1.1 시퀀스를 개별 변수로 나누기

: N개의 요소를 가진 튜플이나 시퀀스(문자열, 리스트, 튜플)을 변수 N개로 나누기

```
[ex1]
p =( 4,5 )
x , y = p   #거꾸로 하면 안됨
print(x)    #4
print(y)    #5
print( x, y)    # 4 5
```

```
[ex2]
data = [ 'ABC' , 50, 91.1, (2012,12,21)]
name , shares, price, (year, mon, day) = data # [?] 위의 튜플을 리스트로 바꾼 후 (year,mon,day)로 입력해도 출력됨
print(year) #2012
```

—> 요소 개수가 일치하지 않으면 에러 발생

- 언패킹은 튜플이나 리스트 뿐만 아니라 순환 가능한 모든 객체(문자열, 파일, 이터레이터, 제너레이터 포함)에 적용 가능

cf.
packing : 하나의 변수에 여러개의 값을 넣는 것
unpacking : 패킹된 변수에서 여러개의 값을 꺼내오는 것

ex) c = (3 ,4)
d, e = c #c값을 언패킹하여 d,e에 값을 넣음
f = d , e #변수 d, e를 에 패킹 # 튜플로만 들어가는 건가?
print(c, f) #(3, 4) (3, 4)

cf.
이터레이터 : 반복 가능한 개체 (.next 가 가능하면 이터레이터)
ex)이터레이터 = iter(list)
—> list를 이터레이터로 만듦. ist는 반복가능하지만 이터레이터는 아님 명시적으로 반복가능한 객체로 만들어서 사용해줘야함 뭔말이여(이터레이터 모르겠음 일단 패스)
제너레이터 : 이터레이터를 만들어주는 것(반복가능한 객체를 만들어주는 행위)
[참고]http://hackersstudy.tistory.com/9

```
[ex3]
s = 'Hello'
a,b,c,d,e= s
print(s)
```

- 언패킹 시 특정 값을 무시하는 방법도 있음. 이때, 선택한 변수명을 다른 곳에서 이미 사용하고 있는지 확인해야 함([!]해보니까 그럴 경우 동일하게 값이 씌워짐)

```
[ex4]
data = [ 'ABC' , 50, 91.1, (2012,12,21)]
_, shares, price,_ = data
print(shares)
```

1.2 시퀀스를 개별 변수로 나누기

순환체를 언패킹하려는데 요소가 N개 이상 포함되어 '값이 너무 많습니다'라는 예외 발생
—> 별 표현식을 통해 해결
—> [!]제외하는 애들은 변수명 지정, 묶어주고 싶은 애들은 별표로 표시
—> **길이를 알 수 없는 순환체 & 순환체에 들어있는 패턴이나 구조에 사용 시 편리**

[ex1]
학기 성적 산출 시 첫번째, 마지막 과제를 무시하고 나머지 평균을 사용

```
def drop_first_last(grades):
        first, *middle, last = grades
        return avg(middle)
```

[ex2]

```
record = ('dave', 'dave@email.com','777-777','746-555-1212','12')
name, email, *phone_numbers, num = record
print(name)
print(email)
print(phone_numbers) #['777-777', '746-555-1212']
```

별표가 붙어있는 변수가 리스트의 처음에도 위치 가능(가장 최근 지표, 나머지 지표와의 비교)

```
*trailing_qrts, current_qtr = sales_record
trailing_avg = sum(trailing_qrts) / len(trailing_qtrs)
return avg_comparison(trailing_avg, current_qtr)
```

```
trailing, current = [ 10,8,7,1,9,5,10,3]
print(trailing)
print(current)
```

[오홍! 신기]

```
records = [ ('foo',1,2), ('bar','hello'), ('foo',3,4)]
def do_foo(x,y):
        print('foo',x,y)

def do_bar(s):
        print('bar',s)

for tag , *args in records:
        if tag == 'foo':
                do_foo(*args)
        elif tag == 'bar':
                do_bar(*args)
```

- 문자열 프로세싱에서도 활용 가능 (구분자 있으면 그걸 기준으로도 가능하다는 소리, 안어려움)

```
line = 'nobody:*-2:-2:Unprivileged Users:/misoni/virtualEnvironment:/bin/python'
uname, *fields, homedir,sh = line.split(':')
print(uname) #nobody
print(fields) #['*-2', '-2', 'Unprivileged Users']
print(homedir)
print(sh)
```

- 언패킹 이 후 특정값을 버리고 싶을 경우 버리기용 변수 가농 가능 —> _ or ign

```
record = ('AMIE',50, 123.45, (12,18,2012))
name, *_, (*_,year) =record
print(name)
print(year)
```

-재귀 알고리즘을 사용하는 함수 작성도 가능(but, 파이썬의 재귀적 제약으로 실질적으로 사용하기에는 무리)

```
items = [1,10,7,4,5,9]
def sum(items):
        head, *tail = items  # head: 1 tail:[10,7,4,5,9]
        return head + sum(tail) if tail else head #이게 뭔말이야 p5
print( sum(items) )
```

1.3 마지막 N개 아이템 유지

순환이나 프로세싱 중 마지막으로 발견한 N개의 아이템을 유지하고 싶음

collections.deque
ex) 여러줄에 대해 간단한 텍스트 매칭을 수행하고 처음으로 발견한 N라인을 찾음

```
from collections import deque
def search(lines, pattern, history = 5 ):
        previous_lines = deque(maxlen=history)
        for line in line:
                yield line, previous_lines
        previous_lines.append(line)

#파일 사용 예
if __name__ == '__main__':
        with open('somefile.txt') as f:
                for line, prevlines in search(f, 'python', 5):
                        for pline in prevlines:
                                print(pline, end='')
                        print(line,end='')
                        print('-'*20)

무슨 말이여…..
```

