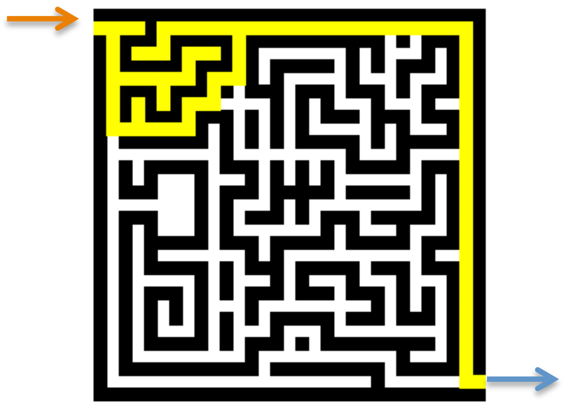
CS110 Project 1: Maze Solver

Prof. Karpenko

Due: Friday, October 17th, 2014, 11:59pm

While working on this project, you will be using many constructs we talked about in class in the past month: conditional statements, loops, nested loops, lists and files. The goal of the project is to read a maze from a text file, draw it on the screen using turtle graphics, and then solve the maze using the Wall Follower algorithm described below. Your program should draw your progress as you are solving the maze. Please use the template for project 1 that was emailed to you.

In the image below, the walls of the maze are shown in black, the entrance to the maze is on the top left (shown by the orange arrow in the image below), the exit is on the bottom right (shown by the blue arrow), and the path taken by the turtle from the entrance of the maze to the exit is shown in yellow.
Note: Your program does **not** need to draw the arrows.



Reading the maze file

This function should be called **readMaze** and take the file name with the maze as a parameter. The function should return the matrix for the maze as described below.

We have provided you with several text files with mazes (testMaze1, testMaze2, testMaze3). Each text file looks similar to the one shown below: # character is used for the wall, and white spaces between these characters describe corridors where one can "walk". The entrance is the white square in the first column of the second row; the exit is in the last column of the row that is last-1 row.

```
###########
      # # #
# ### # # #
# #   # # #
# ##### # #
#       # #
# ####### #
# #       #
# ####### #
#
###########
```

You need to read this file line by line and save the maze in a matrix represented in Python by **the list of lists**. Each line in this file will become a row in this matrix. In the matrix, the wall should be represented by an integer 1, and the corridor should be represented by the integer 0. Here is an example of how your list might look like (1 is the wall, 0 is the corridor):

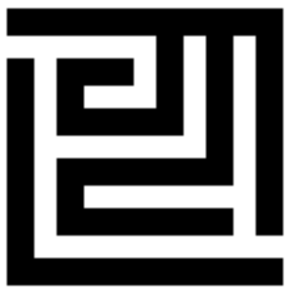
```
[ [ 1,1,1,1,1,1,1,1,1,1,1,1],
  [ 0,0,0,0,0,0,0,0,0,0,0,0],
  [ 0,0,1,0,1,0,1,0,0,0,0,0],
  [ 1,0,1,1,1,0,1,0,1,0,1],
  [ 1,0,0,0,0,0,0,0,0,1,0],
  [ 1,1,1,1,1,1,1,1,1,1,1 ] ]
```

Note: This matrix is **not** the matrix corresponding to the maze shown above, it's just a random example.

Function **readMaze** should **return this matrix.**

Drawing the maze

Given the matrix with the maze, draw it using turtle graphics. Each "cell" of the maze should be represented by a little black square (using a square with the side of length 10 works well):



Your code for drawing the maze should be in the function called **drawMaze** that takes three parameters: a maze represented by the matrix (the list of lists), a turtle, and a color:

def drawMaze(maze, t, col):

Start drawing the maze from the top left corner or the screen (define constants BEG_X, BEG_Y that will represent where you want to start drawing your maze. Something like -150, 150 might work well for the provided mazes). You would need to go along each row i of the matrix of the maze, and for each row i, you would go along all the columns (let us call the current column j). For each cell (i,j) of the maze, draw a little square (with side = 10) . Think about what formula you will use to compute a point x,y where you need to start drawing your square given i and j.

You should write a function that draws a single square corresponding to the cell (i, j) of the maze. **drawSquare** should take the following parameters: i , j (row and column of the maze), turtle t, the length of the side of the square, and color. Parameter side should be given a default value of 10, and the parameter color should be given a default value of "black":

def drawSquare(i, j, t, side=10, col="black"):

drawMaze can then call drawSquare for each cell (i,j) of the maze.

Speed

Drawing with turtle graphics can be slow. To speed up drawing the maze (especially when you are debugging your code), do the following: if **sc** is your screen object, call:

```
sc.tracer(0)
drawMaze(maze, t, "black")
sc.update()
```

Main function

You should have a function in the program called **main** that takes two parameters, the name of the file that contains a maze and the name of the algorithm to use for solving the maze ("WallFollower", "RandomMouse", "Pledge" or "FloodFill"). If you are not implementing the extra credit, you will call the main function with "WallFollower" as the algorithm.

