Departamento de Informática
Universidad Carlos III de Madrid

Bachelor's Degree in Data Science and Engineering

Programming

June 2019

General indications

- You have **3 hours**.
- You are not permitted to leave the room during the exam unless you have handed it in.
- The exams cannot be written in pencil.
- All answer sheets should be numbered and with your name on them.
- Built-in functions not seen during the course are forbidden; their use will be penalized.
- Notes, slides, books, etc. are allowed; electronic devices are not allowed.

Exercise 1 (5p)

Minesweeper is a game where the aim is to correctly identify the location of all mines in a grid. The player is given a grid of hidden cells in the beginning of the game. Each cell contains either a mine (indicated by *), or an empty cell. Empty cells have a number indicating the count of mines in the adjacent cells. (Two cells are adjacent to each other if they are connected horizontally, vertically, or diagonally.) Empty cells can have counts from 0 (no adjacent mines) up to 8 (all adjacent cells are mines). Cells with a 0 indicate that there are not adjacent mines to that particular cell; cells with a 1 indicate that there is one mine adjacent to that particular cell; cells with a 2 indicate that there are two mines adjacent to that particular cell; and so on. The game starts creating a grid with the size and number of mines specified by the player. The grid is initially hidden. In each turn, the player introduces the row and the column of the cell to reveal. Then the grid with all revealed cells is printed. If the cell contains a mine, the game ends and the player loses the game. Otherwise, the number of adjacent mines is revealed. The game continues until all non-mine cells are revealed, case in which the player wins the game. Finally, the game reveals the grid.

The aim of this exercise is to implement the Minesweeper game, where the grid consists of a matrix NxM. The matrix is represented as a list of lists that stores objects of type `Cell`, which is defined as follows:

```
class Cell:
    def __init__(self):
        self.mine = False #indicates if the cell contains a mine or not
        self.hidden = True #indicates if the cell is hidden
        self.character = None #indicates the character stored in the cell
```

You are asked to:

1. (1pt) Create a function `createMatrix(rows,columns,mines)` that given the number of rows, the number of columns, and the number of mines, returns a matrix of the given size and with the given number of mines randomly distributed in different cells of the grid.
2. (2pt) Create a function `setNumbers(matrix,rows,columns)` that updates the matrix by including the count of adjacent mines for each cell in the matrix.
3. (0.25pt) Create a function `printMatrix(matrix)` that prints the matrix. If a cell is hidden, it will print `-`. Otherwise, it will print the character of the cell.
4. (0.25pt) Create a function `revealMatrix(matrix)` that reveals all cells in the matrix.
5. (1.5pt) In a main program, ask the player the size of the grid and the number of mines and simulate the game. The game must warn the player when any of the introduced information is out of bounds and ask for it again. Implement auxiliary methods if needed.

An example of the expected execution when loosing the game:

```
Number of rows 2
Number of columns 3
Number of mines 1

Let's start the game!

Row? 0
Column? 3
Row or column out of bounds!

Row? 1
Column? 0
 -  -  -
 *  -  -

You lost!!
1 1 0
* 1 0
```

An example of the expected execution when winning the game:

```
Number of rows 2
Number of columns 3
Number of mines 2

Let's start the game!

Row? 0
Column? 2
 -  -  1
 -  -  -

Row? 1
Column? 0
 -  -  1
 2  -  -

Row? 1
Column? 1
 -  -  1
 2  2  -

Row? 1
Column? 2
 -  -  1
 2  2  1

You won!!
* * 1
2 2 1
```

Solution

```
from random import randrange

class Cell:

    def __init__(self):
        self.mine = False
        self.hidden = True
        self.character = None

def createMatrix(rows,columns,mines):
    matrix = []
    for _ in range(rows):
        r = []
        for _ in range(columns):
            r.append(Cell())
        matrix.append(r)
    while mines > 0:
        r = randrange(rows)
        c = randrange(columns)
        if not matrix[r][c].mine:
            matrix[r][c].mine = True
            matrix[r][c].character = '*'
            mines -= 1
    return setNumbers(matrix,rows,columns)


def setNumbers(matrix,rows,columns):

    for r in range(0,rows):
        for c in range(0,columns):
            if not matrix[r][c].mine:
                n = 0
                if r > 0 and matrix[r-1][c].mine:
                    n += 1
                if r > 0 and c < (columns - 1) and matrix[r-1][c+1].mine:
                    n += 1
                if c < (columns - 1) and matrix[r][c+1].mine:
                    n += 1
                if r < (rows - 1) and c < (columns - 1) and matrix[r+1][c+1].mine:
                    n += 1
                if r < (rows - 1) and matrix[r+1][c].mine:
                    n += 1
                if r < (rows - 1) and c > 0 and matrix[r+1][c-1].mine:
                    n += 1
                if c > 0 and matrix[r][c-1].mine:
                    n += 1
                if r > 0 and c > 0 and matrix[r-1][c-1].mine:
                    n += 1
                matrix[r][c].character = str(n)
    return matrix

def printMatrix(matrix):
    for row in matrix:
        for c in row:
            if c.hidden:
                print(' - ',end='')
```

```
            else:
                print(' %s'%c.character,end=' ')
        print()

def checkBound(c, bound):
    if c >= 0 and c < bound:
        return True
    return False

def revealMatrix(matrix):
    for row in matrix:
        for c in row:
            print(c.character,end=' ')
        print()

## main program

rows = int(input('Number of rows '))
columns = int(input('Number of columns '))
mines = int(input('Number of mines '))
print("Let's start the game!")
counter, seen = rows * columns - mines, 0
end = False
matrix = createMatrix(rows,columns,mines)
while not end:
    repeated = False
    while not repeated:
        r = int(input('Row? '))
        c = int(input('Column? '))
        if not checkBound(r,rows) or not checkBound(c,columns):
            print('Row or column out of bounds!')
        elif matrix[r][c].hidden:
            matrix[r][c].hidden = False
            repeated = True
            seen += 1
    printMatrix(matrix)
    if matrix[r][c].character == '*':
        print('\nYou lost!!')
        end = True
    elif seen == counter:
        print('\nYou won!!')
        end = True
revealMatrix(matrix)
```