아이템을 찾는 코드를 작성할 대 yield를 포함한 제네레이터 함수를 만듦
-> 검색 과정과 결과를 사용하는 코드 분리 가능 (제너레이터는 4.3참고)
deque(maxlen=N)으로 고정 크기 **큐**생성 (큐에 대해서 알아보장)
큐가 꽉찬 상태에서 새로운 아이템을 넣으면 마지막 아이템이 자동으로 삭제됨

```
q = deque(maxlen=3)
q.append(1)
q.append(2)
q.append(3)
print(q)  #deque([1, 2, 3], maxlen=3)

q.append(4)
print(q) #deque([2, 3, 4], maxlen=3)

q.append(5)
print(q) #deque([3, 4, 5], maxlen=3)
```

```
q=deque()
q.append(1)
q.append(2)
q.append(3)
print(q)

q.appendleft(4)
print(q) # >>>deque([4, 1, 2, 3])
q.pop()
print(q) #>>>deque([4, 1, 2 ]) 맨 끝에꺼 삭제
q.popleft()
print(q) #>>>deque([1, 2]) 맨 오른쪽 삭제
```

—> 큐의 양끝에 아이템을 넣거나 빼는 작업 ; 시간 복잡도 O(1) 소요 (< 리스트 작업 O(N)) 무슨 말인지 모르겠으나 리스트보다 빠르다는 말인듯

1.4 N 아이템의 최대 혹은 최소값 찾기

컬렉션 내부에서 가장 크거나 작은 N개의 아이템을 찾기

heapq모듈에 이 용도에 적합한 nlargest() / nsmallest() 두가지 함수 활용
- 찾고자 하는 아이템 개수가 상태적으로 작을 때 알맞음
- 최소값이나 최대값을 구하려 한다면 (N이 1) : min(), max() 사용
- N의 크기가 컬렉션 크기과 비슷해지면 컬렉션 정렬 후 사용하는 것이 빠름(sorted(items)[:N] / sorted(items)[-N:] 이용)

컬렉션? sorted? 힙?

```
import heapq
nums = [ 1,8,2,23,7,-4,18,23,42,37,2]
print(heapq.nlargest(3,nums)) #>>>[42, 37, 23] 큰 순서대로 나옴
print(heapq.nsmallest(3,nums)) #>>>[-4, 1, 2] 작은 순서대로 나옴

portfolio = [
        {'name': 'IBM', 'shares':100, 'price':91.1},
        {'name':'AAPL','shares':5, 'price':543.22},
        {'name':'FB','shares':200,'price':21.09},
        {'name':'HPQ','shares':35,'price':31.75},
        {'name':'YHOO', 'shares':45, 'price':16.35},
        {'name':'ACME','shares':75, 'price':115.65}
]

cheap = heapq.nsmallest(3,portfolio, key= lambda s: s['price']) lambda 알아보기
print (cheap)#[{'name': 'YHOO', 'shares': 45, 'price': 16.35}, {'name': 'FB', 'shares': 200, 'price': 21.09}, {'name':
'HPQ', 'shares': 35, 'price': 31.75}]
expensive = heapq.nlargest(3,portfolio, key= lambda s: s['price'])
print(expensive) #[{'name': 'AAPL', 'shares': 5, 'price': 543.22}, {'name': 'ACME', 'shares': 75, 'price': 115.65},
{'name': 'IBM', 'shares': 100, 'price': 91.1}]
```

가장 작거나 큰 N개의 아이템을 찾고 있고 N이 컬렉션 전체 크기보다 작으면 더 나은 성능을 제공함 (확 와닿게 이해는 안됨)
- 데이터를 힙으로 정렬시켜 놓는 리스트로 전환

```
import heapq
heap = list(nums) #원래 리스트인데 왜 리스트로 바꿈?
heapq.heapify(heap)
print(heap) #>>>[-4, 2, 1, 23, 7, 2, 18, 23, 42, 37, 8]
```

- 힙의 가장 중요한 기능은 heap[0]이 가장 작은 아이템이 됨
- N이 힙의 크기라고 하면 O(logN)의 시간 복잡도가 소요됨
- 가장 작은 수 세개를 연속으로 뽑으려면 아래와 같이 수행하면 됨

```
print( heapq.heappop(heap)) #>>> -4  0번째요소 뽑아주고, 그 요소를 삭제해주는 듯
print( heapq.heappop(heap)) ##>>> 1
print( heapa.heappop(heap) ) ##>>> 2
```

1.5 우선 순위 큐 구현

주어진 우선 순위에 따라 아이템을 정렬하는 큐를 구현, 항상 우선 순위가 가장 높은 아이템을 먼저 팝하도록 만들어야 함

몰라유

```
import heapq
class PriorityQueue:
        def __init__(self):
                self._queue =[]
                self._index = 0

        def push(self, item, priority):
                heapq.heappush(self._queue, (-priority, self._index, item)) #튜플형태

#priority : 큐 내부 아이템을 가장 높은 우선 순위>낮은 우선순위 순으로 정렬하기 무효화 됨( 가장 낮은 값에서 높은 값으로 정렬되는 일반적인 힙과는 반대)
#index : 우선순위가 동일한 아이템의 순서를 정할 때 사용, 일정하게 증가하는 인덱스 값을 유지하기 때문에 힙에 아이템이 삽입된 순서대로 정렬할 수 있음(우선순위가 동일한 경우에도 역할을 함)

                self._index += 1

        def pop(self):
                return heapq.heappop(self._queue)[-1] #맨 오른쪽에 있는 애빼기
```

```
class Item:
        def __init__(self, name):
                self.name = name
        def __repr__(self):
                return 'Item({!r})'.foramt(self.name)

q =PriorityQueue()
q.push(Item('foo'),1)
q.push(Item('bar'),5)
q.push(Item('spam'),4)
q.push(Item('grok'),1)
q.pop()
q.pop()
q.pop()
```

