**Assessment Details**

|  |
| --- |
| Please ensure that you have read Sections 1-3 of this document before going further. **Problem Overview:** In this assignment, you are developing a retail management system as in Assignment 1 using the object-oriented paradigm. Same as in Assignment 1, this retail management system is for a department store. The cashiers or the store manager of the department store are the ones that use this system to process customers' purchases. You are required to implement the program following the below requirements. Note the requirements in this assignment maybe more complex compared to those in Assignment 1. Also, we will provide you with some sample .txt files, but you should change the data in these files to test your program as during the marking, we will use different text files to test your program. |

**A - Functionalities Requirements:**

There are **4 levels**, please ensure you only attempt one level after completing the previous level.

**------------------------------------------- PASS Level (12 marks) ------------------------------------------**

At this level, your program will have some basic classes with specifications as below. You may need to define methods wherever appropriate to support these classes. At the end of the PASS level, your program should be able to run with a menu described in the Operations.

**Customers:**

1. **Class Customer**

All customers have a unique **ID**, unique **name** (a name will not include any digit). You are required to write the class named **Customer** to support the following:

1. Attributes **ID** and **name**
2. Attribute **value** to store the total money the customer spent to date. A new customer will have the value of **value** to be 0 before placing any order.
3. Constructor takes the values of **ID**, **name, value** as arguments
4. Appropriate getter methods for the attributes of this class
5. A method **get\_discount(self, price)** which returns (0, price) – where the first return value represents the discount rate associated with the customer and the second value represents the input price. This method serves as a super method and will have a more complex implementation in the subclasses.
6. A method **display\_info** that prints the values of the Customer attributes and the discount rate associated with the customer.
7. **Class Member**

A member is a customer with a membership. When placing an order, a member will be offered a discount. All members are offered a discount of a flat rate (i.e., the discount rate is the same for all orders – this is to distinguish from the discount of VIPMember below). The class **Member** should have the following components:

1. An attribute for **discount rate**, by default, it is 5%.
2. Constructor takes the appropriate parameters/arguments (be careful)
3. Appropriate getter methods for the attributes of this class
4. A method **get\_discount(self, price)** that takes the price of the order and returns both the discount rate and the price after the discount. For example, this method returns (0.05, 950) when the discount rate is 5% and the order's price is 1000$.
5. A method **display\_info(self)** that prints the values of the Member attributes.
6. A method **set\_rate** to adjust the flat rate of discount. This affects all members.
7. **Class VIPMember**

A VIP member is a customer with a VIP membership. All VIP members are offered a discount based on two rates: the first rate applies when the price of the order is smaller or equal to a threshold ($1000 by default), and the second rate applies when the order's price exceeds this threshold. For example, with the threshold being 1000$, then, when a VIP member named Sarah places an order that costs 800$, the discount rate for this order is the 1st discount rate; when Sarah places an order that costs 1200$, the discount rate for this order is the 2nd discount rate.

NOTE for all VIP members, the 2nd discount rate is always 5% more than the 1st discount rate. The discount rates might be different among the VIP members. If not specified, the first and second discount rates are set as 10% and 15%, respectively. On the other hand, the threshold applies to all VIP members, i.e., all VIP members have the same threshold.

The class **VIPMember** should support the following components

1. Attributes to support the two discount rates and the threshold limit
2. Necessary constructors
3. Appropriate getter methods for the attributes of this class
4. A method **get\_discount(self, price)** that takes the price of the order and returns both the discount rate and the price after the discount.
5. A method **display\_info** that prints the values of the VIPMember attributes.
6. A method **set\_rate** to adjust the discount rates of each individual VIP member.
7. A method **set\_threshold** to adjust the threshold limit. This affects all VIP members.

**Products**

1. **Class Product**

This class is to keep track of information on different products that the department store sells. This class supports the following information:

• **ID**: A unique identifier of the product (e.g., P1, P2, P3)

• **Name**: The name of the products (you can assume the product names are unique and they do not include any digit)

• **Price**: The price of the product

• **Stock**: the quantity of the product available in stock

You need to define appropriate attributes and methods to support the class **Product**. Note the stock quantity obviously will be changed. The product's price may also be changed by users.