Your main function should call readMaze to read the maze file into a matrix, then create the turtle and call drawMaze. For part 2, main should also call the appropriate function to solve the maze.

def main(filename, algorithm):

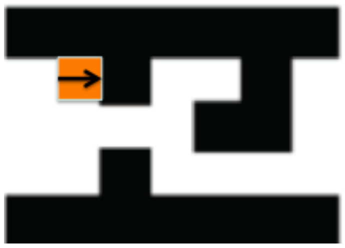
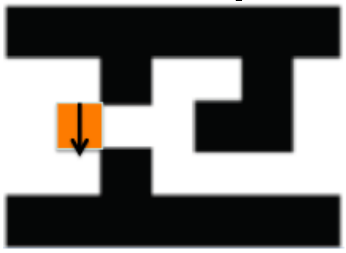
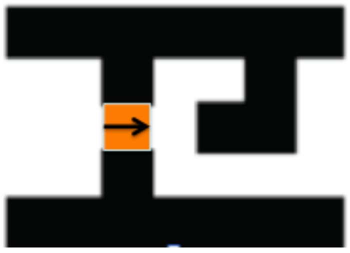
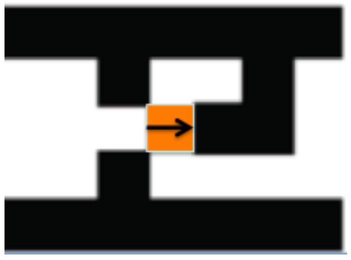
Important: Do not change signatures (names, parameters, etc.) of any functions!

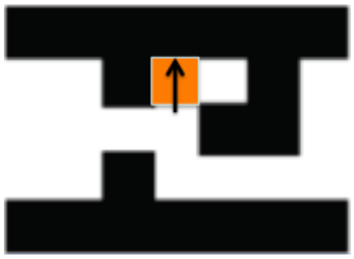
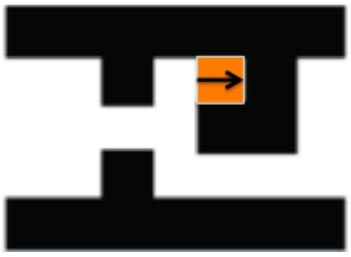
Part 2: Solving the maze

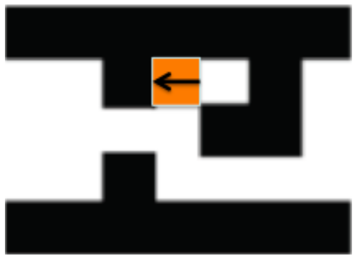
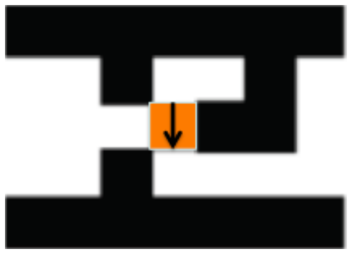
You are required to implement the algorithm called the **Wall Follower** for solving mazes. Wall Follower uses a simple rule: whenever you reach a junction, turn left if you can (or, alternatively, always turn right if you can). You will implement the left-hand version of the algorithm for project 1. Imagine a human (let us call her Alice) walking through the maze, holding her hand on the left wall as she walks around the maze. When Alice comes to a junction, she will do the following:

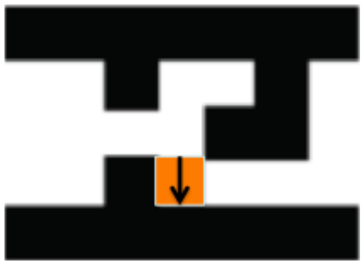
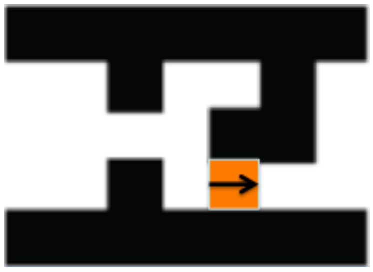
- go to the left if she can (if there is no wall blocking her from going to the left),
- If she cannot go to the left, she will go straight if she can (unless there is a wall there).
- If she can not go to the left or forward, she will go to the right (again, assuming there is no wall immediately on the right, blocking her path).
- If there are walls on the left, straight ahead, and on the right, Alice should turn around and go backwards.
- She should go in the same direction until she comes to a junction. When she is at a junction, use the algorithm we just described to decide where to go next.

Consider the following example (a) to (j). The walls of this simple maze are shown in black, the person walking through the maze is shown by the orange square. The black arrow shows the current direction in which the person is going.


