Programming Fundamentals (COSC2531)
Assignment 2

Assessment Type	**Individual assignment** (no group work). Submit online via Canvas/Assignments/Assignment 2. Marks are awarded per rubric (please see the rubric on Canvas). Clarifications/updates may be made via announcements. Questions can be raised via the lectorial, practical sessions or Canvas discussion forum. Note the Canvas discussion forum is preferable.
Due Date	**End of Week 12** (exact time is shown in Canvas/Assignments/Assignment 2) Deadline will not be advanced nor extended. Please check Canvas/Assignments/Assignment 2 for the most up to date information regarding the assignment. As this is a major assignment, a university standard late penalty of 10% per each day (e.g., 3 marks/day) applies for up to 5 days late, unless special consideration has been granted.
Weighting	**30 marks out of 100**

1. Overview

The main objective of this assignment is to familiarize you with object-oriented design and programming. Object-oriented programming helps to solve complex problems by coming up with a number of domain classes and associations. However, identifying meaningful classes and interactions requires a fair amount of design experience. Such experience cannot be gained by classroom-based teaching alone but must be gained through project experience. This assignment is designed to introduce different concepts such as inheritance, abstract classes, method overloading, method overriding, and polymorphism.

You should develop this assignment in an iterative fashion (as opposed to completing it in one sitting). You can and should get started now (when this assignment specification is posted on Canvas) as there are concepts from previous lessons that you can employ to do this assignment.
If there are questions, you can ask via the lectorial, practical sessions or the Canvas discussion forum (Canvas/Discussions/Discussion on Assessment 2). Note that the **Canvas discussion forum is preferable** as it allows other students to see your questions as well. Also, you should ask questions in a general manner, for example, you should replicate your problem in a different context in isolation before posting, and you must not post your code on the Canvas discussion forum.

2. Assessment Criteria

This assignment will determine your ability to:

i. Follow coding, convention and behavioural requirements provided in this document and in the course lessons;

ii. Independently solve a problem by using programming concepts taught over the duration of the course;
iii. Write and debug Python code independently;
iv. Document code;
v. Provide references where due;
vi. Meet deadlines;
vii. Seek clarification from your "supervisor" (instructor) when needed via the Canvas discussion forums; and
viii. Create a program by recalling concepts taught in class, understand and apply concepts relevant to solution, analyse components of the problem, evaluate different approaches.

3. Learning Outcomes

This assignment is relevant to the following Learning Outcomes:

1. Analyse simple computing problems.
2. Devise suitable algorithmic solutions and code these algorithmic solutions in a computer programming language.
3. Develop maintainable and reusable solutions.

Specifically, upon the completion of this assignment, you will be able to:

- Demonstrate knowledge of basic concepts, syntax and control structures in programming
- Devise solutions for simple computing problems under specific requirements
- Encode the devised solutions into computer programs and test the programs on a computer
- Demonstrate understanding of standard coding conventions and ethical considerations in programming

4. 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 (download on Canvas), 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.

Requirements: Your code must meet the following **functionalities**, **code** and **documentation** requirements. Your submission will be graded based on the **rubric** published on Canvas. Please ensure you read all the requirements and the rubric carefully before working on your assignment.

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:

i. Attributes **ID** and **name**

ii. 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.

iii. Constructor takes the values of **ID**, **name, value** as arguments

iv. Appropriate getter methods for the attributes of this class

v. 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.

vi. A method **display_info** that prints the values of the Customer attributes and the discount rate associated with the customer.

```
Customer:
  id
  name
  value = 0

  __init__():
    …

#getters

  def get_discount(self, price ):
    return (discount_rate, price)
  def display_info():
    print(id, name, value, discount_rate)
```

2. 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:

i. An attribute for **discount rate**, by default, it is 5%.

ii. Constructor takes the appropriate parameters/arguments (be careful)

iii. Appropriate getter methods for the attributes of this class

iv. 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$.

v. A method **display_info(self)** that prints the values of the Member attributes.

vi. A method **set_rate** to adjust the flat rate of discount. This affects all members.

```
class Member extends Customer:
  discount_rate =  0.05
  __init()__:
    …

#getters

  def get_discount(self, price ):
    return (discount_rate, price)
  def display_info():
    print(id, name, value, discount_rate)
  def set_rate(rate):
    self.discount_rate = rate
```

