CS Scholars - Programming Hw3 - Written
Due Date: Friday 07/22 EOD

Name:

AndrewID:

For full credit on the assignment, complete either all Review + Core problems (#1-#10) or all Core + Spicy problems (#1-#3, #7-#13).

Bonus problems are related to the Advanced Track content, and are optional.

Core Problems - Written

#1 - Nested Code Tracing

Can attempt after Nesting and Top-Down Design lecture

Consider the following mystery code:

```python
def f(x):
    total = 0
    while x > 0:
        print("while:", x)
        digit = x % 10
        if digit < 5:
            print("if:", digit)
            for i in range(0, 10, digit):
                total = total + i
                print("for:", i, "-", total)
        else:
            print("else:", digit)
            total = 0
        x = x // 10
        print() # this prints a blank line
    return total
```

What will be printed if we call **`f(634)`**? Don't just run the code in the editor - try to trace the code and figure out the answer on your own. (You may want to use a variable table).

#2 - String Code Tracing

Can attempt after Strings and Lists lecture

Assuming that the following two lines of code have been run:

```
s1 = "coding is cool"
s2 = "CMU rocks!"
```

What will each of the following expressions evaluate to? Don't just run the code in the editor- try to figure out the answer on your own.

Expression	**Value**
`s1[7] + s2[6]`	
`s1[1] + s2[len(s2)-1]`	
`s1[7:11]`	
`s2[2:len(s2)-2]`	
`s1[::4]`	

#3 - List Code Tracing

Can attempt after Strings and Lists lecture

Consider the following code:

```
 1 lst = [ "hello", "world", "and", "all" ]
 2 allLetters = ""
 3 result = []
 4 for word in lst:
 5     tmp = ""
 6     for i in range(len(word)):
 7         if word[i] not in allLetters:
 8             tmp += word[i]
 9             allLetters += word[i]
10     result = result + [tmp]
11 print(result)
```

How many times will the loop on line 4 iterate?

In the first iteration of the loop on line 4, how many times will the loop on line 6 iterate?

What will this code print when run on the given input?

In general (non-code) terms, what does this code do?

Programming Problems

Each of these problems should be solved in the starter file available on the course website. They should be submitted to the Gradescope assignment Hw3 - Programming to be autograded.

Make sure to 'Run file as Script' to check your work before submitting, then check the autograder feedback after you submit!

Review Problems - Programming

#4 - `threeLinesArea`

Can attempt after the Nesting and Top-Down Design lecture

For this problem, you will implement **four functions**. One, **`threeLinesArea(m1, b1, m2, b2, m3, b3)`**, will be the primary function; its job is to take the slopes and intercepts of three lines and calculate the area between them. The other three functions support this function in its work.

The first helper function, **`lineIntersection(m1, b1, m2, b2)`**, takes the slopes and intercepts of two lines and returns the x value of the point where they intersect. Recall that this can be calculated as:

$$\frac{b2 - b1}{m1 - m2}$$

The second helper function, **`distance(x1, y1, x2, y2)`**, takes two (x, y) points and returns the distance between them. Recall that this can be calculated as:

$$\sqrt{(x2 - x1)^2 + (y2 - y1)^2}$$

The third helper function, **`triangleArea(a, b, c)`** takes three numbers (lengths of the sides of a triangle) and returns the area of a triangle with those sides. This can be calculated with Heron's formula as:

$$\sqrt{s(s-a)(s-b)(s-c)}$$

Where **s** is the semi-perimeter of the triangle, calculated as:

$$\frac{a+b+c}{2}$$

Once you've implemented the three helper functions, use them in `threeLinesArea` to:

- Find the points where each pair of lines intersects (noting that once you have the x coordinate, you can calculate the y coordinate based on the line's equation)
- Find the lengths of the three sides of the triangle formed by connecting these three points
- Find and return the area of the triangle formed by these three lines

You may assume that the lines definitely form a triangle (the lines are not parallel).

#5 - `getAllWords`

Can attempt after the Strings and Lists lecture

Write the function `getAllWords(text)` which takes a string (a piece of text) and returns a sorted list of all the words, all in lowercase and with no duplicates. You are guaranteed that the text has no punctuation, and that all words are separated by spaces.

