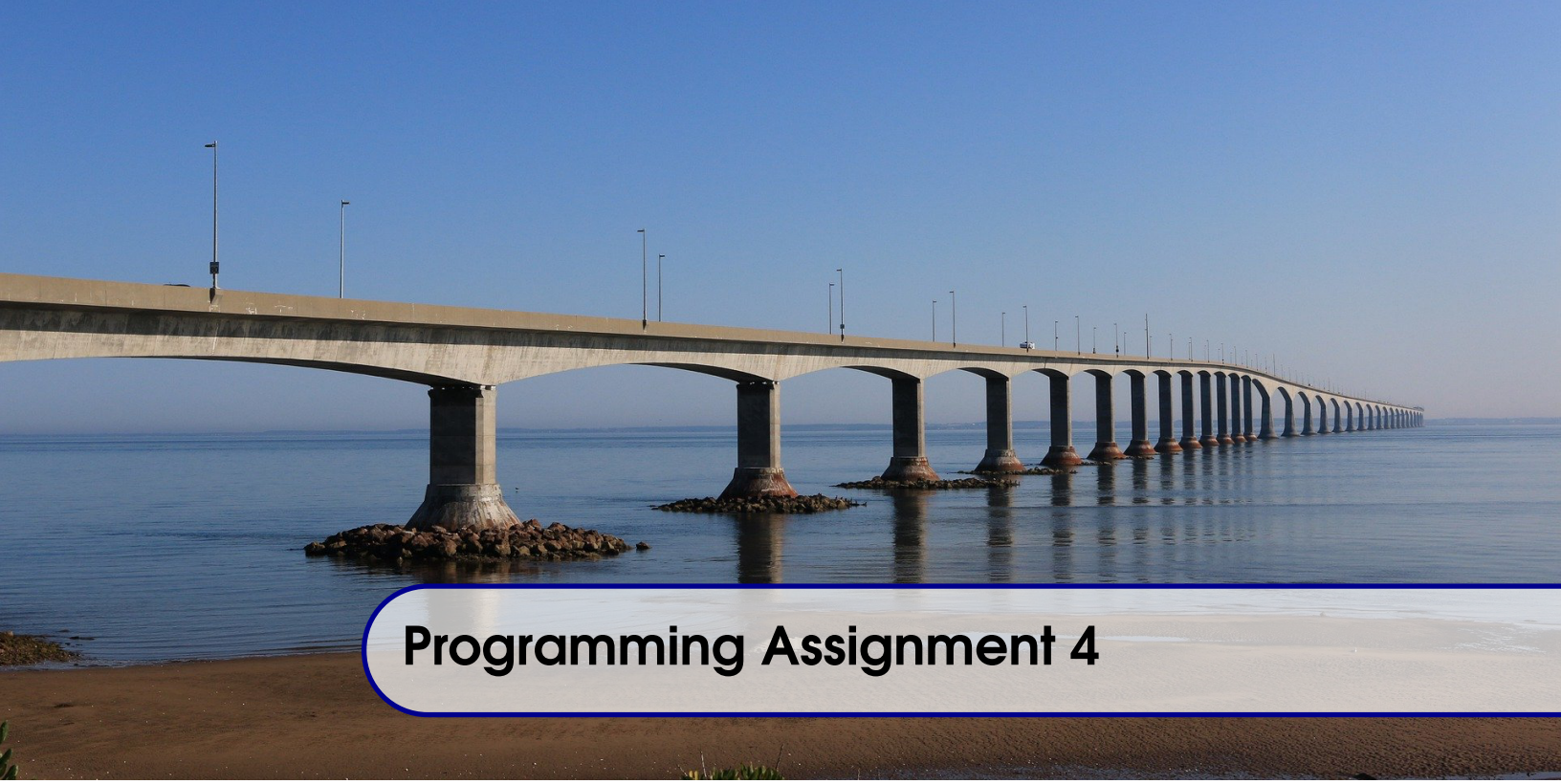

Programming Assignment 4

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 use of string indexing and splicing
- Make use of Lists, Dictionaries, and File I/O

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).

