Assignment 07

Due at 10:00am on Wednesday, March 16, 2016

Assignment Guidelines:

- **This assignment tests material introduced in Module 06.**
- **You may use structural or generative recursion in your solutions (unless otherwise indicated).**
- **Do not use any Python constructs introduced in later modules (i.e. do not use loops from module 07).**
- **Read each question carefully for specific restrictions.**
- **Solutions must be placed in files a07qX.py, for X=1,2,3,4, and must be in Python.**
- Download the testing module from the course web page. Include `import check` in each solution file. Review course notes and the style guide to determine check functions to use for testing. A tolerance of 0.0001 is acceptable if using `check.within`.
- Do not import any modules other than `math` and `check`.
- Do not define helper functions locally. Define all helper functions above the required function.
- Examples and tests are not required for any helper functions, but purpose statements and contracts are.
- Do not use any other Python functions not discussed in class or explicitly allowed elsewhere. See the allowable functions post on Piazza. You are always allowed to define your own helper functions, as long as they meet the assignment restrictions.
- You may use global constants in your solutions, but global variables can only be used for testing.
- Download the interface file from the course Web page to ensure that all function names are spelled correctly and each function has the correct number and order of parameters. Use the function headers in your submitted files for each question.
- For full style marks, your program must follow the **Python section** of the CS116 Style Guide. In particular,
  - Be sure to include all the steps of the design recipe for all required functions: including purpose and effects statements, contracts (including requirements), examples (note the new style), and tests.
  - Templates are not required.
  - The purpose should be written in your own words and must include meaningful use of a function's parameter names.
  - There will be marks assigned for the appropriate use of constants, helper functions, choice of meaningful names, and appropriate use of whitespace.
- The solutions you submit must be entirely your own work. Do not look up either full or partial solutions on the Internet or in printed sources.
- Do not send any code files by email to your instructors or ISAs, or post any code publicly on Piazza. It will not be accepted by course staff as an assignment submission, even if you are having issues with MarkUs.
- Course staff will not debug code emailed to them or posted privately on Piazza.
- Test data for all questions will always meet the stated assumptions for consumed values.
- No late assignments will be accepted.
- Check MarkUs and your basic test results to ensure that your files were properly submitted. In particular:
  - A misnamed file or function will receive no marks.
  - Do not copy any text from the interactions window in WingIDE or from this pdf into your programs (even as comments). It will make your submitted file unreadable in MarkUs and you will receive no marks (correctness or style) for that question.
- Be sure to review the Academic Integrity policy on the Assignments page.

Coverage: Module 6
Language: Python 3

1. Bubble sort is yet another sorting algorithm. The basic idea is to "bubble" the smallest element (or the largest element) to the end of the list by **comparing two adjacent list elements (switching them if needed), along the list.** (Note, this is different from selection sort. In selection sort, you use max or min to find the largest or the smallest element). Then, bubble the second smallest, then the third smallest, and so on. An animation of bubble sort can be found at: https://upload.wikimedia.org/wikipedia/commons/c/c8/Bubble-sort-example-300px.gif .

   Write a Python program `bubble_sort` which consumes a list of integers, `lst`, and produces the same list items, but sorted in descending order. For example:

```
bubble_sort([])=>[]
bubble_sort([6, -5 ,9]) => [9, 6,-5])
bubble_sort([ -9, 0, 66, 5, 3, 97,-34, 8, 21, 16])
        =>[97, 66, 21, 16, 8, 5, 3, 0, -9, -34]
```

   Do not mutate the original list `lst`. Do not use sort functions or abstract list functions for Question 1.

2. Humans use different bases to represent numbers. You have seen decimal numbers and binary numbers used before. For this question, you will be using trinary numbers, also called base-3 numbers, in which only digits 0, 1, and 2 are used. Trinary numbers can be used in logic circuits, quantum logic gates, fractals, Cantor sets, and computing systems. The table below shows the relationships between decimal numbers and trinary numbers.

decimal numbers	0	1	2	3	4	5	6	7	…
trinary numbers	0	1	2	10	11	12	20	21	…

   A 4-digit trinary number "abcd" represents a decimal number: $a*3^3+b*3^2+c*3^1+d*3^0$, where a,b,c,d are digits 0, 1, or 2. For example: trinary number 21 represents decimal number: $2*3^1+1*3^0 = 7$. Another example: trinary number 2012 represents decimal number: $2*3^3+0*3^2+1*3^1+2*3^0 = 59$.

   Write a Python program `sym_tri_num` that consumes a natural number `n (n>=1)` and produces a list of all **symmetric** trinary numbers of length `n` in increasing numerical order. Use strings to represent trinary numbers. In a symmetric trinary number, the first digit is the same as the last digit, the second digit is the same as the second last digit, and so on. For example, 2201022 is a symmetric trinary number, while 2201122 is not. For this question, leading zeros are allowed and should be included in your produced list.

