Question 5: Write a program that reads in a tsv file and computes instructions based on the contents of each line. The first line of the file consists of the labels: operator, first and second. The items in the operator column determine the operation that the program should perform and the items in the first and second columns will be the arguments of that operator. For each line, the program should print the result of combining the arguments using the operator. The operators should have the following interpretations:

- plus – add two integers (first and second)
- string plus – concatenate two strings (first and second)
- Xs and Ys – create a string consisting of first instances of ’x’ followed by second instances of ’y’. In other words, the variables first and second will determine how many xs and ys.
- Otherwise – if there is any other operator, the program should indicate that there is “No Valid Result”

For example, suppose the input file is called “sample2.tsv” and contains the following lines, where multiple spaces represent a tab. For display purposes, the columns are aligned as they would appear in a spreadsheet program:

<table>
<thead>
<tr>
<th>first</th>
<th>operator</th>
<th>second</th>
</tr>
</thead>
<tbody>
<tr>
<td>55</td>
<td>plus</td>
<td>45</td>
</tr>
<tr>
<td>11</td>
<td>string plus</td>
<td>11</td>
</tr>
<tr>
<td>20</td>
<td>times</td>
<td>20</td>
</tr>
<tr>
<td>2</td>
<td>Xs and Ys</td>
<td>5</td>
</tr>
</tbody>
</table>

Given this file, below is an idle session running the sample program:

```python
>>> do_calculations_from_file('sample2.tsv')
55 plus 45 equals 100
11 string plus 11 equals 1111
20 times 20 equals No Valid Result
2 Xs and Ys 5 equals xxyyyy
```
**Question 6:** Write a program called `nearly_colliding_turtles` using turtle graphics that meets the following description:

- It should take 3 parameters: distance1, distance2 and increment
- The program should create 2 turtles: (for example, they can be named turtle1 and turtle2)
- With their pens up, each turtle should travel the distance `distance1`, but in opposite directions. Turtle1, should go directly (90 degrees) downward (with X=0) and Turtle2 should go directly upwards.
- With their pens down, the 2 turtles should now go towards each other, in small steps, taking turns. Each step should be equal to the distance `increment`.
- Right when it would seem like the turtles would collide with each other, they each should turn 45 degrees (to the right from their point of view). Then they travel an additional distance of `distance2`.

For example, the command: `nearly_colliding_turtles(300,100,50)` caused turtle1 and turtle 2, to each go 300 pixels in the opposite directions. Then each got progressively 50 units closer to the center. Finally, when they should have crashed into each other, they each turned 45 degrees (to their right) and went an additional 100 pixels. The final effect looked something like this figure (the arrows represent the final turtle positions):
Question 7: Write a program that simulates a simple card game. Assume the global variable `deck_of_cards` – you don’t need to define it. Furthermore, you can use this function to reset it at the beginning of each game:

```python
def reset_deck():
    global deck_of_cards
    deck_of_cards = ['A','2','3','4','5','6','7','8','9','10','J','Q','K',
                     'A','2','3','4','5','6','7','8','9','10','J','Q','K',
                     'A','2','3','4','5','6','7','8','9','10','J','Q','K',
                     'A','2','3','4','5','6','7','8','9','10','J','Q','K']
```

The player starts with 0 points. Each round, the player and the computer randomly pick one card from this deck. The two cards are compared to see which is “higher” and based on this, points are added or subtracted from the player’s total. We will assume that each card with a number on it has that number for its value (e.g., a 9 is worth 9). All other cards (A,J,Q,K) are worth 10 points. There are three possible outcomes for each round:

- If the player’s card is higher than the computer’s, the player’s points increase by the 2 card’s difference in value.
- If the computer’s card is higher, the player’s points decrease by this difference.
- If the cards have the same value, there is no change in the number of points.

When the deck is empty (after 26 rounds), the game is over. There are three possible ways the game can end:

- If the player has more than 0 points, they win.
- If the player has less than 0 points, they lose.
- If the player has exactly 0 points, it is a tie.

A sample game follows:

```bash
>>> simple_card_war()
You picked 3
You lost 7 points
The computer picked 9
You picked 5
You gained 4 points
The computer picked 4
You gained 1 points
You picked J
You picked Q
You gained 7 points
You picked 9
No change in points
You picked K
You gained 1 points
The computer picked Q
You gained 2 points
You picked 7
You picked 8
The computer picked 5
You gained 3 points
You gained 2 points
The computer picked A
You lost 3 points
You picked K
You lost 2 points
You gained 8 points
The computer picked J
You gained 1 points
You picked 10
You picked K
The computer picked 7
You gained 8 points
You picked 9
No change in points
You picked A
The computer picked A
No change in points
You picked J
The computer picked 5
You gained 5 points
You gained 5 points
You picked 4
You gained 10 points
You picked 9
You gained 5 points
The computer picked 5
You gained 1 points
You picked 3
You picked 6
You gained 3 points
You gained 5 points
The computer picked 5
You lost 3 points
You picked 6
The computer picked 6
You gained 6 points
You picked J
You picked 8
The computer picked 3
You have earned -6 points
You lost 3 points
You gained 3 points
You picked Q
The computer picked 2
You gained 6 points
You picked J
You picked 6
The computer picked Q
No change in points
You lost 4 points
You picked 2
```
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*: 

```
'T_0T_1h_2e_3_4b_5o_6o_7k_8'.
```

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: ‘Th e book’[−2:0] evaluates to ‘k’, the last character in the string.

– 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 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)

12. 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.

13. File Input/Output

• os – module including global variables like os.linesep (end of line strings: '\n' or '\r\n') and os.sep (path separators – forward slash '/' or backward slash '\'). The os module also includes functions that interact with the operating system. os.getcwd() returns the current working directory. os.listdir(PATH) returns a list of files in PATH; os.path.isdir(PATH) returns True if PATH is a directory and False otherwise; os.path.isfile(PATH) returns True if PATH is the name of an existing file and False otherwise.
• Streams – Python objects used for reading files and writing to files.
• instream = open('my_file.txt','r') sets the variable instream to the contents of the file 'my_file.txt'. for loops will treat instream as a list of strings, each ending with os.linesep. For most applications, it makes sense to remove these.
• outstream = open('my_file.txt','w') sets the variable outstream to an object that will ultimately be saved as the file my_file.txt. The method outstream.write(string) will write a string to that file. It is a good idea to include \n anywhere you would like a line break in the file as end of lines are not automatic. \n should be used, rather than os.linesep, even in Windows.
• `stream.close()` will close an opened stream. This ends the connection between Python and a file. In the case of output streams (like `outstream`), the content of the stream is written to the file.

• `with open(file,'r') as instream:` or `with open(file,'w') as outstream:` starts a block in which a stream is opened. The body of code indented under these statements can read from or write to the stream. After the block ends, the stream is closed.

14. Error Handling

• `raise Exception(STRING)` – raises an exception (causes an error) and prints out STRING.

• `Try/Except` – Two key words that begin blocks, similar to IF/Else statements. If the code indented under `Try:` does not cause any error, then the following `Except` statements are ignored. If an error is raised, the `Except` statements can “catch” an error. Rather than error, the code indicated under `Except` executes.