

Ch16. 클래스 속성 및 상속

- 클래스 속성 사용하기

- 클래스 속성은 클래스에 바로 속성을 구현

```
def 클래스이름:  
    속성 = 값
```

- 인스턴스 속성 (__init__)

```
class 클래스이름:  
    def __init__(self):  
        self.속성 = 값
```

- 클래스 속성 사용하기

- Person 클래스에 바로 bag 속성을 넣고, put_bag 메서드를 생성
- 인스턴스 두 개를 만든 뒤 각각 put_bag 메서드를 사용함

class_class_attribute.py

```
class Person:
    bag = []

    def put_bag(self, stuff):
        self.bag.append(stuff)

james = Person()
james.put_bag('책')

maria = Person()
maria.put_bag('열쇠')

print(james.bag)
print(maria.bag)
```

실행 결과

```
['책', '열쇠']
['책', '열쇠']
```

- 클래스 속성은 클래스에 속해 있으며 모든 인스턴스에서 공유함

- 클래스 속성 사용하기

```
class Person:
    bag = []
    def put_bag(self, stuff):
        self.bag.append(stuff)
```

james.put_bag('책')

maria.put_bag('열쇠')

The diagram illustrates how class attributes are shared. Two blue arrows originate from the right side of the image. The top arrow starts at the text 'james.put_bag('책')' and points to the 'bag = []' line in the class definition. The bottom arrow starts at the text 'maria.put_bag('열쇠')' and also points to the 'bag = []' line. This visualizes that both instances, 'james' and 'maria', are accessing the same shared class attribute 'bag'.

- 클래스 속성 사용하기

- put_bag 메서드에서 클래스 속성 bag에 접근할 때 self를 사용함
- self는 현재 인스턴스를 뜻하므로 클래스 속성을 지칭하기에는 조금 모호함

```
class Person:
    bag = []

    def put_bag(self, stuff):
        self.bag.append(stuff)
```

- 클래스 속성에 접근할 때는 클래스 이름으로 접근하면 좀 더 코드가 명확해짐

- 클래스 속성

```
class Person:
    bag = []

    def put_bag(self, stuff):
        Person.bag.append(stuff)    # 클래스 이름으로 클래스 속성에 접근
```

- 클래스 속성 사용하기

- Person.bag : 클래스 Person에 속한 bag 속성이라는 것을 명확히 함
- 클래스 바깥에서도 클래스 이름으로 클래스 속성에 접근하면 됨

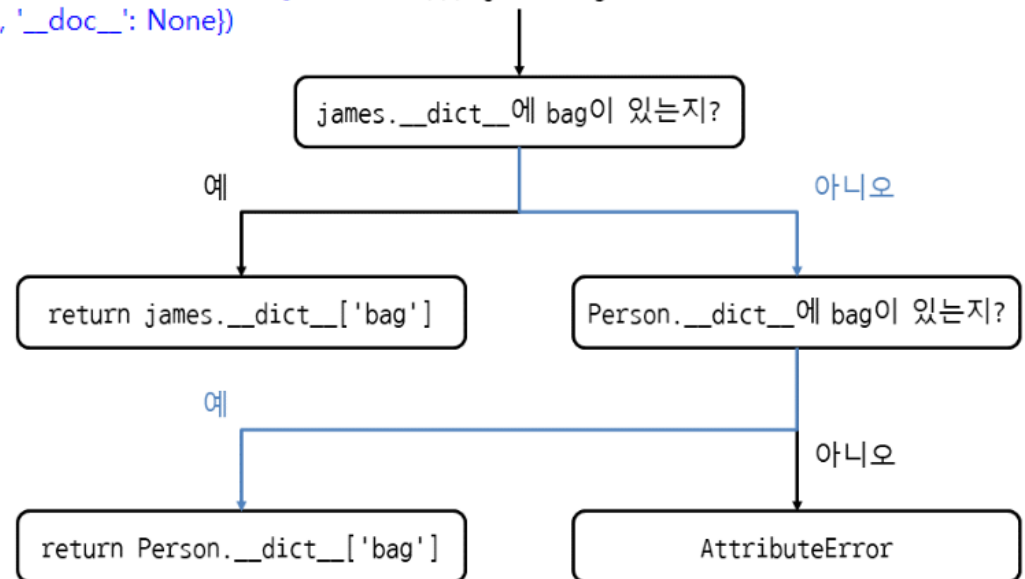
```
print(Person.bag)
```

• Cf) 속성, 메서드 이름 찾는 순서

- 인스턴스, 클래스 순으로 이름 찾음
- 인스턴스 속성이 없으면 클래스 속성을 찾음
- 인스턴스, 클래스에서 `__dict__` 속성을 출력, 현재 인스턴스와 클래스 속성을 딕셔너리로 확인 가능

```
>>> james.__dict__
{}
>>> Person.__dict__
mappingproxy({'__module__': '__main__', 'bag': ['책', '열쇠'], 'put_bag': <function Person.put_bag at 0x00000000031BB048>, '__dict__': <attribute '__dict__' of 'Person' objects>, '__weakref__': <attribute '__weakref__' of 'Person' objects>, '__doc__': None})
```

```
>>> james.bag
```



