4.6 추가 상태를 가진 제너레이터 함수 정의

[문제] 제너레이터 함수 정의 시 사용자에게 노출할 추가상태를 넣고 싶음
[해결] __iter__() 메소드에 제너레이터 함수코드를 넣어서 클래스로 구현

```
from collections import deque
class linehistory:
    def __init__(self, lines, histlen=3):
        self.lines =lines
        self.history = deque(maxlen =histlen)

    def __iter__(self):
        for lineno, line in enumerate(self.lines,1): #self.lines,1 세기 숫자 1부터 시작
            self.history.append((lineno,line))
            yield line

    def clear(self):
        self.history.clear()
```

cf. deque(데크) : 큐 중에 한쪽 방향이 아닌 양방향 큐. 양쪽방향에서 요소(element) 추가/삭제 가능
cf. enumerate : 루프 인덱스와 값을 한쌍으로 가져옴

클래스를 사용하려면 일반 제너레이터 함수처럼 대해야 함.
인스턴스를 만들기 때문에 history속성이나 clear()메소드 같은 내부 속성에 접근할 수 있음

```
with open('somefile.txt') as f:
    lines = linehistory(f)
    for line in lines:
        if 'python' in line:
            for lineno, hline in lines.history:
                print('{}:{}'.format(lineno,hline), end ='')
```

[토론]
제너레이터 사용 시 모든 작업을 함수로만 하려고 함. 제너레이터 함수가 프로그램의 다른 부분과 일반적이지 않게 (속성 노출, 메소드 호출로 조절하기 등) 상호작용해야 할 경우 코드가 복잡해짐
이럴 경우 클래스 정의만을 사용
제너레이터를 __iter__() 메소드에 정의한다 해도 알고리즘 작성방식에는 변화가 없음. 클래스의 일부라는 점으로 인해 사용자에게 속성과 메소드를 쉽게 제공할 수 있음

for문 대신 다른 기술을 사용해서 순환하면 iter()를 호출할 때 추가적으로 작업을 해야할 필요가 생김

```
f = open('somefile.txt')
lines =linehistory(f)
next(lines) #제너레이터라서 실행이 안됨

#iter()를 먼저 호출하고 순환 시작
it = iter(lines)
next(it) #'hello word\n
next(it) # this is a test\n'
```

4.7 이터레이터의 일부 얻기

[문제] 이터레이터가 만드는 데이터의 일부를 얻고 싶지만 일반적인 자르기 연산자가 동작하지 않음
[해결] 이터레이터와 제너레이터 얻기 위해선 itertools.islice() 함수 사용

```
def count(n):
    while True:
        yield n
        n += 1
c = count(0)
c[10:20]
# Traceback (most recent call last):
#   File "<input>", line 6, in <module>
# TypeError: 'generator' object is not subscriptable


import itertools
for x in itertools.islice(c,10,20):
    print(x)

# 10
# 11
# 12
# 13
# 14
# 15
# 16
# 17
# 18
# 19
```

[토론]
이터레이터와 제너레이터는 데이터의 길이를 알 수 없고, 인덱스 구현x라서 일반적으로 일부를 잘라낼 수 없음.
islice()의 실행결과는 원하는 아이템의 조각을 생성하는 이터레이터지만 시작 인덱스까지 모든아이템을 소비하고 버리는 식으로 수행. 그리고 그 뒤의 아이템은 마지막 인덱스르 만날 때까지 islice 객체가 생성

주어진 이터레이터 상에서 islice()가 데이터를 소비함!
이터레이터를 뒤로 감을 수는 없기 때문에 뒤로 돌아가는 동작이 중요하다면 데이터를 먼저 리스트로 변환하는 것이 좋음

