COMP4004 Programming and Problem Solving
End of Semester Coursework

Value: 25 marks

Mode: Individual work

Deadline for final submission: Friday week 12 (16 Dec) at 13:00

Submission is by Turnitin

Revision History: Revised 19 Nov 2022 to correct some typos in Exercise 4 (`addBlock`). Revised 24 Nov to correct a typo in Exercise 15 (`jump`)

Introduction

Important Notes:

- We have set out the end of semester coursework as a single document here. To assist you to complete it we will also create some Replit exercises with automated tests. These Replit exercises will become available in week 9.
- The mark scheme for the coursework is set out at the end of this document.

In this coursework you will create software to implement a simple game. As is often the case in 'real world' programming, some decisions have already been made regarding the way the game will be implemented. And as in the real world, you must abide by these decisions rather than working to an 'improved' specification of your own[1]. In this instance we have already decided how the state of the game will be represented, and we have already specified a set of functions that operate on that state.

The Game

In this section we explain how the game is played. There is a separate document on the Moodle site that gives a short example of game play. You might find that it is easiest to have a quick look at that example first, and then continue reading this section.

The game is played on an infinite board. Each square of this board either contains an *entity* or is empty. An entity can be either the *player* (there is only one player) or it can be a *non-player entity* (NPE). The non-player entities are *blocks, bread,* and *herbs*. An entity can own integer quantities of various *items*. The items available are *height*, *strength*, and *agility*. At any point in time the player may be facing up, down, left, or right.

[1] If a developer does that in the 'real world' they are likely to be reprimanded by their employer. I know because I have been that developer!

The version of the game you will construct will have a command line interface (but you should construct it in such a way that you could easily change to, say, a GUI interface). The user interacts with the game by typing *requests*, which are of one or two-word text strings.

The requests available are as follows.

Request	Meaning
step	Move the player into the facing square (i.e., move her by one square, in the direction in which she is facing). This request succeeds if, and only if, the facing square is unoccupied and on the board.
turn right turn left turn up turn down	Change the player's direction so that she is facing right, left, up, or down. These requests always succeed.
show board	Print out a grid showing the region of the board surrounding the player, in which entities are represented by their Unicode characters, and empty squares are represented as dots (illustrated in example of game play). You are free to choose how big this region should be.
show player	Print out the position of the player, and a list of all the items that she owns (illustrated in example of game play).
show facing	If there is an NPE in the facing square (that is to say the square that is adjacent to the player, in the direction she is facing), print out a description of that NPE, including its position and a list of the items it owns (illustrated in example of game play).
quit	Quit the game

When there is an NPE in the facing square, additional requests become possible, as follows:

Request	Meaning
jump	Jump over the entity in the facing square, and into the square behind it. This request succeeds if the facing square is occupied, the square behind it is unoccupied, and the entity in the facing square can be jumped (in later sections we will explain the criteria that determine whether an entity can be jumped).
eat	This request succeeds if the facing square contains an entity that allows itself to be eaten. See the description of the Herb and Bread entities below.
batter	This request succeeds if the facing square contains an entity that allows itself to be battered. See the description of the Block entity below.

Entities and Items

The game contains the following types of entity:

Entity Type	Description
Player	The game contains exactly one player, and the player is the only entity that can move. The player can have items named “agility” and “strength”, which affect her ability to jump over and batter other entities. When the board is printed, the player is represented by an arrow indicating the direction that she is facing.

Block	A block contains a “height” item, whose quantity indicates the agility required to jump over it. The height must be in the range 1 to 9. When the board is printed, a block is represented by a single digit indicating its height. If the square facing the player contains a block, the user can make `jump` and `batter` requests. A `jump` request succeeds if the square behind the block is empty, and the player’s agility is greater than the height of the block. A successful `jump` request reduces the player’s agility by one and moves her into the square behind the block. An unsuccessful `jump` request does not change anything. A `batter` request succeeds if, and only if, the player is facing a block and has a strength greater than zero. A successful `batter` request has the following effects: ● If the height of the block is less than 2, it is removed from the board. Otherwise, its height is reduced by 2. ● The player’s strength is reduced by 1. An unsuccessful `batter` request does not change anything.
Herb	A herb entity can possess an “agility” item. If it is in the square facing the player, the user can type an `eat` request. This results in the herb being deleted from the board and all its agility being transferred to the player. When the board is printed out, a herb item is represented by the character #. A herb can be jumped over with no change to the items owned by either the player or the herb.
Bread	A bread entity can possess a “strength” item. If it is in the square facing the player, the user can type an `eat` request. This results in the bread being deleted from the board and all its strength being transferred to the player. When the board is printed out, a bread item is represented by the character @. A bread entity can be jumped over with no change to the items owned by either the player or the bread.

