Quiz #4: Iteration Protocol, Generators, and Iterable Decorators ICS-33 Spring 2017

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 then call **next** on the value **iter** returns (recall **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). 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(s): in fact, some will be infinite. Remember the "exchange **print** vs. **yield**" heuristic for writing generators from the notes.

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')**; on 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. Hint: use a **for** loop, remembering the last value produced (the first value is always 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 **group** 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 gropu('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 ≤ `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 `AssetionError` 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 and its `in` operator. Even if the `iterable` is infinite, this generator decorator should work and produce a finite number of values.

f. The `shuffle` 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`. For example

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

prints `afhbgicjdke`. 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 no bigger than the number of parameters (I stored `iter` called on each parameter in such a `list`). So if `shuffle` is called with `5` iterators, you can create a list whose length is `5`.

2. (5 pts) The `Backwardable` class decorates iterables (and is itself iterable), allowing calls to both the `next` and `prev` functions: `next` is standard and defined in `builtins`; I've defined a similar `prev` function in `q4solution.py` (to call the `__prev__` method defined in `Backwardable`'s `B_iter` class). So, we can move both forward and backward in the iterable that `Backwardable` class decorates. This problem uses only classes, not generators.

For example, given `i = iter(Backwardable('abc'))` then we could call both `next(i)` and `prev(i)` to iterate over the string. The sequence of calls on the left would produce the values on the right (the far-right is `print(i)`).

Executes	**Prints**	**What `print(i)` would print (see below) after the call**
Before execution		`_all=[], _index=-1`
`next(i)`	`'a'`	`_all=['a'], _index=0`
`next(i)`	`'b'`	`_all=['a', 'b'], _index=1`
`prev(i)`	`'a'`	`_all=['a', 'b'], _index=0`
`#prev(i)`	would raise `AssertionError` exception	
`next(i)`	`'b'`	`_all=['a', 'b'], _index=1`
`next(i)`	`'c'`	`_all=['a', 'b', 'c'], _index=2`
`prev(i)`	`'b'`	`_all=['a', 'b', 'c'], _index=1`
`next(i)`	`'c'`	`_all=['a', 'b', 'c'], _index=2`
`next(i)`	raises `StopIteration` exception	

`Backwardable` takes any `iterable` as an argument. As with other classes decorating iterators, it defines only the `__init__` and `__iter__` methods, with `__iter__` defining its own special `B_iter` class for actually doing the iteration. I've supplied the `__init__` and `__str__` for this class; do not change these. You must write only the `__next__` and `__prev__`methods. Here is a brief description of the attributes defined in `__init__` do.

- The `_all` attribute stores a `list` remembering all the values returned via `__next__`, so we can go backwards and forwards through the values already produced by `Backwardable`'s iterable argument.
- The `_iterator` attribute can be passed to `next` to produce a new value or raise `StopIteration`.
- The `_index` stores the index in `_all` of the **value most recently returned** from a call of the `__next__` or `__prev__` methods. It typically is incremented/decremented in calls to `__next__` or `__prev__`.

You must write the code in `__next__` and `__prev__` that coordinates these attributes to produce the behavior illustrated in the example above.

How does `__next__` work? Depending on value of `_index` and the length of `_all`, it might just return a value from `_all`; but if _index is at the end of the `list`, `__next__`will need to call `next` on `_iterator` to get a new one to return (while also appending this new value at the end of `_all`). Ultimately `_index` must be updated as appropriate.

How does `__prev__` work? It just returns a value from the `_all list`; but it raises the `AssertionError` exception if an attempt is made to get a value previous to the first value produced by the iterable.