PYTHON 반복자/생성자 이해하기

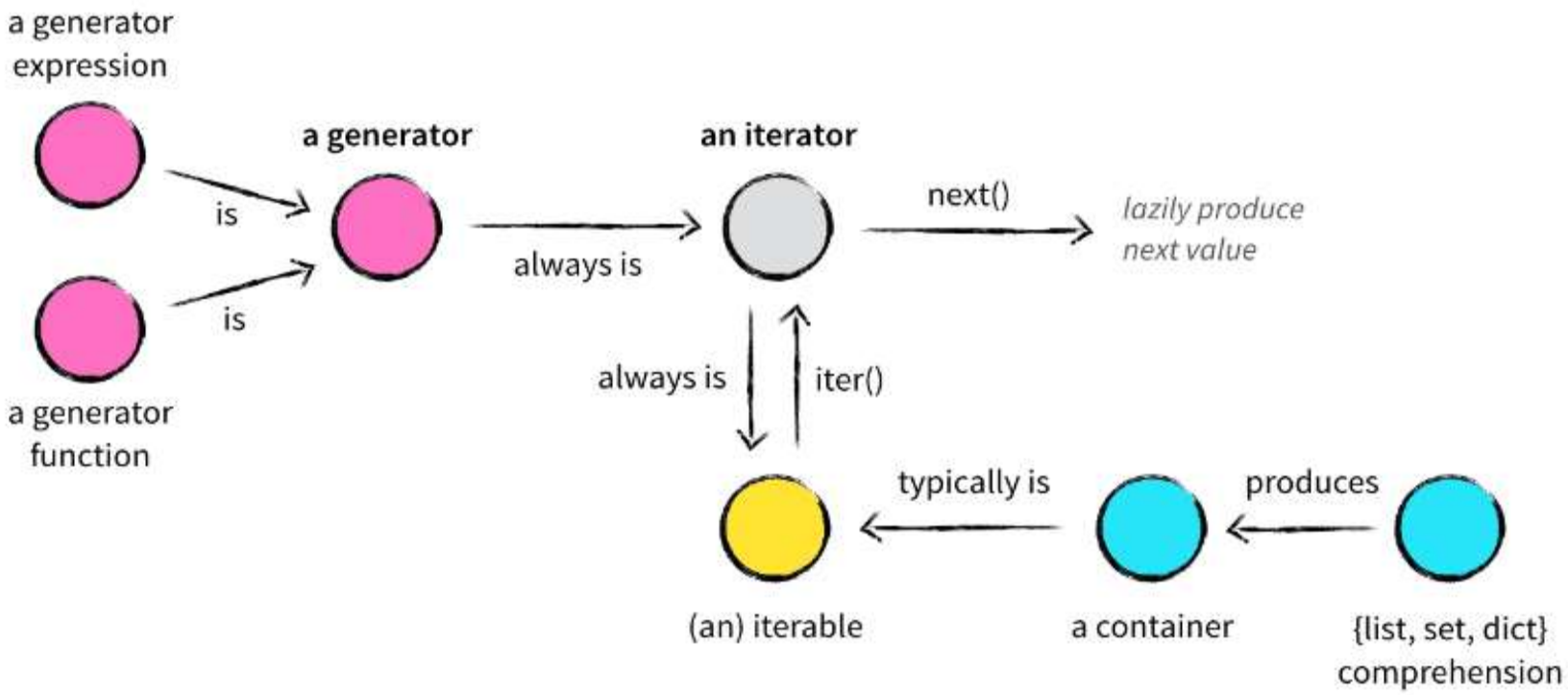
Moon Yong Joon

반복자와 생성자 이해

iterator/generator

iterator 타입 관계

파이썬 언어의 내장타입 클래스는 iterable, generator 생성되면 iterator로 처리가 가능



collections.abc 모듈 diagram

collections.abc 모듈 class diagram

iterator

Iterator는 next () 메서드를 사용하여 시퀀스의 다음 값을 가져 오는 객체입니다.

generator

생성기는 yield 메소드를 사용하여 일련의 값을 생성하거나 산출하는 객체/함수입니다.

Iterator/Generator 구조

Iterator와 Generator 차이는 close, send, throw 등의 메소드가 추가 됨

```
import collections.abc
print(collections.abc.Iterator.__bases__)

print(dir(collections.abc.Iterator))
```

```
(<class 'collections.abc.Iterable'>,)
['__abstractmethods__', '__class__',
'__delattr__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__',
'__gt__', '__hash__', '__init__', '__iter__',
'__le__', '__lt__', '__module__', '__ne__',
'__new__', '__next__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__',
'__sizeof__', '__slots__', '__str__',
'__subclasshook__', '_abc_cache',
'_abc_negative_cache',
'_abc_negative_cache_version',
'_abc_registry']
```

```
import collections.abc

print(collections.abc.Generator.__bases__)
print(dir(collections.abc.Generator))
```