Representing the State of the Game

The state of the game must be represented using nested dictionaries, as set out in the following sections. We first explain how entities are represented, then how the positions of non-player entities are represented, then put it all together to explain how the state of the game, in its entirety, is represented,

Representing Entities

Each entity must be represented using a dictionary with some or all of the following keys

Key	**Allowed values**	**Entity types that have the key**
`‘type’`	`‘player’`,`’herb’`,`’bread’`, or `‘block’`	All entities
`‘agility’`	Non-negative integer	player, herb
`‘strength’`	Non-negative integer	player, bread
`‘height’`	Positive integer	block
`‘orientation’`	`‘right’`,`’left’` ,`’up’` or `‘down’`	player (the value indicates the direction in which the player is facing)

Example: The dictionary

```
{'type': 'block', 'height': 3}
```

represents a block with height 3.

Representing the Positions of Non-Player Entities

The positions of all entities other than the player are represented using a dictionary whose keys are tuples representing the column and row number of the square on which the entity is located, and whose associated value is a dictionary representing the entity, in the manner described in the previous subsection[2].

Example: The dictionary

```
{(5, 0): {'type': 'block', 'height': 3}, (1, 2): {'type': 'herb',
'agility': 6}}
```

indicates that the non-player entities are a block with height 3 at column 5, row 0; and a herb with agility 6 at column 1 row 2.

Representing the Entire State

The state of the game is represented by a dictionary with three key-value pairs, as follows

Key	**Value**
`playerSquare`	A tuple indicating the column and row number of the square at which the player is located.
`player`	A dictionary describing the player entity, in the format previously described.
`others`	A dictionary describing the non-player entities, and their positions, in the format previously described.

Example: The following dictionary

```
{'playerSquare': (1, 1),
'player': {'type': 'player', 'orientation': 'down', 'agility': 0,
'strength': 0},
'others': {(5, 2): {'type': 'block', 'height': 3}, (1, 2): {'type': 'herb',
'agility': 6}}}
```

describes a game in which:

- The player is at column 1, row 1, facing downwards, with 0 agility, and 0 strength
- There is a block of height 3 at column 5 row 2.
- There is a herb with agility 6 at column 1, row 2.

This game state is illustrated in Figure 1, in which the player is represented by an arrow, the herb by the character `#`, and the block by an integer indicating its height.

[2] This representation makes life easy if you know what square you are interested in, and want to find out what entity, if any, is at that square. If you know the entity you are interested in, and want to find its location, then the representation is harder to use. This is why we don't use it to represent the position of the player.

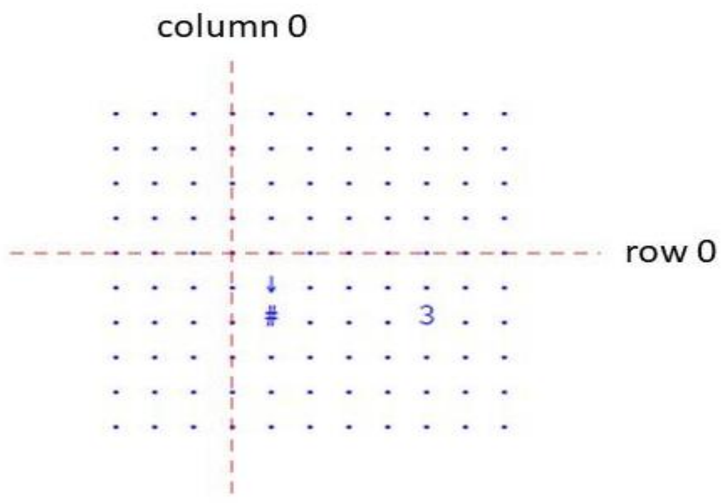

Figure 1 Game State Described by Example Dictionary

What You Have to Do

You must implement a set of functions that will allow you to access and change the state of the game, then use these to write a function that will allow a user to play the game with a command line interface, then describe (without writing code) how they could be used to implement a version of the game with a GUI interface.

