Programming Fundamentals (COSC2531)
Final Coding Challenge

Assessment Type	**Individual assessment** (no group work). Submit online via Canvas/Assignments/Final Coding Challenge. Marks are awarded per rubric (please see the rubric on Canvas). Clarifications/updates may be made via announcements. Questions can be raised via the Canvas discussion forum.
Due Date	**End of Week 14** (exact time is shown in Canvas/Assignments/Final Coding Challenge) Deadline will not be advanced nor extended. Please check Canvas/Assignments/Final Coding Challenge 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 applies for up to 5 days late, unless special consideration has been granted.
Weighting	**40 marks out of 100**

1. Overview

The main objective of this final project is to assess your capability of program design and implementation for solving a non-trivial problem. You are to solve the problem by designing a number of classes, methods, code snippets and associating them towards a common goal. If you have questions, ask via the relevant Canvas discussion forums 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 in this course;
iii. Design an OO solution independently and write/debug in Python code;
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 (i.e. Python).
3. Develop maintainable and reusable solutions using object-oriented paradigm.

4. Assessment Details

Please ensure that you have read Sections 1-3 of this document before going further.

Problem Overview: In this final coding challenge, you are asked to develop a Python program named **my_tournament.py** that can read data from files and perform some operations. You are required to implement the program following the below requirements. Note we will give you some initial files for you to run with your developed program, BUT you should change data in these files to test your program. During the marking, we will use different data/files to test the behavior of 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 parts**, please ensure you only attempt one part after completing the previous part.

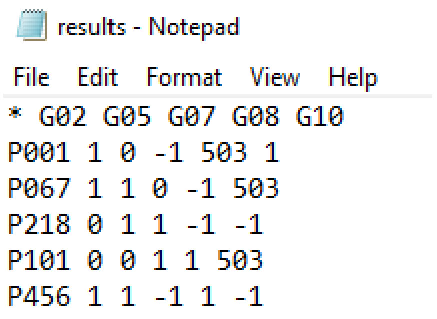
-- PART 1 (20 marks) --

Your project is to implement the required functionalities in object-oriented (OO) style with at least three classes: ***Tournament***, ***Player***, and ***Game***. You need to design appropriate static/instance variables, constructors, and methods in these classes. All the class related info should be encapsulated inside of the corresponding class.

In this part, your program can read from a text file (which store the player result data) specified in command line, then create a list of *Player* objects, a list of *Game* objects, and a 2D-list (or other data types) to store the data (the player results). Your program should create a *Tournament* object, call its *read_results(file_name)* method to load data from the text file (*file_name*), and then call its *display_results()* method to display the results in the required format (as specified in the following).

Below is an example of the text file that stores the player result data – see the next page (in the sequel, we will call this file: results file). Data fields are separated by spaces and new lines. The first row contains game IDs and the first column contains player IDs. The first field in the data, the top left corner, is always the character *"*"*. You can assume the number of games or the number of players will never be more than 10.

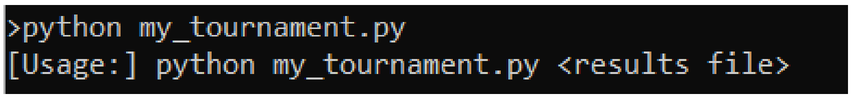
The file stores each player's results in all the games. A result of *"1"* means the player wins the game, a result of *"0"* means the player loses the game, a result of *"-1"* means the player does not play the game, and a result of *"503"* means the player is playing the game (ongoing) and the result is not available yet. In this part, you can assume all the results are strictly integers. You can assume there are no duplicate or redundant rows or columns. And you can assume the format of the data in the file is always correct.



```
* G02 G05 G07 G08 G10
P001 1 0 -1 503 1
P067 1 1 0 -1 503
P218 0 1 1 -1 -1
P101 0 0 1 1 503
P456 1 1 -1 1 -1
```

