파이썬 프로그래밍

약한 참조, 반복자, 발생자

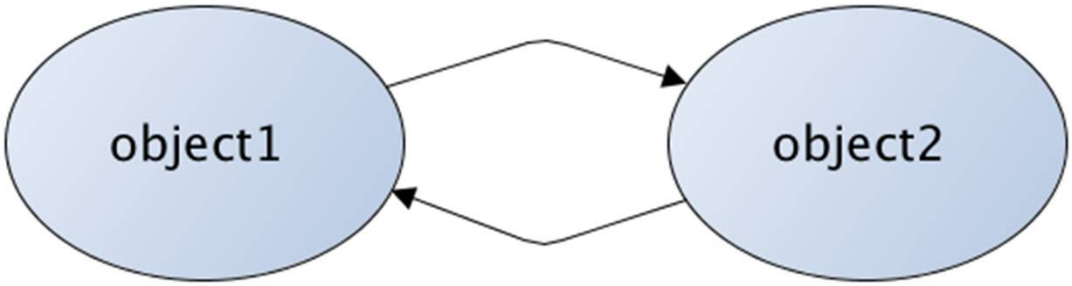
한국기술교육대학교
온라인평생교육원

▣ 약한 참조

1. 약한 참조의 정의

• 약한 참조 (Weak Reference)
- 레퍼런스 카운트로 고려되지 않는 참조

2. 약한 참조의 필요성



1) 레퍼런스 카운트가 증가되지 않으므로 순환 참조가 방지된다.
• 순환 참조 (Cyclic Reference)
- 서로 다른 객체들 사이에 참조 방식이 순환 형태로 연결되는 방식
- 독립적으로 존재하지만 순환 참조되는 서로 다른 객체 그룹은 쓰레기 수집이 안된다.
* 주기적으로 순환 참조를 조사하여 쓰레기 수집하는 기능이 있지만, CPU 자원 낭비가 심하다.
* 이러한 쓰레기 수집 빈도가 낮으면 순환 참조되는 많은 객체들이 메모리를 쓸데없이 점유하게 됨
2) 다양한 인스턴스들 사이에서 공유되는 객체에 대한 일종의 케시(Cache)를 만드는 데 활용된다.

▪객체는 식별자를 통해 객체를 reference하면 reference 카운트가 존재
▪레퍼런스 카운터 : 객체를 만들면 항상 증가
▪약한 참조 사용 시 객체의 레퍼런스 카운트를 증가시키지 않음
▪obj1, obj2 → 다른 객체들이 참조하지 않으므로 쓰레기
▪obj1, obj2 →즉, 사용되지 않을 객체이기 때문에 삭제 필요
▪하지만, 값을 가지기 때문에 쓰레기 수집기에서 수집 X

▣ 약한 참조

3. 약한 참조 모듈

1) weakref.ref(o)

- weakref 모듈의 ref(o) 함수
  - 객체 o에 대한 약한 참조를 생성한다.
  - 해당 객체가 메모리에 정상적으로 남아 있는지 조사한다.
    * 객체가 메모리에 남아 있지 않으면 None을 반환한다.
- 약한 참조로 부터 실제 객체를 참조하는 방법
  - 약한 참조 객체에 함수형태 호출

```
import sys
import weakref # weakref 모듈 임포트
class C:
    pass
c = C() # 클래스 C의 인스턴스 생성
c.a = 1 # 인스턴스 c에 테스트용 값 설정
print "refcount -", sys.getrefcount(c) # 객체 c의 레퍼런스 카운트 조회
print

d = c # 일반적인 레퍼런스 카운트 증가 방법
print "refcount -", sys.getrefcount(c) # 객체 c의 레퍼런스 카운트 조회
print

r = weakref.ref(c) # 약한 참조 객체 r 생성
print "refcount -", sys.getrefcount(c) # 객체 c의 레퍼런스 카운트 조회 --> 카운트 불변
print
```

```
refcount - 2

refcount - 3

refcount - 3
```

