CPSC 231 Assignment 4

A Robot Escape Game

(Learning Objectives: understand the use of classes and objects, and file operations)

Weight: 5% of final grade

Due date: Friday December 8, at 11:59pm Mountain Time (You are encouraged to submit on or before December 6, the last day of classes, since continuous tutorials run until December 6)

Submission: one Python .py file, submit on the D2L Dropbox. You may submit as many times as you wish, and only the latest submission will be graded.

Extensions: You may use your personal days to extend the deadline. An extension request must be submitted using the request form: https://forms.office.com/r/2wN7KNhYEK

You have a total of 5 personal days for the entire semester. No penalties for using personal days. All personal day extensions are automatically approved after a request form is submitted. An assignment will not be accepted if it is late with no approved extensions, other than in exceptional circumstances.

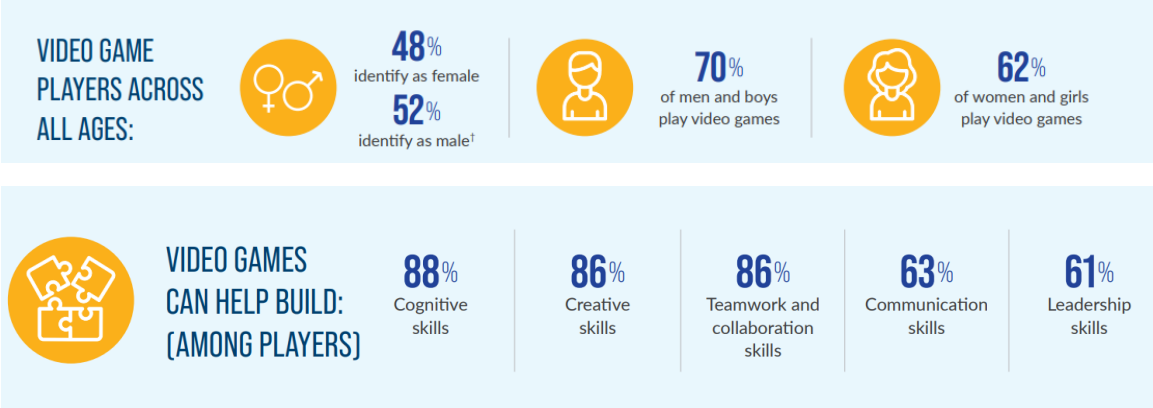
Academic Integrity: This work must be completed individually. Please follow academic integrity rules as discussed during lecture. You may discuss your ideas in English (not in Python) with other students as much as you like, but make sure that when you write your code that it is your own. A good rule of thumb is to never share your code with anyone, except your instructor and TAs.

Notes:

- You are only allowed to import anything that has been discussed by your instructor during lecture, such as import math, or import numpy.
- The existing file game.py imports your code in board.py. You should not change anything in game.py.
- You are allowed to define additional functions, classes, class variables, or methods as you see fit.
- For the purpose of the assignment, you are not allowed to use “break”, “continue”, “exit()”, or “quit()” in your code. Instead of stopping your code halfway when encountering some undesirable conditions, learn to adjust your conditions so that your code runs only in the desirable conditions.
- You would be assigned a grade of no higher than C if these rules are not followed.

Detailed Descriptions

The video game industry is a multi-billion dollar industry and it is growing so fast that it is making more money than movie, TV, and music industries combined.



Source: 2022 Essential Facts About the Video Game Industry, Entertainment Software Association.

In this assignment, you will create your own game called Robot Escape.

Step 1: Watch the video posted on D2L to see how the game is played when completed.

This is a puzzle game. There are up-to-four robots on the board at the start. Every step, every robot can move by one square (they have to all move in the same direction at the same time). The player's goal is to help all robots escape through the door. The walls are electrified, so if a robot hits a wall, the robot dies immediately.

Step 2:

Create a new py file to work on. Name your file **board.py.** If it is not called this name, rename it.

Step 3:

Download the new file, **game.py,** from D2L. game.py must be placed in the same directory as your own board.py file. Start the game by running game.py, not board.py.

If it says that you don't have **pygame**, please install it first. The video on D2L shows you how to do it using PyCharm.

Download **map.txt**, **players.txt**, and **exit.txt** from D2L and place them in the same directory.

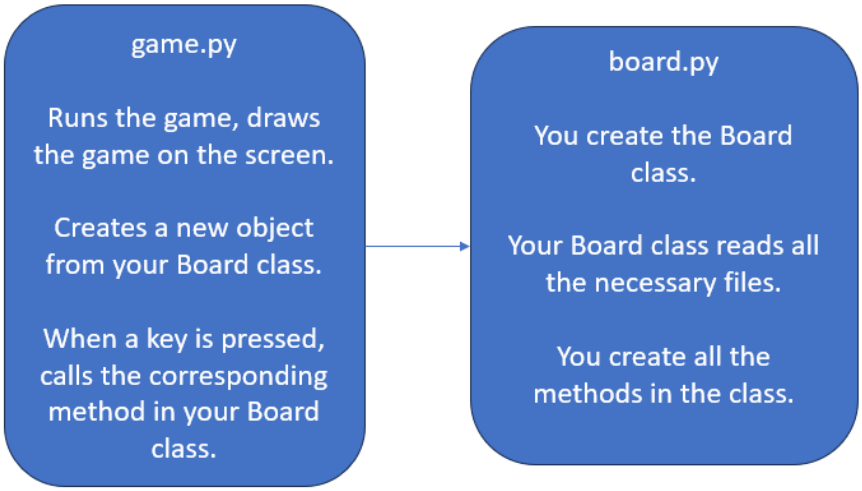
The **map.txt** file contains the layout of the map for the game, showing where the walls are and where the empty spaces are.

