파이썬 프로그래밍

약한 참조, 반복자, 발생자

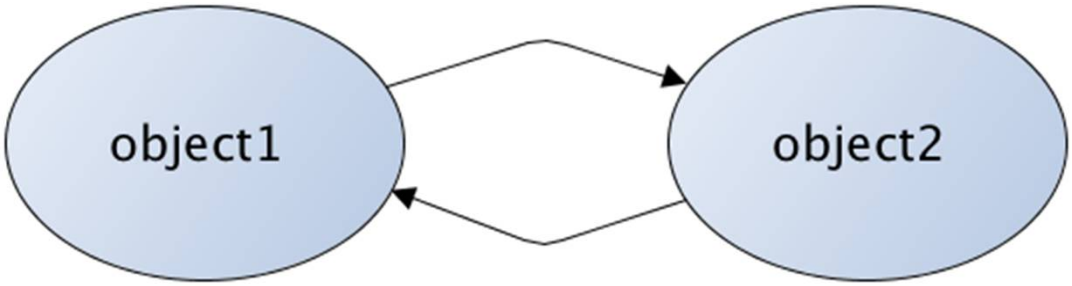
한국기술교육대학교
온라인평생교육원

▣ 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

파이썬 프로그래밍

약한 참조, 반복자, 발생자

한국기술교육대학교
온라인평생교육원

▣ 반복자

1. 반복자 객체

- 반복자 객체
  - next() 메소드를 지니고 있는 객체
  - next() 메소드로 더 이상 자료를 넘겨줄 수 없을 때 StopIteration 예외가 발생한다.
- 반복자 객체 생성 방법
  - iter(o) 내장 함수
  - 객체 o의 반복자 객체를 반환한다.
- 반복자 객체의 효율성
  - 반복자가 원 객체의 원소들을 복사하여 지니고 있지 않다.

```
I = iter([1,2,3])
print I

print I.next()
print I.next()
print I.next()
print I.next()
```

```
<listiterator object at 0x102648cd0>
1
2
3
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-20-98af67aeb8bc> in <module>()
      5 print I.next()
      6 print I.next()
----> 7 print I.next()

StopIteration:
```

▪iter()함수에서 반환되어지는 I가 반복자 객체

▪첫번째 I next() →1, 두 번째 →2, 세 번째 →3

▪마지막 I.next()는 가져올 수 있는 원소가 없어서 StopIteration 호출

▪I는 각 리스트의 원소를 뽑아낼 수 있는 기능만 있고 카피한 것은 X

▣ 반복자

1. 반복자 객체

- 리스트 객체에 대해 일반적인 for ~ in 반복 문 사용예

```
def f(x):
    print x + 1

for x in [1,2,3]:
    f(x)
```

```
2
3
4
```

- 리스트 객체에 반복자를 활용한 예

```
def f(x):
    print x + 1

t = iter([1,2,3])
while 1:
    try:
        x = t.next()
    except StopIteration:
        break
    f(x)
```

```
2
3
4
```

▪반복자 객체는 next()라는 메소드 가짐

▣ 반복자

1. 반복자 객체

- for ~ in 구문에 반복자를 활용할 수 있다.
  - for 문이 돌때 마다 반복자 객체의 next() 함수가 자동으로 호출되어 순차적으로 각 객체에 접근 가능하다.
  - StopIteration이 발생하면 for ~ in 구문이 멈춘다.

```
def f(x):
    print x + 1

t = iter([1,2,3])
for x in t:
    f(x)
```

```
2
3
4
```

▪반복자 객체를 for~in 구문의 in 뒤에 바로 사용 가능

▪t가 가지고 있는 next() 메소드도 자동으로 호출

▣ 반복자

1. 반복자 객체

```
def f(x):
    print x + 1

for x in iter([1,2,3]):
    f(x)
```

```
2
3
4
```

```
def f(x):
     print x + 1

for x in iter((1,2,3)):
     f(x)
```

```
2
3
4
```

▪반복자 객체를 정의하지 않고 바로 iter를 이용하여 직접 사용도 가능

▪반복자를 배우는 이유 : 발생자를 배우기 위한 것

▪iter의 객체로 리스트, 튜플 가능

▣ 반복자

2. 클래스에 반복자 구현하기

- 내장 함수 iter(o)에 대응되는 __iter__(self)의 구현
  - 객체 o에 iter(o)를 호출하면 자동으로 __iter__(self) 함수 호출
  - __iter__(self) 함수는 next() 함수를 지닌 반복자 객체를 반환해야 한다.

