Generator

This five part assignment offers practice iterating over lists, ranges, and strings using for and while loops as well as using a generator of random integers. It's also a good exercise in writing unittest test cases.

Part 1:

Write a function count_vowels(s) that takes a string as an argument and returns the number of vowels ('a', 'e', 'i' 'o', 'u'') in the string. Should you use a *for* or *while* loop? (Implement this as a function, not as a class with a method.)

Be sure to include unittest test cases to demonstrate that your code works properly, e.g.

```
def count_vowels(s):
    return 0  # Your definition goes here

class CountVowelsTest(unitttest.TestCase):
    def test_count_vowels(self):
        self.assertEqual(count_vowels('hello world'), 3)
        self.assertEqual(count_vowels('hEllO wrld'), 2)
        self.assertEqual(count_vowels('hll wrld'), 0)
```

Hint: Python offers an 'in' operator that evaluates to True or False, e.g.

```
    "a" in "aeiou"
```

evaluates to True.

Hint: Strings can easily be converted to all lower case with the string.lower() method, e.g. "HeLlO wOrLd".lower() == "hello world"

Part 2:

Write a function that takes two arguments: (1) a target item to find, and (2) a list. Your function should return the index (offset from 0) of the ***last*** occurrence of the target item or **None** if the target is not found. E.g. the last occurrence of 33 is at offset 3 in the list [42, 33, 21, 33] because 42 is offset 0, the first 33 is at offset 1, 21 is offset 2, and the last 33 is offset 3.

Use unittest to test your function with several different target values and lists.

Next, test your function with a character as the target, and a string as the list, e.g. find('p', 'apple'). What should happen?

Be sure to use unittest to demonstrate that your code works properly.

Part 3:

The fractions in Homework03 are correct, but not simplified, e.g. 2/4 can be simplified to 1/2, 14/21 can be simplified to 2/3, and 63/9 can be simplified to 7/1. Recall from elementary school that you can simplify fractions by finding the Greatest Common Factor (GCF) and then dividing the number and denominator. Here's pseudocode for one solution:

```
  start = the min of abs(numerator) and abs(denominator)
       # the absolute value is important for negative values
  for each integer gcf from start down to 2
    if numerator mod gcf == 0 and denominator mod gcf == 0 then
       gcf is the greatest common factor that evenly divides both the
         numerator and denominator
          return a new Fraction with numerator / gcf and denominator / gcf
   if you don't find a gcf, then return a copy of the Fraction because it can't be
simplified
```

Extend your Fractions class from HW03 and add a new simplify(self) method that returns a new Fraction that is simplified or just returns a copy of self if self can't be simplified. E.g.

str(Fraction(9, 27).simplify()) == str(Fraction(1, 3))

Hint: Note that testing

Fraction(9, 27).simplify() == Fraction(1, 3)

is not sufficient because

Fraction(9, 27) == Fraction(1, 3)

Add unittest cases to test your new method.

Hint: what happens if the fraction has a negative numerator or denominator?

Hint: 8 / 4 == 2.0 so you may want to do int(8/4) before creating a new Fraction.

Part 4:

Recall that Python's built-in enumerate(seq) function is a generator that returns two values on each call to next(): the offset of the value and the value. E.g.

```
for offset, value in enumerate("hi!"):
    print(offset, value)
```

generates the output:

```
0 h

1 i

2 !
```

Write a function, my_enumerate(seq) that provides the same functionality WITHOUT calling enumerate(). Be sure to include an automated test to validate your solution.

Hint: My solution has only 5 lines of code.

Hint: The hardest part may be the automated test. My automated test uses a list to collect the result of my_enumerate() and compares that to the expected output.

Part 5: (OPTIONAL - NOT REQUIRED but interesting challenge)

(a) Write a generator that returns a potentially infinite sequence of random integers between a min and max value. E.g. say that min = 0 and max = 10, then the generator returns a sequence of random integers between 0 and 10 inclusive, one on each call to the generator function.

Hint: The following code generates a random number between 0 and 10 inclusive.

import random

r = random.randint(0, 10) # return a random integer between 0 and 10 inclusive.

(b) Use the random number generator from (a) in a function,

find_target(target, min_value, max_value, max_attempts)

where find_target() passes min_value and max_value to the random integer generator and then loops, reading random values from the random integer generator until the specified target is found, then returns how many random integers were read before finding the target or None if the target isn't found in max_attempts tries.

E.g. consider the call find_target(3, 0, 10, 100) where we're asking how many random integers between 0 and 10 are generated when we find the first occurrence of 3. Say the random integer generator generated the sequence [9, 4, 1, 3, 2, ...] . The generator returns a single value at a time and find_target() continues asking for more values until the target, 3, is found and then returns 4 because 4 random integers were generated when the target was found.

Note that the random generator may randomly choose values that never match the target value. The max_attempts parameter to find_target() is an escape hatch to force find_target() to return if the target value is not found to avoid a potential infinite loop.

Note: while you should pass min_value and max_value from find_target() when creating the generator, DO NOT include the max_attempts logic in the generator. The generator should return a potentially infinite number of random values.

One of the challenges of this assignment is how to test find_target() automatically since it relies on a random number generator. One technique is to create a generator that can generate only a single random value which then must be returned on the first call to the generator. E.g.

find_target(3, 3, 3, 1)

must return 1 because the random generator must generate a random number n, where 3 <= n <= 3, i.e. n == 3. Since the random number generator must return a sequence of the value 3, then find_target(3, 3, 3, 1) will find the target value of 3 on the first attempt.

You should also raise ValueError exceptions if there are any problems with the relationship between target, min_value, and max_value, e.g.

min_value <= target <= max_value

You might find that writing your code, and then stepping through your solution with the debugger and variable explorer in VS Code may help you to understand what your program is doing correctly (or not).

Include all of your functions in a single file that includes your functions and unittest tests for each of your functions.

Upload your Python program to Canvas. Please be sure to include **your name** in the file name, e.g. HW04-Rowland.py

As always, please email questions to me.