클래스 속성을 찾는 과정

- 인스턴스 속성 사용하기

– 가방을 여러 사람이 공유하지 않으려면 bag을 인스턴스 속성으로 만들면 됨

class_instance_attribute.py

```
class Person:
    def __init__(self):
        self.bag = []

    def put_bag(self, stuff):
        self.bag.append(stuff)

james = Person()
james.put_bag('책')

maria = Person()
maria.put_bag('열쇠')

print(james.bag)
print(maria.bag)
```

실행 결과

```
['책']
['열쇠']
```


- 인스턴스 속성 사용하기
 - 인스턴스 속성은 인스턴스 별로 독립되어 있으며 서로 영향을 주지 않음
 - 클래스 속성과 인스턴스 속성의 차이점
 - 클래스 속성 : 모든 인스턴스 공유, 인스턴스 전체가 사용해야 하는 값 저장
 - 인스턴스 속성 : 인스턴스 별로 독립, 각 인스턴스가 값을 따로 저장

- 비공개 클래스 속성 사용하기

- 클래스 안에서만 접근할 수 있고, 클래스 바깥에서는 접근할 수 없음

```
def 클래스이름:
    __속성 = 값    # 비공개 클래스 속성
```

- 클래스에서 공개하고 싶지 않은 속성이 있다면 비공개 클래스를 사용해야 함

class_private_class_attribute_error.py

```
class Knight:
    __item_limit = 10    # 비공개 클래스 속성

    def print_item_limit(self):
        print(Knight.__item_limit)    # 클래스 안에서만 접근할 수 있음

x = Knight()
x.print_item_limit()    # 10

print(Knight.__item_limit)    # 클래스 바깥에서는 접근할 수 없음
```

실행 결과

```
10
Traceback (most recent call last):
  File "C:\project\class_private_class_attribute_error.py ", line 11, in <module>
    print(Knight.__item_limit)    # 클래스 바깥에서는 접근할 수 없음
AttributeError: type object 'Knight' has no attribute '__item_limit'
```

- 정적 메서드 사용하기

- 인스턴스를 통하지 않고 클래스에서 바로 호출
- 정적 메서드는 다음과 같이 메서드 위에 @staticmethod를 붙임
- 정적 메서드는 매개변수에 self를 지정하지 않음

```
class 클래스이름:
    @staticmethod
    def 메서드(매개변수1, 매개변수2):
        코드
```

- @staticmethod처럼 앞에 @이 붙은 것을 데코레이터라고 하며
메서드(함수)에 추가 기능을 구현할 때 사용

- cf) 데코레이터

- 데코레이터는 장식하다, 꾸미다라는 뜻의 decorate에 er(or)을 붙인 말인데 장식하는 도구 정도로 설명할 수 있음
- 클래스에서 메서드를 만들 때 @staticmethod, @classmethod, @abstractmethod 등을 붙였는데, 이렇게 @로 시작하는 것들이 데코레이터임
- 함수(메서드)를 장식한다고 해서 이런 이름이 붙었음

```
class Calc:
    @staticmethod    # 데코레이터
    def add(a, b):
        print(a + b)
```

- cf) 데코레이터

- 데코레이터는 함수를 수정하지 않은 상태에서 추가 기능을 구현할 때 사용함

function_begin_end.py

```
def hello():  
    print('hello 함수 시작')  
    print('hello')  
    print('hello 함수 끝')  
  
def world():  
    print('world 함수 시작')  
    print('world')  
    print('world 함수 끝')  
  
hello()  
world()
```

실행 결과

```
hello 함수 시작  
hello  
hello 함수 끝  
world 함수 시작  
world  
world 함수 끝
```

- cf) 데코레이터

decorator_closure.py

```
def trace(func):
    def wrapper():
        print(func.__name__, '함수 시작')
        func()
        print(func.__name__, '함수 끝')
    return wrapper

def hello():
    print('hello')

def world():
    print('world')

trace_hello = trace(hello)
trace_hello()
trace_world = trace(world)
trace_world()
```

호출할 함수를 매개변수로 받음
호출할 함수를 감싸는 함수
__name__으로 함수 이름 출력
매개변수로 받은 함수를 호출
wrapper 함수 반환