▣ 약한 참조

3. 약한 참조 모듈

▪getrefcount() : 객체의 레퍼런스 카운트가 몇인지 알아보는 메소드
▪weakref →약한 참조를 만들 수 있는 모듈
▪c 라는 식별자가 실제 객체를 가리키는 레퍼런스의 카운트가 1
▪c = C( ) →c가 레퍼런스
▪파이썬 내부에서 눈에 보이지 않는 레퍼런스 존재 →2
▪레퍼런스 카운트가 1이 되면 c 객체는 사라짐
▪c의 레퍼런스 값을 d에 카피해줌 →d의 레퍼런스 카운트 증가
▪weakref.ref() →레퍼런스 카운트 증가 X

▣ 약한 참조

3. 약한 참조 모듈

```
print r # 약한 참조(weakref) 객체
print r() # 약한 참조로 부터 실제 객체를 참조하는 방법: 약한 참조 객체에 함수형태로
          호출
print c
print r().a # 약한 참조를 이용한 실제 객체 멤버 참조
print


del c # 객체 제거
del d
print r() # None을 리턴한다
print r().a # 속성도 참조할 수 없다
```

```
<weakref at 0x10d83e998; to 'instance' at 0x10d893830>
<__main__.C instance at 0x10d893830>
<__main__.C instance at 0x10d893830>
1

None
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-17-0a1d693859de> in <module>()
      8 del d
      9 print r() # None을 리턴한다
---> 10 print r().a # 속성도 참조할 수 없다

AttributeError: 'NoneType' object has no attribute 'a'
```

▪만든 약한 참조의 활용

▪r( ) →실제 약한 참조가 참조하고 있는 객체를 반환

▪r( ) = c

▪c, d →일반적인 레퍼런스 변수로 삭제하면 레퍼런스 카운트 감소

▪r( ) →실제 약한 참조가 참조하고 있는 객체를 반환

▣ 약한 참조

3. 약한 참조 모듈

- 내장 자료형 객체 (리스트, 튜플, 사전 등)에 대해서는 약한 참조를 만들 수 없다.

```
d = {'one': 1, 'two': 2}
wd = weakref.ref(d)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-7-b2a48d12fd2b> in <module>()
      1 d = {'one': 1, 'two': 2}
----> 2 wd = weakref.ref(d)

TypeError: cannot create weak reference to 'dict' object
```

▪약한 참조는 기본적으로 제공하는 내장자료형 객체에는 사용 X

▪사전 객체 →내장자료형 객체

▪사용자가 정의한 클래스의 인스턴스에만 약한 참조 사용 가능

▣ 약한 참조

3. 약한 참조 모듈

2) weakref.proxy(o)

- weakref의 proxy(o)는 객체 o에 대한 약한 참조 프록시를 생성한다.
  - 프록시를 이용하면 함수 형식을 사용하지 않아도 실제 객체를 바로 참조할 수 있다.
  - ref(o) 함수보다 더 선호되는 함수

```
import sys
import weakref
class C:
    pass

c = C()
c.a = 2
print "refcount -", sys.getrefcount(c) # 객체 c의 레퍼런스 카운트 조회
p = weakref.proxy(c) # 프록시 객체를 만든다
print "refcount -", sys.getrefcount(c) # 객체 c의 레퍼런스 카운트 조회 --> 카운트 불변
print
print p
print c
print p.a
```

```
refcount - 2
refcount - 2

<__main__.C instance at 0x10d8a9998>
<__main__.C instance at 0x10d8a9998>
2
```

▣ 약한 참조

3. 약한 참조 모듈

▪weakref.proxy( ) 가 좀 더 활용도 높음
▪p = weakref.proxy( ) →p는 자연스러운 레퍼런스 만들어짐
▪print p →실제 p가 가리키는 실제 객체 호출
▪weakref.proxy() →레퍼런스 카운트 증가 X
▪객체 C, p는 동일한 객체지만 p는 weakref
▪proxy를 쓰면 코딩량을 줄일 수 있음
▪ref()함수보다는 proxy()가 좀 더 편하게 p 이용가능

