

F01 Team 1D - Impossible Maze

Members:

- Sng Wei Qi Amos (1005952)
- Soh Zhi Ying (1006359)
- Edwin Wongso (1006200)
- Zaina Aafreen (1006145)
- Tan Yu Quan (1006355)

Documentation

Since our mini-games are designed to be modular, our program can be broken down into one main program to four sub-programmes. The sub-programmes are as follows:

- Typing Game
- Number Pattern Game
- Quick Math Game
- General Knowledge

NOTE: Although we mention it as an individual programme, we combine them into one python file to improve reliability.

Main Programme

We created a maze by first creating a 10x10 grid of closed cells. Our program removes some walls to make a maze. We used a 2d list in python to represent the grid in code. The maze generation algorithm is called the hunt and kill algorithm.

The way this maze-generation algorithm works:

1. Scan the grid row by row, column by column like reading through a paragraph of words.
2. If the program finds a non-visited cell, it chooses that cell as the “parent”.
3. Start a random walk from that cell to adjacent cells, the “children” of the parent. Recursively construct the path by randomly choosing up, left, down, right at every cell, and remove the wall between the next cell and the current cell, i.e. if you go right, remove the current cell’s rightwall and the subsequent cell’s leftwall. Avoid visited cells in the path.
4. When the program reaches a cell which has no available adjacent squares to visit, the walk ends, and we have to find a new initial cell.
5. Repeat step 1.
6. This new initial cell must have adjacent cells that have been visited. Remove the wall between a randomly chosen visited adjacent cell, this unites previously created paths to new paths, and ensures that every square in the grid is reachable from every other square, and that the maze has a solution.
7. The program chooses a random cell on the first column and opens the left wall of this cell, this is the entrance of the maze, and then the program chooses a random cell on the last column of the maze and opens the right wall of this cell, this the exit of the maze.

Initialising variables:

score = [1000] (we used a single-element list to represent score because lists are mutable and far simpler to deal with compared to global variables in python)

Font_parameter = ('Arial',16,'normal') (A tuple to specify the font for any text that turtle.write() uses.

Size = [10] (Number of rows and columns of the maze)

START_X = -250 (The x-coordinate of the grid's top left corner. Cell (0,0) in the maze.

START_Y = 185 (The x-coordinate of the grid's top left corner. Cell (0,0) in the maze.

Side_length = 36 (Length of one cell in pixels, used when drawing squares in turtle)

Assign numbers to each direction so later we can randomise more easily when choosing directions

Left = 1

Right = 2

Up = 3

Down = 4

grid = [[]] (A 2d list that will be made up of Square objects)

visited_sq = [[]] (A 2d list indicating which cells have been visited)

maze = turtle.Turtle() (A turtle pen for drawing maze)

sprite = turtle.Turtle() (A turtle for drawing the sprite)

textpen = turtle.Turtle() (A turtle for writing text using textpen.write("") command)

Scorepen = turtle.Turtle() (A turtle pen to write and update the score and number of steps left, this scorepen is separate from the textpen turtle so that even if we clear all text, the score is still displayed)

def clear_maze_and_sprite():

- Uses maze.clear() and sprite.clear() to clear the maze and player sprite from the screen, this function is called whenever a minigame is about to be played so that the txt appears on a white background with no maze or sprite.

def update_score_and_steps(): Uses scorepen to update the score and number of steps in the top right corner of the screen.

class Maze_control: A class that defines a single cell in the maze.

def __init__(self, centre_coordinate, closed_upwall, closed_rightwall, closed_downwall, closed_leftwall):

- Each cell has five attributes: upwall, leftwall, downwall, rightwall and center coordinate. Upwall, leftwall, downwall and rightwall are booleans, if set to true, it means that wall is closed, else if false, then the wall is open. Initially all four walls are closed for every cell in the 10x10 grid, but during maze generation some walls will be opened.
- This function sets the centre coordinate for each cell so that later, it's easier to instruct the turtle pen to go to that center, and draw the walls it has to. The four walls of the square object are all set to closed at first, but later on they may be removed. The state of these walls are booleans, True meaning closed and False meaning opened.

def draw_walls(self): After the maze has been created in code form, if wall is present this function uses turtle.pendown() to draw the wall of each cell in the grid, else uses penup() to avoid drawing that wall.

def refill_possible_choices(self, row, col):

1. This function is called during maze generation later, when we are constructing a random walk from a parent cell.
2. This function operates on one cell in the grid at a time, grid[row][col] where row and col were passed as parameters to this function.
3. We start with
4. choices = [up, left, down, right]
5. (remember that we assigned numbers to these directions previously)
6. Erase upward direction (up) from choices if grid[row-1][col] has been visited, or if row-1<0, as that cell is non-existent.