▣ 반복자

2. 클래스에 반복자 구현하기

```
class Seq:
    def __init__(self, fname):
        self.file = open(fname)
    #def __getitem__(self, n):
    #    if n == 10:
    #        raise StopIteration
    #    return n
    def __iter__(self):
        return self
    def next(self):
        line = self.file.readline() # 한 라인을 읽는다.
        if not line:
            raise StopIteration     # 읽을 수 없으면 예외 발생
        return line                 # 읽은 라인을 리턴한다.

s = Seq('readme.txt')     # s 인스턴스가 next() 메소드를 지니고 있으므로 s 인스턴스
                            자체가 반복자임
for line in s: # 우선 __iter__() 메소드를 호출하여 반복자를 얻고, 반복자에 대해서
                for ~ in 구문에 의하여 next() 메소드가 호출됨
    print line,

print

print Seq('readme.txt')

print list(Seq('readme.txt'))  # list() 내장 함수가 객체를 인수로 받으면 해당 객체의
                                반복자를 얻어와 next()를 매번 호출하여 각 원소를 얻어온다.
print tuple(Seq('readme.txt')) # tuple() 내장 함수가 객체를 인수로 받으면 해당 객체의
                                반복자를 얻어와 next()를 매번 호출하여 각 원소를 얻어온다.
```

▣ 반복자

2. 클래스에 반복자 구현하기

```
abc
def
ghi

<__main__.Seq instance at 0x10ddc5680>
['abc \ n', 'def \ n', 'ghi \ n']
('abc \ n', 'def \ n', 'ghi \ n')
```

- __iter__ →iter 내장함수와 매칭됨
- iter내장함수는 반복자 객체(자기 자신) 를 리턴
- self가 next()를 가지고 있음
- s 객체는 사용자가 생성한 클래스의 객체
- s에 __iter__이 존재해야 함 →없으면 error 발생
- for~in 구문에 s가 들어가면 iter 내장함수 호출됨
- self는 next() 메소드 호출
- for~in 구문에 iter 함수를 적지 않고 바로 리스트 적어도 무관
- 내장함수임과 동시에 리스트 클래스의 생성자
- 일단 iter를 불러준 뒤 반복자를 가져옴 →next 메소드 호출

▣ 반복자

3. 사전의 반복자

- 사전에 대해 for ~ in 구문은 키에 대해 반복한다.

```
d = {'one':1, 'two':2, 'three':3, 'four':4, 'five':5}
for key in d:
    print key, d[key]
```

```
four 4
three 3
five 5
two 2
one 1
```

▪d에 대해서 먼저 iter 함수가 불림

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

```
four 4
three 3
five 5
two 2
one 1
```

▪반복자는 next를 가짐

▪next() →각각의 아이템의 key 값만 돌려줌

▪for~in에 쓰여지는 객체는 반드시 반복자 객체를 가짐

▣ 반복자

3. 사전의 반복자

• d.iterkeys() 함수
- 사전 d가 지닌 키에 대한 반복자 객체를 반환한다.

```
for key in d.iterkeys():   # 키에 대한 반복자, d.iterkeys() 가 반환한 반복자에 대해
                             next() 함수가 순차적으로 불리워짐
    print key,
```

```
four three five two one
```

▪iterkeys() →key 값만 가져옴
▪itervalues() →value 값만 가져옴

```
keyset = d.iterkeys()
print keyset.next()     # 반복자 객체는 항상 next() 메소드를 지니고 있음
for key in keyset:      # keyset 반복자에 대해 next() 메소드가 순차적으로 호출됨
    print key,
```

```
four
three five two one
```

▪iterkeys() →key 값만 가져와서 반복하는 반복자
▪next가 한 번 호출되면 반복자는 그 다음 next는 다음 원소 활용

▣ 반복자

3. 사전의 반복자

• d.itervalues() 함수
- 사전 d가 지닌 값에 대한 반복자 객체를 반환한다.

```
for value in d.itervalues():    # 값에 대한 반복자
    print value,
```

```
4 3 5 2 1
```

```
for key, value in d.iteritems(): #(키,값)에 대한 반복자
    print key, value
```

```
four 4
three 3
five 5
two 2
one 1
```