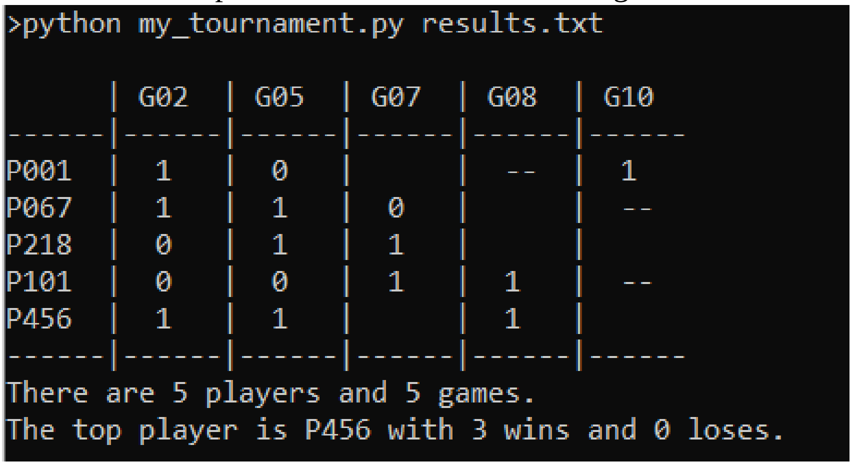
Your program should show usage if no results file is passed in as a command line argument. Otherwise, it can display a table showing the players results, and a message showing the player with the highest number of wins. In the printed table, if a player does not play a game (i.e. when the result is -1), the data field at that particular location of game and player is empty. If a game is ongoing (i.e. when the result is 503), then a double dash (--) is shown at that particular data field. The formatted message needs to be exactly as below:

1. This is when no results file is passed in as a command line argument.



```
>python my_tournament.py
[Usage:] python my_tournament.py <results file>
```

2. This is when the results file is passed in as a command line argument.



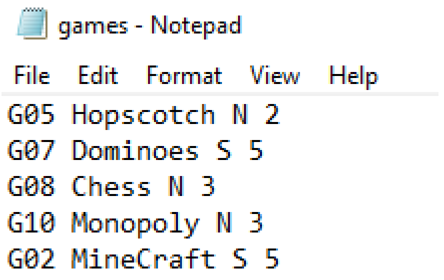
```
>python my_tournament.py results.txt

      |  G02 |  G05 |  G07 |  G08 |  G10
------|------|------|------|------|------
P001  |  1   |  0   |      |  --  |  1
P067  |  1   |  1   |  0   |      |  --
P218  |  0   |  1   |  1   |      |
P101  |  0   |  0   |  1   |  1   |  --
P456  |  1   |  1   |      |  1   |
------|------|------|------|------|------
There are 5 players and 5 games.
The top player is P456 with 3 wins and 0 loses.
```

- PART 2 (You must only attempt this part after completing PART 1 – 4 marks) -

In this part, your program can support more information of games. Now, apart from the ID, each game will have a name, and a weight; both IDs, names and weights can be modified. There are two types of games: one is *Special Game*, and one is *Normal Game*. All the special games have a same weight, by default, it is *5.0*. Each normal game will have each own weight and the weights of the normal games are required to be smaller than *5.0* and always be a positive number. You should define appropriate private variables, getters and setters for games.

