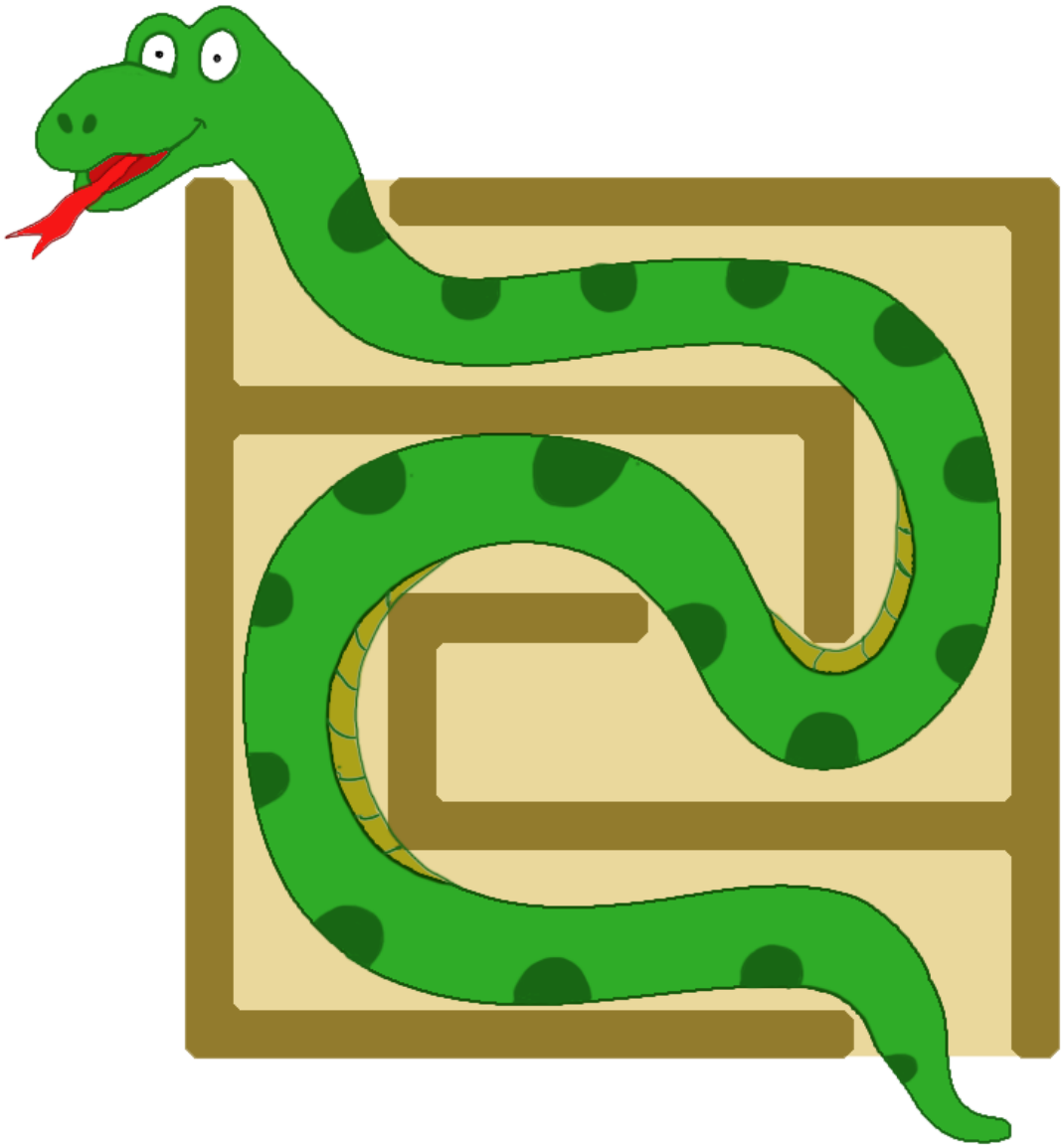
Python-Maze-World-pyamaze-

The module **pyamaze** is created for the easy generation of random maze and apply different search algorithm efficiently.

The main idea of this module, **pyamaze**, is to assist in creating customizable random mazes and be able to work on that, like applying the search algorithm with much ease. By using this module, you don't need to program the GUI and also you don't need the Object-Oriented Programming since the module will provide you the support. This module uses the **Tkinter** GUI framework which is built-in in Python and you don't need to install any framework to use this module.

