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:

```python
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 set_up_4_turtles():
    import turtle
    global my_turtle1
    global my_turtle2
    global my_turtle3
    global my_turtle4
    global my_screen
    my_screen = turtle.Screen()
    my_turtle1 = turtle.Turtle()
    my_turtle2 = turtle.Turtle()
    my_turtle3 = turtle.Turtle()
    my_turtle4 = turtle.Turtle()

def three_paths_connected():
    set_up_4_turtles()
    leftword = 0
    for next_turtle in [my_turtle1, my_turtle2, my_turtle3]:
        leftword = leftword + 120
        next_turtle.left(leftword)
        next_turtle.pd()
        next_turtle.fd(100)
    my_turtle4.pu()
    my_turtle4.right(90)
    my_turtle4.fd(100)
    my_turtle4.left(90)
    my_turtle4.pd()
    my_turtle4.circle(100)

three_paths_connected()

Question: Draw (approximately) the configuration that would be drawn by the turtle program.
def add_en(word):
    if word == 'do':
        output = 'done'
    elif word == 'be':
        output = 'been'
    elif len(word) < 3:
        output = '*FAIL*' 
    elif word.endswith('do') and (len(word)>3) and (not word[-4] in 'np'):
        output = word+'ne'
    elif word[-3:] == 'eak':
        output = word[:-3]+'o'+word[-1]+'en'
    elif (word[-2:] in ['de','te']) and (word[-3] in 'aeiou') and  
    (not word[-4] in 'aeiou'):
        output = word[:-1]+word[-2]+'en'
    elif (word[-1]=='e'):
        output = word+'n'
    else:
        output = word + 'en'
    return(output)

for word in ['redo','be','speak','freeze','write','arise']:
    print(word,add_en(word))

Question: Indicate what would be printed out by the code above.
def analyze_word(word):
    lower = 0
    upper = 0
    number = 0
    unknown = 0
    end_punctuation = False
    begin_punctuation = False
    for character in word:
        if character in 'ABCDEFGHIJKLMNOPQRSTUVWXYZ':
            upper += 1
        elif character in 'abcdefghijklmnopqrstuvwxyz':
            lower += 1
        elif character in '0123456789':
            number += 1
        elif character in ',.?"\';:-':
            if character == word[0]:
                begin_punctuation = True
            if character == word[-1]:
                end_punctuation = True
            # mid-word punctuation is ignored
        else:
            unknown += 1
    if upper > 0:
        if lower > 0:
            return('captialized',begin_punctuation,end_punctuation)
        elif (upper>1) and (lower == 0):
            return('allcaps',begin_punctuation,end_punctuation)
        else:
            return('abbreviation',begin_punctuation,end_punctuation)
    elif lower > 0:
        return('lower',begin_punctuation,end_punctuation)
    elif number > 0:
        return('number',begin_punctuation,end_punctuation)
    else:
        return('other',begin_punctuation,end_punctuation)

def sentence_analysis(sentence):
    words = sentence.split(' ')
    analysis = []
    num = 0
    for word in words:
        num = num + 1
        word_result,begin_punct,end_punct = analyze_word(word)
        if begin_punct:
            print('** Begin punctuation **
print('Number:',num,'Word:',word,'Type:',word_result)
        if end_punct:
            print('** End punctuation **

sentence_analysis('Somebody said @#$%ˆ& * 5 times in one day.')

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

def sort_string(instring):
    string_list = list(instring)
    string_list.sort()
    new_string = ''.join(string_list)
    # join joins a list of strings together to form
    # a single string, e.g., ''.join(['abc', 'e', 'f', 'G']) would return
    # 'abcefG'
    return(new_string)

def double_sort(string_list):
    output = []
    for input_string in string_list:
        output.append(sort_string(input_string))
        output.sort()
    return(output)

big_list = ['XX OOX X', 'XX O XOO ', 'O XOO O 0X', 'X0XXX', 'O0X000 00']
for new_string in double_sort(big_list):
    print(new_string)

Question: Indicate what would be printed out by the code above.
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 using the turtle module that draws an arrow inside of a circle. The arrow should bisect the circle. The function should take one argument: length.

The arrow should consist of 3 lines: 2 short lines forming a 100 degree angle that is bisected by the third and longest line: the shaft. The shaft should have a length of length. The short lines should each have a length of length/3.

The circle should have a radius of 1/2 of length (a diameter equal to length). In fact, the arrow’s shaft should extend along one diameter of the circle.

For example, the command draw_arrow_inside_circle(100) should generate the following picture:

![Diagram of an arrow inside a circle with the shaft extending along a diameter.](image-url)
Question 6: Write a program that takes a list of lists as an argument parameter (called `vector_list`) and re-orders the `vector_list` according to the sums of the numbers in these lists. The list with the smallest sum should be first, the list with second smallest sum should be second, etc. For example, running the program as shown should produce the following results:

```python
>>> recipes = [[5, 4, 10], [10, 2, 1], [4, 7, 0]]
>>> score_and_sort(recipes)
[[4, 7, 0], [10, 2, 1], [5, 4, 10]]
>>> stories = [[100, 5, 20, 30], [100, 20, 10, 105], [40, 70, 19, 100]]
>>> score_and_sort(stories)
[[100, 5, 20, 30], [40, 70, 19, 100], [100, 20, 10, 105]]
```

Note that the function `sum` takes a list of numbers as an argument and returns the sum of those numbers, e.g., `sum([100,5,20,30])` returns 155. Remember that `list.sort()` puts a list in sort order.
