




School of Computing Technologies

COSC2531 Programming Fundamentals

Assignment 2

	Assessment Type: Individual assignment; no group work. Submit online via Canvas → Assignments → Assignment 2. Marks awarded for meeting requirements as closely as possible. Clarifications/updates may be made via announcements/relevant discussion forums.
	Due date: end of Week12; Deadlines will not be advanced nor extended. Please check Canvas → Assignments → Assignment 2 for the most up to date information. As this is a major assignment, a university standard late penalty of 10% per each working day applies for up to 5 working 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. Develop this assignment in an iterative fashion (as opposed to completing it in one sitting). You can and should get started now.

If there are questions, you must ask via the relevant Canvas discussion forums in a general manner (replicate your problem in a different context in isolation before posting).

2. Assessment Criteria

This assessment will determine your ability to:

1. Follow coding, convention and behavioral requirements provided in this document and in the lessons.
2. Independently solve a problem by using programming concepts taught over the first several weeks of the course.
3. Write and debug Python code independently.
4. Document code.
5. Ability to provide references where due.
6. Meeting deadlines.

7. Seeking clarification from your “supervisor” (instructor) when needed via discussion forums.
8. Create a program by recalling concepts taught in class, understanding and applying concepts relevant to solution, analysing components of the problem, evaluating different approaches.

3. Learning Outcomes

This assessment is relevant to the following Learning Outcomes:

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

4. Assessment details

Note: Please ensure that you have read sections 1-3 of this document before going further. Your code must meet the following code and documentation requirements.

This assessment requires you to build a warehouse management application based on Assignment 1 using object oriented approach.

===== PASS Level =====

You may need to define methods wherever appropriate to support these classes.

Customer and subclasses

1. Class Customer

All customers have a unique **ID** and **name**. All existing customers are offered discounts. You are required to write the class named **Customer** to support the following:

- (i) Attributes **ID** and **name**
- (ii) Constructor taking the value for **ID** and **name** as arguments
- (iii) Appropriate getter/accessor methods for the attributes of this class.
- (iv) A method `get_discount(self, price)` which should be an empty super method.

2. Class RetailCustomer

All retail customers are offered a discount of a flat rate, given the customer has made an order before. The class should have the following components:

- (i) An attribute for rate of discount, by default that is 10%.
- (ii) Constructor with appropriate arguments/parameters (be careful).
- (iii) A method `get_discount` method which takes the price and returns the discount offered. For example it returns 10 for a price of \$100 with a discount of 10%.
- (iv) Appropriate getter/accessor methods for the attributes.
- (v) A method `displayCustomer` that prints the values of the RetailCustomer attributes.
- (vi) A method `setRate` to adjust the flat rate of discount. That affects all retail customers.

3. Class WholesaleCustomer

All wholesale customers are offered a discount based on two rates, one for amounts up to a threshold, \$1,000 inclusive by default, and one for the amounts exceeding that threshold. The second rate is always 5% more than the first rate. For example a 11% discount for \$1,000 will associate with a 16% discount for \$1,200 if the threshold is \$1,000. The threshold applies for all wholesale customers but not the rates. The rates are always 10% and 15% if not specified.

This class should support the following components

- (i) Class attributes to support the two rates.
- (ii) Necessary constructors.
- (iii) A method `get_discount(self, price)` which takes the price and returns the discount offered.
- (iv) Appropriate getter/accessor functions for the class attributes.
- (v) A method `displayCustomer` that prints the values of the ~~RetailCustomer~~ `WholesaleCustomer` attributes.
- (vi) A method `setRate` to adjust the discount rates if they are not 10% and 15%.

Product

4. Class Product

This class is to keep track of information of different products in the warehouse. This class supports the following information.

- **ID** – A unique identifier for the product (e.g. “P11”)
- **Name** – The name of the product which is always a single word.
- **Price** – The price of the product.
- **Stock** – The quantity of the product available in stock.

You need to define the appropriate variables and methods to support class Product. Note the quantity in stock obviously will change. The price of a product may also be changed by the user.

Orders

5. Class Order

This class is to handle customer orders. This class supports the following information of an order.

- **Customer** – who made the order (should it be ID, name or something else?)
- **Product** – what has been ordered (how about this?)
- **Quantity** – The quantity of the product ordered by the customer.