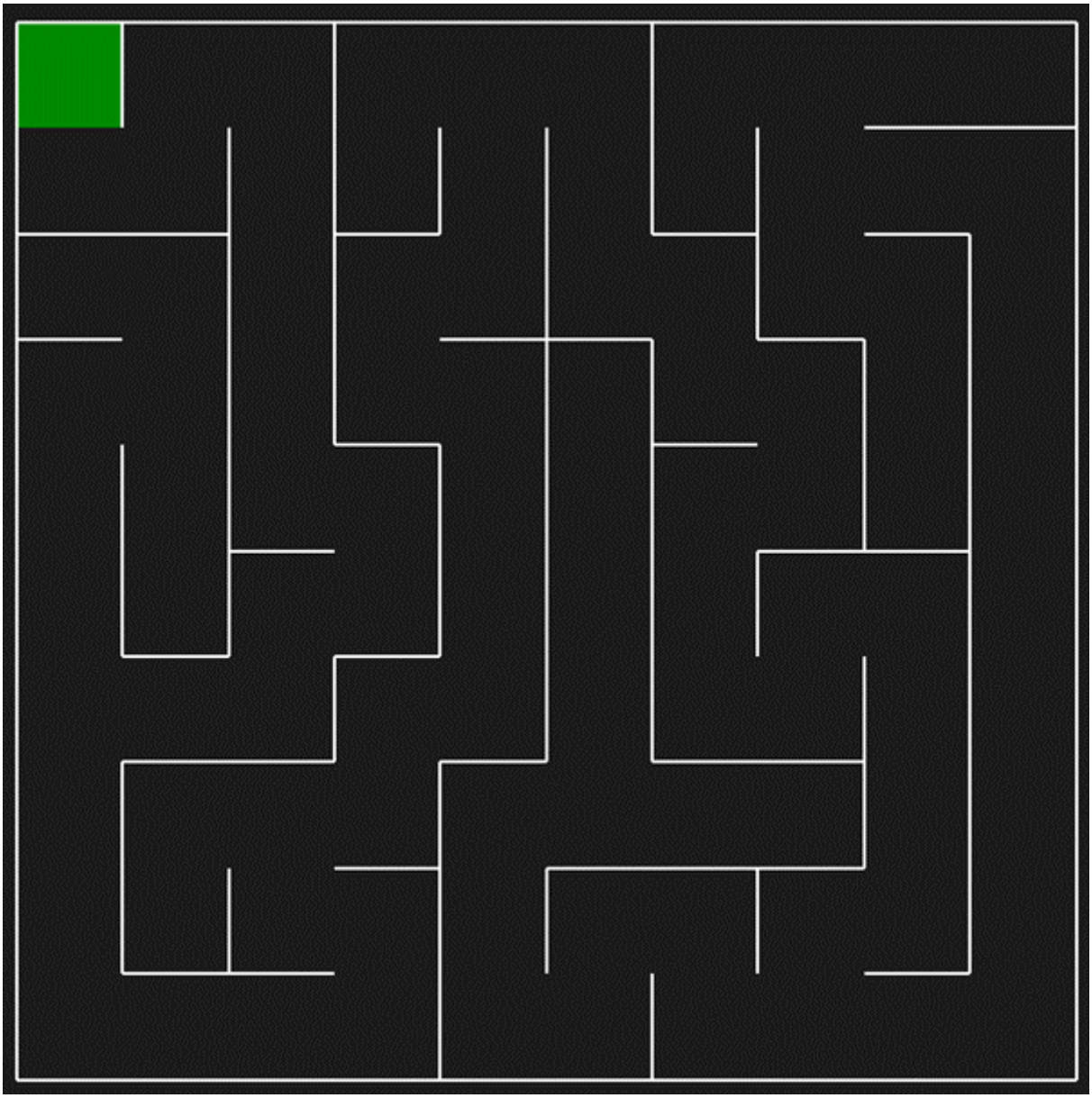


Here is the detail on how to use the module

Generate a Maze: To simply generate a maze, you need to create the maze object and then apply the ***CreateMaze*** function. The last statement will be applying the function ***run*** to run the simulation.

