Gold hw2pr2: The sleepwalking student

Copied from: https://www.cs.hmc.edu/twiki/bin/view/CS5/SleepwalkingStudentGold on 3/22/2017

[35 points; individual or pair] filename: `hw2pr2.py`

For hw2pr2.py you will write Python functions to investigate the behavior of a sleepwalking student, a.k.a., a "random walk."

You should place your functions in a file named `hw2pr2.py`.

Part 1: Copy this function: `rs()`

Start your file with this header and function, named `rs()`

```
# CS5 Gold, hw2pr2
# Filename: hw2pr2.py
# Name:
# Problem description: Sleepwalking student

import random

def rs():
    """ rs chooses a random step and returns it
        note that a call to rs() requires parentheses
        inputs: none at all!
    """
    return random.choice([-1,1])
```

You can call `rs()` function whenever you want to obtain a new, random step: either `1` or `-1`.

An advantage of this is that it is easy to change what is meant by "random step" in the future, without changing any other code!

`import` vs `from ... import *`

If you use the library-import statement

```
import random
```

you can use the `random` library, but you will need to preface each call with the library's name:

```
random.choice( [-1,1] )
```

Another way to import libraries is to use

```
from random import *
```

In this case, you can simply type `choice( [-1,1] )` , which is a bit shorter.

A problem could arise if you had *another* function named `choice` already. For the moment, that isn't a concern.

A reminder on *string multiplication*

For this problem, string *multiplication* is very useful. Here is a reminder:

```
In [1]: print('spam'*3)

spamspamspam

In [2]: print('start|' + '_'*10 + '|end')

start|__________|end
```

In this latter example, `'_'*10` specified how much space to place between `'start|'` and `'|end'`.

It's nice to use underscores (or some other "ocean") in which your sleepwalker will venture!

Part 2: Write: `rwpos( start, nsteps )`

Next, write a function named `rwpos( start, nsteps )` which takes two inputs:

- an integer `start`, representing the starting position of our sleepwalker, and

- a nonnegative integer `nsteps`, representing the number of random steps to take from this starting position.

The name, `rwpos` is a reminder that this function should return the **r**andom **w**alker's **pos**ition.

Write `rwpos` so that it returns the **position** of the sleepwalker after `nsteps` random steps, where each step moves according to `rs()`, which means either plus `1` or minus `1` from the previous position.

Example `random` code from class...

Here is some of the `random` number-guessing code from class, if you'd like to use it as a starting point...

Printing/debugging code to include

As part of your rwpos function, include a line of debugging code that prints what `start` is each time the function is called. Include the string `start is`, too, as in the examples below.

Remember that, because each step is random, the exact values your function produces will likely be different than these, though the overall behavior should be the same:

```
In [1]: rwpos( 40, 4 )
start is 40
start is 41
start is 42
start is 41
start is 42
Out[1]: 42

In [2]: rwpos( 40, 4 )    # won't be the same each time...
start is 40
start is 39
start is 38
start is 37
start is 36
Out[2]: 36
```

Is it 4 or 5 printed lines?

You may have four lines of output instead of five -- this most likely depends on whether or not you print when the base case is hit. Either way is completely fine for this problem.

No loops!

Even if you've used `while` or `for` loops in the past, for this problem we ask you to **use recursion**.

These assignments are primarily to develop *design* skills -- specifically, recursive design. Don't worry -- there will be plenty of loops later in the term.

Part 3: Write `rwsteps(start, low, hi)`

Next, write `rwsteps( start, low, hi )` which takes three inputs:

- an integer `start`, representing the starting position of our sleepwalker,
- an integer `low`, which will always be nonnegative, representing the smallest value our sleepwalker will be allowed to wander to, and
- an integer `hi`, representing the highest value our sleepwalker will be allowed to wander to.

You may assume that `hi >= start >= low`.

What should `rwsteps` do? It should simulate a random walk, printing each step (see below). Also, as soon as the sleepwalker reaches *at or beyond* the `low` or `hi` value, the random walk should stop. When it does stop, `rwsteps` must return the **number of steps** that the sleepwalker took in order to finally reach the lower or upper bound.

Printing/debugging code: In `rwsteps` include a line of debugging code that prints a visual representation of your sleepwalker's position while wandering!

Feel free to be more creative than a simple `'S'` character. For example, consider `0->-<` (a true sleepwalker!)

As an extra-credit challenge (a fun one), you might create a more elaborate sleepwalker simulation that changes its looks depending on which direction it's heading (eyes looking left or right?). Or, it could interact with some other items/people/things on its path - see the extra-credit, below.

Examples Here are two plain-wandering examples, one using spaces and one using the underscore character (making it easier to see what's going on than with spaces!). One has walls on either side and one does not. The specifics of spacing, walls, etc, are entirely up to you -- be creative! Also, as a reminder, you can create a string of 10 underscore characters with `10*'_'`: string-multiplication is helpful here!

```
In [1]: rwsteps( 10, 5, 15 )
     |_____S_____|
     |____S______|
     |___S_______|
     |__S________|
     |___S_______|
     |____S______|
     |___S_______|
     |__S________|
     |_S_________|
     |S__________|
Out[1]: 9                     # here is the return value!

In [2]: rwsteps( 10, 7, 20 )
           S
            S
             S
            S
           S
          S
           S
          S
           S
          S
         S
        S
Out[2]: 11
```

Use recursion to implement `rwsteps` for this problem.