Clarification: Added 7 Nov 2022: You must complete this exercise without adding any `import` statements to your code.

Part A: Functions to Access and Change the Game State

You must implement the following functions, which access and change the game state. Some (but not all) of these functions return an integer value indicating success or failure. In general, negative return values indicate failure to perform an operation, non-negative ones indicate success.

A function should not produce any output unless the specification says that it should. The only functions in part A that produce output are `showBoard`, `showPlayer`, and `showFacing`.

Exercise 1: `initialiseState`

Write a function with the following first line

```
def initialiseState(col,row,orientation,agility,strength):
```

The function should return a game state containing only the player, with the player's column and row being specified by the values of the parameters, `col`, `row`, her orientation specified by the value of `orientation` (which should be `'right'`,`'left'`,`'up'`,or `'down'`) and her agility and strength by the values of `agility` and `strength`.

The function should **not** check whether the values of the parameters are legal. The effect of calling the function with illegal values (for example an `orientation` value of `‘upwards’`) should **not** be to return some sort of error code or to directly throw an exception (although it may result in an exception being thrown by some function that is called later).

Example of use (in a Python console). In this example, and those that follow, user input is in ***bold green italics***. Output is in black.

```
> state = initialiseState(1,2,'down',3,4)
> print(state)
{'playerSquare': (1, 2), 'player': {'type': 'player', 'orientation':
'down', 'agility': 3, 'strength': 4}, 'others': {}}
```

Exercise 2: addHerb

Write a function with the following first line

```
def addHerb(state,col,row,agility):
```

The function should add a herb to the game. The `state` parameter should be a list representing the game state. The values of the `col`, `row`, and `agility` parameters are as in Exercise 1, except that now their values specify the properties of a herb instead of the player. The function should **not** check whether the values of the parameters are legal and should not return a value.

Example of use (in a Python console, **bold black text** is used to draw the reader’s attention to elements of the state that have changed and would not be bold in the original output).

```
> state = initialiseState(1,2,'down',3,4)
> print(state)
{'playerSquare': (1, 2), 'player': {'type': 'player', 'orientation':
'down', 'agility': 3, 'strength': 4}, 'others': {}}

> addHerb(state,5,6,7)
> print(state)
{'playerSquare': (1, 2), 'player': {'type': 'player', 'orientation':
'down', 'agility': 3, 'strength': 4}, 'others': {(5, 6): {'type': 'herb',
'agility': 7}}}
```

Exercise 3: addBread

Write a function with the following first line

```
def addBread(state,col,row,strength):
```

The function should add a bread entity to the game. The `strength` parameter has an integer value representing the strength associated with the bread. The values of the other parameters are as in Exercise 2, except that now their values specify the properties of a bread, instead of a herb, entity. The function should **not** check whether the values of the parameters are legal and should not return a value.

Example of (in a Python console, **bold black text** is used to draw the reader's attention to elements of the state that have changed and is not part of the original output).

```
> state = initialiseState(1,2,'down',3,4)
> print(state)
{'playerSquare': (1, 2), 'player': {'type': 'player', 'orientation':
'down', 'agility': 3, 'strength': 4}, 'others': {}}

> addBread(state,5,6,7)
print(state)
{'playerSquare': (1, 2), 'player': {'type': 'player', 'orientation':
'down', 'agility': 3, 'strength': 4}, 'others': {(5, 6): {'type': 'bread',
'strength': 7}}}
```

Exercise 4: `addBlock`

Write a function with the following first line

```
def addBlock(state,col,row,height):
```

The function should add a `block` entity to the game. The `height` parameter has an integer value representing the height of the bloc The values of the other parameters are as in Exercise 2, except that now their values specify the properties of a `block`, instead of a `herb`, entity. The function should **not** check whether the values of the parameters are legal and should not return a value.

Example of (in a Python console, **bold black text** is used to draw the reader's attention to elements of the state that have changed and is not part of the original output).

```
> state = initialiseState(1,2,'down',3,4)
> print(state)
{'playerSquare': (1, 2), 'player': {'type': 'player', 'orientation':
'down', 'agility': 3, 'strength': 4}, 'others': {}}

> addBlock(state,7,8,3)
> print(state)
{'playerSquare': (1, 2), 'player': {'type': 'player', 'orientation':
'down', 'agility': 3, 'strength': 4}, 'others': {(7, 8): {'type': 'block',
'height': 3}}}
```

Exercise 5: `getEntityAt`

Write a function with the following first line:

```
def getEntityAt(state,col,row):
```

The first parameter is the state of the game, the second and third are a column and row on the game. If there is an entity at the specified column and row, then the function returns the dictionary that describes that entity. This should be a reference to the actual dictionary used in the state, not a copy of that dictionary. If there is no entity at the specified column and row, the function should return an empty dictionary.

