제 15 장　상속

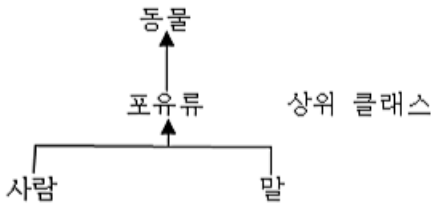
상속은 클래스를 확장하는 중요한 기능이다. 이미 존재하는 클래스를 이용하여 새로운 클래스를 쉽게 정의할 수 있으며, 여러 클래스를 조합하여 하나의 새로운 클래스를 만들수도 있다. 이장에서는 상속 클래스를 만드는 방법과 클래스 간의 관계, 인스턴스 객체와 클래스의 관계 등을 살펴보기로 한다.

15.1 상속

상속은 클래스가 갖는 중요한 특징이다. 상속이 중요한 이유는 재사용성에 있다. 상속받은 클래스는 상속해준 클래스의 속성을 사용할수 있으므로, 추가로 필요한 기능만을 정의하거나, 기존의 기능을 변경해서 새로운 클래스를 만들면 된다.

- 상속과 이름 공간

클래스 A에서 상속된 클래스 B가 있다고 하자. 클래스 A를 기반 클래스, 부모 클래스 또는 상위 클래스라고 하며, 클래스 B를 파생클래스, 자식클래스 또는 하위 클래스라고 한다.



어떤 클래스를 상위클래스로, 어떤 클래스를 하위 클래스로 해야 하는지 혼동되면 두 클래스의 is-a 관계를 생각하면 된다.

'사람' 은 '포유류' 이고 '포유류' 는 '동물' 이다.

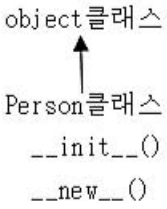
여기서 사람 -> 포유류 -> 동물의 관계가 성립한다. 동물은 포유류의 상위 클래스이고 포유류는 사람의 상위 클래스이다. is-a 관계는 두 개의 클래스의 계층적인 관계를 따질 때 사용한다.

하위 클래스는 상위 클래스의 모든 속성을 그대로 상속받으므로 하위 클래스에는 상위 클래스에 없는 새로운 기능이나 수정하고 싶은 기능만을 재정의하면 된다. 즉, 동일하지는 않으나 기존의 코드와 유사한 새로운 기능이 필요할 때 상속을 이용한다. 다음은 상속에 관한 간단한 예이다. Person 클래스를 보자

```
>>> class Person:
...     def __init__(self, name, phone = None):
...         self.name = name
...         self.phone = phone
...     def __repr__(self):
...         return '<Person{} {}>'.format(self.name, self.phone)
```

이 클래스는 개인에 관한 정보(간단하게 이름과 전화번호)를 가지고 있다. 클래스 어디에서도 상속받지 않는 것처럼 보이지만 파이썬 3의 모든 클래스는 object 클래스의 하위 클래스이다. 다음처럼 기반 클래스의 목록을 알아볼수 있다.

```
>>> Person.__bases__
(object,)
```



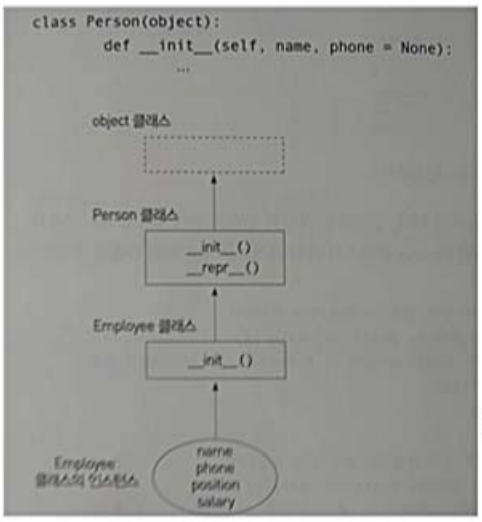
모든 클래스의 최상위 기반 클래스는 object이다.

개인이 취업하게 되면 근로자가 된다. 근로자는 개인의 일반 특성이 있으면서도 근로자의 특성도 있다. Employee 클래스를 Person 클래스의 하위 클래스로 정의하면 다음과 같다.

```
>>> class Employee(Person):              # 상위 클래스는 괄호안에 표현한다.
...     def __init__(self, name, phone, position, salary):
...         Person.__init__(self, name, phone)   # Person 클래스의 생성자 호출
...         self.salary = salary

>>> class Employee(Person):              # 상위 클래스는 괄호안에 표현한다.
...     def __init__(self, name, phone, position, salary):
...         super().__init__(name, phone)                # 상위 클래스의 _init_() 메서드
...         self.position = position
...         self.salary = salary
```

Employee 클래스는 Person 클래스의 속성인 _init_()와 _repr_() 메서드를 그대로 상속받았지만 자신의 _init_() 메서드를 다시 정의했다. 첫 번째의 Employee._init_() 메서드는 person._init_() 메서드를 호출하여 인스턴스 객체의 이름 공간에 name고 phone을 만들고, position과 salary 멤버를 만든다. 두 번째의 Employee 클래스는 상위 클래스를 명시적으로 지정하지 않으나 super() 메서드를 통해서 얻어낸다. 이경우는 바운드 메서드 호출을 수행해야 하므로 self를 전달하지 않는다.

