CSE 101: Introduction to Computational and Algorithmic Thinking

Stony Brook University

Lab Assignment #4

Spring 2018

Assignment Due: February 23, 2018 by 11:59 pm

Assignment Objectives

This lab assignment will give you practice with while-loops and Boolean operators.

Getting Started

Visit Piazza and download the "bare bones" file `lab4.py` onto your computer. Open `lab4.py` in PyCharm and fill in the following information at the top:

1. your first and last name as they appear in Blackboard
2. your Net ID (e.g., jsmith)
3. your Stony Brook ID # (e.g., 111999999)
4. the course number (CSE 101)
5. the assignment name and number (Lab #4)

Submit your final `lab4.py` file to Blackboard by the due date and time. Late work will not be graded. Code that crashes and cannot be graded will earn no credit.

Part I: Make it a Meal (20 points)

Fast food restaurants often offer customers the option of ordering a set of items that constitute a meal. For instance, if a person buys a hamburger, fries and a soft drink at the same time, the restaurant might offer the items at a discount.

Imagine a strange fast food restaurant that offers these items for sale: dumplings, hamburgers, ramen, salads, cups of soda and bottles of water.

These items are represented using the following strings:

- dumplings: `'dumpling'`
- hamburgers: `'hamburger'`
- ramen: `'ramen'`
- salads: `'salad'`
- cups of soda: `'soda'`

- bottles of water: `'water'`

Discounts are offered to patrons who order one or more of these combinations:

- Asian Fusion: a dumpling and ramen
- The Heart Attack: a hamburger and a soda
- The Unhappy Meal: a salad and a bottle of water

Write the function `find_combos(orders)`, which takes a list of strings called `orders` as its argument. The strings are chosen from the set of six strings given above and represent a collection of items that a patron wants to purchase at the restaurant. (You may assume that only valid strings appear inside `orders`.) The function counts the number of meal combinations it can make from the list of strings given. For instance, if the `orders` list were `['water', 'soda', 'water', 'soda', 'salad', 'ramen', 'water', 'dumpling', 'salad', 'dumpling']`, the function would detect one Asian Fusion meal, zero Heart Attack meals, and two Unhappy Meals. The function would return the list `[1, 0, 2]` as a result. Note that the counts are returned in this order inside the returned list:

```
[# of Asian Fusions, # of Heart Attacks, # of Unhappy Meals]
```

For this part of the assignment you do not need to use while-loops. Simply use a for-loop to traverse over the list of strings. However, you probably *will* want to use Boolean operators to help you detect the meal combinations.

Examples:

`find_combos(['dumpling', 'water', 'dumpling', 'dumpling', 'ramen', 'ramen', 'water'])`	`[2, 0, 0]`
`find_combos(['soda', 'dumpling', 'hamburger', 'hamburger', 'hamburger', 'water', 'dumpling', 'ramen'])`	`[1, 1, 0]`
`find_combos(['salad', 'soda', 'dumpling', 'hamburger', 'water', 'water', 'soda', 'ramen', 'salad'])`	`[1, 1, 2]`
`find_combos(['water', 'water', 'soda', 'soda', 'salad', 'dumpling', 'soda', 'dumpling'])`	`[0, 0, 1]`
`find_combos(['water', 'soda', 'water', 'salad', 'soda', 'hamburger', 'soda', 'hamburger', 'soda', 'salad', 'soda'])`	`[0, 2, 2]`
`find_combos(['hamburger', 'water', 'ramen', 'salad', 'water', 'soda', 'soda', 'soda', 'ramen', 'salad', 'ramen', 'ramen', 'dumpling', 'ramen', 'hamburger'])`	`[1, 2, 2]`
`find_combos(['water', 'soda', 'hamburger', 'soda', 'dumpling', 'ramen', 'dumpling', 'hamburger', 'soda', 'soda', 'salad', 'soda', 'salad', 'salad', 'dumpling'])`	`[1, 2, 1]`
`find_combos(['water', 'ramen', 'dumpling', 'ramen', 'salad', 'dumpling', 'ramen', 'soda', 'ramen', 'dumpling'])`	`[3, 0, 1]`
`find_combos(['salad', 'ramen', 'salad', 'ramen', 'soda', 'water', 'salad', 'ramen'])`	`[0, 0, 1]`
`find_combos(['ramen', 'water', 'ramen', 'soda', 'ramen', 'hamburger', 'soda', 'ramen', 'dumpling', 'hamburger'])`	`[1, 2, 0]`

