CIS 210
Fall 2013 Final Exam Key

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

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

```
def distinct(txt):
    unique = set()
    count = 0
    for w in txt.split():
        if not (w in unique):
            unique.add(w)
            count += 1
    return count

def q1():
    print( distinct("sauce for the goose sauce for the gander"))
```

5

This is a count of unique words. I gave partial credit to those who counted unique characters (a little less if you counted letters but not the spaces). Note that the meaning of split( ) and a short description of the set data structure was included on the quick-reference sheet I handed out.

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

```
def amplify(li, n):
    for i in range(len(li)):
        li[i] = n * li[i]

def q2():
    x = [3, 4, 5]
    amplify(x,2)
    sum = 0
    for item in x:
        sum += item
    print(sum)
```

24

This question was mainly about using a function to modify a list (because we pass a reference to the list, rather than copying it).

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

```
def addlist(a, b):
    for i in range(len(a)):
        if i < len(b):
            a[i] += b[i]

def q3():
    x = [1, 2, 3]
    y = [3, 2, 1]
    z = x
    addlist(x, y)
    sum = 0
    for item in z:
        sum += item
    print(sum)
```

12

The point of this question is shared references. I wanted to make sure you understood that z and x are references to the same object. The comparison to len(b) in addlist is to ensure that references to $b[i]$ are not out of range.

4. [10 points] For this problem, you may find it useful to draw a simple map or diagram of a swamp. What does q4( ) print?

```
class Monster:
    def __init__(self, name, x, y, reach):
        self.name = name
        self.xcoord = x
        self.ycoord = y
        self.reach = reach

    def in_reach(self, x, y):
        dx = self.xcoord - x
        if dx < 0:
            dx = 0 - dx
        dy = self.ycoord - y
        if dy < 0:
            dy = 0 - dy
        return self.reach >= dx + dy

class Swamp:
    def __init__(self):
        self.monsters = [ ]

    def add(self,m):
        self.monsters.append(m)

    def safe(self, x, y):
        for m in self.monsters:
            if m.in_reach(x,y):
                return False
        return True

def q4():
    sw = Swamp()
    sw.add( Monster("Grendel", 0,0,  1))
    for x in range(2):
        for y in range(2):
            if sw.safe(x,y):
                print("Safe spot:", x, y)
```

Safe spot: 1 1
A common error was printing safe spots at coordinates including 2, in addition to the safe spot at 1,1. Since this was clearly just a misinterpretation of range(2), I only took off 2 points for that. Note that reach is calculated in "Manhattan distance." Some people seem to have used Euclidean distance, either not reading the code of in_reach or misinterpreting it.

(score)

5. [10 points] Class `Scores` is part of a class record for keeping homework scores. Complete the `average` method.

```
class Scores:
    """Project scores in Snowman Construction 101"""
    def __init__(self):
        """New record of scores.  None recorded yet"""
        self.project_scores = [ ]

    def record(self,score):
        """Record a homework score.
        Args:
            score:  integer, the new homework score to record.
        """
        self.project_scores.append(score)

    def average(self):
        """Calculate average (arithmetic mean) of recorded homework scores.
        Args:
            none  (uses previously recorded scores)
        Returns:
            Arithmetic mean of recorded scores, that is,
            sum of scores / number of scores.  If no scores
            have been recorded, return 0.
        """
        # Your code here

        if len(self.project_scores) == 0:
            return 0
        sum = 0
        for score in self.project_scores:
            sum += score
        return sum / len(self.project_scores)
```

This is obviously checking to see if you understand how classes and objects work, including how the fields ("attributes") in objects are accessed. Most of you didn't have trouble with it.

(score)

6. [15 points] Finish the function `uniform` below, consistent with its docstring. You may write additional functions to simplify your code.

```
def uniform(li):
    """Determines whether all the rows of integers in li have the same sum.
    Arguments:
      li: A list of lists of integers
    Returns:
      True if each of the lists in li has the same sum; False otherwise
    Examples:
       uniform( [[50, 50], [100, 0], [25, 25, 25, 25]] ) = True
       uniform( [[ -5, 5, 0 ], [13, -7, -6], [ ]] ) = True
       uniform( [[ 192, 344, 17]] ) = True
       uniform( [ ] ) = True
       uniform( [[ 7, 3], [5, 2]] ) = False
       uniform( [[17], [ ]] ) = False
    """
    ### Your code here
    if len(li) == 0:
        return True
    std = sum(li[0])
    for row in li[1:]:
        if sum(row) != std:
            return False
    return True

def sum(li):
    """Sum of integers in list li.
    Args:
       li:  A list of zero or more integers
    Returns:
       sum of items in li
    """
    result = 0
    for item in li:
        result += item
    return result
```

This question was to check your understanding of array (list) accesses and loop idioms, specifically the *for all x in li: p(x)* idiom. I also wanted you to handle the special case of an empty list correctly. Some of you remembered that Python has a built-in sum function, which I forgot ... so you didn't have to write the sum function above. Some of you are still getting mixed up between looping through indexes (*for i in range(len(li)):*) and looping through elements (*for elem in li:*).

(score)

7. [20 points] In the following question, add a number of days to a date. For example, January 15 + 20 days is February 4. For the sake of simplicity we ignore leap year. Finish the function.

```
MONTHS = ["X","Jan","Feb","Mar","Apr","May","Jun","Jul","Aug","Sep","Oct","Nov","Dec"]
DAYS_IN_MONTH = [ 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 ]

def days_ahead( start_month, start_day, ahead ):
    """Calculate start date + days ahead (ignoring leap year).
    Args:
        start_month: integer month number, 1..12
        start_day: integer day of month, 1..DAYS_IN_MONTH[start_month]
        ahead:  integer, number of days ahead
    Returns: string consisting of 3-letter month name and day, which is start month
        and day + ahead.  (We treat February as always having 28 days.)
    Examples:
        days_ahead(1, 31, 0) = "Jan 31"
        days_ahead(1,15, 20) = "Feb 4"
        days_ahead(11,20,90) = "Feb 18"
    """
    # Your code here
    day = start_day + ahead
    month = start_month
    while day > DAYS_IN_MONTH[month]:
        day -= DAYS_IN_MONTH[month]
        month += 1
        if month > 12:
            month = 1
    return "{} {}".format(MONTHS[month], day)
```

There were many possible correct solutions, more almost correct solutions, and a vast number of ways to get this one wrong, so grading was a challenge. I expected this one to be tough. I was pleasantly surprised that many of you got it right or almost right, and some solutions were better than the one I originally wrote — the solution above is improved based on what I saw in some of the cleanest of the student solutions. No special cases at all!

Two major variations were using recursion instead of a loop, or having a loop that counts *days_ahead* down one day at a time, incrementing the month when needed. Common mistakes included not resetting the month to 1 (January) when you go past 12 (December), making a recursive call but not doing anything with the results, or putting operations in the wrong order so that days of the wrong month was used.

(score)