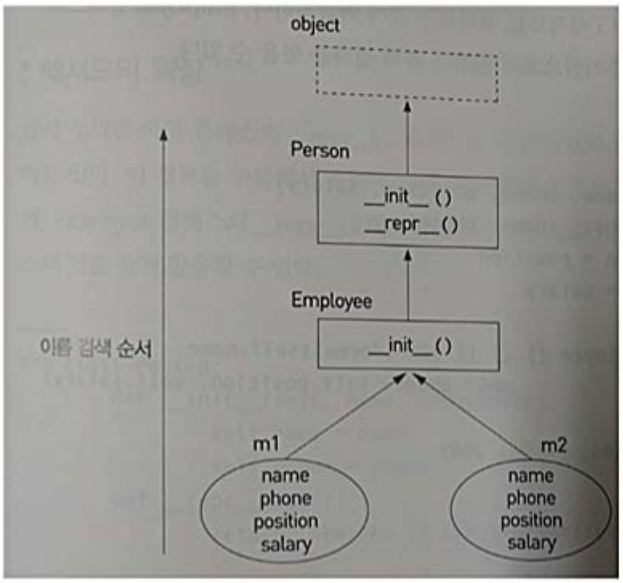
Person클래스는 명시적으로 기술된 상위 클래스가 없지만 다음과 같이 object 클래스로 상속받은것과 동일한 다.



(그림) Person과 Employee 클래스의 상속 관계

```
>>> m1 = Employee('손창희', 5564, '대리', 200)
>>> m2 = Employee('김기동', 8546, '과장', 300)
>>> print(m1.name, m1.position)
손창희 대리
>>> print(m2.name, m2.position)
김기동 과장
>>> print(m1)          # Person._repr_() 메서드 호출
<Person손창희 5564>
>>> print(m2)
<Person김기동 8546>
```

상위 클래스와 하위 클래스는 별도의 이름공간을 가지며 계층적인 관계를 가진다. 클래스 객체와 인스턴스 객체도 역시 모두 별도의 이름 공간과 계층적인 관계를 가진다. 그림으로 그리면 그림과 같다.



멤버 이름 m1.name이 참조될 때, 우선 name이란 이름을 m1에서 찾고, 없으면 Employeef란 클래스 이름 공간에서 찾는다. 그다음에 Person 클래스 이름공간에서 찾고, 그래도 없으면 AttributeError 에러를 발생시킨다. 메서드 이름도 마찬가지이다. m1._repr_()은 Employee와 Person 클래스 이름 공간의 순서로 찾는다. 이러한 계층적인 관계는 얼마든지 확장할 수 있으며, 클래스 인스턴스를 통하여 참조되는 모든 이름은 아래에서 위쪽으로 이동하면서 찾으며 가장 먼저 찾은 이름이 취해진다.

- 메서드 대치

앞의 예어서 print() 함수를 사용하였을 때 상속받은 하위 클래스 Employee는 상위 클래스 Person의 _repr_() 메서드를 호출하였다. Employee 클래스의 정보를 함께 출력하기 위해서 Employee 클래스 자체에 _repr_() 메서드를 정의하는 것이 바람직하다. Employee 클ㄹ래스에 다음과 같이 _repr_()메서드를 추가함으로써 원하는 출력 결과를 얻으수 있다.

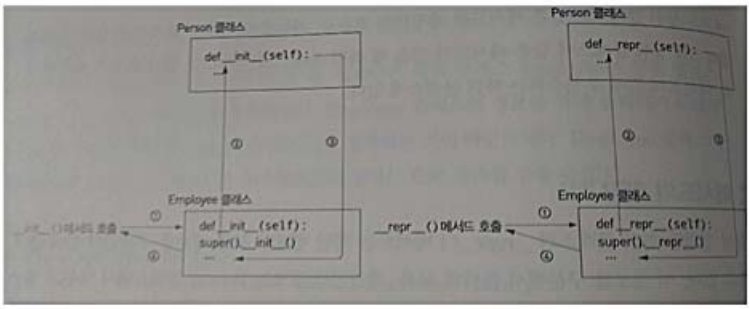
```
>>> class Employee(Person):
    def __init__(self, name, phone, position, salary):
        super().__init__(name, phone)
        self.position = position
        self.salary = salary
    def __repr__(self):
        return '<Employee {} {} {} >'.format(self.name,self.phone,self.position,self.salary)
>>> m1 = Employee('손창희', 5564, '대리', 200)
>>> print(m1)
<Employee 손창희 5564 대리 >
```

이것은 상위 클래스의 같은 메서드를 재정의한 경우로, 기능을 대치하는 효과가 발생한다. 하위클래스와 상위 클래스에 같은 메서드가 있을 때 하위 클래스의 메서드를 먼저 취하기 때문이다. 즉, 메서드의 검색 우선순위는 하위 클래스에 있다.