   For example:

```
sym_tri_num(1) =>['0', '1', '2']
sym_tri_num(3) =>['000','010','020','101','111','121','202','212','222']
sym_tri_num(4),
     => ['0000','0110','0220','1001','1111','1221','2002','2112','2222']
```

   You may use abstract list functions for Question 2, but do not use sort functions (built-in or your own).

3. Most word processing software provides a function for counting words. Upon request, word processing software will provide the total number of words in a given piece of text.

   Write a Python function `word_count` which consumes a string, `text`, and produces `None`. The function should print the total word count, as well as the number of occurrences of each individual word in alphabetic order. Here a word is defined as a sequence of non-blank, non-period, and non-comma characters. In other words, any combinations of blanks, commas, and periods are used as separators between words. Upper case letters and lower case letters are considered as different letters. Beside English letters, numbers, symbols, blanks, commas, and periods are allowed. Alphabetic order is the order produced by Python sort methods. For example, if `l=['3','$','a','$','!','A']`, after `l.sort()`, `l = ['!','$','$','3','A','a']`.

   For example,

   `word_count("")` prints (without the quotes): `"There are a total of 0 words."`

   `word_count("Up,up,up,to the sky,  ...  3 ups.")` prints (without the quotes):

```
"There are a total of 8 words.
3 - 1 times.
Up - 1 times.
sky - 1 times.
the - 1 times.
to - 1 times.
up - 2 times.
ups - 1 times."
```

   `word_count("mixed-with-symbols!!! hi2342648 @#$%^&*")` prints (without the quotes):

```
"There are a total of 3 words.
@#$%^&* - 1 times.
hi2342648 - 1 times.
mixed-with-symbols!!! - 1 times."
```

   If `text` is `"Once upon a time, there was a mountain. In the mountain, there was a temple. In the temple, there was an old monk. The old monk was telling a story to a young monk. The story went like this, once upon a time, there was a mountain..."`, then `word_count(text)` prints (without the quotes):

```
"There are a total of 47 words.
In - 2 times.
Once - 1 times.
The - 2 times.
a - 7 times.
an - 1 times.
like - 1 times.
monk - 3 times.
mountain - 3 times.
old - 2 times.
once - 1 times.
story - 2 times.
telling - 1 times.
temple - 2 times.
```

```
the - 2 times.
there - 4 times.
this - 1 times.
time - 2 times.
to - 1 times.
upon - 2 times.
was - 5 times.
went - 1 times.
young - 1 times."
```

You may use sort functions and abstract list functions for Question 3.

4. In Brazil, churrascaria is a popular type of restaurants which serve BBQed meats. The waiters walk around the restaurant with skewers full of various BBQed meats, slicing meat onto guests' plates. Such restaurants also provide each guest with a wooden block on their table. This block has one side painted green and the other side painted red. If you want more BBQed meats, you turn the green side up. Waiters will come by and fill your plate. If you have enough on your plate or you simply want to take a break, you put the red side up and waiters will leave you alone.

   You and three friends decide to go to a local churrascaria restaurant. To make the evening more fun, you also have decided to dance a little before you eat. Here is how the dance goes:

   Step 0: The four of you sit around a table. Each of you chooses an initial colour of your wooden block. Say they are: red, green, green, red. The initial seating plan will look like the following:

<table>
<tr><td>you
(red)</td><td rowspan="2">table</td><td>friend #1
(green)</td></tr>
<tr><td>friend #3
(red)</td><td>friend #2
(green)</td></tr>
</table>

   Notice the seating plan starts at the top-left seat and moves clockwise.

   Step 1: If all the blocks are green, or all the blocks are red, the dance is over and you can eat. If not, go to next step.

   Step 2: Choose the adjacent blocks, or the diagonal blocks (defined below), and change one or both wooden block colours.

   For any seating plan, **top left and top right positions** are considered **adjacent**. **Top left and bottom right** positions are considered **diagonal**. Therefore, for the above seating plan, the positions of you and friend #1 are considered **adjacent**. The positions of you and friend #2 are considered **diagonal**.

   Step 3: Roll two dice, and the four of you will dance and rotate seats clock-wise. The number of seats to rotate is the sum of the two dice values. Everyone brings their wooden block with him/her when they dance. (For this question, we will not provide test cases in which the sum of the dice values are always 4, 8,

or 12. That means, you and friends actually change seats during the dance most of the times, rather than returning to your original seats all the time.)

Assume that you have completed the first round move, ie. have turned both diagonal blocks green side up (see round descriptions below), and the dice sum was 7. In this case, you would have rotated 7 seats clock-wise. Therefore, from the above initial seating plan in Step 0, your new seating plan will look like the following (after turning the diagonal blocks green and rotating 7 seats):

<table>
<tr><td>friend #1
(green)</td><td rowspan="2">table</td><td>friend #2
(green)</td></tr>
<tr><td>you
(green)</td><td>friend #3
(red)</td></tr>
</table>

Then, repeat Step 1, and possibly Steps 2 and Step 3.