데코레이터에 호출할 함수를 넣음
반환된 함수를 호출
데코레이터에 호출할 함수를 넣음
반환된 함수를 호출

실행 결과

```
hello 함수 시작
hello
hello 함수 끝
world 함수 시작
world
world 함수 끝
```

- cf) 데코레이터

```
def trace(func):                                     # 호출할 함수를 매개변수로 받음
```

```
    def wrapper():                                   # 호출할 함수를 감싸는 함수
```

```
        def trace(func):                             # 호출할 함수를 매개변수로 받음
            def wrapper():                             # 호출할 함수를 감싸는 함수
                print(func.__name__, '함수 시작')      # __name__으로 함수 이름 출력
                func()                                 # 매개변수로 받은 함수를 호출
                print(func.__name__, '함수 끝')
            return wrapper                             # wrapper 함수 반환
```

```
trace_hello = trace(hello)    # 데코레이터에 호출할 함수를 넣음
trace_hello()                 # 반환된 함수를 호출
trace_world = trace(world)    # 데코레이터에 호출할 함수를 넣음
trace_world()                 # 반환된 함수를 호출
```

- trace에 다른 함수를 넣은 뒤 반환된 함수를 호출하면 해당 함수의 시작과 끝을 출력할 수 있음

- cf) 데코레이터

- 호출 할 함수 위에 @데코레이터 형식 지정

```
@데코레이터
def 함수이름():
    코드
```

decorator_closure_at_sign.py

```
def trace(func):
    def wrapper():
        print(func.__name__, '함수 시작')
        func()
        print(func.__name__, '함수 끝')
    return wrapper

@trace # @데코레이터
def hello():
    print('hello')

@trace # @데코레이터
def world():
    print('world')

hello() # 함수를 그대로 호출
world() # 함수를 그대로 호출
```

실행 결과

```
hello 함수 시작
hello
hello 함수 끝
world 함수 시작
world
world 함수 끝
```


- cf) 데코레이터

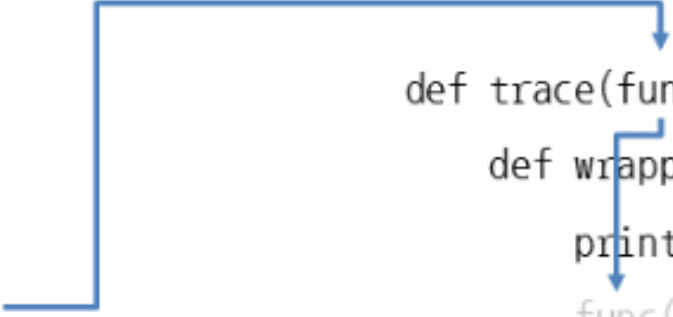
```
@trace    # @데코레이터
def hello():
    print('hello')

@trace    # @데코레이터
def world():
    print('world')

hello()   # 함수를 그대로 호출
world()   # 함수를 그대로 호출
```

```
@trace
def hello():
    print('hello')
```

```
def trace(func):
    def wrapper():
        print(func.__name__, '함수 시작')
        func() hello()
        print(func.__name__, '함수 끝')
    return wrapper
```



- **cf) 데코레이터**

- 정의 : 특정 함수를 랩핑 후 추가적으로 기능 추가, 새로운 함수 작성
- 언제 사용할까?
 - 로그를 남길 때
 - 유저의 로그인 상태를 확인하여 로그인 페이지로 리다이렉트(redirect)
 - 프로그램 성능을 위한 테스트

- 정적 메서드 사용하기

class_static_method.py

```
class Calc:
    @staticmethod
    def add(a, b):
        print(a + b)

    @staticmethod
    def mul(a, b):
        print(a * b)

Calc.add(10, 20) # 클래스에서 바로 메서드 호출
Calc.mul(10, 20) # 클래스에서 바로 메서드 호출
```

실행 결과

```
30
200
```

- 정적 메서드는 self를 받지 않으므로 인스턴스 속성에는 접근할 수 없음
- 정적 메서드는 인스턴스 속성, 인스턴스 메서드가 필요 없을 때 사용함
- **메서드의 실행이 외부 상태에 영향을 끼치지 않는 순수 함수를 만들 때 사용함**
- 순수 함수는 부수 효과가 없고 입력 값이 같으면 언제나 같은 출력 값을 반환
- 정적 메서드는 인스턴스의 상태를 변화시키지 않는 메서드를 만들 때 사용

- 정적 메서드 사용하기

```
class Calc:
    @staticmethod
    def add(a,b):
        print(a+b)

    @staticmethod
    def mul(a,b):
        print(a*b)
```

```
a=Calc()
a.add(10,20)
a.mul(10,20)
```