You'll want to solve this problem by using **built-in methods**. Try using the following approach:

- Use a method to make all the text lowercase
- Use a method to split up the text by spaces
- Create a new list that contains each word once (the `in` operator will be useful)
- Use a method to sort the list

#6 - Simple Grid Game

Can attempt after the User Interaction lecture

Using the interaction framework we discussed in class, build a simple grid game that has the following properties.

- An 8x8 grid is shown on the 400px x 400px screen
- A single randomly-chosen cell on the grid is 'selected' (colored yellow); the rest of the cells are left blank
- If the user presses the Up, Down, Left, or Right arrow keys, the selected cell moves one cell in the appropriate direction
- If the selected cell would move off the grid when the user presses an arrow key, the cell does not move
- If the user clicks on the selected cell, it teleports to a random location on the grid. (If the user clicks anywhere else in the grid, nothing happens).

Hint: you can represent a random cell as a random row and a random column.

Another Hint: to check whether the user has clicked on the selected cell, check whether the clicked x and y position falls within the expected left-right and top-bottom bounds of the selected cell.

Logistical note: the starter code for this problem is in the hw3.py file; you can find it by scrolling below the main set of problems.

Core Problems - Programming

#7 - `printPrimeFactors`

Can attempt after Nesting and Top-Down Design lecture

Using the algorithm shown below, write the function `printPrimeFactors(x)` which takes a positive integer x and prints all of its prime factors in a nice format.

A prime factor is a number that is both prime and evenly divides the original number (with no remainder). So the prime factors of 70 are 2, 5, and 7, because 2 * 5 * 7 = 70. Note that 10 is not a prime factor because it is not prime, and 3 is not a prime factor because it is not a factor of 70.

Prime factors can be repeated when the same factor divides the original number multiple times; for example, the prime factors of 12 are 2, 2, and 3, because 2 and 3 are both prime and 2 * 2 * 3 = 12. The prime factors of 16 are 2, 2, 2, and 2, because 2 * 2 * 2 * 2 = 16. We'll display repeated factors on a single line as a power expression; for example, 16 would display 2 ** 4, because 2 is repeated four times.

Here's a high-level algorithm to solve this problem. Start by iterating through all possible factors. When you find a viable factor, repeatedly **divide the number** by that factor until it no longer evenly divides the number. Our algorithm looks something like this:

```
1. Repeat the following procedure over all possible factors (2 to x)
    a. If x is evenly divisible by the possible factor
        i. Set a number count to be 0
        ii. Repeat the following procedure until x is not divisible by the
            possible factor
            1. Set count to be count plus 1
            2. Set x to x divided by the factor
        iii. If the number count is exactly 1
            1. Print the factor by itself
        iv. If the number count is greater than 1
            1. Print "f ** c", where f is the factor and c is the count
```

As an example, if you call `printPrimeFactors(600)`, it should print

```
2 ** 3
3
5 ** 2
```

#8 - `getSecretMessage`

Can attempt after the Strings and Lists lecture

You can hide a secret message in a piece of text by setting a specific character as a key. Place the key before every letter in the message, then fill in extra (non-key) letters between key-letter pairs to hide the message in noise.

For example, to hide the message "computer" with the key "q", you would start with "computer", turn it into "q**c**q**o**q**m**q**p**q**u**q**t**q**e**q**r**", and then add extra letters as noise, perhaps resulting in "orup**qc**rzyp**qo**m**qm**hcy**qp**whh**qu**t**qt**xt**qe**ye**qr**pa". To get the original message back out, copy every letter that occurs directly after the key, ignoring the rest.

Write a function `getSecretMessage(s, key)` that takes a piece of text holding a secret message and the key to that message and returns the secret message itself. For example, if we called the function on the long string above and "q", it would return "computer". You are guaranteed that the key does not occur in the secret message.

Hint: loop over every character in the string. If the character you're on is the key, add the **next** character in the string to a result string.

#9 - `sumAnglesAsDegrees`

Can attempt after the Strings and Lists lecture

When analyzing data, you need to convert the data from one format to another before processing it. For example, you might have a dataset where angles were measured in radians, yet you want to find the sum of the angles in degrees.