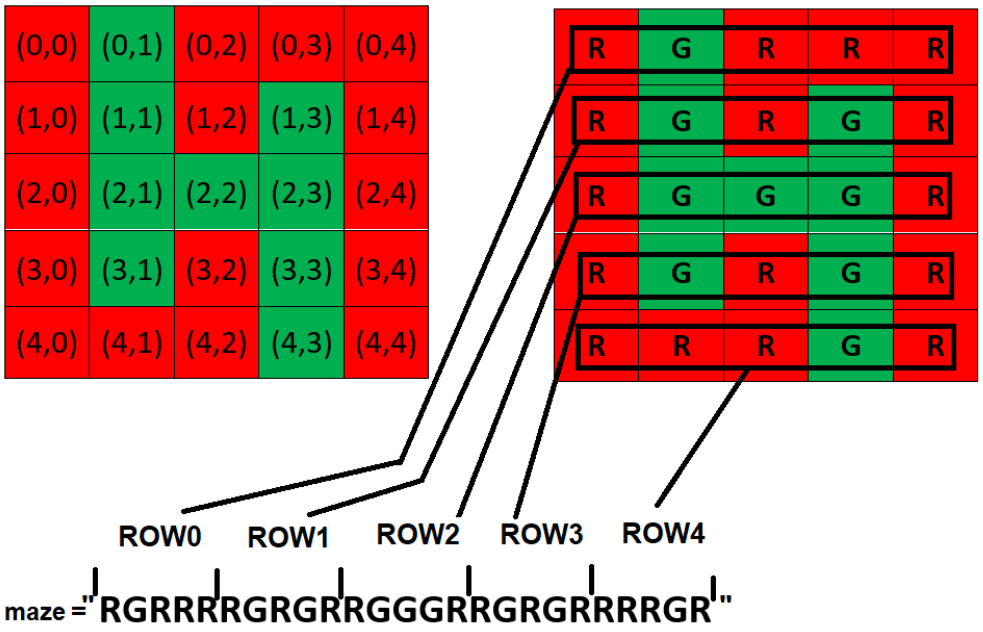

Figure 1: A random 5×5 maze

The "mazeAsn4" Module

This assignment will use an updated version of *maze* generator that is very similar to the *mazeSmart* module used in the previous assignment. To use the functions of this module, download the file *mazeAsn4.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 "*mazeAsn4*" module and call its functions:
Assume that you are writing a program in *main.py* under your PyCharm project
Download *mazeAsn4.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 mazeAsn4 as mz
Call the function *function_name* of the *mazeAsn4* module
mz.function_name(arguments)

What functions are there in the "mazeAsn4" 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(problem_num)
Description: This function creates and returns a "maze" string of size $n \times n$.
When problem_num is 0: The function asks the user to input a valid value ($>= 3$) of maze size n. The function validates the value of n and returns a string "maze". The value "0" should be used for problem 1.
When problem_num is 1: The function randomly generates and returns a maze string of size $n \times n$, where $3 <= n <= 9$. The value "1" should be used for problem 2.
When problem_num is any other number: The function will raise an error and stop the program execution.

get_source()
Description: This function returns the coordinates of the source cell as a list in the format $[[\text{"}source\_x\text{"}, a], [\text{"}source\_y\text{"}, b]]$. Here *a, b* are the *x, y* coordinates of the source cell.
Example:
$[[\text{"}source\_x\text{"}, 12], [\text{"}source\_y\text{"}, 0]]$ represents a source cell at coordinates $(12, 0)$.

get_destination()
Description: This function returns the coordinates of the destination cell as a list in the format $[[\text{"}destin\_x\text{"} : a], [\text{"}destin\_y\text{"} : b]]$. Here *a, b* are the *x, y* coordinates of the destination cell.
Example:
$[[\text{"}destin\_x\text{"} : 4], [\text{"}destin\_y\text{"} : 13]]$ represents a destination cell at coordinates $(4, 13)$.

Problem 1

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

1. Using the functions of the "mazeAsn4" module, first create and print a new maze. Then, print the coordinates of the source and the destination cells in the format shown below:
   Source coordinates: X: 0 Y: 1
   Destination coordinates: X: 4 Y: 4
2. Create a function "compute_path" that computes and returns a list containing a valid conditional path from the source cell to the destination cell if there exists such a path. If a valid path does not exist, then the function should return an empty list. This function should have exactly three parameters: the first for the maze string returned from the "create_maze" function, the second for the list returned from the "get_source" function, and the third for the list returned from the "get_destination" function.
   Note: 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) There are no restrictions on the column-wise moves i.e. you can stay in the same column, move backward to the previous column, or move forward to the next column.
   (d) To summarize, if the current cell is (i, j), then there are only three possible moves ahead: (i+1, j-1), (i+1, j), and (i+1, j+1).

Hint: You can use the following algorithm to compute a valid path:
Algorithm compute_path(maze, src, dest)
#maze: maze string returned from the create_maze function
#src: list returned from the get_source function
#dest: list returned from the get_destination function
1 Create a list T and add source tuple as a list [(src_x, src_y)] to T
2 While T is not empty:
    2.1 Remove the first list element E from T
    2.2 Obtain last tuple (i,j) of sub-list E
    2.3 Explore all 3 possible moves from (i,j) i.e. (i+1,j-1), (i+1,j), and (i+1,j+1)
    2.4 For all valid moves, concatenate the valid moves to the sub-list E and add the newly found partial paths to T
    2.5 If any of the three moves leads to the destination cell, then we have found a valid path; exit the loop with a successful path search
3 If a valid path is found, then return the path as a list; otherwise, return an empty list

Figure 2 shows an example of the list manipulation using the algorithm compute_path.

```
R R G R R
R G R G R
G R R G R
R R G R R
R G R R R
Maze: RRGRRRGRGRGRRGRRRGRRRGRRR
Source: (0,2); Destination: (4,1)

Different iterations changes the list T as shown below:
List T: [(0,2)]
List T: [[(0,2),(1,1)], [(0,2),(1,3)]]
List T: [[(0,2),(1,3)], [(0,2),(1,1), (2,0)]]
List T: [[(0,2),(1,1), (2,0)], [(0,2),(1,3), (2,3)]]
List T: [[(0,2),(1,3), (2,3)]]
List T: [[(0,2),(1,3), (2,3), (3,2)]]
List T: [[(0,2),(1,3), (2,3), (3,2),(4,1)]]
Path found: [(0,2),(1,3), (2,3), (3,2),(4,1)]
```

Figure 2: Example showing the list manipulation using the algorithm compute_path.

