CS Scholars Programming - Final Evaluation
Evaluation Period: Thursday 08/12 12:00pm - 1:15pm EST

Name:

AndrewID:

This final evaluation will test the knowledge you have accumulated over the course. You should complete this evaluation during the class period on Thursday 08/12 (from 12pm - 1:15pm EST) and submit it to Gradescope by 1:30pm EST on the same day.

This final evaluation is open-note, but closed to collaboration. You are welcome to consult the course notes, personal notes, and your past homework assignments during the evaluation, but you should not communicate with anyone outside of the course staff during the period of the evaluation. In particular, do not collaborate with other students.

The final evaluation consists of both a written portion and a programming portion. The written problems should be completed in this starter file; the programming problems should be completed in the programming starter file. For full credit, complete problems #1-#5 (graded out of 80 points); for full credit on the spicy version of the evaluation, complete #6 as well.

As a general guideline: each written problem should take approximately 10 minutes to complete, and each programming problem should take approximately 15 minutes.

Problems #4 and #5 can be tested with the test functions included in the starter file. Problem #6 should be tested interactively by running the simulation.

Written Problems [40pts]

#1 - Programming Building Blocks - 15pts

Programming Basics, Functions, Conditionals, Loops

Consider the following program:

```
1  def f(x, y):
2      if x < y:
3          z = x + y // 2
4      else:
5          z = x - y // 2
6      print(x, y, z)
7      return z
8
9  x = 0
10 i = 8
11 while i > 0:
12     x = f(x, i)
13     i = i - 2
14 print("i:", i)
15 print("x:", x)
```

What will this program print when it runs? **Hint:** make a variable table on scratch paper!

#2 - References - 15pts

Strings and Lists, References and Memory

Consider the following code:

```
1  A = [4, 7, 9]
2  B = [4, 7, 9]
3  C = B
4  D = C
5  # PAUSE
6  A.pop(0)
7  B[1] = 1
8  C = C + [0]
9  D.append(42)
```

Which variables share the **same reference** when the code reaches line #5 (`# PAUSE`) ?

Which variables share the **same reference** when the code finishes running?

List all the line numbers where the line of code performs a **destructive** action.

Which of the following is the correct set of values held by the four variables at the end of the code?

- ☐ `A = [7, 9] ; B = [4, 1, 9] ; C = [4, 7, 9, 0] ; D = [4, 7, 9, 42]`
- ☐ `A = [7, 9] ; B = [4, 1, 9] ; C = [4, 7, 9, 0, 42] ; D = [4, 7, 9, 0, 42]`
- ☐ `A = [7, 9] ; B = [4, 1, 9, 42] ; C = [4, 1, 9, 0] ; D = [4, 1, 9, 42]`
- ☐ `A = [7, 1, 42] ; B = [7, 1, 42] ; C = [7, 1, 0] ; D = [7, 1, 42]`
- ☐ `A = [7, 9, 42] ; B = [7, 1] ; C = [7, 9, 0] ; D = [7, 9, 42]`
- ☐ `A = [7, 1, 0, 42]; B = [7, 1, 0, 42]; C = [7, 1, 0, 42]; D = [7, 1, 0, 42]`
- ☐ `A = [7, 9, 0, 42] ; B = [7, 1] ; C = [7, 9, 0, 42] ; D = [7, 9, 0, 42]`

#3 - Programming Meta-Skills - 10pts

Algorithms, Testing and Debugging, Style

The following three questions test your understanding of programming meta-skills.

A: Say you've written an algorithm at a **high** level of abstraction to instruct a person in how to clean their teeth. You tell them to:

1. Put toothpaste on their toothbrush
2. Brush their teeth
3. Rinse their mouth
4. Rinse their toothbrush

Which of the following revisions would move this algorithm to a **medium** level of abstraction? **Select all that apply.**

- ☐ Add more detail to Step 1
  - Ex: Open the toothpaste container, squeeze a small drop onto the brush part of the toothbrush, close the container
- ☐ Break Step 2 into parts
  - Ex: Brush the front of your top teeth, then the back, then the edges, etc.
- ☐ Combine Steps 3 and 4 into one general step about rinsing
  - Ex: Rinse and you're done
- ☐ Add a Step 5 about cleaning up afterwards
  - Ex: Put away the toothbrush and toothpaste in their proper places

---

B: Say you encounter the following error message while coding (cont'd on next page):

```
1  def makePairs(lst):
2      result = []
3      for i in range(len(lst)):
4          result.append([lst[i], lst[i+1]])
5      return result
6
7  assert(makePairs([1, 2, 3]) == [ [1, 2], [2, 3] ])
```

```
Traceback (most recent call last):
  File "test.py", line 7, in <module>
    assert(makePairs([1, 2, 3]) == [ [1, 2], [2, 3] ])
  File "test.py", line 4, in makePairs
    result.append([lst[i], lst[i+1]])
IndexError: list index out of range
```

What is the **type** of this error?

- ☐ Syntax Error
- ☐ Runtime Error
- ☐ Logical Error

Which of the following steps would **most** help you debug this problem? Multiple answers may be correct, but you should just select the best answer.