- static Method는 인스턴스나 클래스를 인자로 받지 않습니다.
- static Method는 클래스 내부에 선언되어 클래스 네임스페이스 안에 저장된다는 점을 제외하면 일반 method들과 크게 다른 점이 없습니다.
- static method의 특징은 클래스를 통해서도, 인스턴스를 통해서도 호출

- 클래스 메서드 사용하기

- 클래스 메서드는 메서드 위에 @classmethod를 붙임
- 클래스 메서드는 첫 번째 매개변수에 cls를 지정해야 함(cls는 class에서 따옴)

```
class 클래스이름:
    @classmethod
    def 메서드(cls, 매개변수1, 매개변수2):
        코드
```

- 클래스 메서드 사용하기

- 클래스 Person을 만들고 인스턴스가 몇 개 만들어졌는지 출력하는 메서드 구현

class_class_method.py

```
class Person:
    count = 0    # 클래스 속성

    def __init__(self):
        Person.count += 1    # 인스턴스가 만들어질 때
                             # 클래스 속성 count에 1을 더함

    @classmethod
    def print_count(cls):
        print('{0}명 생성되었습니다.'.format(cls.count))    # cls로 클래스 속성에 접근

james = Person()
maria = Person()

Person.print_count()    # 2명 생성되었습니다.
```

2명 생성되었습니다.

- 클래스 메서드 사용하기

```
class Person:
    count = 0    # 클래스 속성

    def __init__(self):
        Person.count += 1    # 인스턴스가 만들어질 때
                             # 클래스 속성 count에 1을 더함
```

```
@classmethod
def print_count(cls):
    print('{0}명 생성되었습니다.'.format(cls.count))    # cls로 클래스 속성에 접근
```

```
james = Person()
maria = Person()

Person.print_count()    # 2명 생성되었습니다.
```

- 정적 메서드처럼 인스턴스 없이 호출 가능하다
- 메서드 안에서 클래스속성, 클래스 메서드에 접근해야 할 때 사용

- 클래스 메서드 사용하기

```
class Calc:
    history = []

    @classmethod
    def history_fun(cls):
        for item in Calc.history:
            print(item)

    def __init__(self, num1, num2):
        self.num1 = num1
        self.num2 = num2

    def sum(self):
        result = self.num1 + self.num2
        Calc.history.append("add : " + str(result))
        return self.num1 + self.num2

    def sub(self):
        result = self.num1 - self.num2
        Calc.history.append("sub : " + str(result))
        return self.num1 - self.num2
```

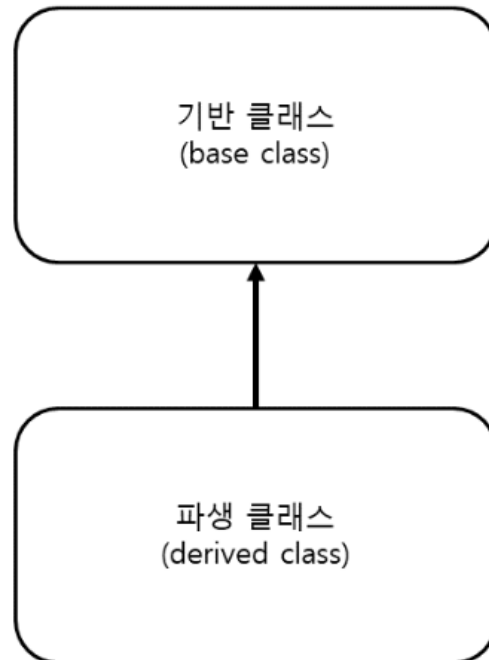
```
C2 = Calc(20, 10)
print(C2.sum())
print(C2.sub())

Calc.history_fun()
```

```
30
10
add : 30
sub : 10
```

