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. Recall that the midterm will be moved back to 2/8, so the due date for this quiz is now officially moved back to 2/3.

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 generator functions 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. Remember that if a generator function explicitly calls **next** and in doing so **next** raises a **StopIteration** exception, the generator function code must handle it and then execute a **return** statement (which Python will translate into raising a **StopIteration** exception). A generator function should not itself raise a **StopIteration** exception.

a. The **running_count** generator takes an **iterable** and a predicate function (taking one argument and returning a **bool** value) as parameters: it produces a running count of how many values up to (and including) the one just iterated over, satisfied the predicate. For example

```
for i in running_count('bananastand', lambda x : x in 'aeiou'):
    print(i,end=' ')
```

prints the values **0 1 1 2 2 3 3 3 4 4 4**.

b. The **stop_when** generator takes an **iterable** and a predicate as parameters: it produces every value from the iterable, but stopping (and not producing) the first value for which the predicate returns **True**. For example

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

prints **comb**: **u** is the first character greater than **q**, so it is not included and the generator stops.

c. The **yield_and_skip** generator takes an **iterable** as a parameter: it produces every value in the iterable except if it produces an **int n**, it produces that value but then skips the next **n** values.

```
for i in yield_and_skip([1,2,1,3,'a','b',2,5,'c',1,2,3,8,'x','y','z',2]):
    print(i,end=' ')
```

prints **1 1 a b 2 1 3 z 2**. Hint: I called **iter** and **next** directly, using a **while** and **for** loop.

d. The **windows** generator takes an **iterable** and two **int**s (call them **n** and **m**; with **m**'s default value **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: You can use a single `for` loop or call `iter` and `next` directly, using a `list` that always contains `n` values, so it doesn't violate the conditions for using `iterable`s.

e. 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, `alternate` produces no more values. For example

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

prints `afhbgic`. You **may not use** a call to `zip` function. Hint: You can call `iter` and `next` directly, using a `while` and `for` loop. You can create a `list` whose length is the number of arguments: 3 for the call to `alternate` above, but no bigger.

f. The `myzip` generator takes any number of `iterables` (call the number `n`) it produces `n-tuples` of the first value in each iterable, the second value of each iterable, etc; if one iterable is exhausted before all the others, its position in the `n-tuple` should store `None`. For example

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

prints the 5 `3-tuples ('a','f','h')('b','g','i')('c',None,'j')('d',None,'k')('e',None, None)`.

You **may not** use the `zip` function. Hint: You can call `iter` and `next` directly, using a `while` and `for` loop. You can use a flag or a counter to remember information about `next` raising (or failing to raise) exceptions. You can create `lists/tuples` whose length is the number of arguments:3 for the call to `myzip` above.

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 by having it remember a `list` of previous values. In addition, the `clear` function can be called to forget permanently earlier values that are unneeded (saving space). Use only classes, not generator functions.

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)`). See a longer example (also using `clear`) in the `q4solution.py` file.

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 written **__init__** and **__str__** for this class; do not change these. You write only the **__next__**, **__prev__**, and **__clear__** methods. Here is a brief description of the attributes defined in **__init__**.

- 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__**, **__prev__** , and **__clear__** 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; or the first value after **clear** is called, which resets which value is "first" (see below).

How does **__clear__** work? It resets the attributes, forgetting any previous values produced (but remembering the current one, if there is one, and any later ones); with this method, if we are done looking at the earlier part of an iterator with many values, we can forget those values and reclaim the **list** space storing them.