Example of use (in a Python console):

```
> state = initialiseState(1,2,'down',3,4)
> addHerb(state,5,6,7)

> ent1 = getEntityAt(state,5,6)
> print(ent1)
{'type': 'herb', 'agility': 7}

> ent2 = getEntityAt(state,1,2)
> print(ent2)
{'type': 'player', 'orientation': 'down', 'agility': 3, 'strength': 4}

> ent3 = getEntityAt(state,8,9)
> print(ent3)
{}
```

Exercise 6: `showBoard`

Write a function with the following first line:

```
def showBoard(state,cols,rows):
```

The first parameter is the state of the game, and it should print out a grid of characters displaying a region of the game, centred on the player, of width 2*`cols`+1, and of height 2*`rows` + 1.

In this grid:

- Empty squares are represented using the character '.'
- The player is represented by an arrow indicating the direction in which she is facing. You should use the arrow characters whose Unicode values (in hexadecimal) are 0x2190 (left arrow), 0x2191 (up arrow), 0x2192 (right arrow), and 0x2193 (down arrow).
- Herb entities are represented by the character '#'
- Bread entities are represented by the character '@'
- Block entities are represented by an integer whose value is the height of the block

The grid should be displayed so that column numbers are increasing from left to right and row numbers from top to bottom. Here is an example of use (in a Python console):

```
> state = initialiseState(0,0,'down',3,4)
> addHerb(state,3,2,6)
> addBread(state,-1,3,5)
> addBlock(state,4,-1,3)
> showBoard(state,6,4)
. . . . . . . . . . . . .
. . . . . . . . . . . . .
. . . . . . . . . . . . .
. . . . . . . . . . 3 . .
. . . . . . ↓ . . . . . .
. . . . . . . . . . . . .
. . . . . . . . . # . . .
. . . . . @ . . . . . . .
. . . . . . . . . . . . .
```

Exercise 7: getPlayer

Write a function with the following first line:

```
def getPlayer(state):
```

The parameter of the function is the dictionary representing the state of a game. It should return the dictionary representing the player within this state. The returned value should be a reference to the player dictionary, not a copy of that dictionary. This means that changes made to the returned dictionary will change the state of the game.

Example of use (in a Python console). Bolding is used to indicate values that have changed and is not part of the original output. Note what happens to the strength of the player and note that variables p1 and p2 end up referring to the same dictionary.

```
> state = initialiseState(1,2,'down',5,6)
> p1 = getPlayer(state)
print(p1)
{'type': 'player', 'orientation': 'down', 'agility': 5, 'strength': 6}
> p1['strength'] = 99
> p2 = getPlayer(state)
> print(p2)
{'type': 'player', 'orientation': 'down', 'agility': 5, 'strength': 99}
```

Exercise 8: getPlayerSquare

Write a function with the following first line:

```
def getPlayerSquare(state):
```

The function should return the tuple that describes the player’s position.

Example of use

```
> state = initialiseState(1,2,'down',5,6)
> square = getPlayerSquare(state)
> print(square)
(1, 2)
```

Exercise 9: `turn`

Write a function with the following first line:

```
def turn(state,orientation):
```

This function should change the direction in which the player is facing. The `state` parameter is the current state of the game, and the value of the `orientation` parameter should be either `'up'`, `'down'`, `'left'` or `'right'`. If the `orientation` parameter has a legal value, the state should be updated to make the player face the specified direction, and the function should return a non-negative integer value. If the `direction` parameter has an illegal value then the state should not change, and the function should return a negative value. You are free to choose which values to use, so long as they are negative for failure, and non-negative for success.

