

CIS 210
Fall 2013 Midterm

Write your name at the top of **each page** before you begin. [5 points]

1. [5 points] What does `q1( )` print?

```
def q1():
    count = 0
    for x in ['a', 'b', 'c']:
        for y in ['b', 'c', 'd']:
            if x == y:
                count += 1
    print(count)
```

2

Note that the nested loops will test all combinations of the first list with the second. Two of the elements will match: *b* with *b*, and *c* with *c*.

2. [5 points] What does `q2( )` print?

```
def xcount(s):
    """Sorry, no docstring to explain it"""
    count = 0
    for ch in s:
        if ch == "x":
            count += 1
    return count
```

```
def xavg(txt_list):
    nrows = len(txt_list)
    sum = 0
    for line in txt_list:
        sum += xcount(line)
    avg = sum / nrows
    return avg
```

```
def q2():
    print( xavg(["abxxcdxx", "xxm", "abcd"]))
```

2

xcount counts the number of *x* in a string. It counts 4 *x* in the first element of the list, 2 in the second, and 0 in the third. $(4 + 2 + 0)/3 = 2$.

3. [5 points] What does `q3( )` print?

```
def clamp( ar, floor ):
    for i in range(len(ar)):
        if ar[i] < floor:
            ar[i] = floor

def q3() :
    temps = [ 30, 31, 32, 34, 34, 34 ]
    clamp(temps, 32)
    ntemps = len(temps)
    sum = 0
    for tmp in temps:
        sum += tmp
    avgtemp = sum / ntemps
    print(avgtemp)
```

33

`clamp` changes the `temps` array to `[32, 32, 32, 34, 34, 34]`, so `q3` calculates $(32 + 32 + 32 + 34 + 34 + 34)/6$. Regroup as $3(32 + 34)/6 = (32 + 34)/2 = 66/2 = 33$, or just eyeball the array to see that an equal number of 32 and 34 must average 33.

4. [5 points] What does `q4( )` print?

```
def periscope(x,y):
    x = 2 * x
    y = 2 * y
    return x - y

def q4():
    x = 7
    y = 5
    z = periscope(x,y)
    print(x + y + z)
```

16. This point of this question was primarily to see if you understood scope and parameter passing. Function `periscope` has its own `x` and `y`, which are initially copies of the values passed from function `q4`. It returns the value 4, but does not alter the values of `x` and `y` in `q4`. The value printed is therefore $7 + 5 + 4 = 16$. (Contrast this to `q3`, in which `clamp` is passed a reference to the list from `q3`, and is able to change the list value shared with `q3`.)

5. [12 points] In our project flooding the cavern, we used a list of lists to represent a rectangular grid of cells. Here we test a list of lists to see if it could represent a rectangular grid. Finish the function `is_rectangular`, consistent with its docstring.

```
def is_rectangular(grid):
    """
    Check whether the rows of s are all the same length.
    Args:
        grid: A list of lists, e.g., [ [ 1, 2, 3], [4, 5, 6] ]
    Returns:
        True if each of the rows of grid have the same number of elements.
    Examples:
        is_rectangular([[ 1, 2 ], [3, 4], [5, 6]]) = True
        is_rectangular[[ 1, 2 ], [3, 4, 5], [6, 7]]) = False
        is_rectangular([[ ], [ ]]) = True
        is_rectangular([ ]) = True
    """
    nrows = len(grid)
    if nrows == 0:
        return True
    rowlen = len(grid[0])
    for row in grid:
        if len(row) != rowlen:
            return False
    return True
```

There are of course variations on this. One of them eliminates the special case for `nrows == 0`, instead comparing to `grid[0]` within the loop body ... which might be just slightly slower but is definitely more elegant!

6. [13 points] Finish the function `longest_dup` below, consistent with its docstring.

```
def longest_dup(s):
    """
    Measure the longest sequence of a single character in s.
    Args:
        s: A string containing only letters
    Returns:
        An integer, which is the length of the longest substring in
        s made up of all the same character.

    Examples:
        longest_dup("abcccdef") = 3
        longest_dup("aaabbcccd") = 3
        longest_dup("aaaxaaaxaaa") = 3
        longest_dup("abcde") = 1
        longest_dup("") = 0
    """
    longest = 0
    this_chain = 0
    prev = ""
    for letter in s:
        if letter == prev:
            this_chain += 1
        else:
            prev = letter
            this_chain = 1
        if this_chain > longest:
            longest = this_chain
    return longest
```

Again, there are many possible variations. Correct versions inevitably need to keep two different variables: One recording the length of the chain the current sequence (`this_chain`, in my solution), and another recording the length of the longest chain observed so far (`longest` in my solution). The test for a new longest sequence must either take place every time through the loop (as in my code above), or else it needs to be checked one more time after the loop. If it is checked only when the current letter does not match, it won't be updated at the end of a string like "abcxxx".

A different approach is to use nested loops, searching forward to determine the length of a sequence starting from each character. This is much less efficient, but I didn't mark points off for that. It's also harder to get right.