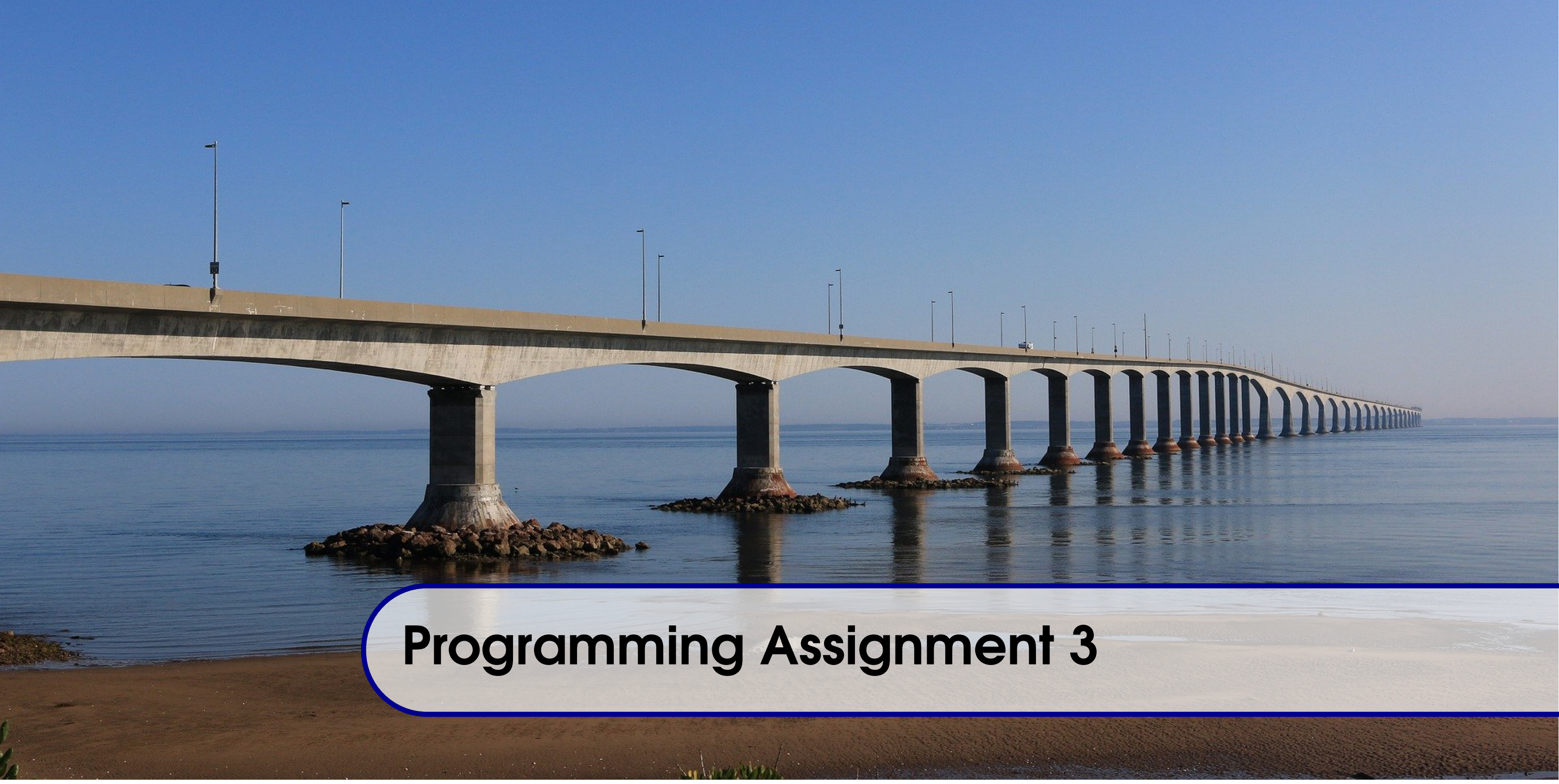


# Programming Assignment 3

**Assignment Objectives**

By completing this assignment you are demonstrating your ability to:

- Decompose problems to create new solutions
- Create and call functions
- Make use of objects and modules
- Make a use of string indexing and splicing

## Scenario

In this course you will have 4 assignments, all of which are going to be related to the same topic: Maze!

