파이썬 프로그래밍

상속과 다형성

한국기술교육대학교
온라인평생교육원

▣ 클래스 상속

1. 클래스 상속과 이름 공간의 관계

- 상속의 이유
  - 코드의 재사용
  - 상속받은 자식 클래스는 상속을 해준 부모 클래스의 모든 기능을 그대로 사용
  - 자식 클래스는 필요한 기능만을 정의하거나 기존의 기능을 변경할 수 있음

```
class Person:
    def __init__(self, name, phone=None):
        self.name = name
        self.phone = phone
    def __str__(self):
        return '<Person %s %s>' % (self.name, self.phone)
```

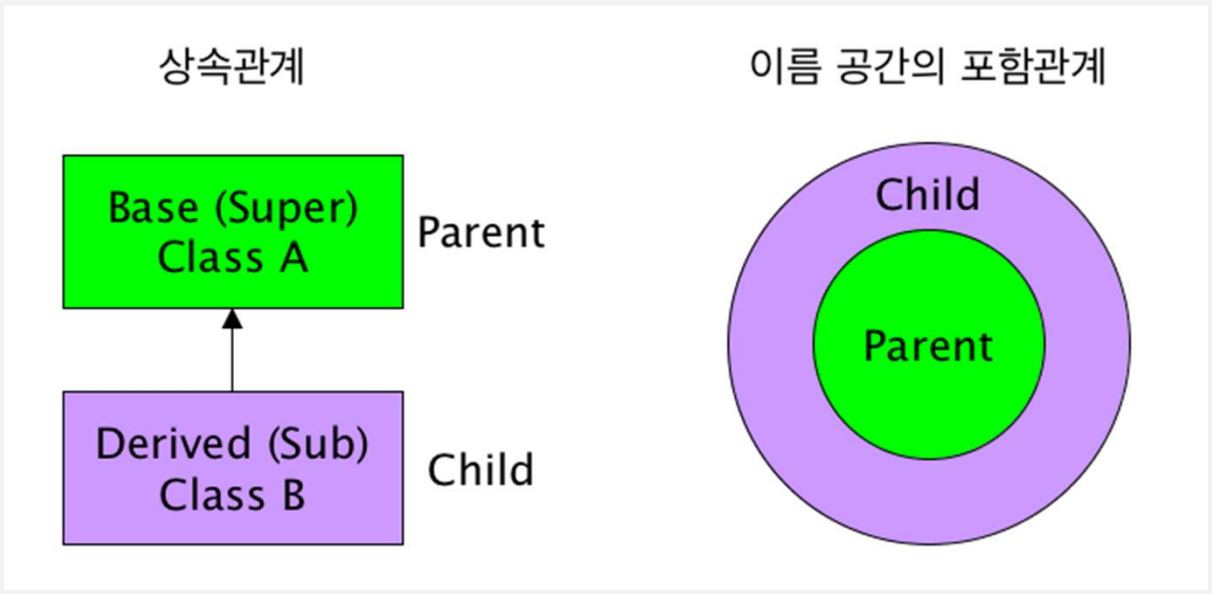
```
class Employee(Person):                    # 괄호 안에 쓰여진 클래스는 슈퍼클래스를 의미한다.
    def __init__(self, name, phone, position, salary):
        Person.__init__(self, name, phone) # Person클래스의 생성자 호출
        self.position = position
        self.salary = salary
```

- person의 내용이 그대로 Employee에서 활용 가능
- self → 인스턴스 메소드가 기본적으로 가져가는 인자
- %s %s → 문자열 포맷팅
- class 이름 (상속 받으려는 class 이름)
- __init__를 사용하여 생성자 정의 → 필요한 기능만을 새로 정의

▣ 클래스 상속

1. 클래스 상속과 이름 공간의 관계

- 이름 공간의 포함관계
  - 자식 클래스 > 부모 클래스



- Person = 부모 클래스
- Employee = 자식 클래스
- Derived : 유도가 되어진, 상속되어진
- 자식 클래스 > 부모 클래스
- 자식 클래스가 부모 클래스를 가지며 더 많은 식별자를 가짐

▣ 클래스 상속

1. 클래스 상속과 이름 공간의 관계

```
p1 = Person('홍길동', 1498)
print p1.name
print p1

print

m1 = Employee('손창희', 5564, '대리', 200)
m2 = Employee('김기동', 8546, '과장', 300)
print m1.name, m1.position  # 슈퍼클래스와 서브클래스의 멤버를 하나씩 출력한다.
print m1
print m2.name, m2.position
print m2
```