7. Erase rightward direction (right) from choices if `grid[row][col+1]` has been visited, or if `col+1>size`, as that cell is non-existent.
8. Erase downward direction (down) from choices if `grid[row+1][col]` has been visited, or if `row+1>size`, as that cell is non-existent.
9. Erase leftward direction (left) from choices if `grid[row-1][col]` has been visited, or if `col-1<0`, as that cell is non-existent.
10. return modified choices

def find_initial_cell(self):

1. For every row in grid:
 - a. For every column in row:
 - i. If `visited[row][col] = 0`:
 1. If `row>0` and `col>0`:
 2. Out of all of the adjacent visited cells, choose one randomly and construct a path from that cell to the current cell by removing the wall between them. Assign this last visited cell to a variable `previous_square`.
2. Construct a path from this cell using `self.find(adjacent(row,col))`.

def find_adjacent(self,row,col):

1. Sets visited_sq[row][col]=1 to mark current cell as visited
2. Calls refill_possible_choices() to eliminate impossible directions.
3. Assigns the return value from the last step to a list called choices.
4. If choices is empty, then end the function here as the current cell has no more unvisited neighbours.
5. Takes random direction from choices and assigns it to direction
6. Breaks down the wall in that direction, and the opposite wall for the neighbouring cell.
7. For example, if direction == left, grid[row][col].leftwall is set to false meaning open, and grid[row][col+1].rightwall is set to False as well, this makes a path between these two cells in the grid.
8. Then the function recursively calls itself, but this time it passes the adjacent cell's row and column as input parameters, (e.g. if direction == left, the function recursively calls itself using row, col-1 as input parameters in the subsequent call).

def refill_last_visited_choices(self, row, col):

- When finding a new initial cell, we use this function to create a list of possible directions to choose from when finding a previously visited cell to connect the current initial cell to.
- Create a list of directions called choices.
- choices = [up, right, down, left]
- For each direction, if the square adjacent to the current cell is unvisited or non-existent, remove it from choices. For example if visited[row+1][col] == 0, remove the down direction from the choices list.

Class Sprite(center):

def __init__(self, center coordinate, player row, player column):

This function has 3 attributes. The location of the sprite's current center coordinate, the current row, and current column of the player in the grid / maze.

def draw_sprite(self):

1. sprite.clear() to remove the sprite from the screen
2. Move the sprite pen to the current self.center_coordinate. Then move the pen to the top left corner of the sprite square.
3. For count in range 0<=count<=3:
 - a. Draw a line.
 - b. Rotate sprite 90 degrees clockwise.

Now the sprite is in its new location

def moveup(self): Checks to see if the sprite can move up,

1. if up wall is open and up cell is within grid dimensions:
 - a. sprite moves up by one cell
 - b. returns true,
2. else:
 - a. nothing happens
3. the function returns false.

def moveright(self): Checks to see if the sprite can move right,

1. if right wall is open and right cell is within grid dimensions:
 - a. sprite moves right by one cell
 - b. returns true,
2. else:
 - a. nothing happens
 - b. the function returns false.

def movedown(self): Checks to see if the sprite can move down,

1. if down wall is open and down cell is within grid dimensions:
 - a. sprite moves down by one cell
 - b. returns true,
2. else:
 - a. nothing happens
 - b. the function returns false.

def moveleft(self): Checks to see if the sprite can move left,

1. if left wall is open and left cell is within grid dimensions:
 - a. sprite moves left by one cell.
 - b. returns true,
2. else:
 - a. nothing happens
 - b. the function returns false.

def follow_instructions(self,instructions):

1. Function takes in a string called instructions with characters (u/d/r/l) indicating the direction the player wants to go.