Question 7: Write a dice rolling game called `race_to_100` in which a player plays against the computer program. At the beginning of the game, the player and the computer each roll the computer equivalent of a 6 sided die (`random.randint(1,6)`). The game ends either when:

- at least one of the players (computer or human) has 100 points or more.
- at least one of the players has 0 points or less.

At the end of the game, the player with the highest points wins. In each round after the first, the human player can choose how the next die roll will be applied to the current scores. They can choose for the next roll to be: 1) added to the current roll; 2) multiplied by the current roll or 3) subtracted from the current roll. A sample game follows:

```python
>>> race_to_100()
Initial rolls are: you 4 computer 6
Should the next roll be: 1) multiplied, 2) added or 3) subtracted
Choose 1 or 2 or 3: 2
You rolled 5 and the computer rolled 4
The new scores are: you 9 computer 10
Should the next roll be: 1) multiplied, 2) added or 3) subtracted
Choose 1 or 2 or 3: 3
You rolled 6 and the computer rolled 2
The new scores are: you 3 computer 8
Should the next roll be: 1) multiplied, 2) added or 3) subtracted
Choose 1 or 2 or 3: 2
You rolled 5 and the computer rolled 4
The new scores are: you 8 computer 12
Should the next roll be: 1) multiplied, 2) added or 3) subtracted
Choose 1 or 2 or 3: 2
You rolled 6 and the computer rolled 1
The new scores are: you 14 computer 13
Should the next roll be: 1) multiplied, 2) added or 3) subtracted
Choose 1 or 2 or 3: 2
You rolled 4 and the computer rolled 1
The new scores are: you 24 computer 19
Should the next roll be: 1) multiplied, 2) added or 3) subtracted
Choose 1 or 2 or 3: 2
You rolled 6 and the computer rolled 5
The new scores are: you 81 computer 45
Should the next roll be: 1) multiplied, 2) added or 3) subtracted
Choose 1 or 2 or 3: 2
You rolled 4 and the computer rolled 6
The new scores are: you 101 computer 60
You won in 14 turns!
```
Glossary for Python 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 repeat 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 sequence. 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*: [0 'T', 1 'h', 2 'e', 3 'b', 4 'o', 5 'o', 6 'r', 7 'k', 8 's']. Similarly, the indices in [0 'The', 1 'book', 2 'is', 3 'there', 4] indicate positions in the list ['The', 'book', 'is', 'here'].

negative indices can be used to count positions from the end. Thus -1 is equivalent to the position at one less than the length of the string; -2 is equivalent to the position at two less than the length of the string; etc. The the negative positions around The book would be labeled as follows: 'T_h_e_o_k'.

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'. The negative indices can be similarly applied, e.g., 'The book'[-1] evaluates to 'k', the last character in the string.

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 define a sequence of numbers based on the length of a sequence starting from 0. If given 2 arguments, the second argument is the length of a sequence starting from 0 and the first argument is a starting point within that sequence.
  - 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 instances of character. For example, 'Hello World'.split(' ') will return the list ['Hello', 'World'].
  - string.lower() converts string to lower case; string.upper() converts a string to upper case.
  - string.index(item) returns the position index item occurs in the list – it is an error if the item is not in the string. This works with both characters and substrings.

- 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. Methods and functions can modify existing lists. Several operations may apply to the same list, each one causing that list to be different in some way. This contrasts with immutable objects like strings (see append and extend below). New strings are created by applying functions to old strings. These new strings can then be used, e.g., 'abc'.upper() creates a new list 'ABC'.
  - 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 pop off an item at a particular position if you use pop with an index. For example, suppose my_list = ['a','b','c'], then next_letter = my_list.pop(1) would result in next_letter being equal to 'b' and my_list being set to ['a','c'].


You can add a list of items to a list via `extend`. For example, suppose `my_list = ['a','b','c']`. Then `my_list.extend(['d','e'])` will add the items in the 2nd list to the end of the first one, setting it to `['a','b','c','d','e']`.

other list methods: `list.reverse()` – turns a list backwards; `list.sort()` – puts a list in sort order; etc.

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 `break`. After a break, the statement after the loop executes.
   - It is common to exit a loop of any kind by using a `return` – this also exits the function.
   - It is common to initialize a variable outside a loop (called an accumulator) that then gets incremented inside the loop.

7. `while loops`
   - First line `While (BOOLEAN-EXPRESSION):`
   - The loop keeps executing the indented body until BOOLEAN-EXPRESSION evaluates as `False`.
   - If BOOLEAN-EXPRESSION is always `True`, the loop is endless.
   - Typically BOOLEAN-EXPRESSION contains one or more variable(s), such that some values of these variables (or this variable) cause BOOLEAN-EXPRESSION to evaluate as `True` and other values cause it to evaluate as `False`.
   - The body of the loop can change these variables. The loop starts when BOOLEAN-EXPRESSION is `True`. Then, under most circumstances, BOOLEAN-EXPRESSION eventually evaluates as `False` and the loop halts. For example, if the BOOLEAN-EXPRESSION is `stop == False`, setting `stop` to `True` inside the loop, will cause the loop to finish.
   - It is common to use accumulator variables in a similar manner as with `for loops`.
   - `break` and `return` behave the same way for `while loops` as they do with `for loops`.

8. `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`.

9. 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 Y
- Not X returns True if X is False

10. 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.
- my_turtle.setposition(X,Y) – moves the turtle to the position with coordinates (X,Y). A straight line is drawn from the current position to that position if the pen is down.

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