3. 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.

```
class VIPMember extends Member:
  second_discount_rate =  0.05
  threshold = 1000
  vip_fee = 200
  __init()__:
    …

#getters

  def get_discount(self, price ):
    return (discount_rate, price)
  def display_info():
    print(id, name, value, discount_rate)
  def set_rate(rate):
    self.discount_rate = rate
  def set_threshold(value):
    g_threshold = value
```

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

i. Attributes to support the two discount rates and the threshold limit
ii. Necessary constructors
iii. Appropriate getter methods for the attributes of this class
iv. 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.
v. A method **display_info** that prints the values of the VIPMember attributes.
vi. A method **set_rate** to adjust the discount rates of each individual VIP member.
vii. A method **set_threshold** to adjust the threshold limit. This affects all VIP members.

Products

4. Class Product

```
Class Product:
  id
  name
  price
  stock

  __init()__:

  def setPrice(value):
  def setQuantity(value):
```

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

5. Class Order

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

```
Class Order:
  customer
  product
  quantity
  #other_att

  __init()__:

  def setPrice(value):
  def setQuantity(value):
```

- **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

6. 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.

```
C1, James, 0, 500.2
C2, Lily, 0, 102.0
M3, Tom, 0.05, 250.0
V5, Harry, 0.125, 600.0
C8, Annie, 0, 50.0
V10, Sarah, 0.15, 1420.5
M11, Wilson, 0.05, 200.5
```

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).

```
class Records:
  customer_list = []
  product_list = []

  def read_products(self):
  def read_customers(self)
  def find_customer():
  def find_product():
  def list_customers():
  def list_products():

all static var and method????
```

- 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.

```
P1, shirt, 50.0, 30
P2, towel, 20.0, 20
P3, oven, 300.0, 35
P4, kettle, 90.2, 40
P5, microwave, 200.0, 15
P6, candle, 5.0, 25
```

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:

i. *Place an order:* this option allows users to place an order for a customer. Detailed requirements for this option are below (Requirements vi-viii).
ii. *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.
iii. *Display existing products:* this option can display all the information: ID, name, price, stock of all existing products.
iv. *Exit the program:* this option allows users to exit the program.

Other requirements of the menu program are as follows:

v. 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.
vi. 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.
vii. When a customer finishes placing an order,
    a. 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*").
    b. 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.

c. 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.
d. The value of the customer will be increased by the total money they spent on the order (this includes the VIP membership fee).
e. 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.

viii. 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)

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

---------- 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:

i. 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.
ii. 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.
iii. 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.
iv. 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.
v. 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

i. 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.

7. **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:

```
P1, shirt, 50.0, 30
P2, towel, -10, 20
P3, oven, 300.0, 35
P4, kettle, 90.2, 40
P5, microwave, 200.0, 15
P6, candle, , 25
P7, pant, 80.5, 20
B8, houseapp, P3, P4, P5, 10
B9, workwear, P1, P7, 20
```

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.

ii. 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.

iii. 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.

iv. 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.

---------- 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:

i. 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:

```
James, P3, 1, 01/09/2021 10:10:00
Lily, shirt, 2, 05/09/2021 14:00:00
M3, pant, 1, 12/10/2021 09:05:00
Harry, P4, 3, 20/12/2021 15:20:00
Tom, microwave, 1, 04/01/2022 09:10:00
V10, workwear, 2, 25/01/2022 20:00:00
```

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.

ii. 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.

iii. 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.

iv. 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.

v. 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).

vi. 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.

vii. Note, in this part, you need to analyse the requirements and update some classes so that your program can satisfy the requirements listed above.

---------- 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:

- 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.
- 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.

```
James, P3, 1, 01/09/2021 10:10:00
Lily, shirt, 2, towel, 5, 05/09/2021 14:00:00
M3, pant, 1, shirt, 2, B9, 1, 12/10/2021 09:05:00
Harry, P4, 3, 20/12/2021 15:20:00
Tom, microwave, 1, kettle, 1, 04/01/2022 09:10:00
V10, workwear, 2, 25/01/2022 20:00:00
```

  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.
- 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.

ljust(5)

	P1	*P2*	*P3*	*P4*	*P5*	*P6*	*P7*	*B8*	*B9*
James	*0*	*0*	*1*	*0*	*0*	*0*	*0*	*0*	*0*
Lily	*2*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*
Tom	*0*	*0*	*0*	*0*	*1*	*0*	*1*	*0*	*0*
Annie	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*

OrderNum	*1*	*0*	*1*	*0*	*1*	*0*	*1*	*0*	*0*
OrderQty	*2*	*0*	*1*	*0*	*1*	*0*	*1*	*0*	*0*

- 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).
- 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).

