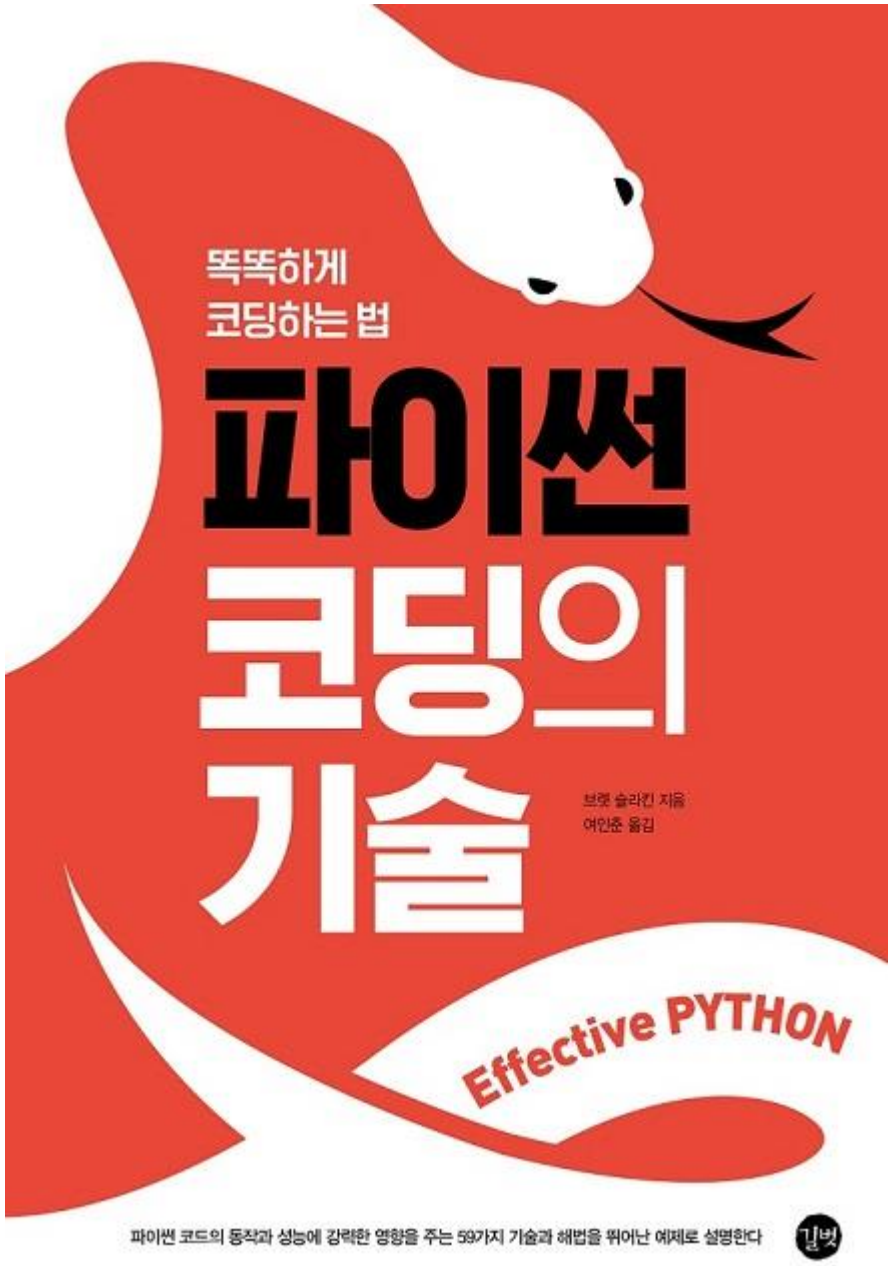
Effective python

강서연

2020. 02. 04



Scope

```
lst = [1, 2, 3]
def foo1():
    lst.append(5)
foo1()
print(lst)

def foo2():
    lst +=[5]

foo2()
```

```
[1, 2, 3, 5]
Traceback (most recent call last):
  File "C:/Users/PLAS/Desktop/3_winter/006764/test.py", line 10, in <module>
    foo2()
  File "C:/Users/PLAS/Desktop/3_winter/006764/test.py", line 8, in foo2
    lst +=[5]
UnboundLocalError: local variable 'lst' referenced before assignment
```

Foo1은 lst에 할당하지 X
Foo2의 lst+=[5]는 실제로 lst= lst +[5]이므로 할당 시도

➔ 할당 시점에는 컴파일러가 로컬 변수로 간주함.

