**Theory and Foundations**

Everyone, at some point at his life, has tried to solve a maze. As defined in [Wikipedia](https://en.wikipedia.org/wiki/Maze), a maze is a path, or a collection of paths, typically from an entrance to a goal. In other words, there is an entrance and an exit and starting from the entrance, you need to navigate through the complex paths and walls and find a path towards the exit.

Of course, there are variations of this simple game. For example you can have multiple entrances or exits, you may have multiple paths that lead you to the exit and many more. In this article we are going to examine the simplest form. One entrance and one exit.

Even if a very big portion of everyone reading this has solved (or at least tried to solve) a maze, only a small percentage has created a maze out of thin air. Even less has wondered how can one create a “good” maze. There is a handful of algorithms out there that can be used to create mazes. Here we will examine the [Randomized Prim’s Algorithm](https://en.wikipedia.org/wiki/Maze_generation_algorithm).

Randomized Prim’s Algorithm consists of the following steps:

1. Start with a grid full of walls
2. Pick a cell, mark it as part of the maze. Add the walls of the cell to the walls of the list
3. While there are walls in the list:  
   1. Pick a random wall from the list. If only one of the two cells that the wall divides is visited, then:  
   a) Make the wall a passage and mark the unvisited cell as part of the maze  
   b) Add the neighboring walls of the cell to the wall list.  
   2. Remove the wall from the list

Let’s take a look and see how we can automate this.

**Implementation**

First, we need a maze. At first our maze will be empty. We have not decided if any block of the maze will be a cell or a wall. We will denote walls with ‘w’, cells with ‘c’ and unvisited blocks with ‘u’. So:

cell = 'c'  
wall = 'w'

For a fixed height and width, we will create a function that creates an empty maze.

def init\_maze(width, height):  
 maze = []  
 for i in range(0, height):  
 line = []  
 for j in range(0, width):  
 line.append('u')  
 maze.append(line) return maze

For debugging purposes, we will also create a function that prints the maze in a user friendly format. In order to be able to easily distinguish walls, cells and unvisited blocks, we will paint each letter with a different color, depending on the letter. To do so, we will use colorama .

from colorama import init, Fore# colorama needs to be initialized in order to be used  
init()def print\_maze(maze):  
 for i in range(0, len(maze)):  
 for j in range(0, len(maze[0])):  
 if maze[i][j] == 'u':  
 print(Fore.WHITE, f'{maze[i][j]}', end="")  
 elif maze[i][j] == 'c':  
 print(Fore.GREEN, f'{maze[i][j]}', end="")  
 else:  
 print(Fore.RED, f'{maze[i][j]}', end="")  
 print('\n')

Setting height=11 and width=27 we get this output:

u u u u u u u u u u u u u u u u u u u u u u u u u u uu u u u u u u u u u u u u u u u u u u u u u u u u u uu u u u u u u u u u u u u u u u u u u u u u u u u u uu u u u u u u u u u u u u u u u u u u u u u u u u u uu u u u u u u u u u u u u u u u u u u u u u u u u u uu u u u u u u u u u u u u u u u u u u u u u u u u u uu u u u u u u u u u u u u u u u u u u u u u u u u u uu u u u u u u u u u u u u u u u u u u u u u u u u u uu u u u u u u u u u u u u u u u u u u u u u u u u u uu u u u u u u u u u u u u u u u u u u u u u u u u u uu u u u u u u u u u u u u u u u u u u u u u u u u u u

Of course, you cannot see the color here, but if you run it, you will see that all the blocks are white. And this sums up the first step.

Step two instructs us to pick a spot in the maze and set it as a free spot. And then add the walls of it in a list. So first, let’s pick our starting points:

starting\_height = int(random.random()\*height)  
starting\_width = int(random.random()\*width)

We need to make sure that we do not start on a block that is on the edge of the maze:

if starting\_height == 0:  
 starting\_height += 1  
if starting\_height == height-1:  
 starting\_height -= 1if starting\_width == 0:  
 starting\_width += 1  
if starting\_width == width-1:  
 starting\_width -= 1

So now we will mark this block as a path and add the surrounding walls to the walls list:

maze[starting\_height][starting\_width] = cell  
walls = []  
walls.append([starting\_height-1, starting\_width])  
walls.append([starting\_height, starting\_width-1])  
walls.append([starting\_height, starting\_width+1])  
walls.append([starting\_height+1, starting\_width])

In order to complete step two, we need to denote the blocks round the starting cell as walls:

maze[starting\_height-1][starting\_width] = wall  
maze[starting\_height][starting\_width-1] = wall  
maze[starting\_height][starting\_width+1] = wall  
maze[starting\_height+1][starting\_width] = wall

Step three is the complex one in this algorithm. Let’s break it down.