This class can update information in the corresponding customer and product if necessary (be careful with your overall class design). Define the appropriate variables and methods to support the Order class.

You may need to find a way to manage multiple orders. It is up to you how that can be handled, e.g. by a dedicated class, or by a part of an existing 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
- a list of exiting products

This class has a method `readCustomers`, which can read a comma separated file (CSV) called "`customers.txt`" and add existing customers to the customer list. See example below.

```
1,  Bryan, R, 12.5, 100
2,  Zoran, W,  8.5,  200
3,  Linda, W, 10.0,  300
14, Jack,  R, 12.5,  400
```

Customers are always in this format: ID, Name, **R**etail customer or **W**holesale customer, discount rate and total (value of all orders made in the past).

Assume there is never error in this file.

This class has another method `readProducts` that can read another CSV file called “`products.txt`” and add the products stored in that file to the product list. See the example.

```
P1, Wine,      10.0,  50
P2, Chocolate, 5.00,  100
P3, Candle,    1,     1000
```

Products at the Pass level are always in this format: ID, Name, Price per Unit, In-stock Quantity. Their IDs always start with letter P.

You can assume no errors in the file “`products.txt`”.

This class has two methods `findCustomer` and `findProduct`. They are to search through the lists to find out whether a given customer or a given product exists or not. If found, the corresponding customer or product will be returned. Otherwise, **None** will be returned. Note both customer and product can be searched using either ID or name.

This class also has methods `listCustomers` and `listProducts`. They can list all the customers and products on screen. The display format is the same as that in the text files shown above. These methods can be used to validate the reading from “`customers.txt`” and “`products.txt`”.

Operations

This can be treated as the main class of your program. It supports a **menu** and can do the following:

1. When your program starts, it looks for “`customers.txt`” and “`products.txt`” in the local directory. If found, data will be read into the program accordingly. The program will display the menu. If any file is missing, the program will quit with an error message.
2. From the menu, the user can list all customers and list all products. The display format is the same as that specified in **Records**.
3. Your program will allow the user to manually enter an order as specified in Assignment 1. You can assume no error in input at this level. So you don't have to handle errors.
4. The total price of the order can be displayed as a formatted message to the user. Note appropriate discount should apply.

```
<customer name> purchased <quantity> x <product name>
Unit price: <product price>
Total price: <total price>
```

5. After making an order, the user can list the product information to see the reduction in the stock quantity of that product.
6. When a task is accomplished, the menu will appear again for the next task. The program always exits gracefully from the menu, e.g. there is an exit option on the menu.

=== Credit level === You must ONLY attempt this level after you complete the PASS level

7. Class Combo

At this level you need to add a Combo class which is a special kind of product. That means multiple products can be sold together as one product. For example a Christmas pack consists of a bottle of wine, two packs of chocolate and three candles.

You can assume all parts of the combo are existing products in the system. These products are combined together to make a combo.

The price of a combo is 90% of the total price of all individual products. For example if a bottle of wine is \$10, a pack of chocolate is \$5 and one candle is \$1, then the aforementioned Christmas pack will sell at a price of $(10 + 5 + 5 + 1 + 1 + 1) \times 90\% = \20.7 .

Each combo has an ID and name as well. It also keeps track of the quantity available in stock.

You need to define the appropriate variables and methods to support the Combo class.

The CSV file "`products.txt`" at this level may look like this:

```
P1, Wine,      10.0,   50
P2, Chocolate,  5.00,  100
P3, Candle,     1,    1000
P4, Flower,    -1,     15
C1, ChristmasPack, P1, P2, P2, P3, P3, P3, 10
```

The ID of a Combo always starts with letter C. The data format of a combo is different. It contains all the products in the combo. For duplicate products, e.g. two packs of chocolate, it simply repeats the product ID. The number at the end is the quantity of the combo in stock.

You can assume combos are always stored at the end of the file, after all normal products, and a combo never contains a product this is not in the file. You can again assume no errors in the file "`products.txt`".

Exceptions

At this level you are required to define various exceptions to handle the following issues that your program may encounter during operation.

- (i) The quantity entered by the user while ordering a product is NOT valid. (be careful)
- (ii) The product name or ID entered by the user does NOT exist.
- (iii) The product price is not set, or negative, when the user tries to order it.
- (iv) A new customer is trying to order a free product (price is 0).

Operations

As this level, your program also supports **the menu** specified in PASS level. Extra features are added to the menu.