**Orders**

1. **Class Order**

This class is to handle customers' orders. This class supports the following information of an order:

• **Customer**: the one who place the order (can be a normal customer, a customer with a normal membership, or a customer with a VIP membership). Note you need to think/analyse if this should be an ID, name, or something else.

• **Product**: the product of the order. Note you need to think/analyse if this should be an ID, name, or something else.

• **Quantity**: the quantity of the product ordered by customers.

• You need to think if there are any extra attributes and methods you want to define in this class

Note that this class can update information in the corresponding customer and destination if necessary. For example, an object from the class Order can update the information of the corresponding customer (e.g., discount rate) or/and the product (e.g., stock). Therefore, you need to define appropriate variables and methods to support this class.

**Records**

1. **Class Records**

This class is the central data repository of your program. It supports the following information:

* **A list of existing customers** – you need to think what you should store in this list (ID, name, or something else?)
* **A list of existing products** – you need to think about what you should store in this list (ID, name, or something else?)
* This class has a method named **read\_customers** that can read a comma-separated file called *customers.txt* and add the customers in this file to the customer list of the class. See an example of the *customers.txt* file below.

Text, letter

Description automatically generated

In this file, customers are always in this format: *ID*, *name*, *discount rate,* and *value*. For example, in the 1st line, the *ID* is C1, the *name* is James, the *discount rate* of this customer is 0, and the *value* is 500.2. Note that for VIP members, the first discount rates will be stored. A normal customer has an ID starting with the letter *"****C****"*. A member (customer with a normal membership) has an ID starting with the letter *"****M****"*. A VIP member (customer with a VIP membership) has an ID starting with the letter *"****V****"*. The numbers in the ID after these characters (C, M, V) are all unique (i.e., 1, 2, 3, 5… are unique; for example, if there is a customer with the ID of C1, there won't be a member with the ID of M1). In this part, you can assume there will be no error in this *customers.txt* file (e.g., the data format is always correct, and the values are always valid).

* This class has another method named **read\_products** that can read another comma-separated value file called *products.txt* and add the products stored in that file to the product list of the class. See an example of the *products.txt* file below. Text

  Description automatically generated

In this file, products are always in this format: *ID, name, price* (unit price per each product), and the *stock*. The IDs of all products always start with the letter "***P***". All the product IDs are unique. You can assume there will be no error in this file (e.g., the data format is always correct, and the values are always valid).

* This class also has two methods **find\_customer** and **find\_product**. These two methods are to search through the list of customers and products to find out whether a given customer or a given product exists or not. If found, the corresponding customer and product will be returned, otherwise, None will be returned. Note that both the customer and the product can be searched using either ID or name.
* This class also has two methods **list\_customers** and **list\_products**. These two methods can list all the existing customers and products on screen. The display format is flexible, and note that you can display all necessary information, and make sure at least all the information as in the *customers.txt* and *products.txt* files should be displayed. These methods can be used to validate the reading from the .txt files *customers.txt* and *products.txt.*

NOTE you are allowed to add extra attributes and methods in this class if these attributes and methods make your program more efficient.

**Operations**

This can be considered the main class of your program. It supports a menu with the following options:

1. *Place an order*: this option allows users to place an order for a customer. Detailed requirements for this option are below (Requirements vi-viii).
2. *Display existing customers*: this option can display all the information: ID, name, discount rate (1st discount rate for VIP member), value, threshold limit (for VIP member) of all existing customers.
3. *Display existing products:* this option can display all the information: ID, name, price, stock of all existing products.
4. *Exit the program:* this option allows users to exit the program.

Other requirements of the menu program are as follows:

1. When the program starts, it looks for the files *customers.txt* and *products.txt* in the local directory (the directory that stores the .py file of the program). If found, the data will be read into the program accordingly, the program will then display a menu with the 4 options described above. If any file is missing, the program will quit gracefully with an error message indicating the corresponding file is missing.
2. Your menu program will allow the user to place an order as specified in PART 1 of Assignment 1. Note that in this assignment, the customer can choose to get a normal membership or a VIP membership (a VIP membership will cost 200$ more). More detailed information regarding the membership choice is in section vii below. Also, note that you do not need to handle errors in input in this part. For example, similar to PART 1 of Assignment 1, you can assume users always enter valid products, valid product prices, and valid *"y"* or *"n"* answers. You can also assume users always enter the membership type correctly, for example, *"M"* for a normal membership, and *"V"* for a VIP membership.
3. When a customer finishes placing an order,
4. If the customer is a new customer, you need to add the information of that customer into your data collection (think/analyse carefully which information you need to add to your data collection). If the customer answers *"n"* for the question of becoming a member, then the customer is just a normal customer. If the customer answers *"y"*, then the program will ask what type of member the customer wants. If the answer is *"M"*, then the customer will become a member (a customer with a normal membership). If the answer is *"V"*, then the customer will become a VIP member (a customer with a VIP membership). Note that the customer will need to pay an extra 200$ for becoming a VIP member. Discount is NOT applied to this 200$ membership fee. Again, you can assume the users enter the membership type correctly ("*M*" or "*V*").
5. If the customer is an existing customer, you need to update the information of that customer in your data collection (think/analyse carefully which information you need to update). Note, for existing customers, you DO NOT need to ask if they want a membership (normal or VIP membership). This is slightly different compared to the requirements in Assignment 1, so please be careful.
6. Note that, be careful when you add a new customer to your data collection. As mentioned in the description of the class **Customer**, both the ID and the name of a customer are unique.
7. The value of the customer will be increased by the total money they spent on the order (this includes the VIP membership fee).
8. After each order, the stock of the chosen product will be reduced by the quantity in the order. At this level, you do not need to handle errors when the quantity in the order is larger than the stock quantity.
9. The total cost of an order can be displayed as a formatted message as below (for existing customers, new customers who are normal customers, or customers with normal membership):

*<customer name > purchases <quantity> x <product>.*

*Unit price: <the price of the product> (AUD)*

*<customer name> gets a discount of <discount percentage>%.*

*Total price: <the total price> (AUD)*

The formatted message is as below for new customers who register to be VIP members:

*<customer name > purchases <quantity> x <product>.*

*Unit price: <the price of the product> (AUD)*

*Membership price: <the price of VIP membership> (AUD)*

*<customer name> gets a discount of <discount percentage>%.*

*Total price: <the total price> (AUD)*

1. When a task is accomplished, the menu will appear again for the next task. The program always exits gracefully from the menu.

**Example Program**

We demonstrate a **sample program** that satisfies the requirements specified in Section 4. Note that this is just an example, so it is okay if your program looks slightly different, but you need to make sure that **your program satisfies the requirements listed in Section 4**.

**PASS Level**

As an example, this is how the output screen of our sample program looks like for the PASS level, when we choose option 1, which is to place an order with a customer named *Huong*, ordering 1 *shirt* and not registering for a membership. In your program, you can, and you should use different values. Also, you should test your program with different test cases, e.g., customers choose *y* (yes) for the membership registration options, to make sure your program satisfy the requirements of this level. Note that here, the program is implemented with the object-oriented paradigm and the classes described in the PASS level

Text

Description automatically generated

**---------- CREDIT level (3 marks, please do not attempt this level before completing the PASS level) ------------**

At this level, you need to handle exceptions compared to the PASS level. At this level, you are required to define various custom exceptions to handle the below issues:

1. Display an error message if the product entered by the user does not exist in the product list. When this error occurs, the user will be given another chance, until a valid product is entered.
2. Display an error message if the product quantity is 0, negative, not an integer, or larger than the stock quantity of the product. When this error occurs, the user will be given another chance, until a valid product quantity is entered.
3. Display an error message if the answer by the user is not *y* or *n* when asking if the customer wants a membership. When this error occurs, the user will be given another chance, until a valid answer (i.e., *y*, *n*) is entered.
4. Display an error message if the answer by the user is not *M* or *V* when asking the membership type. When this error occurs, the user will be given another chance, until a valid answer (i.e., *M*, *V*) is entered.
5. Display an error message if there are any errors in the files *customers.txt*, *products.txt*, e.g., wrong data format, invalid customer IDs, etc. When an error occurs, the program will display a message indicating something wrong with the files, and then exit.

