Homework 6 Instructions

Included with this instruction file is the file `HW6.py`, containing the skeleton code for the homework.

Please read and follow all the directions carefully!

Put your name in the appropriate comment at the top of the program file `HW6.py`. Turn in a zipped file as described in the *About Homework* document.

Note: None of these exercises should print anything! Also, you should not need to import anything other than from `itertools`. If you think you need to, email me to tell me what and why and I can tell you why not. There is one version of a function that, in some versions of an implementation, might need something from the `collections` module, but it's not a likely option. And, as always, you can assume that you will get valid input, so you do not need to have any try/except blocks or other validation code.

Review Exercises

Exercise 1: separate (10 pts)

Edit the function `separate`, that takes a string as input and returns a list containing the individual characters of the string. Return all the characters lowercased and do not remove duplicates. Add an optional keyword argument `sort`, with a default of `False`. If `sort` is `True`, will return the characters in "ASCII-betical" sorted order. Namely, sorted by their ASCII character definitions, which is pretty much what you would expect.

Note: You have to make your own argument list for this one! If you think you need `*args` and/or `**kwargs`, think again!

Hints: Remember that strings are iterables. Consider the built-in function `sorted`, or the list method `sort`.

```
>>> from HW6 import separate
>>> separate("hello")
['h', 'e', 'l', 'l', 'o']
>>> separate("hello", sort=True)
['e', 'h', 'l', 'l', 'o']
>>> separate("hello", sort=False)
['h', 'e', 'l', 'l', 'o']
>>> separate("A LONGER string")
['a', ' ', 'l', 'o', 'n', 'g', 'e', 'r', ' ', 's', 't', 'r', 'i', 'n', 'g']
>>> separate("A LONGER string", sort=True)
[' ', ' ', 'a', 'e', 'g', 'g', 'i', 'l', 'n', 'n', 'o', 'r', 'r', 's', 't']
```

Exercise 2: number_of_vowels (10 pts)

Edit the function `number_of_vowels` so that it returns the number of vowel letters in the input string, ignoring the case of the letters in the string. Note we do not count `y` as a vowel.

```
>>> from HW6 import number_of_vowels
>>> number_of_vowels("It's a Lovely Day!") == 5
True
>>> number_of_vowels("HOW MANY VOWELS ARE HERE?")
```

```
8
```

Generator & Iterator Exercises

Exercise 3: special_nums (10 pts)

Edit the function `special_nums` so that it returns a **generator** that *yields*, in order, the numbers from 1 through (and including) 300 that are divisible by both 6 and 10. **Use a list comprehension or generator expression. Do not use built-in iter() function**

```
>>> from HW6 import special_nums
>>> nums = special_nums()
>>> iter(nums) == nums
True
>>> next(nums) == 30
True
>>> next(nums) == 60
True
>>> next(nums)
90
>>> next(nums)
120
>>> next(nums)
150
>>> next(nums)
180
>>> next(nums)
210
>>> next(nums)
240
>>> next(nums)
270
>>> next(nums)
300
>>> next(nums)
Traceback (most recent call last):
  [...]
StopIteration
```

Exercise 4: evens (10 pts)

Edit the function `evens` so that it returns a **generator** that *yields* the numbers in the input sequence that are even, in order.

Do not use built-in iter() function or create any new lists.

It should work like this:

```
>>> from HW6 import evens
>>> numbers = [89, 32, 4, 35, 22, 8, 14, 3]
>>> even_numbers = evens(numbers)
>>> even_numbers
<generator object evens.<locals>.<genexpr> at 0x10d2c29d0>
>>> iter(even_numbers) == even_numbers
True
>>> next(even_numbers) == 32
True
>>> next(even_numbers) == 4
True
>>> next(even_numbers) == 22
True
```

```
>>> next(even_numbers)
8
>>> next(even_numbers)
14
>>> next(even_numbers)
Traceback (most recent call last):
  [...]
StopIteration
```

One characteristic of generators is that if there is an error part-way through, you still get the values from before the error. This may or may not be a good thing... If we used a list comprehension to make a list of the even elements (and there was a bad element), we would not get the list because the error would happen before the list was complete. Here is an example where the last item in the list is another list, which is invalid for the modulo function. If we had written `evens` with a list comprehension first, then we would get the error immediately after the call to `evens`.

