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 turtle_setup2():
    global my_screen
    global turtle1
    global turtle2
    my_screen = turtle.Screen()
turtle1 = turtle.Turtle()
turtle2 = turtle.Turtle()

def move_both_turtles(distance):
    turtle1.fd(distance)
turtle2.fd(distance)

def turn_turtles_opposite(degrees):
    turtle1.left(degrees)
turtle2.left(degrees+180)

def square(turtle, side):
    for rep in range(4):
        turtle.fd(side)
turtle.left(90)

def turtle_square_eight():
    turtle_setup2()
turtle1.pd()
turtle2.pd()
turn_turtles_opposite(45)
square(turtle1, 40)
square(turtle2, 40)
turtle1.pu()
turtle2.pu()

def main():
    turtle_square_eight()

main()

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

def get_sequence_pairs_from_string(string):
    output = ''
    for character1 in string:
        for character2 in string:
            if character1 != character2:
                output = output + character1 + character2 + ' '
    return(output[:-1])

def main():
    output = get_sequence_pairs_from_string('abc')
    print(output)

main()

Question: What is the value of the local variable output when it is printed out during the execution of the main function?

Answer:
def breeding_rabbits(generations):
    ## this function simulates the population growth in rabbits
    ## when there are no predators
    total1 = 0
    total2 = 0
    for number in range(1, generations + 1):
        if total1 == 0:
            total1 = 1
            ## We buy one pair of baby rabbits
        elif total2 == 0:
            total2 = 1
            ## After one month they mature and are ready to mate
        elif total1 <= total2:
            total1 = total1 + total2
        else:
            total2 = total1 + total2
            ## Afterwards there are always some rabbits that are too
            ## young to reproduce and all the other rabbits do reproduce.
            ## Thus rather than doubling, you simply add the 2 previous totals.
            ## The lower total is the number that can reproduce and the higher total
            ## is the total number of rabbits (including the immature ones).
        if total1 > total2:
            return(total1)
        else:
            return(total2)

def main():
    ## from a previous test
    output = breeding_rabbits(10)
    print(output)

main()

Question: What is the value of the local variable output when it is printed out during the execution of the main function?

Answer:
Question 4

def p_latin_ify(word):
    if (len(word) >= 3) and (word[:2] == 'qu'):
        return(word[2:] + word[2:] + 'ay')
    elif (len(word) >= 2) and (word[0] in 'aeiou'):
        return(word + 'yay')
    else:
        vowel_position = 0
        found = False
        while (vowel_position < len(word)) and (not found):
            if word[vowel_position] in 'aeiou':
                found = True
            else:
                vowel_position = vowel_position + 1
        if vowel_position < (len(word) - 1):
            return(word[vowel_position:] + word[vowel_position:] + 'ay')
        else:
            return(word)

def main():
    output = []
    for word in ['pickle', 'queen', 'clam', 'banana', 'earth']:
        output.append(p_latin_ify(word))
    print(output)

main()

Question: What is the value of the local variable output when it is printed out during the execution of the main function?

Answer:
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:

```python
>>> items = [['plutifier', ['Q4545', 300], ['Q4547', 400], ['P5151', 375], ['Q4545', 290]],
            ['tripod', ['C2422', 75], ['C2424', 125], ['F3539', 170], ['F3545', 250]],
            ['strainer', ['P4332', 200], ['P4335', 220], ['P5001', 500], ['P6001', 700], ['P3001', 20]],
            ['Booster', ['Q347', 20]],
            ['Video Enhancer', ['R1234', 1400], ['R1234', 75], ['R1234', 200]]

>>> estimate_budget(items)
Most Expensive Items: [['Q4547', 400], ['F3545', 250], ['P6001', 700], ['Q347', 20], ['R1234', 1400]]
Least Expensive Items: [['Q4545', 290], ['C2422', 75], ['P3001', 20], ['Q347', 20], ['R1234', 75]]
minimum 480
maximum 2770
```

Write a program like `estimate_budget`, demonstrated above. It should take a list like `items` as a parameter. The list is a list of lists, each of these lists referring to a record about an item that is part of some project. Each record consists of: the name of the item followed by a list of serial numbers and prices for different models of this item. For example, there are 4 tripods listed with prices ranging from model 'C2422', which cost $75 to model 'F3545', which costs $250. For each item, the program finds the model costing the maximum and the minimum amounts and records them in two lists (printed above). Then it totals up the maximum and minimum amounts (printed above), giving the user some idea of how much they should expect the project to cost.
Question 6: Write a program that uses the turtle module to draws a parallelogram. It should take the length of two of the sides and the measure of one of the angles as input parameters. For example, `draw_parallelogram(30, 100, 72)` would produce the following picture:

![Parallelogram](image)

Hint: Opposite angles of a parallelogram are equal. Two adjacent angles sum to 180 degrees.

