Name: ________________________________
Net ID ________________

There are 2 sections, the first section is worth 50 points and the second section is worth 50 points, for a total of 100. Answer all 4 questions in Section 1. Answer 2 out of the 3 questions in Section 2.

It is essential that you PUT YOUR NAME AND NET ID ON ALL TEST MATERIALS. It can be difficult to identify the author of an unsigned test and it would be better to avoid this problem.

There is a GLOSSARY OF TERMS at the end of the test. Please feel free to look up some of the basics in this glossary. I will also answer any reasonable look-up style questions as I am more interested in your ability to reason than your ability to memorize.

Section 1: (50 points) Each example consists of Python code, followed by questions and places to fill in answers. Please read all questions carefully and answer them as precisely as possible. Assume that there are no bugs in the code that will make the system crash, although the code may not solve a problem perfectly. If you find anything that you think is a bug, there is either a typo (and I should fix it for everyone) or you are mistaken.

Sample Question A:

```
output = '1'+'1'
```

Question: What is the value of the variable `output`?

Answer: '11'

Note: Attention to detail is important. The quotes indicate that it is a string. Partial credit is possible. For example, leaving out the quotes would have lost just a little bit, but answering 2, would have resulted in an incorrect answer.

Question 1

```
def turtle_line_up():
    import turtle
    global my_turtle1
    global my_turtle2
    global my_screen
    my_screen = turtle.Screen()
    my_turtle1 = turtle.Turtle()
    my_turtle2 = turtle.Turtle()
    my_turtle1.pu()
    my_turtle2.pu()
    my_turtle1.setposition(100,0)
    my_turtle2.setposition(-100,0)
    my_turtle1.pd()
    my_turtle2.pd()
    my_turtle1.setposition(0,0)
    my_turtle2.setposition(0,0)

turtle_line_up()
```

Question: Draw (approximately) the configuration that would be drawn by the turtle program.
Question 2

```python
def word_combo(word1, word2):
    output = ''
    for index in range(len(word1)):
        output += word1[index] + word2[index]
    return(output)

def word_list_combo(wordlist1, wordlist2):
    output = []
    for index in range(len(wordlist1)):
        output.append(word_combo(wordlist1[index], wordlist2[index]))
    return(output)

def main():
    list1 = ['rie', 'sar', 'ars', 'sads']
    list2 = ['asd', 'tis', 'cos', 'trut']
    output = word_list_combo(list1, list2)
    for word in output:
        print(word)

main()
```

Question: Indicate what would be printed out by the code above.
def name_sum(name):
    total = 0
    name = name.lower()
    offset = ord('a')-1
    ## used to figure out numeric value of letter
    ## note that the unicode value of a is 97
    ## so offset should equal 96
    for char in name:
        if (char <= 'z') and (char >= 'a'):
            ## makes sure character is a letter
            unicode_value = ord(char)
            value = unicode_value - offset
            ## 'a' is 97 and offset is 96
            ## so the value for 'a' should be 97-96=1
            total = total + value
    return(total)

value = name_sum('Adam')
print(value)

Question: Indicate what would be printed out by the code above.
Question 4

def vowel_count(word):
    total = 0
    for char in word:
        if char in 'aeiou':
            total = total + 1
        elif char == 'y':  ## let’s count y as half a vowel
            total = total + .5
    return(total)

def vowel_count_sort(inlist):
    pairs = []
    for word in inlist:
        pairs.append([vowel_count(word), word])
    pairs.sort()
    output = []
    for pair in pairs:
        output.append(pair[1])
    return(output)

word_list = ['first','hymn','compromise','polynomial','roadway','yak']
output = vowel_count_sort(word_list)
for word in output:
    print(word,end=' ')
print()
Section 2 (50 points): Answer 2 of the 3 questions in this section. For each question, you do, write a Python program as specified. If you choose to answer all 3 questions, please indicate which ones you would like to count for the test.

