit(self):
        if self.grade > 5:
            return 10
        else:
            return 5

# Child Class : CustomerClass 를 상속 받는 EmployeeCustomerClass
class EmployeeCustomerClass(CustomerClass):
    employee_num = "" # Child Class 만이 가지는 독자적인 Attribute

    # Method Overriding
    def get_benefit(self):
        if self.grade > 5:
            return 10 + 5
        else:
            return 5 + 5
```

super() method

- super() : 자식 클래스 내에서 부모 클래스의 내용을 호출할 때 사용하는 method

```
# 직사각형 Class
class Rectangle:
    def __init__(self, length, width):
        self.length = length
        self.width = width

    def area(self):
        return self.length * self.width

    def perimeter(self):
        return 2 * self.length + 2 * self.width

# 정사각형 Class
class Square(Rectangle):
    def __init__(self, length):
        # super() Method 를 사용하여 부모 클래스의 __init__을 이용하여 초기화
        super().__init__(length, length)
```

super() method

- super() : 자식 클래스 내에서 부모 클래스의 내용을 호출할 때 사용하는 method

```
# 정사각형 Class
class Square(Rectangle):
    def __init__(self, length):
        # super() Method 를 사용하여 부모 클래스의 __init__을 이용하여 초기화
        super().__init__(length, length)

# 정육면체 Class
class Cube(Square):
    def surface_area(self):
        face_area = super().area()
        return face_area * 6

    def volume(self):
        face_area = super().area()
        return face_area * self.length
```

HAS-A Relation

- IS-A Relation : 부모 클래스와 자식 클래스를 표현할 때 사용
- HAS-A Relation : 특정 클래스가 다른 클래스를 attribute로서 가지게 되는 경우
- Eg. Movie HAS-A actor.

```
class Movie:
    def __init__(self, name):
        self.name = name
        self.actor_list = []

    def add_actor(self, actor):
        self.actor_list.append(actor)

class Actor:
    def __init__(self, name):
        self.name = name


movie = Movie(name="Ad Astra")
brad = Actor(name="Brad Fitt")

# Movie HAS-A Actor
movie.add_actor(brad)
```

Summary

- Class Inheritance
- Class Multiple Inheritance
- Method Overloading vs Overriding
- super() method
- HAS-A Relation