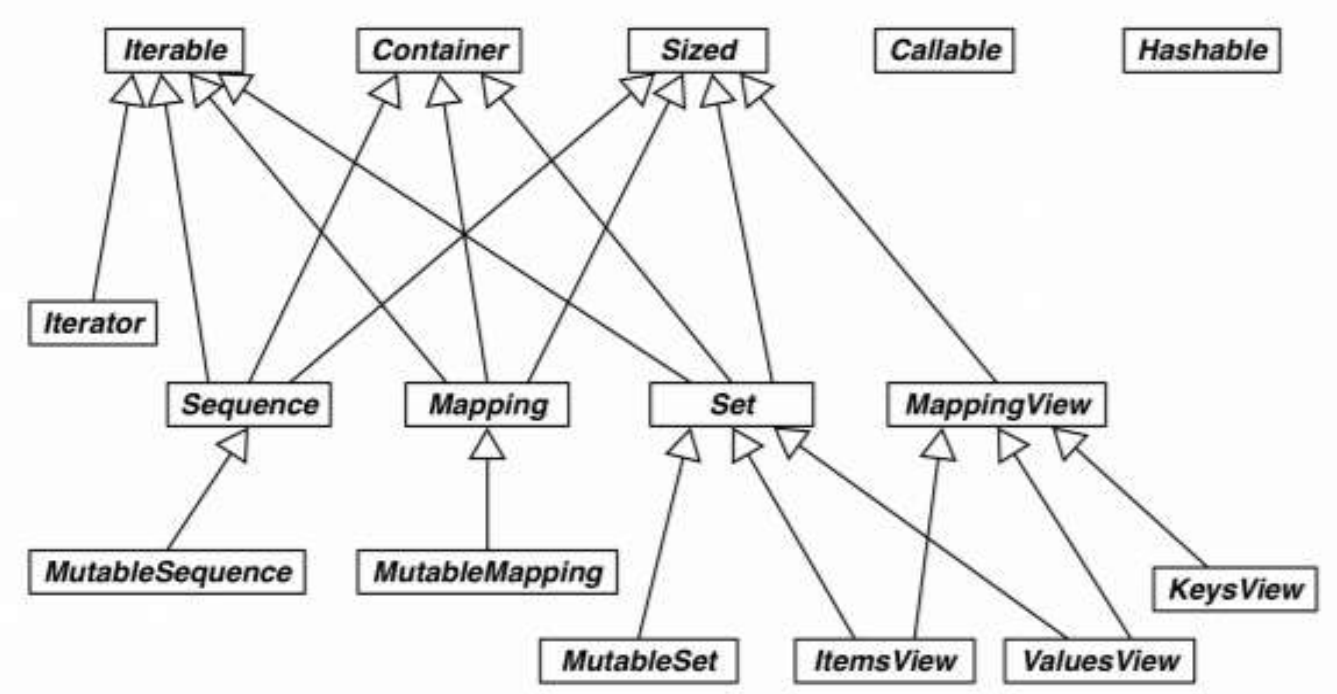
```
(<class 'collections.abc.Iterator'>,)
['__abstractmethods__', '__class__',
'__delattr__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__',
'__gt__', '__hash__', '__init__', '__iter__',
'__le__', '__lt__', '__module__', '__ne__',
'__new__', '__next__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__',
'__sizeof__', '__slots__', '__str__',
'__subclasshook__', '_abc_cache',
'_abc_negative_cache',
'_abc_negative_cache_version',
'_abc_registry', 'close', 'send', 'throw']
```

7

추상클래스 관계

collections.abc 모듈 diagram

collections.abc 모듈 class diagram



Fluent python 참조

ITERATOR 이해

Iterable/iterator 이해

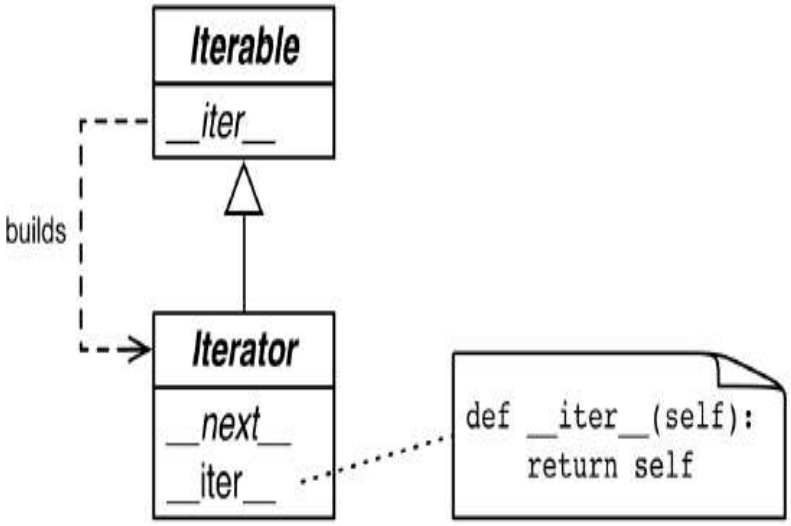
collections.abc 모듈 관계

ABC	Inherits from	Abstract Methods	Mixin Methods
Container		__contains__	
Iterable		__iter__	
Iterator	Iterable	__next__	__iter__

```
import collections.abc as cabc
import abc

class Iterator(cabc.Iterable):
    __slots__ = ()
    @abc.abstractmethod
    def __next__(self):
        'Return the next item from the iterator. When exhausted, raise StopIteration'
        raise StopIteration
    def __iter__(self):
        return self
    @classmethod
    def __subclasshook__(cls, C):
        if cls is Iterator:
            if (any("__next__" in B.__dict__ for B in C.__mro__) and
                any("__iter__" in B.__dict__ for B in C.__mro__)):
                return True
        return NotImplemented
```