4.8 순환 객체 첫 번째 부분 건너뛰기

[문제] 순환 객체의 아이템 순환 시, 처음 몇 가지는 관심없고 중간부터 순환하고 싶음
[해결] itertools.dropwhile() : 함수 사용을 위해 순환 객체 입력.
반환된 이터레이터는 넘겨준 함수가 True를 반환하는 동안 시퀀스의 첫번째 아이템 무시, 그 이후에는 전체 시퀀스 생성

```
def count(n):
    while True:
        yield n
        n += 1
c = count(0)
#c[10:20]
# Traceback (most recent call last):
#   File "<input>", line 6, in <module>
# TypeError: 'generator' object is not subscriptable
```

```
import itertools
for x in itertools.islice(c,10,20):
    print(x)

# 10
# 11
# 12
# 13
# 14
# 15
# 16
# 17
# 18
# 19
```

```
with open('/etc/passwd') as f:
    for line in f:
    print(line, end='')

#User database
#Note that this file is consulted directly only when the system is running
#in single-user mode. At other times, this information is provided by
#Open Directory
#..
#nobody : *:-2:-2:Unpriviledged User:/var/empty:/usr/bin/false
#root:*:0:0:System Administrator :/var/root:/bin/sh

#처음 나오는 주석을 무시하려면 아래와 같이 함
from itertools import dropwhile
with open('etc/passwd') as f:
    for line in dropwhile(lambda line: line.startswith('#'),f):
        print(line,end='')
#nobody : *:-2:-2:Unpriviledged User:/var/empty:/usr/bin/false
#root:*:0:0:System Administrator :/var/root:/bin/sh
```

```
#어디까지 생략해야할 지 정확한 숫자를 알고 있다면 itertools.islice()
from itertools import islice
items = ['a','b','c',1,4,10,15]   #얘는 제너레이터가 아닌데 굳이 왜 islice를 쓰는겨
for x in islice(items, 3, None):
    print(x)
```

#islice()에 전달한 마지막 None인자는 처음 세 아이템 뒤에 오는 모든 것을 원한 :3 이 아니라 3:

[토론]
dropwhile()과 islice()함수는 다음과 같이 복잡한 코드를 작성하지 않도록 도와줌

```
#처음 주석을 건너뜀
with open('/etc/passwd') as f:
    while True:
        line = next(f, '')  #이게 제너레이터 인가?
        if not line.startswith('#'):
            break #반복만 빠져나옴
#남아 있는 라인 처리
while line:
    #의미 있는 라인으로 치환
    print(line, end='') #end='' 줄바꿈 방지
    line = next(f, None)
```

```
#순환 객체 첫부분을 건너뛰는 것은 간단히 전체를 걸러내는 것과는 조금 다름.
with open('etc/passwd') as f:
    lines = (line for line in f if not line.startswith('#'))
    for line in lines:
        print(line, end='')
```

#파일 전체에 걸쳐 주석으로 시작하는 모든 라인 무시
제공한 함수가 만족하는 동안의 아이템은 무시하고 그 뒤에 나오는 아이템은 필터링 없이 모두 반환
순환 가능한 모든 것에 적용 가능하다. 처음에 크기를 알 수 없는 제너레이터 파일 등 모든 것이 포함됨

4.9 가능한 모든 순열과 조합 순환

[문제] 아이템 컬렉션에 대해 가능한 모든 순열과 조합을 순환하고 싶음
[해결] itertolls.permutations(): 아이템 컬렉션을 받아 가능한 모든 순열을 튜플 시퀀스로 생성

```
items =['a','b','c']
from itertools import permutations
for p in permutations(items):
    print(p)

# ('a', 'b', 'c')
# ('a', 'c', 'b')
# ('b', 'a', 'c')
# ('b', 'c', 'a')
# ('c', 'a', 'b')
# ('c', 'b', 'a')
```

```
#짧은 길이의 순열을 원할 경우 선택적으로 길이 인자 지정 가능
for p in permutations(items,2):
    print(p)
# ('a', 'b')
# ('a', 'c')
# ('b', 'a')
# ('b', 'c')
# ('c', 'a')
# ('c', 'b')
for p in permutations(items,1):
    print(p)
# ('a',)
# ('b',)
# ('c',)
#itertools.combinations()는 입력 받은 아이템의 가능한 조합 생성
```

```
from itertools import combinations
for c in combinations(items,3): #애는 숫자 안써주면 오류남
    print(c)
# ('a', 'b', 'c')
for c in combinations(items,1):
    print(c)
# ('a',)
# ('b',)
# ('c',)
```

```
from itertools import combinations_with_replacement
for c in combinations_with_replacement(items,3): #중복 조합
    print(c)
```