—> 가장 높은 순위의 아이템을 반환 , 순위가 같은 경우 삽입된 순서와 동일하게 반환

* heapq모듈의 사용법
heapq.heappush()
heapq.heappop()
list_queue의 첫번째 아이템이 가장 작은 순위를 가진 것처럼 아이템을 삽입하거나 제거

heaps.heappop()메소드는 항상 가장 작은 아이템을 반환하여 큐의 팝이 올바른 아이템에 적용될 수 있도록 함
합의 크기가 N일 때 푸시와 팝의 시간 복잡도가 O(logN)이므로 N이 커진다해도 상당히 효율적

1) 순서를 매길 수 없는 Item 인스턴스

```
a= Item('foo')
b= Item('bar')
a < b
Traceback (most recent call last):
  File "<input>", line 26, in <module>
TypeError: '<' not supported between instances of 'Item' and 'Item'
```

2) (priority, item) 튜플 : 우선순위가 다른 경우에만 비교 가능

```
a = (1,Item('foo'))
b = (5,Item('bar'))
a < b
c = (1, Item('grok'))
a < c
Traceback (most recent call last):
  File "<input>", line 30, in <module>
TypeError: '<' not supported between instances of 'Item' and 'Item'
```

3) 인덱스 추가 (priority, idex, item)

```
a = (1,0,Item('foo'))
b = (5,1,Item('bar'))
c = (1,2,Item('grok'))
print( a < b) #>>> True
print( a < c) #>>> True
```

스프레드 간 통신에 이 큐를 사용하려면 올바른 락과 시그널을 사용해야함 (12.3)

1.6 딕셔너리의 키를 여러 값에 매핑하기

딕셔너리의 키를 하나 이상의 값에 매핑하고 싶다(multidict)

딕셔너리 : 하나의 키에 하나의 값이 매핑되어 잇는 것
키에 여러 값을 매핑하려면 그 여러 값을 리스트나 세트와 같은 컨테이너에 따로 저장해두어야 함

```
d = { 'a' : [1,2,3],
              'b' : [4,5]
}

e = { 'a' : {1,2,3},
              'b' : {4,5}
}
```

튜플은?

사용 목적에 따라 리스트와 세트를 사용 —> 리스트 : 삽입 순서 o, 튜플 : 순서상관 x, 중복 x
이러한 딕셔너리를 쉽게 만들기 위해서는 collections모듈의 defaultdict를 사용
defaultdict 기능 중 에는 첫 번째 값을 자동으로 초기화하는 것이 있어서 아이템 추가에만 집중할 수 있음 (뭔말?)

```
from collections import defaultdict

d= defaultdict(list)
d['a'].append(1)
d['a'].append(2)
d['b'].append(4)
print(d) # >>> defaultdict(<class 'list'>, {'a': [1, 2], 'b': [4]})

d = defaultdict(set)
d['a'].add(1)
d['a'].add(2)
d['b'].add(4)
print(d) # >>> defaultdict(<class 'set'>, {'a': {1, 2}, 'b': {4}})
```

defaultdict사용 시 딕셔너리에 존재하지 않는 값이라도 한번이라도 접근 했던 키의 엔트리를 자동으로 생성함 ,

```
from collections import defaultdict

d= defaultdict(list)
d['a'].append(1)
d['a'].append(2)
d['b'].append(4)
print(d) # >>> defaultdict(<class 'list'>, {'a': [1, 2], 'b': [4]})

d = defaultdict(set)
d['a'].add(1)
d['a'].add(2)
d['b'].add(4)
print(d) # >>> defaultdict(<class 'set'>, {'a': {1, 2}, 'b': {4}})
```

이런 동작성이 마음에 들지 않는다면 딕셔너리의 setdefault() 사용

```
d = {} #일반 딕셔너리
d.setdefault('a',[]).append(1)
d.setdefault('a',[]).append(2)
d.setdefault('b',[]).append(4)

print(d) #{'a': [1, 2], 'b': [4]}
```

```
d = {} #일반 딕셔너리
d.setdefault('a',[]).append(1)
d.setdefault('a',[]).append(2)
d.setdefault('b',{}).add(4)

print(d) #튜플은 왜 안되는겨?
```

But, setdefault()가 자연스럽지 않다고 생각. 첫번째 값에 대해서 항상 새로운 인스턴스를 생성한다는 점은 언급하지 않더라도 말이다(빈 리스트[]를 만들었음)

멀티딕트를 만드는 건 어렵지 않음 but, 첫번째 값에 대한 초기화를 스스로 하려면 복잡한 과정이 필요함

```
d = {}
for key, value in pairs:
        if key not in d:
                d[key] = []
        d[key].append(value)

d = defaultdict(list)
for key, value in pairs:
        d[key].append(value)
뭔소리여

#pairs 없다고 오류남?
```

1.7 딕셔너리 순서 유지

딕셔너리를 만들고 순환이나 직렬화할 때 순서를 조절하고 싶음
딕셔너리 내부 아이템 순서를 조절하기 위해서는 collections 모듈의 OrderedDict사용

```
from collections import OrderedDict
d = OrderedDict()
d['foo'] =1
d['bar']= 2
d['spam'] = 3
d['grok'] = 4
print(d) #OrderedDict([('foo', 1), ('bar', 2), ('spam', 3), ('grok', 4)]) 이건 : 없이 , 로 되어 있넹 그냥 형태가 다른건가?
print(type(d)) #<class 'collections.OrderedDict'>
for key in d:
        print(key, d[key])
# foo 1
# bar 2
# spam 3
# grok 4
```

```
d = {}
d['foo'] =1
d['bar']= 2
d['spam'] = 3
d['grok'] = 4
print(d) #{'foo': 1, 'bar': 2, 'spam': 3, 'grok': 4}
```