iterable과 iterator의 차이

Container 타입은 반복할 수는 있는 iterable이지만 완전한 iterator 객체가 아님

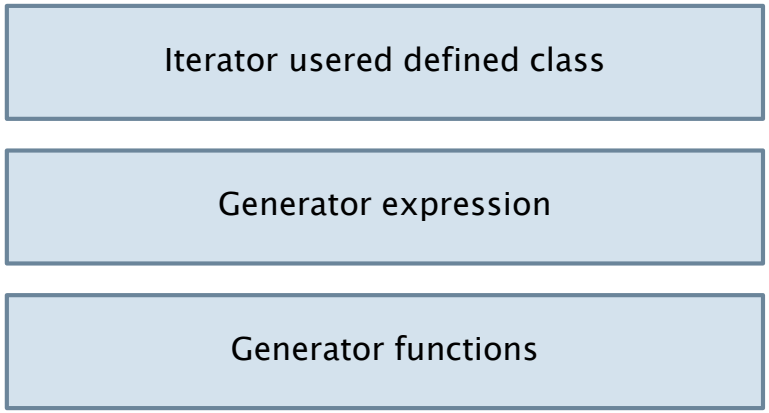



```
import collections.abc as abc

x = [1,2,3,4]

print(issubclass(type(x),abc.Iterable))
print(issubclass(type(x),abc.Iterator))
print(type(x))

xi = iter(x)
print(type(xi))
```

```
True
False
<class 'list'>
<class 'list_iterator'>
```

iterator types 타입 구조

iterator types은 스페셜 메소드 __iter__, __next__가 구현되어 있어야 함



iter 함수의 특징

1. 객체가 구현되는지, __iter__인지, iterator를 얻기 위해 객체를 호출하는지 검사합니다.
2. __iter__가 구현되지 않았지만 __getitem__이 구현되면 Python은 인덱스 0 (제로)로부터 순서에 따라 항목을 꺼내려고 하는 반복자.
3. 실패하면 Python은 TypeError를 발생시킵니다.

__iter__/__next__

iterator protocol은 기본으로
__iter__/__next__ 메소드가 존재하는 객체

```
x = [1,2,3,4]
xx = iter(x)
print(type(xx))
print(dir(xx))
```

<class 'list_iterator'>
['__class__', '__delattr__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__',
'__gt__', '__hash__', '__init__', '__iter__', '__le__',
'__length_hint__', '__lt__', '__ne__', '__new__',
'__next__', '__reduce__', '__reduce_ex__',
'__repr__', '__setattr__', '__setstate__',
'__sizeof__', '__str__', '__subclasshook__']

iter 함수의 특징 : 예시 1

1. 객체가 구현되는지, __iter__인지, iterator를 얻기 위해 객체를 호출하는지 검사합니다.

```
from collections import abc

class Foo:
    def __iter__(self):
        pass

print(issubclass(Foo, abc.Iterable))
f = Foo()
print(isinstance(f, abc.Iterable))
```

```
True
True
```

iter 함수의 특징 : 예시 2

2. __iter__가 구현되지 않았지만 __getitem__이 구현되면 Python은 인덱스 0 (제로)로부터 순서에 따라 항목을 꺼내려고 하는 반복자.

```
from collections import abc

class Foo:
    def __getitem__(self,index):
        pass


print(issubclass(Foo, abc.Iterable))
f = Foo()
print(isinstance(f, abc.Iterable))
sf = iter(f)

print(isinstance(sf, abc.Iterable))
```

```
False
False
True
```

next 함수 처리

sequence 타입 등을 iter()로 처리하면 iterator 객체가 만들어지고 실행됨

```
x = [1,2,3,4]
xx = iter(x)
print(type(xx))
print(next(xx))
print(next(xx))
print(next(xx))
print(next(xx))
print(next(xx))
```

```
<class 'list_iterator'>
1
2
3
4

---------------------------------------------------------------
StopIteration                                 Traceback
<ipython-input-125-22ab6bdd228d> in <module>()
```

sentinel

iter 함수 내에 sentinel에 값이 세팅되면 callable(함수 등)의 결과가 setinel보다 작을 때까지만 실행

```
help(iter)
```

```
Help on built-in function iter in module

iter(...)
    iter(iterable) -> iterator
    iter(callable, sentinel) -> iterator
```

```
x = 0
xc = lambda  : x+1
xx = iter(xc, 3)
print(next(xx))
x += 1
print(next(xx))
x += 1
print(next(xx))
x += 1
```

```
1
2

---------------------------------------------
StopIteration                             Traceb
<ipython-input-135-0a6cb70f6d9f> in <module>()
```

Iterator : getitem

사용자 정의 클래스 :getitem

Sentence 클래스를 sequence 구조로 정의.__iter__가 미존재

```
import re
import reprlib
RE_WORD = re.compile('\w+')
class Sentence():
    def __init__(self, text):
        self.text = text
        self.words = RE_WORD.findall(text)
    def __getitem__(self, index):
        return self.words[index]
    def __len__(self):
        return len(self.words)
    def __repr__(self):
        return 'Sentence(%s)' % reprlib.repr(self.text)


print(dir(Sentence))
s3 = Sentence('Pig and Pepper')
print(s3.words)
```

words라는 속성을 list 타입으로 만들어서 iterable한 class 생성

```
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format_
_', '__gt__', '__hash__', '__init__', '__le__', '__len__', '__lt__', '__module__',
ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__w
['Pig', 'and', 'Pepper']
```