3. Print the computed path in the format as shown in figure 3 if found, otherwise print "A valid path does not exist!"

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


 G G G


 G G G
Source coordinates: X: 0 Y: 0
Destination coordinates: X: 2 Y: 2
Valid path found: [(0, 0), (1, 1), (2, 2)]
******PROBLEM 2******
Successfully created maze.txt with dictionary keys and values!

Process finished with exit code 0
```

Figure 3: Sample program run of completed Problems 1 and 2.

Problem 2

This problem is the continuation of problem 1. You have already created the maze, printed the maze, obtained the coordinates of the source and the destination cells, and computed a valid conditional path if there exists such a path. Using the maze produced by the functions of the "mazeAsn4" module, write a program in Python that performs the following tasks in order:

1. Using the functions of the "mazeAsn4" module, create 10 mazes with valid paths. If the original maze (the one created in problem 1) has a valid path, then consider this as the first maze with a conditional valid path. If the original maze does not have a valid path, then repeatedly perform the following tasks until a maze with a valid path is found: create a new maze, get its source and destination cells, and compute a conditional valid path if there exists such a path (*hint*: make use of the code that you wrote for problem 1). Repeat the process to produce 10 mazes with valid paths.
2. Create a dictionary with maze numbers as keys and their respective valid paths as values. A key for any maze number X should be in the format "MazeX", where $1 <= X <= 10$.
3. Create a new text file "maze.txt" and write the dictionary created in step 2 into this text file. The contents of text file should be in the format as shown in figure 4. At the end of your program, print "Successfully created maze.txt with dictionary keys and values!".

```
Maze1:['GRGGGGGGG', [(0, 0), (1, 1), (2, 2)]]
Maze2:['GGGRRGGGGRRRGGGRGGRGRGGGG', [(0, 0), (1, 1), (2, 2), (3, 2), (4, 3)]]
Maze3:['GGGGGRGGGGRGGGGRRGGGRGGRRRGGGGGRRGGRGGGRGGGGGGGRRGGRGRGGGGGGRGGG', [(0, 1), (1, 1), (2, 2), (3, 3), (4, 4), (5, 5), (6, 6), (7, 6)]]
Maze4:['GGGGRGRGGGGGGGGRGGGGGGGRRGGGRGGRRRGGGGRGRGRGGGGGGRGGRRGGGGRGGRGGG', [(0, 2), (1, 1), (2, 1), (3, 2), (4, 3), (5, 4), (6, 5), (7, 5)]]
Maze5:['GGGRGGGGGGGGRRGG', [(0, 1), (1, 1), (2, 2), (3, 3)]]
Maze6:['GGGRGGGGGGGGRRRGGGGRGRGG', [(0, 0), (1, 0), (2, 1), (3, 2), (4, 3)]]
Maze7:['RGGRRGRRGGGGGGGGRRGGRGRRRGRGGGRGGRRGRGRRGGGGGGRRGRRGRGGRGRGGRRG', [(0, 2), (1, 2), (2, 3), (3, 4), (4, 4), (5, 5), (6, 6), (7, 7)]]
Maze8:['GRGGGGRRGRRRGGGGGGGRGGGRGGRRRRGRGRRGGRGGRGGRGGGGGGRGGGGRRGRRRRGGGGGGGRRGGRGGRGGRRGRRGGGGGGRRGGRGGGGG', [(0, 4), (1, 3), (2, 4), (3, 5), (4, 6), (5, 7), (6, 6), (7, 7), (8, 7), (9, 8)]]
Maze9:['GGRGRRGRRRGGRRGGGRGGGGGRGGRRGRGGRRGGGRRRGGGGRRGGGGRRRRRGRGGGRGGGGGGGGGGGRRGRGGGGR', [(0, 1), (1, 1), (2, 2), (3, 3), (4, 4), (5, 3), (6, 3), (7, 4), (8, 5)]]
Maze10:['RGRGGGRGG', [(0, 0), (1, 1), (2, 2)]]
```

Figure 4: Sample of contents of maze.txt.

*You can only use the concepts that you have studied in CS1910 so far.

*(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. 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, src, dest = mz.test_maze()
Returns:
maze: a maze string similar to the return of create_maze() function
src: list of source coordinates similar to the return of get_soruce() function
dest: list of destination coordinates similar to the return of get_destination() function

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.