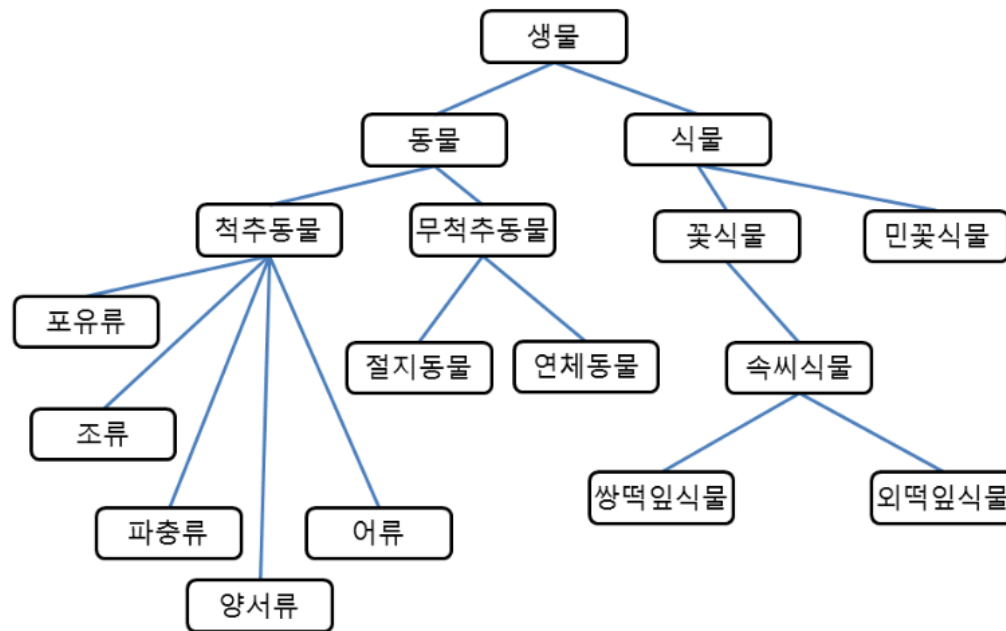

- 클래스 상속 사용하기

- 물려받은 기능을 유지한 채로 다른 기능을 추가할 때 사용하는 기능
- 기반 클래스(base class) : 기능을 물려주는 클래스
- 파생 클래스(derived class) : 상속을 받아 새롭게 만드는 클래스



• 클래스 상속 사용하기

- 기반 클래스는 부모 클래스(parent class), 슈퍼 클래스(superclass)라고 부르고, 파생 클래스는 자식 클래스(child class), 서브 클래스(subclass)라고도 부름
- 클래스 상속도 기반 클래스의 능력을 그대로 활용하면서 새로운 클래스를 만들 때 사용함
- 상속은 기존 기능을 재사용할 수 있어서 효율적임



- **사람 클래스로 학생 클래스 만들기**

- 클래스 상속은 다음과 같이 클래스를 만들 때 ()(괄호)를 붙이고 안에
기반 클래스 이름을 넣음

```
class 기반클래스이름:  
    코드
```

```
class 파생클래스이름(기반클래스이름):  
    코드
```

- **사람 클래스로 학생 클래스 만들기**

- 간단하게 사람 클래스를 만들고 사람 클래스를 상속받아 학생 클래스를 구현

class_inheritance.py

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

class Student(Person):
    def study(self):
        print('공부하기')

james = Student()
james.greeting()    # 안녕하세요.: 기반 클래스 Person의 메서드 호출
james.study()       # 공부하기: 파생 클래스 Student에 추가한 study 메서드
```

실행 결과

```
안녕하세요.
공부하기
```

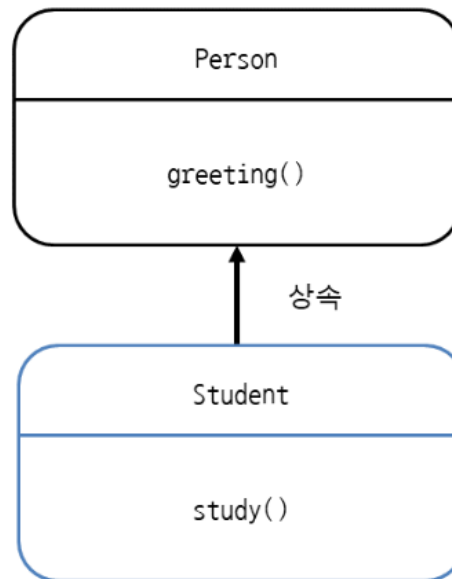
- Person 클래스의 기능을 물려받은 Student 클래스가 됨

- 사람 클래스로 학생 클래스 만들기

- Student 클래스에는 greeting 메서드가 없지만 Person 클래스를 상속받았으므로 greeting 메서드를 호출할 수 있음

```
james = Student()  
james.greeting()    # 안녕하세요.: 기반 클래스 Person의 메서드 호출
```

```
james.study()       # 공부하기: 파생 클래스 Student에 추가한 study 메서드
```



- **사람 클래스로 학생 클래스 만들기**
 - 클래스 상속은 기반 클래스의 기능을 유지하면서 새로운 기능을 추가 가능
 - 클래스 상속은 연관되면서 동등한 기능일 때 사용
 - 학생은 사람이므로 연관된 개념이고, 학생은 사람에서 역할만 확장되었을 뿐 동등한 개념

- Cf) 상속 관계 확인

- Issubclass : 클래스의 상속 관계를 확인하고 싶을 때 사용
해당 클래스가 기반 클래스의 파생 클래스인지 확인
 - Issubclass(파생클래스, 기반클래스)
 - 참이면 True, 거짓이면 False

```
>>> class person :  
    pass
```

```
>>> class Person:  
    pass
```

```
>>> class Student(Person):  
    pass
```

```
>>> issubclass(Student,Person)  
True
```

- 상속 관계

class_is_a.py

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

class Student(Person):
    def study(self):
        print('공부하기')
```