**Question 5:** Write a program that returns a list of ingredients for making soup. You can assume that your program has access to the following variables (you do not have to define them in your program):

```python
bases = ['tomato base', 'onion base', 'split pea base', 'lentil base', 'coconut base', 'miso base']
other_ingredients = ['carrots', 'potatoes', 'onions', 'squash', 'beans', 'tofu', 'noodles', 'rice']
spices = ['dill', 'oregano', 'garlic', 'sugar', 'dill', 'cinnamon', 'ginger', 'black pepper', 'cilantro']
```

The output of the program should be a list of lists. Each sublist should contain a number (indicating an amount) and the name of an ingredient. Sample output of the program could be:

```python
```

The program should have the following structure:

1. Initialize a variable `output` as the empty list
2. Set a variable `base` randomly to a member of the list `bases`. This is the base ingredient of the soup. Add a list consisting of 8 and the type of base to output. This indicates that the soup contains 8 units of its base. The sample soup has an onion base.
3. Add a list consisting of a number and the word ’salt’ to the output. The number should be 1 if the user wants the soup to be low salt and 2 otherwise (query the user).
4. If the user wants the soup to be spicy, add 3 units of 'cayenne' (i.e., append [3, 'cayenne'] to the output list).
5. Loop through the list of `other_ingredients`. For each ingredient, there is a 30% chance the ingredient should be added. Randomly generate a number to determine this, e.g., use `random.random()` to generate a number from 0 to 1 and only add the ingredient if the result is less than or equal to .3. if the ingredient gets added, randomly choose an amount from 1 to 3, e.g., notice that the sample soup contains 2 units of squash.
6. Loop through the list of spices. Use approximately the same technique as with the ingredients, accept assume there is a 50% chance for each spice. Also assume that the amount ranges from 1 to 3.
7. Return or print out the resulting list of ingredients for the soup.

The following is a sample interaction with the program, followed by printing each of the ingredients on a separate line.

```python
>>> question_5()
Do you want the soup to be low salt?
Yes or No? yes
Do you want the soup to be spicy?
Yes or No? yes
[8, 'split pea base'],
[1, 'salt']
[3, 'cayenne']
[3, 'carrots']
[1, 'beans']
[3, 'tofu']
[3, 'dill']
[2, 'oregano']
[3, 'garlic']
[1, 'sugar']
[1, 'dill']
[2, 'cinnamon']
[2, 'ginger']
[3, 'black pepper']
```
Question 6: Write a turtle program called *turtleBeans* that draws a number of adjacent circles surrounded by a rectangle. The example picture below is the output of such a program. The main function should take two parameters: *circles* and *radius*. The sample output was produced by the command *turtleBeans(5,50)*. Keep in mind that: (i) the radius of a circle is one half of its diameter; (ii) the height of the surrounding rectangle should be the same as the diameter of the circle; (iii) *turtle.circle(radius)* draws a circle above where the turtle starts out and ends up where it starts after drawing the circle; (iv) the edge of the circle is one radius away from the center of the circle.
**Question 7:** Write a game program called `dice_blackjack` in which a player (or user) plays a game against the computer (aka, the dealer). This is a simpler program than the blackjack program that you did for homework.

In the first round, the player and the dealer each roll two virtual dice (a number from 1 to 6 is returned by `random.randint(1,6)`). Their respective scores are initialized to double the roll of these two dice. For example, if the player rolls a 6 and a 2, their initial score is 16.

The following steps repeat until the player indicates that they do not want any more rolls:

- The player is offered a chance to roll again. The remaining steps are only completed if the player agrees.
- Two dice are rolled.
- The player must choose to (1) add both dice to their score; (2) add only the first die to their score; or (3) add only the second die to their roll. Their score is updated accordingly.
- If the player’s score is greater than 21, they lose.

Unless the player has already lost, the following steps repeat as long as the dealer’s score is below 17 (the loop exits as soon as the dealer’s score reaches 17 or more).

- Only complete the remaining steps if the dealer’s score is less than 17. Otherwise exit the loop.
- Two dice are rolled. The die with the higher score is called `high_die` and the die with the lower score is called `low_die`. (If the dice are the same, then `high_die` equals `low_die`.)
- If dealer’s score + `low_die` is greater than 21, the player wins and the game ends.
- If the dealer’s score + `high_die` is less than or equal to 21, add `high_die` to their score.
- If the dealer’s score + `low_die` is less than or equal to 21, add `low_die` to their score.

