COMP 6411: ASSIGNMENT 1

Important Info:

1. *Feel free to talk to other students about the assignment and exchange ideas. That's not a problem. However, when you write the code, you must do this yourself. Source code cannot be shared under any circumstance. This is considered to be plagiarism.*

2. *Assignments must be submitted on time in order to received full value. Appropriate late penalties (< 1 hour = 10%, < 24 hours = 25%) will be applied, as appropriate. You **MUST** verify the contents of your assignment AFTER you submit. You will be graded on the version submitted at the deadline – no other version will be accepted at a later date.*

3. *The graders will be using a Python 3.x interpreter to test your code. Make sure that your code is written and tested in a version 3.x environment (your docker installation uses this by default). Also ensure that your code can run directly from the command line.*

DESCRIPTION: In this assignment, you will have a chance to gain experience with the Python programming language. Python is an easy to use, dynamically typed Object Oriented language. It is syntactically similar to C-style languages like C++, C# and Java, but is somewhat simpler to understand.

You will be working with basic objects and various Python data structures in order to develop a simple console-based game that allows people to test their memory. The idea is as follows.

The game will display a square grid, with the columns labeled with letters, and the rows with numbers. Elements within the grid will be associated with pairs of integers (a pair of 0's, and pair or 1's, a pair of 2's, etc.). Initially, we will not be able to see any of the numbers, as they will be hidden by 'X' symbols. Your job as a player is to find the hidden pairs in as few guesses as possible. A simple menu will allow you to interact with the game.

So let's look at an example. The image to the right displays the game interface when using a 4x4 grid. You will note the following:

```
--------------------
|    PEEK-A-BOO    |
--------------------

     [A]  [B]  [C]  [D]

[0]  X    X    X    X

[1]  X    X    X    X

[2]  X    X    X    X

[3]  X    X    X    X

1. Let me select two elements
2. Uncover one element for me
3. I give up – reveal the grid
4. New game
5. Exit

Select:
```

- The title "PEEK-A-BOO" is displayed first.

- Next, we have the grid. Column names use letters (A to D in this case), while rows are labeled with numbers (0 to 3 in this case).
- Any individual cell in the grid is identified by a letter/number combination (e.g., A0, B2).
- When the game begins, all cells display an X (i.e., the number in the cell is 'hidden').
- Below the grid is a simple menu that allows you to play the game.
- The game title, grid, and menu will be displayed at all times.

Before we go any further, let's look at the pattern of numbers that are actually hidden behind the X symbols.

You will notice that Menu Option 3 is entitled "I give up – reveal the grid". In short, this ends the game and displays all hidden numbers. So if we select that option, we might see a grid display that looks like the image below.

```
     [A]  [B]  [C]  [D]
[0]  0    2    6    1
[1]  3    7    6    2
[2]  5    4    7    0
[3]  4    5    3    1
```

Note that a 4 x 4 grid has 16 cells in total. As such, it will hold 8 pairs of integers (from a pair of 0's up to a pair of 7's). In contrast, a 6 x 6 grid has 36 cells and therefore 18 pairs of integers (from 0's up to 17's).

So when a new game is started, the grid must be initialized with a new set of pairs, which should be randomly distributed throughout the grid. To demonstrate this feature, we can use Menu Option 4 to start a new game, and then use Menu Option 3 to reveal the new distribution.

```
     [A]  [B]  [C]  [D]
[0]  5    6    4    6
[1]  2    1    7    3
[2]  0    1    7    5
[3]  2    4    0    3
```

If we do that, we might see the set of numbers shown in the image on the left. Note that we again have 8 pairs of integers but the values are distributed in a completed different way. This, of course, is important since the player must not be able to simply memorize a fixed pattern.