사용자 정의 클래스: iter 처리

Sentence로 하나의 인스턴스를 만들고 iter 함수로 iterator 인스턴스를 만들어서 실행

```
s4 = Sentence('Pig and Pepper') #
it = iter(s4) #
print(dir(it))  # doctest: +ELLIPSIS
print(next(it)) #
print(next(it))
print(next(it))
```

```
['__class__', '__delattr__', '__dir__', '__doc__', '__eq__',
 '__init__', '__iter__', '__le__', '__length_hint__', '__lt_
 '__repr__', '__setattr__', '__setstate__', '__sizeof__', '_
Pig
and
Pepper
```

Iterator : __iter__/__next__

사용자 정의 클래스 : iter/next

SentenceIterator 클래스를 sequence 구조로 정의 __iter__ /__next__ 존재

```
import re
import reprlib
RE_WORD = re.compile('\w+')

class SentenceIterator:
    def __init__(self, words):
        self.words = words
        self.index = 0
    def __next__(self):
        try:
            word = self.words[self.index]
        except IndexError:
            raise StopIteration()
        self.index += 1
        return word

    def __iter__(self):

        for word in self.words:
            yield word
        return

class Sentence:
    def __init__(self, text):
        self.text = text
        self.words = RE_WORD.findall(text)
    def __repr__(self):
        return 'Sentence(%s)' % reprlib.repr(self.text)
    def __iter__(self):
        return SentenceIterator(self.words)
```

사용자 정의 클래스 실행

Sequence 클래스는 __iter__호출시
SequenceIterator를 실행해서 처리

```
s5 = Sentence('Pig and Pepper') #
it = iter(s5) #
print(dir(it))  # doctest: +ELLIPSIS
print(next(it)) #
print(next(it))
print(next(it))
```

```
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__f
 '__hash__', '__init__', '__iter__', '__le__', '__lt__', '__module__', '__ne_
ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__'
Pig
and
Pepper
```

Iterator : next 메소드

피보너치를 만들기

for 문으로 실행하면 next() 메소드가 호출되어 exception이 나오면 끝냄

```
class Fibs(object):
    def __init__(self):
        self.a = 0
        self.b = 1
    def next(self):
        self.a, self.b = self.b, self.a+self.b
        return self.a
    def __iter__(self):
        return self


f = Fibs()
print(f.next())
print(f.__dict__)
print(iter(f))
print(f.next())
print(f.__dict__)
print(f.next())
print(f.__dict__)
```

```
1
{'a': 1, 'b': 1}
<__main__.Fibs object at 0x7f29246459d0>
1
{'a': 1, 'b': 2}
2
{'a': 2, 'b': 3}
```

내림차순 Iterator 만들기

for 문으로 실행하면 next() 메소드가 호출되어 exception이 나오면 끝냄

```
class countdown(object):
    def __init__(self, start):
        self.count = start
    def __iter__(self):
        return self
    def next(self):
        if self.count <=0:
            raise StopIteration
        r = self.count
        self.count -= 1
        return r

c = countdown(10)
for i in c :
    print(i)
```

```
10
9
8
7
6
5
4
3
2
1
```

올림차순 Iterator 만들기

객체 내에 __iter__가 존재하고 이를 계속하게 부르는 next 메소드가 정의 되어야 함

```
class yrange:
     def __init__(self, n):
          self.i = 0
          self.n = n

     def __iter__(self):
          return self

     def next(self):
          if self.i < self.n:
               i = self.i
               self.i += 1
               return i
          else:
               raise StopIteration()

y = yrange(3)
print(y.next())
print(y.next())
print(y.next())
print(y.next())
```

```
0
1
2

---------------------------------------------------------------
StopIteration                                 Traceback (
<ipython-input-284-28cc7ef64c8a> in <module>()
```

내장 타입 (ITERABLE) 이해하기

Iterable

SEQUENCE 상속 class

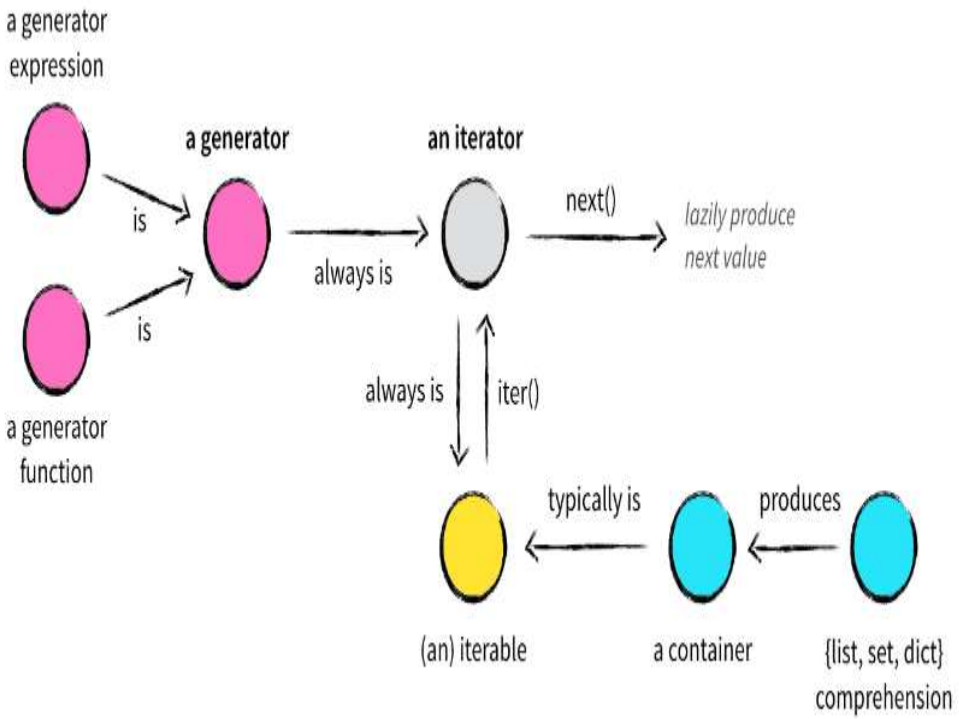
ABC	Inherits from	Abstract Methods	Mixin Methods
Container		__contains__	
Iterable		__iter__	
Sized		__len__	
Callable		__call__	
Sequence	Sized, Iterable, Container	__getitem__, __len__	__contains__, __iter__, __reversed__, index, and count
MutableSequence	Sequence	__getitem__, __setitem__, __delitem__, __len__, insert	Inherited Sequence methods and append, reverse, extend, pop,remove, and __iadd__
ByteString	Sequence	__getitem__, __len__	Inherited Sequence methods