- 메서드의 확장

앞서 정의한 하위 클래스의 _repr_() 메서드는 개인 정보와 직원 정보를 구분없이 함께 출력하고 있다. 이 정보를 구분해서 출력해 보자. 개인적인 정보븝 Person 클래스에서 다루고 있는데, Employee 클래스의 _repr_() 메서드에서 중복적으로 코드를 정의하지 않고 Person 클래스의 것을 함께 활용할 수 있다.

```
>>> class Person:
...     def __init__(self, name, phone=None):
...         self.name = name
...         self.phone = phone
...     def __repr__(self):
...         return '<Person {} {}>'.format(self.name, self.phone)
>>> class Employee(Person):
...     def __init__(self, name, phone, position, salary):
...             super().__init__(name, phone)
...             self.position = position
...             self.salary = salary
...     def __repr__(self):
...             # 여기서 Person의 _repr_() 메서드를 호출한다.
...             s = super().__repr__()
...             return s + '<Employee {} {}>'.format(self.position, self.salary)
>>> p1 = Person('이강성',5284)
>>> print(p1)
<Person 이강성 5284>
>>> m1 = Employee('손창희', 5564, '대리', 200)
>>> print(m1)
<Person 손창희 5564><Employee 대리 200>
```



하위 클래스 Employee의 _repr_()나 _init_() 메서드의 호출은 super()._repr_(),super()._init_() 메서드와 같은 명시적인 호출로 확장된다. 이처럼 하위 클래스에서 그 속성을 변화시키기 위해서 상위 클래스의 메서드를 호출하고, 그 결과를 활용하는 것을 메섣의 확장자라고 볼수 있다. 메서더의 확장과 치환은 하위 클래스에서 상위 클래스 메서드를 호출하느냐 하지 않느냐에 따라 구분된다.

- 상속 클래스의 예

상속관계를 잘 나타내 주는 예로 도형을 생각해 볼수 있다. 모든 도형의 기본은 점이다. 점 두 개가 모이면 선이되고, 점을 중심으로 일정 거리의 선을 그으면 원이된다. 따라서 점을 상위 클래스로 원을 만들 수 있고, 원을 상위 클래스로 원주를 만들 수 있다.

```
>>> import math
>>> class Point:
...     def __init__(self, x, y):
...             self.x = x
...             self.y = y
...     def area(self):         #  점의 면적은 0이다.
...             return 0
...     def move(self, dx, dy):
...             self.x += dx
...             self.y += dy
...     def __repr__(self):
...             return 'x=%s y=%s' % (self.x, self.y)
...
>>> class Circle(Point):
...     def __init__(self, x, y, r):
...             super().__init__(x, y)
...             self.radius = r
...     def area(self):
...             return math.pi * self.radius * self.radius
...     def __repr__(self):
...             return '{} radius={}'.format(super().__repr__(), self.radius)
...
>>> class Cylinder(Circle):
```

```
...     def __init__(self, x, y, r, h):
...             super().__init__(x, y, r)
...             self.height = h
...     def area(self):         # 원주의 표면적 = 위아래 원의 면적 + 기중의 표면적
...             return 2 * Circle.area(self) + 2 * math.pi * self.radius * self.height
...     def volume(self):       # 체적
...             return Circle.area(self) * self.height
...     def __repr__(self):
...             return '{} heigh={}'.format(super().__repr__(), self.height)
...
>>> if __name__ == '__main__':
...     p1 = Point(3, 5)
...     c1 = Circle(3, 4, 5)
...     c2 = Cylinder(3, 4, 5, 6)
...     print(p1)
...     print(c1)
...     print(c2)
...     print(c2.area(), c2.volume())
...     print(c1.area())
...     c1.move(10, 10)
...     print(c1)
...
x=3 y=5
x=3 y=4 radius=5
x=3 y=4 radius=5 heigh=6
345.5751918948772 471.23889803846896
78.53981633974483
x=13 y=14 radius=5
```

Point 클래스는 도형의 기본 속성을 가지는 클래스이다. 도형의 면적과 이동 그리고, 출력 메서드를 가진다. 원은 점에 반지름이 추가된 속성을 가지는 도형이다. Point 클래스의 생성자를 그대로 이용한다. Circle과 Cylinder 클래스는 각각 클래스에 적합한 area() 메서드와 _repr_() 메서드를 정의하고 있다. 또한, Cyclinder 클래스는 추가적인 메서드 volume 정의하고 있다. 이것을 기초로 좀더 다양한 클래스와 메서드를 정의할 수 있을 것이다.

- 파이썬과 가상 함수

파이썬 클래스의 모든 메서드는 가상 함수이다. 가상함수란 메서드의 호출이 참조되는 클래스 인스턴스에 따라서 동적으로 결정되는 함수를 말한다. 파이썬 클래스의 모든 메서드는 가상 함수이다. 다음은 가상 함수를 사용한 예이다.

```
>>> class Base:
...     def f(self):
```

```
...             self.g()        # 함수 g를 호출한다.
...     def g(self):
...             print('Base')
...
>>> class Derived(Base):
...     def g(self):            # 클래스 Derived의 함수 g
...             print('Derived')
...
>>> b = Base()
>>> b.f()
Base
>>> a = Derived()
>>> a.f()
Derived
```

앞의 클래스 Base에서 함수 f는 함수 g를 호출한다. 그런데 클래스 인스턴스 b를 통하여 f를 호출하였을때는 클래스 Base의 함수 g를 함수 f가 호출하지만, 클래스 인스턴스 a를 통하여 함수 f를 호출하였을때에는 클래스 Base의 함수 g가 아니라 클래스 derived의 함수 g를 호출한다. 이렇게 객체에 따라서 해당 객체에 연관된 함수가 호출되는 것을 가상 함수라고 한다.