Scope

- 한 스코프에서 변수에 할당하면, 해당 변수는 해당 스코프에서 local 변수가 됨.
- 그리고 다른 비슷한 이름의 변수는 가려짐.

- Foo에서 x에 할당하는데, 먼저 쓰인 print(x)에서 초기화되지 않은 지역변수를 프린트하려고 해서 오류 발생

```
>>> x = 10
>>> def bar():
...     print x
>>> bar()
10
```

works, but this code:

```
>>> x = 10
>>> def foo():
...     print x
...     x += 1
```

results in an UnboundLocalError:

```
>>> foo()
Traceback (most recent call last):
  ...
UnboundLocalError: local variable 'x' referenced before assignment
```

```
>>> x = 10
>>> def foobar():
...     global x
...     print x
...     x += 1
>>> foobar()
10
```

[참고]

Scope

```python
# Example 3
def sort_priority2(numbers, group):
    found = False
    def helper(x):
        if x in group:
            group = {2, 3, 5}
            return (0, x)
        return (1, x)
    numbers.sort(key=helper)
    return found
```

```
Traceback (most recent call last):
  File "C:/Users/PLAS/Desktop/3_winter/006764/item_15.py", line 52, in <module>
    found = sort_priority2(numbers, group)
  File "C:/Users/PLAS/Desktop/3_winter/006764/item_15.py", line 47, in sort_priority2
    numbers.sort(key=helper)
  File "C:/Users/PLAS/Desktop/3_winter/006764/item_15.py", line 43, in helper
    if x in group:
UnboundLocalError: local variable 'group' referenced before assignment
```

Scope – lambda

```
>>> squares = []
>>> for x in range(5):
...     squares.append(lambda: x**2)
```

```
>>> squares[2]()
16
>>> squares[4]()
16
```

- 예상하는 결과값 : [0,1,4,9,16]

- X가 람다에 로컬변수가 아니고 바깥 스코프에서 정의됨.

➔정의될 때가 아닌, 람다가 호출될 때 변수에 access

```
for x in range(5):  x: 4
    squares.append(lambda : x**2)

squares[2]()
squares[4]()
```

Scope – lambda

```
>>> y = 10
>>> (lambda x: x + y)(1)
11
```

매개변수 | 반환값

Map의 매개변수 중function | map의 매개변수로 쓰인 iterable 객체

```
>>> list(map(lambda x: x + 10, [1, 2, 3]))
[11, 12, 13]
```

- 람다 로컬변수에 저장해야 함.
  - Loop돌며 Sqares에 lambda가 appen될 때 global변수 X의 값이 아닌 람다의 지역변수 n에 0~4가 저장됨.

```
squares = []
for x in range(5):
    squares.append(lambda n=x: n**2)
```

```
>>> squares[2]()
4
>>> squares[4]()
16
```

BetterWay25. super로 부모 클래스를 초기화하자

- 상속

```
class 부모클래스:
    ...내용...

class 자식클래스(부모클래스):
    ...내용...
```

- 기존에는 자식 클래스에서 부모클래스의 __init__ 을 직접 호출하는 방법으로 부모 클래스 초기화 했었음.

```
# Example 1
class MyBaseClass(object):
    def __init__(self, value):
        self.value = value

class MyChildClass(MyBaseClass):
    def __init__(self):
        MyBaseClass.__init__(self, 5)

    def times_two(self):
        return self.value * 2

foo = MyChildClass()
print(foo.times_two())
```

```
10
```

BetterWay25. super로 부모 클래스를 초기화하자

- Python 다중상속 가능
  - 문제점1. __init__호출순서가 명시되지 않음

```
class MyBaseClass(object):
    def __init__(self, value):
        self.value = value


# Example 2
class TimesTwo(object):
    def __init__(self):
        self.value *= 2

class PlusFive(object):
    def __init__(self):
        self.value += 5


# Example 3
class OneWay(MyBaseClass, TimesTwo, PlusFive):
    def __init__(self, value):
        MyBaseClass.__init__(self, value)
        TimesTwo.__init__(self)
        PlusFive.__init__(self)


# Example 4
foo = OneWay(5)
print('First ordering is (5 * 2) + 5 =', foo.value)
```

```
# Example 5
class AnotherWay(MyBaseClass, PlusFive, TimesTwo):
    def __init__(self, value):
        MyBaseClass.__init__(self, value)
        TimesTwo.__init__(self)
        PlusFive.__init__(self)


# Example 6
bar = AnotherWay(5)
print('Second ordering still is', bar.value)
```

```
First ordering is (5 * 2) + 5 = 15
Second ordering still is 15
```

BetterWay25. super로 부모 클래스를 초기화하자

- Python 다중상속 가능
  - 문제점2. 다이아몬드 상속 : 서브클래스가 계층 구조에서 같은 슈퍼클래스를 둔 서로 다른 두 클래스에서 상속받을 때 발생함.