MAPPING 모듈 관계

ABC	Inherits from	Abstract Methods	Mixin Methods
Mapping	Sized, Iterable, Container	__getitem__, __iter__, __len__	__contains__, keys, items, values, get, __eq__, __ne__
MutableMapping	Mapping	__getitem__, __setitem__, __delitem__, __iter__, __len__	Inherited Mapping methods pop, popitem, clear, update, and setdefault

내장타입 클래스

파이썬 언어의 내장타입 클래스는 iterable이지 iterator는 아님




```
import collections.abc as abc

print(issubclass(str, abc.Container))
print(issubclass(list, abc.Container))
print(issubclass(dict, abc.Container))
print(issubclass(set, abc.Container))

print(issubclass(str, abc.Iterable))
print(issubclass(list, abc.Iterable))
print(issubclass(dict, abc.Iterable))
print(issubclass(set, abc.Iterable))

print(issubclass(str, abc.Iterator))
print(issubclass(list, abc.Iterator))
print(issubclass(dict, abc.Iterator))
print(issubclass(set, abc.Iterator))
```

```
True
True
True
True
True
True
True
True
False
False
False
False
```

Iterable : list

list instance → listiterator

list 인스턴스에 iter()로 새로운 listiterator 객체를 만들면 내부에 next 메소드가 만들어짐

```
l= [1,2,3,4]
print(iter(l))
print(list.__dict__['__iter__'])

it = iter(l)
print(it.next())
print(it.next())
print(it.next())
print(it.next())
print(it.next())
```

```
<listiterator object at 0x7f43b55e0f90>
<slot wrapper '__iter__' of 'list' objects>
1
2
3
4

---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-79-6157d9d45f07> in <module>()
      8 print(it.next())
      9 print(it.next())
---> 10 print(it.next())

StopIteration:
```

for 문의 특징

for 표현식 in iterable : 로 처리하므로 iterator 객체로 자동변환해서 처리함

```
: for i in [1,2,3,4] :
      print(i)

  for j in iter([1,2,3,4]) :
      print(j)
```

```
1
2
3
4
1
2
3
4
```

ITERTOOLS 모듈 이해하기

count/cycle/repeat 증가

itertools.count/cycle/repeat를 이용해서 증가 및 반복

Iterator	Arguments	Results	Example
count()	start, [step]	start, start+step, start+2*step, ...	count(10) --> 10 11 12 13 14 ...
cycle()	p	p0, p1, ... plast, p0, p1, ...	cycle('ABCD') --> A B C D A B C D ...
repeat()	elem [,n]	elem, elem, elem, ... endlessly or up to n times	repeat(10, 3) --> 10 10 10

```
import itertools
gen = itertools.count(1, .5)
print(next(gen))
print(next(gen))
print(next(gen))
print(next(gen))
```

```
1
1.5
2.0
2.5
```

accumulate : 누적값

itertools. accumulate 를 이용해서 누적의 값을 구하기

```
import itertools
gen = itertools.accumulate([1,2,3])
for i in gen :
    print(i)
```

```
1
3
6
```

chain : 연결하기

itertools. chain 를 이용해서 문자열, 리스트 등을 연결하기

```
import itertools
gen = itertools.chain('ABC', 'DEF')
for i in gen :
    print(i,end=" ")
print()
gen1  = itertools.chain.from_iterable(['ABC', 'DEF'])
for i in gen1 :
    print(i,end=" ")
```

```
A B C D E F
A B C D E F
```

Combinatoric generators

Combinatoric generators

Combinatoric generators에 대한 함수들

Iterator	Arguments	Results
`product()`	p, q, ... [repeat=1]	cartesian product, equivalent to a nested for-loop
`permutations()`	p[, r]	r-length tuples, all possible orderings, no repeated elements
`combinations()`	p, r	r-length tuples, in sorted order, no repeated elements
`combinations_with_replacement()`	p, r	r-length tuples, in sorted order, with repeated elements
`product('ABCD', repeat=2)`		`AA AB AC AD BA BB BC BD CA CB CC CD DA DB DC DD`
`permutations('ABCD', 2)`		`AB AC AD BA BC BD CA CB CD DA DB DC`
`combinations('ABCD', 2)`		`AB AC AD BC BD CD`
`combinations_with_replacement('ABCD', 2)`		`AA AB AC AD BB BC BD CC CD DD`

permutations: 순열

itertools. combinations 를 이용해서 조합을 구하기

순열의 정의

① 서로다른 n개
② **중복을 허락하지 않고** r개를
③ **일렬로 나열**하는 수

중복X : 서로 다른 것을 사용해야 함을 의미
일렬로 나열 : 순서가 있다
▶ 실전에서는 직위나 위치등 모든 구분되는 모든 것에 배치를 하는 경우를 말함

