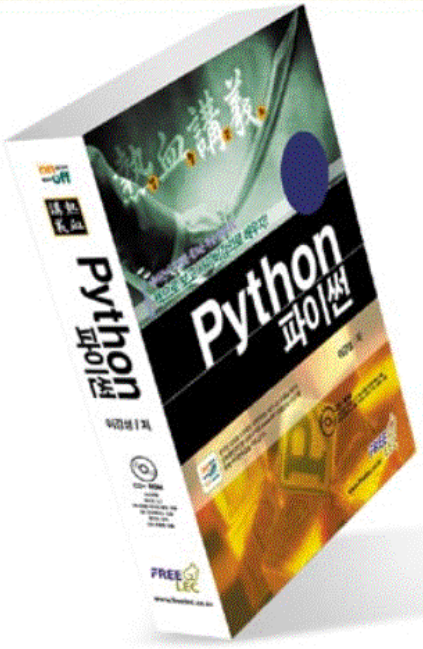
熱血講義

프리렉의 열혈강의 시리즈

Python 파이썬



열혈강의 Python 파이썬

❖제 14 강

주제 : 약한 참조, 반복자, 발생자

이강성 (gslee@mail.gwu.ac.kr)

❖강의 목차

1. 약한 참조(weak reference)
2. 반복자(iterator)
3. 발생자(generator)

14-1 약한 참조

- 약한 참조(weak reference)란?
  - 레퍼런스 카운트로 고려되지 않는 레퍼런스
  - 순환 참조로 인한 쓰레기 수집 기능을 방해받지 않기 위해서
- weakref 모듈 지원(2.1)
  - weakref.ref() – 약한 참조 객체 생성
  - weakref.proxy() – 프록시 객체 생성
  - weakref.WeakValueDictionary()
  - weakref.WeakKeyDictionary()

14-1 약한 참조

- **weakref.ref()**

```
>>> import weakref
>>> class C:
        a = 1


>>> c = C()
>>> r = weakref.ref(c)
>>> r().a
1
>>> del c
>>> r()     # None
>>> r().a   # AttributeError 예외 발생
```


14-1 약한 참조

- **weakref.proxy()**

```
>>> import weakref
>>> class C:
        a = 1


>>> c = C()
>>> p = weakref.proxy(c) # 괄호 없이 사용
>>> p.a
1
```

14-1 약한 참조

- getweakrefcount(obj)
  - 약한 참조의 개수 알아보기
- getweakrefs(obj)
  - 약한 참조 객체 얻기

```
>>> c = C()              # 참조할 객체를 만든다
>>> r = weakref.ref(c)# weakref
>>> p = weakref.proxy(c)      # proxy weakref
>>> weakref.getweakrefcount(c) # weakref 개수
2
>>> weakref.getweakrefs(c)    # weakref 목록
[<weakref at 0xb0aa90; to 'instance' at
0xf401f0>, <weakref at 00F43290 to instance at
00F401F0>]
```


14-1 약한 참조

- 약한 사전(weak dictionary)
  - 캐쉬(cache)라고도 한다
  - 키(key) 혹은 값(value)으로 사용되는 객체는 약한 참조가 가능해야 한다.
  - 사전의 키(key) 혹은 값(value)으로 약한 참조를 갖는다.
  - 객체가 삭제되면 자동적으로 캐쉬에 있는 키, 값 쌍도 삭제된다.
  - WeakValueDictionary – 값으로 약한 참조를 갖는다
  - WeakKeyDictionary – 키로 약한 참조를 갖는다

14-1 약한 참조

- WeakValueDictionary

```
>>> class C:
        pass
>>> c = C()
>>> c.a = 4
>>> d = weakref.WeakValueDictionary()
>>> d[1] = c   # 항목 생성
>>> d.items()  # 사전 내용 확인
[(1, <__main__.C instance at 0x00F54A90>)]
>>> d[1].a     # 참조
4
>>> del c      # 객체 삭제
>>> d.items()  # 항목이 사라졌다
[]
```


14-1 약한 참조

- WeakKeyDictionary

```
>>> d = weakref.WeakKeyDictionary()
>>> c = C()
>>> c.a = 4
>>> d[c] = 1
>>> d.items()
[(<__main__.C instance at
0x00F901F0>, 1)]
>>> del c
>>> d.items()
[]
```

14-2 반복자(iterator) – 2.2

- 기존 인덱싱의 문제점
  - __getitem__ 메써드를 통해서 이루어지는 인덱싱은 인덱싱이 중요하지 않은 순차 처리에도 사용되어 왔다
  - 예:

14-2 반복자(iterator)

```
>>> class Seq:
        def __init__(self, fname):  # 읽을 파일 이름 설정
            self.file = open(fname)
        def __getitem__(self, k):
            line = self.file.readline()
            if not line: raise IndexError
            return line   # 읽은 라인을 리턴한다.

>>> S = Seq('readme.txt')
>>> for line in S:   # 각 라인에 대해서
        print line,

>>> S[3] # 3은 의미 없다
```