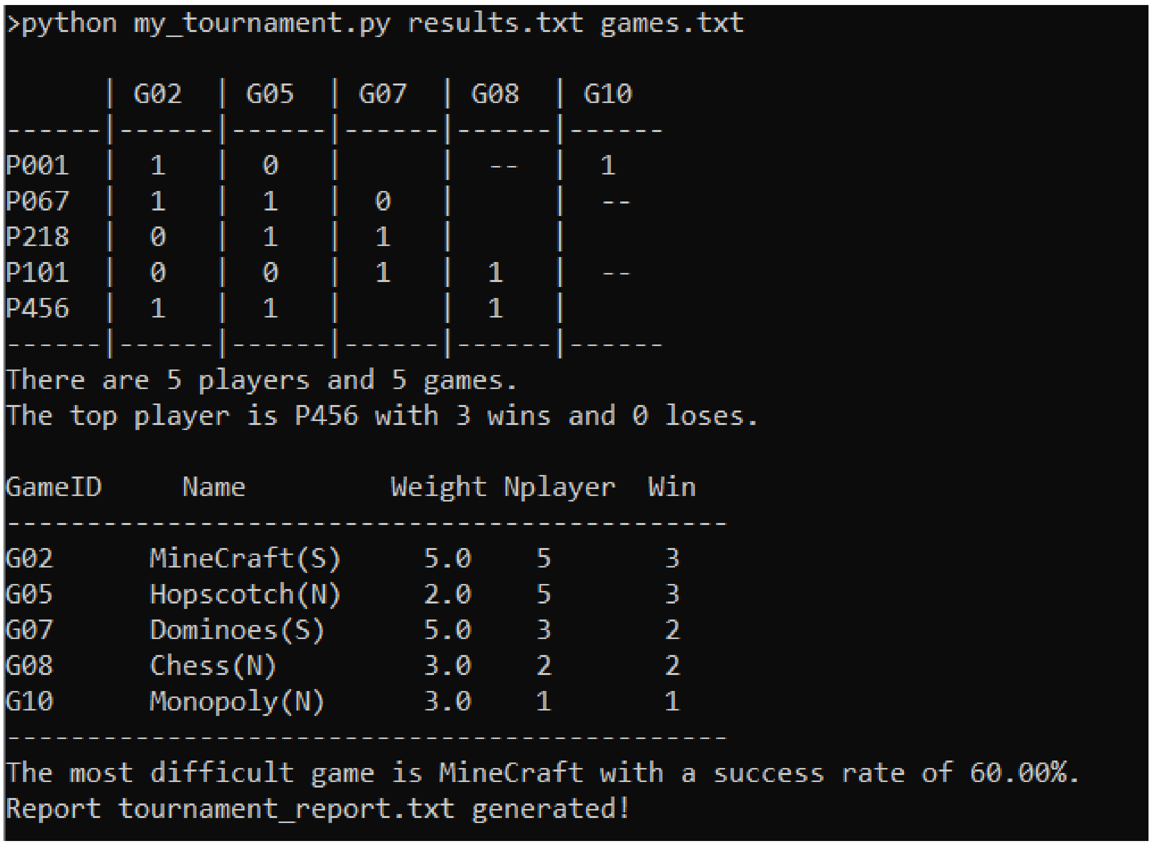
Also, a game should have a method to compute some statistics: number of players finished the game, number of wins, number of loses, number of on-going games, etc. (it is your choice to compute the necessary statistics to be used in your program). Also, you can define extra methods for the games if necessary.

Your program now can read one more file which stores the information of the games (see an example below). The file includes the game IDs, game names (all names are one-word names), the types of the games (*"S"* means special game and *"N"* means normal game), and the weight of each game. You can assume there are no duplicate or redundant games. You can assume all games available in the tournament appeared in this file (games file) and in the previous results file (in PART 1). And you can assume the format of the data in this file is always correct.



```
G05 Hopscotch N 2
G07 Dominoes S 5
G08 Chess N 3
G10 Monopoly N 3
G02 MineCraft S 5
```

Your program now can print the game summary on screen and save that summary into a file named *tournament_report.txt*. For example, given the above *games.txt* file and the *results.txt* file as in PART 1, the displayed message should look like below (note the content within the *tournament_report.txt* should also look the same).



```
>python my_tournament.py results.txt games.txt

      | G02  | G05  | G07  | G08  | G10
------|------|------|------|------|------
P001  |  1   |  0   |      |  --  |  1
P067  |  1   |  1   |  0   |      |  --
P218  |  0   |  1   |  1   |      |
P101  |  0   |  0   |  1   |  1   |  --
P456  |  1   |  1   |      |  1   |
------|------|------|------|------|------
There are 5 players and 5 games.
The top player is P456 with 3 wins and 0 loses.

GameID     Name         Weight Nplayer  Win
---------------------------------------------
G02      MineCraft(S)     5.0    5       3
G05      Hopscotch(N)     2.0    5       3
G07      Dominoes(S)      5.0    3       2
G08      Chess(N)         3.0    2       2
G10      Monopoly(N)      3.0    1       1
---------------------------------------------
The most difficult game is MineCraft with a success rate of 60.00%.
Report tournament_report.txt generated!
```

In the above message, the *Weight* column displays the weight with 1 digit after the decimal point. The *Nplayer* column displays the number of players who already finished the game (need to be integers), the *Win* column displays the number of players who won the game (need to be integers). Note, the players with ongoing games are not counted in these two columns. In the *Name* column, apart from the names of the games, your program should also display the type of the game (e.g. *"S"* stands for special game and *"N"* stands for normal game).

Note, apart from a game table, your program should also display a message indicating the most difficult game. The most difficult game is the one with the lowest success rate. The success rate is computed by dividing the number of wins over the number of players who finished playing the game (players with ongoing games are not counted). If there are multiple games with the lowest success rates, you can choose to either display one game or display multiple games.