(a)

(b)

(c)

(d)

(e)  (f) 

(g)  (h) 

(i)  (j) 

Implementation:

You will need to write the following two functions to implement the algorithm:

def followLeftWall(maze, t, col) :

We will start at the entrance of the maze (i =1, j =0), with the initial direction "east", and then will keep calling nextMove function to determine where to go next, until we reach the exit of the maze. The exit is at i = the last to last row, j = last row. After calling nextMove, we should update i and j accordingly. For instance, if nextMove returns "east", we should update j to j + 1 and i will remain the same.

This function should show the progress as it solves the maze: the current position in the maze should be shown by the yellow square. Here are the steps of the algorithm:

- Start at i =1, j = 0 (Entrance to the maze). Draw a yellow square there.
- Start with the initial direction set to "east"
- Repeat until you find the exit (exit is at i=last to last row, j = last row):
  - Call nextMove to determine where to go

  - Update i and j accordingly:
    - For instance, if nextMove is "east": update (i, j) to (i, j+1)
  - Draw a yellow square at (i,j)

def nextMove(maze, i, j, currDirection):
Computes the direction in which you need to go based on your current direction ("east", "west", "north" or "south") and the point where you are at in the maze (i, j). For instance, if you are currently going east, it will check if there is a wall on the left of you, and if not, your new direction will be "north"; if there is a wall on the left, it will check whether other options are possible: going "east", "south", "west", in this order of preference.
Note that when we say "turn left", we mean "turn left **relative to your current direction**". For instance, if you are currently moving "east", turning left means going "north"; while if you were going "north", turning left means going "west" etc. Before you start coding this function, go over the example above.

Your **main** function should call readMaze to read the maze file into a matrix, then create the turtle and call drawMaze. Finally, you should call followLeftWall from the main function.

See the extra credit section for the description of other maze solving algorithms you can implement.

Using constants

For this assignment, you are not allowed to use **hard-coded values**. Let us say I ask you to use 1 to represent a wall. If you use 1 everywhere in your code, it means it is "hard-coded". It's a bad idea. Instead, at the very top of your project file, **define a constant** WALL and make this constant 1.

```
WALL = 1
```

The names of constants are usually in capital letters. They are variables that are not supposed to change (their value does not change). After defining this constant, you can use the name WALL instead of using 1. For instance, when you are checking if the value at the cell (i. j) is 1, you can write

```
if maze[i]j[] == WALL:
```

Submission: Submit your projects by uploading the files to Canvas by the deadline.

Due date: October 17th, 2014, 11:59pm

Submit your fully completed **project1.py** (part 1 and part 2) with the following functions:

- main
- readMaze,
- drawSquare,
- drawMaze
- followLeftWall
- nextMove

The only function that you should call from your program should be main.

Look at the template file and make sure all your functions have correct signatures (name and parameters).

Thoroughly test your code before submitting it. Projects that do not run on IDLE 3.4 will be assigned an automatic 0. If you need help finding syntax errors in your code, please come see the instructor or the TAs.

Grading:

This project is worth 5% of your total grade. **You are not allowed to use any code from the web or collaborate on the project with anybody.** I will randomly select several people from each section of cs110, and ask them to come for an interactive code-walkthrough. If you submit the code that you can not explain to me during the code review, you will get a 0 for the project. As usual, you can ask the instructor, the TAs or tutors from the CS tutoring center for help.

Extra Credit: For extra credit, you will need to implement additional algorithms for solving the maze (Random Mouse, Flood Fill or Pledge Algorithm). Only start on the extra credit **after** you completed and thoroughly tested the required part of the project. Extra credit will not be graded if your main algorithm is not working correctly. Do not submit code from the web. You are not allowed to use recursion (that we did not cover yet) for implementing extra credit.

I described Random mouse and Flood Fill below. You can also read about the maze solving algorithms on Wikipedia:

http://en.wikipedia.org/wiki/Maze_solving_algorithm

Random Mouse Algorithm (1.5 pt)

- Start going in one direction; follow this passage through any turnings until you get to a junction.
- Choose randomly between straight, right and left
- If you can't go straight, left or right, then go backwards till the next intersection

Flood Fill Algorithm (2 pt)

See lecture 20 for the example.

- Pretend that you are standing at the entrance with the hose. Turn the hose on, the water will start filling the maze. Assume that the water can not go through the walls.
- We start with marking the entrance cell as REACHED (it means the water reached that cell)
- While we did not "reach" the exit:
    - Scan the whole maze (in i and j)
    - Consider the cell (i, j)
        - If it's a wall, or if the water already "reached" this cell, skip it
        - Otherwise set the value at (i,j) to REACHED if one of the immediate neighbors (to the north, south, east or west) is REACHED (you can set the REACHED constant to 2)

Pledge Algorithm (1.5 pt)

This is a modification of the Wall Follower Algorithm. We discussed it in class on Friday. Look it up on Wikipedia.