Next compare the player and dealer’s score. If the player’s score is higher, the player wins. Otherwise the dealer wins (i.e., the dealer wins if there is a tie).

Two Sample games follow:

```python
>>> dice_blackjack()
Your current score is 16
Would you like a hit?
Yes or No? yes
Dice1 is 3 and Dice2 is 2
Your current score is: 16
Choose 1 if you want to use dice1, 2 if you want dice2, and 3 if you want both. 1, 2 or 3? 3
Your current score is 21
Would you like a hit?
Yes or No? no
You win! Your score: 21 Computer’s score 18
```
Basic Stuff to Look Up for the Test

1. Some Basics

- **return(X)** causes the current function to exit and cause the expression represented by the function call to evaluate as X. For example given the following steps, the value of output would be 5:

  ```python
def add(num1, num2):
    return (num1+num2)
output = add(2, 3)
```

- **print(X)** prints X to the screen. This is only for the benefit of the user. It is not useful for having programs interact.

- The parameters of a function are the local variables inside of the parentheses in the function definition. They are useful when you have functions call functions.

- **input(prompt)** is used to ask a human being a question so that a program can interact with a human being. This is useful when you want a human being to enter information interactively. input statements should be used only when human interaction is appropriate. input statements return a string corresponding to what the user typed in. It may be necessary to convert this string to some other data type, e.g., an integer (with int) or a float (with float).

- The operator + will add two numbers or concatenate two strings

- The operator * will multiple two numbers or print a string some number of times.

- The operator ** will represent exponents, e.g., 5**2 == 25.

2. Division and Modulus

- 5 // 2 == 2
- 5/2 == 2.5
- 5%2 == 1

3. More Math

- round(4.5) == 4, round(4.6) == 5, round(-4.5) == -4, round(-4.6) == -5
- round(2/3,2) == .67 ## The second argument of round indicates number of decimal places. The default is to round to the nearest whole number.
- math.floor(4.9) == 4, math.floor(-4.9) == 5
- math.ceil(4.9) == 5, math.ceil(-4.9) == -4
- math.trunc(4.9) == 4, math.trunc(-4.9) == -4
- math.pi == 3.141592653589793 – a variable for the value of pi

4. sequences

- object made up of other objects in an order
- the function len(sequence) returns the number of items in the sequence
- the operator in tests for membership in sequence, e.g., ('a' in 'abc') would have the value True.
- sequences are used in for loops (see below)
- indices and slices
  - Indices in a sequence are numbers from zero to the length of the string. Zero refers to the position before the first item in the string and the length of the string refers to the position following the last item. Thus each item in the sequence are between two consecutive indices. For example, the subscripted numbers indicate indices for the string `The book: '0T1h2ε3 a4b5ω0σ7k8`. Similarly, the indices in `[0 'The',1 'book',2 is,3 'there',4] indicate positions in the list ['The', 'book', 'is', 'here'}.
sequence[num] indicates an element in a sequence beginning at num (a number from zero to one less than the length of the string), e.g., 'The book'[4] evaluates to 'b'; ['The', 'book', 'is', 'here'][0] evaluates to 'The'.

sequence[num1:num2] indicates a subsequence beginning at position num1 and ending at num2, e.g., 'The book'[4:6] evaluates to 'bo'; ['The', 'book', 'is', 'here'][0:2] evaluates to ['The', 'book'].

Leaving out the number before the colon suggests that a subsequence begins at the beginning of the sequence and leaving out the number after the colon suggests that the subsequence ends at the end of the list. Thus 'The book'[:3] evaluates as 'The' and ['The', 'book', 'is', 'here'][2:] evaluates as ['is', 'here'].