**Operations**

1. In this level, in the "*Place an order*" option, your program will allow ordering a Bundle, which is a special product. It means multiple products can be offered together as one product. For example, a bundle can consist of an oven, a kettle, and a microwave. You can assume all parts of a bundle are existing products in the system.

The price of a bundle is 80% of the total price of all individual products. For example, if an oven costs 300$, a kettle costs 80$, pot costs 30$, and glass costs 15$, then the price of this bundle is 80% x (300 + 80 + 30 + 15) = 340$.

To support this feature, you need to add one more class (**Bundle**) to your program.

* **Class Bundle:** Each bundle has a unique **ID** and **name** (as with **Product**). You need to define the appropriate attributes and methods to support the class **Bundle**.

With this modification, the CSV file *products.txt* at this level may look like this: Text, letter

Description automatically generated

The ID of a Bundle always starts with the letter "B". Note that the data format of a bundle is different compared to a normal product; it includes the IDs of the product components, and at the end, it includes the stock quantity of the bundle. The IDs/names of the products and bundles are all unique. You can assume all the products in a bundle are existing products and unique (no duplicates). You can assume bundles are always stored at the end of a file, after all normal products.

1. Also, at this level, your program should display an error message if the product price is not set, 0, or negative when the user tries to order it. If this error occurs, your program will then go back to the main menu. You can assume the product prices in the *products.txt* file are always either empty or are valid numbers.
2. At this level, for the option "*Display existing products*", when displaying bundles, your program will display the ID, name, the IDs of the components, and the stock. On the other hand, the products information is the same as in the PASS level.
3. Finally, your program should support both products' IDs and names when placing an order. For example, instead of entering the product names like in the PASS level, now, users can enter the product IDs when placing an order.

**Example Program:**

**CREDIT Level**

As an example, this is what the output screen of our sample program looks like for the CREDIT level, when we choose option 1, which is to place an order with a customer named *Huong*, ordering 1 *shirt* and registering for a VIP membership. Here, we test if the program can handle some types of invalid inputs. You should test your program with different test cases to make sure your program satisfies all the requirements of this level.

Text

Description automatically generated

**---------- DI Level (3 marks, please do not attempt this part before completing the CREDIT level) -----------**

In this part, there are some additional main features for some classes in your program. Some features might be challenging. Details of these features are described as follows.

**Class Order**: This class now supports **date** information, i.e., it now has an attribute name **date** that stores which date and time the order are made (you can use an external Python module for this feature).

**Operations**

Your program now can:

1. Automatically load previous orders that are stored in a comma-separated file named *orders.txt* that is located in the local directory (same directory with the .py file). Below is an example of the *orders.txt* file:

Text

Description automatically generated

Each line in the file is an order. The format is always *customer name, product, product quantity,* and *date*. You can assume all the customers in the *orders.txt* are existing customers (their names are inside the *customers.txt* file). You can assume all the products in the *orders.txt* are existing products (their names/IDs are inside the *products.txt* file). Both customers and products can be referred by IDs or by names in this *orders.txt* file. You can assume all other information (product quantity, date) in this *orders.txt* file is always valid.

1. Errors when loading the *orders.txt* file should also be handled. When there are any errors loading the file, your program will print a message saying "*Cannot load the order file. Run as if there is no order previously.*" and run as if there is no order previously.
2. Your menu program should have an option *"Adjust the discount rates of a VIP member"* to adjust the discount rates of the VIP members. The option will ask for the name or ID of the VIP member, then ask for the new first discount rate (the second discount rate will then be adjusted accordingly). Invalid customers (non-existent or non-VIP customers) will cause the program to print a message saying: "*Invalid customer!*", and then go back to the main menu. Invalid discount rate inputs (non-number or negative discount rates) should be handled via exceptions, and the user will be given another chance until a valid input is entered. Also, your program should support both customers' IDs and names in this option, i.e., users can type either customer name or customer ID.
3. Your menu program should have an option "*Adjust the threshold limit of all VIP members*" to adjust the threshold limit of all the VIP members. This adjustment will affect all VIP members in all future orders. Invalid threshold inputs (non-number of 0 or negative threshold) should be handled via exceptions; the user will be given another chance until a valid input is entered.
4. Your menu program should have an option "*Display all orders*" to display all previous orders. The formatted message should be similar to in the *orders.txt* file (can be slightly different but all the information as in the *orders.txt* file should be shown).
5. Your menu program should have an option "*Display all orders of a customer*" to display all previous orders of a particular customer. Users can pass in the name or ID of a customer. The formatted message is similar to the previous requirement on the option "*Display all orders*". Note if the name or the ID of the customer is invalid, your program should print a message saying: "*Invalid customer!*", then go back to the main menu.
6. Note, in this part, you need to analyse the requirements and update some classes so that your program can satisfy the requirements listed above.

