Remember, if an argument is **iterable**, it means that you can call only **iter** on it, and then call **next** on the value **iter** returns (**for** loops do this automatically). There is no guarantee you can call **len** on the **iterable** or index/slice it. You **may not copy all the values** of an **iterable** into a **list** (or any other data structure) so that you can perform these operations (that is not in the spirit of the assignment, and some iterables could produce an infinite number of values, so such copying is impossible). You **may** create local data structures storing as many values as the arguments or the result that the function returns, but not all the values in the iterable.

1. (20 pts) Write generator functions below (each one is worth 4 points) that satisfy the following specifications. You **may not** import any of the generators in **itertools** or any other modules to write your generators. You **may** use any of the standard functions like **zip** and **enumerate**.

a. The **sequence** generator takes any number of **iterable**s as parameters: it produces every value from the first **iterable**, followed by every value from the second iterable, etc.,. Hint: I used only **for** loops. For example

```
for i in sequence('abc', 'd', 'ef', 'ghi'):
    print(i,end='')
```

prints **abcdefghi**: all characters from **'abc'**, followed by all in **'d'**, followed by all in **'ef'** etc..

b. The **group_when** generator takes one **iterable** and one predicate as parameters: it produces **list**s that each end in a value from the **iterable** where the predicate is **True**. If the **iterable** ends on a value for which the predicate returns **False**, **yield** a final list containing all the values from the one after the previous end to the last value produced by the **iterable**. Hint: I used a **for** loop . For example

```
for i in group_when('combustibles', lambda x : x in 'aeiou'):
    print(i,end='')
```

prints the 5 **lists** **['c', 'o']['m', 'b', 'u']['s', 't', 'i']['b', 'l', 'e']['s']**.

c. The **drop_last** generator takes one **iterable** and one **int** as a parameter (call it **n**): it produces every value from the **iterable** except for the last **n** values (without being able to count how many values the **iterable** produces) Hint: I used an explicit call to **iter** and a **while** loop and a comprehension that creates a **list** that stores at most **n** values (so that data structure is allowed here). For example

```
for i in drop_last('combustible', 5):
    print(i,end='')
```

prints **combus**; if the **iterable** produces < **n** values, it terminates immediately, on the first call to **next**.

d. The **yield_and_skip** generator takes one **iterable** and one function (which takes one argument and returns an **int**) as parameters: it produces a value from the **iterable** but then it then skips the number of values specified when the function argument is called on the just-produced value. Hint: I used an explicit call to **iter** and a **while** and **for** loop. For example

```
for i in yield_and_skip('abbabxcabbcaccabb',lambda x : {'a':1,'b':2,'c':3}.get(x,0)):
    print(i,end='')
```

prints **abxccab**; prints **a** then skips 1; prints **b** then skips 2; prints **x** then skips 0; prints **c** then skips 3; …

e. The **alternate_all** generator takes any number of **iterables** as parameters: it produces the first value from the first parameter, then the first value from the second parameter, ..., then the first value from the last parameter; then the second value from the first parameter, then the second value from the second parameter, ..., then the second value from the last parameter; etc. If any **iterable** produces no more values, it is ignored. Eventually, this generator produces every value in each **iterable**. Hint: I used explicit calls to **iter**, and a **while** and **for** loop, and a **try**/**except** statement; you can create a **list** whose length is the same as the number of parameters (I stored **iter** called on each parameter in such a list). For example

```
for i in alternate_all('abcde','fg','hijk'):
    print(i,end='')
```

prints **afhbgicjdke**.

2. (5 pts) Define a generator named **min_key_order** whose parameter is a **dict** that has keys that can all be compared with each other (e.g., all numbers, all strings, etc; don't check this property). The **min_key_order** generator produces a **2-tuple** of the **dict**'s **keys** and **values**, in increasing order of the **keys**. For example:

```
for i in min_key_order ({1:'a', 2:'x', 4:'m', 8:'d', 16:'f'}):
   print(i,end='')
```

prints: `(1,'a')(2,'x')(4,'m')(8,'d')(16,'f')`

It would be trivial to write this generator as

```
def min_key_order (adict):
    for k,v in sorted(adict.items()):    # iterate over a sorted list of keys/values
        yield (k,v)
```

but the generator must have one more property: it must iterate over mutations made to the **dict** while the **dict** is being iterated over. That is

```
d = ({1:'a', 2:'x', 4:'m', 8: 'd'}
i = min_keys_order(d)
print(next(i))
print(next(i))
del d[8]
print(next(i))
d[16] = 'f'
d[32] = 'z'
print(next(i))
print(next(i))
```

```
Prints:
(1, 'a')
(2, 'x')
(4, 'm')
(16, 'f')
(32, 'z')
```

Note that the process of iterating through the **dict** should not mutate the **dict** (although the **dict** can be mutated directly by code during the iteration, as is shown above),

Hints: The first time **next** is called, find the minimum key in the **dict** (it is unique, if the **dict** isn't empty) and **yield** it and the key and its value first; later, repeatedly find the smallest key bigger than the previously **yield**ed key (so long as there is one) and **yield** the key and its value; you may create a local **list**/**tuple** whose length is no bigger than the **dict** (although there is a way to solve this problem without such a data structure, but it requires more code).