1. The total price of the order can be displayed as a formatted message to the user. Note appropriate discount should apply.

```
<customer name> purchased <quantity> x <product name>  
Unit price: <product price>  
Total price: <total price>  
Remaining stock: <quantity in stock after the order>
```

2. The program will display an error message if the product name or ID manually entered by the user does not exist in the product list. If that occurs, the user will be given another chance, until an existing product is entered. This is handled as an exception.
3. If a product with no price is ordered, your program will display an error message and go back to the menu.
4. When a new name or ID is entered by the user, your program needs to confirm whether a new customer is to be created. If yes, ask for the name, the category (R or W) and automatically generate a unique numeric ID for the new customer. Then immediately proceed with the order.
5. Note no new customer is entitled for discount or free products. These errors occurred during manual ordering should be handled gracefully with exceptions.

=== DI level === You must ONLY attempt this level after you complete the CREDIT level

Class Order

This class at DI level supports date information.

- **Date** – Which date was the product ordered (use an existing Python module)

Exceptions

At the DI level you are required to define more exceptions to handle more issues that may happen during the execution of your program.

- (i) The quantity in an order is larger than the quantity in stock.
- (ii) Invalid input is entered when the user tries to adjust the discount rate of customer(s) or adjust the threshold for wholesale customers.
- (vii) Other errors that may occur, especially during IO operations.

Operations

At the ID level, orders can be entered as a CSV file.

1. When select the order option on the menu, your program should first look for a text file on the local directory "`orders.txt`", which is also a CSV file. See an example below.

```
Linda, P1, 2
Linda, P3, 15
Jack, Chocolate, 5
Zoran, Wine, 10, What a bargain!
2, C1, 11, 2020-12-20
```

2. Each line in the file is one order. The format is always: Customer, Product, Quantity. A record may also contain other information such as comments, date at the end. See the 4th and 5th lines in the above example.
3. Both customer and product can be referred by ID or by name in this file.
4. At this level you can assume the ordered quantity in the file "`orders.txt`" will never below the quantity in stock. Assume the orders ONLY contain existing customers and existing products of which the prices are non-negative.
5. Errors occurred during manual ordering should be handled gracefully with exceptions. For example if the quantity entered in an order is larger than the quantity in stock. Possible IO issues, e.g. a corrupted file, should be handled gracefully as well.
6. Your program allows the user to adjust the discount rate for a particular customer or adjust the threshold for wholesale customers. Invalid input should be handled.
7. There is an option on the menu at this level to show the orders. See example below. All existing products and customers are listed, although some of them did not appear in any orders.

	P1	P2	P3	P4	C1
Linda	2	0	15	0	0
Jack	0	5	0	0	0
Zoran	10	0	0	0	11
Bryan	0	0	0	0	0

=== HD level === You must ONLY attempt this level after you complete the DI level

Classes

At this level you can optimize the class design and add components to support the functionalities described below.

Operations

At this level your program should support the following.

1. When the order option on the menu is selected, your program will now ask the user to enter the file name, for example `"March/orders1.txt"`. Assume the file is always a CSV file containing correct order information. However you need to handle possible IO issues gracefully using exceptions, e.g. no such file or a corrupted file. Re-try should be permitted if there is a problem.
2. The user can select the above option multiple times so multiple orders can be carried out. The user could call the same order file multiple times or call different order files. Nevertheless, the product and customer information should be updated accordingly with each read of an order file.
3. Your program can use command line arguments to accept two file names, the first being the customer file, the second being the product file. If no argument is provided, then the program will look for `"customers.txt"` and `"products.txt"` as specified in the PASS level. If wrong number of arguments are provided, the program will show the correct usage and quit.
4. At this level, the order quantity in an external file may be below the quantity in stock. Your program needs to handle such kind of situations gracefully using appropriate exception mechanism. If that happens, the particular order will be ignored. An error message from the exception will be displayed. Other valid orders in the same file will be processed as usual.
5. Through the menu, user can invoke a method named as `"replenish"`, which sets the stock of all products according to a number entered by the user. For example, if the user entered 50, then every product will increase by 50 if that product's stock is below 50. Otherwise the stock will remain unchanged.
6. The menu at the HD level has an option to reveal the most valuable customer. That is based on the total value of all orders from that customer, counting discounts.
7. The menu now has an option to reveal the most popular product. That is based on how many times that product appears in an order, not the total quantity in all its orders. An order of 100 units still counts as 1. (may be challenging)
8. At this level, the menu option to show orders will provide more information. If the example `"orders.txt"` on Page 8 was entered twice, then the output will look like this:

	P1	P2	P3	P4	C1	TOTAL
Linda	4	0	30	0	0	\$63
Jack	0	10	0	0	0	\$43.75
Zoran	20	0	0	0	22	\$599.69
Bryan	0	0	0	0	0	\$0

OrderNum	4	2	2	0	2	10
OrderQty	24	10	30	0	22	86

9. When your program terminates, it will update the customer file and the product file, no matter whether they are the default files or the files provided through command line. In the customer file, the discount rate, the total value may change. New customer may be added. In the product file, the price and the number in stock may change.

=== For All Levels ===

Time management

You are required to make **one submission every week** after the release of this assignment, regardless what level you are attempting. This is to help your time management which is a critical skill in professional development. Last minute coding through sleepless nights never produces good quality code.

Don't worry about the code quality of your early submissions. We only mark the final submission but would like to see your progress. Marks will be given for regular updates. (see the rubric)

=====

The following code requirements must be applied in a fashion similar to what has been shown in lesson materials.

The program must be entirely in one Python class/file `ProgFunA2_<Your Student ID>.py`
 Other names will not be accepted.

Code formatted consistently. Must not include any unused/irrelevant code (even inside comments); what is submitted must be considered the final product.

Use appropriate data types and handle user inputs properly.

Must not have redundant conditions/parts in your statements.

Students must demonstrate their ability to manipulate (standard) Python lists on their own without using external classes/libraries.

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. Follow answers given by your “client” or “supervisor” (the teaching team) under **Canvas→Discussions→‘Assignment 2’** when in doubt.

Documentation requirements

Write comments in the same Python file, before code blocks (e.g. before methods, loops, ifs, etc) and important variable declarations. DO NOT write a separate file.

The comments in this assignment **MUST** contain the following information:

1. [Your name and student ID.](#)

2. [The highest level you have attempted.](#) That means you have completed all requirements of the levels below. Mark will be only given at the lowest level of partial completion. For example, you completed the PASS level, tried 50% of the CREDIT level, 30% of the DI level and 10% of the HD level, then your submission will be marked at the CREDIT level.

3. [Your submission history.](#) That must match with the submission record on Canvas. Note you are required to make one submission every week after the release of this assignment.

4. [Explanation of your code in a precise but succinct manner.](#) It should include a brief analysis of your approaches and evaluation instead of simply translating the Python code to English. For example you may explain your class design and your code segment.

5. [Any problems of your code and requirements that you have not met.](#) For example, situations that might cause the program to crash or behave abnormally, or ideas/attempts for completing a certain functionality. Write these in the approximate locations within your code.

No need to handle or address errors that are not covered in the course.

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 acknowledge the sources and give references using IEEE referencing style.

Where: Add a code comment near the work to be referenced and include the reference in the IEEE style.

How: To generate a valid IEEE style reference, please use the [citethisforme tool](#) if unfamiliar with this style. Add the detailed reference before any relevant code (within code comments).

6. Submission format

Submit **one file** [ProgFunA2_<Your Student ID>.py](#) showing the final output of your program via [Canvas→Assignments→Assignment 2](#). It is the responsibility of the student to correctly submit

their files. Please verify that your submission is correctly submitted by downloading what you have submitted to see if the files include the correct contents.

7. Academic integrity and plagiarism (standard warning)

Academic integrity is about honest presentation of your academic work. It means acknowledging the work of others while 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), summarised, paraphrased, discussed or mentioned in your assessment through the appropriate referencing methods,
- Provided a reference list of the publication details so your reader can locate the source if necessary. This includes material taken from 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 behaviours, including:

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

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

8. Assessment declaration

When you submit work electronically, you agree to the [assessment declaration](#).

Rubric (see more details on Canvas)

Assessment Task	Marks
PASS Level Requirement	12 marks
CREDIT Level Requirement	3 marks
DI Level Requirement	3 marks
HD Level Requirement	6 marks
Others	
1. Code quality and style	1.5 + 1.5 + 0.5 x 6 marks
2. Comments / Analysis/ Reflection	
3. Weekly submissions	