순열의 수식 표현은

$${}_nP_r = n(n-1)(n-2)\cdots(n-r+1)$$

$$= \frac{n!}{(n-r)!}$$

```
import itertools
gen = itertools.permutations([1,2,3],2)
print(next(gen))
print(next(gen))
print(next(gen))
print(next(gen))
print(next(gen))
print(next(gen))
print(next(gen))
```

```
(1, 2)
(1, 3)
(2, 1)
(2, 3)
(3, 1)
(3, 2)
---------------------------------------------------
StopIteration                             Traceback
```

product: 중복순열

itertools. product 를 이용해서 조합을 구하기

중복순열의 정의

① 서로다른 n개

② **중복을 허락하고** r개를

③ **일렬로 나열**하는 수

중복O : 같은 것을 반복해서 사용해도 됨을 의미함

일렬로 나열 : 순서가 있다

▶ 실전에서는 직위나 위치등 모든 구분되는 모든 것에 배치를 하는 경우를 말함

중복순열의 수식 표현은

$${}_n\Pi_r = n^r$$

```
import itertools
gen = itertools.product([1,2,3],[1,2,3])
for i in gen :
    print(i)
```

```
(1, 1)
(1, 2)
(1, 3)
(2, 1)
(2, 2)
(2, 3)
(3, 1)
(3, 2)
(3, 3)
```

combinations: 조합

itertools. combinations 를 이용해서 조합을 구하기

조합(Combination)

집합에서 일부 원소를 취해 부분집합을 만드는 것

C(n,r) -> 이항계수 : n개 원소에서 r개의 부분집합을 고르는 조합의 경우의 수

$$C(n,r) = \frac{n!}{r! * (n-r)!}$$

```
import itertools
gen = itertools.combinations([1,2,3],2)
print(next(gen))
print(next(gen))
print(next(gen))
print(next(gen))
```

```
(1, 2)
(1, 3)
(2, 3)

---------------------------------------------------------------
StopIteration                                 Traceback
```

GENERATOR 이해하기

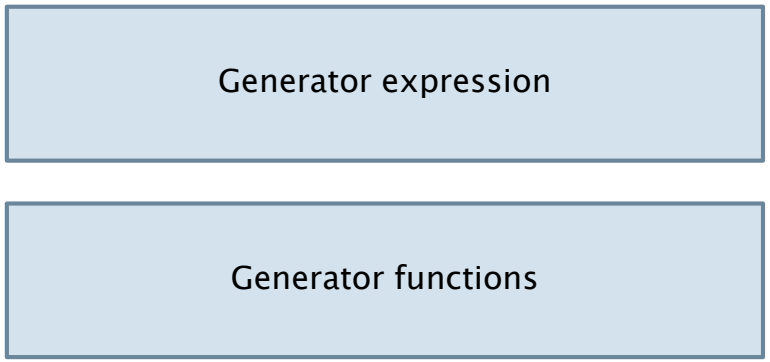
generator 관계

collections.abc 모듈 관계

ABC	Inherits from	Abstract Methods	Mixin Methods
Container		__contains__	
Iterator	Iterable	__next__	__iter__
Generator	Iterator	send, throw	close, __iter__, __next__

generator types 타입 구조

generator types은 iterator 스페셜 메소드 __iter__, __next__와 close, send, throw 메소드가 추가 구현되어 있어야 함



generator 표현식

Generator 생성

파이썬에는 tuple(immutable)에는 comprehension이 존재하지 않고 syntax 상으로 generator 가 생성

```
import collections.abc as abc
g = (i for i in range(3))
print(g)
print(issubclass(type(g), abc.Generator))
print(dir(g))
```

```
<generator object <genexpr> at
0x00000000051BB9E8>
True
 ['__class__', '__del__', '__delattr__', '__dir__',
'__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__gt__', '__hash__',
'__init__', '__iter__', '__le__', '__lt__',
'__name__', '__ne__', '__new__', '__next__',
'__qualname__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__',
'close', 'gi_code', 'gi_frame', 'gi_running',
'gi_yieldfrom', 'send', 'throw']
```

Generator 호출

generator를 호출하면 next 메소드를 통해 하나씩 호출해야 함. StopIteration 이 발생하면 종료됨

```
squares = ( i*i  for i in range(5))
for i in squares :
    print(i)
```

```
0
1
4
9
16
```

```
squares = ( i*i  for i in range(5))
while True :
    print(next(squares))
```

```
0
1
4
9
16

---------------------------------------------------
StopIteration                             Traceback
```

generator 직접 처리

Generator 호출 : send

send함수로 보낼 때는 일단 None을 인자로 전달하고 처리해야 함

```
squares = ( i*i  for i in range(5))

#send None value to a just-started generator
squares.send(None)
while True :
    print(next(squares))
```

```
1
4
9
16

---------------------------------------------------------
StopIteration                                 Traceback
```

unpack 처리

Iterable/generator/file unpack

iterable, generator, file 등도 모두 unpack처리가 가능함

```
g = (x for x in [1,2,3])
a,b,c = g
print(a, b, c)
```

```
(1, 2, 3)
```

```
%%writefile abc.txt
abcd
efgh
ijkl
```

```
Writing abc.txt
```

```
f = open("abc.txt","r")
a,b,c = f
print(a,b,c)
```

```
('abcd\n', 'efgh\n', 'ijkl')
```

```
i = iter([1,2,3])
a,b,c = i
print(a,b,c)
```

```
(1, 2, 3)
```