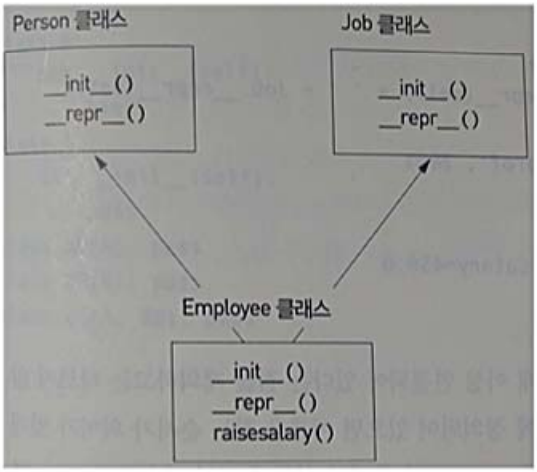
- 다중 상속

두 개 이상의 클래스로부터 상속받는 것을 다중상속이라고 한다. 앞서 정의한 클래스를 약간 수정해서 클래스를 다시 만들어 보자. Employee 클래스는 개인의 속성과 직업속성을 동시에 가지고 있으므로 Person 과 Job 두 개의 클래스로부터 상속받을 수 있다. Person 클래스는 name과 phone 멤버와 _init_()와 _repr_() 메서드를 가지며, Job 클래스는 position과 salary 멤버와 _init_()와 _repr_() 메서드를 가진다. 이들로부터 상속받아 만들어진 Employee 클래스는 이들의 속성을 모두 가진다. 클래스의 정의는

```
class Employee(person, job):
```

와같이 기반 클래스 이름들을 나열하면 된다.

(그림) 다중 상속의 예

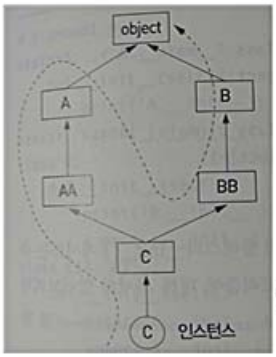


```
>>> class Person:
...     def __init__(self, name, phone=None):
...             self.name = name
...             self.phone = phone
...     def __repr__(self):
...             return 'name={} tel={}'.format(self.name, self.phone)
...
>>> class Job:
...     def __init__(self, position, salary):
...             self.position = position
...             self.salary = salary
...     def __repr__(self):
...             return "position={} salary={}".format(self.position, self.salary)
...
>>> class Employee(Person, Job):
...     def __init__(self, name, phone, position, salary):
...             Person.__init__(self, name, phone)      # 언바운드 메서드 호출
...             Job.__init__(self, position, salary)    # 언바운드 메서드 호출
...     def raisesalary(self, rate):
...             self.salary = self.salary * rate
...     def __repr__(self):
...             # 언바운드 메서드 호출
...             return Person.__repr__(self) +' '+Job.__repr__(self)
...
>>> e = Employee('gslee',5244,'prof',300)
>>> e.raisesalary(1.5)
>>> print(e)
name=gslee tel=5244 position=prof salary=450.0
```

다중 상속과 단일 상속은 이름 공간이 두 개 이상 연결되어 있다는 점을 제외하고는 다르지 않다. 만일 같은 이름이 두 상위 클래스 모두에 정의되어 있으면 이름을 찾는 순서가 의미가 있게 된다. Employee 클래스는 두 상위 클래스(Person, Job)중에서 왼쪽에 먼저 기술된 Person 클래스의 이름공간을 먼저 찾는다.

- 메서드 처리순서

예를들어, 다음 그림에서와 같이 복잡한 상속 관계를 보자.



(그림) 다중 상속에서 이름 공간을 찾는 순서

```
>>> class A:
...     def __init__(self):
...             pass
...
>>> class B:
...     def __init__(self):
...             pass
...
>>> class AA(A): pass
...
>>> class BB(B): pass
...
>>> class C(AA, BB): pass
...
>>> c = C()
```

c = c()로 호출된 _init_() 메서드의 검색 순서는 다음과 같다.

C -> AA -> A -> BB -> B -> object

이 메서드 검색 순서는 _mro_ 속성으로나 mro() 메서드로 확인할 수 있다.

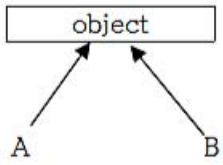
```
>>> C.__mro__
(<class '_main_.C' >, <class '_main_.AA' >, <class '_main_.A' >, <class' _main_.BB' >, <class
 '_main_.B' >, <class 'object' >)
>>> C.mro()
(<class '_main_.C' >, <class '_main_.AA' >, <class '_main_.A' >, <class' _main_.BB' >, <class
 '_main_.B' >, <class 'object' >)
```

따라서 먼저 찾은 A 클래스의 _init_() 메서드가 수행되고 B 클래스의 _init_() 메서드는 수행되지 않는다. 메서드 탐색 순서를 결정하는 MRO C3 알고리즘에 대해 자세히 알아보려면 "The Python 2.3 Method Resolution Order 문서를 참조하기 바란다.

