Problem 1: Evolving Lists and Lights On!

The bird's-eye view

The end goal of this lab is to build a game called "Lights On" or "Lights Out," the goal of which is to get all of a grid of (initially random) lights to be on (or off). Each time the user selects a light, that light toggles from `0` to `1` or from `1` to `0`. The challenge is that **the neighboring light(s) also toggle**. That is, the one, two, three or four lights directly to the N, S, E and W of the selected light also change state (either on to off, or off to on). In our game, the lights do **not** "wrap around", e.g., the leftmost light is **not** a neighbor to the rightmost light.

You can play a 2d version of Lights Out here: http://www.logicgamesonline.com/lightsout/

You'll implement a 1d version in this problem.

This problem will guide you in developing several Python functions that will allow you to implement this game. In doing so, we will introduce and reinforce the concepts of:

- Decomposing problems into small pieces, i.e., functions
- Composing functions to create more powerful ones
- Recursion: writing functions in terms of themselves
- Representing data as lists
- Design/test/repeat: an exploratory approach to computation
- Also, we'll use *default inputs* to Python functions. They're a useful shortcut!

Get started by making a copy of the trinket below.

Try it out!

Although the lab will lead you through the decomposition of this problem and composition of its solution, it's a good idea to run the provided code and read it through to gain an intuitive understanding of its structure.

Try out the call

```
>>> evolve( [1,2,3], 0
)
[1, 2, 3]   (g: 0)
[2, 3, 4]   (g: 1)
[3, 4, 5]   (g: 2)
[4, 5, 6]   (g: 3)
[5, 6, 7]   (g: 4)
[6, 7, 8]   (g: 5)
```

and read through both the `evolve` function and the `mutate` function so that this behavior makes sense!

Note that we abbreviated `gen` with `g` in the printing above—certainly feel free to adapt the printing to be as you'd like it!

Next, try out the same call to `evolve` ***without*** its second input:

```
>>> evolve( [1,2,3] )   # notice this has only one
input
[1, 2, 3] (g: 0)
[2, 3, 4] (g: 1)
[3, 4, 5] (g: 2)
[4, 5, 6] (g: 3)
[5, 6, 7] (g: 4)
[6, 7, 8] (g: 5)
```

Here, you are seeing Python's ability to use *default input values*. A function can include a default value when it is declared:

```
def evolve(oldL, curgen=0):
```

and, when no input is provided, it will use that default value. Note that the default values have to be *later* in the list of inputs than the non-default values. Otherwise Python won't know where to put the inputs provided!

Writing New Versions of `mutate`

For each of the following questions, define a **new** function named `mutate` that produces the desired sequence of lists. There is a completed example in Question 0.

For each of these questions, write a new `mutate` in order to produce the behavior you want. You'll need to change the version of `mutate` that `evolve` is using each time you add a `mutate` function. Make sure all of the intermediate versions are still in your file (but this way, only the last one will be tested). This first `mutate` is simply an example as a guide.

Question 0

Write a `mutate0` function that yields the following behavior. Note that we're still calling `evolve`, but it's `mutate0` that you're writing. Don't forget to change the version of `mutate` that `evolve` is calling!

```
>>> evolve( [1,2,3]
)
[1, 2, 3] (g: 0)
[2, 4, 6] (g: 1)
[4, 8, 12] (g: 2)
[8, 16, 24] (g: 3)
[16, 32, 48] (g: 4)
[32, 64, 96] (g: 5)
```

Answer to Question 0

The idea here is that each output element is double the corresponding input element. Thus, the code is the following, simply cut, pasted, and modified from the old `mutate`:

```
def mutate0(i, oldL):
    """ takes as input an index (i) and an old list (oldL)
        mutate returns the ith element of a NEW list!
        * note that mutate returns ONLY the ith element
          mutate thus needs to be called many times in
evolve
    """
    new_ith_element = 2 * oldL[i]
    return new_ith_element
```

Question 1

Write a `mutate1` function that yields the following behavior:

```
>>> evolve( [1,2,3] )
[1, 2, 3] (g: 0)
[1, 4, 9] (g: 1)
[1, 16, 81] (g: 2)
[1, 256, 6561] (g: 3)
[1, 65536, 43046721] (g: 4)
[1, 4294967296L, 1853020188851841L] (g:
5)
```

Hint: Notice that each element is the *square* of the one above it.