14-2 반복자(iterator)

- 반복자 객체
  - iter() 내장 함수로 만들어진다.
  - next() 메써드를 갖는다
  - next() 메써드가 더 이상 자료를 넘겨줄 수 없는 경우에 StopIteration 예외를 발생한다
- for 문은 반복자 객체에 의해 수행된다(2.2)
  - for x in obj: f(x)
- 예:

14-2 반복자(iterator)

```
>>> I = iter([1,2])  # 리스트에 대한 반복자 객체를 만든다.
>>> I
<iterator object at 0x009F9A60>
>>> I.next()   # 순차적으로 자료를 얻는다.
1
>>> I.next()
2
>>> I.next()  # 더이상 자료가 없으면 StopIteration 예외 발생
Traceback (most recent call last):
  File "<pyshell#71>", line 1, in ?
    I.next()
StopIteration
```

14-2 반복자(iterator)

- 클래스에 반복자 구현하기
  - __iter__ 을 구현하고 next() 메쏘드가 있는 객체를 리턴한다
  - 클래스 자체를 반복자로 만들려면 next 메쏘드를 구현한다
- 예:

14-2 반복자(iterator)

```
>>> class Seq:
        def __init__(self, fname):
            self.file = open(fname)
        def __iter__(self):
            return self
        def next(self):
            line = self.file.readline()
            if not line: raise StopIteration
            return line


>>> S = Seq('readme.txt')
>>> for line in S:
        print line,
```

14-2 반복자(iterator)

- 사전의 반복자
  - 사전은 그 자체가 키에 대한 반복자이다
  - iterkeys, itervalues, iteritems 메써드는 키, 값 혹은 (키, 값) 튜플의 반복자를 리턴한다.

```
>>> d = {'one':1, 'two':2, 'three':3, 'four':4, 'five':5}
>>> for key in d: # 또는 d.iterkeys()
        print key, d[key]
```


14-2 반복자(iterator)

- 파일 객체의 반복자
  - 파일 객체는 라인 단위의 반복자를 지원

```
>>> f = open('readme.txt')
>>> for line in f:
        print line,
```

14-2 반복자(iterator)

- 호출 가능한 객체에 반복자 적용하기
  - for val in iter(callable, endval): ...
  - callable이 endval 값을 리턴할 때까지 ...을 반복

```
/* C */
while ( (line = readline()) != NULL)
        printf(line);


# Python
for line in iter(raw_input, ''):
        print line
```


14-3 발생자(generator) – 2.2

- 기존의 함수 호출 방식
  - 함수가 호출 될 때 인수들과 내부 변수들이 새로운 영역(대부분은 스택)에 만들어지고 리턴할 때 메모리에서 사라진다
- 발생자란?
  - 함수가 호출된 후에 되돌아 갈 때, 메모리가 해제되지 않고 그대로 남아있다
  - 다시 그 함수가 호출될 때 이 전에 수행이 종료되었던 지점 이후를 계속 진행

14-3 발생자(generator)

- 발생자의 예 1

```
>>> from __future__ import generators
>>> def generate_ints(N):
    for i in range(N):
        yield i

>>> gen = generate_ints(10)
>>> gen
<generator object at 0x01370B30>
```

```
>>> gen.next()
0
>>> gen.next()
1
>>> for k in gen:
        print k,

2 3 4 5 6 7 8 9
```


14-3 발생자(generator)

- 발생자의 예 2 – 피보나치 수열

```
>>> def fibonacci(a=1, b=1):
    while 1:
        yield a
        a, b = b, a+b

>>> for k in fibonacci():
        if k > 100: break
        print k,

1 1 2 3 5 8 13 21 34 55 89
```

14-3 발생자(generator)

- 발생자의 예 3 – 홀수 집합 만들기

```
>>> def  odds(limit=None):
        k = 1
        while not limit or limit >= k:
            yield k
            k += 2


>>> for k in odds(20):
        print k,

1 3 5 7 9 11 13 15 17 19
>>>  list(odds(20))
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
```


14-3 발생자(generator)

- 발생자의 예 4 – 트리 탐색

```
from __future__ import generators
class Tree(object):
    def __init__(self, data=None, left=None, right=None):
        self.data = data          # 노드의 값
        self.left = left          # 왼쪽 노드
        self.right = right        # 오른쪽 노드

    def inorder(self):            # inorder 탐색
        if self.left:
            for x in self.left.inorder():
                yield x
        yield self
        if self.right:
            for x in self.right.inorder():
                yield x
```

14-3 발생자(generator)

```
def tree(list):
    n = len(list)
    if n == 0:
       return None
    i = n / 2
    return Tree(list[i], tree(list[:i]), tree(list[i+1:]))

if __name__ == '__main__':
    t = tree('abcdef')
    for el in t.inorder():
        print el.data,
```

```
a b c d e f
```