OrderDict는 나중에 직렬화하거나 다른 포맷으로 인코딩할 다른 매핑을 만들 때 특히 유용
ex) JSON인코딩에 나타나는 특정 필드의 순서를 조절하기 위해 OrderedDict에 다음과 같이 데이터를 생성

```
import json
print( json.dumps(d)) #{"foo": 1, "bar": 2, "spam": 3, "grok": 4}
```

OrderedDict는 내부적으로 더블 링크드 리스트(doubly linked list)로 삽입 순서와 관련있는 키를 기억함
새로운 아이템을 처음으로 삽입하면 리스트의 제일 끝에 위치시킴
기존 키에 재할당을 한다하더라도 순서에는 변화가 생기지 않음
—> 더블 링크드 리스트를 사용하기 때문에 OrderedDict의 크기는 일반적인 딕셔너리에 비해 두배로 큼
메모리 소비가 크기 때문에 사용 시 유용성을 살펴야함

1.8 딕셔너리 계산

딕셔너리 데이터에 여러 계산을 수행 (최소값, 최대값, 정렬 등)
딕셔너리 내용애 대해 유용한 계산을 하려면 딕셔너리의 키와 값을 zip()으로 뒤집어 주는 것이 좋음(아래 예제 보셈)

```
prices = {
            'ACME' : 45.23,
            'AAPL' : 612.78,
            'IBM' : 205.55,
            'HPQ' : 37.20,
            'FB' : 10.75
}
min_price = min( zip(prices.values(), prices.keys())) #>>> (10.75, 'FB')
max_price = max( zip(prices.values(), prices.keys())) #>>> (10.75, 'FB') #zip역할이 뭐여

a = zip(prices.values(), prices.keys()) # >>> <zip object at 0x10521b708>
```

데이터에 순서를 매기기 위해서는 zip(), sorted()를 함께 사용

```
prices_sorted = sorted(zip(prices.values(), prices.keys())) #이게 어떻게 가격 낮-> 높 순으로 정렬이 되는거지?
print(prices_sorted)
#>>> [(10.75, 'FB'), (37.2, 'HPQ'), (45.23, 'ACME'), (205.55, 'IBM'), (612.78, 'AAPL')]
```

계산을 할 때 zip()은 단 한번만 소비할 수 있는 이터레이터를 생성함(아래 예 참고)

```
prices_and_names = zip(prices.values(), prices.keys())
print(min(prices_and_names))  #(10.75, 'FB')
print(max(prices_and_names)) # ValueError : max()인자가 비어 있음
```

딕셔너리에서 일반적인 데이터 축소를 시도하면 오직 **키**에서만 작업이 이루어짐

```
print( min(prices) ) #>>>AAPL
print( max(prices) ) #>>>IBM
```

딕셔너리의 값을 계산하고 싶으면 **values()** 메소드 사용

```
print( min(prices.values()) ) #10.75
print( max(prices.values()) ) #612.78
```

키 & 일치하는 값 정보를 알고 싶으면? (ex) 어떤 주식의 값이 가장 낮을까?)

```
min(prices, key= lambda k: prices[k] ) #>>>FB    엉엉 람다를 몰러
max(prices, key= lambda k: prices[k] ) #>>>AAPL
min_values = prices[ min(prices, key=lambda k:prices[k])] #>>>10.75
```

zip()을 포함한 해결책은 딕셔너리의 시퀀스를 (value, key) 페어로 뒤집는 것으로 문제를 해결함
튜플에 비교를 수행하면 값(value) 요소를 먼저 비교하고 뒤이어 키(key)를 비교함
—> 명령어 하나만으로 정확히 우리가 원하는 데이터 축소&정렬을 수행함 이게 왜 축소도 되는거야?

여러 엔트리가 동일한 값을 가지고 있을 때 비교 결과를 결정하기 위해서 키를 사용함
(ex) min(), max()를 계산할 때 중복된 값이 있으면 가장 작거나 큰 키를 가지고 있는 엔트리 반환) 키 말하는 건감?

```
prices = {'AAA' : 45.23, 'ZZZ': 45.23}
min(zip(prices.values(), prices.keys()))
max(zip(prices.values(), prices.keys()))

print(min(zip(prices.values(), prices.keys()))) #>>> (45.23, 'AAA')
print(max(zip(prices.values(), prices.keys()))) #>>> (45.23, 'ZZZ')
```

1.9 두 딕셔너리의 유사점 찾기

두 딕셔너리가 있고 여기서 유사점을 찾고 싶음(동일한 키, 동일한 값)

```
a = { 'x': 1,  'y' : 2 , 'z':3}
b = { 'w': 10, 'x': 11,  'y':2}
```

유사점을 찾기 위해서는 keys()와 items() 메소드에 집합 연산을 수행해야 함

* 동일한 키 찾기
a.keys() & b.keys() #>>>{'x', 'y'}
* a에만 있고 b에는 없는 키 찾기
print(a.keys() - b.keys()) #{'z'}
* (키, 값)이 동일한 것 찾기
print(a.items() & b.items()) #{('y', 2)} # items는 키, 밸류 같이 value는 밸류만? 이게 차이점인가?

