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 Thursday, 11:30pm. I will post my solutions to EEE reachable via the **Solutions** link on Friday morning.

Remember, if an argument is `iterable`, it means that you can call only `iter` on it, and you can call `next` on the value `iter` returns. There is no guarantee you can call `len` on the iterable or index it. You cannot **copy all the values** of an `iterable` into a `list` so that you can perform these operations.

1. (20 pts) Write generators 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 functions like `zip` and `enumerate`.

a. The `start_when` generator takes an `iterable` and a predicate (a function of one argument that returns a `bool`) as parameters: it produces every value from the `iterable`, starting with the first value for which the predicate returns `True` (later values are not tested). Hint: you can use a `for` or `while` loop. For example

```
for i in start_when('combustible', lambda x : x >= 'q'):
    print(i,end='')
```

prints `ustible`: `u` is the first character greater than `q`, so it doesn't produce the letters `c`, `o`, `m`, and `b`.

b. The `differences` generator takes two `iterables` as parameters: it produces a 3-`tuple` for every difference in value in the `iterable`s, showing the index (the index of the first value in each `iterable` is `1`) and the different values in each `iterable`. Hint: use a `for` loop with `zip` and `enumerate`. For example

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

prints `(2, '.', 'x') (6, '5', '2') (9, '6', '8')`; on the values in indexes `2`, `6`, and `9` are different.

c. 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. Hint: use a `for` loop, remembering the last value produced (how do you handle this to always produce the first value?). For example

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

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

d. The `alternate` 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, this generator produces no more values.. For example

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

prints `afhbgic`. Hint: I called `iter` and `next` directly, using a `list` that I iterated over with a `for` loop inside a `while` loop. This `list` that I created just has one value for each `iterable`; it doesn't store all the values the `iterable`s produce, so it doesn't violate the conditions for using `iterable`s.

e. The `windows` generator takes one `iterable` and two `int`s (call them `m` and `n`; `m`'s default value is `1`) as parameters: it produces `list`s of `n` values: the first `list` contains the first `n` values; every subsequent `list` drops the first **m** from the previous **list** and adds the next **m** values from the **iterable**, until there are fewer than **n** values to put in the returned **list**. For example

```
for i in windows('abcdefghijk', 4,2):
    print(i,end='')
```

prints **['a','b','c','d'] ['c','d','e','f'] ['e','f','g','h'] ['g','h','i','j']**. Hint: I called **iter** and **next** directly, and used a **list** that always contains **n** values, so it doesn't violate the conditions for using **iterable**s

2. (5 pts) Write a function named **ascending**, which is passed an integer argument (call it **n**) and an **iterable** argument whose values are all comparable (e.g., they could be all **int** or all **str**). This function returns a **list** of 2-**tuples**, whose values are the first and last values in a sequence that ascends (gets strictly bigger) for **n** or more numbers: for example, **ascending(3,[5,5,2,4,6,2,4])** returns **[(2,6)]** and **ascending(3,[2,3,1,4,6,7,2,0,2,4,3])** returns **[(1,7), (0, 4)]**. Raise an **AssertionError** if **n** is not at least **2**

You **must** use a **while** loop, and explicit calls to **iter** and **next**; the only data structure that you can create is the **list** that is returned; you cannot create any intermediate data structures to help the computation: e.g., you cannot create a **list** with all the values in the **iterable** to process it more easily.

Hint: This function is intricate. I used the following variables: **start** stores the start of each possible ascending sequence, **prev** and **curr** are the previous and current value gotten from the iterable, and **up** counts how many values have been increasing since **start** was set. Comparing **prev** to **curr** determines whether the current ascending sequence is longer (just increase **up**) or that the current ascending sequence has ended: if ended, we check **up** to determine whether to put **start** and **prev** (the first and last value in the ascending sequence) into the answer **list**, and then reset **start** and **up** for a new possible ascending sequence. My function body was 20 lines long. It included one **try**/**except** statement, one **while** loop, and three **if** statements (one in the **except** clause).