Okay, so that's what the numbers would look like. So back to the game itself. Again, when we start a new game, all we see is a grid of X symbols. Our job is to find the pairs. We will do this primarily by selecting Menu Option 1, which will allow us to specify a pair of cells to test. When we do this, we will be prompted to enter the first cell (using the letter/number format) and then the second cell (i.e., there will be two separate input prompts, once for each cell). So the two input prompts will appear below the menu and might look like this:

```
Enter cell coordinates (e.g., a0): a0
Enter cell coordinates (e.g., a0): b0
```

Note that column letters may be given either as upper case or lower case (e.g., A or a). In any case, once this selection is made, the contents of the two cells will be revealed (i.e., the two X's will be replaced by the integers associated with each of the two cells. Early in the game, it is likely that your pair will not match, as depicted by the figure below (A0 = 7, B0 = 0). In this case, **the two numbers will remain visible for 2 seconds**, at which point they will be replaced by X's again. As a player,

```
     [A]  [B]  [C]  [D]

[0]  7    0    X    X

[1]  X    X    X    X

[2]  X    X    X    X

[3]  X    X    X    X
```

your job is to try to remember the contents of these cells as you uncover other cells in the grid, so that you can eventually match them with the proper cell.

Of course, if the two cells match, then the numbers will remain visible and the player continues to search for other pairs.

It is also possible for a player to manually turn cells over if they want to solve the game a little more quickly. To do so, they will select Menu Option 2. In this case, they will be prompted to enter the coordinates for a single cell. The associated X will be immediately replaced by the number in that cell.

So that's the basic idea. Using Menu Options 1 and 2 (but mostly Option 1), the player will work through the grid until she/he has uncovered all of the pairs in the grid. At that point the game will indicate that the player has won and will provide a score.

What is this score? In short, it is a number between 0 and 100 (higher is better) that represents how quickly the player was able to find all pairs. Basically, the score reflects the number of guesses actually taken *versus* the minimum number actually required. So, for example, the minimum number of guesses for a 4 x 4 grid would be 8 (one guess for each valid pair). If the player actually uses just 8 guesses (VERY unlikely), they will get a score of 100. The score itself is a very simple calculation, and is computed as follows:

```
Score = (minimum_possible_guesses / actual_guesses) * 100
```

That's it – the closer your actual guess count is to the minimum possible number of guesses, the closer your score will be to 100.

Note: Using Option 2 counts as <u>two guesses</u>, so using it a lot will produce a very low score.

The figure below illustrates the display when the player wins.

```
--------------------
|    PEEK-A-BOO    |
--------------------

     [A]  [B]

[0]  1    0

[1]  0    1

Oh Happy Day. You've won!! Your score is: 50.0

1. Let me select two elements
2. Uncover one element for me
3. I give up - reveal the grid
4. New game
5. Exit

Select:
```

In this case, the player has solved the puzzle but has done so in 4 guesses, which is twice as many as the minimum required for a 2 x 2 grid. So her score is 50.0.

Important: If a player solves the puzzle and has done so by making at least one valid guess, then the score will always be greater than 0.

So how does a player actually get a score of 0? Using the example above, it would be possible to use Menu Option 2 to manually uncover each of the four cells,

without ever making a single guess. In this case, the victory message would be replaced by the following text:

```
You cheated – Loser!. You're score is 0!
```

In practice, the game may be played with either a 2x2 grid, a 4x4 grid, or a 6x6 grid. The size of the grid will be passed as a command line argument when the program is run (i.e., either 2, 4, or 6).

One last thing that has to be mentioned is that the program must have basic error checking. Otherwise, the program will fail constantly, as the player accidentally inputs a bad selection. The error checking includes:

1. Check the command line argument to make sure that a 2, 4, or 6 has been specified. The program can be stopped immediately if the grid size is invalid.
2. Check the menu option selections. It must be a number between 1 and 5. If a bad selection has been made, the player must be informed and asked to enter the option again (the program should NOT just stop).
3. The cell coordinates must be verified to ensure that the column (a letter) and the row (the number) are valid for this grid. If not, the player should be informed and asked to enter the cell(s) again. The image below illustrates user feedback when invalid entries are made. Note as well that the player should not be able to enter the same cell twice when making a guess (e.g., a0 and a0). This will make things easier for you.

```
Select: 2
Enter cell coordinates (e.g., a0): z1
Input error: column entry is out of range for this grid. Please try again.

Enter cell coordinates (e.g., a0): a9
Input error: row entry is out of range for this grid. Please try again.
```

So that's it. The game is not hard to describe and is fairly simple in its logic. There is just a set of small steps that must be properly coordinated so that the game runs properly.