위와 같은 연산을 사용해서 딕셔너리의 내용을 수정하거나 걸러낼 수 도 있음!
예를 들어 선택한 키를 삭제한 새로운 딕셔너리를 만들고 싶을 때 사용
* 특정 키를 제거한 새로운 딕셔너리 만들기
c= {key:a[key] for key in a.keys() - {'z','w'}}
print(c) #>>> {'x': 1, 'y': 2}

딕셔너리를 키와 값의 매핑으로 이루어짐. 딕셔너리의 keys() 메소드는 키를 노출하는 키-뷰(key-view) 객체를 반환
키 뷰에는 잘 알려지지 않았지만 합/교/여 집합 연간 기능이 있음
—> 딕셔너리 키에 연산을 수행하려면 집합으로 변환할 필요없이 키-뷰 객체를 바로 사용하면 됨

딕셔너리의 items() 메소드는 (key,value) 페어로 구성된 아이템-뷰(item-view)객체를 반환
—> 집합 연산과 유사한 것을 지원하므로 두 딕셔너리에 동일한 키-값 페어를 찾을 때 사용할 수 있음

values() 메소드는 집합 연산을 지원하지 않음. 키와는 다르게 값-뷰가 유일하다는 보장이 없기 때문
—> 비교를 수행해야 한다면 먼저 값을 집합으로 변환하면 됨 변환은 어케함?

1.10 순서를 깨지 않고 시퀀스의 중복 없애기

시퀀스에 중복된 값을 없애고 싶지만 아이템의 순서는 유지하고 싶다

시퀀스 값이 해시(hash)가능하다면 세트와 제너레이터를 사용해서 쉽게 해결 할 수 있음
해시…? 스터디 때 말한거???일단 나누고 맞는지 보는거 맞남

```
def dedupe(items):
        seen = set()
        for item in items:
                if item not in seen:
                        yield item
                        seen.add(item)

a = [ 1,5,2,1,9,1,5,10]
print( list(dedupe(a)) ) # >>> [1, 5, 2, 9, 10] 중복 없애주는거면 걍 set만해주고 list로 바꿔줘도 되지 않나? 왜 제너레이터를…? 해시 효율이랑 상관있는 건감….?
```

—> 시퀀스의 아이템이 해시 가능한 경우에만 사용 가능

해시가 불가능한 타입 (ex) dict)의 중복을 없애려면 수정이 필요함

```
def dedupe(items, key= None):
        seen =set()
        for item in items:
                val = item if key is None else key(item)
                if val not in seen:
                        yield item
                        seen.add(val) 뭔 소리당가….
```

key인자의 목적은 중복 검사를 위해 함수가 시퀀스 아이템을 해시 가능한 타입으로 변환한다고 명시하는데 있음

```
a = [ {'x':1, 'y':2} ,  {'x':1, 'y':3} , {'x':1, 'y':2} , {'x':2, 'y':4}]
list(dedupe( a, key = lambda d: (d['x'],d['y'])))  이거 결과값 어떻게  봐
                                >>> [{'x':1, 'y':2} , {'x':1, 'y':3}, {'x':2, 'y':4}]
list(dedupe( a, key= lambda d: d['x']))
                                >>> [{'x':1, 'y':2}, {'x':2, 'y':4} ]  한가지만 있을 경우 제일 먼저 있는거 반환?
```

필드가 하나거나 커다란 자료 구조에 기반한 값의 중복을 없앨 때도 동작함 (뭐 그런갑다)

중복을 없애려면 대개 세트를 만드는 것이 가장 쉬움
a = [1,5,2,1,9,1,5,10]
print (set(a)) #{1, 2, 5, 9, 10}
print (a) # [1,5,2,1,9,1,5,10] 어떨때는 b=set(a) 이렇게 안해주고 set(a)를 만해줘도 a가 바뀌는 애들이 있던데
But, 기존의 데이터 순서가 훼손됨 —> 앞의 방법 활용(나중에 해보기 제너레이터 몰라서 못하겠음)

제너레이터 함수를 사용했기 때문에 단순히 리스트 프로세싱 말고도 아주 일반적인 목적으로 함수를 사용할 수 있음.
(예로 들어 파일을 읽어 들일 때 중복된 라인을 무시하려면 다음 코드 사용)

```
with open(somefile,'r') as f:
        for line in dedupe(f):
                ...
```

1.11 슬라이드 이름 붙이기

프로그램 코드에 슬라이스(slice)를 지시하는 하드코딩이 너무 많아 이해하기 어려운 상황.

고정된 문자열로부터 특정 데이터를 추출하는 코드가 있다고 가정

```
##### '0123456789012345678901234567890123456789012345678901234567890123456789'
record = '.....100       .....513.25     ..........' #이게 의미하는게 뭐야….
cost = int(record[20:32] * fload(record[40:48])

* 의미없는 하드코에 이름을 붙여서 이후에 이해하기가 훨씬 수월하도록 이름을 붙여줌!
SHARES = slice(20,32)
PRICE = slice(40,48)

cost = int(record[SHARES]) * float(record[price])
```

slice() : 내장함수, 슬라이스 받는 모든 곳에 사용할 수 있는 조각을 생성

```
items = [0,1,2,3,4,5,6]
a= slice(2,4)       # print(a) >>> slice(2, 4, None)
items[2:4]                  #print(items[2:4]) >>       [2,3]
items[a]            #print(items[a]) >> [2,3]
items[a] = [10,11]
items                       #print(items) >>> [0,1,10,11,4,5,6]
del items[a]
items                       #[0,1,4,5,6]

[] 이렇게 자르는거랑  slice차이? 간격 조정할 수 있다는 건감?
```

slice인스턴스가 있다면 s.start와 s.stop, s.step 속성을 통해 좀 더 많은 정보를 얻을 수 있음
a = slice(10,50,2)
a.start #10
a.stop #50
a.step #2

cf. https://www.programiz.com/python-programming/methods/built-in/slice

start - starting integer where the slicing of the object starts

stop - integer until which the slicing takes place. The slicing stops at index **stop - 1**.

step - integer value which determines the increment between each index for slicing