Example of use (in a Python console – **bold** text is used to highlight changes in state and is not part of the console output). In this example the `turn` function returns 1 if it is successful.

```
state = initialiseState(1,2,'down',5,6)
print(state)
{'playerSquare': (1, 2), 'player': {'type': 'player', 'orientation':
'down', 'agility': 5, 'strength': 6}, 'others': {}}
turn(state,'left')

1
print(state)

{'playerSquare': (1, 2), 'player': {'type': 'player', 'orientation':
'left', 'agility': 5, 'strength': 6}, 'others': {}}
```

Exercise 10: `step`

Write a function with the following first line:

```
def step(state):
```

This function should move the player by one square in the direction she is facing, so long as this is possible. If the step would move the player onto a square occupied by another entity, then the state should remain unchanged, and the function should return negative value. Otherwise, the state should be altered so as to change the position of the player in the appropriate way, and the function should return a non-negative value.

Example of use (in a Python console). **Bold** text is used to highlight changes in state and would not be bold in the original console output. In this example the function returns 1 if the step has been successfully executed.

```
state = initialiseState(1,2,'right',5,6)
print(state)
{'playerSquare': (1, 2), 'player': {'type': 'player', 'orientation':
'right', 'agility': 5, 'strength': 6}, 'others': {}}

step(state)
1

print(state)
{'playerSquare': (2, 2), 'player': {'type': 'player', 'orientation':
'right', 'agility': 5, 'strength': 6}, 'others': {}}
```

Exercise 11: `showPlayer`

Write a function with the following first line:

```
def showPlayer(state):
```

The parameter of the function is the state of the game, and it should print out the position of the player, and the quantities of the items she owns.

Example of use (in a Python console):

```
> state = initialiseState(1,2,'right',5,6)
> showPlayer(state)
column:  1
row:  2
orientation :  right
agility :  5
strength :  6
```

Exercise 12: `showFacing`

Write a function with the following first line:

```
def showFacing(state):
```

The parameter of the function is the state of the game, and the function should behave as follows:

If there is an entity in the square that is adjacent to the player in the direction that the player is facing, then the position of that entity, and the quantities of the items it owns, should be printed out. Otherwise, a message indicating that the square is unoccupied should be printed out.

Example of use (in a Python console).

```
> state = initialiseState(1,2,'right',5,6)
> addBlock(state,2,2,10)
> showFacing(state)
column:  2
row:  2
type :  block
height:  10
```

Exercise 13: `eat`

Write a function with the following first line:

```
def eat(state):
```

The value of the parameter `state` should be the current state. The effect of a call to the function should be that:

- If the player is adjacent to, and facing, a herb or bread entity then the agility, or strength, of that entity is transferred to the player, and the entity is removed from the board. In other words, the state changes in the way described in the rules of the game. In this case, the function should return a non-negative integer.
- In all other cases the state of the game is unchanged, and the function returns a negative value. You should use different negative values to distinguish between the case where the facing square is empty and the case where it contains an inedible entity (i.e., a block).

Example of use (in a Python Console)

```
> state = initialiseState(4,3,'right',1,2)
> addHerb(state,5,3,7)
> showPlayer(state)
column:  4
row:  3
orientation :  right
agility :  1
strength :  2
> showFacing(state)
column:  5
row:  3
type :  herb
agility :  7

> eat(state)
1

> showPlayer(state)
column:  4
row:  3
orientation :  right
agility :  8
strength :  2
> showFacing(state)
No entity in facing square
```

Exercise 14: `batter`

Write a function with the following first line:

```
def batter(state):
```

The value of the parameter `state` should be the current state. The effect of a call to the function should be that:

- If the player is adjacent to, and facing, a block entity then the state changes in the way specified by the rules. In other words:
  - If the height of the block is less than 2, it is removed from the board. Otherwise, its height is reduced by 2.
  - The player's strength is reduced by 1.

  In this case the function should return a non-negative integer value.

- In all other cases the state of the game is unchanged, and the function returns a negative value. You should use different negative values to distinguish between the case where the facing square is empty and the case where it contains an entity which is not a block.

Example of use (in a Python Console)

```
> state = initialiseState(6,5,'right',3,4)
> addBlock(state,7,5,3)
> showPlayer(state)
column:  6
row:  5
orientation :  right
agility :  3
strength :  4
> showFacing(state)
column:  7
row:  5
type :  block
height :  3
batter(state)
1
> showPlayer(state)
column:  6
row:  5
orientation :  right
agility :  3
strength :  3
showFacing(state)
column:  7
row:  5
type :  block
height :  1
> batter(state)
1
> showPlayer(state)
column:  6
row:  5
orientation :  right
agility :  3
strength :  3
> showFacing(state)
No entity in facing square
```