- super() 함수

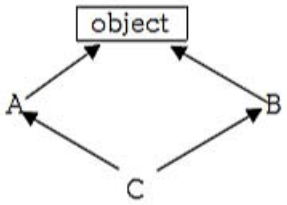
상위 클래스를 동적으로 얻어내는 super() 함수는 다중 상속으로 가면 클래스 상속 관계에 따라서 다른 결과를 내기도 한다. super() 함수는 mro() 함수가 출력하는 클래스 순서에 따라 super() 함수를 호출하는 현재 클래스의 다음 클래스를 결과로 반환한다. 예를 들어, mro() 함수가 [A, B, C]이고 super(0 함수를 호출하는 클래스가 B이면 super() 함수의 출력은 C가 된다. 예를들어 다음 코드의 A._init_() 메서드 안에서 super(0 함수는 object 클래스를 의미한다.

```
>>> class A(object):
...     def _init_(self):
...         print('A.__init__')
...         super().__init__()
...
>>> class C(A):
...     def _init_(self):
...         print('C.__init__')
...         super().__init__()
```



하지만, 다음 코드에서 A._init_() 메서드 안에서 super(0 함수는 B를 의미한다. 왜냐하면 클래스 C의 메서드 처리 순서는 [C, A, B, object]이기 때문이다.

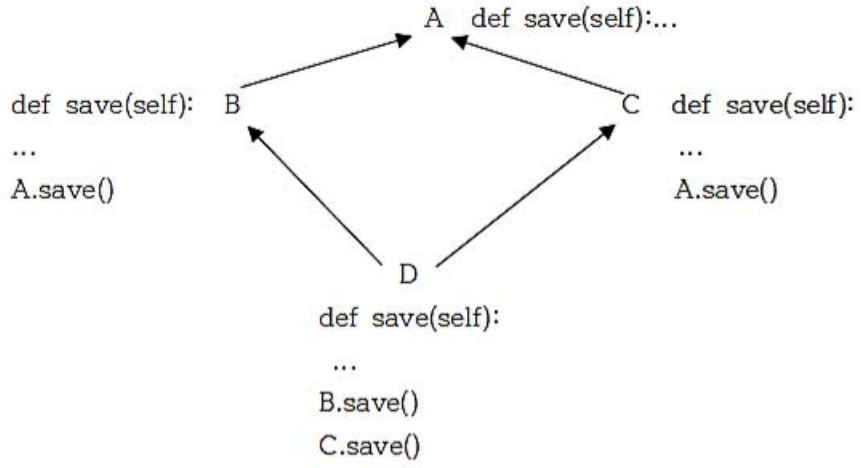
```
# diamond1.py

class A:
    def __init__(self):
        print('A.__init__')
        super().__init__()
class B:
    def __init__(self):
        print('B.__init__')
        super().__init__()
class C(A, B):
    def __init__(self):
        print('C.__init__')
        super().__init__()

c = C()
```



```
C.__init__
A.__init__
B.__init__
```

다른 예로 클래스의 데이터들을 파일에 저장하는 경우를 보자.

```
# diamond2.py

class A:
    def save(self):
        print('A save called')

class B(A):
    def save(self):
        print('B save called')
        A.save(self)

class C(A):
    def save(self):
        print('C save called')
        A.save(self)

class D(B, C):
    def save(self):
        print('D save called')
        B.save(self)
        C.save(self)

d = D()
d.save()
```



save() 호출 순서 : DBACA

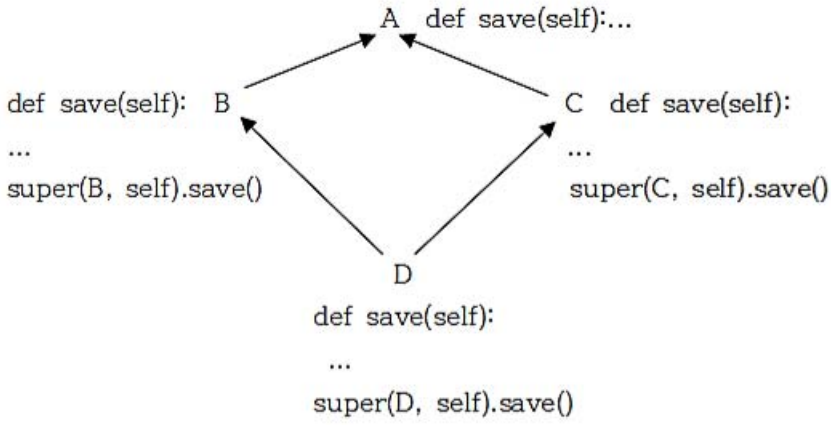
```
D save called
B save called
A save called
C save called
A save called
```

이것을 수행하면 A 클래스의 데이터가 두 번이나 저장된다. 즉, B에 의해서 한번, C에 의해서 한번이다.

```
D save called
B save called
A save called
C save called
A save called
```

이 문제는 다음과 같이 super() 함수를 사용하여 해결할수 있다. 각 클래스의 save() 함수는 mro() 함수의 순서에 따라 D.save(), B.save(), C.save(), A.save() 순으로 한번식만 호출된다.

```
# diamond3.py

class A:
    def save(self):
        print('A save called')

class B(A):
    def save(self):
        print('B save called')
        super(B, self).save()

class C(A):
    def save(self):
        print('C save called')
        super(C, self).save()

class D(B, C):
    def save(self):
        print('D save called')
        super(D, self).save()

d = D()
d.save()
```



save() 호출 순서 : DBCA

```
D save called
B save called
C save called
A save called
```