4.10 인덱스-값 페어 시퀀스 순환

[문제] 시퀀스 순환 시 어떤 요소를 처리하고 있는 지 번호를 알고 싶음
[해결] enumerate()

```
my_list =['a','b','c']
for idx, val in enumerate(my_list):
    print(idx,val)
# 0 a
# 1 b
# 2 c

my_list =['a','b','c']
for idx, val in enumerate(my_list,1):
    print(idx,val)
# 1 a
# 2 b
# 3 c
```

```
#에러메세지에 파일의 라인 번호를 저장하고 싶은 경우 유용
def parse_data(filename):
    with open(filename, 'rt') as f:
        for lineno, line in enumerate(f,1):
            fields = line.split()
            try:
                count=int(fields[1])
            except ValueError as e:
                print('Line {}: Parse error: {}'.format(lineno,e))
```

특정값의 출현을 위한 오프셋(offset)추적에 활용하기 좋다. 파일 내의 단어를 출현한 라인에 매핑하려면 enumerate()로 단어를 파일에서 발견한 라인 오프셋에 매핑

```
from collections import defaultdict
word_summary= defaultdict(list)
with open('myfile.txt','r') as f:
    lines = f.readlines()

for idx, line in enumerate(lines):
    words = [w.strip().lower() for w in line.split() ]   #줄을 공백으로 나눈애들 공백없애고 소문자로 근데 이거 어떻게 들어가는거임? append도 아니고??
    for word in words:
        word_summary[word].append(idx)
```

파일 처리 후 word_summary를 출력하면 각 단어를 키로 갖는 딕셔너리 형태가 됨
#키에 대한 값은 그 단어가 나타난 라인의 리스트가 됨
한 라인에 두번 나오면 그 라인은 두번 리스팅 되어 단순 지표를 알아볼 수 있도록 함 ? 키값은 고유값 아닌감?

[토론]
카운터 변수 생성보다 enumerate() 사용하셈 짱짱멘
enumerate() 가 반환하는 값은 연속된 튜플을 반환하는 이터레이터인 enumerate 객체의 인스턴스임
이 튜플은 전달한 시퀀스에 next()를 호출해 반환된 카운터와 값으로 이루어져 있음

한번 더 풀어줘야 하는 튜플의 시퀀스에 enumerate()를 사용하는 경우 조심

```
data = [(1,2),(3,4),(5,6),(7,8)]
#올바른 방법
for n, (x,y) in enumerate(data):
for n,x,y in enumerate(data):
```

4.11 여러 시퀀스 동시에 순환

[문제] 여러 시퀀스에 들어잇는 아이템을 동시에 순환하고 싶음
[해결] zip()함수 사용

```
xpts = [1,5,4,2,10,7]
ypts = [101,78,37,15,62,99]
for x,y in zip(xpts,ypts): #tuple(x,y)를 생성하는 이터레이터 순환, 한쪽 시퀀스의 모든 입력이 소비되었을 때 정리
    print(x,y)

# 1 101
# 5 78
# 4 37
# 2 15
# 10 62
# 7 99
```

```
a = [1,2,3]
b = ['x','y','z','w','a']
from itertools import zip_longest
for i in zip_longest(a,b):
    print(i)
# (1, 'x')
# (2, 'y')
# (3, 'z')
# (None, 'w') #길이 안맞는 애들은 다 none

for i in zip_longest(a,b,fillvalue=0):
    print(i) #길이 안맞는 애들은 다 0
# (1, 'x')
# (2, 'y')
# (3, 'z')
# (0, 'w')
# (0, 'a')
```