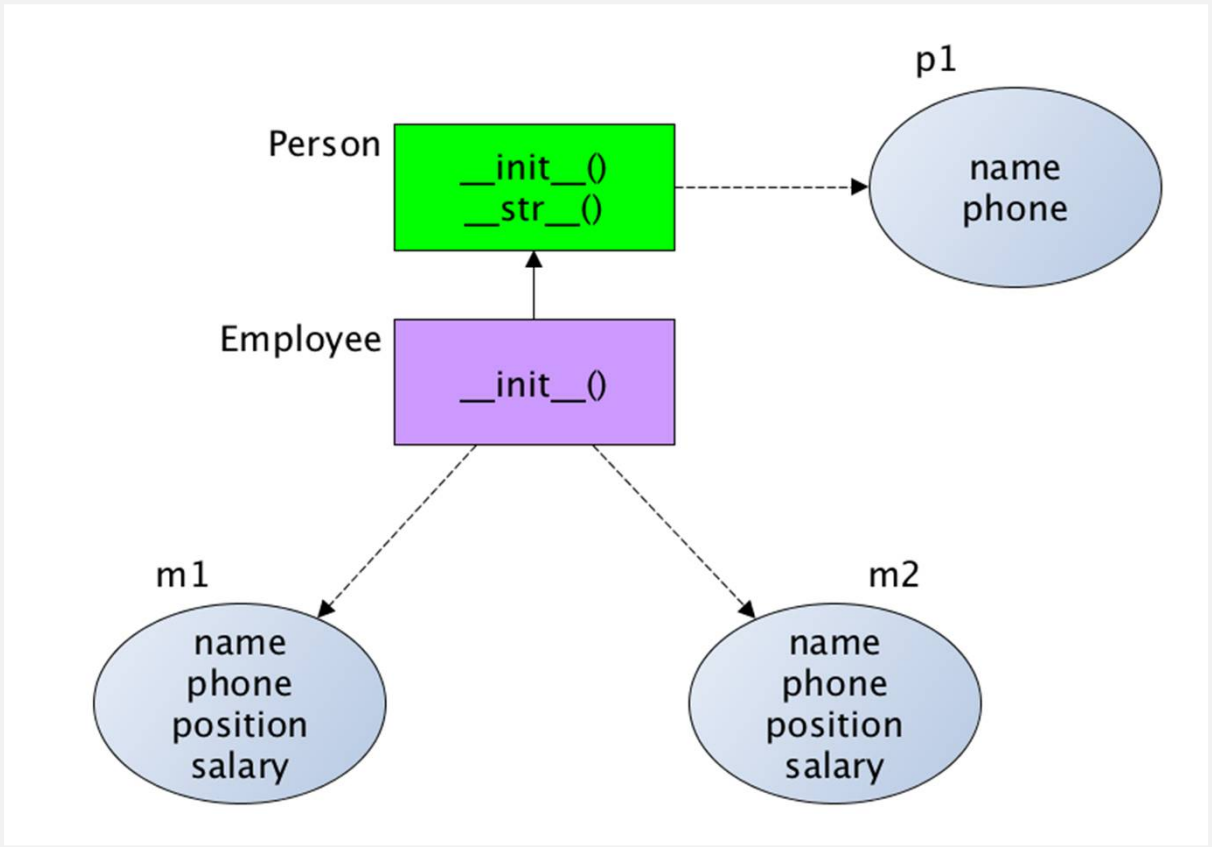
```
홍길동
<Person 홍길동 1498>

손창희 대리
<Person 손창희 5564>
김기동 과장
<Person 김기동 8546>
```

- p1 → __str__ 메소드 호출
- m1 은 employee 객체이기 때문에 그 안에서 __str__ 찾음
- 없으면 자연스럽게 부모 클래스에 가서 __str__찾음

▣ 클래스 상속

1. 클래스 상속과 이름 공간의 관계



- 자식 클래스에 있는 init이 부모 클래스의 init을 overiding 함
- override: 재정의

▣ 클래스 상속

2. 생성자 호출

- 서브 클래스의 생성자는 슈퍼 클래스의 생성자를 자동으로 호출하지 않는다.

```
class Super:
    def __init__(self):
        print 'Super init called'

class Sub(Super):
    def __init__(self):
        print 'Sub init called'

s = Sub()
```

```
Sub init called
```