Hints: this problem can be tricky because you are both adding a random step **and** adding to the ongoing count of the total number of steps!

One way to do this is to use the line `rest_of_steps = rwsteps( newstart, low, hi )` as the recursive call, *with an appropriate assignment to* `newstart` on the line above it, and an appropriate use of `rest_of_steps` in the return value below it... .

Recursion limit exceeded? You can get more memory for recursion by adding these lines to the top of your file:

```
import sys
sys.setrecursionlimit(50000)
```

This provides 50000 function calls in the recursive stack.

Want to slow down your sleepwalker? You can also slow down the simulation by adding these lines to the top of your file:

```
import time
import sys
```

Then, in your `rwsteps` or `rwpos` functions, you can include the lines

```
sys.stdout.flush()    # forces Python to print everything _now_
time.sleep(0.1)       # and then sleep for 0.1 seconds
```

Adjust as you see fit!

Part 4: Create simulations to analyze your random walks

To analyze random walks, we need two terms:

1. The ***"signed-displacement"*** is the number of steps *away from the start* that the random walker has reached. It is signed, because displacements to the right are considered positive and displacements to the left are considered negative. This is natural: to find the signed displacement, simply subtract: it's the ending position of the random walker minus the starting position of the random walker. To do this, you will write a variation of `rwopos`, not `rwsteps`.

2. The ***"squared-displacement"*** is the *square* of the number of steps away from the start that the random walker has reached. That is, it is the square of the signed displacement.

With these two terms in mind, here are the two questions we ask you to investigate:

- What is the average final *signed-displacement* for a random walker after making `100` random steps? What about after `N` random steps? As described above, the signed-displacement is just the output of `rwpos` minus the `start` location. Do **not** use `abs`.
- What is the average *squared-displacement* for a random walker after making `100` random steps? What about after `N` random steps, in terms of `N`? Be sure you square the signed displacements **before** you sum the values in order to average them!

You should adapt the random-walk functions you wrote to investigate these two questions. In particular, you should

- **To-do item #1** Write a version of `rwpos` that does **not** print any debugging or explanatory information. Rather, it should simply return the final position. Call this new version `rwposPlain` . **Be careful!** the recursive call(s) will need to change so that they call `rwposPlain`, not `rwpos`!
- **To-do item #2** Come up with a plan for how you will answer these questions. This plan should include a list comprehension similar to the following:

  ```
  LC = [ rwposPlain(0,100) for x in range(142) ]
  ```

  Not surprisingly, the `142` will probably be replaced by a variable in your final implementation. To find the *average* of the values created, you will use `sum(LC)`, along with `len(LC)`...
- **To-do item #3** To build intuition, run the above list comprehension at the Python `>>>` prompt. Look at the resulting value of `LC` (there will be 142 elements). Also, find the average of `LC`.
- **To-do item #4** Write two more functions:
  - `ave_signed_displacement( numtrials )`, which should run `rwposPlain(0,100)` for `numtrials` times and return the average of the result. Use the above list comprehension as the first line of your function! (You'll want to replace
  - `ave_squared_displacement( numtrials )`, which should run `rwposPlain(0,100)` for `numtrials` times and return the average of the ***squares*** of the results! One way to do this is to create a

slightly different list comprehension. Remember that `x**2` is Python's way of squaring `x`.

- Then, use your functions and reflect on the results you find from these computational tests. To do this, place your answers inside your python program file by either making them comments (using the `#` symbol) OR, ***even easier***, including them in triple-quoted strings (since they can include newlines). For example,

```
"""
     In order to compute the average signed displacement for
     a random walker after 100 random steps, I ...
         (briefly explain what you did and your results)

     Be sure to copy the data and average from at least
     one of your runs of ave_signed_displacement and
     at least one of your runs of ave_squared_displacement
"""
```

Thus, your file should include

- (1) answers to these two questions and how you approached them and
- (2) the above Python functions, including `ave_signed_displacement( numtrials )` and `ave_squared_displacement( numtrials )`

Make sure to include explanatory docstrings and comments for each function you write!

Please include any references you might have used - you're welcome to read all about random walks online, if you like.

However, you should feel free not to bother - whether your answers/analyses are correct or not will have *no effect* on the grading of this Part 4 of this problem!

Rather, it will be graded on whether your functions work as they should, whether they *would be helpful* in answering those questions, and in the clarity and effectiveness of your write-up.

Extra: Optional Ex. Cr. variations

For up to +5ec points (optional), feel free to make variations in the ASCII wandering of your sleepwalker(s)...

- For example, a particularly creative ASCII emoji or boundary might be worth +1 or +2 points.
- A character that changes depending on whether it's moving left or right might be worth +2 or +3 points.
- A separate `rwsteps` function that has more than one wanderer (perhaps interacting with each other) might be +3 or +4 points
- Combining all of these or doing something totally crazy (2d, anyone?) might be +4 or +5 points ...

Be sure to add an clear and obvious comment bragging about your extras -- we don't want to miss them!

Submit!

Submit your `hw2pr2.py` file at the usual place!