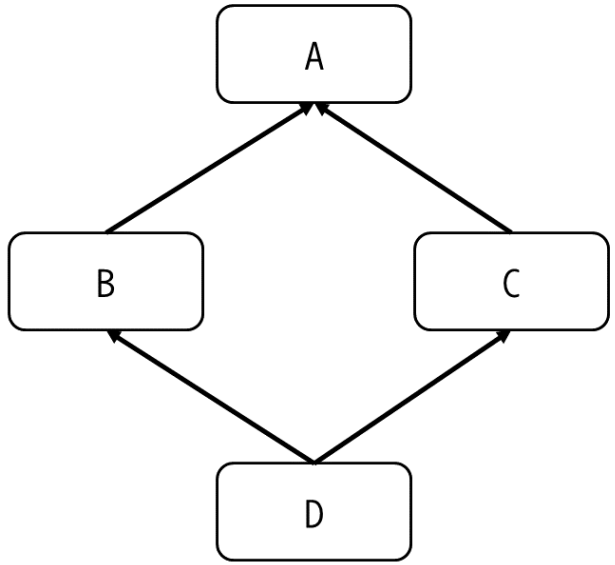
```
# Example 7
class TimesFive(MyBaseClass):
    def __init__(self, value):
        MyBaseClass.__init__(self, value)
        self.value *= 5

class PlusTwo(MyBaseClass):
    def __init__(self, value):
        MyBaseClass.__init__(self, value)
        self.value += 2


# Example 8
class ThisWay(TimesFive, PlusTwo):
    def __init__(self, value):
        TimesFive.__init__(self, value)
        PlusTwo.__init__(self, value)

foo = ThisWay(5)
print('Should be (5 * 5) + 2 = 27 but is', foo.value)
```

```
Should be (5 * 5) + 2 = 27 but is 7
```



BetterWay25. super로 부모 클래스를 초기화하자

- Super 내장함수로 메서드 해석순서(MRO) 정의
  - 어떤 슈퍼클래스부터 초기화할지 결정(깊이우선, 왼쪽에서 오른쪽으로 등)
  - 다이아몬드계층구조에 있는 공통 슈퍼클래스 단 한번만 실행

```
class MyBaseClass(object):
    def __init__(self, value):
        self.value = value

class TimesFiveCorrect(MyBaseClass):
    def __init__(self, value):
        super(TimesFiveCorrect, self).__init__(value)
        self.value *= 5

class PlusTwoCorrect(MyBaseClass):
    def __init__(self, value):
        super(PlusTwoCorrect, self).__init__(value)
        self.value += 2

class GoodWay(TimesFiveCorrect, PlusTwoCorrect):
    def __init__(self, value):
        super(GoodWay, self).__init__(value)

before_pprint = pprint
pprint(GoodWay.mro())
from pprint import pprint
pprint(GoodWay.mro())
pprint = pprint
```

Python2에선 생략 시 오류발생

```
Should be 5 * (5 + 2) = 35 and is 35


[<class '__main__.GoodWay'>,
 <class '__main__.TimesFiveCorrect'>,
 <class '__main__.PlusTwoCorrect'>,
 <class '__main__.MyBaseClass'>,
 <class 'object'>]
```

BetterWay25. super로 부모 클래스를 초기화하자

- Python2에선 super호출하면서 현재 클래스의 이름을 지정해야 함. 클래스이름 변경 시 모든 코드 수정해야함.
- Python3에서는 super에 인수 생략 가능
  - Super 항상 사용하는게 좋음.

```
# Example 12
class Explicit(MyBaseClass):
    def __init__(self, value):
        super(__class__, self).__init__(value * 2)

class Implicit(MyBaseClass):
    def __init__(self, value):
        super().__init__(value * 2)

assert Explicit(10).value == Implicit(10).value
```

BetterWay26. 믹스인 유틸리티 클래스에만 다중 상속을 사용하자.