- 인스턴스 객체와 클래스의 관계를 확인하기

인스턴스 객체의 클래스를 알아내려면 _class_ 속성을 이용하고, 인스턴스 객체와 클래스와의 관계를 파악하려면 isinstance(instance, class) 메서드를 사용한다.

```
>>> isinstance(123, int)
True
```

클래스 인스턴스인 경우의 예를 보자.

```
>>> class A: pass
>>> class B: pass
>>> class C(B): pass
>>>
```

```
>>> c = C()
>>> c.__class__          #  인스턴스 객체의 클래스
<class  '_main_.C' >
>>> isinstance(c, A)    # c는 A의 인스턴스 객체가 아님
False
>>> isinstance(c, C)            # c는 C의 인스턴스 객체
True
>>> isinstance(c, B)            # c는 B의 인스턴스 객체
True
```

- 클래스 간의 상속 관계 알아내기

1. 두 클래스 간의 상속관계

두 클래스간의 상속관계를 알아내려면 issubclass() 내장함수를 사용한다.

```
>>> issubclass(C, B)
True
>>> issubclass(C, A)
False
```

2. 상위 클래스의 목록얻기

어떤 클래스의 상위 클래스를 알아보려면 _bases_ 멤버를 이용한다. _bases_는 어떠한 클래스로부터 직접 상속받았는지를 튜플로 알려주는 변수이다.

```
>>> class A: pass
>>> class B(A): pass
>>> class C(B): pass

>>> C.__bases__          # 바로 위의 상위 클래스 목록
(<class  '_main_.B' >,)
```

전체적으로 모든 상위 클래스의 목록을 얻으려면 재귀적인 코드를 직접 작성하거나 inspect 모듈의 sgemro() 함수를 사용한다.

```
>>> import inspect
>>> inspect.getmro(C)
(<class' _main_.C' >, <class' _main_.B' >, <class' _main_.A' >, <class  'object' >)
```

전체적으로 내포된 클래스 구조를 알고 싶으면 inspect 모듈의 getclasstree() 함수를 사용한다.

```
>>> class A: pass
>>> class AA(A): pass
>>> class B:pass
>>> class BB(B): pass
```

```
>>> class C(AA, BB): pass
>>>
>>> import pprint
>>> pp =  pprint.PrettyPrinter(indent=4)
>>> pp.pprint(inspect.getclasstree([C]))
[   (<class '__main__.AA'>, (<class '__main__.A'>,)),
    [(<class '__main__.C'>, (<class '__main__.AA'>, <class '__main__.BB'>))],
    (<class '__main__.BB'>, (<class '__main__.B'>,)),
    [(<class '__main__.C'>, (<class '__main__.AA'>, <class '__main__.BB'>))]]
```

출력 결과에서 맨 마지막 줄부터 보면, 클래스 C의 상위클래스는 AA와 BB이고, 클래스 B의 상위 클래스는 B이다. 계속해서 이처럼 해석하면 된다.

- 클래스 상속의 예

좀 더 실제적인 예를 가지고 종합적으로 클래스의 상속에 대해 살펴보자.

1. list와 dict 하위 클래스 만들기

내장 자료형인 리스트와 사전, 문자열 등은 사용자 클래스의 상위 클래스가 될수 있다. 리스트의 하위 클래스로 스택 클래스를 만들어보자. 클래스는 정말 간단하다. push 연산은 리스트의 append() 메서드와 같으므로 이름만 정의해 준다. pop 연산은 원래 리스트에 있는 pop() 메서드를 사용한다.

```
>>> class Stack(list):          # 클래스 정의
...     push = list.append
>>> s = Stack()                 # 인스턴스 객체 생성
>>> s.push(4)                   # push 연산
>>> s.push(5)
>>> s
[4, 5]
>>> s = Stack([1, 2, 3])
>>> s. push(5)
>>> s
[1, 2, 3, 4, 5]
>>> s.pop( )                    # pop 연산
5
>>> s.pop( )
4
>>> s
[1, 2, 3]
```

이번엔 리스트를 확장하여 큐클래스를 만들어 보자. 큐에 데이터를 추가하고 꺼내는 enquenue() 와 dequeue() 메서드를 추가한다. enqueue() 메서드는 append() 메서드를 호출한 것과 같고, dequeue() 메서드는 pop(0) 메서드를 호출한 것과 같다. 다음과 같이 큐를 정의한다.

```
>>> class Queue(list):
...     enqueue = list.append
...     def dequeue(self):
...             return self.pop(0)
...
>>> q = Queue()
>>> q.enqueue(1)                # 데이터 추가
>>> q.enqueue(2)
>>> q
[1, 2]
>>> q.dequeue()                 # 데이터 꺼내기
1
>>> q.dequeue()                 # 데이터 꺼내기
2
```

다음 예는 사전 내장 객체를 확장한 것으로 keys() 메서드가 정렬된 키들의 리스트를 넘겨준다.

```
>>> class MyDict(dict):
...             def keys(self):         # 정렬해서 반환한다.
...                     L = super().keys()      # 혹은 L = dict.keys(self)
...                     return sorted(L)
...
>>> d = MyDict({'one':1, 'two':2, 'three':3, 'four':4})
>>> d.keys()
['four', 'one', 'three', 'two']
```