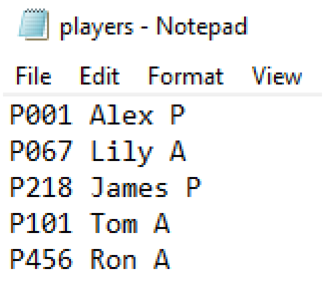
Finally, note that the printed message includes the printed message from PART 1 (and the content in the *tournament_report.txt* file should also include the table and the message from PART 1).

- PART 3 (You must only attempt this part after completing PART 2 – 4 marks) -

In this part, your program can support two types of players: *Professional Players* and *Amateur Players*. A professional player needs to play at least 1 special game and at least 4 games (in total) whilst an amateur player needs to play at least 2 games (in total) and does not require to play special games (note they can still play special games if they want). The classes defined for players should have methods to check whether a player meets these requirements. Also, a player should have a method to compute some statistics that are useful for the program, e.g. number of wins, number of loses, number of on-going games, etc. You can define extra methods for the players if necessary.

Your program now can read player information from a text file (which stores information about the players) from one more command line argument. The text file includes the IDs of the players, the names of the players and the types of the players (*"P"* stands for professional player, and *"A"* stands for amateur player). You can assume the names are only in one word, and they only contain the first names. You can assume there are no duplicate or redundant players. You can assume all players in the tournament appeared in this file and in the previous results file (in PART 1). You can assume the format of the file is always correct.

An example of the players text file is as follows.



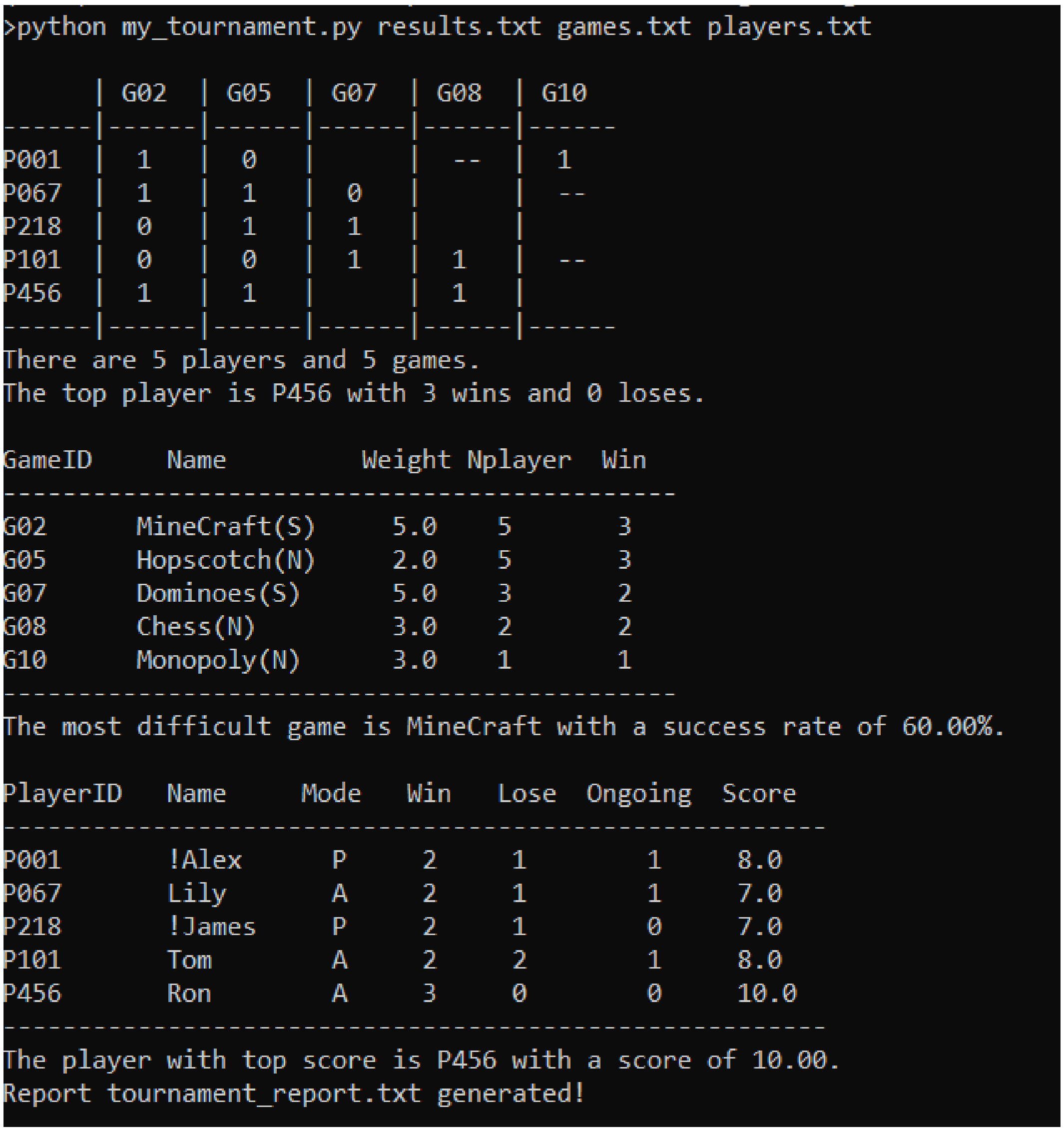
```
P001 Alex P
P067 Lily A
P218 James P
P101 Tom A
P456 Ron A
```