Exercise 2 (5p)

The Way of St. James is a network of pilgrim ways coming together to the Cathedral of Santiago de Compostela. There is not only one single way of the Way of St. James, there are different ways, starting at various places. A traveler's company offers tours to the most popular ways. Therefore, it stores information about each of them. A way is defined by a name, a starting point, an end point, a level of difficulty, a total length, and a tuple with the length of the sections that form the way. The length of each section ranges from 10 to 30 kilometers. The total length of the way is calculated by adding the length of each section. The level of difficulty could be easy, medium, or hard.

1. (1pt) Define the needed classes and implement their constructors.
2. (1pt) Implement a method that returns a list with the three consecutive sections of a way with the highest accumulated length. For instance: in a way with sections (50,65,50,75,80,70,30), the method returns [75,80,70]. Indicate the class where this method belongs to.
3. (1.25pt) Implement a method that given a dictionary of ways and two points `A` and `B` returns `True` if it is possible to create a combined way by joining two ways, where the first one starts at point `A` and ends at any point `Y`; and the second one starts at point `Y` and ends at point `B`. Otherwise, it returns `False`. Indicate the class where this method belongs to.
4. (0.75pt) Implement a method that given a dictionary of ways and a point returns `True` if the point is the starting or end point of any of the ways. Otherwise, it returns `False`. Indicate the class where this method belongs to.
5. (1pt) In a main program:
   - Create a list of 30 points with consecutive names: `Point1`, `Point2`, ..., `Point30`.
   - Create a dictionary of 10 ways, also with consecutive names `Way1`, `Way2`, ..., `Way10`. The starting and end points must be randomly chosen from the previous point list. The number of sections of each way will be a random number between 10 and 20. The length of each section is also randomly chosen as well as the level of difficulty.
   - Ask the user to enter two points `PointA` and `PointB` and print on the screen if there is any combined way that joining two ways goes from `PointA` to `PointB`. If any of these points does not belong to any way, the program must warn the user and ask for a new point.

Solution

```
from random import randrange

class Way:

    def __init__(self,name,start,end,difficulty,sectionsNumber):
        self.name = name
        self.start = start
        self.end = end
        self.difficulty = difficulty
        self.length = 0
        self.sections = ()
        for i in range(sectionsNumber):
            self.sections += (randrange(10,30),)
            self.length += self.sections[i]

    def LargestSections(self):
        largestSections = [0,0,0]
        lengthSections = 0
        for i in range(len(self.sections)-3):
            print('range ',i)
```

```
            l = self.sections[i] + self.sections[i+1] + self.sections[i+2]
            if lengthSections < l:
                lengthSections = l
                largestSections[0] = self.sections[i]
                largestSections[1] = self.sections[i+1]
                largestSections[2] = self.sections[i+2]
        return largestSections

def combinedWay(ways,start,end):
    for initialWay in ways:
        if ways[initialWay].start == start:
            for finalWay in ways:
                if ways[finalWay].start == ways[initialWay].end and
                   ways[finalWay].end == end:
                    return True
    return False

def isPoint(ways, point):
    for way in ways :
        if ways[way].start == point or ways[way].end == point :
            return True
    return False

## main

difficulty = ['eassy','medium','hard']

points = []
for i in range(10):
    points.append('Point' + str(i))

ways = {}
for i in range(5):
    name = 'Way'+str(i)
    start = end = randrange(0,len(points))
    while start == end:
        end = randrange(0,len(points))
    ways[name] = Way(name,points[start],points[end], difficulty[randrange(0,3)],
                     randrange(10,21))


arePoints = False
while not arePoints:
    wayStart = input('Starting point: ')
    wayEnd = input('End point: ')
    if isPoint(ways,wayStart) and isPoint(ways,wayEnd):
        arePoints = True

if combinedWay(ways,wayStart,wayEnd):
    print('There is a new way between %s and %s'%(wayStart,wayEnd))
else:
    print('There is not a new way between %s and %s'%(wayStart,wayEnd))
```