- 자식 클래스에 있는 init이 부모 클래스의 init을 overiding 함

▣ 클래스 상속

2. 생성자 호출

- 서브 클래스의 생성자에서 슈퍼 클래스의 생성자를 명시적으로 호출해야 한다.

```
class Super:
    def __init__(self):
        print 'Super init called'

class Sub(Super):
    def __init__(self):
        Super.__init__(self)   # 명시적으로 슈퍼클래스의 생성자를 호출한다.
        print 'Sub init called'

s = Sub()
```

```
Super init called
Sub init called
```

- sub 클래스의 생성자가 호출되자마자 super 클래스의 init 호출
- 명시적으로 super 클래스의 생성자 호출

▣ 클래스 상속

2. 생성자 호출

- 서브 클래스에 생성자가 정의되어 있지 않은 경우에는 슈퍼 클래스의 생성자가 호출된다.

```
class Super:
    def __init__(self):
        print 'Super init called'

class Sub(Super):
    pass

s = Sub()
```

```
Super init called
```

- 내용이 없다면 무조건 pass 써야 함
- sub 클래스의 init이 없으면 super 클래스의 init 만 호출

▣ 클래스 상속

3. 메소드의 대치(메소드 오버라이드 - Override)

- 서브 클래스에서 슈퍼 클래스에 정의된 메소드를 재정의하여 대치하는 기능

```
class Person:
    def __init__(self, name, phone=None):
        self.name = name
        self.phone = phone
    def __str__(self):
        return '<Person %s %s>' % (self.name, self.phone)

class Employee(Person):
    def __init__(self, name, phone, position, salary):
        Person.__init__(self, name, phone)
        self.position = position
        self.salary = salary

p1 = Person('gslee', 5284)
m1 = Employee('kslee', 5224, 'President', 500)

print p1
print m1
```

```
<Person gslee 5284>
<Person kslee 5224>
```

- m1도 str을 호출하나 Employee 클래스 안에 없어 부모 클래스 활용
- Employee도 str을 가지게 되면 부모 클래스의 str 위로 올라탐
- 부모 클래스의 str은 무시, 자식 클래스의 str을 호출

▣ 클래스 상속

3. 메소드의 대치(메소드 오버라이드 - Override)

```
class Employee(Person):
    def __init__(self, name, phone, position, salary):
        Person.__init__(self, name, phone)
        self.position = position
        self.salary = salary
    def __str__(self):
        return '<Employee %s %s %s %s>' % (self.name, self.phone, self.position,
                    self.salary)

p1 = Person('gslee', 5284)
m1 = Employee('kslee', 5224, 'President', 500)

print p1
print m1
```

```
<Person gslee 5284>
<Employee kslee 5224 President 500>
```

- m1도 str을 호출하나 Employee 클래스 안에 없어 부모 클래스 활용
- Employee도 str을 가지게 되면 부모 클래스의 str 위로 올라탐
- 부모 클래스의 str은 무시, 자식 클래스의 str을 호출

▣ 클래스 상속

4. 다형성(Polymorphism)

• 상속 관계 내의 다른 클래스들의 인스턴스들이 같은 멤버 함수 호출에 대해 각각 다르게 반응하도록 하는 기능
- 연산자 오버로딩도 다형성을 지원하는 중요한 기술
* 예를 들어, a와 b의 객체 형에 따라 a + b의 + 연산자 행동 방식이 변경되는 것

• 다형성의 장점
- 적은 코딩으로 다양한 객체들에게 유사한 작업을 수행시킬 수 있음
- 프로그램 작성 코드 량이 줄어든다.
- 코드의 가독성을 높혀준다.

• 파이썬에서 다형성의 장점
- 형 선언이 없다는 점에서 파이썬에서는 다형성을 적용하기가 더욱 용이하다.
- 실시간으로 객체의 형이 결정되므로 단 하나의 메소드에 의해 처리될 수 있는 객체의 종류에 제한이 없다.
* 즉, 다른 언어보다 코드의 양이 더욱 줄어든다.