## Maze Introduction

A $n \times n$ maze is a collection of paths among interconnected cells in a grid, where $n$ is the number of rows and/or the number of columns. The classic maze path-finding problem tries to find a feasible path from a source cell to a destination cell. Any cell could be of one of the three possible types: $(i)$ accessible cell: a cell which allows a path to pass through it, $(ii)$ blocked cell: a cell that acts as a dead-end and blocks a path or, $(iii)$ null cell: a cell that is neither accessible nor blocked; a null cell is used to initialize and create the maze. In this assignment, we just need to worry about the accessible cells and the blocked cells. Any valid path from a source to a destination can only pass through the accessible cells. In this assignment, we have a better maze-generator that can produce a random maze of size $n \times n$, where $n >= 3$. As shown in figure 1, the first cell has coordinates $(0,0)$. Note that in Python and most of the other programming languages, indexing starts from 0 (and not 1).

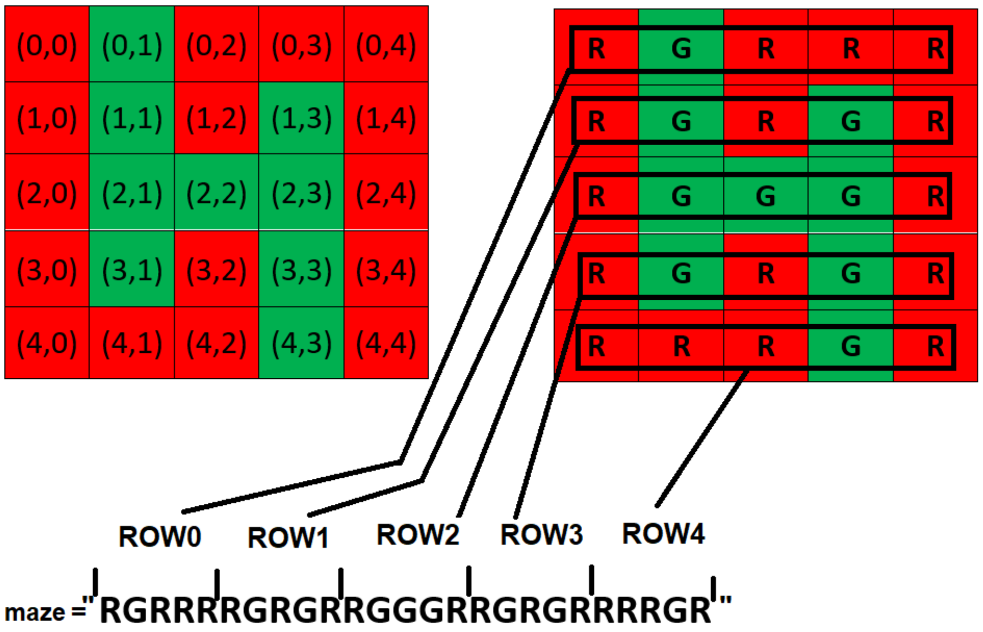


Figure 1: A random $5 \times 5$ maze

## The “mazeSmart” Module

Our *maze* generator just got smarter than before. There is no upper-limit on the maze size anymore. Also, the randomly selected source and the destination cells are not restricted anymore only to the first and the last rows respectively. To use the functions of this module, download the file *mazeSmart.py* and import it into your program. Though the functions have the same names as before and they are meant to perform the same tasks, their functionality have been improved. This is an example of abstraction where you do not have to worry about the hidden details.

**How to import the “*mazeSmart*” module and call its functions:**
# Assume that you are writing a program in *main.py* under your PyCharm project
# Download *mazeSmart.py* and put it inside your PyCharm project (on the same path where ...
# ... *main.py* is)
# Write the following as the first line of your program (in *main.py*)
    *import mazeSmart as mz*
# Call the function function_name of the *mazeSmart* module
    *mz.function_name(arguments)*

**What functions are there in the “mazeSmart” module:**
*print_maze(maze)*
*Description:* This function displays the maze. The accessible cells are labelled “G” i.e. green, the blocked cells are labelled “R” i.e. red, and the null cells are labelled “Y” i.e. yellow. The argument “maze” is a string over an alphabet set “G”, “R”, “Y” that encodes the cell labels of the maze. Row labels of the maze are concatenated to form a single string that represents the maze, see figure 1.

*create_maze(n)*
*Description:* This function creates and returns a "maze" string of size $n \times n$. The value of an argument $n$ should be $>= 3$ for this function to work. This function returns a string "maze".