- 다중상속으로 얻는 편리함과 캡슐화가 필요하다면 믹스인(mix-in)사용을 고려.
- Mixin
  - 특정한 클래스에 상속을 통해 새로운 속성이나 새로운 기능을 추가하는 것
  - 반복코드 최소화, 재사용성 극대화
  - 가장 오른쪽이 상위클래스. 메소드 명이 같을 경우 가장 하위 클래스가 적용됨.

```
class Mixin1(object):
    def test(self):
        print "Mixin1"

class Mixin2(object):
    def test(self):
        print "Mixin2"

class MyClass(BaseClass, Mixin1, Mixin2):
    pass
```

```
>>> obj = MyClass()
>>> obj.test()
Mixin1
```

BetterWay26. 믹스인 유틸리티 클래스에만 다중 상속을 사용하자.

- 바이너리 트리를 딕셔너리로 표현하려고 믹스인 사용하는 예제

```python
# Example 1
class ToDictMixin(object):
    def to_dict(self):
        return self._traverse_dict(self.__dict__)


# Example 2
    def _traverse_dict(self, instance_dict):
        output = {}
        for key, value in instance_dict.items():
            output[key] = self._traverse(key, value)
        return output

    def _traverse(self, key, value):
        if isinstance(value, ToDictMixin):
            return value.to_dict()
        elif isinstance(value, dict):
            return self._traverse_dict(value)
        elif isinstance(value, list):
            return [self._traverse(key, i) for i in value]
        elif hasattr(value, '__dict__'):
            return self._traverse_dict(value.__dict__)
        else:
            return value
```

```python
# Example 3
class BinaryTree(ToDictMixin):
    def __init__(self, value, left=None, right=None):
        self.value = value
        self.left = left
        self.right = right


# Example 4
tree = BinaryTree(10,
    left=BinaryTree(7, right=BinaryTree(9)),
    right=BinaryTree(13, left=BinaryTree(11)))
orig_print = print
print = pprint
print(tree.to_dict())
print = orig_print
```

```
{'left': {'left': None,
          'right': {'left': None, 'right': None, 'value': 9},
          'value': 7},
 'right': {'left': {'left': None, 'right': None, 'value': 11},
           'right': None,
           'value': 13},
 'value': 10}
```

BetterWay26. 믹스인 유틸리티 클래스에만 다중 상속을 사용하자.

- 믹스인은 범용 기능을 교체할 수 있게 만들어서 필요할 때 동작을 오버라이드 할 수 있음.

```
# Example 5
class BinaryTreeWithParent(BinaryTree):
    def __init__(self, value, left=None,
                 right=None, parent=None):
        super().__init__(value, left=left, right=right)
        self.parent = parent
```

기본 구현이 무한루프에 빠지게 됨.

```
# Example 6
    def _traverse(self, key, value):
        if (isinstance(value, BinaryTreeWithParent) and
                key == 'parent'):
            return value.value  # Prevent cycles
        else:
            return super()._traverse(key, value)
```

메서드를 오버라이드해서
부모를 탐색하지 않고 값만
반환하도록 만듦

```
# Example 7
root = BinaryTreeWithParent(10)
root.left = BinaryTreeWithParent(7, parent=root)
root.left.right = BinaryTreeWithParent(9, parent=root.left)
orig_print = print
print = pprint
print(root.to_dict())
print = orig_print
```

```
{'left': {'left': None,
          'parent': 10,
          'right': {'left': None, 'parent': 7, 'right': None, 'value': 9},
          'value': 7},
 'parent': None,
 'right': None,
 'value': 10}
```

BetterWay26. 믹스인 유틸리티 클래스에만 다중 상속을 사용하자.

- 확장
  - binaryTreeWithParent._traverse 정의해서 무한루프 돌지 않음.

```
# Example 8
class NamedSubTree(ToDictMixin):
    def __init__(self, name, tree_with_parent):
        self.name = name
        self.tree_with_parent = tree_with_parent

my_tree = NamedSubTree('foobar', root.left.right)
orig_print = print
print = pprint
print(my_tree.to_dict())  # No infinite loop
print = orig_print
```

```
{'name': 'foobar',
 'tree_with_parent': {'left': None, 'parent': 7, 'right': None, 'value': 9}}
```