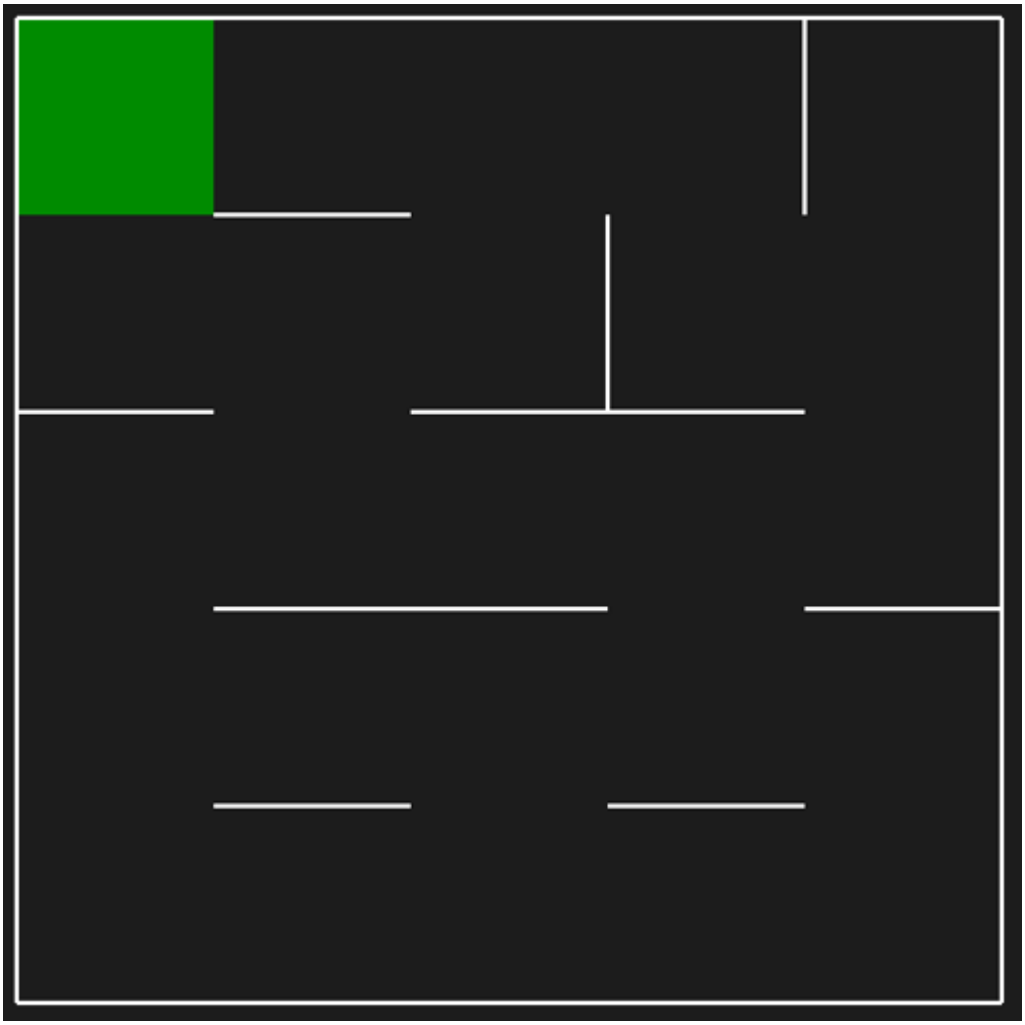
```
from pyamaze import maze
m=maze()
m.CreateMaze()
m.run()
```

A random 10x10 maze will be generated like this:



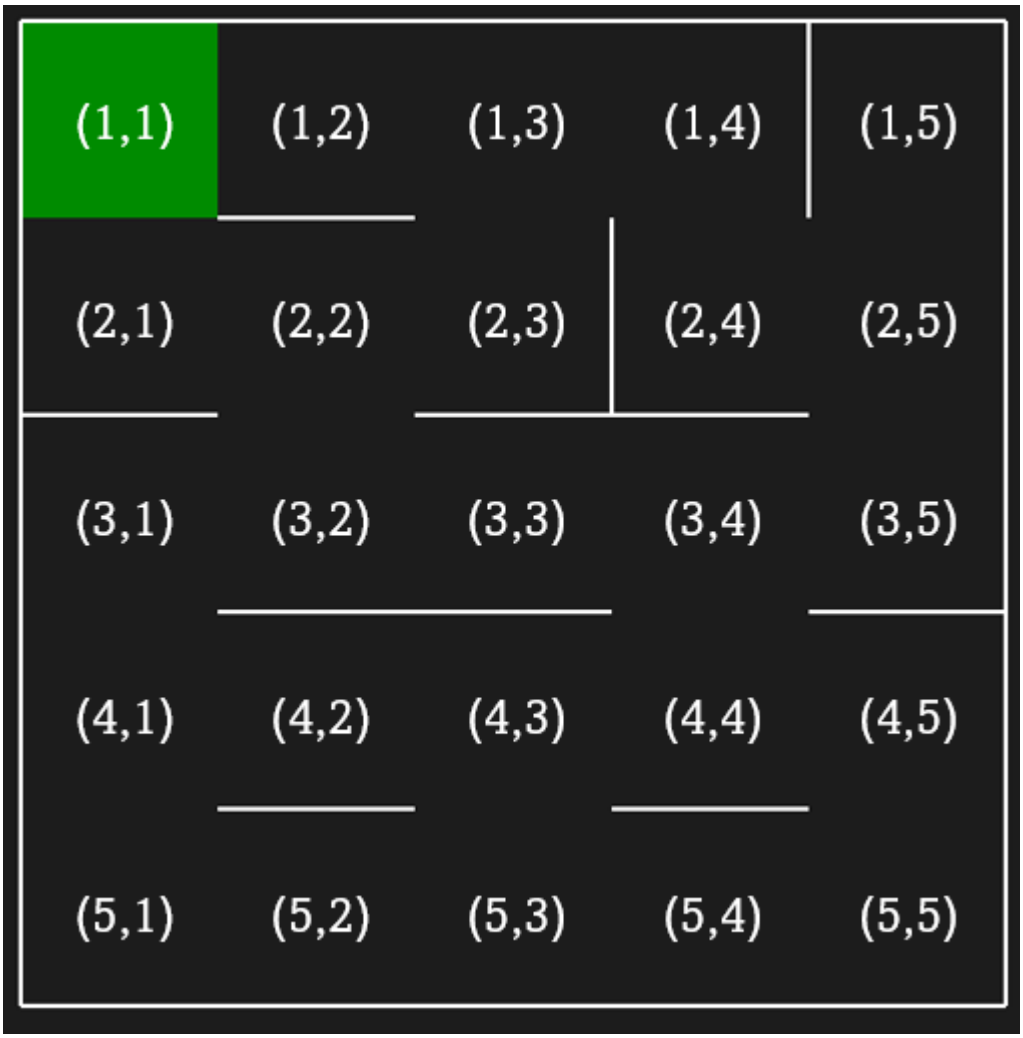
The top-left cell is the goal of the maze and there are ways we can change the goal to any cell. Moreover, by default, a **Perfect Maze** is generated, which means all cells of the maze are accessible and there is one and only one path from any cell to the goal cell. Therefore any cell can be treated as the start cell and hence is not highlighted. Internally the last cell i.e. the last row and last column cell is set as the start cell. So, the general task will be finding the way from the bottom-right cell to the top-left cell. We can change the size of the maze while creating that. For example, a 5x5 maze can be generated as:

```
from pyamaze import maze
m=maze(5,5)
m.CreateMaze()
m.run()
```