▣ 반복자

4. 파일 객체의 반복자

• 파일 객체는 그 자체가 반복자임
- next() 함수에 의해 각 라인이 순차적으로 읽혀짐

```
f = open('readme.txt')
print "f.next()", f.next()
for line in f:   # f.next() 가 순차적으로 호출됨
    print line,
```

```
f.next() 1: Hello World

2: Hello World
3: Hello World
4: Hello World
5: Hello World
```

▪f 파일 객체 자체가 반복자 →next를 가지고 있기 때문
▪콤마(,)가 있어도 옆에 같이 출력 안되는 이유는 개행 문자 때문

파이썬 프로그래밍

약한 참조, 반복자, 발생자

한국기술교육대학교
온라인평생교육원

▣ 발생자

1. 발생자란?

- 발생자(Generator)
  - (중단된 시점부터) 재실행 가능한 함수
- 아래 함수 f()는 자신의 인수 및 내부 변수로서 a, b, c, d를 지니고 있다.
  - 이러한 a, b, c, d 변수들은 함수가 종료되고 반환될 때 모두 사라진다.
- 발생자는 f()와 같이 함수가 종료될 때 메모리에서 해제되는 것을 막고 다시 함수가 호출 될 때 이전에 수행이 종료되었던 지점 부터 계속 수행이 가능하도록 구현된 함수이다.

```
def f(a,b):
    c = a * b
    d = a + b
    return c, d
```

▪발생자는 함수

▪f 함수는 리턴이 되면 a,b,c,d 모두 사라짐

▣ 발생자

1. 발생자란?

- yield 키워드
  - return 대신에 yield에 의해 값을 반환하는 함수는 발생자이다.
  - yield는 return과 유사하게 임의의 값을 반환하지만 함수의 실행 상태를 보존하면서 함수를 호출한 쪽으로 복귀시켜준다.
- 발생자는 곧 반복자이다.
  - 즉, 발생자에게 next() 호출이 가능하다.

```
def generate_ints(N):
    for i in range(N):
        yield i
```

▪발생자를 만들려면 return 대신 yield 키워스 사용

▣ 발생자

1. 발생자란?

```
gen = generate_ints(3) # 발생자 객체를 얻는다. generate_ints() 함수에 대한 초기
                         스택 프레임이 만들어지나 실행은 중단되어 있는 상태임
print gen
print gen.next() # 발생자 객체는 반복자 인터페이스를 가진다. 발생자의 실행이 재개됨.
                   yield에 의해 값 반환 후 다시 실행이 중단됨
print gen.next() # 발생자 실행 재개. yield에 의해 값 반환 후 다시 중단
print gen.next() # 발생자 실행 재개. yield에 의해 값 반환 후 다시 중단
print gen.next() # 발생자 실행 재개. yield에 의해 더 이상 반환할 값이 없다면
                   StopIteration 예외를 던짐
```

```
<generator object generate_ints at 0x10ddd6410>
0
1
2
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-8-65149ac25109> in <module>()
      4 print gen.next() # 발생자 실행 재개. yield에 의해 값 반환 후 다시 중단
      5 print gen.next() # 발생자 실행 재개. yield에 의해 값 반환 후 다시 중단
----> 6 print gen.next() # 발생자 실행 재개. yield에 의해 더 이상 반환할 값이 없다면
                          StopIteration 예외를 던짐

StopIteration:
```

- __generate init__ →발생자 함수
- gen →yield가 반환하는 I 값을 가짐
- 기본적으로는 gen에는 __generate init__ 자체의 객체가 들어감
- 발생자가 곧 반복자로 발생자 gen에게 next 호출 가능
- 마지막 next는 돌려줄 값이 없으므로 StopIteration 발생

▣ 발생자

1. 발생자란?