Question 2

This example uses a slightly longer initial list. Write a `mutate2` function that yields the following behavior:

```
>>> evolve( [1,2,3,42]
)
[1, 2, 3, 42] (g: 0)
[42, 1, 2, 3] (g: 1)
[3, 42, 1, 2] (g: 2)
[2, 3, 42, 1] (g: 3)
[1, 2, 3, 42] (g: 4)
[42, 1, 2, 3] (g: 5)
```

Hint: Each of `mutate2`'s returned values is the value from the old list, `oldL` that is located one index to the left (lower) than the current index. Thus, the `return` line will be

```
return oldL[ SOMETHING
]
```

where `SOMETHING` is a very short expression involving `i` and `1`.

Question 3: A *random* list generator

Write a `mutate3` function that yields a random list of `0`s and `1`s with each generation. It completely ignores the input list! For example (and lots of other output behaviors could occur, as well):

```
>>> evolve( [1,2,3,42] )  # this input list is
ignored!
[1, 2, 3, 42] (g: 0)
[1, 0, 0, 0] (g: 1)
[0, 1, 1, 1] (g: 2)
[0, 1, 1, 1] (g: 3)
[1, 0, 0, 0] (g: 4)
[1, 0, 0, 1] (g: 5)
```

Reminder: The function that chooses an element randomly from a list is called as follows:

```
choice( [0, 1]
)
```

That's all you'll need! (*No need to create lists or list comprehensions here!* Keep in mind that `evolve` already does that for you, and `mutate3` produces **only one element at a time**—namely, element `i`.)

The next part of the problem will build on this randomly-evolving behavior.

Counting Generations

At the moment, your different versions of `mutate` have directed `evolve` to change its input lists in a number of ways, but it so far has not evolved them for any concrete purpose or to achieve any particular result.

In the game of Lights On, the goal is to evolve the list so that all of its values are "on". Throughout the rest of the lab, we will use `1` to indicate that a cell is "on" and `0` to indicate that it is "off". In this portion of the lab, we will experiment with several strategies for evolving a list into a same-length list of all `1`s. From now on, our initial lists will consist only of `0`s and `1`s.

Detecting when we've reached our goal

In your Homework 4, Problem 1 file write a function named `allOnes(L)` that takes as input a list of numbers `L` and returns `True` if all of `L`'s elements are `1` and returns `False` otherwise. Raw recursion is one good way to do this, though not the only one. Notice that the empty list vacuously satisfies the all-ones criterion, because it has no elements at all! Here are some examples to check:

```
>>> allOnes([1,1,1])
True

>>> allOnes([])
True

>>> allOnes([ 0, 0, 2, 2 ])      # this should be False!
False    # but be careful if you use sum(L) == len(L), this will be
True

>>> allOnes([ 1, 1, 0 ])
False
```

Hint: If you use recursion, a natural base case would be to handle the empty list `[]`. Note that `allOnes( [] )` is `True`!

Caution about `True`/`False`!

Especially if you use recursion, you may want to use the line

```
return
True
```

somewhere in your code, as well as the line

```
return False
```

Be **sure** to `return` (and not `print`) these values! Also, watch out that you're returning the *values* `True` and `False`. You **don't** want to return the strings `"True"` and `"False"`!

An improved `evolve` function

Now that you have a function for testing if a list is all ones, improve your `evolve` function in two ways:

- First, change the base case condition so that it stops when the input list is all `1`s. Use your `allOnes` function!
- Second, change `evolve` so that it **returns** the number of generations that were needed to evolve the input into all `1`s.

Suggestions:

- Leave the `print` and pause lines *before* the check to see if the all-ones base case has been reached. That way they will run both when it's the base case and when it's the recursive case.

Trying it out

First, you might want to reduce (or remove) pause produced by the line `time.sleep(0.25)`. A value of a twentieth of a second (or zero) might be better for these trials.

Then, try your new `evolve` function and your random-number-generating `mutate3` functions on input lists with

varying length—after all, their elements are not being used yet. Here are two examples:

```
>>> evolve( [0,0,0,0,1]
)
[0, 0, 0, 0, 0]
[1, 0, 1, 1, 1]
[1, 1, 0, 0, 0]
[1, 0, 1, 0, 1]
[1, 0, 0, 0, 1]
[1, 1, 1, 1, 0]
[0, 1, 1, 0, 0]
[1, 1, 0, 1, 0]
[0, 0, 1, 1, 0]
[0, 1, 1, 1, 1]
[1, 1, 1, 0, 0]
[1, 1, 0, 1, 0]
[1, 1, 1, 1, 1]
12
>>> evolve( [0,1,0,1] )
[0, 1, 0, 1]
[0, 0, 0, 0]
[0, 1, 0, 1]
[1, 1, 0, 1]
[1, 1, 1, 1]
4
```

It's worth mentioning that it can take *much* longer for a 5-element list to randomly come up all-1s. One test we ran took 93 steps.