The first argument of the ***CreateMaze*** function is the row number and the second is the column number. To generate a maze of size 15x20, you should use the function as `m.CreateMaze(15,20)`.

It is important to know the maze parameters in order to use those in the program. Firstly the cells of the maze have two indices, one for row and the other for the column. The indices of the 5x5 maze generated previously are shown here:

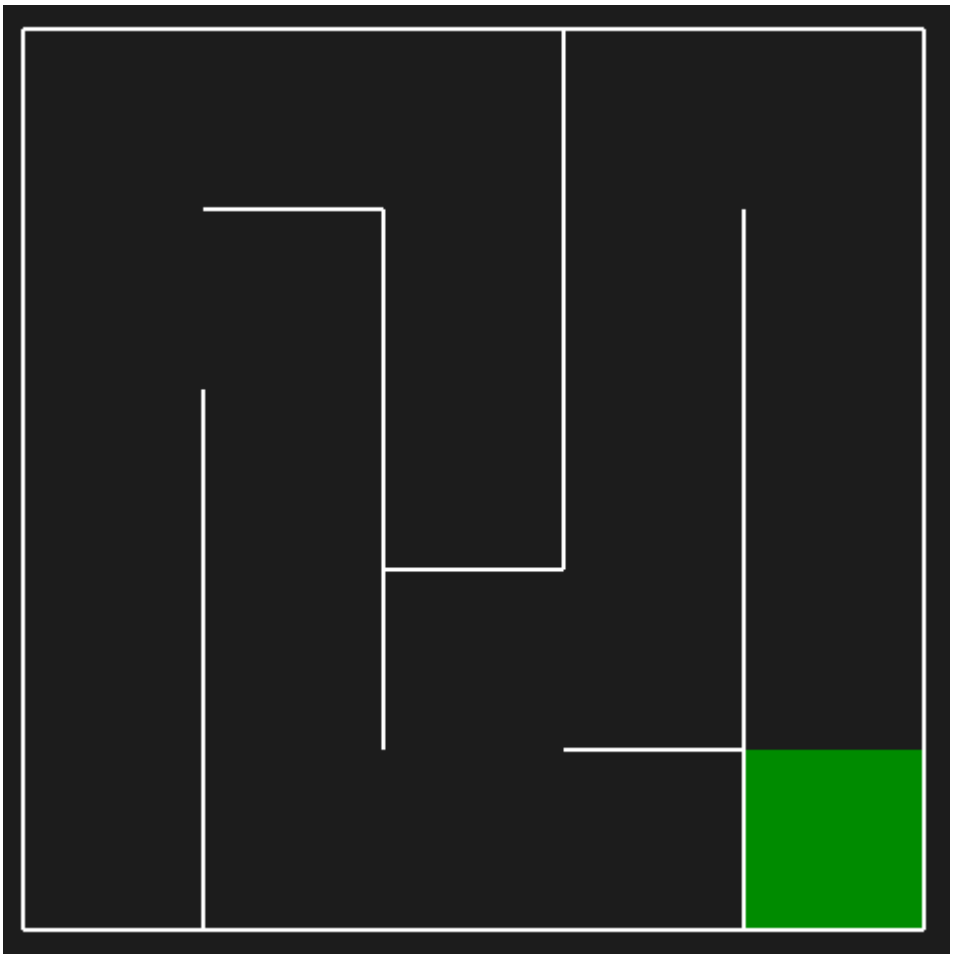


Optional Arguments of CreateMaze:

To customize the generated maze, we can use different Optional Input arguments. **Goal (x and y):** To change the goal cell from (1,1) to some other cell we can provide the optional arguments ***x*** and ***y*** as the goal. For example, to make the cell (2,4) as goal we will use the function as `CreateMaze(2,4)`