2. First it stores the player's current row and column. Then it checks if the path from the user is possible, and doesn't hit any walls. If it does hit walls, the function reverts the row and column of the player back to the initial values that were stored in the function earlier.
3. If the path is impossible, the function returns false, else it returns True.

Now we start actually making the maze. This will also serve as the main function of the entire game.

1. Start by creating variables
2. `cur_x = START_X`
3. `cur_y = START_Y`
4. Use nested for loop to Initialise all the values of `grid[][]` and `visited_sq[][]`
5. We then create a `maze_control` variable called `maze_methods`, and this is just to access any methods in the `maze_control` class such as `find_initial_cell()`, or `refill_possible_cells()`, we won't use `maze_methods` for its objects.
6. We use `maze_methods.refill_possible_choices(0,0)` to start find possible choices for adjacent cells of the cell (0,0).
7. Then we call `maze_methods.find_initial_cell()` which will create the maze using recursion.
8. Then we make a variable called `left_opening` and `right_opening` to specify which row is the entrance and which row is the exit of the maze in.
9. Then we open the right wall of the exit cell and left wall of the entrance cell.
10. We create a `player_start_point` variable that is a tuple, and assign the entrance cell's central x-y coordinates to it.
11. Then we created a `Sprite` variable called `Player` which will be the sprite that the player controls. `player = Sprite(player_start_point, 0, left_opening)`.
12. Then we use `update_score_and_steps` to display the score and `no_of_steps[0]` on the top right of the screen.
13. We then give an intro of the game to the player, we display this intro in the form of textinput and the person just has to press enter to continue.
14. Now we make a while loop to ask the player questions. This portion onwards will act as the main programme that collates all the sub-programme we have created.
 - a. Initialising the following function:
 - i. **`screen.tracer (0)`**
 - ii. **`player .draw_sprite()`**
 - iii. **`maze_methods .redraw_maze()`**
 - iv. **`screen_update()`**
 - b. Initialise the variable needed to pass through into each sub-programme, and we generate a random number between the range of 0 to 3. We then check for the randomly generated number to decide which minigame sub-programme to run.
 - c. Divert to the individual sub-programme
 - d. Receive the result from sub-programme. If they have successfully completed it, the number of steps they can take is updated based on the difficulty they have chosen initially. For example, if the player has successfully complete the sub-programme of difficulty 3, they will be able to move 3 steps in the maze.

- e. If they did not succeed in 14(d), they will be looped back to 14(b).
 - f. The maze and score will be updated via running **update_score_and_steps()**, **maze_methods.redraw_maze()** and **player.draw_sprite()**.
 - g. We will prompt the user in the direction to move. While loop is used to check for the validity of the input from the user
 - i. If the input length is not the same as the number of steps permitted, it will state that it is invalid and prompt the user to input again.
 - ii. If the input contains letters that are not defined or stated in the instruction, it will state that it is invalid and prompt the user to input again.
 - iii. The input is then passed into **player.follow_instruction()** to check. It will state that it is invalid and prompt the user to input again if the instruction is not valid. For example, user gave directions that causes him to run into a wall.
 - iv. If the instruction is valid, we will reinitialise the number of steps back to 0 and exit the loop
 - h. If they have reached the end of the maze, it will exit the while loop.
15. Clear the screen and thank the user for playing our game.

Sub-Programme 1: Typing Game

Class typinggame:

Creates a class type object to store attributes of the player throughout the playthrough of the typing game.

def __init__(self). Creates attributes for 1 instance of the player playing the game.

def reset_canvas(self,x,y). This function is used to clear the current writings on the turtle screen. The parameters x and y are used to set the position of the turtle after clearing the canvas.

def game_instructions(self). This function displays the instructions to the typing game on the turtle screen when executed. It will ask the user to press Enter to proceed once they are done reading the instructions.

def select_level(self). This function calculates the time the player has and number of sentences the player has to complete based on the level he has selected. It updates the attribute typinggame.required_sentence to the number of sentences the user has to complete. It also updates the attribute typinggame.time to the amount of time the user has to complete the number of sentences in typinggame.required_sentence. The function also reiterates to the player how many sentences they must complete and the amount of time they must complete the sentences in the turtle screen.