- 상속은 명확하게 같은 종류이며 동등한 관계일 때 사용함
- "학생은 사람이다."라고 했을 때 말이 되면 동등한 관계임
- 상속 관계를 영어로 is-a 관계라고 부름(Student is a Person)

- 기반 클래스의 속성 사용하기

class_inheritance_attribute_error.py

```
class Person:
    def __init__(self):
        print('Person __init__')
        self.hello = '안녕하세요.'

class Student(Person):
    def __init__(self):
        print('Student __init__')
        self.school = '파이썬 코딩 도장'

james = Student()
print(james.school)
print(james.hello)  # 기반 클래스의 속성을 출력하려고 하면 에러가 발생함
```

실행 결과

```
Student __init__
파이썬 코딩 도장
Traceback (most recent call last):
  File "C:\project\class_inheritance_attribute_error.py", line 14, in <module>
    print(james.hello)
AttributeError: 'Student' object has no attribute 'hello'
```

- 실행을 해보면 에러가 발생하는데 .Person의 __init__ 메서드가 호출되지 않으면 self.hello = '안녕하세요.'도 실행되지 않아서 속성이 만들어지지 않음

- **super()로 기반 클래스 초기화하기**
 - super()를 사용해서 기반 클래스의 `__init__` 메서드를 호출해줌
 - `super().메서드()`

class_inheritance_attribute.py

```
class Person:
    def __init__(self):
        print('Person __init__')
        self.hello = '안녕하세요.'

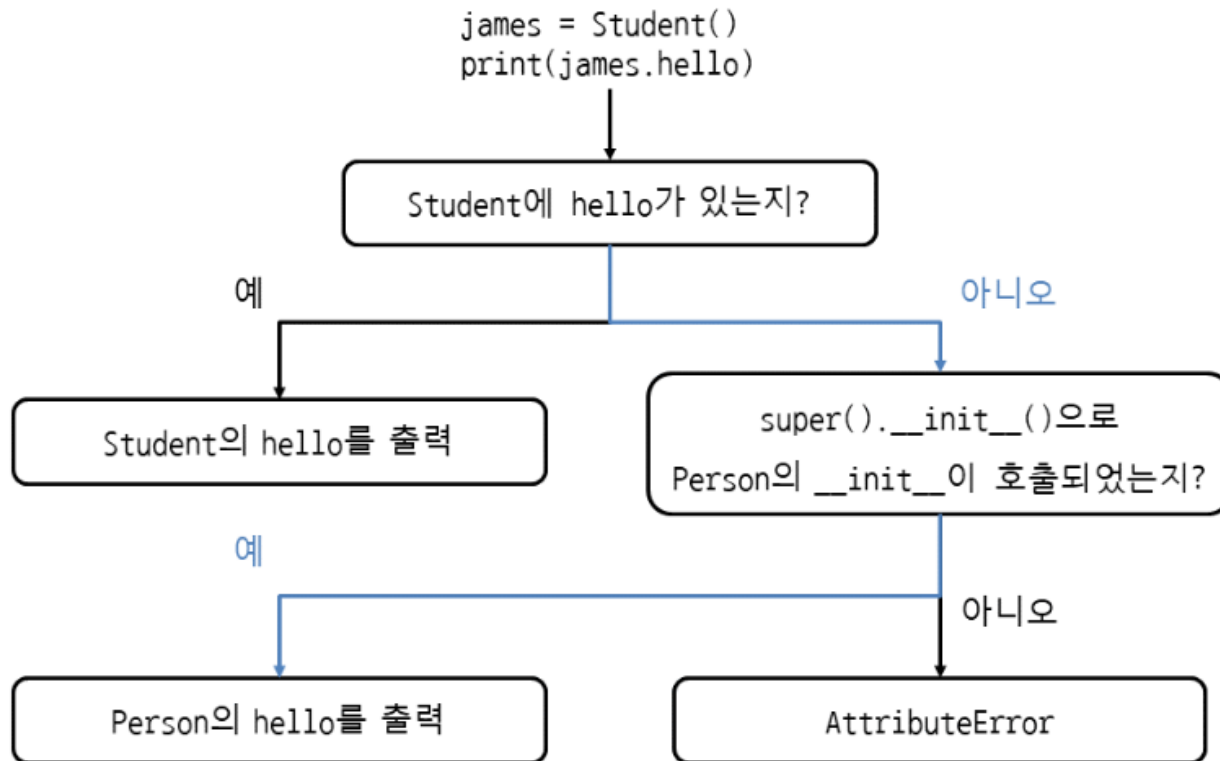
class Student(Person):
    def __init__(self):
        print('Student __init__')
        super().__init__()          # super()로 기반 클래스의 __init__ 메서드 호출
        self.school = '파이썬 코딩 도장'

james = Student()
print(james.school)
print(james.hello)
```