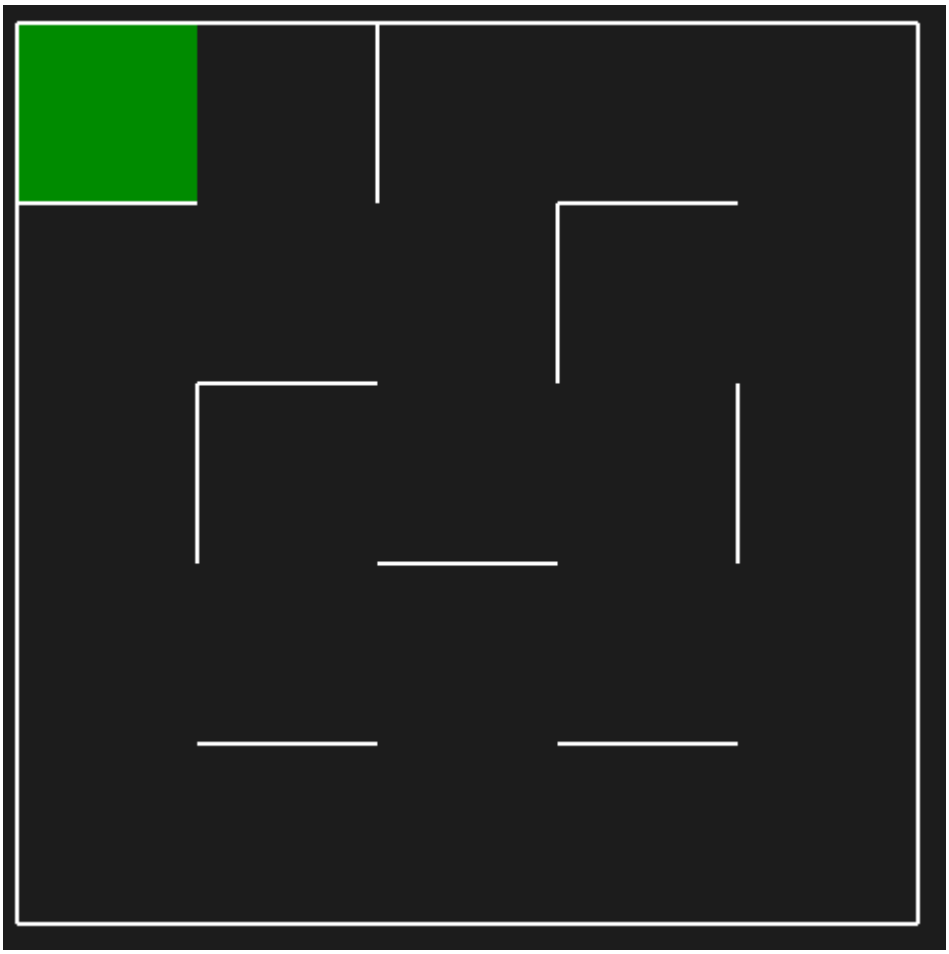
Pattern: We can generate a Horizontal (or Vertical) pattern maze. A horizontal pattern maze means the maze will have longer horizontal lines compared to the vertical line and similarly for the vertical pattern, the vertical maze lines will be longer. We can set the optional argument pattern to **'h'** or **'H'** for the horizontal pattern and **'v'** or **'V'** for the vertical pattern. For example, to generate a vertical pattern 5x5 maze with goal as cell (5,5), we will use the function as: `m.CreateMaze(5,5,pattern='v')`

Sample generated maze is:



Multiple Paths Maze:

By default, the generated maze is **Perfect Maze** meaning just the one path from any cell to the goal cell. However, we can generate a maze with multiple paths by setting the optional argument ***loopPercent*** to some positive number. ***loopPercent*** set to highest value **100** means the maze generation algorithm will maximize the number of multiple paths for example as: `m.CreateMaze(loopPercent=100)` The generated maze is shown here:



Save the generated Maze:

There is also the possibility to save any generated maze for future use. For that, we need to set the optional argument ***saveMaze*** to ***True***. The randomly generated maze will be saved in the working folder as a CSV file. The CSV file will contain the information of all cells inside the Maze and the information of the opened and closed walls in East West North and South directions. 1 means the path is opened in that direction and 0 means it is closed.

We can later use the CSV file to generate the same old maze by using the ***loadMaze*** option and providing the CSV file.

With this feature, we can also manually customize the Maze by changing the CSV file. To add or remove one wall from the Maze, we should change two values of the CSV file. While loading a Maze from the CSV file, the size of the Maze while creating the maze does not matter since the information about the size is also loaded from the CSV file.