[토론]
zip()은 데이터를 묶어야 할 때 주로 사용

```
headers = ['name','shares','price']
values = ['ACME',100,409.1]

s = dict(zip(headers,values))
print(s) #{'name': 'ACME', 'shares': 100, 'price': 409.1}

for name, val in zip(headers,values):
    print(name, '=',val)

# name = ACME
# shares = 100
# price = 409.1
```

일반적이이진 않지만 zip()에 시퀀스를 두 개 이상 입력할 수 있음. 이런 경우 결과적으로 튜플에는 업력한 시퀀스의 개수 만큼이 아이템이 포함됨

```
a = [1,2,3]
b = [10,11,12]
c = ['x','y','z']
for i in zip(a,b,c):
    print(i)
# (1, 10, 'x')
# (2, 11, 'y')
# (3, 12, 'z')
```

zip()이 결과적으로 이터레이터를 생성함! 묶은 값이 저장된 리스트가 필요하다면 list() 함수 사용

```
zip(a,b)
<zip object at 0x103484a48>
list(zip(a,b))
[(1, 10), (2, 11), (3, 12)]
```

4.12 서로 다른 컨테이너 아이템 순환

[문제] 여러 객체에 동일한 작업을 수행해야 하지만 객체가 서로 다른 컨테이너에 들어있음.
중첩된 반복문을 사용해 코드의 가독성을 해치고 싶지 않음
[해결] itertools.chain() : 순환 가능한 객체를 리스트로 받고 마스킹을 통해 한번에 순환하 수 있는 이터레이터를 반환

```
from itertools import chain
a = [1,2,3,4]
b = ['x','y','z']
for x in chain(a,b):
    print(x)
# 1
# 2
# 3
# 4
# x
# y
# z
```

chain()은 일반적으로 모든 아이템에 동일한 작업을 수행하고 싶지만 이 아이템이 서로 다른 세트에 포함되어 있을 때 사용함
#여러 아이템 세트

```
active_items = set()
inactive_items = set()

#모든 아이템 한번에 순환
for item in chain(active_items, inactive_items):
    #작업

#이거는 별루임
for item in active_items:
    #작업
for item in inactive_items:
    #작업
```

itertools.chain()은 하나 혹은 그 이상의 순환 객체를 인자로 받음. 입력 받은 순환 객체 속 아이템을 차례대로 순환하는 이터레이터 생성.
-> 우선적으로 시퀀스를 하나로 합친 다음 순환 < chain()이 효율적

#비효율 : 두개를 합친 전혀 다른 시퀀스 생성 (a,b가 동일한 타입이어야 함)
for x in a+b:

#더 나은
for x in chain(a,b):

-> 입력한 시퀀스의 크기가 아주 크거나 타입이 다른 경우 chain() 사용

4.13 데이터 처리 파이프라인 생성

[문제] 데이터 처리를 데이터 처리 파이프 라인과 가은 방식으로 순차적으로 처리하고 싶음(unix 파이프 라인과 비슷하게) ex) 처리해야 할 방대한 데이터가 있지만 메모리에 한꺼번에 들어가지 않는 경우
[해결] 제너레이터 함수를 사용하는 것이 처리 파이프 라인을 구현하기 좋음.
ex)방대한 양의 로그 파일이 들어있는 디렉터리에 작업할 경우

```
foo/
  access-log-012007.gz
```

```
        access-log-022007.gz
        access-log-032007.gz
        ...
        access-log-012008
      bar/
        access-log-092007.bz2
        ...
        access-log-022008
```

