Lab 6 – CSE 101 (Spring 2020)

The objective of this lab assignment is to become more comfortable writing Python programs that feature Boolean operators, while loops, and lists.

You will together in groups for part of this lab, but everyone will be expected to write their own solution on their individual computer and submit it on Blackboard.

Initial Setup

Download lab6.py and open it in PyCharm. You will use this file to answer all the questions.

Once finished, submit the completed `lab6.py` file on Blackboard.

1. Write 3n + 1 Sequence String Generator function (1 point)

In `lab6.py` complete the function `sequence` that takes one parameter: `n`, which is the initial value of a sequence of integers.

The *`3n + 1` sequence* starts with an integer, `n` in this case, wherein each successive integer of the sequence is calculated based on these rules:

1. If the current value of `n` is **odd**, then the next number in the sequence is three times the current number plus one.
2. If the current value of `n` is **even**, then the next number in the sequence is half of the current number. **Note:** Make sure to use integer division.
3. Integers in the sequence are generated based the two rules above until the current number becomes `1`.

Your function should start with an empty string and append characters to the string as described next. The function keeps calculating the numbers in the *`3n + 1` sequence*, and while there are still numbers left in the sequence (meaning that the number has not reached `1` yet), it keeps appending letters to the string based on these rules:

1. If the number is divisible by `2`, then append 'A' to the string.
2. If the number is divisible by `3`, then append 'B' to the string.
3. If the number is divisible by `5`, then append 'C' to the string.
4. If the number is divisible by `7`, then append 'D' to the string.

Note: The original number `n` also should contribute a letter (or letters) to the string. Also, append letters to your accumulating string for **all** conditions met above. That means, for example, if the current number is `6`, the letters 'A' and 'B' will both be appended to the string, in that order. Apply the rules in the order given. For example, for the number `6`, your code must append the letters in the order 'AB', not 'BA'.

Note: If you find any helper functions useful, feel free to use them. In fact, I suggest that you try to define and use helper functions. This idea applies not only to this problem, but to any problem.

In the end, your function should return the string generated (accumulated) by the procedure described above. If `n` initially is less than `1`, the function returns an empty string.

Example calls:

```
sequence(4)  # returns 'AA'
sequence(12) # returns 'ABABBACCAAAA'
sequence(24) # returns 'ABABABBACCAAAA'
```

Below is a closer look at calling `sequence(12)`:

```
Number(n) | 12 | 6  | 3  | 10 | 5  | 16 | 8  | 4  | 2
----------+----+----+----+----+----+----+----+----+----
Letters   | AB | AB | B  | AC | C  | A  | A  | A  | A
```

2. Help Desk Simulator (1 point)

In this problem you will write a function that simulates an IT help desk session. An IT technician offers to help different customers, who have lined up with their technical problems to fix.

In `lab6.py` complete the function `session_simulator` that takes the following two parameters, in this order:

1. `clients`: a list of integers in which each integer represents the minutes required to resolve a particular client's technical problem.
2. `regular`: an integer representing the amount of regular time (in minutes) that the technician **plans** to stay for the help session (to help all the clients).

In addition, the IT technician helps clients by the following rules:

1. First let's understand how time works in this problem. For simplicity, say the technician plans to stay for a regular help session that is 30 minutes long in total. If the first client takes 30 minutes, then the regular session is considered ended. Any additional time that this client needs or that other clients need will be reduced according to the rules described below.
2. The technician takes clients in order one-by-one: first come, first served. You may assume that no new clients arrive once the session has started (i.e. the list of clients your function receives initially does not change).
3. Because the technician is nice, he will help all the clients even when the session has officially ended. However, any clients he takes **after** the regular session time has ended will receive only **half** of the normal time that the client would have required. **Note:** You must use integer division to compute the reduced time amount. (In fact, we are using **integer divisions throughout this function.**) This means that if the normal time required is 7, then the shortened time will be: `(the normal time required // 2) = (7 // 2) = 3`
4. When a client arrives during the technician's planned regular time but needs more time than the technician has remaining in the regular session, the client will receive all of the remaining time in regular session, plus half **of the excess time** the client wanted. For example, suppose a client needs 10 minutes of the technician's time but there are only 4 minutes remaining in the planned regular session. The client will be given all 4 minutes of the remaining time plus half of the excess 6 minutes that he wanted: `4 + (10-4)//2 = 7` minutes total.

Your function should keep running until all clients have been helped. The function should return the **total time spent** by the technician (in minutes). You may assume that all arguments to the function will be valid. However, it is possible that no client shows up for the help session. In that case, the function should simply return 0 (because they helped no one).

Example calls:

```
session_simulator([5, 5], 10)            # returns 10
session_simulator([1, 5, 4, 11], 10)     # returns 15
session_simulator([5, 7, 5, 6], 20)      # returns 21
```

3. Café Simulator (1 point)

In `lab6.py` complete the function `cafe_day` that takes one parameter, `orders`, which is a list of drink orders for the day at a cafe. The list `orders` contains one or more sub-lists, in which each sub-list represents a particular drink order. In each sub-list, there are four elements, in this order:

1. A string that indicates the customer's **membership**, which will be one of these three categories:
    - **Platinum** membership is represented by the letter 'P'
    - **Gold** membership is represented by the letter 'G'
    - **Silver** membership is represented by the letter 'S'
2. An integer that represents the number of **large** drinks for the particular order.
3. An integer that represents the number of **medium** drinks for the particular order.
4. An integer that represents the number of **small** drinks for the particular order.