As a thought experiment, how many steps would you *expect*—over many trials—for a 5-element list to randomly generate all 1s?

Please add a short comment answering this question at this point in your code.

User Input

At the moment, your evolver for lists does not use any *human* input. This section adds your input to the game—first by toggling each light individually and then to toggle a light and all of its neighbors simultaneously. This will implement the full game of *Lights On*!

Evolving lists with user input

The approach to evolving lists thus far is, well, a bit too random. This section will enable the user to guide the process by choosing an element from the list.

First, copy this `mutate4` function and change `evolve` to use this new version of `mutate`:

```
def mutate4(i, oldL, user=0):
    """ takes as input an index (i) and an old list (oldL)
        mutate returns the ith element of a NEW list!
        * note that mutate returns ONLY the ith element
          mutate thus needs to be called many times in evolve
    """
    if i == user:
        new_ith_element = 1        # this makes the game easy!
    else:
        new_ith_element = oldL[i] # the new is the same as the
old
    return new_ith_element
```

This version of `mutate` takes a third input, named `user`. This third input will be the index the user chooses. Note that this new `mutate` changes AT MOST the single element that matches the input named `user`. All other returned values are simply the old values `oldL[i]`.

Next, change the line in `evolve` that reads

```
newL = [ mutate4(i,oldL) for i in range(len(oldL))
]
```

and replace it with these two lines in its place:

```
user = input("Index? ")
newL = [ mutate4(i,oldL,user) for i in range(len(oldL))
]
```

which asks the user for an index value, gives the user's input the variable name `user`, and then passes that variable into the new `mutate4` function as its third input.

Try running `evolve( [0,0,0,0] )`. Now, the execution should pause and wait for you to enter the index of one of the list items.

Here is an example run of ours:

```
>>> evolve( [0,0,0,0]
)
[0, 0, 0, 0] (g: 0)
Index? 3
[0, 0, 0, 1] (g: 1)
Index? 2
[0, 0, 1, 1] (g: 2)
Index? 1
[0, 1, 1, 1] (g: 3)
Index? 0
[1, 1, 1, 1] (g: 4)
4
```

It's clear we need to add some additional challenge to this game!

Toggling the lights

Create a new `mutate5` **function** so that when the user chooses a light, whose index will be held in the variable `user`, then that light will *toggle* from `0` to `1` or from `1` to `0`, as appropriate.

Hint: If the old value of the light is `oldL[i]`, what will you get if you subtract that value from `1`?

Be sure to test your code by running something like this:

```
>>> evolve( [1,0,1,1]
)
[1, 0, 1, 1] (g: 0)
Index? 2
[1, 0, 0, 1] (g: 1)
Index? 3
[1, 0, 0, 0] (g: 2)
Index? 1
[1, 1, 0, 0] (g: 3)
Index? 2
[1, 1, 1, 0] (g: 4)
Index? 3
[1, 1, 1, 1] (g: 5)
5
```

Hint: In your `mutate5` function you will want to use an `if` and an `else`. Consider testing whether `i` is the same value as `user`, that is, `user i ==` .

At this point, it's a game, it's still not really a challenge to win. You'll fix that next.

Toggling neighboring lights, too

Now, you are finally ready to implement the full 1D version of "Lights On".

Make a new `mutate6` function so that the game play is as it should be—that is, when you toggle one light, the lights next to it also toggle.

Suggestions: Don't change `evolve` except to change what `mutate` is being called! Only `mutate` needs to change. Remember that `mutate` only returns the `i`th element—there no way to change that (with our current structure). However, you can make the test for when to toggle the lights more inclusive!

For example, above you allowed the light to toggle ONLY when `user i ==` . Now, you'll want to have a test with some other possibilities as well—that is,

```
if i == user or
:
```

What other values could `i` equal for which we want to toggle the lights?