- a 에 들어오는 형태에 따라서 + 연산자가 다르게 행동
- a에 클래스가 들어오면 이에 대응되는 __add__ 메소드가 호출
- 파이썬은 디폴트로 다형성이 잘 제공되고 있음

▣ 클래스 상속

4. 다형성(Polymorphism)

```
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()
```

```
멍멍
꽥꽥
...
```

- Animal에 있는 cry를 Dog의 cry가 override 함
- Fish는 cry가 없으므로 Animal의 내용을 그대로 사용
- in 뒤에는 튜플 안에 객체 3개 존재
- each의 객체 형이 동적으로 결정되므로 그때마다의 cry가 다르게 사용

파이썬 프로그래밍

상속과 다형성

한국기술교육대학교
온라인평생교육원

▣ 내장 자료형과 클래스의 통일

1. 리스트 서브 클래스 만들기

- 내장 자료형(list, dict, tuple, string)을 상속하여 사용자 클래스를 정의하는 것
  - 내장 자료형과 사용자 자료형의 차이를 없에고 통일된 관점으로 모든 객체를 다룰 수 있는 방안
- 클래스 정의는 새로운 자료형의 정의임

```
a = list()
print a
print dir(a)
```

```
<Person gslee 5284>
<Person kslee 5224>
```

- 상속을 받을 때 list를 상속 받음
- a = [] == a = list()
- list를 클래스 명으로 하여 ()를 붙여 인스턴스를 만듦
- 기본적으로 a는 일반적인 list
- 소문자 list가 클래스 명

▣ 내장 자료형과 클래스의 통일

1. 리스트 서브 클래스 만들기

- 아래 예제는 내장 자료형인 list를 상속하여 뺄셈 연산(-)을 추가함

```
class MyList(list):
    def __sub__(self, other):   # '-' 연산자 중복 함수 정의
        for x in other:
            if x in self:
                self.remove(x)     # 각 항목을 하나씩 삭제한다.
        return self

L = MyList([1, 2, 3, 'spam', 4, 5])
print L
print

L = L - ['spam', 4]
print L
```

```
[1, 2, 3, 'spam', 4, 5]

[1, 2, 3, 5]
```

- - 가 __sub__에 대응됨
- other → ['spam', 4]
- 상속받는 클래스의 객체는 부모 클래스의 타입과 동일
- L은 리스트이면서 1,2,3,'spam',4,5 원소를 가짐
- print L에 대응되는 __str__이 없음
- 하지만 클래스 list가 가지고 있는 _str__ 활용
- 내장자료형인 list에는 __sub__은 정의가 되어 있지 않음
- 따라서 MyList에서는 생성자를 더 해 더 풍부하게 만듦
- self → list
- L에는 append가 없지만 기존 list는 가지고 있으므로 수행됨

▣ 내장 자료형과 클래스의 통일

1. 리스트 서브 클래스 만들기

1) Stack 클래스 정의 예
- 슈퍼 클래스로 list 클래스를 지닌다.
- 즉, list 클래스를 확장하여 Stack 클래스를 정의함

```
class Stack(list):  # 클래스 정의
    push = list.append

s = Stack()         # 인스턴스 생성

s.push(4)
s.push(5)
print s
print

s = Stack([1,2,3])
s.push(4)
s.push(5)
print s
print

print s.pop()       # 슈퍼 클래스인 리스트 클래스의 pop() 메소드 호출
print s.pop()
print s
```

```
[4, 5]

[1, 2, 3, 4, 5]

5
4
[1, 2, 3]
```

▣ 내장 자료형과 클래스의 통일

1. 리스트 서브 클래스 만들기

- Stack 에는 push와 pop 연산자 존재
- push를 따로 정의 X → list가 가지고 있는 append를 공유
- print s → Stack 안 str 없으므로 list의 str 활용
- pop은 Stack에 없으나 list에 존재하여 활용 가능

▣ 내장 자료형과 클래스의 통일

1. 리스트 서브 클래스 만들기

2) Queue 클래스 정의 예
- 슈퍼 클래스로 역시 list를 지닌다.
- 즉, list 클래스를 확장하여 Queue 클래스를 정의함

```
class Queue(list):
    enqueue = list.append
    def dequeue(self):
        return self.pop(0)

q = Queue()
q.enqueue(1)      # 데이터 추가
q.enqueue(2)
print q

print q.dequeue() # 데이터 꺼내기
print q.dequeue()
```

```
[1, 2]
1
2
```

- enqueue() : 리스트 맨 뒤에 원소 추가 = append
- dequeue() : 리스트 맨 앞에서 원소 꺼냄

▣ 내장 자료형과 클래스의 통일

2. 사전 서브 클래스 만들기

```
a = dict()
print a
print dir(a)
```

```
{}
['__class__', '__cmp__', '__contains__', '__delattr__', '__delitem__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__getitem__', '__gt__', '__hash__', '__init__',
'__iter__', '__le__', '__len__', '__lt__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__setattr__', '__setitem__', '__sizeof__', '__str__', '__subclasshook__', 'clear',
'copy', 'fromkeys', 'get', 'has_key', 'items', 'iteritems', 'iterkeys', 'itervalues', 'keys',
'pop', 'popitem', 'setdefault', 'update', 'values', 'viewitems', 'viewkeys', 'viewvalues']
```