Theme:

The default theme of the Maze is the **Dark** theme and we can change that to the **Light** theme by using the argument ***theme*** and setting that to the Light theme. We have a ***COLOR*** class as well inside the **pyamze** module to manage different colors. To set the theme to light, we can set that as a **COLOR** class object as:

```
m.CreateMaze(theme=COLOR.light)
```

or we can also provide the value as string as:

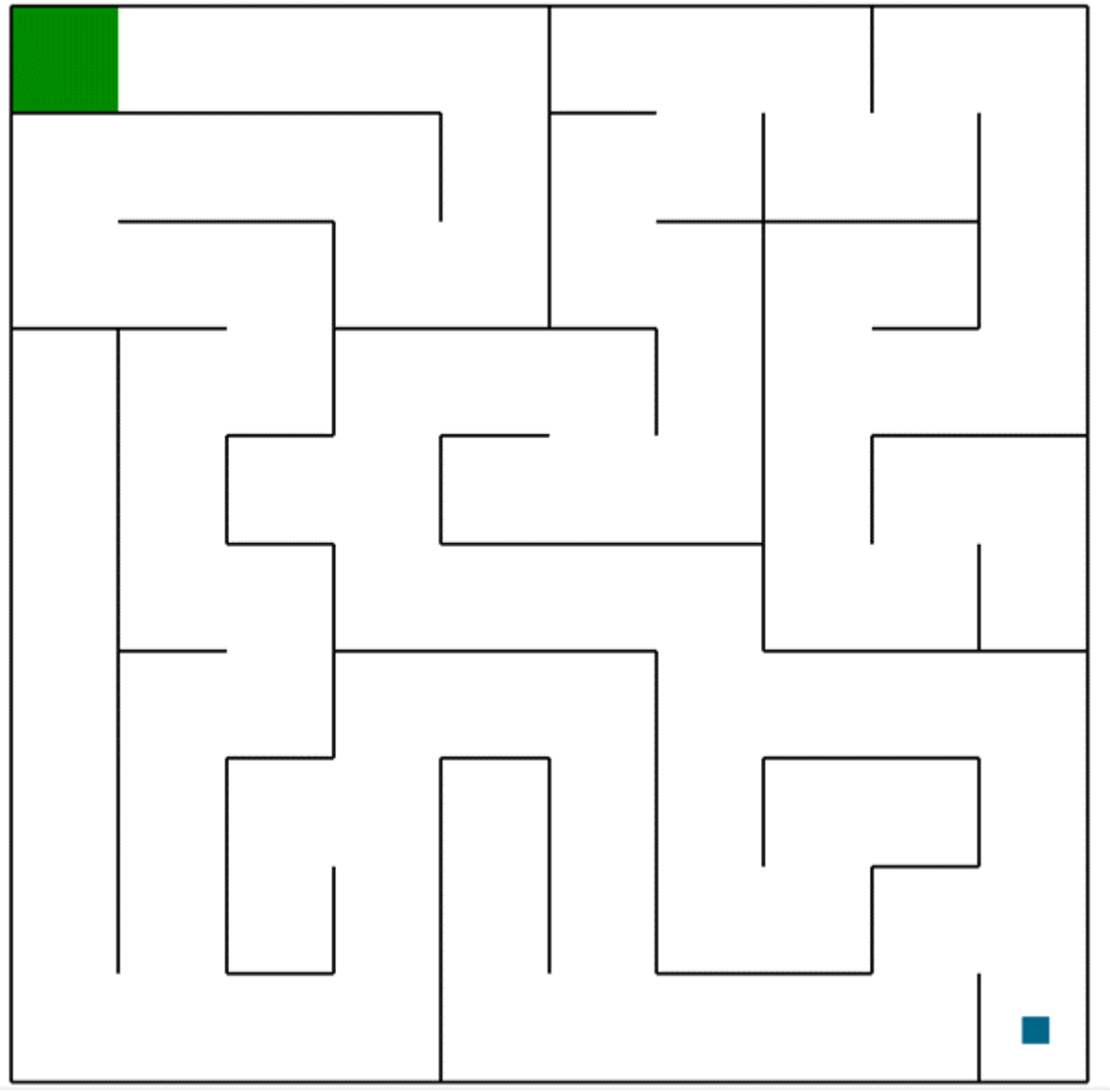
```
m.CreateMaze(theme="light")
```

Placing Agents inside the Maze:

We can place **agent** (one or more) inside the Maze. An agent can be thought of as a physical agent like a robot or it can simply be used to highlight or point a cell in the maze. For that, we have the ***agent*** class in the module **pyamaze**. After importing the agent class, we can create the agent object and we should provide the parent Maze as the first input argument. By default, the agent will be placed on the start cell (the last cell) of the Maze.

Here is complete code to create an agent on the default sized maze with light theme:

```
from pyamaze import maze,COLOR,agent
m=maze(10,10)
m.CreateMaze(theme=COLOR.light)
a=agent(m)
m.run()
```



Now let's see different Optional Arguments of the agent class.

Location of the agent: The default location of the agent is the start cell of the Maze which is the bottom-right corner of the Maze. You can change the location by setting the values ***x*** and ***y*** for the cell.

The agent object has the two attributes ***x*** and ***y*** that you can access and set those later after an agent has been created. Moreover, the agent has an attribute ***position*** set to the tuple **(x,y)** that provides the complete ***x*** and ***y*** information as one parameter. You can also set it to some other value to change the position of the agent.