Part II: Blackjack Dice (20 points)

For this part you will implement a two-player version of the card game Blackjack but with six-sided dice instead of playing cards. The function `blackjack_dice()` will simulate the game with no interaction from the user. The game rules are as follows:

- Players take turns "rolling" two six-sided dice, each accumulating a running total. We will see later how the rolling of dice will be simulated.
- If a player's total is less than 16, the player must roll the dice and add the total to his score.
- If a player's total is greater than or equal to 16, the player does not roll the dice.
- If a player's total equals exactly 21, the game ends immediately with a win for that player.
- If a player's total becomes greater than 21 ("busting"), the game ends immediately with a win for the other player.
- Thus, players continue rolling dice and accumulating totals while all of the following conditions are true:
  - Neither player has reached 21 exactly.
  - Neither player has busted (exceeded a total of 21).
  - At least one player still has a score of less than 16.

If the game ends with neither player hitting 21 or busting, then the player whose score is closest to 21 wins the game. In the event of a tie, the function returns the list `[0, 0]`.

The function takes a single argument, `dice`, which is a list of 30 or so integers in the range 1 through 6, inclusive. Rolling of dice is simulated by the function by reading integers from this list two at a time. One way to keep track of which numbers the function should read next is to maintain an index variable (e.g., `next_die`) that is updated as values are read out. This variable would be initialized to zero at the top of the function. As an example:

```
die1 = dice[next_die]
die2 = dice[next_die+1]
next_die += 2
# update a player's score using die1+die2
```

Once a game-ending condition has been reached, the function returns a list that contains two values: first, the number of the player who won (1 or 2) and second, the score of the winning player.

A few runs of the game are given below with print statements that illustrate how the games proceed. *Your solution should not contain print statements.* However, while working on your solution you may find it helpful to include such print statements.

Sample Game Run #1: Player 1 wins by earning a score closer to 21 than Player 2

```
Dice: [5, 1, 6, 1, 1, 3, 3, 6, 5, 5, 1, 2, 2, 2, 6, 1, 3, 2, 6, 2, 3, 6, 4,
       2, 4, 2, 4, 2, 3, 6]
Player 1's score: 0
    Player 1 rolled: 5 1
    Player 1's new score: 6
```

```
Player 2's score: 0
    Player 2 rolled: 6 1
    Player 2's new score: 7
Player 1's score: 6
    Player 1 rolled: 1 3
    Player 1's new score: 10
Player 2's score: 7
    Player 2 rolled: 3 6
    Player 2's new score: 16
Player 1's score: 10
    Player 1 rolled: 5 5
    Player 1's new score: 20
(Note: Player 2 does not roll again because his score is >= 16.)
Return value: [1, 20]
```

Sample Game Run #2: Player 1 wins by earning a score of exactly 21

```
Dice: [4, 1, 1, 4, 4, 3, 2, 5, 6, 3, 2, 3, 3, 1, 3, 3, 5, 3, 3, 1, 6, 5, 1,
       4, 6, 2, 2, 4, 4, 3]
Player 1's score: 0
    Player 1 rolled: 4 1
    Player 1's new score: 5
Player 2's score: 0
    Player 2 rolled: 1 4
    Player 2's new score: 5
Player 1's score: 5
    Player 1 rolled: 4 3
    Player 1's new score: 12
Player 2's score: 5
    Player 2 rolled: 2 5
    Player 2's new score: 12
Player 1's score: 12
    Player 1 rolled: 6 3
    Player 1's new score: 21
(Note: the game ends immediately. Player 2 does not get to roll again.)
Return value: [1, 21]
```

