generator expression

generator expression

An expression that returns an iterator. It looks like a normal expression followed by a for expression defining a loop variable, range, and an optional if expression. The combined expression generates values for an enclosing function:

위와 같은 generator 함수를 좀 더 쉽게 사용할 수 있도록 generator expression 을 제공한다.
list comprehension 과 비슷하지만, [ ] 대신 ( ) 를 사용하면 된다.

아래는 0 ~ 9 사이 정수 중 홀수 를 list 와 generator object 로 생성한 예제이다.

```
>>> [ i for i in xrange(10) if i % 2 ]
[1, 3, 5, 7, 9]
>>> ( i for i in xrange(10) if i % 2 )
<generator object <genexpr> at 0x7f6105d90960>
```

( ) 를 사용하면 위와 같이 generator object 로 생성됨을 볼 수 있다.

generator 를 왜 사용하는가

그렇다면 generator 를 사용했을 때 어떤 점이 유용할 것인가.

먼저는, **memory를 효율적으로 사용**할 수 있다.
아래 예제를 통해 list comprehension 과 generator expression 으로 각각 생성했을 때 메모리 사용 상태를 보자.

```
>>> import sys
>>> sys.getsizeof( [i for i in xrange(100) if i % 2] )    # list
536
>>> sys.getsizeof( [i for i in xrange(1000) if i % 2] )
4280
>>> sys.getsizeof( (i for i in xrange(100) if i % 2) )    # generator
80
>>> sys.getsizeof( (i for i in xrange(1000) if i % 2) )
80
```

list 의 경우 사이즈가 커질 수록 그만큼 메모리 사용량이 늘어나게 된다. 하지만, generator 의 경우는 사이즈가 커진다 해도 차지하는 메모리 사이즈는 동일하다. 이는 list 와 generator의 동작 방식의 차이에 기인한다.

list 는 list 안에 속한 모든 데이터를 메모리에 적재하기 때문에 list의 크기 만큼 차지하는 메모리 사이즈가 늘어나게 된다. 하지만 generator 의 경우 데이터 값을 한꺼번에 메모리에 적재 하는 것이 아니라 next() 메소드를 통해 차례로 값에 접근할 때마다 메모리에 적재하는 방식이다.

따라서 list 의 규모가 큰 값을 다룰 수록 generator의 효율성은 더욱 높아지게 된다.

즉, 오소가 많은 array-like 자료형을 순회해야 할 때는 'list'보다 'generator'함수를 사용하는 게 좋다.

먼저 python docs 의 generator 에 대한 정의를 보자.

> generator
>
> A function which returns an iterator. It looks like a normal function except that it contains yield statements for producing a series of values usable in a for-loop or that can be retrieved one at a time with the next() function. Each yield temporarily suspends processing, remembering the location execution state (including local variables and pending try-statements). When the generator resumes, it picks-up where it left-off (in contrast to functions which start fresh on every invocation).

generator 는 간단하게 설명하면 iterator 를 생성해 주는 function 이다. iterator 는 next() 메소드를 이용해 데이터에 순차적으로 접근이 가능한 object 이다. iterator 에 대한 자세한 설명은 링크에서 확인. (iterable 과 iterator 의 의미)

generator 는 일반적인 함수와 비슷하게 보이지만, 가장 큰 차이 점은 **yield** 라는 구문일 것이다. 아래는 generator 함수의 예시 구문이다.

```
def generator(n):
    i = 0
    while i < n:
        yield i
        i += 1
```

일반 함수와의 차이는 yield 외에는 없다. 그렇다면 먼저 yield 구문이 무엇인지 먼저 알아보자.

yield

The yield statement is only used when defining a generator function, and is only used in the body of the generator function. Using a yield statement in a function definition is sufficient to cause that definition to create a generator function instead of a normal function.

When a generator function is called, it returns an iterator known as a generator iterator, or more commonly, a generator. The body of the generator function is executed by calling the generator's next() method repeatedly until it raises an exception.

When a yield statement is executed, the state of the generator is frozen and the value of expression_list is returned to next()'s caller. By "frozen" we mean that all local state is retained, including the current bindings of local variables, the instruction pointer, and the internal evaluation stack: enough information is saved so that the next time next() is invoked, the function can proceed exactly as if the yield statement were just another external call.

As of Python version 2.5, the yield statement is now allowed in the try clause of a try ... finally construct. If the generator is not resumed before it is finalized (by reaching a zero reference count or by being garbage collected), the generator-iterator's close() method will be called, allowing any pending finally clauses to execute.

yield 는 generator 가 일반 함수와 구분되는 가장 핵심적인 부분이다. yield 를 사용함으로서 어떤 차이가 있게 되는지 살펴보자.

먼저, 일반적인 함수의 경우를 생각해보자.
일반적인 함수는 사용이 종료되면 결과값을 호출부로 반환 후 함수 자체를 종료시킨 후 메모리 상에서 클리어 된다.

하지만, yield 를 사용할 경우는 다르다.

generator 함수가 실행 중 yield 를 만날 경우, 해당 함수는 그 상태로 정지 되며, 반환 값을 next() 를 호출한 쪽으로 전달 하게 된다. 이후 해당 함수는 일반적인 경우 처럼 종료되는 것이 아니라 그 상태로 유지되게 된다. 즉, 함수에서 사용된 local 변수나 instruction pointer 등과 같은 함수 내부에서 사용된 데이터들이 메모리에 그대로 유지되는 것이다.

아래 예시를 보자.

```
def generator(n):
    i = 0
    while i < n:
        yield i
        i += 1

for x in generator(5):
     print x

0
1
2
3
4
```

위 구문을 하나하나 살펴보자.

1. for 문이 실행되며, 먼저 generator 함수가 호출된다.
2. generator 함수는 일반 함수와 동일한 절차로 실행된다.
3. 실행 중 while 문 안에서 yield 를 만나게 된다. 그러면 return 과 비슷하게 함수를 호출했던 구문으로 반환하게 된다. 여기서는 첫번재 i 값인 0 을 반환하게 된다. 하지만 반환 하였다고 **generator 함수가 종료되는 것이 아니라 그대로 유지한 상태이다.**
4. x 값에는 yield 에서 전달 된 0 값이 저장된 후 print 된다. 그 후 for 문에 의해 다시 generator 함수가 호출된다.
5. 이때는 **generator 함수가 처음부터 시작되는게 아니라 yield 이후 구문부터 시작되게 된다.** 따라서 i += 1 구문이 실행되고 i 값은 1로 증가한다.
6. 아직 while 문 내부이기 때문에 yield 구문을 만나 i 값인 1이 전달 된다.
7. x 값은 1을 전달 받고 print 된다. (이후 반복)