Iterable (이터러블)

iterable에 대한 python docs의 정의를 보자.

Iterable

An object capable of returning its members one at a time. Examples of iterables include all sequence types (such as **list**, **str**, and **tuple**) and some non-sequence types like **dict** and **file** and objects of any classes you define with an **__iter__()** or **__getitem__()** method. Iterables can be used in a for loop and in many other places where a sequence is needed (**zip()**, **map()**, ...). When an iterable object is passed as an argument to the built-in function **iter()**, it returns an iterator for the object. This iterator is good for one pass over the set of values. When using iterables, it is usually not necessary to call iter() or deal with iterator objects yourself. The **for** statement does that automatically for you, creating a temporary unnamed variable to hold the iterator for the duration of the loop. See also iterator, sequence, and generator.

iterable 의 의미는 member를 하나씩 차례로 반환 가능한 object를 말한다.
iterable 의 예로는 sequence type인 **list, str, tuple** 이 대표적이다.

```
>>> for x in range(5):
...     print x
...
0
1
2
3
4
```

우리가 당연하게 사용했던 위와 같은 for 문은 사실 range() 로 생성된 list가 iterable 하기 때문에 순차적으로 member들을 불러서 사용이 가능했던 것이다.

non-sequence type 인 dict 나 file 도 iterable 하다고 할 수 있다. dict 또한 for 문을 이용해 순차적으로접근이 가능하다.

```
>>> x = { 'a': 1, 'b' : 2, 'c': 3 }
>>>
>>> for y in x:
...     print y
...
a
c
b
```

또한 __iter__() 나 __getitem__() 메소드로 정의된 class 는 모두 iterable 하다고 할 수 있다.

iterable 은 for loop 말고도, zip(), map()과 같이 sequence 한 특징을 필요로 하는 작업에 유용하게 사용된다.

zip([iterable, ...])
This function returns a list of tuples, where the i-th tuple contains the i-th element from each of the argument sequences or iterables.

map(function, iterable, ...)
Apply function to every item of iterable and return a list of the results.

zip() 이나 map() 함수의 경우 iterable 을 argument 로 받는 것으로 정의되어 있다.

Iterator (이터레이터)

Iterator

An object representing a stream of data. Repeated calls to the iterator's **next()** method return successive items in the stream. When no more data are available a **StopIteration** exception is raised instead. At this point, the iterator object is exhausted and any further calls to its **next()** method just raise **StopIteration** again. Iterators are required to have an **__iter__()** method that returns the iterator object itself so every iterator is also iterable and may be used in most places where other iterables are accepted. One notable exception is code which attempts multiple iteration passes. A container object (such as a **list**) produces a fresh new iterator each time you pass it to the **iter()** function or use it in a **for** loop. Attempting this with an iterator will just return the same exhausted iterator object used in the previous iteration pass, making it appear like an empty container.

Iterator 는 next() 메소드로 데이터를 순차적으로 호출 가능한 object 이다. 만약 next() 로 다음 데이터를 불러올수 없을 경우 (가장 마지막 데이터인 경우) StopIteration exception을 발생시킨다.

그렇다면, iterable 한 object들은 iterator 인가?
결론부터 말하자면, iterable 이라고 해서 반드시 iterator 라는 것은 아니다.

```
>>> x = [1,2,3]
>>> next(x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: list object is not an iterator
```

list 는 iterable 이지만, 위와 같이 next() 메소드로 호출해도 동작하지 않는다. iterator 가 아니라는 에러 메시지를 볼 수 있다.
만약, iterable 을 iterator 로 변환하고 싶다면, **iter()** 라는 built-in function 을 사용하면 된다.

```
>>> x = [1,2,3]
>>> type(x)
<type 'list'>
>>> y = iter(x)
>>> type(y)
<type 'listiterator'>
```

위와 같이 iter() 함수를 사용하여 list 를 listiterator 타입으로 변경 가능하다.

```
>>> next(y)
1
>>> next(y)
2
>>> next(y)
3
>>> next(y)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

이제 next() 를 이용해 list의 정보를 하나씩 꺼낼 수 있다. 그리고 마지막 정보를 호출 한 이후에 next() 를 호출하면 StopIteration 이라는 exception 이 발생됨을 볼 수 있다.

하지만, 우리가 list 나 tuple 같은 iterable 한 object 를 사용할때 굳이 iter() 함수를 사용하지 않아도 for 문을 사용하여 순차적으로 접근이 가능하였다. 이것은 for 문으로 looping 하는 동안, python 내부에서 임시로 list를 iterator로 자동 변환해주었기 때문이다.