Sample Game Run #3: Player 2 wins because Player 1 busts

```
Dice: [1, 1, 3, 1, 5, 1, 6, 4, 1, 6, 5, 1, 2, 6, 4, 2, 6, 5, 4, 6, 1, 5, 3,
       4, 1, 3, 6, 3, 2, 3]
Player 1's score: 0
    Player 1 rolled: 1 1
    Player 1's new score: 2
Player 2's score: 0
    Player 2 rolled: 3 1
    Player 2's new score: 4
Player 1's score: 2
```

```
    Player 1 rolled: 5 1
    Player 1's new score: 8
Player 2's score: 4
    Player 2 rolled: 6 4
    Player 2's new score: 14
Player 1's score: 8
    Player 1 rolled: 1 6
    Player 1's new score: 15
Player 2's score: 14
    Player 2 rolled: 5 1
    Player 2's new score: 20
Player 1's score: 15
    Player 1 rolled: 2 6
    Player 1's new score: 23
(Note: the game ends immediately because Player 1 busted.)
Return value: [2, 20]
```

Some Hints:

Here is a general outline of how you might want to structure the main part of your function:

```
while at least one player needs to roll the dice again
    if Player 1's score is less than 16 then
       roll the dice
       add the sum of the dice to Player 1's score
       check for a win by Player 1 (return if Player 1 has won)
       check for a loss by Player 1 (return if Player 1 has lost)
    if Player 2's score is less than 16 then
       ...similar to the steps for Player 1...

After the while-loop:
compare the scores for the two players to determine the winner
(and return the result)
```

Examples:

`blackjack_dice([5, 1, 6, 1, 1, 3, 3, 6, 5, 5, 1, 2, 2, 2, 6, 1, 3, 2, 6, 2, 3, 6, 4, 2, 4, 2, 4, 2, 3, 6])`	`[1, 20]`
`blackjack_dice([3, 4, 2, 6, 4, 5, 6, 5, 6, 4, 5, 2, 5, 2, 5, 4, 5, 5, 4, 5, 4, 2, 3, 3, 5, 5, 1, 6, 4, 2])`	`[2, 19]`
`blackjack_dice([3, 3, 2, 6, 5, 2, 5, 1, 3, 1, 2, 5, 5, 4, 5, 4, 4, 1, 6, 6, 3, 1, 3, 4, 1, 3, 5, 3, 1, 5])`	`[2, 21]`
`blackjack_dice([4, 3, 4, 3, 5, 6, 4, 3, 3, 2, 1, 4, 4, 3, 2, 6, 4, 2, 2, 1, 1, 3, 5, 1, 1, 1, 2, 5, 6, 1])`	`[2, 19]`
`blackjack_dice([4, 3, 2, 3, 1, 1, 1, 5, 4, 6, 3, 4, 4, 3, 2, 1, 6, 5, 2, 6, 6, 5, 5, 1, 5, 3, 5, 3, 4, 4])`	`[1, 19]`
`blackjack_dice([4, 1, 1, 4, 4, 3, 2, 5, 6, 3, 2, 3, 3, 1, 3, 3, 5, 3, 3, 1, 6, 5, 1, 4, 6, 2, 2, 4, 4, 3])`	`[1, 21]`
`blackjack_dice([4, 4, 6, 3, 4, 6, 1, 3, 3, 3, 1, 3, 1, 6, 1, 5, 5, 5, 6, 4, 4, 3, 5, 2, 2, 4, 3, 5, 2, 3])`	`[2, 19]`
`blackjack_dice([5, 4, 2, 6, 3, 1, 1, 3, 5, 5, 3, 6, 2, 5, 2, 2, 2, 4, 4, 1, 4, 4, 5, 3, 1, 5, 5, 4, 5, 5])`	`[2, 12]`
`blackjack_dice([1, 2, 4, 2, 2, 4, 6, 3, 4, 6, 2, 2, 5, 4, 3, 3, 2, 1, 1, 3, 6, 3, 4, 6, 3, 5, 2, 2, 2, 2])`	`[0, 0]`
`blackjack_dice([3, 5, 1, 2, 6, 4, 4, 5, 1, 6, 4, 6, 2, 5, 5, 2, 6, 4, 3, 1, 6, 5, 1, 5, 6, 3, 6, 1, 4, 5])`	`[2, 19]`
`blackjack_dice([5, 3, 5, 2, 5, 5, 4, 2, 6, 6, 3, 2, 4, 4, 3, 6, 5, 5, 2, 2, 1, 1, 4, 5, 5, 4, 4, 3, 4, 2])`	`[1, 18]`
`blackjack_dice([1, 1, 3, 1, 5, 1, 6, 4, 1, 6, 5, 1, 2, 6, 4, 2, 6, 5, 4, 6, 1, 5, 3, 4, 1, 3, 6, 3, 2, 3])`	`[2, 20]`
`blackjack_dice([4, 3, 2, 2, 3, 3, 4, 3, 6, 2, 5, 4, 3, 5, 5, 1, 5, 4, 1, 4, 2, 4, 5, 1, 4, 4, 5, 6, 2, 1])`	`[1, 21]`
`blackjack_dice([3, 5, 4, 2, 3, 3, 5, 3, 4, 1, 5, 3, 5, 6, 2, 3, 3, 4, 5, 4, 2, 2, 3, 3, 1, 4, 1, 6, 2, 6])`	`[1, 19]`
`blackjack_dice([2, 5, 6, 4, 2, 1, 6, 5, 2, 4, 3, 5, 6, 4, 5, 1, 3, 3, 4, 3, 5, 3, 1, 4, 6, 4, 2, 1, 6, 2])`	`[2, 21]`
`blackjack_dice([1, 2, 6, 1, 2, 6, 5, 4, 6, 6, 6, 3, 5, 2, 1, 1, 3, 6, 4, 4, 4, 3, 5, 4, 3, 4, 4, 2, 1, 1])`	`[2, 16]`
`blackjack_dice([4, 5, 6, 5, 1, 2, 3, 2, 4, 1, 1, 1, 1, 1, 5, 6, 2, 2, 3, 5, 5, 6, 3, 1, 4, 6, 2, 3, 2, 3])`	`[1, 17]`
`blackjack_dice([5, 1, 1, 4, 4, 1, 2, 5, 5, 1, 4, 4, 1, 3, 2, 1, 4, 4, 1, 5, 2, 3, 6, 5, 6, 3, 3, 6, 5, 2])`	`[2, 20]`
`blackjack_dice([1, 5, 2, 3, 4, 6, 4, 6, 6, 6, 2, 5, 2, 5, 2, 6, 1, 1, 4, 5, 3, 1, 6, 4, 1, 1, 4, 5, 1, 2])`	`[1, 16]`
`blackjack_dice([3, 3, 1, 5, 2, 6, 3, 5, 1, 3, 2, 2, 6, 2, 4, 4, 5, 1, 2, 6, 4, 2, 3, 5, 1, 2, 3, 4, 1, 2])`	`[0, 0]`

How to Submit Your Work for Grading

To submit your `.py` file for grading:

1. Login to Blackboard and locate the course account for CSE 101.
2. Click on "Assignments" in the left-hand menu and find the link for this assignment.
3. Click on the link for this assignment.
4. Click the "Browse My Computer" button and locate the `.py` file you wish to submit. Submit only that one `.py` file.
5. Click the "Submit" button to submit your work for grading.

Oops, I messed up and I need to resubmit a file!

No worries! Just follow the above directions again. We will grade only your last submission.