2. 다중 스레드의 생성

threading 모듈을 이용하면 스레드 객체를 생성할 수 있다. 이 클래스의 하위 클래스를 정의하고 run() 메서드만 정의하면, 스레드를 쉽게 사용할수 있다. 다음예를 살펴보자.

```
# multithread.py

import time
from threading import *                         # 스레드 클래스를 제공하는 모듈 : threading
class MyThread(Thread):                         # 하위 클래스 MyThread를 정의한다.
    def __init__(self):
        super().__init__()              # 기반 클래스의 초기화 루틴을 불러야 한다.

    def run(self):                              # 실제적으로 실행을 위해서 정의해야 할 부분이다.
        for el in range(10):            # 10번 반복한다.
            print('{}=>{}\n'.format(self.getName(), el), end=' ')
            time.sleep(0.01)            # 0.01초 대기
```

```
thread1 = MyThread()                    # 스레드 객체(인스턴스) thread1 생성
thread2 = MyThread()                    # 스레드 객체(인스턴스) thread2 생성
thread1.start()                                 # 스레드 실행 시작. run() 메서드가 호출된다.
thread2.start()
```

코드를 실행한 결과는 다음과 같다.

```
Thread-6=>0
Thread-7=>0
  Thread-7=>1
 Thread-6=>1
 Thread-7=>2
 Thread-6=>2
 Thread-7=>3
 Thread-6=>3
 Thread-7=>4
 Thread-6=>4
 Thread-7=>5
 Thread-6=>5
 Thread-7=>6
 Thread-6=>6
 Thread-7=>7
 Thread-6=>7
 Thread-7=>8
 Thread-6=>8
 Thread-7=>9
 Thread-6=>9
```

두 개의 스레드가 번갈아서 출력된 것을 알수 있다.

threading 모듈은 일반적으로 병행처리에 필요한 다양한 연산들, 예를 들어 조건변수, 이벤트, 록, 세마포어 등을 제공한다. 24장 프로세스 다루기에서 자세한 내용을 살펴볼수 있다.

3. 간단한 명령어 해석기 설계

만일 독자가 어떤 시스템을 설계하고, 일반 사용자로 하여금 그것을 사용하게 할 때 간단한 명령어 해석기가 필요하다면 파이썬이 쉽게 해결해 준다. cmd 모듈의 cmd 클래스는 독자가 정의하는 명령어를 쉽게 만들도록 도와준다. 독자는 그저 다음 내용을 하면 된다.

* 하위 클래스를 하나 만든다(MyCmd)
* 상위 클래스의 생성자 루틴을 수행한다.
* 프롬프트 모양을 정한다(self.prompt)
* 나머지 독자가 필요로 하는 명령어에 필요한 멤버를 만든다.

명령어를 정의하는 방법은 메서드를 만드는 것이다. 예를들어 add란 명령어를 만들고 싶으면 do_add란 메서드를 정의하는 것이다. add명령어에 대한 도움말 메서드는 help.add이다. do_add는 self를 제외한 한 개의 인수를 받는데, 이 인수는 독자가 add 명령과 함께 넣어준 나머지 문자열이다. 실행예를 보면 알수 있다.

MyCmd 클래스는 간단하게 세 개의 명령(add, show, EOF)과 세 개의 도움말(help_add, help_show, help_EOF)이 있다.

```
# mycmdline.py

import sys, cmd

class MyCmd(cmd.Cmd):
    def __init__(self):
        super().__init__()
        self.prompt = "--> "
        self.list = []
    def do_add(self, x):                        # add 명령
        if x and (x not in self.list):
            self.list.append(x)
    def help_add(self):                 # add 도움말
        print('help for add')
    def do_show(self, x):               # show 명령
        print(self.list)
    def help_show(self):                        # show 도움말
        print('help for show')
    def do_EOF(self, x):                        # EOF 키가 입력되었을 때(Ctrl+Z 또는 Ctrl+D)
        sys.exit(0)
    def help_EOF(self):                 # EOF 키 도움말
        print('quit the program')

if __name__ == '__main__':
    c = MyCmd()
    c.cmdloop()
```

코드를 실행한 결과는 다음과 같다.

```
-->help

Documented commands (type help <topic>):
========================================
EOF  add  help  show
```

```
-->help add
help for add
-->help show
help for show
-->show
[]
-->add a
-->show
['a']
-->add b
-->show
['a', 'b']
-->
```

15.2 다형성

다형성이란 '여러 형태를 가진다.' 는 의미의 그리스어에서 유래된 말로, 상속 관계에서 다른 클래스의 인스턴스 객체들이 같은 멤버 함수의 호출에 대해 각각 다르게 반응하도록 하는 기능이다.

예를 들어, a+b라는 연산을 수행할 때 + 연산은 객체 a와 b에 따라 동적으로 결정된다. 이미 연산자 중복에서 배운것처럼 a._add_(b)가 호출되는 것이다. 객체 a와 b가 정수이면 정수형 객체의 _add_ 메서드를, 문자열이면 문자열 객체의 _add_가 내부적으로 호출된다.(가상함수) 이처럼 동일한 이름의 연산자라 해도 객체에 따라 다른 메서드가 호출되는 것이 다형성이다.