Note: I'm just showing this to you because it is interesting. You do not need to worry about checking bad input in the sequence.

```
>>> bad_numbers = [2, 3, 4, 5, 6, [2]]
>>> bad_nums = evens(bad_numbers)
>>> next(bad_nums)
2
>>> next(bad_nums)
4
>>> next(bad_nums)
6
>>> next(bad_nums)
Traceback (most recent call last):
  [...]
TypeError: unsupported operand type(s) for %: 'list' and 'int'
```

Exercise 5: continuous1234 (10 pts)

Edit the function `continuous1234` so that it returns an inexhaustible (never-ending) **generator/iterator** that *returns* the numbers 1, 2, 3, 4 forever.

Absolute Requirement: Use something from itertools. This is a test of your ability to read and understand documentation. ;-)

```
>>> from HW6 import continuous1234
>>> n1234 = continuous1234()
>>> iter(n1234) == n1234
True
>>> next(n1234) == 1
True
>>> next(n1234) == 2
True
>>> next(n1234) == 3
True
>>> next(n1234) == 4
True
>>> next(n1234) == 1
True
>>> next(n1234) == 2
True
>>> next(n1234) == 3
True
>>> next(n1234) == 4
True
>>> next(n1234) == 1
True
```

```
>>> next(n1234) == 2
True
[...etc... forever...]
```

Exercise 6: reverse_iter (20 pts)

Edit the function `reverse_iter` so that it returns a **generator** that *yields* the items given in the input sequence in reverse order. Think about what we learned in week 3, and also what we learned in week 2 about how to reverse a sequence. Note that after using `reverse_iter`, the original sequence should be unchanged.

Do not create any new lists.

Don't use built-in functions or methods like `reverse()`, `reversed()`, `iter()`. **Please note that this should also work when the input is a tuple (eg, if `nums = (1, 2, 3, 4)` in the example below), because tuples are sequences.**

```
>>> from HW6 import reverse_iter
>>> nums = [8, 3, 6]
>>> it = reverse_iter(nums)
>>> next(it) == 6
True
>>> next(it)
3
>>> next(it)
8
>>> next(it)
Traceback (most recent call last):
  [...]
StopIteration
>>> nums
[8, 3, 6]
>>> items = ['a', 'b', 'c']
>>> it = reverse_iter(items)
>>> iter(it) is it
True
>>> next(it)
'c'
>>> next(it)
'b'
>>> next(it)
'a'
>>> next(it)
Traceback (most recent call last):
  [...]
StopIteration
```

Exercise 7: ReverseIter class (30 pts)

Create an *iterator* class `ReverseIter` that takes an input sequence and creates an instance that is an *iterator* that iterates over the input sequence in reverse order. (When done, you will appreciate generator functions as in problem 6).

Note that after using `ReverseIter`, the original sequence should be unchanged. A `ReverseIter` instance returns an iterator of the input sequence, which will return the elements of the sequence in reverse order, raising a StopIteration error when the reversed sequence is exhausted.

Remember that this is an *iterator*, and iterators are single-use *iterables*. When the instance is exhausted (finished iterating over the sequence), you need to raise your own `StopIteration` Error when there are no more items. Subsequent calls to next() should continue to generate a `StopIteration` Error, because iterators are single-use iterables. There should be no try/except blocks in the class definition.

Don't use built-in functions or methods like `reverse()`, `reversed()`, `iter()`. You may use the `len()` function. Please note that this should also work when the input is a tuple (eg, if `nums = (1, 2, 3, 4)` in the example below).

The class should not print anything!

```
>>> from HW6 import ReverseIter
>>> nums = [8, 3, 6]
>>> it = ReverseIter(nums)
>>> iter(it) is it
True
>>> next(it) == 6
True
>>> next(it)
3
>>> next(it)
8
>>> next(it)
Traceback (most recent call last):
  [...]
StopIteration
>>> nums
[8, 3, 6]
>>> items = ['a', 'b', 'c']
>>> it = ReverseIter(items)
>>> next(it) == 'c'
True
>>> iter(it) == it
True
>>> next(it)
'b'
>>> next(it)
'a'
>>> next(it)
Traceback (most recent call last):
  [...]
StopIteration
>>> items
['a', 'b', 'c']
>>> next(it)  # Make sure it stays finished.
Traceback (most recent call last):
  [...]
StopIteration
```

Note. Remember that from the point of view of the code using them, the behavior of a generator and an iterator appear the same. They are both single-use iterators; the difference is internal to the code.

My email is dianechen.ucsdext@gmail.com. Please do not hesitate to email me if you have questions.