▣ 약한 참조

3. 약한 참조 모듈

```
import weakref
class C:
    pass

c = C() # 참조할 객체 생성
r = weakref.ref(c) # weakref 생성
p = weakref.proxy(c) # weakref 프록시 생성
print weakref.getweakrefcount(c) # weakref 개수 조회
print weakref.getweakrefs(c) # weakref 목록 조회
```

```
2
[<weakref at 0x10de07c58; to 'instance' at 0x10de06e60>,
 <weakproxy at 0x10de07ba8 to instance at 0x10de06e60>]
```

▪getweakrefcount() →약한 참조가 몇 개인지 알아보는 메소드

▪getweakrefs() →반환되는 리스트 안 원소가 약한 참조 c

▣ 약한 참조

4. 약한 사전

- 약한 사전 (Weak Dictionary)
  - 사전의 키(key)나 값(value)으로 다른 객체들에 대한 약한 참조를 지니는 사전
  - 주로 다른 객체들에 대한 캐시(Cache)로 활용
  - 일반적인 사전과의 차이점
    * 키(key)나 값(value)으로 사용되는 객체는 약한 참조를 지닌다.
    * 실제 객체가 삭제되면 자동적으로 약한 사전에 있는 (키, 값)의 쌍도 삭제된다.
    * 즉, 실제 객체가 사라지면 캐시역할을 하는 약한 사전에서도 해당 아이템이 제거되므로 효율적인 객체 소멸 관리가 가능하다.

▪효율적인 메모리 관리 가능

▣ 약한 참조

4. 약한 사전

- weakref 모듈의 WeakValueDictionary 클래스
  - weakref 모듈의 WeakValueDictionary 클래스의 생성자는 약한 사전을 생성한다.

```
import weakref
class C:
    pass

c = C()
c.a = 4
d = weakref.WeakValueDictionary() # WeakValueDictionary 객체 생성
print d

d[1] = c # 실제 객체에 대한 약한 참조 아이템 생성
print d.items() # 사전 내용 확인
print d[1].a # 실제 객체의 속성 참조

del c # 실제 객체 삭제
print d.items() # 약한 사전에 해당 객체 아이템도 제거되어 있음
```

```
<WeakValueDictionary at 4526484584>
[(1, <__main__.C instance at 0x10dccad40>)]
4
[]
```

▪WeakValueDictionary() →클래스로 클래스의 생성자를 호출하는 것
▪WeakValueDictionary라는 새로운 자료형의 객체를 d에 할당
▪사전이지만 weak(약한) 사전
▪1이라는 key값의 value로 c를 넣을 수 있음
▪d[1] →사전의 검색 = c
▪del c →레퍼런스 카운트가 1로 줄음

▣ 약한 참조

4. 약한 사전

- 일반 사전을 통하여 동일한 예제를 수행하면 마지막에 해당 객체를 삭제해도 일반 사전 내에서는 여전히 존재함

```
class C:
    pass

c = C()
c.a = 4
d = {} # 일반 사전 객체 생성
print d

d[1] = c # 실제 객체에 대한 일반 참조 아이템 생성
print d.items() # 사전 내용 확인
print d[1].a # 실제 객체의 속성 참조

del c # 객체 삭제 (사전에 해당 객체의 레퍼런스가 있으므로 객체는 실제로 메모리
        해제되지 않음)
print d.items() # 일반 사전에 해당 객체 아이템이 여전히 남아 있음
```

```
{}
[(1, <__main__.C instance at 0x10d893878>)]
4
[(1, <__main__.C instance at 0x10d893878>)]
```

▪일반적인 사전은 del c를 해도 삭제가 안됨

▪이유 : 사전 자체가 객체를 레퍼런스로 가지고 있음

▪약한 사전은 레퍼런스를 약한 참조로 가지고 있어서 삭제 가능