Name:_____ **Date:**____/____/_____

CSCSCI-UA.0002 – Midterm – Practice Questions on Functions and Loops

1. There's an error in both code samples below. Circle the line where the error occurs. To the right of each code sample, explain why there's an error: (3 points)

a)
```
a = greet_with_hello('yo yo meow')

def greet_with_hello(name):
    print('hello %s' % (name))
```
`greet_with_hello is not yet defined`

b)
```
for outer in range(5):
    print(inner)
    for inner in range(5):
        print(outer)
```
`inner not yet defined`

c)
```
limit = 5.0
total = 0
for i in range(limit):
    total += i
```
`range only accepts floats`

2. Using the code in the 1st column, answer the questions in the second and third columns. If the question asks for output, **error** or no output is always a possible answer.

Code	Question #1	Question #2

`def say_cheese(n):`
`  s = n * 'cheese'`
`  print(s)`
`talk = say_cheese(3)`
	What is the **output** of this program?
	`cheesecheesecheese`
		What is the **value** of the variable called **talk**?
		`None`

`exclamation = 'boo'`
`whisper = 'shhh'`
`def do_something():`
`  exclamation = 'bye!'`
`  print(whisper)`
`do_something()`
`print(exclamation)`
	What is the **first** line of **output** for this program?
	`shhh`
		What is the **second** line of **output** for this program?
		`boo`

`def join_three(a, b, c):`
`  return '%s, %s, %s' % (a, b, c)`
`c, b, a = 3, 2, 1`
`res = join_three(c, b, a)`
`print(res)`
	What is the **output** of this program?
	`3, 2, 1`
		What data **type** is returned from join_three?
		`string`

`def gimme_a_break():`
`  print('OK')`
`  for i in range(3):`
`  print(i)`
`  break`
`  print(i, 'again')`
`  print('WAT')`
 	
`res = gimme_a_break()`
`print(res)`
`print('FIN')`
	What is the **output** of this program?
	`OK`
	`0`
	`WAT`
	`None`
	`FIN`
		What is the **output** of this program if `break` were replaced with `continue`?
		`OK`
		`0`
		`1`
		`2`
		`WAT`
		`None`
		`FIN`

`def do_this_thing(limit):`
`  print('hmmmm')`
`  for i in range(limit):`
`  print(i)`
`  return i`
`  print('YES!')`
 	
`value = do_this_thing(3)`
`print(value)`
`print('done')`
	What is the **output** of this program?
	
	`hmmmm`
	`0`
	`0`
	`done`
		Draw an arrow to the **argument**, and draw an arrow to the **parameter** for the definition and execution of `do_this_thing`.
		
		`limit is the parameter`
		`3 is the argument`

3. Write a function called `distance`. It should have 4 parameters: the x and y values of one point on the coordinate plane and the x and y values of a second point on the coordinate plane. It should calculate the distance between both points and return the resulting value.

The distance formula is: $d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$

```
import math
def distance(x1, y1, x2, y2):
    d = math.sqrt((x2 - x1) ** 2 + (y2 - y1) ** 2)
    return d
```

4. Assuming that the distance formula specified in #3 exists, use it in a program that asks for the x and y values of two points and then prints out the result with exactly 2 decimal places:

```
enter x1 > 2
enter y1 > 5
enter x2 > 9
enter y2 > 1
The distance between these two points is  8.06
```

```
p1_x = int(input('Enter x1 >'))
p1_y = int(input('Enter y1 >'))
p2_x = int(input('Enter x2 >'))
p2_y = int(input('Enter y2 >'))
result = distance(p1_x, p1_y, p2_x, p2_y)
print('The distance between these two points is %s' % format(result, '.2f'))
```

5. Create your own function to approximate the square root of a number. Use Newton's method to do so:

   a) to take the square root of a number, n...
   b) start with almost any approximation (for example, our first guess may just be n/2)
   c) find a better approximation by using the following formula:
   d) new_approximation = ½ * (approximation + n / approximation)
   e) continue to find a better approximation until the difference between the new approximation and the old is very small (for example, < 0.01)

```
def newtons(n):
    threshold = 0.01
    guess = n
    new_guess = n / 2
    while abs(new_guess - guess) > threshold:
        print(guess, new_guess)
        guess = new_guess
        new_guess = 0.5 * (guess + n / guess)
    return new_guess
```

6. Write a function, **dec_to_bin**, that converts a decimal number (an int) into a binary number (represented as a string). In both versions, return an empty string if the number is not within 0 and 255. Otherwise, return the binary number as a string.

   a) It has one **parameter**, the decimal number to convert, and it **returns** a string, the equivalent binary number
   b) continually subtract powers of 2 (starting with 2 ^ 7) from the decimal number as long as that power of 2 is less than or equal to the decimal number
   c) if you subtract a power of 2, then count that place (that bit) as 1
   d) if the power of 2 is greater than the decimal value, then don't subtract and count the place (bit) as 0
   e) example: convert 131 →
      - 128 (2 ^ 7) is less than 131, so subtract it and record a 1 for that place: decimal number is now 3, and binary is 1
      - all other powers of 2 below 2 ^ 7 up intil 2 ^ 1 are more than the decimal number, so records 0's for those: 100000
      - 2 (2 ^ 1) is less than 3, so subtract it and record a 1 for that place: decimal number is now 1, and binary is 1000001
      - finally, 1 is left, and that can be added as the last bit: 10000011
   f) example usage: print(dec_to_bin(6)) → '00000110'

```
def dec_to_bin(n):
    bin_num = ''
    max_exp = 7
    for exp in range(max_exp, -1, -1):
        place = 2 ** exp
        if place <= n:
            n -= place
            bin_num += '1'
        else:
            bin_num += '0'

    return bin_num
```

7. Write a function called **`mult_table`** using the requirements and sample output below. It should have:

- **1 parameter**, an integer, that specifies the dimensions of the table it produces
- a r**eturn value** that's a string string containing a table
- the table should contain the **product** of the **row** number and **column** number in each cell; for example, in row 3, column 2, the value is 6 because it is the result of multiplying 3 * 2 (the row and column numbers)
- the width of each cell should be dynamic depending on the number of characters of the longest product plus 1 to accommodate a space between cells (in the example, the width of the longest product, 49, is 2...with 1 added for a space, for a total of 3)

```
print(mult_table(7))
```

```
1  2  3  4  5  6  7
2  4  6  8 10 12 14
3  6  9 12 15 18 21
4  8 12 16 20 24 28
5 10 15 20 25 30 35
6 12 18 24 30 36 42
7 14 21 28 35 42 49
```

```
def mult_table(n):
    cell_width = len(str(n * n)) + 1
    table = ''
    for row_num in range(1, n + 1):
        row = ''
        for col_num in range(1, n + 1):
            row += format(row_num * col_num, '>%s' % cell_width)
        table += row + '\n'
    return table
```