- When your program terminates, it will update the all the files (customer, products, and orders) based on the information when the program executes.

B - Code Requirements:

The program **must be entirely in one Python file named ProgFunA2_<Your Student ID>.py**. For example, if your student ID is s1234567, then the Python file must be named ProgFunA2_s1234567.py. Other names will not be accepted.

Your code needs to be formatted consistently. You must not include any unused/irrelevant code (even inside the comments). What you submitted must be considered as the final product.

You should use appropriate data types and handle user inputs properly. You must not have any redundant parts in your code.

You must demonstrate your ability to program in Python by yourself, i.e., you should not attempt to use external special Python packages/libraries/classes that can do most of the coding for you. **The only Python libraries allowed in this assignment are sys, datetime, copy, and os.**

Note that in places where this specification may not tell you how exactly you should implement a certain feature, you need to use your judgment to choose and apply the most appropriate concepts from our course materials. You should follow answers given by your "client" (or "supervisor" or the teaching team) under Canvas/Discussions/Discussion on Assessment 2.

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.

D - Rubric:

Overall:

Level	Points
PASS level	12
CREDIT level	3
DI level	3
HD level	6
Others (code quality, modularity, comments)	3
Others (weekly submission)	3

More details of the rubric of this assignment can be found on Canvas (here). Students are required to look at the rubric to understand how the assignment will be graded.

5. 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**.

5.1. 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 of Section 4.

```
Welcome to the RMIT retail management system!

############################################################
You can choose from the following options:
1: Place an order
2: Display existing customers
3: Display existing products
0: Exit the program
############################################################

Choose one option: 1

Enter the name of the customer [e.g. Huong]:
Huong

Enter the product [enter a valid product only, e.g. shirt, towel, oven, kettle]:
shirt

Enter the product quantity [enter a positive integer only, e.g. 1, 2, 3]:
1

This is a new customer. Does the customer want to have a membership [enter y or n]?
n

------------------------------------------------
Huong purchases 1 x shirt.
Unit price:                    50.0 (AUD)
Huong gets a discount of 0 %.
Total price:                   50.0 (AUD)
```

5.2. 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.

```
Welcome to the RMIT retail management system!

############################################################
You can choose from the following options:
1: Place an order
2: Display existing customers
3: Display existing products
0: Exit the program
############################################################

Choose one option: 1

Enter the name of the customer [e.g. Huong]:
Huong

Enter the product [enter a valid product only, e.g. shirt, towel, oven, kettle]:
adad
Product is invalid. Please enter a valid product!

Enter the product [enter a valid product only, e.g. shirt, towel, oven, kettle]:
shirt

Enter the product quantity [enter a positive integer only, e.g. 1, 2, 3]:
-5
Product quantity is not a number, negative or larger than the stock quantity!

Enter the product quantity [enter a positive integer only, e.g. 1, 2, 3]:
1

This is a new customer. Does the customer want to have a membership [enter y or n]?
y

What kind of membership the customer wants?
V

------------------------------------------------
Huong purchases 1 x shirt.
Unit price:                    50.0 (AUD)
Membership price:             200.0 (AUD)
Huong gets a discount of 10.0 %.
Total price:                  245.0 (AUD)
```

5.3. 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.

```
Welcome to the RMIT retail management system!

################################################################
You can choose from the following options:
1: Place an order
2: Display existing customers
3: Display existing products
4: Adjust the discount rates of a VIP member
5: Adjust the threshold limit of all VIP members
6: Display all orders
7: Display all orders of a customer
0: Exit the program
################################################################

Choose one option: 6
James, oven, 1, 2021-09-01 10:10:00
Lily, shirt, 2, 2021-09-05 14:00:00
Tom, pant, 1, 2021-10-12 09:05:00
Harry, kettle, 3, 2021-12-20 15:20:00
Tom, microwave, 1, 2022-01-04 09:10:00
Sarah, workwear, 2, 2022-01-25 20:00:00
```