*While there are walls in the list pick a random wall from the list*

This is simple enough to start and build on this as we progress:

while walls:  
 rand\_wall = walls[int(random.random()\*len(walls))-1]

The next instruction is:

*If only one of the two cells that the wall divides is visited*

Actually this is a condition. We need to check the surrounding blocks of the wall we are currently processing. Remember that we picked a wall at random in the previous step. We need to check if *the two cells that wall divides*. But how does a wall divide two cells?

We have two possibilities here. First we need to check the blocks to the left and right of the wall we are processing and then we need to check the blocks above and below the wall. Let’s visualize it in order to understand it:

Case 1 (we check the blocks at left and right of the selected wall): u  
**u** w **c**  
 uCase 2 (we check the blocks above and below the selected wall):  
 **u**  
u w u  
 **c**

This are 2 cases that our condition will hold. Of course, it can be mirrored, so we have 4 cases in general. Let’s add this check to our code:

while walls:  
 rand\_wall = walls[int(random.random()\*len(walls))-1] if maze[rand\_wall[0]][rand\_wall[1]-1] == 'u' and maze[rand\_wall[0]][rand\_wall[1]+1] == 'c': if maze[rand\_wall[0]-1][rand\_wall[1]] == 'u' and maze[rand\_wall[0]+1][rand\_wall[1]+1] == 'c': if maze[rand\_wall[0]+1][rand\_wall[1]] == 'u' and maze[rand\_wall[0]-1][rand\_wall[1]] == 'c': if maze[rand\_wall[0]][rand\_wall[1]+1] == 'u' and maze[rand\_wall[0]][rand\_wall[1]-1] == 'c':

Here, we need to be extra careful. We are accessing data inside a list using indexes. We need to make sure we are always accessing a correct index. Meaning that if the selected wall is the one on the first line of the maze, we cannot go and check the cell above it, since it would create an IndexError in Python. So let’s add these checks:

while walls:  
 rand\_wall = walls[int(random.random()\*len(walls))-1] if rand\_wall[1] != 0:  
 if maze[rand\_wall[0]][rand\_wall[1]-1] == 'u' and maze[rand\_wall[0]][rand\_wall[1]+1] == 'c': if rand\_wall[0] != 0:   
 if maze[rand\_wall[0]-1][rand\_wall[1]] == 'u' and maze[rand\_wall[0]+1][rand\_wall[1]+1] == 'c': if rand\_wall[0] != height-1:  
 if maze[rand\_wall[0]+1][rand\_wall[1]] == 'u' and maze[rand\_wall[0]-1][rand\_wall[1]] == 'c': if rand\_wall[1] != width-1:  
 if maze[rand\_wall[0]][rand\_wall[1]+1] == 'u' and maze[rand\_wall[0]][rand\_wall[1]-1] == 'c':

Now that we added this logic, we can continue. If any of the four conditions holds, then:

*Make the wall a passage and mark the unvisited cell as part of the maze*

This part is a little bit tricky. We need to make sure that every wall that we are going to turn into passage, does not have more than one cell around it. If we do not make this kind of check now, we will end up with a maze that has clusters of passages and the maze will not be good. So we will add an extra check if the surrounding cells are less than two. But first we need to create a function that checks that:

def surroundingCells(rand\_wall):  
 s\_cells = 0  
 if (maze[rand\_wall[0]-1][rand\_wall[1]] == 'c'):  
 s\_cells += 1  
 if (maze[rand\_wall[0]+1][rand\_wall[1]] == 'c'):  
 s\_cells += 1  
 if (maze[rand\_wall[0]][rand\_wall[1]-1] == 'c'):  
 s\_cells +=1  
 if (maze[rand\_wall[0]][rand\_wall[1]+1] == 'c'):  
 s\_cells += 1 return s\_cells

So now we can incorporate that code in the main function:

while walls:  
 rand\_wall = walls[int(random.random()\*len(walls))-1] if rand\_wall[1] != 0:  
 if maze[rand\_wall[0]][rand\_wall[1]-1] == 'u' and maze[rand\_wall[0]][rand\_wall[1]+1] == 'c':  
 s\_cells = surroundingCells(rand\_wall)  
 if s\_cells < 2:

For simplicity and readability purposes I am adding the extra code needed only in the first case. But the same applies for the rest.

