

WASHINGTON STATE UNIVERSITY VANCOUVER

PROGRAM DESIGN AND DEVELOPMENT - CS 121

---

## Assignment 5 Part 1

---

*Professor:*  
Ben MCCAMISH

September 29, 2020

# Overall Assignment

---

Your code should all be contained in a single file (excluding additional files I provide) titled `assignment5p1.py`. Make sure that they all work on the Ubuntu OS on the lab machines. Your program must be written for Python 3.6+.

You may have noticed that this task (programming Lights On) is bigger than what I have asked you to do so far. This program will require more than one function, and some planning about how to organize the data the game needs to keep track of and how the functions interact and modify this data.

## 1 Getting started

---

Create a new file called `assignment5p1.py` and put the a header at the top of that file similar to this:

```
# Assignment 5 Part 1  
# John Smith  
# September 24, 2019
```

Although I will lead you through the decomposition of this problem, I would like you to do some brainstorming to help you practice breaking a problem down into smaller pieces. Without looking ahead, in comments at the top of that file, answer the following:

1. What data will your program need to keep track of?
2. What will your program need to do with that data?

Try to be specific, describing what you could do with a very short function (e.g., one of the things your program will need to do might be "Take input from the user about which light they want to toggle")

## 2 Implementing piece by piece

---

A central piece of data to keep track of, which you likely noted above, is the current state of the board. One of the central pieces of functionality will be to modify this data after each turn, in response to the user's input. This first section explores how we will represent the board state, and has you write a number of functions that modify this data in various ways.

To simplify things, we will begin working with a 1D version of the game, where the lights exist in a row, rather than a grid. As you might have already guessed, we will use a list to store the current state of the board.

In this section, we will not yet implement the data modifications for Lights On, nor will we (yet) respond to user input. That will come later. The purpose of this section is simply for you to practice modifying the values in a list.

Copy (or confirm they exist) these three functions (and import lines) into your `assignment5p1.py` file:

```
import time # provides time.sleep(0.5)
from random import choice # provides choice( [0,1] ), etc.
import sys # larger recursive stack
```

```
sys.setrecursionlimit(100000) # 100,000 deep
```

```
def runGenerations( L ):
    """
    runGenerations keeps running evolve...
    """

    print( L )          # display the list, L
    time.sleep(0.5)      # pause a bit
    newL = evolve( L )   # evolve L into newL
    runGenerations( newL ) # recurse
```

```
def evolve( L ):
    """
    evolve takes in a list of integers, L,
    and returns a new list of integers
    considered to be the "next generation"
    """

    N = len(L) # N now holds the size of the list L

    return [ setNewElement( L, i ) for i in range(N) ]
```

```
def setNewElement( L, i, x=0 ):
    """
    setNewElement returns the NEW list's ith element
    input L: any list of integers
    input i: the index of the new element to return
    input x: an extra, optional input for future use
    """

    return L[i] + 1
```

One note on this code: the `x=0` in the third argument to `setNewElement` is an *optional* input. That is, if you provide a third input to `setNewElement`, the value you provide becomes `x`. However, if you do not provide a third input, the value `x = 0` is used instead. Here, `x` isn't used, though it will be in the future.

### 3 Test it out

---

Save your code. Then test it out with `runGenerations([1,2,3])`.

You should see the following output:

```
[1, 2, 3]
[2, 3, 4]
[3, 4, 5]
[4, 5, 6]
[5, 6, 7]
```

You'll need to stop the function - otherwise it will run until its memory (or recursion limit) runs out.

Write a comment of 2-3 sentences describing what's happening in the above example. Be sure to include a brief description of how each of the three functions contributes to the behavior. Remember that you can use Python's per-line comment character, `#`, or you can simply place your comments in a triple-quoted string in your `assignment5p1.py` file.

```
"""
like this
"""
```

**NOTE:** To succeed in this lab, it is extremely important that you understand exactly how and why `runGenerations` works. If you aren't 100% certain you understand what's going on, ask for help. Later problems will depend on your understanding of `runGenerations`, `evolve`, and `setNewElement`. I would highly recommend you begin this assignment early to have enough time to ask questions!

### 4 Questions

---

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

You will create a new function for each question with a slightly modified function definition. For each of these questions, paste a new `setNewElement` function after your most recently defined one, and then change it to match the behavior you want. Also change the name to match the question. For example, if I were writing a function for Question 1, the function would be defined as `setNewElementq1`. That way, all of the intermediate versions will still be in your file but autolab can test them individually. You will need to change the call in `evolve` to to get the proper output when calling `runGenerations`.