실행 결과

```
Student __init__
Person __init__
파이썬 코딩 도장
안녕하세요.
```

- 기반 클래스의 속성을 찾는 과정



- 기반 클래스를 초기화하지 않아도 되는 경우
 - 만약 파생 클래스에서 `__init__` 메서드를 생략한다면 기반 클래스의 `__init__`이 자동으로 호출되므로 `super()`는 사용하지 않아도 됨

class_inheritance_no_init.py

```
class Person:
    def __init__(self):
        print('Person __init__')
        self.hello = '안녕하세요.'

class Student(Person):
    pass

james = Student()
print(james.hello)
```

실행 결과

```
Person __init__
안녕하세요.
```

- 파생 클래스에 `__init__` 메서드가 없다면 기반 클래스의 `__init__`이 자동으로 호출되므로 기반 클래스의 속성을 사용할 수 있음

- 메서드 오버라이딩 사용하기

- 메서드 오버라이딩 : 상위 클래스의 메서드를 하위 클래스에서 재 정의
- Person의 greeting 메서드가 있는 상태에서 Student에도 greeting 메서드를 만듦

class_method_overriding.py

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

class Student(Person):
    def greeting(self):
        print('안녕하세요. 저는 파이썬 코딩 도장 학생입니다.')

james = Student()
james.greeting()
```

실행 결과

안녕하세요. 저는 파이썬 코딩 도장 학생입니다.

- 메서드 오버라이딩 사용하기

- 오버라이딩(overriding) :
사전적의미 :무시하다, 우선하다라는 뜻
말 그대로 **기반 클래스의 메서드를 무시하고 새로운 메서드를 만든다는 뜻**
- Person 클래스의 greeting 메서드를 무시하고 Student 클래스에서 새로운 greeting 메서드를 만들었음
- 메서드 오버라이딩은 보통 프로그램에서 어떤 기능이 같은 메서드 이름으로 계속 사용되어야 할 때 메서드 오버라이딩을 활용함
- Student 클래스에서 인사하는 메서드를 greeting2로 만들어야 한다면 모든 소스 코드에서 메서드 호출 부분을 greeting2로 수정해야함

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

```
def greeting(self):  
    print('안녕하세요. 저는 파이썬 코딩 도장 학생입니다.')
```

- 이럴 때는 기반 클래스의 메서드를 재활용하면 중복을 줄일 수 있음

- 메서드 오버라이딩 사용하기

- 오버라이딩된 메서드에서 `super()`로 기반 클래스의 메서드를 호출

class_method_overridding_super.py

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

class Student(Person):
    def greeting(self):
        super().greeting()    # 기반 클래스의 메서드 호출하여 중복을 줄임
        print('저는 파이썬 코딩 도장 학생입니다.')

james = Student()
james.greeting()
```

실행 결과

```
안녕하세요.
저는 파이썬 코딩 도장 학생입니다.
```

- 중복되는 기능은 파생 클래스에서 다시 만들지 않고, 기반 클래스의 기능을 사용하면 됨
- 메서드 오버라이딩은 원래 기능을 유지하면서 새로운 기능을 덧붙일 때 사용함

- 다중 상속 사용하기

- 다중 상속은 여러 기반 클래스로부터 상속을 받아서 파생 클래스를 만드는 방법임

```
class 기반클래스이름1:  
    코드  
  
class 기반클래스이름2:  
    코드  
  
class 파생클래스이름(기반클래스이름1, 기반클래스이름2):  
    코드
```