- ☐ Carefully read the second-to-last line in the error message
- ☐ Carefully read Line 4 and the context around it
- ☐ Carefully read Line 7 and the context around it
- ☐ Add a print statement that checks the value of `i` before line 4
- ☐ Add a print statement that checks the value of `lst` before line 4
- ☐ Add a print statement that checks the value of `result` before line 5
- ☐ Run `makePairs([1, 2, 3])` to see what it returns

C: What is the **biggest** problem with the following segment of code, which is supposed to find the index of the largest value in a 2D list?

```
def biggestIndex(lst):
    tmp1= 0
    tmp2 =0
    for a in range(len(lst)):
        for b in range(len(lst[a])):
            if lst[a] [b] > lst[ tmp1 ][ tmp2 ]:
                tmp1=a
                tmp2 = b
            else:
                pass
    return [tmp1, tmp2]
```

- ☐ The code is not **optimal**
- ☐ The code is not **efficient**
- ☐ The code is not **clear**
- ☐ The code is not **robust**

Programming Problems [60pts]

#4 - Code Writing Fundamentals - 20pts

Programming Basics, Function Calls, Function Definitions

Write the function `randomlyClose(cutoff)`. This function takes an integer as a parameter, then generates two random numbers (each in the range [1, 100], inclusive). If the difference between those two numbers is less than or equal to the given `cutoff`, the function should return `True`; otherwise, it should return `False`.

Clarifying examples:

- If the function was given a cutoff of 20 and randomly generated the integers 93 and 88, it would return `True` (as the difference between 93 and 88, 5, is less than 20)
- If it was given a cutoff of 50 and generated the numbers 81 and 11, it would return `False` (as the difference between 81 and 11 is 70, larger than 50).
- If it was given a cutoff of 40 and generated the numbers 19 and 59, it would return `True` (as the difference between 19 and 59 is exactly 40).

Hint: recall that the `random` library has a function `randint(a, b)` that generates a random number in the range [a, b].

Note: the test case for this function should work if you produce random numbers in the way we expect, but may fail if you use a different approach. We'll hand-grade this problem to check that it's implemented correctly; the test case is just there to help you check your work. You might still get full credit even if the test case fails as long as your code is correct.

#5 - Control and Data Structures - 20pts

Conditionals, Loops, Strings, Lists

Write the function `lowerToUpper(words)` which takes a list of words (strings) and returns a new list that only contains the words that were **all-lowercase**. However, those returned words should be made **all-uppercase** before being returned.

For example, `lowerToUpper(["Hello", "isn't", "it", "Beautiful", "today"])` would return `["ISN'T", "IT", "TODAY"]`.

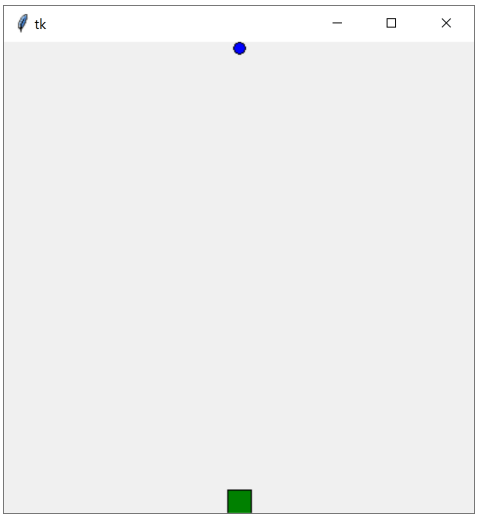
Hint: the string library has a method `islower()` which returns `True` if the string is all-lowercase and `False` otherwise, and a method `upper()` which returns a new all-uppercase version of the string.

#6 - SPICY: Interactive Programs - 20pts

Simulation, Large Programs

Write a simple simulation in our simulation framework (in a 400x400 window) that represents a plant growing over time as it is watered.

Step 1: The plant should be represented as a skinny green rectangle centered at the bottom of the screen. The water droplet can be represented as a small blue circle that starts centered at the top of the screen. So at the beginning of the simulation, the screen might look like this:



Step 2: As time passes, the water droplet should 'fall' towards the bottom of the screen at a steady rate.

Step 3: If at any point the water droplet overlaps with the plant rectangle, the droplet should 'disappear' (move it to somewhere under the bottom of the window to accomplish this), and the plant should grow a small amount towards the top of the screen. It should still be rooted at the bottom- just make the rectangle taller.

How can we tell if the droplet overlaps with the rectangle? We'll make it simple and say that it counts if the *bounding box* of the droplet overlaps, not the circle itself. We've included a pre-written **helper function**, `isOverlapping(bounds1, bounds2)`, that takes two lists - the `[left, top, right, bottom]` boundaries of two rectangles - and returns whether or not they overlap. You can then call this function on the boundaries of the plant and droplet every time the droplet moves to see if the two are overlapping.

Step 4: Finally, we'd like to be able to water the plant more than once, so add two ways to reset the position of the water droplet:

- If the user clicks on the screen, the droplet moves to be centered at that location
- If the user presses 'Enter' (and only when the user presses 'Enter'), the droplet moves to a random position on the screen

The droplet should still fall normally after being moved in either of these ways.

Here's a link to a gif of the finished game to clarify how it should work:
http://www.krivers.net/CSS-m21/demo.gif

Finished early? **For extra credit:** instead of having only one drop of water on the screen at a time, generate a **new** drop of water every time the user clicks the screen or presses Enter. You'll want to store these drops of water in a **list** in `data`, and iterate over that list in all parts of the code that deal with water droplets. Make sure to make a safety submission before you do this, just in case you break your original approach!