Your program now can print a summary of players on screen and store that summary in the text file *tournament_report.txt* (from PART 2). Given the *players.txt* file above, and the *games.txt* file in PART 2, the *results.txt* file in PART 1, the displayed message should look like below (and so does the content in the *tournament_report.txt* file) – see the next page. You can assume users always type the file names in the right order in the command line, e.g. the results file first, the games file second, and the players third.

The *Mode* column stores the types information of players (*"P"* for professional players and *"A"* is for amateur players). The *Win* column stores the number of wins of each player, the *Lose* column stores the number of loses of each player, and the *Ongoing* column stores the number of games each player starts playing but results are not made available. The *Score* column stores the score of each player. The scores are computed based on the weights of the games and the win/lose status of the player on those games. For example, if a player wins games G02 and G10 whilst losing game G05; and the game G02 has a weight of 5.0 whilst game G10 has a weight of 3.0, then the score of this player can be computed as 5.0*1 + 3.0*1 = 8.0. Note if a player has an on-going game (game without result yet), this won't be included in the score. If a player hasn't had any result yet, then the value in the *Score* column is a double dash (--).

In addition, in the *Name* column, if a player doesn't satisfy the requirement on the number of minimum special games or games, an exclamation mark (*!*) is added at the beginning of their names (e.g. *"!Alex"* and *"!James"* in the screenshot below).

Finally, a message indicating the player with the highest score will also be displayed. Note, the content in the *tournament_report.txt* needs to be the same as the printed message.



```
>python my_tournament.py results.txt games.txt players.txt

      | G02  | G05  | G07  | G08  | G10
------|------|------|------|------|------
P001  |  1   |  0   |      |  --  |  1
P067  |  1   |  1   |  0   |      |  --
P218  |  0   |  1   |  1   |      |
P101  |  0   |  0   |  1   |  1   |  --
P456  |  1   |  1   |      |  1   |
------|------|------|------|------|------
There are 5 players and 5 games.
The top player is P456 with 3 wins and 0 loses.

GameID     Name         Weight Nplayer  Win
---------------------------------------------
G02      MineCraft(S)     5.0    5       3
G05      Hopscotch(N)     2.0    5       3
G07      Dominoes(S)      5.0    3       2
G08      Chess(N)         3.0    2       2
G10      Monopoly(N)      3.0    1       1
---------------------------------------------
The most difficult game is MineCraft with a success rate of 60.00%.

PlayerID   Name     Mode   Win   Lose  Ongoing  Score
-------------------------------------------------------
P001        !Alex     P      2     1        1      8.0
P067        Lily      A      2     1        1      7.0
P218        !James    P      2     1        0      7.0
P101        Tom       A      2     2        1      8.0
P456        Ron       A      3     0        0      10.0
-------------------------------------------------------
The player with top score is P456 with a score of 10.00.
Report tournament_report.txt generated!
```

- PART 4 (You must only attempt this part after completing PART 3 – 8 marks) -

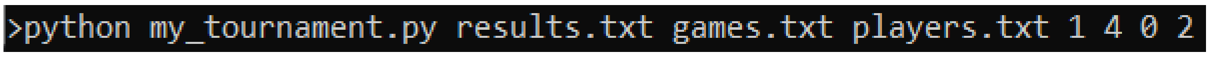
In this part, your program can handle some variations in the files:

1. When the results file is empty, your program will exit and display a message indicating no results are available for the tournament.

