Homework 4:

Farmers' Market Classes SI 206

For this assignment, you will be writing and correcting methods so that customers can successfully order and pay for food at the farmers' market using the new collective ordering system which has cashiers take orders (and payment) and pass the orders on to the stalls. You will also be writing and correcting test cases, so you can guarantee that every step from the order being taken to being processed is accurate and your customers are happy!

Review the starter code thoroughly before beginning this assignment, as understanding how the classes interact with each other is important. Take notes or draw a diagram if necessary. We cannot emphasize how important this step is.

Overview

Customer Class

The *Customer* class represents a customer who will order from the stalls. Each customer object has 2 instance variables: **name** (a string representing a customer's name) and **wallet** (a float showing how much money is in the customer's market payment card). The *Customer* class also includes several methods: ***__init__***, ***reload_money*** (which adds a passed amount to the **wallet**), ***submit_order*** (which you will implement – see details below), ***validate_order*** (which takes a **cashier**, a **stall,** the **item_name,** and the **quantity** and places an order at that cashier to be delivered to that stall), and ***__str__*** *(which prints the customer's information).*

Cashier Class

The *Cashier* class represents a cashier at the market. A cashier object has 2 instance variables: **name** (a string representing a cashier's name), **directory** (a list of stalls). The *Cashier* class includes several methods: ***__init__, has_stall*** *(which returns whether the stall is in the cashier's* **directory***),* ***add_stall*** *(which adds a new stall to the cashier's current* **directory***),* ***receive_payment*** *(which takes the customer's money and adds it to the stall's* **earning***),* ***place_order*** *(which passes the order, including the ordered food items and quantity, to the stall and this function returns the cost of this order (quantity ** **cost**)), and lastly, the ***__str__*** *method (which returns a string representing the cashier, see the starter code for details).*

Stall Class

The *Stall* class represents a vendor's stall. Each stall object has 4 instance variables: **name** (a string which is the name of the stall), **inventory** (a dictionary which holds the names of the food as the keys and the quantities of each food as the values), **earnings** (a float for the amount of earnings the stall currently has) and **cost** (the cost to the customer for each food. For simplicity,

the cost will be the same for all foods in the same stall). You will be in charge of implementing the Stall class – see details below.

Tasks to Complete

● Complete the *Customer* Class

○ Complete the ***submit_order*** method in the *Customer* class. This method takes a **cashier**, a **stall** and an **amount** as parameters, and has the customer pay the cashier the specified amount using the ***receive_payment*** method in the cashier class (i.e., it deducts money from the customer's **wallet** and adds it to the stall). See the test cases under ***test_make_payment*** for clues on how this method should behave.

● Create and implement the *Stall* class with the following methods

○ A ***constructor (__init__)*** that initializes the instance variables **name**, **inventory, cost** per food (default = 7), and **earnings** (default = 0).

○ A ***process_order*** method that takes the food **name** and the **quantity**. If the stall has enough food, it will decrease the quantity of that food in the **inventory**. Questions for you to think about: should ***process_order*** take other actions? If so, add it in your code.

○ A ***has_item*** method that takes the **food name** and the **quantity** and returns True if there is enough food left in the inventory and False otherwise.

○ A ***stock_up*** method that takes the **food name** and the **quantity**. It will add the quantity to the existing quantity if the item exists in the **inventory** dictionary or create a new item in the **inventory** dictionary with the item name as the key and the quantity as the value.

○ A ***compute_cost*** method that takes the **quantity** and returns the total for an order. Since all the foods in one stall have the same cost, you only need to know the quantity of food items that the customer has ordered.

○ A ***__str__*** method that returns a string with the information in the instance variables using the format shown below:

Expected output for printing a stall object:

"Hello, we are [NAME]. This is the current menu [INVENTORY KEYS AS LIST]. We charge $[COST] per item. We have $[EARNINGS] in total."

```
Hello, we are The Tamale Train. This is the current menu tacos, water.
We charge $10 per item. We have $700 in total.
```

- **Implement a *Main()* method**

    - Create at least *two inventory dictionaries* with at least 3 different types of food. The dictionary keys are the food items and the values are the quantity for each item.
    - Create at least 3 *Customer* objects. Each should have a unique **name** and unique amount of money in their **wallet**.
    - Create at least 2 *Stall* objects. Each should have a unique **name**, **inventory** (use the inventory that you just created)**,** and **cost.**
    - Create at least 2 *Cashier* objects. Each should have a unique **name** and **directory (a list of stalls)**.
    - Have each customer place at least one order (by calling ***validate_order***) and try all cases in the ***validate_order*** function above. See starter code for hints of all cases.

- **Write and Correct Test Cases**

    - Note: Many test cases have already been written for you. **Please do not edit test cases outside of the ones below.** As you are working on one test case, feel free to comment out the test cases that you are not working on, but be sure to uncomment all test cases before you turn in your homework.
    - ***test_compute_cost*** has an error. Can you correct it? Also what are the correct numbers to make this test pass? Correct the mistakes in this test.
    - Complete ***test_has_item****,* which tests the ***has_item*** method in the *Stall* class. We have provided 3 scenarios for you to test (please refer to the starter code).
    - Complete ***test_validate_order****,* which tests the ***validate_order*** method in the *Customer* class. The ***validate_order*** method places an order of items from a stall to be carried out by a cashier, but only if several conditions are met: if the customer has enough money in their wallet to pay for the transaction and if the stall has enough items in stock.

      When writing tests for ***test_validate_order***, please write comments for each test case describing what scenarios you are testing, similar to the comments in ***test_has_item***

      Example output for this test case:

      Don't have enough money for that :(Please reload more money!
      Our stall has run out of [Food Item] :(Please try a different stall!
      Sorry, we don't have that vendor stall. Please try a different one!

    - Complete ***test_reload_money*****,** this tests if the customer can add money into their wallet.

Grading Rubric (60 points)

Note that if you use hardcoding (specify expected values directly) in any of the methods by way of editing to get them to pass the test cases, or you edit any test cases other than the ones you have been directed to, you will NOT receive credit for those related portions.

Note - use the provided methods you will earn credit if you implement the functionality instead.

- 15 points for correctly implementing the ***Stall*** class (3 points per first four methods, and 3 points for the last two methods).
- 5 points for correctly completing the ***submit_order*** method in the *Customer* class.
- 5 points for creating the customer, cashier, and stall objects in the **main** method and correctly placing an order for each customer.
- 3 points for correcting ***test_compute_cost***
- 15 points for writing non-trivial test methods for ***test_has_item*** (5 points per scenario correctly tested).
- 15 points for writing non-trivial tests for ***test_validate_order*** (at least three scenarios; 5 points per scenario correctly tested).
- 2 points for writing a test case for ***test_reload_money***

Extra Credit (6 points)

To gain extra credit on this assignment, please complete the following task:

For every 10^{th} customer that places an order at a cashier, run a lucky draw with a 5% probability of giving the customer a $10 reward in their wallet.