Goal of the agent: The default goal for the agent is the goal of the Maze meaning the target for the agent is to reach the goal of the Maze. However, if you want to change the goal of the agent, you can do it by setting the argument ***goal*** while creating the agent. A two-valued tuple should be assigned as the goal of the agent.

Size of the Agent: By default, the size of the agent is smaller than the cell dimensions. You can set the argument ***filled*** to ***True*** and the agent will fill the whole cell.

Shape of the agent: By default, the agent is of square shape and there is a second option of shape ***'arrow'*** that you can set to the ***shape*** argument and the agent will be arrow-head shaped. This will differentiate the front and other sides of the agent. The argument ***filled*** has no effect if the shape is set to an arrow.

See the footprints: When you will implement some search algorithm and the agent will move in the maze, it can be the requirement to visualize the complete path trace. For that, we can change the optional argument ***footprints*** equal to ***True*** and whenever the agent changes its position, an impression of footprints will be imposed on the previous location. Footprints is just the shape of the agent but with a different color shade. See the output yourself for this code:

```
from pyamaze import maze,COLOR,agent
m=maze(5,5)
m.CreateMaze()
a=agent(m,shape='arrow',footprints=True)
a.position=(5,4)
a.position=(5,3)
a.position=(5,2)
m.run()
```

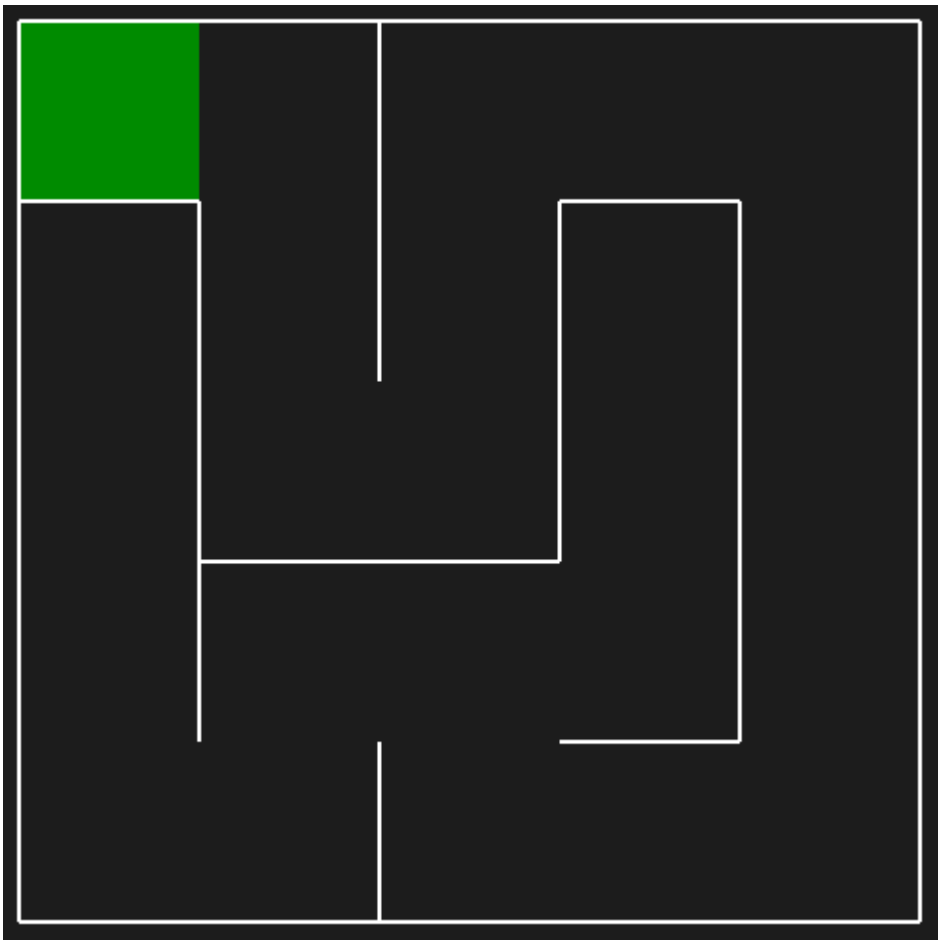
Other attributes of the Maze class: A few attributes of the Maze class that you should know to implement some search algorithm are:

Rows: ***m.rows*** gives the number of rows of the maze **m**. **Columns:** ***m.cols*** gives the number of columns of the maze **m**. **Grid:** ***m.grid*** is a list with all cells from (1,1) to last. **Map of the Maze:** A Maze is generated randomly. It is important to know the information of different opened and closed walls of the Maze. That information is available in the attribute ***maze_map***. It is a dictionary with the keys as the cells of the Maze and value as another dictionary with the information of the four walls of that cell in four directions; East, West, North and South.