2. The results in the results file might be characters (e.g. *"NA"*, *"x"*). In these cases, they will be treated as same as *-1*, except *"TBA"* or *"tba"*, which means the game is ongoing (same as *503*).
3. When there are any formatting issues in any line of the files (e.g. commas are used instead of white space, the IDs are not in the right format, more columns compared to the required number of columns, etc.), the program will exit and display a message indicating there is a problem with the corresponding file.
4. When the files are missing or cannot be found, then your program should print a message indicating the files are missing and then quit gracefully. You can assume users always type the file names in the right order in the command line, e.g. the results file first, the games file second, and the players third.

The program will have some additional requirements:

1. The game table (produced from PART 2) is sorted (from high to low) based on the success rates. The player table (produced from PART 3) is sorted (from high to low) based on the player scores.
2. The *tournament_report.txt* is accumulated, which means when the program is run, it will not overwrite the previous report, but instead, it places the new report on top of the file (i.e. the newest report is always at the top of the *tournament_report.txt* file). In addition, the date and time when the report was generated (in the format *dd/mm/yyyy hh:mm:ss*, e.g. *01/03/2021 09:45:00*) are also saved in the text file for each report.
3. The program can now modify the requirement of minimum number of special games and total games for both professional players and amateur players by passing 4 **optional** additional command-line arguments. The order of these optional command line arguments is in the sequence: the minimum number of special games for professional players, the minimum number of total games for professional players, the minimum number of special games for amateur players, the minimum number of total games for amateur players. All these numbers need to be nonnegative integers. Your program should handle the case when users input invalid numbers (not integer, nonnegative integers). You can assume the users always pass in the numbers in the correct order. If users do not pass in these 4 optional arguments, the program will run using the default setting of the minimum number of special games and total games as in PART 3. If users pass in 1 to 3 additional arguments, then the program will exit and display a message saying that users should pass in 4 additional arguments (the additional arguments means the 4 numbers, not the text files).

   ```
   >python my_tournament.py results.txt games.txt players.txt 1 4 0 2
   ```

   In the printed message on screen and in the *tournament_report.txt*, your program should have one sentence stating the values of these 4 numbers that users passed in.
4. Exceptions are used to handle the above-mentioned invalid numbers.

B - Code Requirements:

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

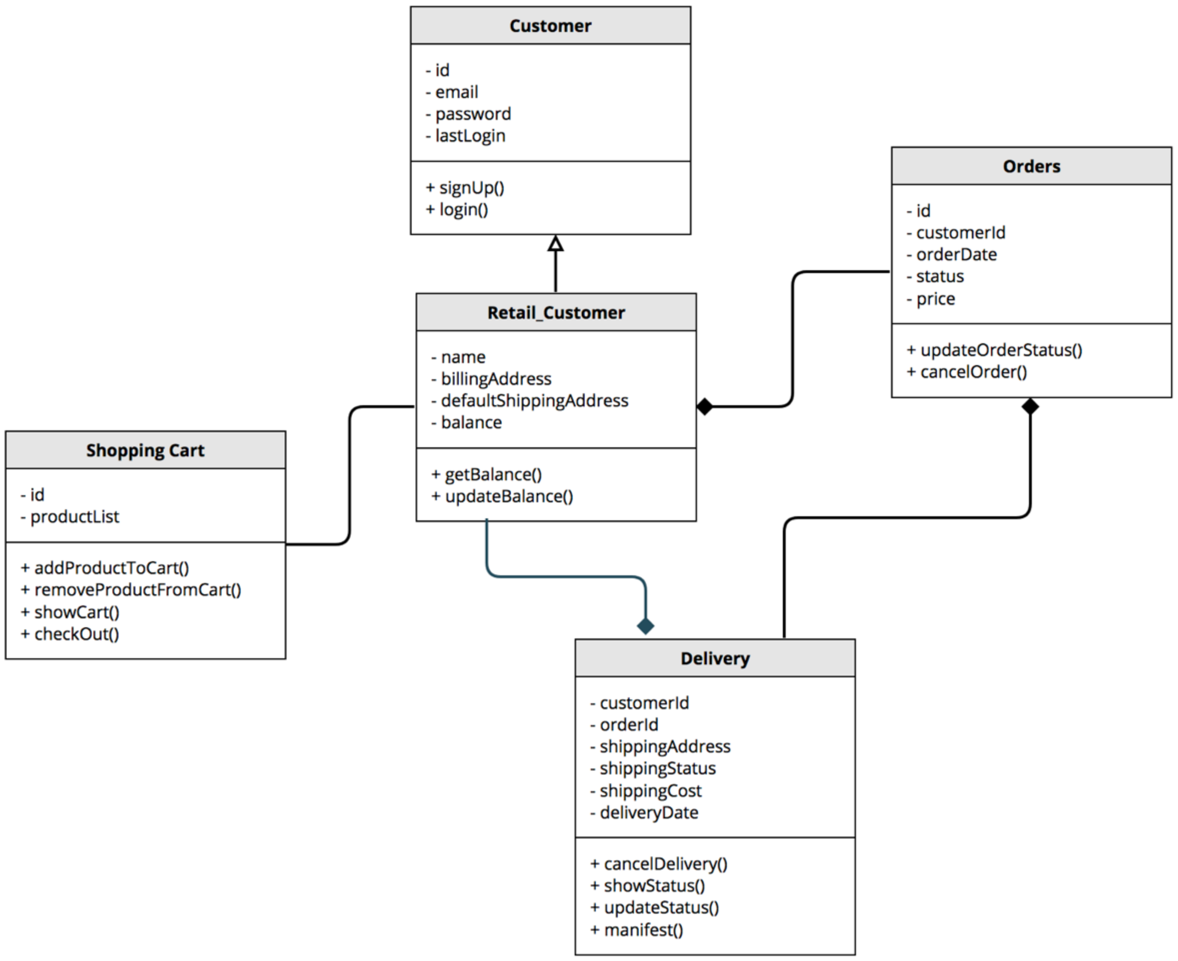
Your program at all levels should be fully OO, e.g. no variables, methods or code snippets dangling outside a class. Your main program should simply create an object and run its methods to invoke methods from other classes to perform the required functionalities.

You should test/verify the program with different text files (not just run with our text files) to ensure your program satisfy all the required functionalities.

In this challenge, your program has no interaction with users during execution. It simply runs the code, read the files, and display the desired messaged and/or generate the report text file.

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 design your classes carefully. You may use a diagram to assist the design. In PARTs 3 and 4, you are required to provide a detailed diagram to show your class design. An example of a diagram is as below (note this is a class diagram of a different programming assignment). In the diagram, variables and methods of each class are shown. Note if your code is at PART 1 or PART 2, you do not need to submit any diagram, a diagram would NOT result in any mark.



You could use tools like Powerpoint, Keynotes or online tools like moqups.com to draw the diagram. **The diagram needs to be submitted in jpg, gif, png or PDF format**.

Finally, 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 Final Coding Challenge.

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).

At the beginning of your Python file, your code must contain the following information:

1. **Your name and student ID.**
2. **The highest part that you have attempted.** This means you have completed all the requirements of the parts below. Mark will be only given at the lowest level of partial completion. For example, if you completed PART 1, tried 50% of PART 2, 30% of PART 3, 10% of PART 4, then your submission will be marked at the PART 2 only (we will ignore PART 3 and PART 4, so please make sure you fully finish one part before moving to the next one).
3. **Any problems of your code and requirements that you have not met.** For example, scenarios that might cause the program to crash or behave abnormally. Note, you do no need to handle or address errors that are not covered in the course.

Besides, the comments (documentation) in this final coding challenge 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:

Part	**Points**
Part 1	20
Part 2	4
Part 3	4
Part 4	8

Others (code quality, modularity, comments)	4

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

4. Submission

As mentioned in the Code Requirements, **you must submit only one zip file named ProgFunFinal_<Your Student ID>.zip** via Canvas/Assignments/Final Coding Challenge. The zip file contains:

- The main Python code of your program, named **my_tournament.py**
- A diagram **in one of the formats: JPG, GIF, PNG or PDF** (if you attempt PARTs 3 and 4)
- Other Python files written by you to be used by your main application.

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 zip file submitted is the one that will be marked. **NOTE, your code must be able to run under command-line.**

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 (or part of a day) late. For example, if an assignment is worth 40 marks and it is submitted 1 day late, a penalty of 10% or 4 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