The **players.txt** file contains the starting locations of the robots.

The **exit.txt** file contains the location of the door.

While your code should work with any valid files as defined below, these are one set of examples for you to test with.

Here's an overview of the entire system:



Importantly, game.py doesn't know anything about the rules of the game – it just calls the methods you created, and draws whatever your methods tell it to draw.

Step 4:

Do not change anything in **game.py**. Start writing all your code in **board.py.**

Step 5: Create a new class called **Board**. Board has the following methods you must create:

__init__(self)

Returns: nothing.

The __init__ method should read the content from the map, the players, and the exit files. File reading should be protected with try/except. Even if the reading of any of these files failed, your code should not crash.

In the map file, # represents a wall. A space represents an empty space.

A valid map file always has 12 rows, and each row always has 16 columns. Other than that, your code has to read whatever is in the file. If the file contains any other symbol, treat it as an empty space.

A valid players file contains minimum 1, maximum 4 rows, with each row representing the row number and column number of a robot, separated by one single space. For example, 6 4 means this robot is at row 6 and column 4. Rows and columns start counting at 0 (just like every index in Python).

A valid exit file contains exactly 1 row, which contains the row number and column number of the door.

get_board(self)

Parameters: Nothing

Returns: a 2D list.

The return value is a 2D list of 12 rows by 16 columns. It represents the current board of the game. You can assume the board is always 12 by 16. Every item in the 2D list is a string of length 1:

"#" represents a wall.

"P" represents a robot controlled by the player.

"E" represents the door to escape to. There is only one door.

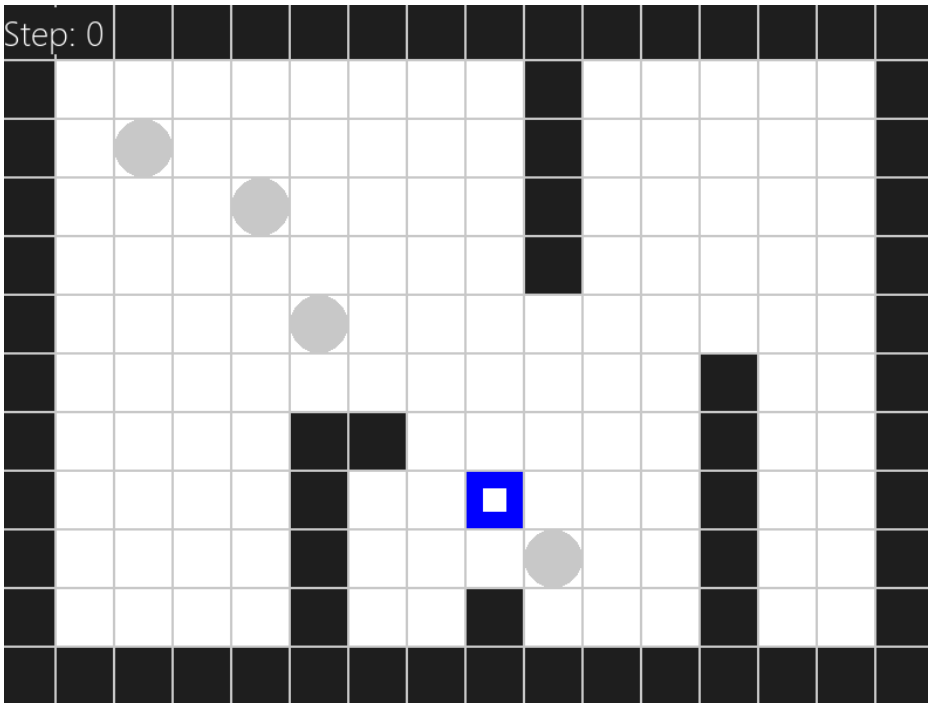
" " (one space) represents an empty space.

The 2D list should not contain any other symbols.

