CSSE120 Sequences and mutation

Name: ______ **SOLUTION** ________ CM: ____________Section: _____ Grade: ______of 10

1. The following two functions both return the list `[1, 2, 3, ... n]`, for the given ***n***. They are the same except for the bold-italicized lines.

```
def using_concatenation(n):
    new = []
    for k in range(1, n + 1):
        new = new + [k]
    return new
```

```
def using_append(n):
    new = []
    for k in range(1, n + 1):
        new.append(k)
    return new
```

With your instructor: open today's project and examine module **m0r_concatenation_vs_append**. Per the instructions in that module, read the code, run the module, and answer the questions in it (with your instructor's help as needed). [My computer: **150, 345, 880 & 1485**, for the 4 questions.]

Then ***circle*** which of the above implementations ***is better***. ***Why*** **is it better?** The one on the ***right*** is better because it is **MUCH faster** (and uses much less space).

2. Continuing the previous problem, circle ***True*** or ***False*** for each of the following.

Each time through the loop:

a. The implementation on the ***left*** ** ***mutates*** ** **new**. True or ***False***

b. The implementation on the ***left*** ** ***re-assigns*** ** **new**. ***True*** or False

c. The implementation on the ***right*** ** ***mutates*** ** **new**. ***True*** or False

d. The implementation on the ***right*** ** ***re-assigns*** ** **new**. True or ***False***

3. Consider the code below.

```
def increment_last_number(numbers):
    new = []
    for k in range(len(numbers)):
        new.append(numbers[k])
    new[len(new) - 1] = new[len(new) - 1] + 1
    return new
```

```
def main():
    r = [4, 20, 6, 10]
    s = increment_last_number(r)
    print(r)
    print(s)
```

When ***main*** runs, what does it print? `[4, 20, 6, 10]` followed by `[4, 20, 6, 11]`

4. The function in the previous problem returned a new list that is a copy of the given list, except that the last number in the list is incremented by 1. Write the code for a ***mutate_last_item*** function that ***mutates*** its given list of numbers so that the last number in the list is incremented by 1. (Hint: it is a one-liner!)

```
def mutate_last_number(numbers):
    numbers[len(numbers) - 1] = numbers[len(numbers) - 1] + 1
or  numbers[-1] = numbers[-1] + 1
or  numbers[-1] += 1
```

5. What advantage does ***increment_last_number*** have over ***mutate_last_number***? ***Safer*** (does not modify its argument).

6. What advantage does ***mutate_last_number*** have over ***increment_last_number***? **Runs MUCH faster.**

7. Which of the following are patterns that the video presented for iterating through items in a sequence? Check all that apply. ***All of them should be checked.***

______ Beginning to end

______ Two places at once

______ Selecting items

______ Parallel sequences

______ Finding something

______ Max or min

8. Complete the implementation of the following function:

```
def get_max(numbers):
    """ Returns the largest number in the given non-empty list. """
    biggest = numbers[0]           or    index = 0
    for k in range(1, len(numbers)):     for k in range(1, len(numbers)):
        if numbers[k] > biggest:             if numbers[k] > numbers[index]:
            biggest = numbers[k]                 index = k
    return biggest                       return numbers[index]
```

9. Suppose that instead of the largest number in the given non-empty list (as in the previous problem), you wanted to return the largest number ***at an odd index (position)*** in the given non-empty list. What change(s) would you make to the code in your answer to the previous problem?

Change the starting place to ***index 1*** (instead of index 0) and change the range to **`range(3, len(numbers), 2)`** [Note: starts at ***1 (or 3)***, goes up by 2]

Or, you could leave the range as is and check inside the loop if **`k`** is odd, but that runs twice as slowly.

10. Suppose that you wanted to find the largest ***positive*** number in a given non-empty list. That is a much harder problem than either of the preceding problems. Why?

You have to find a positive number in the list (or determine that there is no positive number) to use as a “starting point” for your largest-positive-in-list, or otherwise deal with the “starting point”.

11. What is the output of the following code?

```
def mystery(s):                           cs
    for k in range(1, len(s)):            ss
        print(s[k-1], s[k])               se
                                          e1
mystery('csse120')                        12
                                          20
```

12. Write one line of code to print both the first and last characters in the string variable called **`clown`**.

`print(clown[0], clown[len(clown) - 1])` *or* the 2nd item could be `clown[-1]`

13. Write a single line of code that has approximately the same effect as **`nums = nums + [17]`**, but ***mutates*** the **`nums`** list instead of re-assigning it. `nums.append(17)`

14. Search online for “list remove python” to try to find the 3 functions/methods to remove an item from a list. List the names of those 3 functions/methods below. Then search for the Stack Overview post titled

"Difference between ___, ___ and ___ on lists" (but replacing the underscores with the 3 names you found) and read its excellent explanation for the differences between the 3 functions/methods. `remove` `del` `pop`

From: https://stackoverflow.com/questions/11520492/difference-between-del-remove-and-pop-on-lists/11520540

`remove` removes the *first* matching ***value***, not a specific index:

```
>>> a = [0, 2, 3, 2]
>>> a.remove(2)
>>> a
[0, 3, 2]
```

`del` removes the item at a specific **index**:

```
>>> a = [3, 2, 2, 1]
>>> del a[1]
>>> a
[3, 2, 1]
```

`pop` removes the item at a specific **index** *and returns it*.

```
>>> a = [4, 3, 5]
>>> a.pop(1)
3
>>> a
[4, 5]
```