추가적으로 indices(size) 메소드를 사용하면 특정 크기의 스퀀스에 슬라이스를 매핑할 수 있음
이렇게 하면 튜플(start, stop, step)을 반환
모든 값은 경계를 넘어서지 않도록 제약이 걸려있음(인덱스에 접근할 때 IndexError 예외가 발생하지 않도록 하기 위함)

```
s = 'Helloworld'
a.indices(len(s)) # ?
for i in range(*a.indices(len(s))) :
        print(s[i]) ···ㅠㅠ
```

1.12 시퀀스에 가장 많은 아이템 찾기

```
collections.Counter 이용 ( 심지어는 지금 같은 상황에 꼭 알맞는 most_common() 메소드도 제공)
words = [ 'look', 'into','my', 'eyes' , 'look', 'into','my', 'eyes',
                    'the','eyes','the','eyes','the','eyes','not','around','the',
                    'eyes','dont', 'look','around','the','eyes','look','into',
                    'my','eyes',"your're",'under' ]

from collections import Counter
word_counts = Counter(words)
top_three =word_counts.most_common(3)
print(top_three) #>>>[('eyes', 8), ('the', 5), ('look', 4)]
```

Counter객체에는 해시 가능한 모든 아이템을 입력할 수 있음. 내부적으로 Counter는 아이템이 나타난 횟수를 가리키는 딕셔너리

```
word_counts
#Counter({'eyes': 8, 'the': 5, 'look': 4, 'into': 3, 'my': 3, 'around': 2, 'not': 1, 'dont': 1, "your're": 1, 'under': 1})

word_counts['not'] #>>> 1
word_counts['eyes'] # >>> 8
```

카운트를 수동으로 증가시키고 싶으면 단순히 더하기 사용

```
morewords = ['why','are','you','not','looking','in','my', 'eyes']
for word in morewords:
        word_counts[word] += 1
        #print(word_counts)
word_counts['eyes']
```

or update()메소드 사용

```
word_counts.update(morewords)
a = Counter(words)
#>>>Counter({'eyes': 8, 'the': 5, 'look': 4, 'into': 3, 'my': 3, 'around': 2, 'not': 1, 'dont': 1, "your're": 1, 'under': 1})
b= Counter(morewords)
#>>>Counter({'why': 1, 'are': 1, 'you': 1, 'not': 1, 'looking': 1, 'in': 1, 'my': 1, 'eyes': 1})
print(a)
print(b)
#카운트 합치기
c = a+b
#>>> Counter({'eyes': 9, 'the': 5, 'look': 4, 'my': 4, 'into': 3, 'not': 2, 'around': 2, 'dont': 1, "your're": 1, 'under': 1,
'why': 1, 'are': 1, 'you': 1, 'looking': 1, 'in': 1})
```

—> 데이터 개수를 파악하는 문제는 **Counter** 객체로!!! 딕셔너리를 직접 사용해서 아이템 개수를 세는 방식보다는 Counter 사용을 권장

1.13 일반 키로 딕셔너리 리스트 정렬

딕셔너리 리스트가 있고, 하나 혹은 그 이상의 딕셔너리 값으로 이를 정렬하고 싶음
operator모듈 itemgetter 함수 사용

```
rows = [
        {'fname':'Brian','lname':'Jones','uid':1003},
        {'fname':'David','lname':'Beazley','uid':1002},
        {'fname':'John','lname':'Cleese','uid':1001},
        {'fname':'Big','lname':'Jones','uid':1004}
]

from operator import itemgetter

rows_by_fname = sorted(rows, key=itemgetter('fname'))
rows_by_uid =sorted(rows, key=itemgetter('uid'))

print(rows_by_fname) #[{'fname': 'Big', 'lname': 'Jones', 'uid': 1004}, {'fname': 'Brian', 'lname': 'Jones', 'uid': 1003},
{'fname': 'David', 'lname': 'Beazley', 'uid': 1002}, {'fname': 'John', 'lname': 'Cleese', 'uid': 1001}]

print(rows_by_uid) ##[{'fname': 'John', 'lname': 'Cleese', 'uid': 1001}, {'fname': 'David', 'lname': 'Beazley', 'uid':
1002}, {'fname': 'Brian', 'lname': 'Jones', 'uid': 1003}, {'fname': 'Big', 'lname': 'Jones', 'uid': 1004}]
```

itemgetter() 함수에는 키를 여러 개 전달할 수도 있음

```
rows_by_lfname = sorted(rows, key=itemgetter('lname','fname'))
print(rows_by_lfname)

# [{'fname': 'David', 'lname': 'Beazley', 'uid': 1002},
#  {'fname': 'John', 'lname': 'Cleese', 'uid': 1001},
#  {'fname': 'Big', 'lname': 'Jones', 'uid': 1004},
#  {'fname': 'Brian', 'lname': 'Jones', 'uid': 1003}]
```

[설명]
- 키워드 인자 key를 받는 내장함수 sorted()에 rows를 전달함
- 이 인자는 rows로부터 단일 아이템을 받는 호출 가능 객체를 입력으로 받고 정렬의 기본이 되는 값을 반환함
- itemgetter()함수는 그런 호출 가능 객체를 생성 잘모르겠는데 아주 대충은 알겠음
- operator.itemgetter()함수는 rows레코드에서 원하는 값을 추출하는데 사용하는 인덱스를 인자로 받음(딕셔너리 키 이름, 숫자 리스트 요소, 객체의 __getitem__() 메소드에 넣을 수 있는 모든 값 가능)
- itemgetter()에 여러 인텍스를 전달하면 sorted()가 튜플 정렬 순서에 따라 순서를 잡음
- 여러 필드 동시 정렬 가능

itemgetter()의 기능을 때때로 lambda표현식으로 대체할 수 있음

```
rows_by_fname = sorted(rows, key= lambda r: r['fname'])
rows_by_lfname = sorted(rows, key= lambda r: ( r['lname'] , r['fname']))
```