5.3. 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.

```
Welcome to the RMIT retail management system!

################################################################
You can choose from the following options:
1: Place an order
2: Display existing customers
3: Display existing products
4: Adjust the discount rates of a VIP member
5: Adjust the threshold limit of all VIP members
6: Display all orders
7: Display all orders of a customer
8: Summarize all orders
9: Reveal the most valuable customer
10: Reveal the most popular product
0: Exit the program
################################################################

Choose one option: 1
```

```
Enter the name of the customer [e.g. Huong]:
Huong

Enter the product [enter a valid product only, e.g. shirt, towel, oven, kettle]:
oven

Enter the product quantity [enter a positive integer only, e.g. 1, 2, 3]:
1

Does the customer want to order more products?
y

Enter the product [enter a valid product only, e.g. shirt, towel, oven, kettle]:
shirt

Enter the product quantity [enter a positive integer only, e.g. 1, 2, 3]:
2

Does the customer want to order more products?
n

This is a new customer. Does the customer want to have a membership [enter y or n]?
y

What kind of membership the customer wants?
V

----------------------------------------
Huong purchases 1 x oven.
Unit price:                   300.0 (AUD)
Huong purchases 2 x shirt.
Unit price:                    50.0 (AUD)
Membership price:             200.0 (AUD)
Huong gets a discount of 10.0 %.
Total price:                  560.0 (AUD)
----------------------------------------
```

6. Submission

As mentioned in the Code Requirements, **you must submit only one file named ProgFunA2_<Your Student ID>.py** via Canvas/Assignments/Assignment 2. It is your responsibility to correctly submit your file. Please verify that your submission is correctly submitted by downloading what you have submitted to see if the file includes the correct contents. The final .py file submitted is the one that will be marked.

Weekly Submission

You are required to submit your code every week starting from Week 9 to Week 11, and the final version before the due date in Week 12. In each weekly submission, you need to write some code demonstrating some parts of your program (at least 50 lines of code per week – not include comments). You will be awarded marks for the weekly submissions. If your code in the weekly submissions is related to the assignment and satisfy the condition of at least 50 lines of code per week, then you are awarded full 1 mark per each weekly submission (maximum 3 marks for 3 weeks, excluding the final submission). If your code in the weekly submission is not satisfied the criteria mentioned in the previous sentence, you are awarded 0.5 marks for each weekly submission. If you do not submit any file, you are not awarded any mark for that week.

Late Submission

All assignments will be marked as if submitted on time. Late submissions of assignments without special consideration or extension will be automatically penalised at a rate of 10% of the total marks available per day. For example, if an assignment is worth 30 marks and it is submitted 1 day late, a penalty of 10% or 3 marks will apply. This will be deducted from the assessed mark. Assignments will not be accepted if more than five days late, unless special consideration or an extension of time has been approved.

Special Consideration

If you are applying for extensions for your assessment within five working days after the original assessment date or due date has passed, or if you are seeking extension for more than seven days, you will have to apply for Special Consideration, unless there are special instructions on your Equitable Learning Plan.

In most cases you can apply for special consideration online here. For more information on special consideration, visit the university website on special consideration here.

5. Referencing Guidelines

What: This is an individual assignment, and all submitted contents must be your own. If you have used sources of information other than the contents directly under Canvas/Modules, you must give acknowledgement of the sources, and give references using the IEEE referencing format.

Where: You can add a code comment near the work (e.g., code block) to be referenced and include the detailed reference in the IEEE style.

How: To generate a valid IEEE style reference, please use the citethisforme tool if you're unfamiliar with this style.

6. Academic Integrity and Plagiarism (Standard Warning)

Academic integrity is about honest presentation of your academic work. It means acknowledging the work of others whilst developing your own insights, knowledge and ideas. You should take extreme care that you have:

- Acknowledged words, data, diagrams, models, frameworks and/or ideas of others you have quoted (i.e., directly copied), summarized, paraphrased, discussed or mentioned in your assessment through the appropriate referencing methods.
- Provided a reference list of the publication details so your readers can locate the source if necessary. This includes material taken from the internet sites.

If you do not acknowledge the sources of your material, you may be accused of plagiarism because you have passed off the work and ideas of another person without appropriate referencing, as if they were your own.

RMIT University treats plagiarism as a very serious offence constituting misconduct. Plagiarism covers a variety of inappropriate behaviors, including:

- Failure to properly document a source
- Copyright material from the internet of databases
- Collusion between students

For further information on our policies and procedures, please refer to the University website (link).

7. Assessment Declaration:

When you submit work electronically, you agree to the assessment declaration: https://www.rmit.edu.au/students/student-essentials/assessment-and-results/how-to-submit-your-assessments