Add a comment to each `setNewElement` function to indicate which question it is intended to answer. Also, you should probably create a `main`, otherwise autograder may continue with infinite loops and never complete the autograding. Add the following to the bottom of your file and then in `main` make your calls to `runGenerations`.

```
def main():

if __name__ == '__main__':
    main()
```

## Question 0

Write a `setNewElement` function that yields the following behavior when `runGenerations( [1,2,3] )` is called.

```
[1, 2, 3]
[2, 4, 6]
[4, 8, 12]
[8, 16, 24]
[16, 32, 48]
[32, 64, 96]
[64, 128, 192]
```

## 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 `setNewElement`:

```
def setNewElementq0( L, i, x=0 ):
    """
    setNewElement returns the NEW list's ith element
    input L: any list of integers
    input i: the index of the new element to return
    input x: an extra, optional input for future use
    """

    return L[i]*2
```

## Question 1

Write a `setNewElement` function that yields the following behavior when `runGenerations( [1,2,3] )` is called.

```
[1, 2, 3]
[1, 4, 9]
[1, 16, 81]
[1, 256, 6561]
[1, 65536, 43046721]
```

## Question 2

This example uses a slightly longer initial list. Write a `setNewElement` function that yields the following behavior when `runGenerations( [1,2,3,4,5,42] )` is called.

```
[1, 2, 3, 4, 5, 42]
[42, 1, 2, 3, 4, 5]
[5, 42, 1, 2, 3, 4]
[4, 5, 42, 1, 2, 3]
[3, 4, 5, 42, 1, 2]
[2, 3, 4, 5, 42, 1]
[1, 2, 3, 4, 5, 42]
[42, 1, 2, 3, 4, 5]
[5, 42, 1, 2, 3, 4]
```

**Hint:** each returned value is the value from the old list, `L`, one index to the left (lower) than the current index. Thus, the `return` line will be `return L[ SOMETHING ]` where `SOMETHING` is a very short expression involving `i` and `1`.

### Question 3

Write a `setNewElement` function that yields the following behavior when `runGenerations( [1,2,3,4,5,42] )` is called.

```
[1, 2, 3, 4, 5, 42]
[2, 3, 4, 5, 42, 1]
[3, 4, 5, 42, 1, 2]
[4, 5, 42, 1, 2, 3]
[5, 42, 1, 2, 3, 4]
[42, 1, 2, 3, 4, 5]
[1, 2, 3, 4, 5, 42]
[2, 3, 4, 5, 42, 1]
```

**Hint:** this is the opposite of the previous example. However, depending on how you implement it, you may need an `if` and an `else` to handle the very last column.

### Question 4: A random list generator...

Write a `setNewElement` function that yields a random list of 0s and 1s with each generation. It completely ignores the input list! For example (and lots of other behaviors could occur, as well) when I call `runGenerations( [1,2,3,4,5,42] )` I get the output:

```
[1, 2, 3, 4, 5, 42]
[0, 0, 1, 1, 1, 0]
[0, 0, 1, 1, 0, 0]
[1, 0, 0, 1, 0, 0]
[0, 1, 0, 1, 1, 0]
[0, 0, 0, 1, 0, 0]
```

**Reminder:** the random-choice function is `choice( [0,1] )` – that’s all you’ll need!

## 5 Determining Victory

---

At the moment, the `runGenerations` function has evolved its input lists in a number of ways, but it so far has not evolved them for any 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 1s. From now on, our initial lists will consist only of 0s and 1s.

In your `assignment5p1.py` 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
```

**Caution about True/False!** You will 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 boolean values `True` and `False`. You DON'T want to return the strings "True" and "False".

## Improving the function `runGenerations`

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

1. First, add a base case to the recursion, so that `runGenerations` stops when the input list is all 1s. An alternative is to remove the recursion and use a loop instead.
2. Second, change `runGenerations` so that it returns the number of generations needed to evolve the input into all 1s.

## 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 whether it's the base case or the recursive case that runs afterward. In order to return the number of generations required to evolve the list into all ones, consider the `mylen` recursive example written in class. The idea is to add 1 for each evolve step. That 1 needs to be added to the number of steps needed to evolve the new list into all-ones.

## Testing it out

First, you might want to reduce or remove the half-second pause produced by the line `time.sleep(0.5)`. A value of a twentieth of a second (or zero) might be better for these trials.

Then, try your new `runGenerations` function on input lists with varying numbers of 0s. You should use the random element chooser that you wrote at the end of the previous part of the lab as your `setNewElement` function. Here are two examples:

```
>>> runGenerations([0,0,0,0,1])
```

```
[0, 0, 0, 0, 1]
[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
```

```
>>> runGenerations([0,1,0,1,1])
```

```
[0, 1, 0, 1, 1]
[0, 0, 0, 1, 0]
[0, 1, 0, 1, 1]
[1, 1, 0, 1, 1]
[1, 1, 1, 1, 1]
4
```

## Autolab Notes:

---

- Submit your python code as you go to determine whether you answered the questions correctly.
- Autograder has its limits, so it cannot test the random methods. This means half of your grade will be determined by the TA. Get started early so you can confirm correct output during lab times.
- Make sure to include a main function.
- (5%) `setNewElement()`
- (10%) `setNewElementq1()`
- (10%) `setNewElementq2()`
- (10%) `setNewElementq3()`
- (5%) `setNewElementq4()`
- (10%) `allOnes()`
- (50%) TA performing their own tests on `setNewElementq4()`, `newGenerations()`, and commenting/style.