Your function should process all orders, then return a floating-point number that represents the revenue generated for the day in dollars. *Don't round the return value!* The prices for different sizes are:

Drink Size	Price
Large	$3.50
Medium	$2.50
Small	$1.25

In addition, there will be different privileges depending on different memberships, as described below:

1. For a **Platinum** member: If the customer purchases 3 or more large drinks, **OR** 4 or more medium drinks, then he/she gets 3 free small drinks, but *at most* 3.
2. For a **Gold** member: If the customer purchases at least 10 drinks (in any combination of small, medium and large), then he/she gets a 20% discount.
3. For a **Silver** member: The customer receives 2% off of total price regardless of number of drinks ordered.

Invalid values: Your function should be able to handle invalid values outlined below. In these cases, skip the *entire* drink order and continue to the next one. Valid drink orders have the following characteristics:

- Each sub-list has **exactly** four values.
- The first element in the sub-list, which is the **membership**, is one of the three strings 'P', 'G' or 'S' (all lowercase letters are invalid).
- The number of drinks for any size is always greater than or equal to zero.

If the *orders* list is empty, the function should simply return 0.0.

Example calls:

```
orders1 = [['P', 5, 0, 4]]
cafe_day(orders1) # returns 18.75

orders2 = [['S', 1, 2, 3], ['P', 5, 0, 4], ['G', 4, 4, 2]]
cafe_day(orders2) # returns 51.955

orders3 = [['G', 4, 3, 2], ['S', 0, 0, 10], ['P', 1, 4, 3]]
cafe_day(orders3) # returns 49.75
```

4. Premium Airlines (1 point)

In `lab6.py` complete the function `premium_airlines` that takes the following three parameters, in this order:

1. `membership`: the string `'Diamond'`, `'Platinum'`, `'Regular'`. Any other string is considered invalid input.
2. `price`: a positive integer that represents the price of a plane ticket.
3. `points`: a positive integer that represents the number of points earned in the airline reward program.

The *Premium Airlines* company wants to increase its sales by rewarding customers based on: their membership, price they paid for their ticket, and points they have accumulated by flying with Premium Airlines. You are tasked to create a system that rewards customers accordingly. Since the company is generous, it wants to give its customers as many points as possible per purchase.

Here are the policies:

1. If `membership` is either `'Diamond'` or `'Platinum'` and the `price` is 5000 or more, add 35 to the current `points`.
2. If `membership` is `'Diamond'`, the `price` is 2000 or more, and `points` is more than 300, add 30 to the current `points`.
3. If `membership` is `'Platinum'` and the `price` is 2000 or more, add 20 to the current `points`.
4. If `membership` is `'Diamond'`, the `price` is 500 or more, and `points` is 100 or more, add 10 to the current `points`.
5. If `membership` is `'Regular'` and the `price` is 5000 or more, add 5 to the current `points`.
6. If `membership` is `'Diamond'` and `points` is 25 or more, add 2 to the current `points`.
7. In all other cases, add 0 to `points`.

After adjusting the value of `points` according to the above rules, the function returns the new value of `points`.

In a situation where customer satisfies more than one criterium, the company only gives points for the criteria that awards the customers most points, *ignoring* anything else they quality for.

For example, if the `membership` is **'Diamond'**, `price` is **5000**, and `points` is **30**, the function returns **65**. That is, Total points (65) = Current points (30) + Points awarded (35).

Your function should be able to return the total points the person has earned (`points` they have + whatever they earn) based on the `price` of the ticket they purchased, the type of `membership` they have, and `points` they currently have.

Example calls:

```
premium_airlines('Diamond', 4999, 700)   # returns 730
premium_airlines('Regular', 5000, 300)   # returns 305
premium_airlines('Platinum', 500, 1000)  # returns 1000
```

5. Unfair Hiring System (1 point)

In `lab6.py` complete the function `unfair_hiring_system` that takes the following two parameters, in this order:

1. `applications`: a list of strings containing some combination of `'Strong'`, `'Fair'`, `'Poor'`, and `'Disaster'`.
2. `mood`: a positive integer that indicates the starting mood of the recruiter. A higher value increases their chances of hiring weaker applicants.

A recruiter wants to hire people for their firm. They are given a list of `applications` to look at and needs to make a list of applicants they want to hire. The function will return a list of the *indices* of those people hired from the `applications` list. They start off at a moodiness level of `mood`, but as time progresses, they start feeling exhausted and start judging applications more harshly. If their mood drops to 0, then they stop recruiting and rejects all remaining applicants.

Your function must mimic the above scenario. The recruiter analyzes each application in turn from the list as follows:

1. If the current application is `'Strong'`, then append the index of the application to the list to be returned and add 2 to their current `mood`.
2. If the current application is `'Fair'` and their current `mood` is 50% or more of their starting `mood` level, then append the index of the application and add 1 to their current `mood`.
3. If the current application is `'Fair'` and their current `mood` is less than 50% of their starting `mood` level, then reject the application and subtract 2 from their current `mood`.
4. If the current application is `'Poor'` and their current `mood` is 75% or more of their starting `mood` level, then append the index of the application and subtract 1 from their current `mood`.
5. If the current application is `'Poor'` and their current `mood` is less than 75% of their starting `mood` level, then reject the application and subtract 5 from their current `mood`.
6. If the current application is `'Disaster'`, then reject the application and subtract 10 from their current `mood`.
7. Every application they check decreases their `mood` by 1, regardless of how good the application is. This is in addition to the changes made due to other rules above.

Example calls:

```
    application1 = ['Strong', 'Fair', 'Disaster']
    application2 = []
    application3 = ['Disaster', 'Poor', 'Disaster', 'Disaster', 'Disaster', 'Poor',
'Disaster', 'Poor']

    unfair_hiring_system(application1, 100)   # returns [0, 1]
    unfair_hiring_system(application2, 50)    # returns []
    unfair_hiring_system(application3, 200)   # returns [1, 5]
```