GENERATOR (함수)

Moon Yong Joon

Yield 결과

함수 결과 처리-yield

return 를 yield로 대체할 경우는 Generator가 발생

- 먼저 yield 의 뜻을 알아보면, 여기서는 '내다, 산출하다' 정도의 뜻으로 보인다.
- yield 를 사용하면 값을 바로 반환하는 것이 아니고, next() 를 통해 iteration 이 소진될 때까지 순차적으로 진행되며 값을 반환하고,

함수 결과 처리-yield from

3.3버전부터 yield from으로 for문을 한번 축약해서 처리가 가능함

```
def chain(*generators):
    for generator in generators:
        print(generator)
        for i in generator :
            yield i

c = chain(['a','bc'])

print(next(c))
print(next(c))
```

```
['a', 'bc']
a
bc
```

```
def chain(*generators):
    for generator in generators:
        print(generator)
        yield from generator

c = chain(['a','bc'])

print(next(c))
print(next(c))
```

```
['a', 'bc']
a
bc
```

일반함수와 생성자 함수 비교

일반 함수는 함수가 호출되면 결과값을 리턴하고 생성자 함수는 생성자객체를 생성한 후에 next 로 실행

```
def add_func(a, b):
    return a + b


def add_coroutine(a, b):
    yield a + b


# 함수호출
x = add_func(1, 2)
print(x)

# 제너레이터 호출
adder = add_coroutine(1, 2)
x = next(adder)
print(x)

#제너레이터 종료
x = next(adder)
```

```
3
3

---------------------------------------------
StopIteration
```

Generation :function

- 함수를 호출해도 계속 저장 함수를 호출
- 처리가 종료되면 exception 발생

```
def genFunc(n) :
    for i in n :
        yield i

gl = genFunc([1,2,3])
print(gl)
while True :
    print(next(gl))
```

```
<generator object genFunc at 0x00000000051DE938>
1
2
3

---------------------------------------------------
StopIteration                             Traceback
```

반복 호출 방법

함수 반복 호출

함수도 호출 방법에 따라 다양한 구현 및 처리가 가능

연속(재귀)호출	함수를 인자값을 바꿔가면 처리가 완료 될 때까지 연속해서 호출하여 처리
특정 시점 호출	함수를 구동시켜 필요한 시점에 호출하여 결과 처리(iteration, generation)
부분 호출	함수를 인자별로 분리하여 호출하면서 연결해서 결과를 처리

Generator 함수 : 단일호출

Yield에 값을 부여해서 함수 처리

```
def simple_generator_function():
    yield 1
    yield 2
    yield 3
for value in simple_generator_function():
    print(value)

our_generator = simple_generator_function()
print(next(our_generator))
print(next(our_generator))
print(next(our_generator))
```

```
1
2
3
1
2
3
```

Generator 함수 : 연속 호출

- 함수를 호출(next())해도 계속 저장 함수를 호출
- 처리가 종료되면 exception 발생

```
import numbers

def num_integer(num) :
    if issubclass(type(num),numbers.Integral) :
        return True

def gen_range(begin,step, end ) :

    if num_integer(begin) &  num_integer(step)& num_integer(end):
        result = begin
        forever = end
    else :
        result = 0
        forever = 0

    while  result < end:
        yield result
        result += 1

x = gen_range(1,1,3)
print(next(x))
print(next(x))
print(next(x))
```

```
1
2

---------------------------------------------------------------------
StopIteration                             Traceback (most recent ca
```

일반함수 재귀호출

함수 정의시 함수가 여러 번 호출될 것을 기준으로 로직을 작성해서 동일한 함수를 연속적으로 처리할 도록 호출

```
def factorial(n):
    print("factorial has been called with n = " + str(n))
    if n == 1:
        return 1
    else:
        result = n * factorial(n-1)
        print("intermediate result for ", n, " * factorial(" ,n-1, "): ",result)
        return result

print(factorial(5))
```

```
factorial has been called with n = 5
factorial has been called with n = 4
factorial has been called with n = 3
factorial has been called with n = 2
factorial has been called with n = 1
('intermediate result for ', 2, ' * factorial(', 1, '): ', 2)
('intermediate result for ', 3, ' * factorial(', 2, '): ', 6)
('intermediate result for ', 4, ' * factorial(', 3, '): ', 24)
('intermediate result for ', 5, ' * factorial(', 4, '): ', 120)
120
```

자신의 함수를 계속 호출하면 stack에 새로운 함수 영역이 생겨서 처리한다

Generator 함수 : 재귀 호출

recursive 함수 호출 대신 생성자 함수를 무한loop로 작동 시키고 생성자 함수로 재귀호출을 처리.

