



ARAPAHOE COMMUNITY COLLEGE

PROJECT 1

Turtle Maze

Prepared for: CSC-119 Students

Prepared by:
Marshall Gaucher

Due December 8th at 11:59 AM MT

Beta:v1.0.0.0

Contents

0.1	Introduction	3
0.2	Problem	3
0.3	General Procedure	3
0.4	Maze Representation	4
0.5	Turtle Window Representation	5
0.6	Turtle Navigation	7
0.6.1	Manual	7
0.6.2	Autonomous	8
0.7	Measuring Path Time	9
0.8	Turtle Location Approximation	10
0.9	Program Design	11
0.10	Program Requirements	13

List of Figures

1	Python 2D Maze Exmaple	4
2	10x10 2D Maze: Start, End, Obstacle	5
3	Python Turtle Window Example	6
4	Python Turtle Window	6
5	Python Elapsed Time Example	9
6	Output of Turtle's location variance	10
7	Python Coordinate Translation (Maze - Turtle Window)	11
8	Program 1: High-level Architecture	12

0.1 Introduction

In Chapter 6 we learned about the Turtle module. We will take some of these concepts and design a Turtle Maze program. We will use the Turtle Graphic Window as a visual representation of an abstract map and a 2D list to represent the underlying maze model.

0.2 Problem

In this program you will design a maze for your Turtle to navigate through. You will ask the user to enter a start and an end location. The user can then select if they want to manually navigate the maze or autonomously navigate the maze. For this program you are only required to implement manual or autonomous navigation, not both. However, if you want to implement both you can for extra credit.

Once you have a valid start and end location, the Turtle can begin navigating. Along the way there might be some obstacles. You will write functions to create these obstacles. The Turtle must be able to navigate around the obstacles and reach the user's specified end point.

0.3 General Procedure

1. Prompt the user for a start point (x,y location)
2. Prompt the user for an end point (x,y location)
3. Prompt the user to create an obstacle
4. Update your maze with this obstacle
5. Allow the user to navigate the maze
 - (a) Manually
 - (b) Autonomously
6. Display start and end time
7. Display distance traveled
8. Output 2D maze values to a file

0.4 Maze Representation

The 2D maze is the underlying representation of your Turtle Graphic Window. It allows you to create representations of locations such as start, end, and obstacle. The below code illustrates how you can create a 2D maze, mark these locations, and output values to a file.

```
1  #mark a start location
2  maze[0][0]=1
3
4  #mark a end location
5  maze[2][2]=2
6
7  #mark a obstacle
8  maze[1][1]=3
9
10 #create a 10x10 maze
11 maze = [[0] * 10 for i in range(10)]
12
13 #open a file to output maze data
14 output_file = open("maze_data.txt", 'w')
15
16 for row in maze:
17     for column in row:
18         #output each row to file
19         output_file.write(str(column))
20 output_file.write('\n')
21
22 #close the maze data file
23 output_file.close()
```

Figure 1: Python 2D Maze Exmaple

1	0	0	0	0	0	0	0	0	0
0	3	0	0	0	0	0	0	0	0
0	0	2	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0

Figure 2: 10x10 2D Maze: Start, End, Obstacle

0.5 Turtle Window Representation

Python Turtle is a simple GUI library that allows users to implement graphics. This will be used as a visual representation of the underlying 2D maze. The below code shows how to create a basic Turtle window.

```
1  #import module needed for Turtle
2  import turtle
3
4  #create 200 x 200 pixel Turtle window
5  turtle.setup(200,200)
6  window = turtle.Screen()
7  window.title("Turtle Program 1")
8
9  #create a Turtle
10 the_turtle = turtle.getturtle()
11
12 #set the Turtle in the center of screen
13 the_turtle.setposition(0,0)
```

Figure 3: Python Turtle Window Example

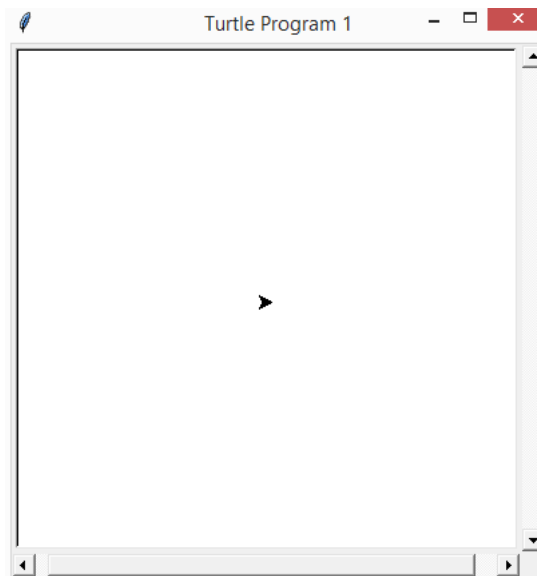


Figure 4: Python Turtle Window

0.6 Turtle Navigation

In Program 1 you must implement a mechanism for the user to navigate the Turtle around the Graphic Window. One way is to implement a key binding system with the arrow keys on your keyboard. This will allow you to capture each bound key pressed, by the user, and conditionally branch on this accordingly.