Try your game from a known starting position in order to test it out:

```
>>> evolve( [1,0,0,1,0,0,1,1]
)
[1, 0, 0, 1, 0, 0, 1, 1] (g:
0)
Index? 4
[1, 0, 0, 0, 1, 1, 1, 1] (g:
1)
Index? 2
[1, 1, 1, 1, 1, 1, 1, 1] (g:
2)
2


>>> evolve( [0,0,0,0,0,0,0,0]
)
[0, 0, 0, 0, 0, 0, 0, 0] (g:
0)
Index? 0
[1, 1, 0, 0, 0, 0, 0, 0] (g:
1)
Index? 3
[1, 1, 1, 1, 1, 0, 0, 0] (g:
2)
Index? 7
[1, 1, 1, 1, 1, 0, 1, 1] (g:
3)
Index? 6
[1, 1, 1, 1, 1, 1, 0, 0] (g:
4)
Index? 7
[1, 1, 1, 1, 1, 1, 1, 1] (g:
5)
5
```

We still need a random starting point—that's next!

Starting from a random binary list

Create a function `randBL(N)` which will output a random binary list (a random list of `N` zeros and ones). This function should take in a nonnegative integer, `N` and should return a list of length `N`, in which each element is randomly either `0` or a `1`.

Raw recursion is one way to handle this; list comprehensions are another. **Warning**: One thing that *won't* work is the following:

```
return [ choice([0,1]) ] * N   # won't
work!!
```

The reason is that the above line only returns a list of `N` zeros or `N` ones.

Here are two examples of `randBL` in action—be sure to test your function, though the outputs are likely to be different:

```
>>> randBL( 5 )
[1, 0, 1, 1, 0]

>>> randBL( 9 )
[0, 0, 1, 1, 1, 0, 1, 1,
0]
```

This `randBL` function makes it easy to start a new, random game.

Also, you should feel free to change the formatting of your printed output to make the game easier to play—for example:

- Print a list of the indices above the current game state
- Let the `Index?` question come at the end of the previous line

or any other formatting you might like. Here's what ours turned out to be:

```
>>> evolve( randBL(8) )

[0, 1, 2, 3, 4, 5, 6, 7]
[0, 1, 1, 0, 1, 1, 0, 0] (g: 0) Index?
1

[0, 1, 2, 3, 4, 5, 6, 7]
[1, 0, 0, 0, 1, 1, 0, 0] (g: 1) Index?
2

[0, 1, 2, 3, 4, 5, 6, 7]
[1, 1, 1, 1, 1, 1, 0, 0] (g: 2) Index?
7

[0, 1, 2, 3, 4, 5, 6, 7]
[1, 1, 1, 1, 1, 1, 1, 1] (g: 3)
3
```

Let the computer play!?

To finish things up, change `evolve` so that the computer gets to play! That is, have the computer choose (randomly) one of the possible indices in place of theline that currently lets the user (human) provide input.

That's it—once you make the switch from human choice of a square to a computer-made, random choice, the machine should play the game (making random selections) until it's solved.

Warning: One-sixth of all binary lists are dead ends! That is, there is **no** combination of toggled lights that will end up at the all-ones list. So, if the computer (or you) seems unable to solve one of the starting configurations, stop the program and try again!

Congratulations! You've complete this problem!

Submit Homework 4, Problem 1

35.0/35.0 points (graded)

To submit your Homework 4, Problem 1 code, you'll need to copy it from the trinket above and paste it into the box below. After you've pasted your code below, click the "Check" button.

Please remove any delays, such as use of `time.sleep()`, from your code before you submit it. These delays will cause your assignment to run too long, and the grader will produce a timeout error.

IMPORTANT: Make sure that there aren't spaces at the beginning of your code, and that you copied all of the characters. If there are extra spaces or you are missing spaces, our server won't be able to run your code and we won't be able to give you any of the points you deserve for your hard work.

1

2

3

4

5

6

7

8

9

10

```
import time
```

11

```
from random import *
```

12

13

14

```
def mutate(i,
oldL):
```

15

```
    """ Accepts an index (i) and an old list
(oldL).
```

16

```
        mutate returns the ith element of a NEW
list!
```

17

```
        * Note that mutate returns ONLY the ith
element
```

18

```
          mutate thus needs to be called many times in
evolve.
```

19

```
"""
```

20

```
    new_ith_element = 1 +
oldL[i]
```

21

```
    return
new_ith_element
```

22

23

```
def mutate0(i,
oldL):
```

24

```
    """ Accepts an index (i) and an old list
(oldL).
```

25

```
        mutate returns the ith element of a NEW
list!
```

Press ESC then TAB or click outside of the code editor to exit
correct

correct

Test results
CORRECT See full outputSee full output

You have used 2 of 3 attempts Some problems have options such as save, reset, hints, or show answer. These options follow the Submit button.