1.14 기본 비교 기능 없이 객체 정렬

동일한 클래스 객체를 정렬해야 하는데 이 클래스는 기본적인 비교 연산을 제공하지 않음

sorted()는 key인자에 호출가능한 객체를 받아 sorted가 객체 비교에 사용할 수 있는 값 반환
ex) 애플리케이션에 User인스턴스를 시퀀스로 갖고 있고 이를 user_id요소를 기반으로 정렬하고 싶음.
이럴 경우 User인스턴스를 입력으로 받고 user_id를 반환하는 코드 작성할 수 있음 ? ???

```
class User:
        def __init__(self, user_id):
                self.user_id = user_id
        def __repr__(self):
                return 'User({})'.format(self.user_id)

users = [User(23), User(3),User(99)]
users # >>> [User(23), User(3), User(99)]

sorted(users, key=lambda u : u.user_id) #이건 프린트 안해도 걍 나오네 [User(3), User(23), User(99)]
```

lambda를 사용할지 attrgetter()를 사용할지 여부는 개인의 선호도에 따라 고고싱

attrgetter()의 속도가 빠른 경우가 종종 있고 동시에 여러 필드를 추출하는 기능이 추가되어 있음(이는 딕셔너리의 operator.itemgetter()를 사용하는 것과 유사한 점이 있음 1.13)
예를 들어 User인스턴스에 first_name과 last_name 속성이 있으면 다음과 같이 정렬할 수 있음

```
by_name = sorted(users, key=attrgetter('last_name','first_name'))

min(users, key= attrgetter('user_id')) #User(3)
max(users, key= attrgetter('user_id')) #User(99) 실행안대유ㅠㅠㅠ
```

1.15 필드에 따라 레코드 묶기

일련의 딕셔너리나 인스턴스가 있고 특정 필드 값에 기반한 그룹의 데이터를 순환하고 싶음
itertools.groupby()함수는 데이터를 묶는 유용

```
rows = [
        {'address': '5412 N CLARK' , 'date' : '07/01/2012'},
        {'address': '5148 N CLARK' , 'date' : '07/04/2012'},
        {'address': '5800 E 58TH' , 'date' : '07/02/2012'},
        {'address': '2122 N CLARK' , 'date' : '07/03/2012'},
        {'address': '5645 N RAVENSWOOD' , 'date' : '07/02/2012'},
        {'address': '1060 N ADDISON' , 'date' : '07/02/2012'},
        {'address': '4801 N BROADWAY' , 'date' : '07/01/2012'},
        {'address': '1039 W GRANVILLE' , 'date' : '07/04/2012'},
]
```

날짜로 구분 지을 데이터 조각을 순환해야함
1) 우선 원하는 필드에 따라 정렬(이번 경우엔 date필드)
2) intertools.groupby()사용

```
from operator import itemgetter
from itertools import groupby

#원하는 필드로 정렬
rows.sort(key=itemgetter('date'))

#그룹 내부에서 순환
for date, items in groupby(rows, key= itemgetter('date')):
        print(date)
        for i in items:
                print ( '___', i)


# 07/01/2012
#___{'address': '5412 N CLARK', 'date': '07/01/2012'}
#    {'address': '4801 N BROADWAY', 'date': '07/01/2012'}
# 07/02/2012
#    {'address': '5800 E 58TH', 'date': '07/02/2012'}
#    {'address': '5645 N RAVENSWOOD', 'date': '07/02/2012'}
#    {'address': '1060 N ADDISON', 'date': '07/02/2012'}
# 07/03/2012
#    {'address': '2122 N CLARK', 'date': '07/03/2012'}
# 07/04/2012
#    {'address': '5148 N CLARK', 'date': '07/04/2012'}
#    {'address': '1039 W GRANVILLE', 'date': '07/04/2012'}
```

groupby() 함수는 시퀀스를 검색하고 동일한 값(혹은 키 함수에서 변환한 값)에 대한 일련의 실행을 갖음
개별 순환에 대해서 값, 그리고 같은 값을 가진 그룹의 모든 아이템을 만드는 이터레이터를 함께 반환한다 ???

그에 앞서 원하는 필드에 따라 데이터를 정렬해야하는 과정이 중요. groupby()함수는 연속된 아이템에만 동작하기 때문에 정렬과정을 생략하면 원하는 대로 함수를 실행할 수 없음

단순히 날짜에 따라 데이터를 묶어서 커다란 자료 구조에 넣고 원할 때 마다 접근하려면 1.6 defaultdic()를 사용해서 multidict를 구성하는데 나음 (차이를 잘 모르겠음)

```
from collections import defaultdict
rows_by_date = defaultdict(list)
for row in rows:
        rows_by_date[row['date']].append(row)
        print(rows_by_date)

#defaultdict(<class 'list'>,
 {'07/01/2012': [{'address': '5412 N CLARK', 'date': '07/01/2012'}, {'address': '4801 N BROADWAY', 'date':
'07/01/2012'}],
'07/02/2012': [{'address': '5800 E 58TH', 'date': '07/02/2012'}, {'address': '5645 N RAVENSWOOD', 'date':
'07/02/2012'}, {'address': '1060 N ADDISON', 'date': '07/02/2012'}],
'07/03/2012': [{'address': '2122 N CLARK', 'date': '07/03/2012'}],
'07/04/2012': [{'address': '5148 N CLARK', 'date': '07/04/2012'}, {'address': '1039 W GRANVILLE', 'date':
'07/04/2012'}]})
```

```
for r in rows_by_date['07/01/2012']:
   print(r)

{'address': '5412 N CLARK', 'date': '07/01/2012'}
{'address': '4801 N BROADWAY', 'date': '07/01/2012'}
```