The alternative, and much more challenging approach, is to implement an autonomous algorithm that allows the Turtle to navigate from start to end point without user interaction. There are many ways to implement this feature, and the Autonomous subsection will provide some algorithms to explore.

0.6.1 Manual

To simplify our Turtle's navigation mechanism, we will use method bound keyboard events. This will allow us to move the turtle as the user presses arrow keys on their keyboard.

```
1  import turtle
2
3  def move_forward():
4      the_turtle.forward(1)
5      print("forward key pressed")
6      print("location: ", the_turtle.pos(), "heading: ",
7            the_turtle.heading())
8
9  def move_backward():
10     the_turtle.backward(1)
11     print("backward key pressed")
12     print("location: ", the_turtle.pos(), "heading: ",
13           the_turtle.heading())
14
15 def turn_left():
16     the_turtle.left(90)
17     print("left key pressed")
18     print("location: ", the_turtle.pos(), "heading: ",
19           the_turtle.heading())
20
21 def turn_right():
22     the_turtle.right(90)
23     print("right key pressed")
24     print("location: ", the_turtle.pos(), "heading: ",
25           the_turtle.heading())
26
27 #set Turtle window size
```



```
24 | turtle.setup(400,400)
25 |
26 | #get reference to Turtle window
27 | window = turtle.Screen()
28 |
29 | #set window title bar
30 | window.title("My first Turtle Graphics Program")
31 |
32 | #get a Turtle object
33 | the_turtle = turtle.getturtle()
34 |
35 | #set the turtle's position
36 | the_turtle.setposition(0,0)
37 |
38 | #bind up keyboard arrow to move_forward function
39 | window.onkey(move_forward, "Up")
40 |
41 | #bind down keyboard arrow to move_backward function
42 | window.onkey(move_backward, "Down")
43 |
44 | #bind right keyboard arrow to turn_right function
45 | window.onkey(turn_right, "Right")
46 |
47 | #bind left keyboard arrow to turn_left function
48 | window.onkey(turn_left, "Left")
49 |
50 | #setup event handler
51 | window.listen()
```

0.6.2 Autonomous

Below are some common path planning algorithms, you might consider implementing, to support the optional autonomous navigation feature in Program 1: Turtle Maze.

1. Depth-first search (DFS)
2. Breadth-first-search (BFS)
3. Dijkstra
4. A*
5. D*
6. Brushfire
7. Wavefront

0.7 Measuring Path Time

In this program we will measure the elapsed time of our Turtle traveling from start point to endpoint. The below code shows how to measure the elapsed time of a 2 second event.

```
1 import datetime
2 import time
3
4 #Get the current time
5 start_time = datetime.datetime.now()
6
7 #Sleep the program for 2 seconds
8 time.sleep(2)
9
10 #Get the time after the 2 second sleep
11 stop_time = datetime.datetime.now()
12
13 #Calculate time elapsed
14 time_elapsed = (stop_time - start_time)
15
16 print(time_elapsed)
```

Figure 5: Python Elapsed Time Example

0.8 Turtle Location Approximation

Recall early on in the course, we discussed Python's approximation of values as they become more precise. As you navigate the Turtle around, you will start to notice the x and y locations contain floating point precision(*e.g.* 2.07,3.02). This creates a small amount of variance in the translation of x y coordinates of the window, and the row column indexes of the maze.