def timetostart(self). This function asks the player if they are ready to proceed with the game, since the game is time based and every second counts. When the player confirms he is ready by responding with 'y', the function begins a countdown timer of 3 seconds on the turtle screen.

def maingame(self). This function initially sets the attribute typinggame.pass_test to False. It also sets variable switch_sentence to True, which will be used to dictate if the sentence is being switched out depending on if the user enters what is seen on the screen or not successfully.

def get_sentence(self). This function generates the sentences for the player to type by choosing randomised words from the word bank and creates a sentence with less than 20 characters. The generated sentence is then stored in attribute typinggame.sentence

def display_sentence(self). This function displays the sentence generated from **get_sentence** -in the turtle canvas

def display_info(self,time_left,no_completed_sentences). This function takes in the parameters `time_left` and `no_completed_sentences`. It displays these 2 pieces of information on the turtle canvas so that the player knows how many more sentences he must complete and how much more time he has.

def get_player_stats(self). This function calculates the Words per minute (WPM) of the player and Characters per minute (CPM) of the player by counting the number of words and characters in the sentences he has completed divided by the time taken to complete the sentences, multiplied by 60 seconds.

The player can choose to see the info by typing “y” into the input box, or choose not to see by typing “n”.

def gamemechanics(self,no_completed_sentences, completed_sentences, required_sentences, time_left, switch_sentence).

This is the function that runs the game. The pseudocode is as follows.

1. Takes in the parameters `no_completed_sentences`, `completed_sentences`, `required_sentences`, `time_left`, `switch_sentence` that are stored in the function **maingame**.
2. While `time_left > 0` and `switch_sentence == True`, run **get_sentence** to generate a sentence.
3. While `time_left > 0` and `switch_sentence == False`, pass, so that the old sentence is repeated.
4. Execute **display_sentence** to show the sentence the player should type on the turtle screen.
5. Execute **display_info** to inform user of the number of sentence he has remaining and time remaining. (time remaining for the first iteration will be the total time allotted for player to clear the level as the timer has not started)
6. Start the timer and prompt player for input.
7. If player input matches sentence generated,
 - a. Stop the timer
 - b. Calculate the time taken by the player to type the sentence by taking `endtime – start time`.

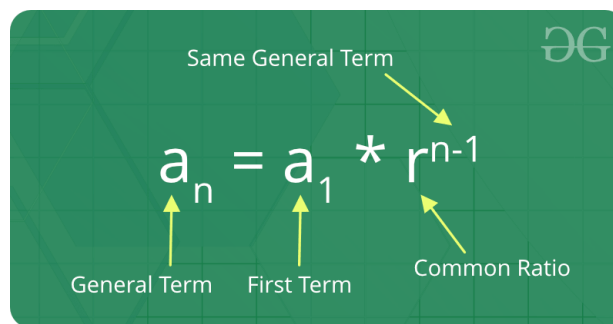
- c. Add 1 to the variable `no_sentences_completed` and calculate the number of remaining sentences by taking `required_sentence - no_completed_sentences`
 - d. Update attribute `typinggame.timeleft` and `typinggame.remaining_sentences`
 - e. If `remaining_sentences == 0` and `time_left > 0`
 - i. Player has passed the test, change attribute `typinggame.pass_test` to `True`.
 - ii. Call function **`display_pass`** to inform the player he has passed the test on turtle.
 - iii. Ask player if he wants to see his typing speed (WPM and CPM) by calling function **`get_player_stats`**
 - f. If `remaining_sentences > 0` and `time_left > 0`
 - i. Player has not completed the game, call function **`game_mechanics`** to form another sentence with the updated parameters and run from point 1.
 - g. If `remaining_sentences > 0` and `time_left < 0`
 - i. Player has failed the test, leave attribute `typinggame.pass_test` as `False`.
8. If player input does not match sentence generated,
- a. Change `switch_sentence` to `False`
 - b. Stop timer and calculate take taken similar to 7b.
 - c. If `time_left > 0`
 - i. Give player another chance to try again and call function **`game_mechanics`**
 - d. Else break the while loop and announce the player failed the test by calling function **`display_fail`**