1.16 시퀀스 필터링

시퀀스 내부에 데이터가 있고 특정 조건에 따라 값을 추출하거나 줄이고 싶은 경우
리스트 컴프리헨션(list comprehension)

```
mylist = [ 1,4,-5,10,-7,2,3,-1]
[n for n in mylist if n > 0]  #>>> [1, 4, 10, 2, 3]
[n for n in mylist if n <0] #>>>[-5,-7,-1]
```

—> 입력된 내용이 매우 크면 매우 큰 결과가 생성될 수 있다는 단점이 있음
—> 생성자 표현식을 통해 값을 걸러낼 수 있음 (이게 왜 위 단점의 해결책이 되는 지 모르겠음)

```
pos = ( n for n in mylist if n > 0) # 이건 또 왜 제너레이터가 되는거여
pos #<generator object <genexpr> at 0x106188620>
for x in pos:
        print(x)
#1
#4
#10
#2
#3
```

리프리 컴프리헨션이나 생성자 표현식에 필터 조건을 만든느 것이 쉽지 않을 때도 있음 (필터링 도중에 예외처리를 하거나 다른 복잡한 내용이 들어가야 할 때)
—> 필터링 코드를 함수 안에 넣고 filter()를 사용하면 ok

```
values = ['1','2','-3','-','4','N/A','5']
def is_int(val):
        try:
                x = int(val)
                return True
        except ValueError:
                return False

ivals = list(filter(is_int,values))
print(ivals) #['1', '2', '-3', '4', '5']
```

filter()는 이터레이터를 생성함. 따라서 결과의 리스트를 만들고 싶으면 list()도 함께 사용해야함

리스트 컴프리헨션과 생성자 표현식은 간단한 데이터를 걸러내기 위한 가장 쉽고 직관적인 방법임. 또한 동시에 데이터 변형 기능도 갖고 있음

```
mylist = [1,4,5,10,-7,2,3,-1]
import math
[math.sqrt(n) for n in mylist if n>0]
#[1.0, 2.0, 2.23606797749979, 3.1622776601683795, 1.4142135623730951, 1.7320508075688772]
```

필터링에는 조건을 만족하지 않는 값을 걸러내는 것 외에도 새로운 값을 치환하는 방식도 있음
예를 들어 리스트에서 양수만 찾아내는 필터링뿐 아니라 잘못된 값을 특정 범위에 들어가도록 수정할 수 있음
필터링 조건을 조건 표현식으로 바꿔주면 간단히 해결 가능

```
clip_neg = [ n if n > 0 else 0 for n in mylist]
clip_neg #>>>[1, 4, 5, 10, 0, 2, 3, 0]
clip_pos = [n if n < 0 else 0 for n in mylist]
clip_pos #>>>[0, 0, 0, 0, -7, 0, 0, -1]
```

순환가능한 것과 Boolean 셀렉터 시퀀스를 입력으로 받는 itertools.compress()가 있음
—>그렇게 입력하면 셀렉터에서 조건이 참인 요소만 골라서 반환함
어떤 시퀀스의 필터링 결과를 다른 시퀀스에 반영하러 할 때 매우 유용함

```
addresses = [
        '5412 N CLARK',
        '5148 N CLARK',
        '5800 E 58TH',
        '2122 N CLARK',
        '5645 N RAVENSWOOD',
        '1060 W ADDISON',
        '4801 N BROADWAY',
        '1039 W GRANVILLE'
]
counts = [0,3,10,4,1,7,6,1]

from itertools import compress
more5 = [n > 5 for n in counts]
print( more5) #[False, False, True, False, False, True, True, False]
print( list(compress(addresses, more5)) ) #['5800 E 58TH', '1060 W ADDISON', '4801 N BROADWAY']
```

주어진 조건에 만족하는 여부를 담은 Boolean시퀀스를 만들어두는 것이 좋음
compress()함수로 True에 일치하는 값만 골라냄
filter()와 마찬가지로 compress()는 일반적으로 이터레이터를 반환. 따라서 결과를 리스트에 담고 싶다면 list()를 사용해야함

1.17 딕셔너리의 부분 추출

딕셔너리의 특정 부분으로부터 다른 딕셔너리를 만들고 싶음
딕셔너리 컴프리헨션(dictionary comprehension) 사용

```
prices = {
            'ACME' : 45.23,
            'AAPL' : 612.78,
            'IBM' : 205.55,
            'HPQ' : 37.20,
            'FB' : 10.75
}

#가격이 200이상인 것에 대한 딕셔너리
p1 = {key: value for key, value in prices.items() if value > 200}
print(p1) #{'AAPL': 612.78, 'IBM': 205.55}

#기술 관련 주식으로 딕셔너리 구성
tech_names = {'AAPL','IBM','HPQ','MSFT'}
p2 = {key: value for key, value in prices.items() if key in tech_names}
print(p2) #{'AAPL': 612.78, 'IBM': 205.55, 'HPQ': 37.2}
```

딕셔너리 컴프리헨션으로 할 수 있는 대부분의 일은 튜플 시퀀스를 만들고 dict()함수에 전달하는 것으로도 할 수 있음(다음 참고)
하지만 딕셔너리 컴프리헨션을 사용하는 것이 더 깔끔, 실행 속도 면에서도 유리

```
p1 = dict((key,value) for key, value in prices.items() if value > 200 )
print(p1) #{'AAPL': 612.78, 'IBM': 205.55}
```

```
tech_names = {'AAPL','IBM','HPQ','MSFT'}
p2 = {key : prices[key] for key in prices.keys() & tech_names}
```