Useful info

The assignment is designed to allow you to focus on getting a better sense of what Python does (and doesn't do) relative to other languages. It is not about designing complex algorithms or exotic interfaces. So absolutely no external 3rd party Python libraries are required, nor should any be used. The code can be written with the modules that are included with the basic python distribution. The markers will not install any additional Python libraries in order to run your code.

In terms of the game display, no complex graphics or drawing functionality is required. To update the screen you will simply clear the screen and use a set of ***print()*** calls to draw the title, grid and menu again. Clearing the screen simply uses the following python statement:

```
os.system("clear")
```

This uses the ***system*** function in the python ***os*** module to invoke "clear" in the linux shell/command line. (Note that "clear" is linux-specific - which will of course work on a docker/linux installation)

Again, there are no "fancy" modules required for the assignment. But there are a few standard python modules that you might want to *import*, including:

- The ***os*** module (for clearing the screen)
- The ***time*** module (for pausing the program)
- The ***random*** module (for any random number generation)
- The ***sys*** module (for command line arguments and program termination)
- The ***string*** module (for additional string functions or properties)

You can, of course, use other standard Python modules, if you feel that they are useful.

There are also a number of built-in functions that might be useful, including:

- The ***int(arg)*** function converts a text/string value into an int (if possible)
- The ***len(list)*** function returns the length of a sequence
- The ***ord(letter)*** function converts a letter into a corresponding integer
- Various built-in string functions like ***isdigit(), upper(),*** and ***split().*** These functions can be called directly on the variable itself.
- In addition, lists have many functions like ***append()*** and ***pop()*** that might be useful.
- A list of general purpose built-in functions can be found at: https://docs.python.org/3/library/functions.html

One additional function worth pointing out is the ***input(prompt)*** function. This is incredibly simple to use and will allow you to collect input from the player.

As a final point, grading will be based entirely on the description provided in this document (basic game functionality, error checking, etc.). So there are no "hidden" grades related to things not described here. There are also no grades associated with general coding style or documentation.

DELIVERABLES: Your submission will have exactly two source files: `game.py`, and `grid.py`. The program itself will be run from `game.py`. In other words, assuming a grid size of 4, you might invoke the program as:

```
python3 game.py 4
```

The `game.py` file will contain code for the user interface (i.e., the menu, the various user prompts, error messages, and the display of the final score).

The `grid.py` file will contain the code for the grid content and any game/scoring data. You can design the grid structure(s) using any data structures that you like (lists, tuples, sets, dictionaries, etc.).

IMPORTANT: The code in the `grid.py` file MUST be written in an Object Oriented style (i.e., with one or more classes). In other words, the grid (and possibly its contents) must be represented as an object (or multiple objects). The code in the `game.py` file can either be object oriented or just simple functions, whatever you prefer.

Do NOT include any other files in your submission.

Your code MUST run from the command line. For grading purposes, both files will be placed in a single folder and the marker will run the program directly from that folder. So your program should not reference any special paths or locations.

Once you are ready to submit, simply combine your two python source files into a single zip file (please use `zip` and not `rar` or other compression formats). The name of the zip file will consist of "a1" + last name + first name + student ID + ".zip", using the underscore character "_" as the separator. For example, if your name is John Smith and your ID is "123456", then your zip file would be combined into a file called a1_Smith_John_123456.zip". The final zip file will be submitted through the course web site on Moodle. You simply upload the file using the link on the assignment web page.

Good Luck