**Example Program**

**DI Level**

As an example, this is what the output screen of our sample program looks like for the DI level, when we choose option 6 to display all existing orders. You should test your program with different test cases to make sure your program satisfies all the requirements of this level.

Text

Description automatically generated

**------------- HD level (6 marks, please do not attempt this level before completing the DI level) --------------**

At this level, there are some additional features for some classes in your program. Note that some of them are very challenging (require you to optimize the class design and add components to support the features). Your program now can:

1. Your program now can use command line arguments to accept the three file names (the first being the customer file name, the second being the product file name, and the third being the order file name). Note the first two files are mandatory, and the third file is optional (i.e., if no order file is supplied, the program will run as if there are no previous orders). If no file names are provided, your program will look for *customers.txt, products.txt,* and *orders.txt* in the local directory. If a wrong number of arguments are provided, the program will display a message indicating the correct usage of arguments and exit.
2. At this level, your program will allow customers to purchase multiple items in each order. The requirements are as in Assignment 1 for this option (requirements 1 and 2 of Part 3). You can design extra classes or modify existing classes to support this requirement. Note that the order file format will be slightly different compared to previous levels to accommodate this requirement. Below is an example of the order file that supports orders with multiple items.

Text, letter

Description automatically generated

Each line includes the customer name/ID, the product name/ID, the corresponding quantity, the product name/ID, the corresponding quantity, …, and finally the ordered date. The items in each order can be repetitive, e.g., an order can have 2 P1, 3 P2, and 1 P1.

1. Your program will now have an option "*Summarize all orders*" to display detailed information about all previous orders. An example is as follows. OrderNum is the number of times the product is ordered (i.e., number of orders). OrderQty is the product quantity that is ordered. Note all existing products and customers are listed, although some of them did not appear in any orders.

A picture containing table

Description automatically generated

1. The menu now has an option "*Reveal the most valuable customer*" to display the customer with the maximum total money they spent to date and the total money they've spent. If there are multiple customers with the same maximum money spent, you can just display only one customer (or all customers, it's your choice).
2. The menu also has an option "*Reveal the most popular product*" to reveal the product with the highest number of orders (based on orders, not on the quantity).
3. When your program terminates, it will update the all the files (customer, products, and orders) based on the information when the program executes.

**Example Program**

**HD Level**

As an example, this is what the output screen of our sample program looks like for the HD level, when we choose option 1 to place an order and now the order can contain multiple items. You should test your program with different test cases to make sure your program satisfies all the requirements of this level.

Text

Description automatically generated

Text

Description automatically generated

**C - Documentation Requirements:**

You are required to write comments (documentation) as a part of your code. Writing documentation is a good habit in professional programming. It is particularly useful if the documentation is next to the code segment that it refers to. NOTE that you don't need to write an essay, i.e., you should keep the documentation succinct.

**Your comments (documentation) should be in the same Python file, before the code blocks (e.g., functions/methods, loops, if, etc.) and important variable declarations that the comments refer to**. Please DO NOT write a separate file for comments (documentation).

The comments (documentation) in this assignment should serve the following purposes:

* Explain your code in a precise but succinct manner. It should include a brief analysis of your approaches instead of simply translating the Python code to English. For example, you can comment on why you introduce a particular function/method, why you choose to use a while loop instead of other loops, why you choose a particular data type to store the data information.
* Document any problems of your code and requirements that you have not met, e.g., the situations that might cause the program to crash or behave abnormally, the requirements your program do not satisfy. Note that you do not need to handle or address errors that are not covered in the course material yet.
* Document some analysis/discussion/reflection as a part of your code, e.g., how your code could be improved if you have more time, which part you find most challenging, etc.