- 위와 같은 세부 동작 방식을 이용하여, 다음과 같이 for ~ in 구문에 적용할 수 있다.

```
for i in generate_ints(5):
    print i,
```

```
0 1 2 3 4
```

▪for~in 구문에 발생자 바로 사용 가능
▪발생자는 곧 반복자 →next를 가지고 있음
▪예외가 발생하면 for~in 구문은 빠져나가게 되어 있음

- 발생자 함수와 일반 함수의 차이점
  - 일반 함수는 함수가 호출되면 그 함수 내부에 정의된 모든 일을 마치고 결과를 반환함
  - 발생자 함수는 함수 내에서 수행 중에 중간 결과 값을 반환할 수 있음
- 발생자가 유용하게 사용되는 경우
  - 함수 처리의 중간 결과를 다른 코드에서 참조할 경우
  - 모든 결과를 한꺼번에 처리하는 것이 아니라 함수 처리 중에 나온 중간 결과를 사용해야 할 경우

▣ 발생자

2. 발생자 구문

- 리스트 내포(List Comprehension)
  - 리스트 객체의 새로운 생성
  - 메모리를 실제로 점유하면서 생성됨

```
print [k for k in range(100) if k % 5 == 0]
```

```
[0, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75, 80, 85, 90, 95]
```

▪k가 리스트의 원소

▣ 발생자

2. 발생자 구문

- 리스트 내포 구문에 []가 아니라 () 사용
  - 리스트 대신에 발생자 생성
  - 처음부터 모든 원소가 생성되지 않고 필요한 시점에 각 원소가 만들어짐
  - 메모리를 보다 효율적으로 사용함

```
a = (k for k in range(100) if k % 5 == 0)
print a
print a.next()
print a.next()
print a.next()
for i in a:
    print i,
```

```
<generator object <genexpr> at 0x10d84df50>
0
5
10
15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90 95
```

▪리스트 내포와 문법이 똑같은데, []가 아닌 () 사용 →발생자

▪리스트 내포 : 리스트 전반적인 내용 모두 생성

▪발생자 : next가 호출될 때마다 리스트 내용 발생

▣ 발생자

2. 발생자 구문

- 아래 예는 sum 내장 함수에 발생자를 넣어줌
  - sum을 호출하는 시점에는 발생자가 아직 호출되기 직전이므로 각 원소들은 아직 존재하지 않는다.
  - sum 내부에서 발생자가 지니고 있는 next() 함수를 호출하여 각 원소들을 직접 만들어 활용한다.
  - 메모시 사용 효율이 높다.

```
print sum((k for k in range(100) if k % 5 == 0))
```

```
950
```

▪sum을 호출하는데 안쪽 인자가 발생자 k를 generate

▣ 발생자

3. 발생자의 활용 예 1 - 피보나치 수열

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

for k in fibonacci():  # 발생자를 직접 for ~ in 구문에 활용
    if k > 100:
        break
    print k,
```

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

▪yield가 있으니까 fibonaci는 발생자

▣ 발생자

4. 발생자의 활용 예 2 - 홀수 집합 만들기

• 반복자를 활용한 예

```
class Odds:
    def __init__(self, limit = None): # 생성자 정의
        self.data = -1                # 초기 값
        self.limit = limit            # 한계 값
    def __iter__(self):               # Odds 객체의 반복자를 반환하는 특수 함수
        return self
    def next(self):                   # 반복자의 필수 함수
        self.data += 2
        if self.limit and self.limit <= self.data:
            raise StopIteration
        return self.data

for k in Odds(20):
    print k,
print
print list(Odds(20)) # list() 내장 함수가 객체를 인수로 받으면 해당 객체의 반복자를
                     얻어와 next()를 매번 호출하여 각 원소를 얻어온다.
```

```
1 3 5 7 9 11 13 15 17 19
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
```

▪Odds(20) →Odds라는 클래스에는 인스턴스 메소드로 if를 가짐

▪반복자를 next 를 가지고 있는 self로 가져옴

▣ 발생자

4. 발생자의 활용 예 2 - 홀수 집합 만들기

- 발생자를 활용한 예

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

for k in odds(20):
    print k,
print
print list(odds(20)) # list() 내장 함수가 발생자를 인수로 받으면 해당 발생자의 next()를
                     매번 호출하여 각 원소를 얻어온다.
```

```
1 3 5 7 9 11 13 15 17 19
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
```

▪for~in 구문에 odds(20)사용 가능 →odds라는 반복자는 곧 발생자

▪list도 역시 발생자를 받으면 반복자로 사용