각 파일에는 다음과 같은 데이터가 담겨 있음

```
      124.115.6.12 - - [10/Jul/2012:00:18:50 -0500] "GET /robots.txt ..." 200 71
      210.212.209.67 - - [10/Jul/2012:00:18:51 -0500] "GET /ply/ ..." 200 11875
      210.212.209.67 - - [10/Jul/2012:00:18:51 -0500] "GET /favicon.ico ..." 404 369
```



```
      61.135.216.105 - - [10/Jul/2012:00:20:04 -0500] "GET /blog/atom.xml ..." 304 -
      ...
```

```
def gen_find(filepat, top):
    #디렉토리 트리에서 와일드 카드 패턴에 매칭하는 모든 파일 이름을 찾음
    for path, dirlist, filelist in os.walk(top):
        for name in fnmatch.filter(filelist, filepat):
            yield os.path.join(path,name)

def gen_opener(filenames):
    #파일 이름 시퀀스를 하나씩 열어 파일 객체 생성
    # 다음 순환으로 넘어가는 순간 파일을 담음
    for filename in filenames:
        if filename.endswith('.gz'):
            f = gzip.open(filename, 'rt')
        elif filename.endswith('.bz2'):
            f = bz2.open(filename, 'rt')
        else:
            f = open(filename, 'rt')
        yield f
        f.close()

def gen_concatenate(iterators):
    # 이터레이터 시퀀스를 합쳐 하나의 시퀀스로 만듦
    for it in iterators:
        yield from it

def gen_grep(pattern, lines):
    #라인 시퀀스에서 정규식 패턴을 살펴봄
    pat = re.compile(pattern)
    for line in lines:
        if pat.search(line):
            yield line
```

함수를 모아 파이프 라인 생성
ex) python이란 단어를 포함하고 있는 모든 로그 라인 찾기

```
lognames = gen_find('access-log*', 'www')
files = gen_opener(lognames)
lines = gen_concatenate(files)
pylines = gen_grep('(?!)python', lines)
for line in pylines:
    print(line)

#파이프 라인을 확장하고 싶으면 제너레이터 표현식으로 표현식을 넣을 수 있음
#전송한 바이트 수를 찾고 총합을 구함
lognames = gen_find('access-log*', 'www')
files = gen_opener(lognames)
lines = gen_concatenate(files)
pylines = gen_grep('(?!)python', lines)
bytecolumn = (line.rsplit(None,1)[1] for line in pylines)
bytes = (int(x) for x in bytecolumn if x != '-')
print('Total', sum(bytes))
```

[토론]
파이프라인으로 데이터를 처리하는 방식은 파싱, 실시간 데이터 읽기, 주기적 폴링 등 다른 문제에도 사용할 수 있음

yield문이 데이터 생성자처럼 동작하고 for문은 데이터 소비자처럼 동작함
제너레이터가 쌓이면 각 yield가 순환을 하면 데이터의 아이템 하나를 파이프 라인의 다음 단계로 넘김
위 예제에서 sum()함수가 실질적으로 프로그램을 운용하며 제너레이터 파이프라인에서 한번에 하나씩 아이템을 꺼냄

-> 각 제너레이터 함수를 작게 모듈화 할 수 있음
-> 메모리 효율성도 높음

gen_concatenate() 의 목적: 입력 받은 시퀀스를 하나로 합치는 것
itertools.chain() 함수가 비슷한 기능이지만 이 함수는 묶어 줄 모든 순환 객체를 인자로 전달해야함

쓰다 포기…..무슨 말이니….?