- dict 는 클래스로, 생성자를 호출하여 a에 넣어 사전 만듦
- a에는 사전에서 많이 호출할 수 있는 메소드들 포함

▣ 내장 자료형과 클래스의 통일

2. 사전 서브 클래스 만들기

- 아래 예제는 keys() 메소드를 정렬된 키값 리스트를 반환하도록 재정의한다.

```
class MyDict(dict):
    def keys(self):
        K = dict.keys(self) # 언바운드 메소드 호출 --> K = self.keys() 라고 호출하면
                                무한 재귀 호출
        K.sort()
        return K

d = MyDict({'one':1, 'two':2, 'three':3})
print d.keys()
print

d2 = {'one':1, 'two':2, 'three':3}
print d2.keys()
```

```
['one', 'three', 'two']

['three', 'two', 'one']
```

- keys는 기존 dict에 존재하지만 재정의하여 사용
- dic.keys() → 클래스의 keys를 부르므로 언바운드 매소드
- self.keys를 부르면 dict의 keys가 아닌 구현하고 있는 keys를 매핑
- 무한루프로 무한 재귀 호출 → error 발생

파이썬 프로그래밍

상속과 다형성

한국기술교육대학교
온라인평생교육원

▣ 상속 관계에 있는 클래스들의 정보 획득

1. 객체가 어떤 클래스에 속해 있는지 확인하기

- 객체의 자료형 비교 방법 I (전통적 방법)

```
import types

print type(123) == types.IntType
print type(123) == type(0)
```

```
True
True
```

- 1,2,3은 정수이므로 IntType을 가지고 있음
- type() → 인수의 타입을 알아보는 내장함수
- types라는 모듈을 import하여 알아보는 방법 → 많이 쓰이지 X

- 객체의 자료형 비교 방법 II (새로운 방법)
  - isinstance() 내장 함수와 기본 객체 클래스 사용

```
print isinstance(123, int)
print int
```

```
 True
<type 'int'>
```

- is로 시작하는 매소드 → 반드시 true, false로 반환
- isinstance(123, int) → 123이 int 타입의 인스턴스인지 물음
- isinstance(객체, 타입)
- 클래스 이름은 하나하나가 다 타입
- int도 하나의 클래스

▣ 상속 관계에 있는 클래스들의 정보 획득

1. 객체가 어떤 클래스에 속해 있는지 확인하기

- 서브 클래스의 인스턴스는 슈퍼 클래스의 인스턴스이기도 하다.

- obj가 클래스 A, B, C의 인스턴스인지 확인
- 클래스 C는 B의 상속을 받아 B, C 클래스의 인스턴스 모두 사용

▣ 상속 관계에 있는 클래스들의 정보 획득

1. 객체가 어떤 클래스에 속해 있는지 확인하기

```
class A:
    pass

class B:
    def f(self):
        pass

class C(B):
    pass

def check(obj):
    print obj, '=>',
    if isinstance(obj, A):
        print 'A',
    if isinstance(obj, B):
        print 'B',
    if isinstance(obj, C):
        print 'C',
    print

a = A()
b = B()
c = C()

check(a)
check(b)
check(c)
```

```
<__main__.A instance at 0x10de34e60> => A
<__main__.B instance at 0x10de34e18> => B
<__main__.C instance at 0x10de34cf8> => B C
```

▣ 상속 관계에 있는 클래스들의 정보 획득

2. 클래스 간의 상속 관계 알아내기

- issubclass() 내장 함수 활용

```
class A:
    pass

class B:
    def f(self):
        pass

class C(B):
    pass

def check(obj):
    print obj, '=>',
    if issubclass(obj, A):
        print 'A',
    if issubclass(obj, B):
        print 'B',
    if issubclass(obj, C):
        print 'C',
    print

check(A)
check(B)
check(C)
```

```
__main__.A => A
__main__.B => B
__main__.C => B C
```

▣ 상속 관계에 있는 클래스들의 정보 획득

2. 클래스 간의 상속 관계 알아내기

- issubclass(클래스, 클래스)
- C는 B를 상속 받아 B클래스의 인스턴스 모두 사용