또다른 클래스의 예로 다음과 같은 동물 관계를 살펴보자.

```
>>> class Animal:
...     def cry(self):
...             print('...')
...
>>> class Dog(Animal):
...     def cry(self):
...             print('멍멍')
...
>>> class Duck(Animal):
...     def cry(self):
...             print('꽥꽥')
...
>>> class Fish(Animal):
...     pass

>>> for each in (Dog(), Duck(), Fish()):
...     each.cry()
```

메서드 cry() 호출에 대해서 각 인스턴스 객체에 해당하는 cry() 메서드가 사용되고 있다. 코드를 실행한 결과는 다음과 같다.

```
멍멍
꽥꽥
...
```

다형성은 객체의 종류에 관계없이 하나의 이름으로 원하는 유사한 작업을 수행시킬수 있으므로 프로그램의 작성과 코드의 이해를 쉽게 해준다.

15.3 캡슐화

일반적으로 객체지향 언어에서 캡슐화란 필요한 메서드와 멤버를 하나의 단위로 묶어 외부에서 접근 가능하도록 인터페이스를 공개하는 것을 의미한다. 파이썬에서 캡슐화는 코드를 묶는 것(캐피지화하는 것)을 의미하며, 반드시 정보를 숨기는 것이 아님을 유의하기 바란다. 캡슐화는 완전히 내부 정보가 숨겨지는 방식으로 구현될 수도 있고, 외부에서 접근 가능하도록 공개된 방식으로 구현될 수도 있다. 정보를 숨기는 것에 정보 은닉이라는 용어를 사용한다.

파이썬에서는 주로 공개방식의 캡슐화를 주로 사용한다. 파이썬의 모든 정보는 기본적으로 공개되어 있다. 관례로 내부적으로만 사용하거나 차기 버전에서 변경 가능성 있는 이름은 밑줄(_)로 시작한다. 이것이 이름을 완전히 숨기는 것을 의미하지는 않는다. 단지 변수가 내부용이라는 것을 알릴뿐이다.

만일 변수나 메서드 이름을 구태여 숨기고자 하면 두 개의 민줄(_)로 시작하는 이름으로 정하면 된다.

```
>>> class A:
...     def __f(self):
...         print('__f called')
...
>>> a = A()
>>> a.__f()
AttributeError                            Traceback (most recent call last)
<ipython-input-5-374f6d3cf95c> in <module>
      1 a = A()
----> 2 a.__f()

AttributeError: 'A' object has no attribute '__f'

>>> dir(a)
['_A__f', ... ]
>>> a._A__f()
__f called
```

결국, 파이썬에서 제공하는 정보 은닉 기능은 크지 않다. 다른 언어에서는 정보 은닉을 중요한 기능으로 간주하여 활용하지만, 파이썬에서는 그렇지 않다. 파이썬에서 외부용과 내부용 구분은 이름짓기로 구분할뿐 그이상의 제약은 가하지 않는다.

15.4 위임

위임은 상속체계 대신에 사용되는 기법으로 어떤 객체가 자신이 처리할수 없는 베시지(메서드 호출)를 받으면, 해당 메시지를 처리할 수 있는 다른 객체에 전달하는 것이다. 또는 다른 객체의 메서드 호출을 중간에 있는 클래스가 대신 위임받아 처리하는 것이다. 위임은 상속체계보다 융통성이 있고 일반적이다.

파이썬에서 위임은 일반적으로 _getattr_() 메서드로 구현된다. 이 메서드는 정의되지 않는 속성을 참조하려고 했을 때 호출된다. 사용법은 다음과 같다.

```
_getattr_(self, name)
```

하나의 인수 name을 가지며, 참조하는 속성 이름이 이 인수를 통해 전달된다. 이 메서드는 구해진 속성 값을 전달하거나, 속성 값이 없다는 것을 나타내기 위해 AttributeError 예외를 발생시켜야 한다.

```
# delegation.py

class Delegation:
    def __init__(self, data):
        self.stack = data
    def __getattr__(self, name):        # 정의되지 않은 속성을 참조할 때 호출한다.
        print('Delegating {} '.format(name), end=' ')
        return getattr(self.stack, name)              # self.stack의 속성을 대신 이용한다.

a = Delegation([1,2,3,1,5])
print(a.pop())
print(a.count(1))
```

여기서, _getattr_() 메서드의 인수 name은 문자열로 된 속성 이름이다. 예를 들어, a.count(1)가 호출되면 'count'가 정의되어 있지 않으므로 a._getattr_('conut')가 호출되고, 반환 값으로 결국은 (a._getattr__('count'))(1)이 수행된다. 결과적으로 self.stack.count(1)을 호출하게 되는 것이다.

코드를 실행한 결과는 다음과 같다.

```
Delegating pop  5
Delegating count  2
```

하지만, _getattr_() 메서드가 모든 메서드를 대신 호출해 주는 것은 아니다. _getattr_() 메서드는 _getitem_과 _repr_, _len_처럼 _로 시작하는 메서드를 필요로 하는 이름들은 잡아내지 못한다. 예를들어, _getitem_을 요구하는 a[0]을 수행하면 _getattr_() 메서드가 수행되지 않고 AttributeError 에러가 발생한다.