• ranges
  – range(5) is approximately equivalent to [0, 1, 2, 3, 4]
  – range(1, 5) is approximately equivalent to [1, 2, 3, 4]

• Strings
  – an empty string has zero characters ""
  – strings are sequences of characters, e.g., 'Hello World!' consists of the items ['H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd', '!']
  – string1.strip(string2) – removes instances of string2 from beginning and end of string. For example, '***Hello World***.strip('*') will return 'Hello World'.
  – string.split(character) – creates a list by dividing a string at each instance of character. For example, 'Hello World'.split(' ') will return the list ['Hello', 'World'].
  – string.lower() converts string to lower case

• Lists
  – A list is represented by square brackets surrounding a list of objects, divided by commas, e.g., ['A', 'List', 'of', 'Strings']
  – Lists are mutable.
  – You can add an object to the end of a list using the append method. For example, suppose my_list = ['a', 'b', 'c']. Then my_list.append('d') will add 'd' to the end of my_list, setting it to ['a', 'b', 'c', 'd'].
  – You can pop an item off the end of a list with the pop method. pop will return the removed item. For example, if my_list is set to ['a', 'b', 'c', 'd'], then my_list.pop() will return 'd' and shorten the list to ['a', 'b', 'c'].
  – You can combine two lists together using extend. If my_list is set to ['a', 'b', 'c'], then the command my_list.extend(['d', 'e', 'f']) will reset my_list to ['a', 'b', 'c', 'd', 'e', 'f'].
  – The method sort will put a list in order based on how < applies to pairs of elements in the list. This will only work if the members of the list are comparable with each other, e.g., you cannot compare an integer to a string.

5. print
  • sep – separator between items
  • end – printed at the end of print statement

6. for loops
  • First Line: for VARIABLE in SEQUENCE:
  • VARIABLE is set to each item in the sequence one at a time
  • The Indented body repeats once for each item in sequence (for each setting of VARIABLE).
  • It is common to exit a loop of any kind by using a return to exit the function.
• It is common to initialize a variable outside a loop (called an accumulator) that then gets incremented inside the loop.

7. **if** statements

• the first line of an **if** statement consists of **if** BOOLEAN-EXPRESSION:
• the body of text indented under the first line is executed if the BOOLEAN-EXPRESSION evaluates to True
• the **if** statement can be followed by optional **elif** statements of the same form, except that the first line begins with **elif**. Each **elif** statement is only evaluated if the BOOLEAN expressions in the **if** and **elif** statements leading up to this one are False.
• The block of **if** and optional **elif** statements can end with an optional **else** statement. The first line is simply **else**:. The body of text under **else** executes if the Boolean expressions for all previous **if** and **elif** statements in the sequence evaluate to False.

8. logical operators

• **X and Y** returns True only if both X and Y are True
• **X or Y** returns True only if X is True, Y is True or both are True
• **X in Sequence** returns True if X is in a member of a sequence, e.g., 'a' in 'abcdefg' would return True
• **X == Y** returns True if X and Y are the same
• **X != Y** returns True if X and Y are the different
• **X < Y** returns True if X is less than Y
• **X > Y** returns True if X is greater than Y
• **X <= Y** returns True if X is less than or equal to Y
• **X >= Y** returns True if X is greater than or equal to
• **Not X** returns True if X is False
• The operators <, >, <=, >= can apply to non-numeric objects. Characters are compared based on their unicode values (so 'a' < 'b'). True is assumed to be greater than False. Sequences are compared lexicographically – the first item in pairs of sequences are compared first, but if they are equal the second items are compared, and so on.

9. **Turtles**

• Screen and Turtle objects are created using the commands turtle.Screen() and turtle.Turtle().
• The turtle is initially in the center of the screen facing rightward.
• **my_turtle.left(degrees)** – rotates the **my_turtle degrees** left (from its perspective).
• **my_turtle.fd(distance)** – moves the **my_turtle distance** units forward.
• **my_turtle.pu()** – picks the pen up
• **my_turtle pd()** – puts the pen down (ready to write)
• **my_turtle.circle(radius)** – creates a circle with radius radius. The circle will be above the direction the turtle was facing when it started drawing. The turtle will move left and up in a circle and end up in the same place as before.

10. **time.sleep(sec)** – pauses for sec seconds (requires the module sleep to be imported)

11. **random** – the random module

• **random.random()** returns a number between 0 and 1
• **random.randint(num1,num2)** returns a number between num1 and num2 (inclusive).
• **random.choice(sequence)** returns member of sequence.