Sub-Programme 2: Number Pattern Game

This sub-programme requires the random, time and turtle library.

def geo_seq(). This function is used to generate a geometric sequence. The following are the pseudocode of the function:

1. The total number of terms (5) to be generated and an empty list for storing it is initialised.
2. First term and common ratio are randomly generated within a specified range
3. For the total number of terms to be generated
 - a. Apply the formula in *Fig 2.1*, where n is the no of time the for loop has looped
 - b. Add the number generated into the list created
4. If the first and second term in the list is exactly the same, rerun the function again. This is one of the bug found during debugging.
5. Make a shallow copy of the list with the sequence. Replace the last element with "_____". This list will be used as the question
6. Return the list with the full sequence and the question list



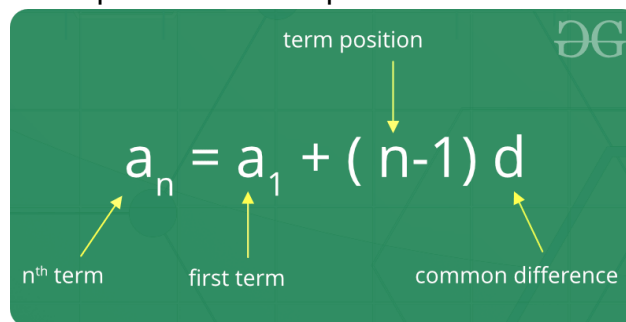
The diagram shows the general term equation for a geometric sequence: $a_n = a_1 * r^{n-1}$. It is set against a green background with a faint grid. Yellow arrows point from labels to parts of the equation: 'General Term' points to a_n , 'First Term' points to a_1 , and 'Common Ratio' points to r . A label 'Same General Term' with an arrow points to the entire equation. A small 'GG' logo is in the top right corner.

Fig 2.1: Geometric Sequence General Term Equation
(<https://www.geeksforgeeks.org/geometric-progression/>)

def arithmetic_seq(). This function is used to generate an arithmetic sequence. The following are the pseudocode of the function:

1. The total number of terms (5) to be generated and an empty list for storing it is initialised.
2. First term, common difference and difference sign are randomly generated within a specified range. Difference sign will determine if the common difference is positive or negative.
3. For the total number of terms to be generated
 - a. Apply the formula in *Fig 2.2*, where n is the no of time the for loop has looped
 - b. Add the number generated into the list created
4. If the first and second term in the list is exactly the same, rerun the function again. This is one of the bug found during debugging.

5. Make a shallow copy of the list with the sequence. Replace the last element with “_____”. This list will be used as the question
6. Return the list with the full sequence and the question list



The diagram shows the equation $a_n = a_1 + (n-1)d$ on a green background. Yellow arrows point from labels to parts of the equation: 'nth term' points to a_n , 'first term' points to a_1 , 'common difference' points to d , and 'term position' points to $(n-1)$. A small 'GG' logo is in the top right corner.

Fig 2.2: Arithmetic Sequence General Term Equation
<https://www.geeksforgeeks.org/arithmetic-progression/>

def f_seq(). This function is used to generate a Fibonacci number sequence. The following are the pseudocode of the function:

1. The total number of terms (5) to be generated and an empty list for storing it is initialised.
2. The first two terms are randomly generated within a specified range.
3. For the total number of terms to be generated - 2 (as first 2 terms are generated)
 - a. Apply the formula in Fig 2.3
 - b. Add the number generated into the list created
4. If the first and second term in the list is exactly the same, rerun the function again. This is one of the bug found during debugging.
5. Make a shallow copy of the list with the sequence. Replace the last element with “_____”. This list will be used as the question
6. Return the list with the full sequence and the question list

$$F_n = F_{n-1} + F_{n-2}$$

Fig 2.3: Fibonacci numbers General Term Equation
https://en.wikipedia.org/wiki/Fibonacci_number

def tri_num(). This function is used to generate a triangular number pattern. The following are the pseudocode of the function:

1. The total number of terms (5) to be generated and an empty list for storing it is initialised.
2. The first terms are randomly generated within a specified range.
3. For the total number of terms to be generated
(But using the first term as the start and first term +5 as the end)
 - a. Apply the formula in Fig 2.4, where n is the no of time the for loop has looped
 - b. Add the number generated into the list created
4. If the first and second term in the list is exactly the same, rerun the function again.
 This is one of the bug found during debugging.
5. Make a shallow copy of the list with the sequence. Replace the last element with "_____". This list will be used as the question
6. Return the list with the full sequence and the question list

$$T_n = \sum_{k=1}^n k = 1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2} = \binom{n+1}{2},$$

Fig 2.4: Triangular Number General Term Equation
https://en.wikipedia.org/wiki/Fibonacci_number

def figurate_num(). This function is used to generate one of the 3 types of figurate number patterns identified. The following are the pseudocode of the function:

1. The total number of terms (5) to be generated and an empty list for storing it is initialised.
2. A random number between 5 to 7 is generated.
3. The starting term is randomly generated in the range of 0 to 5
4. Check the random number generated
 - a. For 5, the formula will be pentagonal numbers
 - b. For 6, the formula will be hexagonal numbers
 - c. For 7, the formula will be heptagonal numbers
5. For the total number of terms generated,
 - a. Apply their respective formula, where n is the no of time the for loop has looped
 - b. Add the number generated into the list created
6. If the first and second term in the list is exactly the same, rerun the function again. This is one of the bug found during debugging.
7. Make a shallow copy of the list with the sequence. Replace the last element with “_____”. This list will be used as the question
8. Return the list with the full sequence and the question list

Pentagonal numbers	$\frac{3n^2 - n}{2}$
Hexagonal numbers	$2n^2 - n$
Heptagonal numbers	$\frac{5n^2 - 3n}{2}$

Fig 2.5: Figurate Number General Term Equation
(http://oeis.org/wiki/Figurate_numbers)

def setup(x,y). This function is created to reduce the repeat of codes when the screen changes from one state to another (eg. Questions to comments). All the commands used are from the turtle library. It does the following (in sequence):

- Clear the screen to get fresh window
- Label the name of the window
- Initialise the window size. It uses the x and y variables that are being input.
- Hide the cursor and setting the position for the test to be displayed

def pattern_minigame(difficulty). This is the function to run the sub-programme 2. The following are the pseudocode of the function:

1. Initialization of parameter
 - a. Level of the game, which is equal to the difficulty of the game that is being input from the main programme. It will be in the form of 1, 2 or 3.
 - b. A list that stores the number of chances for each level
 - c. The current number of chances left, which is extracted from the chance list based on level
 - d. A boolean that state whether the player has passed the mini-game or not. Let it be `pass_level`. The default is being declared as `False`
 - e. A tuple that stores the font parameters. This is used for printing out the text input via turtle
2. Check the level of the game
 - a. If is 1, the question and answer generated will be by running either **`geo_seq()`** or **`arithmetic_seq()`**
 - b. If is 2, the question and answer generated will be by running either **`f_seq()`** or **`tri_num()`**
 - c. Else, the question and answer generated will be by running **`figurate_num()`**The choosing is done by comparing to a number that is randomly generated.
3. While the chance is more than 0 and they have yet passed the level
 - a. Print out the question generated and ask for user input via turtle
 - b. Do a type conversion of string to integer
 - i. If it fails, it will return a value of `None` instead
 - c. Check the user input with the answer generated
 - i. If it is correct, inform the user via turtle
 - ii. Else, deduct the number of chances by 1 and inform the user of the number of chances leftDo note that the tuple generated in 1(e) is used as part of the print condition and before every print, **`setup(-150,-35)`** will be run.
4. Return `pass_level` to the main programme

Sub-Programme 3: Quick Math Game

This sub-programme requires the random, time and turtle library.

def math_minigame(diff). This is the function to run the sub-programme 3. The following are the pseudocode of the function:

1. Clearing the previous turtle interface and introducing rules of sub-programme 3.
2. Setting clear_stage parameter for the number of tries criteria.
 - a. The amount of tries per player is 2.
 - b. When clear_stage is at index 0, tries will be 1 and the 1st question will be displayed based on the difficulty of the game that was input.
 - c. If a player gets the 1st question wrong, clear_stage will be returned a False boolean.
 - d. If a player gets the 1st question correct, tries will be 2 and the 2nd question will be displayed based on the difficulty of the question.
 - e. If a player gets the 2nd question correct, clear_stage boolean will be returned a True boolean.
 - f. If a player gets the 2nd question wrong, clear_stage will be returned a False boolean.