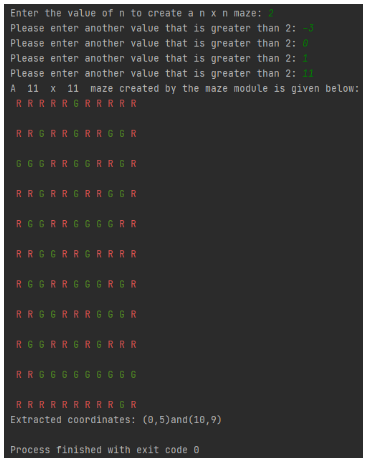
*get_source()*
*Description:* This function returns the coordinates of the source cell in the format "*source_x:a,source_y:b*". Here *a, b* are the *x, y* coordinates of the source cell.
*Example:*
*source_x:12,source_y:0* represents a source cell at coordinates $(12,0)$.

*get_destination()*
*Description:* This function returns the coordinates of the source cell in the format "*destin_x:a,destin_y:b*". Here *a, b* are the *x, y* coordinates of the destination cell.
*Example:*
*destin_x:4,destin_y:13* represents a destination cell at coordinates $(4,13)$.

## Problem 1

Write a program in Python that performs the following tasks in order:

1. Takes the size of maze as an input from the user and assign a variable name "*n*" to this value. Create and print a maze of size "*n*" using the functions of the "mazeSmart" module if "$n > 2$". If "$n <= 2$", then your program should print "Please enter another value that is greater than 2: " and keeps on asking for an input from the user till a valid value of "n" is read.
2. Using the functions of the "mazeSmart" module, get the coordinates of the source and the destination cells.
3. Create a function "process_strings" that extracts the coordinates of the source and the destination cells from the strings returned in the previous step using string indexing and splicing. The function should first create a new string of the coordinates in the format shown below, then print and return it:
"(sx,sy)and(dx,dy)"



```
Enter the value of n to create a n x n maze: 2
Please enter another value that is greater than 2: -3
Please enter another value that is greater than 2: 0
Please enter another value that is greater than 2: 1
Please enter another value that is greater than 2: 11
A  11  x  11  maze created by the maze module is given below:
 R R R R R G R R R R R

 R R G R R G R R G G R

 G G G R R G G R R G R

 R R G R R G R R G G R

 R G G R R G G G G R R

 R R G G R R G R R R R

 R G G R R G G G R G R

 R R G G R R R G G G R

 R G G R R G R G R R R

 R R G G G G G G G G G

 R R R R R R R R R G R
Extracted coordinates: (0,5)and(10,9)

Process finished with exit code 0
```

Figure 2: Sample program run of completed Problem 1.

## Problem 2

This problem is the continuation of problem 1. You have already created the maze, printed the maze, and extracted the coordinates of the source and the destination cells (as a string returned from the "process_strings" function) in problem 1. Using the maze produced by the functions of the "mazeSmart" module, write a program in Python that performs the following tasks in order:

1. Create a function "compute_path" that computes a conditional valid zigzag path between the source and the destination cells. This function should have exactly three parameters: the first for the maze string, the second for the maze size, and the third for the string returned from the "process_strings" function.
   *Note:* The source coordinates are always in the first row or the first column. Similarly, the destination coordinates are always in the last row or the last column. If the source cell exists in the first row, then the destination cell is in the last row. If the source cell exists in the first column, then the destination cell is in the last column. Remember that any valid path can only go through the accessible "green" cells. For this problem, a path has to be valid and should follow the following additional conditions:
   (a) Your function should compute the forward row-wise path if it exists i.e. at any step, you need to move from the current row to the next.
   (b) You can not stay in the same row. Also, you can not go from a row with a higher index to a row with a lower index.
   (c) You can stay in the same column or move forward to the next column. You can not go from a column with a higher index to a column with a lower index.
   (d) You can move only in a zigzag manner i.e. if you stayed in the same column during your previous move, then you should move to the next column and vice versa.
   *Example1:*
   Current cell: (2,2)
   Previous move: (1,2)->(2,2)
   Current move: (2,2)->(3,3)
   *Example2:*
   Current cell: (2,2)
   Previous move: (1,1)->(2,2)
   Current move: (2,2)->(3,2)
   To summarize, if the current cell is (i, j), then there are only two possible moves ahead (i+1,j) or (i+1,j+1). If the previous move was from (i-1,j-1) to (i,j), then the current move will be from (i,j) to (i+1,j). If the previous move was from (i-1,j) to (i,j), then the current move will be from (i,j) to (i+1,j+1).
2. Create another function "print_path" that prints the valid path in the required format as shown in figure 3 if it exits, or print an error message "Valid zigzag path not found!" if there is no valid conditional path from the source to the destination.
3. Call the "compute_path" function to compute a path from the source cell to the destination cell. The "compute_path" function should make a nested function call to the "print_path" function.