To prevent you and your friends from starving for too long, here is an algorithm that you should follow to achieve all blocks showing green or all blocks showing red in no more than 5 dances.

- **Round zero:** initial colour choices.
- **First round:** consider diagonal blocks and turn both blocks green side up. Then roll the dice and rotate.
- **Second round:** consider adjacent blocks. At least one will be green side up as a result of the previous step. If the other is red side up, turn it green side up as well. Then roll the dice and rotate.
- **Third round:** consider diagonal blocks. If one has the red side up, turn it green side up. If both have green side up, turn the top left block red side up. Then roll the dice and rotate.
- **Fourth round:** consider adjacent blocks. Flip both blocks, ie. red turns to green, or green turns to red. Then roll the dice and rotate.
- **Fifth round:** consider diagonal blocks. Flip both blocks. By now, all blocks will have green side up, or all blocks will have red side up.

Note, for Round zero, there is only one step: Step 0. For all other rounds, Step 1 to Step 3 are performed if needed. It is also possible that at any round, you could have all blocks green side up or all blocks red side up. You should eat then since the dance is over. If not, you proceed to the next round.

Write a Python function `churras` that consumes a list of strings, `status`, representing the initial block colours (say, `['red', 'green', 'green', 'green']`), and a list of numbers between 2 and 12 inclusive, called `dice`. These numbers are the sums of the two dice values and the length of the list is 5. The function produces `None` and prints the list of block colours at the end of each round, until all the blocks are showing green or all the blocks showing red. The function then prints a completion message.

For example:

```
churras(['red', 'green', 'green', 'green'],[11,4,12,3,10])
=> None, and prints:

['red', 'green', 'green', 'green']
['green', 'green', 'green', 'green']
Let's eat. We are all very very hungry!
```

```
churras(['red','red','green','green'], [7,8,8,11,6])
=> None , and prints:

['red', 'red', 'green', 'green']
['red', 'green', 'green', 'green']
['green', 'green', 'green', 'green']
Let's eat. We are all very very hungry!

churras(['green','red','red','red'], [11,3,5,7,10])
=> None, and prints:

['green','red','red','red']
['red', 'green', 'red', 'green']
['green', 'red', 'green', 'green']
['green', 'red', 'red', 'green']
['green', 'red', 'green', red']
['red','red','red','red']
Let's eat. We are all very very hungry!
```

The following description shows the detailed seating plan and block colour change during the above call: `churras(['green','red','red','red'], [11,3,5,7,10]).`

Round zero: Initial seating:

- Print: `['green', 'red', 'red', 'red']`
- Check status, if not all green or not all red, move onto next round.

First round: consider diagonal blocks and turn both blocks green side up.

- After turning, we have: ['green', 'red', 'green', 'red'] (Do not print this since we have not danced yet.)
- Roll the dice: get number 11, dance and rotate seats. Print what we have after the rotation: `['red', 'green', 'red', 'green']`
- Check status, if not all green or not all red, move onto next round.

Second round: consider adjacent blocks. At least one will be green side up as a result of the previous step. If the other is red side up, turn it green side up as well.

- After turning, we have: ['green', 'green', 'red', 'green'] (Do not print this since we have not danced yet.)
- Roll the dice: get number 3, dance and rotate seats. Print what we have after the rotation: `['green', 'red', 'green', 'green']`
- Check status, if not all green or not all red, move onto next round.

Third round: consider diagonal blocks. If one has the red side up, turn it green side up. If both have green side up, turn the top left block red side up.

- After turning, we have: ['red', 'red', 'green', 'green'] (Do not print this since we have not danced yet.)
- Roll the dice: get number 5, dance and rotate seats. Print what we have after the rotation: `['green', 'red', 'red', 'green']`
- Check status, not all green or not all red, move onto next round

Fourth round: consider adjacent blocks. Flip both blocks, ie. red turns to green, or green turns to red.

- After turning, we have: ['red', 'green', 'red', 'green'] (Do not print this since we have not danced yet.)
- Roll the dice: get number 7, dance and rotate seats. Print what we have after the rotation: `['green', 'red', 'green', 'red']`
- Check status, if not all green or not all red, move onto next round

Fifth round: consider diagonal blocks. Flip both blocks. By now, all blocks will have green side up, or all blocks will have red side up.

- After turning, all the blocks will have the same colour. In this case, all blocks will have red side up. Done! Print completion message.

Note, you may choose to check status more often than displayed above. Exit the program when all blocks have the same colour.

Do not mutate the original list `status`. You may use sort functions and abstract list functions for Question 4.