Write the function `sumAnglesAsDegrees(angles)` which takes a list of angles in radians (floats) and returns the sum of those angles **in degrees** (an integer). To do this, you will need to change each angle from radians to degrees before adding it to the sum. You can do this with the library function `math.degrees()`. Make sure to round the final result to get an integer answer.

For example, `sumAnglesAsDegrees([math.pi/6, math.pi/4, math.pi])` should convert the radians to approximately `30.0`, `45.0`, and `180.0`, then return `255`.

#10 - Guessing Game

Can attempt after the User Interaction lecture

In the function `guessingGame()`, implement an interactive guessing game that the computer plays with the user. The computer should generate a random number between 1-10 and repeatedly ask the user to guess what it is until the user gets it right.

You can design your guessing game however you like, but it must meet the following constraints:

- The computer must actually pick a random number in the range [1, 10] each time the function is called (don't hardcode it!)
- If the user guesses right, the computer tells the user how many total guesses they made and ends the function
- If the user guesses wrong, the computer tells them whether the number they guessed was too high or too low and makes them guess again
- The program **must not crash** if the user enters an illegal input (a number outside of 1-10, or something that isn't an integer)

For example, here's a possible run of `guessingGame`, with user inputs bolded:

```
Welcome to the guessing game!
I'm thinking of a random number between 1 and 10
Make a guess: five
That's not an integer number! Try again.
Make a guess: 5
Not quite, it's larger than that.
Make a guess: 100
Not quite, it's smaller than that.
Make a guess: 8
You got it! Well done.
You made 4 total guesses.
```

Spicy Problems - Programming

#11 - `mostFactors`

Can attempt after the Nesting and Top-Down Design lecture

Write a function `mostFactors(x, y)` which takes two integers and finds the integer between those two numbers (inclusive) that has the most prime factors. The function should also print a sorted list containing all the prime factors of that number. If two (or more) numbers tie for having the most prime factors, pick the number where the sum of the number's factors is largest.

Example: `mostFactors(100, 110)` would investigate all the numbers between 100 and 110, find that 108 has the most factors (five), print `[ 2, 2, 3, 3, 3 ]`, and return `108`.

Note: when counting factors, we include factors that occur multiple times - for example, 2 divides 108 twice, so it is included twice. We can also include the number itself if the number is prime; however, we do not include 1, as 1 is not prime.

Hint: this problem is easier if you use **top-down design!** What helper function would be most useful to create in this problem? Perhaps something that tells you what the factors of a specific number are…

Note: technically you need lists to solve this problem (which won't be covered until Tuesday's lecture). However, you can solve most of the problem without lists.

#12 - `getCharacterLines`

Can attempt after the Strings and Lists lecture

Assume you're provided a string script that has been formatted in a specific way. Each line of the script begins with a character's name, followed by a colon, followed by their line of dialogue. Lines are separated by newlines, which are represented in Python by the string `'\n'`. For example:

```
'''Buttercup: You mock my pain.
Man in Black: Life is pain, Highness.
Man in Black: Anyone who says differently is selling something.'''
```

Write the function `getCharacterLines(script, character)`, which takes a script and a character name (both strings) and returns a list of the lines spoken by that character. The lines should be stripped of the leading character name and any leading/trailing whitespace. So if we use the following script:

```
'''Burr: Can I buy you a drink?
Hamilton: That would be nice.
Burr: While we're talking, let me offer you some free advice: talk less.
Hamilton: What?
Burr: Smile more.
Hamilton: Ha.
Burr: Don't let them know what you're against or what you're for.
Hamilton: You can't be serious.
Burr: You want to get ahead?
Hamilton: Yes.
Burr: Fools who run their mouths oft wind up dead.'''
```

Then:

```
getCharacterLines(script, "Hamilton") ==
[ "That would be nice.", "What?", "Ha.", "You can't be serious.", "Yes." ]
```

Hint: you'll want to use string and list **methods** and **operations** to make this problem more approachable. Specifically:

- `split` can help you separate the lines of text
- `index` can help you locate where a line of text switches from name to dialogue
- slicing can help you separate the name from the dialogue

#13 - Spicy Grid Game

Can attempt after the User Interaction lecture

Start by going to #6 (Review: Simple Grid Game) and programming the game described there. Once you've done that, add the following features to the game:

- Instead of not allowing the user to move off the board, when the user would move off the board, wrap the selected cell around to the other side of the board (for example, if the cell would move above the top row, wrap it around to the bottom row).
- When the selected cell is clicked and teleports to a new location, color the space it was in before green. This should happen every time the square is clicked (so if the selected cell is clicked in four locations, all four of those locations should be colored green).
- Ensure that when the selected cell teleports, it does **not** teleport to a location that has already been visited (is colored green).
- Ensure that if the selected cell would move on top of a visited cell (green cell) on responding to an arrow key press, it does not move instead.
- When every cell on the grid is colored green, the game displays a message congratulating the user on the screen. The game should no longer be playable at this point.

Hint: you can implement wraparound with if statements, but it may be easier to implement by using the mod operator.

Another Hint: to make sure that you teleport the selected cell to an open location, just repeatedly generate new positions until you find one that wasn't already visited.

Logistical note: the starter code for this problem is in the hw3.py file; you can find it by scrolling below the main set of problems.

Bonus Problems

Advanced Programming - Animation

Using the animation framework covered in the bonus slides, create a simple version of the classic arcade game Snake!

If you've never played Snake before, start by learning how the game works. Google has a simple version you can play: https://www.google.com/search?q=snake+game

Your Snake game should be played on an 8 x 8 grid. The snake can be represented as a connected sequence of green cells; the food can be represented as a red circle in a cell.

Your game should do the following:

- Display the grid, food, and snake.
- Automatically move the snake one cell in the 'forward direction' over a regular time interval (we recommend setting the time rate to 0.4 for a playable game).
- Allow the user to change the 'forward direction' by pressing the up, down, left, and right arrows.
- Allow the snake to 'eat' the food when the head of the snake overlaps the food (by extending the snake body by one length, and placing the food in a randomly-chosen empty cell on the grid).
- End the game when the head of the snake goes off the board or intersects with the body of the snake. The user should no longer be able to play the game at this point.

Hint: the hardest part of Snake is determining how to move the body. Here's a handy trick! Represent the snake as a list of body cell positions ([row, col] lists). To move the snake forward, just add a new cell (the new head) to the front of the list, and remove the cell at the back of the list! If you need to 'grow' the snake, just add the head without removing the last cell.

Logistical note: this work should be submitted to Hw3 - Bonus, not as part of the hw3.py file.

Advanced Computer Science 1 - Efficiency

For the following function, describe an input that would result in **best-case efficiency,** then describe an input that would result in **worst-case efficiency**. This generic input must work at **any possible size**; don't answer 1, for example.

```
def isPrime(num):
    for factor in range(2, num):
        if num % factor == 0:
            return False
    return True
```

Best Case input:	
Worst Case input:	

Advanced Computer Science 2 - Efficiency

For each of the following functions, check the **Big-O function family** that function belongs to. You should determine the function family by considering how the number of steps the algorithm takes grows as the size of the input grows.

```
def countEven(L): # n = len(L)
  result = 0
  for i in range(len(L)):
    if L[i] % 2 == 0:
      result = result + 1
  return result
```

- ☐ O(1)
- ☐ O(logn)
- ☐ O(n)
- ☐ O(nlogn)
- ☐ O(n^2)

```
# n = len(L)
def sumFirstTwo(L):
  if len(L) < 2:
    return 0
  return L[0] + L[1]
```

- ☐ O(1)
- ☐ O(logn)
- ☐ O(n)
- ☐ O(nlogn)
- ☐ O(n^2)

```
# n = len(L1) = len(L2)
def binarySearchAll(L1, L2):
  count = 0
  for item in L1:
    # Hint: what's the complexity of
    # binary search?
    if binarySearch(L2, item) == True:
      count = count + 1
  return count
```

- ☐ O(1)
- ☐ O(logn)
- ☐ O(n)
- ☐ O(nlogn)
- ☐ O(n^2)

```
# n = len(L); original call has i = 0
def recursiveSum(L, i):
  if i == len(L):
    return 0
  else:
    return L[i] + recursiveSum(L, i+1)
```

- ☐ O(1)
- ☐ O(logn)
- ☐ O(n)
- ☐ O(nlogn)
- ☐ O(n^2)