Extra Credit: Write a program `flower` that takes 3 arguments: `number_of_pedals`, `short_side` and `long_side`. It should produce a flower-like design made of several parallelograms with the same dimensions. The variable `number_of_pedals` indicates the number of parallelograms, each of which should be derived by calls to `draw_parallelogram` (part 1 of the question). The variables `short_side` and `long_side` indicate two of the variables for `draw_parallelogram`. The remaining variable, `angle` should be calculated automatically so that the parallelograms are arranged evenly around the center. For example, an angle of 72 degrees would be used if `number_of_pedals` is set to 5, since $5 \times 72 = 360$. Therefore, `flower(5, 30, 100)` would produce the following picture:

![Flower](image)
Question 7: Write a program in which the user plays a card game against the computer. The deck of cards is represented by the variable `deck_of_cards`, defined as follows:

```python
deck_of_cards = [['2', 'C'], ['2', 'D'], ['2', 'H'], ['2', 'S'], 
                 ['3', 'C'], ['3', 'D'], ['3', 'H'], ['3', 'S'], 
                 ['4', 'C'], ['4', 'D'], ['4', 'H'], ['4', 'S'], 
                 ['5', 'C'], ['5', 'D'], ['5', 'H'], ['5', 'S'], 
                 ['6', 'C'], ['6', 'D'], ['6', 'H'], ['6', 'S'], 
                 ['7', 'C'], ['7', 'D'], ['7', 'H'], ['7', 'S'], 
                 ['8', 'C'], ['8', 'D'], ['8', 'H'], ['8', 'S'], 
                 ['9', 'C'], ['9', 'D'], ['9', 'H'], ['9', 'S'], 
                 ['10', 'C'], ['10', 'D'], ['10', 'H'], ['10', 'S'], 
                 ['J', 'C'], ['J', 'D'], ['J', 'H'], ['J', 'S'], 
                 ['Q', 'C'], ['Q', 'D'], ['Q', 'H'], ['Q', 'S'], 
                 ['K', 'C'], ['K', 'D'], ['K', 'H'], ['K', 'S'], 
                 ['A', 'C'], ['A', 'D'], ['A', 'H'], ['A', 'S']]```

The computer and the user will play 5 rounds of a card game. Whoever wins 3 or more rounds wins the game.

In each round, the computer and the user are randomly given 2 cards. The set of two cards is called a “hand”. The cards should be removed from the deck when they are dealt, so they cannot be chosen again in subsequent rounds. Suggestion: use the method `.pop` with a random number that is less than the length of the list, e.g.,

```python
new_card = deck_of_cards.pop(13)
```

would assign `['5', 'D']` to the variable `new_card` and remove the card from the deck. Use `random.randint` to select the random number.

Each card consists of a face value (the first item in the list) and a suit (the second item). Possible face values include: '2', '3', '4', '5', '6', '7', '8', '9', 'J', 'Q', 'K', 'A'. These are listed in order of value, i.e., 'A' is the most valuable card, 'K' is the second most valuable, ... and '2' is the least valuable. Possible suits include: 'C', 'D', 'H' or 'S'.

The computer and player's hands are compared as follows, to see which is better. If the face value of two cards are the same, it is called a pair, e.g., if the first position in both cards is an 'A', than the two cards as a set, are a pair of 'A's. A pair will always beat a non-pair, e.g., the computer would win a hand, if it had the hand `['2', 'C']` and the user had the hand `['2', 'D']`. If both players have a pair or both players have a non-pair, then the hand with the highest valued card wins, e.g., the hand `['K', 'D']`, `['9', 'C']` would beat the hand `['9', 'D']`, `['J', 'H']`. If the top cards are tied, the lower cards are compared, e.g., `['K', 'H']`, `['8', 'C']` would beat `['K', 'D']`, `['2', 'C']`. Finally, in cases of a tie, that round is repeated with new cards (unless there are no more cards left).

For determining the value of a card: First determine the face value by virtue of the order of the face in the list:

```python
face_card_order = ['2', '3', '4', '5', '6', '7', '8', '9', '10', 'J', 'Q', 'K', 'A']
```

which can be derived by the method `.index`, e.g., `face_card_order.index('2')` equals 0, whereas `face_card_order.index('J')` equals 9. Therefore a 'J' is more valuable than a '2'. If the two cards have the same face values, the cards are assumed to have the same value. Thus ties are possible. When comparing two non-pair hands, you should separate each hand into a high card and a low card. The high cards should be compared first, then if the two high cards are tied, the low cards should be compared.
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 multiply two numbers or repeat a string some number of times.

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: '0T1h2e3 4b5o6o7k8'`. 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: ‘∗∗∗8 − 7h − 6e − 5d − 4b − 3o − 2a − 1k∗∗∗’.

- 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.

- **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 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
• $X \geq Y$ returns True if $X$ is greater than or equal to
• $\text{Not } X$ 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)