You can see the value of this attribute and confirm the values with the maze generated as:

```
print(m.maze_map)
```

The example maze generated and the value of the maze_map is shown here:



```
maze_map = {(1, 1): {'E': 1, 'W': 0, 'N': 0, 'S': 0}, (2, 1): {'E': 0, 'W': 0,
'N': 0, 'S': 1}, (3, 1): {'E': 0, 'W': 0, 'N': 1, 'S': 1}, (4, 1): {'E': 0, 'W':
0, 'N': 1, 'S': 1}, (5, 1): {'E': 1, 'W': 0, 'N': 1, 'S': 0}, (1, 2): {'E': 0,
'W': 1, 'N': 0, 'S': 1}, (2, 2): {'E': 0, 'W': 0, 'N': 1, 'S': 1}, (3, 2): {'E':
1, 'W': 0, 'N': 1, 'S': 0}, (4, 2): {'E': 1, 'W': 0, 'N': 0, 'S': 1}, (5, 2):
{'E': 0, 'W': 1, 'N': 1, 'S': 0}, (1, 3): {'E': 1, 'W': 0, 'N': 0, 'S': 1}, (2,
3): {'E': 0, 'W': 0, 'N': 1, 'S': 1}, (3, 3): {'E': 0, 'W': 1, 'N': 1, 'S': 0},
(4, 3): {'E': 1, 'W': 1, 'N': 0, 'S': 1}, (5, 3): {'E': 1, 'W': 0, 'N': 1, 'S':
0}, (1, 4): {'E': 1, 'W': 1, 'N': 0, 'S': 0}, (2, 4): {'E': 0, 'W': 0, 'N': 0,
'S': 1}, (3, 4): {'E': 0, 'W': 0, 'N': 1, 'S': 1}, (4, 4): {'E': 0, 'W': 1, 'N':
1, 'S': 0}, (5, 4): {'E': 1, 'W': 1, 'N': 0, 'S': 0}, (1, 5): {'E': 0, 'W': 1,
'N': 0, 'S': 1}, (2, 5): {'E': 0, 'W': 0, 'N': 1, 'S': 1}, (3, 5): {'E': 0, 'W':
0, 'N': 1, 'S': 1}, (4, 5): {'E': 0, 'W': 0, 'N': 1, 'S': 1}, (5, 5): {'E': 0,
'W': 1, 'N': 1, 'S': 0}}
```

Path from start to goal: The maze generation algorithm used in the **pyamaze** module ***(Recursive Backtracker)*** not just generates a random maze, but also has the information of the path from start to goal. This information is available in the attribute path of the Maze as a dictionary. The key of the path is a cell and value is also a cell representing the movement from key cell to value cell in order to reach the goal.

Move the agent on a path: After creating a Maze and agent (one or more) inside the Maze, we can make the agent move on a specific path. The best will be moving the agent on the ***path*** attribute of the maze. For that, we have a method in the maze class named as ***tracePath*** that takes one dictionary as the input argument. The key of the dictionary is the agent and the value is the path we want that agent to follow. The ***tracePath*** method will simulate the agent moving on the path.

Run this code and see the simulation yourself.

```
from pyamaze import maze,agent
m=maze(20,20)
m.CreateMaze(loopPercent=50)
a=agent(m,filled=True,footprints=True)
m.tracePath({a:m.path})
m.run()
```

There are three ways we can specify a path for the agent to follow.

Path as a Dictionary: As shown above the path can be a dictionary with key-value pairs representing the movement from key-cell to value-cell. **Path as List:** There is also a possibility to provide the path as a List of cells. Then the agent will follow the path starting from the first cell inside the list to the last cell. **Path as String:** We can also provide the path as a string of movement directions (EWNS), e.g. 'EENWWSES' is a string of 8 steps for the agent to follow. Optional Arguments in tracePath method:

There are a couple of Optional arguments as well available with tracePath method.

Kill the agent: It is possible to kill the agent after it completes the path. By setting the argument ***kill*** to ***True*** and the agent will be killed after 300 milliseconds after completing the path. **Movement Speed:** We can control the movement speed of the agent using the argument ***delay*** having the default value of 300 milliseconds. It is a time delay between the movement steps of the agent. **Mark some cells:** For different demonstrations, it might be needed to mark a few cells. For that, there is an option of ***showMarked*** that can be set to ***True*** and any cell present inside the list of maze ***markCell*** will be marked if the agent passes through that cell. **Multiple Agents on different Paths:** There is also the possibility to move multiple agents on their own paths. For that, we can provide more agent-path information inside the input dictionary to the ***tracePath*** method. There will be the movement against all agent-path pairs provided in the dictionary.

Moreover, we can use the ***tracePath*** method multiple times. In that case, firstly all agents-paths provided the first time will complete their paths and then the other agent-path pairs provided second time in tracePath will start their movement and then so on.

Controlling Agents with the keyboard

Finally, there is also the possibility of controlling the agent with keyboard keys. For that, you can use the maze class method ***enableArrowKey*** and as the input argument, provide the agent to control with arrow keys. Likewise, an agent can be controlled using the keys WASD using the method ***enableWASD***.