def question_generator(diff, tries). This function is used to generate the questions. The following are the pseudocode of the function:

1. An empty list named num_list created to store numbers generated from the range.
2. An operator_list is created to store the type or operators + , - , * .
3. To generate a choice of random operator for the question, variable op is created.
4. Range to generate random numbers from 1 to 9 and added to the num_list.
5. Variable qns_string is the base of the question which takes the 1st number from num_list and converts it to a string + a random operator generated in op.
6. A For-loop is written to generate the range based on the difficulty.
 - a. For difficulty 1, the range would be 1 and it takes the base qns_string and adds the 2nd number in the list.
 - b. For difficulty 2, the range would be 2 and it takes qns_string that was already generated in a and adds the 3rd number in the list.
 - c. For difficulty 3, the range would be 3 and it takes qns_string that was already generated in b and adds the 4th number in the list.

7. An eval() function will be used to evaluate the qns_string for Variable ans.
8. Clearing of the previous turtle interface.
9. Using the turtle interface the display qns_string and ask for input from the player.
 - a. Variable player_input is to convert the string into a float.
 - b. If the user types in a word, it will return as 'None' and thus will get the question wrong when there is a ValueError.
10. Clearing of the previous turtle interface.
11. Player's answer evaluation
 - a. If the player's answer is correct for the 1st question, the message shown will be "Going onto the next question". The message will be shown for 1.5s.
 - b. If the player's answer is correct for the 2nd question, the message shown will be "Good Job! Going back to the main page." The message will be shown for 1.5s and will return a True value together with difficulty level.
 - c. If the player's answer is wrong for any question, the message shown will be "You are wrong! Try again next time. Going back to the main page." The message will be shown for 1.5s and will return a True value together with difficulty level.
12. Returns to the main game with boolean and difficulty level.

Sub-Programme 4: Geography

This sub-programme requires the random, time and turtle library.

country_dict is the dictionary containing list of countries with keys lettered A to Z

class geography creates a class type object to store attributes of the player throughout the playthrough of the typing game.

def __init__(self, level) creates attributes for an instance of the player playing a level of the game

def game_intro(self) function clears the turtle screen and displays brief information about the game. Player is prompted to press Enter to continue

def control(self) function runs the game sequence and return self.pass_test and self.level after the game is completed.

def set_difficulty(self) function sets self.no_countries_required to the number of countries required for the level

def get_alphabets(self) function indicates the letters to be randomly generated for question asked

def eliminate_country(self,player_input) function indicates conditions to check input of player

1. if statement to return False if player inputs nothing(None)
2. elif statement to return False if player inputs answer that is not alphabetic
3. if else statement: If player inputs answer that is in the list of countries remove the country from the list of countries and return True, else return False. Input of player is capitalalized before if else statement is applied.

def maingame(self) function indicates how the game works

1. While loop to allow code to be executed repeatedly until condition fails
2. If player inputs "I give up", set self.pass_test is False
3. Break statement issues to end while loop
4. if self.eliminate_country(country) is True,set self.correct_answer to True.
Self.countries_answered increases by 1 and self.display_board() is executed
5. Elif statement: if self.eliminate_country(country) returns False,self.correct_answer is False

6. If else statement to display a message after checking the answer. This line will be run when the while loop is broken, either by the player giving up, or he has satisfied the number of countries he has to name. If self.pass_test is True self.display_passed() is returned. Else self.display_fail() is returned.

def display_alphabet(self) function indicates how size and font at which self.alphabets will be displayed

def ask_player(self) function asks the player to input a country name. If player gave a wrong answer previously, "incorrect" prompt will be displayed. Returns the input of the player.

def display_board(self) function clears screen and indicates font and size at which question is displayed on turtle

def display_passed(self) function indicates a message to be displayed when the player passes the level. Player is prompted to press enter to continue.

def display_fail(self) function indicates a message to be displayed when the player fails the level. Player is prompted to press enter to continue.