When working on this quiz, recall the rules stated on the Academic Integrity statement that you signed. You can download the **q4helper** project folder (available for Friday, on the **Weekly Schedule** link) in which to write/test/debug your code. Submit your completed **q4solution** module online by **Wednesday**, 11:30pm. I will post my solutions to EEE reachable via the **Solutions** link on **Wednesday** evening right at 11:30pm.

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. In fact, your code must work on infinite iterables (see the **primes** and **nth** generators, which work together for testing).

1. (20 pts) Write generators below (a.-e. worth 3 pts each, f. worth 5 pts) that satisfy the following specifications. You **may not** use any of the generators in **itertools** to help write your generators.

a. The **differences** generator takes two **iterables** as parameters: it produces a 3-**tuple** for every pairwise difference in values produced by the **iterable**s, showing the index (assume the index of the first value in each **iterable** is **1**) and the different values in each **iterable**. For example

```
for i in differences('3.14159265', '3x14129285'):
    print(i,end='')
```

prints `(2, '.', 'x') (6, '5', '2') (9, '6', '8')`; the values in indexes **2**, **6**, and **9** are different. Hint: use a **for** loop controlled by a combination of **zip** and **enumerate**.

b. The **once_in_a_row** generator takes one **iterable** as a parameter: it produces every value from the **iterable**, but it never produces the same value twice in a row. For example

```
for i in once_in_a_row('abcccaaabddeee'):
    print(i,end='')
```

prints **abcabde**: if there is a sequence of the same values, one following the other, only one is produced.

c. The **in_between** generator takes an **iterable** as and two predicates (call them **start** and **stop**) as parameters: it produces every value **v** in the iterable that lie between values where **start(v)** returns **True** and **stop(v)** returns **True** (inclusive to these values). For example

```
for i in in_between('123abczdefalmanozstuzavuwz45z',
                    (lambda x : x == 'a'),
                    (lambda x : x == 'z')):
    print(i,end='')
```

prints **abczalmanozavuwz**.

d. The **pick** generator takes an **iterable** and an **int** (call it **n**) as parameters: it produces **list**s of **n** values: the first **list** contains the first **n** values from the **iterable**; the second **list** contains the second **n** values from the **iterable**, etc. until there are fewer than **n** values left to put in the returned **list**. For example

```
for i in pick('abcdefghijklm',4):
    print(i,end='')
```

prints `['a','b','c','d'] ['e','f','g','h'] ['i','j','k','l']`. Hint: I called **iter** and **next** directly, building a **list** with $\leq$ **n** values, so it doesn't violate the conditions for using **iterable**s.

e. The **slice_gen** generator takes one **iterable** and a **start**, **stop** and **step** values (all **int**, with the same meanings as the values in a slice: **[start:stop:step]**, except **start** and **stop** must be non-negative and **step** must be positive; raise an **AssertionError** exception if any is not). It produces all the values in what would be the slice (without every putting all the values in a **list** and slicing it). For example

```
for i in slice_gen('abcdefghijk', 3,7,1):
        print(i,end='')
```

prints the 4 values: **'d'**, **'e'**, **'f'**, and **'g'**: the 3rd, 4th , 5th, and 6th values (start counting at the 0th value).

Hint: you may use the **range** class. Even if the **iterable** is infinite, this generator/decorator should work and produce a finite number of values.

f. 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**.

2a. (4 pts) Lists, sets, and dictionaries are all mutable. Although we can execute code that changes the length of a **list** while it is being iterated over (by calling **append** or **del**), Python does not allow us to change the lengths of a **set** or **dict** while either is being iterated over. When Python runs the following code

```
s = set([1])
for i in s:
    s.add(2)     # this line raises: RuntimeError: Set changed size during iteration
```

Why does Python allow us to change **list** sizes during iteration but not **set**/**dict** sizes? The rationale relates to the fact that we know what order the values in a **list** are iterated over, but not a **set**/**dict**. Because we know this order for a **list** we can determine how values added or deleted will be treated in the iteration; but we cannot determine this information for a **set** or **dict**, so mutating it is prohibited

Write a **list**-like class (implementing some, but not all, list operations) named **ListSI** (SI stand for Special Iterator) that raises a **RuntimeError** exception when executing code that attempts to change the length of a **list** (with **append** and **del**) while the **list** is being iterated over, mostly think iteration by a **for** loop, but also if **__iter__** is called explicitly: **i = iter(aListSI),print(next(i))**, … Iterating over a **list** should leave the list unchanged.

- Objects constructed from the **ListSI** class store 2 attributes: a real list (**_real_list**) and a count of the number of (nested) iterations in which this **ListSI** is being iterated on (**_iter_count**). This **__init__** method, and other dunder methods in the **ListSI** class is already written.
- The **append** and **__delitem__** methods must raise the **RuntimeError** exception (see the **import**) from **builtins**, when they are called while the **ListSI** is being iterated over.
- Inside the **ListSI_iter** class, implement **__init__** and **__next__** so that they cooperate to implement a list iterator (producing the values stored at index **0**, then index **1**, … in the **ListSI**) corretly, while also incrementing/decrementing **_iter_count** as new iterations are started and stopped. At any given time, **iter_count** is the number of nested iterations of the **ListSI** object.

2b. (1 ps) Fill in the body of the **for** loop in the **fool_it** function with code that might normally be found in a loop, so that executing it causes the final **append** to raise a **RuntimeError** exception when it shouldn't: that **append** is outside of any iterations. The code should not examine/manipulate any **ListSI** object.