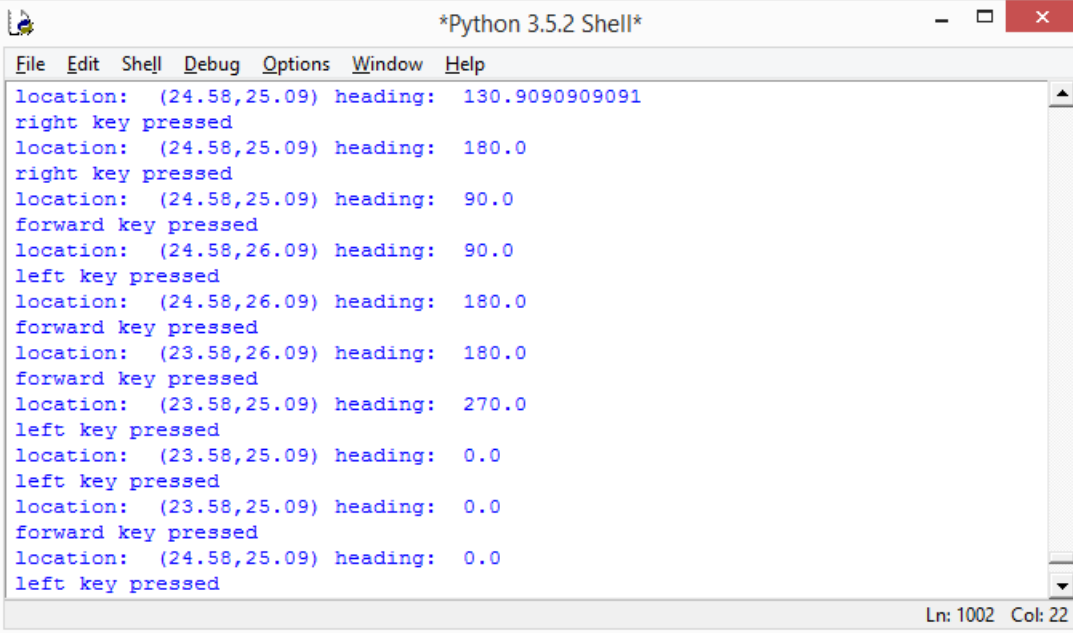
A screenshot of a Python 3.5.2 Shell window. The window has a menu bar with 'File', 'Edit', 'Shell', 'Debug', 'Options', 'Window', and 'Help'. The main text area displays a series of lines showing the Turtle's location and heading at various points, along with key presses. The locations are (24.58, 25.09), (24.58, 25.09), (24.58, 25.09), (24.58, 26.09), (24.58, 26.09), (23.58, 26.09), (23.58, 25.09), (23.58, 25.09), (23.58, 25.09), and (24.58, 25.09). The headings are 130.9090909091, 180.0, 90.0, 90.0, 180.0, 180.0, 270.0, 0.0, 0.0, and 0.0. The key presses are 'right key pressed', 'right key pressed', 'forward key pressed', 'left key pressed', 'forward key pressed', 'forward key pressed', 'left key pressed', 'left key pressed', 'forward key pressed', and 'left key pressed'. The status bar at the bottom right shows 'Ln: 1002 Col: 22'.

Figure 6: Output of Turtle's location variance

Note: 2.07,3.02 is not a valid index for a 2D maze; ideally we'd like 2,3 as this is a valid index in a 2D maze

To simplify the translation between the maze and the window, we must ensure a few design concepts are implemented:

1. Height/Width of the maze and window must be the same
2. Ensure the approximated indexes map to the window x y pixel locations
3. Round the positional points returned from turtle library

```
1 import turtle
2
3 #create 250 x 250 pixel Turtle Window
4 turtle.setup(250,250)
5 window = turtle.Screen()
6
7 #re-map the Turtle Window coordinate system to align
8 #with maze indexes
9 #use the top left corner of the Turtle Window to represent
10 #index 0,0
11 window.setworldcoordinates(0,250,250,0)
12
13 #create a 250 x 250 maze
14 maze = [[0] * 250 for i in range(250)]
```

Figure 7: Python Coordinate Translation (Maze - Turtle Window)

0.9 Program Design

Initially, the program prompts the user for start, end, and obstacle points from the user. Each set will contain a single x and y point. Once these points have been determined to be valid, the Turtle will be placed at the start location. Once the Turtle is placed on the start location, you can start the timer for elapsed time. The user will begin manually navigating the turtle to the marked end location. As the Turtle navigates the maze, it will constantly check if the user's intended movement is valid. If the intended movement is into an obstacle, the program will gracefully handle this condition for the user to continue on a valid path. Additionally, you will keep track of the Turtle's distance traveled. When the Turtle reaches the end location the elapsed timer will stop. The program will prompt the user indicating the Turtle has reached the end location and output the maze values to a file.