생성자함수를 close 시키면 더 이상 호출이 안됨

```
def fib():
    a, b = 0, 1
    while True:
        yield b
        a, b = b, a + b


# Usage:
f = fib()  # Create a new "instance" of the generator coroutine
print(next(f))  # Prints 1
print(next(f))  # Prints 1
print(next(f))  # Prints 2
print(next(f))  # Prints 3
print(next(f))  # Prints 5
f.close()
print(next(f))
```

```
1
1
2
3
5

---------------------------------------------------------------------
StopIteration                             Traceback (most recent
```

COROUTINE

Moon Yong Joon

Coroutine 기본

coroutine 사용 이유

coroutine은 특정 위치에서 실행을 일시 중단하고 다시 시작할 수있는 여러 진입 점을 허용하여 비 선점 멀티 태스킹을 위해 서브 루틴을 일반화하는 컴퓨터 프로그램 구성 요소입

Coroutine은 협동 작업, 예외, 이벤트 루프, 반복자, 무한 목록 및 파이프와 같이 익숙한 프로그램 구성 요소를 구현하는 데 적합

collections.abc 모듈 관계

Generator는 데이터 생성이고 coroutine은 데이터의 소비를 처리하므로 send로 데이터를 전달해야 함

ABC	Inherits from	Abstract Methods	Mixin Methods
Awaitable		__await__	
Coroutine	Awaitable	send, throw	close

Coroutine 초기화

send로 초기화

변수에 yield를 할당할 경우 이 제너레이터 함수에 값을 send 메소드에 None를 전달해서 초기화

```
def cf():
    x = 1
    while True:
        val = yield
        if x == 1 :
            print val
            x = 2
        print val,
    c = cf()
    c.send(None)
for i in range(5) :
    c.send(i)
```

```
0 1 2 3 4
```

next 함수로 초기화

코루틴을 최초 실행여부를 next함수로 초기처리가 필요

```
def cor():  # Coroutine
    while True:
        i = yield
        print('%s consumed' % i)


c = cor()
#conroutine start
next(c)
c.send(1)
c.send(2)
c.send(3)
```

```
1 consumed
2 consumed
3 consumed
```

Decorator를 이용해서 초기화

Decorator로 부분에 초기화를 넣어 실행함수를 처리하도록 정의하고 실행함수에 실행부분을 처리

```
def coroutine(f):
    def wrapper(*args, **kw):
        c = f(*args, **kw)
        c.send(None)     # This is the same as calling ``next()``,
                         # but works in Python 2.x and 3.x
        return c
    return wrapper

@coroutine
def worker():
    try:
        while True:
            i = yield
            print("Working on %s" % i)
    except GeneratorExit:
        print("Shutdown")

w = worker()
w.send(1)
w.send(2)
w.close()
w.send(3)
```

```
Working on 1
Working on 2
Shutdown

---------------------------------------------------------------------------
StopIteration                             Traceback (most recent
```

Coroutine 처리

generator vs.coroutine 1

생성자는 데이터를 생성하므로 반복자 처리하면 되지만 coroutine은 데이터를 전달해야 하므로 실세 coroutine 내에서 데이터를 소비해야 함

```
def squares():
    for i in range(10):
        yield i * i # send data

for j in squares():
    print(j, end=" ")
```

```
0 1 4 9 16 25 36 49 64 81
```

```
def squares():
    while True :
        i = yield # recieve data
        i *=  i
        print(i, end=" ")

s = squares()
s.send(None)
for j in range(10) :
    s.send(j)
```

```
0 1 4 9 16 25 36 49 64 81
```

generator vs.coroutine 2

yield를 문을 보시면, 뒤에 변수가 없을 경우는 yield를 입력으로 받고(coroutine), 변수가 있을 경우는 출력(generator)

```
def coro():
    hello = yield "Hello"
    yield hello


c = coro()
print(type(c))
print(next(c))
print(next(c))
```

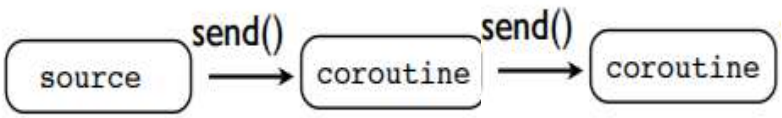
```
<class 'generator'>
Hello
None
```

```
def coro():
    hello = yield
    yield hello
    yield hello

c = coro()
print(type(c))
print(c.send(None))
print(c.send("Hello"))
print(c.send("World"))
```

```
<class 'generator'>
None
Hello
Hello
```

Generator 함수 : 연계 처리

Generator 함수간 정보 전달을 위해서는 send 메소드를 이용해서 처리

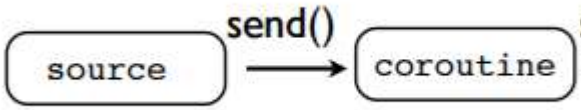



```python
import random

def cf() :
    while True :
        val = yield
        print(val, end="")

def pf(c) :
    while True :
        val = yield
        c.send(val)


c = cf()
# c generator start
c.send(None)

# p generator start
p = pf(c)
p.send(None)

for wow in range(10) :
    p.send(wow)
```

```
0123456789
```

Generator 함수 : 연계 처리

함수 내부에서 타 함수에 send 호출




```
import random

def cf() :
    while True :
        val = yield
        print(val, end="")

def pf(c) :
    while True :
        val = random.randint(1,10)
        c.send(val)
        yield

c = cf()
# c generator start
c.send(None)

# c generator start
p = pf(c)
for wow in range(10) :
    next(p)
```

```
871065594101
```