Exercise 15: ~~eat~~ jump (corrected 24 Nov 2022)

Write a function with the following first line:

```
def jump(state):
```

The value of the parameter `state` should be the current state. The effect of a call to the function should be that:

- If the player can, according to the rules, make a jump, then the state should be adjusted to reflect this. In other words, the player moves into the unoccupied square behind the facing entity, and her agility is reduced in the way set out in the to the rules of the game. The function should then return a non-negative value.
- If a jump is not possible, the state of the game is unchanged, and the function returns a negative value. You should use different negative values to distinguish between the different reasons why a jump is not possible: that the square in which the player would land is occupied; that the square over which the player would jump is not occupied; and that the square over which the player would jump is occupied by a block that is too high.

Exercise 16: readState

Write a function with the following first line:

```
def readState(file):
```

The value of the parameter file is the name of a file, whose contents are in a format described below. The function returns a dictionary describing a game state.

Each line in the file describes an entity and contains between 4 and 6 text strings, separated by white space. The first string is the type of the entity, the next two are integers describing the column and row at which the entity is located. The remaining strings differ according to the entity type. If it is the player, they are the agility and strength of the player (as integers, in that order) and the orientation of the player (which must be `up`, `down`, `left`, or `right`). In any other case, we have a single integer value denoting the value of the height of a block, the agility of a herb, or the strength of a bread entity.

Example of use:

Suppose the file state.txt has the following content

```
player 1 2 3 4 down
bread 3 2 6
block 5 2 2
herb 4 2 7
block 4 4 3
```

Then following input and output would be seen in a Python console (assuming that state.txt is in the console's working directory).

```
state = readState('state.txt')
showBoard(state,6,3)
. . . . . . . . . . . . .
. . . . . . . . . . . . .
. . . . . . . . . . . . .
. . . . . . ↓ . @ # 5 . .
. . . . . . . . . . . . .
. . . . . . . . . 4 . . .
. . . . . . . . . . . . .
```

You may assume that the file is correctly formatted. If it isn't then the function should not print an error message, or throw an exception, and can return any value you like[3].

Part B: Implementing the Command Line Game

Using the functions that you have written in part A, write a function with the following first line:

```
def playConsole(file):
```

This function should allow the user to play the game. The user should be prompted to enter requests which will allow the player to be moved and to interact with entities on the board. The requests should be interpreted in the way set out at the beginning of the document (in the section called "The Game")

The parameter of the finction is the name of a file that specifies the initial state of the game in the format required by the `readState` function.

Part C: Designing a GUI Version of the Game

Outline a design for a GUI version of the game. You do NOT have to write any code for this. You should:

- Sketch out the appearance of the GUI. A hand drawn diagram is sufficient, so long as it is readable.
- Describe how each part of the GUI functions (for example, if there is a button on the GUI, what happens when that button is pressed).
- Without writing any code, explain how the functionality described above could be implemented, using tkinter widgets, and reusing (without modification) the functions implemented in part A. Note this does **not** mean that, when implementing those functions, you should add code that is unnecessary for the console version of the game.

What you must submit

You should submit a single file to Turnitin containing:

- All your code, written in a fixed width font such as `Courier New`. You can, if you wish, submit your work as a text file, but it must have the extension .txt (Turnitin will not cope with .py files).
- A document explaining how your GUI design, as described in part C.

[3] This design decision has been taken purely to allow you to complete this exercise without getting bogged down in detail.

Mark Scheme

There are three components to the mark scheme:

- **Functionality of your code (14 marks):** These marks are given for implementing the functions set out in parts A and B so that they return correct values and produce correct output. Half a mark is awarded for each function in part A, and six marks are awarded for the `playConsole` function set out in part B.
- **Code quality (7 marks):** These marks are awarded for the quality of your code. To get these marks you should:
    - Use meaningful names for variables and functions.
    - Use loops and lists effectively to avoid writing unnecessarily repetitive code.
    - Ensure that your code is no more complicated than it needs to be.
    - Avoid the use of `break`, `continue`, and `pass`.
    - Use `try ... except` blocks cautiously (if you use them at all). In particular, don't use a `try ... except` block for something that can easily be handled using an `if` statement.

  The marks available for code quality will be reduced if you have not implemented all the required functions. For example, you can't get all 7 code quality marks by implementing a single function, no matter how elegantly you did it!

- **GUI Design (4 marks):** These marks are awarded for the GUI design described in part C.