Here's an example of what it might look like:

```
[['#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#'],
 ['#', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', '#', ' ', ' ', ' ', ' ', ' ', '#'],
 ['#', ' ', 'P', ' ', ' ', ' ', ' ', ' ', ' ', '#', ' ', ' ', ' ', ' ', ' ', '#'],
 ['#', ' ', ' ', ' ', 'P', ' ', ' ', ' ', ' ', '#', ' ', ' ', ' ', ' ', ' ', '#'],
 ['#', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', '#', ' ', ' ', ' ', ' ', ' ', '#'],
 ['#', ' ', ' ', ' ', ' ', 'P', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', '#'],
 ['#', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', '#', ' ', ' ', '#'],
 ['#', ' ', ' ', ' ', ' ', '#', '#', ' ', ' ', ' ', ' ', ' ', '#', ' ', ' ', '#'],
 ['#', ' ', ' ', ' ', ' ', '#', ' ', ' ', 'E', ' ', ' ', ' ', '#', ' ', ' ', '#'],
 ['#', ' ', ' ', ' ', ' ', '#', ' ', ' ', ' ', 'P', ' ', ' ', '#', ' ', ' ', '#'],
 ['#', ' ', ' ', ' ', ' ', '#', ' ', ' ', '#', ' ', ' ', ' ', '#', ' ', ' ', '#'],
 ['#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#']]
```

game.py calls this method at every step. This 2D list is what game.py will draw, so whatever you return is whatever will show up on the screen:



update (self, direction)

Parameter: direction, a string whose value is one of the four: "U", "D", "L", "R"

Returns: Nothing.

This method updates the current location of the robots on the board according to the given direction parameter – moving every robot one square up, down, left, or right, depending on whether direction == "U", "D", "L", or "R".

A robot cannot walk through walls or another robot, or walk off screen.

- A robot hitting a wall dies immediately and does not appear on the board anymore.
- A robot hitting the door escapes successfully, and does not appear on the board anymore.
- If a robot on the edge of the screen tries to move off screen, it simply stays where it is. (*make sure your code doesn't crash because you tried to put the robot into row -1 or something*)
- If a robot tries to move onto another robot (all robots move at the same time, but the target robot cannot move due to on the edge of the screen, etc.), it simply stays where

it is. (*This is very difficult to code correctly. There will only be one test case testing this, so if you don't get this to work, don't worry too much*).

Once you created this method, you can run the game and press either WASD or the arrow keys, and this method will be called.

get_state(self)

Parameters: Nothing

Returns: a int.

Returning 1 means all robots escaped successfully, and there are no more robots on the board.

2 means all robots died, and there are no more robots on the board.

3 means some robots escaped and some died, and there are no more robots on the board.

0 means there are some robots on the board and the game is not over yet.

So 1, 2, or 3 means that the game is over in some way.

Once you created this method, you can run the game and the game will be able to end in some way by calling this method every step.

save_game(self)

Parameters: Nothing

Returns: Nothing

This method saves the entire current state of the game in a file (or files) in the current directory. It doesn't matter what the file is called (*but don't call it map.txt because you don't want to overwrite the starting map*), and it doesn't matter what format you use in the file, as long as your own code can read it in the next method.

In the game, press the J key will call this method.

load_game(self)

Parameters: Nothing

Returns: Nothing

This method reads the entire state of the game from a save file in the current directory created by your own save_game() method.

In the game, press the L key will call this method.

Basically, someone can play your game for a bit, save it by pressing J, exit the game, restart it later, and load the exact game state back from the save file by pressing L.

get_steps(self)

Parameters: Nothing

Returns: a int.

The return value is the number of steps taken so far in the game. Every time a WASD/arrow key is pressed, a step is taken. Step starts counting at 0 for a new game.

This method is called by game.py after loading a saved game, since the number of steps will have to be set for the newly loaded game.

Testing!

This is the last assignment, and at this point, hopefully you have developed a habit of testing your code. So, we will not give you access to an auto-grader. But don't be scared! It is easy to test this assignment code: This is a game – play it! If everything works in the game, your code is correct. If it doesn't work, then your code is not correct.

Create your own starting maps! We suggest creating 20 different starting maps to test with, and see if your code can correctly load all of them. Test different starting number of robots and different locations for robots. Play your game, and control your robots to hit everything you can – walls, edges, door, other robots… and see if the game behaves correctly. Play the game so you win, you lose, and everything in between. Save your game, exit, restart the game, play it for a bit, then load your save file. See if you can get the save state back. If it works, you can see it with your own eyes – but keep your eyes sharp, and especially see if the number of steps is loaded correctly.

Grading

Grade Point	Letter Grade	Guidelines subject to notes stated above
4	A+	Fulfill all assignment specs
4	A	One failed test
3.7	A-	Two failed tests
3.3	B+	Three failed tests
3	B	Four failed tests
2.7	B-	Five failed tests
2.3	C+	Six failed tests
2	C	Seven failed tests

1.7	C-	Eight failed tests
1.3	D+	Nine failed tests
1	D	More than nine failed tests, or code does not run due to syntax errors
0	F	Barely started code, or no submission, or late submission with no extension approved

Your submitted code should be clearly documented with comments. Comments need to include: your name, descriptions of what your code does, and citing any sources you have used to complete this assignment. Code without proper comments could receive a letter grade reduction.

The assignment will be graded out of 4, with the grade based on the code's level of functionality and conformance to the specifications.

As a final note:

While we do not permit the sharing of assignment solutions, you are free to build your own game and share/publish your own game wherever you want! You are permitted to reuse the code in game.py as a starting point to build your own game, as long as it does not look similar to the Robot Escape game we used in this assignment, and you don't publish your game calling it a CPSC231 Assignment. Have fun with it in the break!