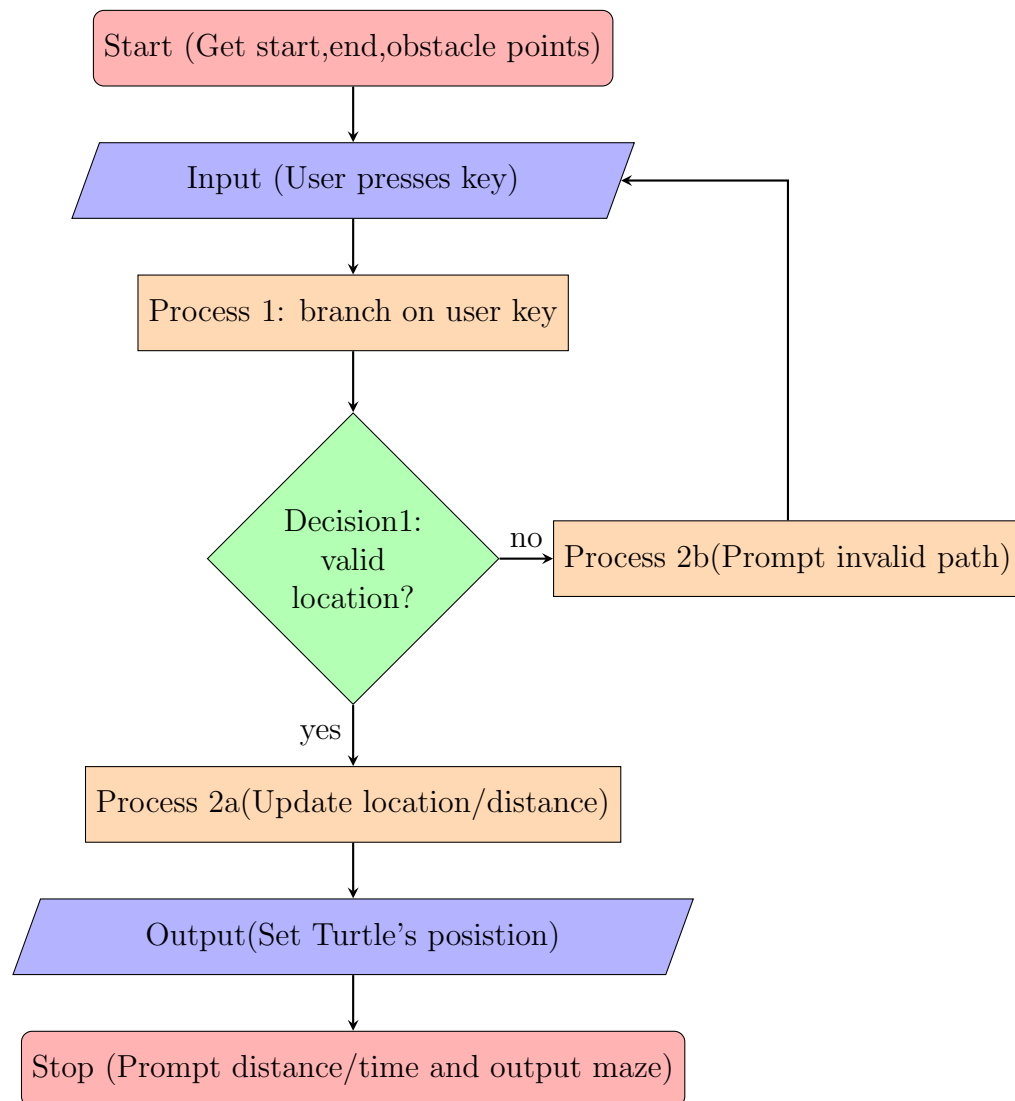


Figure 8: Program 1: High-level Architecture

0.10 Program Requirements

Below are the requirements for Program 1: Turtle Maze

1. The program **MUST** prompt user for a valid start location
2. The program **MUST** mark the start location in the Turtle Graphic Window visually
3. The program **MUST** prompt user for a valid end location
4. The program **MUST** mark the end location in the Turtle Graphic Window visually
5. The program **MUST** prompt user for a valid obstacle location
6. The program **MUST** mark the obstacle location in the Turtle Graphic Window visually
7. The start location **MUST** be marked in the maze with a unique value
8. The end location **MUST** be marked in the maze with a unique value
9. The obstacle location **MUST** be marked in the maze with a unique value
10. The user **MUST** be able to navigate valid paths in the maze
11. The program **MUST NOT** allow a user to navigate through an invalid path
12. The program **MUST** output a prompt to indicate an obstacle is blocking a valid path
13. The program **MUST** prompt the user once the Turtle has reached the end point
14. The program **MUST** output the total elapsed time of the Turtle's path
15. The program **MUST** output the total distance traveled by the Turtle
16. The program **MUST** output the final maze values to a file