4.14 중첩 시퀀스 풀기

[문제] 중첩된 시퀀스를 합쳐 하나의 리스트 생성
[해결] yield from 문이 있는 재귀 제너레이터 생성

```
from collections import Iterable
def flatten(items, ignore_types=(str,bytes)):
    for x in items:
        if isinstance(x,Iterable) and not isinstance(x, ignore_types):
            yield from flatten(x)
        else:
            yield x
items = [1,2,[3,4,[5,6],7],8]
for x in flatten(items):
    print(x)

# 1
# 2
# 3
# 4
~#8
```

isinstance(x, Iterable)이 순환가능한지 확인 -> y : yield from으로 모든 값을 하나의 서브 루틴으로 분출
=> 중첩되지 않은 시퀀스 하나 생성

추가적으로 전달 가능한 인자 ignore_types / not isinstance(x, ignore_types)로 문자열과 바이트가 순환 가능한 것으로 해석되지 않도록 했음
-> 리스트에 담겨 있는 문자열을 전달했을 때 문자르 하나하나 펼치지 않고 문자열 단위로 전개

```
items = ['Dave','Paula',['Thomas','Lewis']]
for x in flatten(items):
    print(x)
```

[토론]
서브루틴으로써 다른 제너레이터를 호출할 때 yield from 사용 (구문 이용하지 않을 경우 추가적인 for문을 작성해야 함)

```
def flatten(items, ignore_types=(str,bytes)):
    for x in items:
        if isinstance(x, Iterable) and not isinstance(x, ignore_types):
            for i in flatten(x):
                yield i
        else:
            yield x
```

문자열과 바이트 형은 하나하나 펼쳐주지 않도록 처리. 펼쳐지지 않을 타입이 있을 경우 ignore_types 인자에 추가

제너레이터 기반의 병렬 처리에서 yield from이 중요한 역할을 함(12.12)

4.15 정렬된 여러 시퀀스를 병합 후 순환

[문제] 정렬된 시퀀스가 여럿 있고, 하나로 합친 후 정렬된 시퀀스 순환
[해결] heapq.merge()
- heapq.merge()에 넣는 시퀀스는 모두 정렬되어 있어야함.
- 앞에서부터 읽어가면서 가장 작은 것부터 데이터 출력.
- 선택한 시퀀스에서 아이템을 읽고 모든 입력을 소비할 때까지 반복 처리

```
import heapq
a = [1,4,7,10]
b = [2,5,8,6,11]
for c in heapq.merge(a,b): #번갈아 가면서 출력되넹
    print(c)
# 1
# 2
# 4
# 5
# 7
# 8
# 6
# 10
# 11
```

[토론] heaps.merge 아이템에 순환적으로 접근하면 제공한 시퀀스를 한꺼번에 읽지 않음
아주 긴 시퀀스로 별다른 무리 없이 사용. 정렬된 두 파일 병합시 사용

```
import heapq
with open('sorted_file_1', 'rt') as file1, \
    open('sorted_file_2','rt') as file2, \
    open('merged_file','wt') as outf:

    for line in heapq.merge(file1,file2):
        outf.write(line)
```

4.16 무한 while 순환문을 이터레이터로 치환

[문제] 함수나 일반적이지 않은 조건 테스트로 인해 무한 while 순환문으로 데이터에 접근하는 코드
[해결] 입출력과 관련있는 프로그램에서 다음과 같이 사용

```
CHUNKSIZE = 8192
def reader(s):
    while True:
        data = s.recv(CHUNKSIZE)
        if data == b'':
            break
        process_data(data)

def reader(s):
    for chunk in iter(lambda: s.recv(CHUNKSIZE), b''):
        process_data(data)

import sys
f = open('/etc/passwd')
for chunk in iter(lambda : f.read(10),''):
    n = sys.stout.write(chunk)
```

[토론]
iter() 기능은 거의 알려져 있지 않음
선택적으로 인자 없는 호출 가능 객체와 종료 값으 입력으로 받음. 주어진 종료 값을 반환하기 전까지 무한 반복해서 호출가능 객체 호출
ex) 소켓이나 파일에서 특정 크기의 데이터 읽을 시 반복적으로 read(), recv() 호출하고 파일끝을 확인해야함
—>이걸 iter() 호출로 하나로 합칠 수 있음
—>lambda를 사용해서 인자를 받지 않는 호출 객체 생성 & 원하는 크기의 인자를 recv(), read()에 전달