*You can only use the concepts that you have studied so far, the use of lists and dictionaries is not allowed.

*(Optional)Every time you create a maze, it randomly produces a new maze for you. This can make testing your program harder. To make the testing of path computation easier, we have added a new function "test_maze()" to the "mazeSmart" module. This function always produces the same maze that has a valid path from the given source to the destination cell. It is recommended that initially, you use this function to write your path computation code for problem#2. Once it works for

the testing function, you should remove the code that makes a call to test_maze() function and use the actual functions of the maze module as instructed in the assignment description. Also, please note that this function does not cover all test cases and just provides a single successful test case with a valid path.

**Usage:** maze, size, sr, de = mz.test_maze()

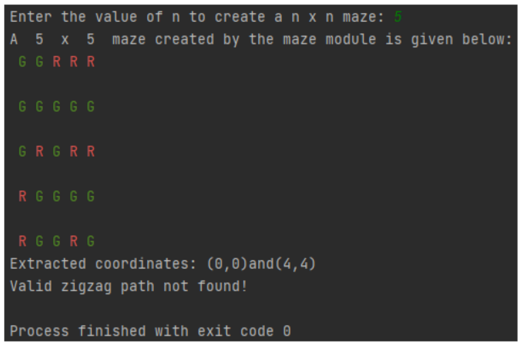
**Returns:**

maze: a maze string similar to the return of create_maze() function

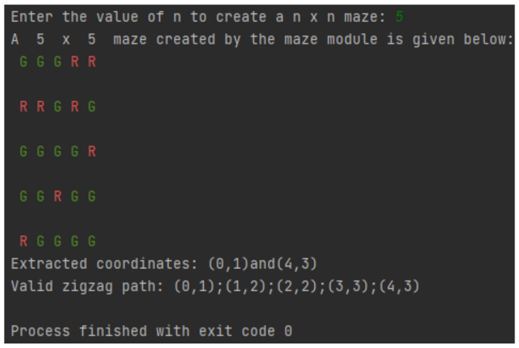
size: maze dimension similar to the value of n that you read from console

sr: string of source coordinates similar to the return of get_soruce() function

de: string of destination coordinates similar to the return of get_destination() function



```
Enter the value of n to create a n x n maze: 5
A  5  x  5  maze created by the maze module is given below:
 G G R R R

 G G G G G

 G R G R R

 R G G G G

 R G G R G
Extracted coordinates: (0,0)and(4,4)
Valid zigzag path not found!

Process finished with exit code 0
```



```
Enter the value of n to create a n x n maze: 5
A  5  x  5  maze created by the maze module is given below:
 G G G R R

 R R G R G

 G G G G R

 G G R G G

 R G G G G
Extracted coordinates: (0,1)and(4,3)
Valid zigzag path: (0,1);(1,2);(2,2);(3,3);(4,3)

Process finished with exit code 0
```

Figure 3: Sample program runs of completed Problem 2.

## Submitting your assignment

Your submission will consist of one zip file containing solutions to all problems (see below). This file will be submitted via a link provided on our Moodle page just below the assignment description. This file must be uploaded by 5pm (Charlottetown time) on the due date in order to be accepted. You can submit your solution any number of times prior to the cutoff time (each upload overwrites the previous one). Therefore, you can (and should) practice uploading your solution and verifying that it has arrived correctly.

**What's in your zip file?**
A zip file is a compressed file that can contain any number of folders and files. Name your zip file using this format.

**Example:**
Submission file: `asnX_studentnum.zip`
Where `X` is the assignment number between 1 and 4, and `studentnum` is your student ID number.

Your zip file should contain a PyCharm project. The project folder should contain all Python files for the successful execution of your programs. Each problem should have a file named `problem#.py` where the # is the problem number.