- 다중 상속 사용하기

- 사람 클래스와 대학교 클래스를 만든 뒤 다중 상속으로 대학생 클래스 구현

class_multiple_inheritance.py

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

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

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

james = Undergraduate()
james.greeting()      # 안녕하세요.: 기반 클래스 Person의 메서드 호출
james.manage_credit() # 학점 관리: 기반 클래스 University의 메서드 호출
james.study()         # 공부하기: 파생 클래스 Undergraduate에 추가한 study 메서드
```

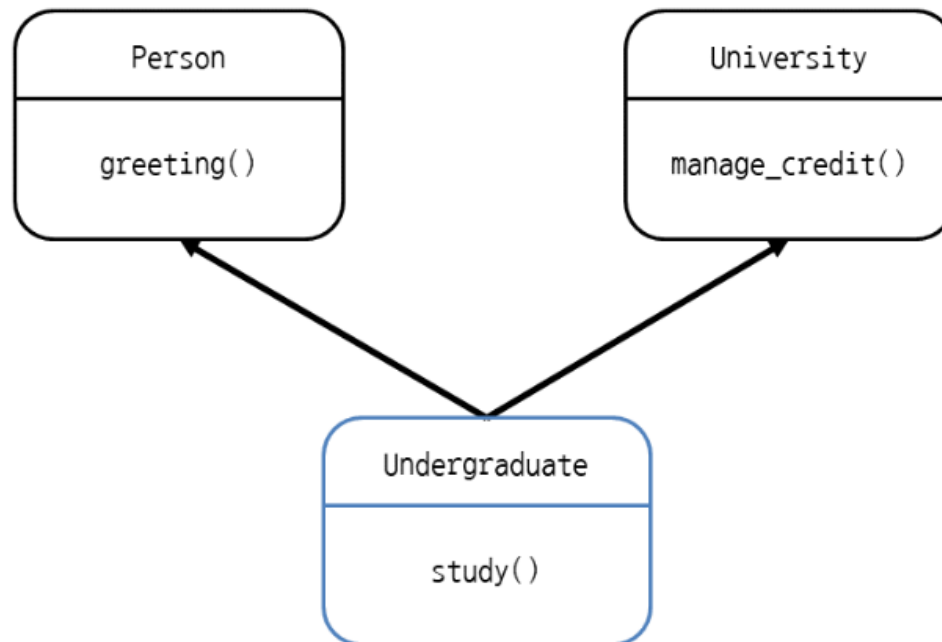
실행 결과

```
안녕하세요.
학점 관리
공부하기
```

- 다중 상속 사용하기

- Undergraduate 클래스의 인스턴스로 Person의 greeting과 University의 manage_credit을 호출할 수 있음

```
james = Undergraduate()  
james.greeting()      # 안녕하세요.: 기반 클래스 Person의 메서드 호출  
james.manage_credit()  # 학점 관리: 기반 클래스 University의 메서드 호출  
james.study()         # 공부하기: 파생 클래스 Undergraduate에 추가한 study 메서드
```



- 다이아몬드 상속

class_diamond_inheritance.py

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

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

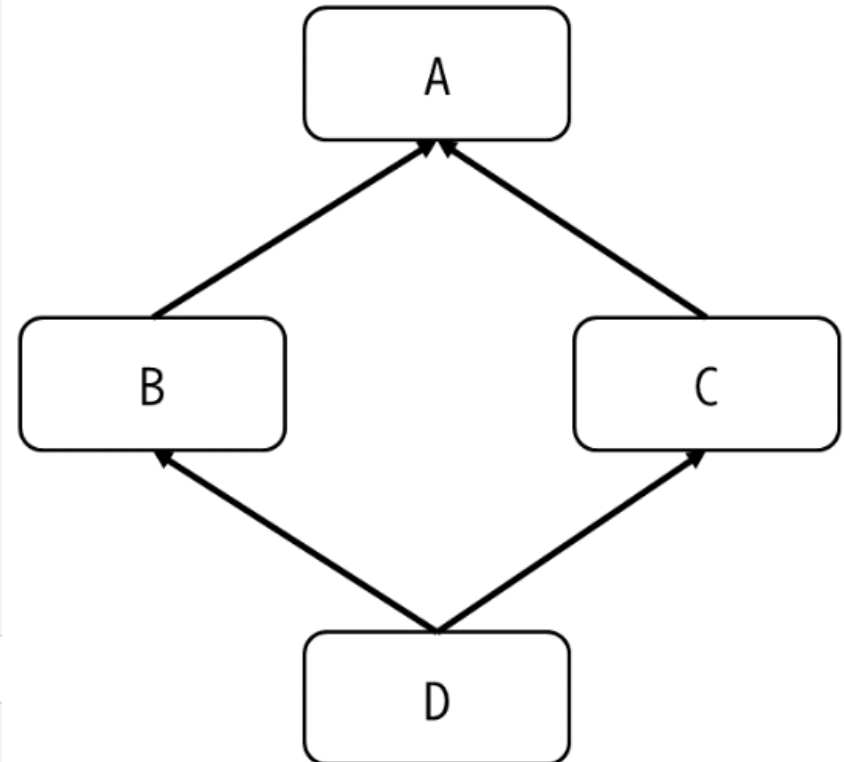
class C(A):
    def greeting(self):
        print('안녕하세요. C입니다.')

class D(B, C):
    pass

x = D()
x.greeting()  # 안녕하세요. B입니다.
```

실행 결과

안녕하세요. B입니다.



- 객체지향 프로그래밍에서는 이런 상속 관계를 다이아몬드 상속이라 부름
- 프로그래밍에서는 이렇게 명확하지 않고 애매한 상태를 좋아하지 않음
- 다이아몬드 상속은 문제가 많다고 해서 죽음의 다이아몬드라고도 부름