After all those conditions hold, we can finally turn this wall into a passage and turn the surrounding walls to maze:

while walls:  
 rand\_wall = walls[int(random.random()\*len(walls))-1]if rand\_wall[1] != 0:  
 if maze[rand\_wall[0]][rand\_wall[1]-1] == 'u' and maze[rand\_wall[0]][rand\_wall[1]+1] == 'c':  
 s\_cells = surroundingCells(rand\_wall)  
 if s\_cells < 2:  
 maze[rand\_wall[0]][rand\_wall[1]] = 'c'  
   
 if (rand\_wall[0] != 0):  
 if (maze[rand\_wall[0]-1][rand\_wall[1]] != 'c'):  
 maze[rand\_wall[0]-1][rand\_wall[1]] = 'w'  
 if ([rand\_wall[0]-1, rand\_wall[1]] not in walls):  
 walls.append([rand\_wall[0]-1, rand\_wall[1]])

Again for simplicity purposes, I am adding only one of the surrounding walls. Of course, we also need to check if we are trying to accessing an invalid index and that we are not adding already existing walls in our list. The code above also completes the next step that is:

*Add the neighboring walls of the cell to the wall list*

Finally, we need to remove this processed block from the walls list. Then we need to continue with the next iteration. Let’s create a function for that:

def delete\_wall(rand\_wall):  
 for wall in walls:  
 if (wall[0] == rand\_wall[0] and wall[1] == rand\_wall[1]):  
 walls.remove(wall)

Add this function to our code. If the wall we are processing does not fall in any of the 4 cases we had earlier, we also need to delete it from the list:

while walls:  
 rand\_wall = walls[int(random.random()\*len(walls))-1]if rand\_wall[1] != 0:  
 if maze[rand\_wall[0]][rand\_wall[1]-1] == 'u' and maze[rand\_wall[0]][rand\_wall[1]+1] == 'c':  
 s\_cells = surroundingCells(rand\_wall)  
 if s\_cells < 2:  
 maze[rand\_wall[0]][rand\_wall[1]] = 'c'  
   
 if (rand\_wall[0] != 0):  
 if (maze[rand\_wall[0]-1][rand\_wall[1]] != 'c'):  
 maze[rand\_wall[0]-1][rand\_wall[1]] = 'w'  
 if ([rand\_wall[0]-1, rand\_wall[1]] not in walls):  
 walls.append([rand\_wall[0]-1, rand\_wall[1]])  
 delete\_wall(rand\_wall)  
 continue  
 continue

If you print the maze when the list of walls is empty, you will notice that there are some cells that will be unvisited. You need to make them walls:

def make\_walls(width, height):  
 for i in range(0, height):  
 for j in range(0, width):  
 if (maze[i][j] == 'u'):  
 maze[i][j] = 'w'

Finally we need to create an entrance and an exit for the maze:

def create\_entrance\_exit(width, height):  
 for i in range(0, width):  
 if (maze[1][i] == 'c'):  
 maze[0][i] = 'c'  
 break  
 for i in range(width-1, 0, -1):  
 if (maze[height-2][i] == 'c'):  
 maze[height-1][i] = 'c'  
 break

**Putting it all together**

If we define the while loop as a function (let’s call it create\_maze ), then we can put this all together and test our script:

cell = 'c'  
wall = 'w'  
unvisited = 'u'  
height = 11  
width = 27maze = init\_maze(width, height)  
print\_maze(maze)starting\_height = int(random.random()\*height)  
starting\_width = int(random.random()\*width)starting\_height == 0:  
 starting\_height += 1  
if starting\_height == height-1:  
 starting\_height -= 1if starting\_width == 0:  
 starting\_width += 1  
if starting\_width == width-1:  
 starting\_width -= 1maze[starting\_height][starting\_width] = cell  
walls = []  
walls.append([starting\_height-1, starting\_width])  
walls.append([starting\_height, starting\_width-1])  
walls.append([starting\_height, starting\_width+1])  
walls.append([starting\_height+1, starting\_width])maze[starting\_height-1][starting\_width] = wall  
maze[starting\_height][starting\_width-1] = wall  
maze[starting\_height][starting\_width+1] = wall  
maze[starting\_height+1][starting\_width] = wallcreate\_maze()  
make\_walls(width, height)  
create\_entrance\_exit(width, height)print\_maze(maze)

And this is pretty much it! Let’s run it and see what comes up!



The final maze

As you can see, we have a maze! Of course, you can change the width and the height to create larger or smaller mazes. Also, by changing the order we are checking the 4 cases may change the structure of our maze. Feel free to try different variations and let me know about the results you get.

The full functional code can be found [here](https://github.com/OrWestSide/python/blob/master/maze.py).