Programming Lab 7 – Robot Maze Solver
PHYS 2511 – Prof. Matthew Newby – Spring 2018

Goal:	Build a simple game with "robots" navigating a maze.
Requirements:	See below.
Inputs:	• Select map
Outputs:	• The paths each robot has taken • The number of turns each robot takes to reach the goal •
Optional:	• Select robot (that is, the maze-solving strategy) • Graph of the path the robot took through the maze. • Allow multiple robots to race each other at the same time (this involves dealing with robot-robot collisions)

Background:

Similar to the board game "RoboRally," you will create a simple game where robots race to reach a goal. The game will consist of a "board," "robots," and a "referee," all in Python.

The board can have any configuration (training boards will be), but consist of:
- Open spaces ("_"): robots may move here.
- Wall spaces ("#"): robots may not move here.
- ~~Pits: ("U"): robots may move here, but are reset to their starting position if they do.~~ (Not implemented this semester)
- Starting spaces ("S"): like open spaces, but robots can start here.
- Goal: ("G"): like open spaces, but a robot wins when it arrives here.

Robots must move from a starting point to the goal. You will have to determine an algorithm to accomplish this, with the "referee" making sure that the robot cannot walk into walls (and can only take legal actions).

Robots start each maze "dumb" - they have no knowledge of their situation, and must be able to solve any solvable maze. They may learn and store information as you see fit.

Robots may only take the following actions:
- Move: move one space up, down, left, or right
- Look: the robot gets to "see" what is in the spaces it can currently move to.

Note: "Solving a maze" is very similar to several problem-solving techniques as applied to scientific data. In many of these algorithms, the "robot" is searching for the set of parameters that best-describe the data, given a specific model. (See "Maximum Likelihood Techniques" for more information)

If you wish to make this competitive:

The first robot to reach the goal is the winner, but the game continues until all robots reach the goal. Each member of your collaboration will code their own “robot,” but may collaborate with others as needed. Robots will take turns, executing one action per turn, with the allowed actions being:

- Move: move one space up, down, left, or right
- Look: the robot gets to “see” what is in the spaces it can currently move to.

A robot may store any information you like, and may run any functions, but once it takes one of the two actions above the other robot(s) get(s) a turn. Robots may always know their current location on the board, the location of the goal, and the outcome of their last action (for exmaple, “hit a wall”). Robots must start ignorant of the board layout.

The referee will manage the game. It will store the board information, the locations of the robots, evaluate each robot’s actions, deal with the consequences of movement (for example, falling into a pit or reaching the goal), and manage turns. The referee must also manage special situations:

- Place robots at random starting locations, and randomize the turn order.
- When a robot reaches the goal, it’s success will be logged (that is, how many turns it took to win), and it will be removed from the board.
- If a robot moves into another robot’s space, the moving robot pushes the stationary robot one space.
- ~~If a robot leaves the map, it will be treated as having fallen into a pit~~.
- A turn limit may be enforced to prevent bad robots from creating infinite loops.

Hints:

Build the referee first, and make sure it can store maps in an intelligent way. Then add “dumb” robots to make sure the referee works.

If you do this without classes, you will need to store each robot’s “knowledge” as global variables (that is, outside of functions). You could then create a function for each robot that returns the one action it will take on it’s turn, the referee would then call this function when it is that robot’s turn. For example:

```
goal_location = <i,j>              # a 2 part list or array
robot1_position = <i,j>            # a 2 part list or array
robot1_last action = “None”        # the robot’s last action
robot1_result = “None”             # the result of the robot’s last action
robot1_map = <2D array>            # if desired, the robot can save what it learns as it moves around

def robot1_action():
        # here you may make decisions based on the allowed current knowledge your robot has
        # then it must return something that the referee will interpret as a legal action,
        # such as “move left”, “look”, etc.
        return <that action>
```

If you use classes, the variables and functions can instead be part a class for each robot.