Appendix A (from Analysis section [page 3 - 29])

A1 - Trace table of merge sort [page 9]:

array = [2,4,6,5,3,1,7]

| **m** | **i** | **j** | **k** | **array1** | **array2** | **array** |
| --- | --- | --- | --- | --- | --- | --- |
| 4 | 0 | 0 | 0 | [2,4,6,5] | [3,1,7] |  |
|  | 1 | 0 | 1 |  |  | [2] |
|  | 1 | 1 | 2 |  |  | [2,3] |
|  | 1 | 2 | 3 |  |  | [1,2,3] |
|  | 2 | 2 | 4 |  |  | [1,2,3,4] |
|  | 3 | 2 | 5 |  |  | [1,2,3,4,6] |
|  |  |  | 6 |  |  | [1,2,3,4,5,6] |
|  |  |  | 7 |  |  | [1,2,3,4,5,6,7] |

A2 - Final version of core objectives [page 15]:

1. **An algorithm to generate the maze**
   1. This is Prim’s algorithm for maze generation.
   2. Other algorithms were considered but this was decided upon based on my coding skills and also its simplicity.
   3. It was also approved by my end-user (interview on page 10-11).
2. **Maths questions that can be stored in an array**
   1. There are 5 categories of questions - GCSE, Easy, Medium, Hard and Ninson’s Specials.
   2. These are based on the topics that the end-user gave.
   3. These are stored in a file (one for each category) then loaded into an array when the game runs.
   4. This is an important aspect of the game as it adds creativity to the concept of the game and can also be used to improve a user’s maths skills.
   5. It also makes the project unique as not many maze games incorporate the element of having maths questions that the user can answer.
3. **Ability to move character around using keyboard**
   1. This is a key idea as many of the best games use arrow keys to move around a character - without this the game wouldn’t be able to function.
   2. Alternate key binds such as WASD are used as well.
4. **Can select answers to maths questions**
   1. If the user isn’t able to answer the maths questions then the game won’t be able to function.
   2. There is a choice of 4 answers and each answer has its own area on the screen that you can click.
5. **Interactive GUI**
   1. It is essential for the user to have a fun time while playing the game and a command-line interface wouldn’t suit this.
   2. 2 possible graphic interface modules I did use are Pygame and Tkinter.
   3. I did use a combination of these 2 as it is easier to incorporate buttons on Tkinter but it is easier to generate a maze with a character that can traverse through it in Pygame.
6. **Scoring system**
   1. Merge sort algorithm has been run on the array to sort out the high scores in descending order.
   2. The score is decided based on 4 scores added together (MAZE\_CONSTANT is 25).
   3. The first of these is a completion bonus which is always 500.
   4. The second of these is a moves bonus which is calculated by “1500 - (MAZE\_CONSTANT \* moves)”.
   5. The third of these is a lives bonus which is calculated by “lives \* MAZE\_CONSTANT”.
   6. The final of these is a question bonus which is calculated by “QuestionScore \* MAZE\_CONSTANT”.
   7. The details for this can be revealed on the ‘High Scores’ window which is a Tkinter window.
7. **File handling to store high scores**
   1. A file is used to store the high scores of players who have beaten the game before.
   2. The file is opened in append mode and the user score added to the document when the user beats the game and scores a score that is high enough for the top 10.
   3. A high scores dictionary and high scores array are used to store the scores with the dictionary also storing the name with the score.
8. **Limited number of moves**
   1. You start off with 5 moves at the start of the game.
   2. Moves are used to move the player one block (10 pixels) at a time.
   3. The user isn’t able to move if they have 0 moves left and a question displays thereafter for a chance to gain moves.
   4. This is so the user cannot complete the maze without answering at least one maths question.
9. **Moves given after answering question correct**
   1. One move is given if the question is not answered correctly and five moves are given if the question is answered correctly.
10. **Set amount of lives**
    1. There are a certain amount of lives per level.
    2. There are 3 lives
    3. Lives are lost if the user crashes into a wall 5 times and the user is reset every time they crash.
    4. The game will end when the user loses all lives.
    5. A lose screen displays if this happens.
11. **Ability to undo moves**
    1. A stack is used to store up to 5 previous moves that the user has done.
    2. A button can be clicked to go back a move.
    3. The coordinates of the move are stored and the player moves to those coordinates.
    4. Coordinates are taken every time the user moves.
12. **Ability to save and load games**
    1. This is useful as the user is able to close the game and pick back up from where they left off.
    2. A load screen and save screen are displayed when the user clicks the button to load and save the game.
13. **Able to select difficulty of maths questions**
    1. This is a menu choice at the start of the game and allows the user to choose how hard they want the maths questions to be.
14. **Hints given with correct solution**
    1. The user has an option to buy a hint.
    2. This shows them part of the correct solution to the maze.
    3. Recursive solving is used to find the correct path and store it in an array.

A3 - Final version of advanced objectives [page 15]:

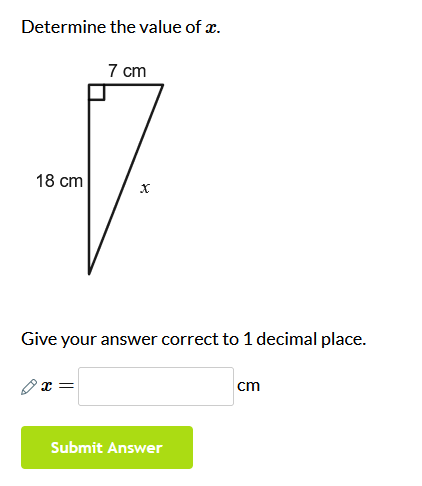
NOTE: None of the advanced objectives have been completed because of time constraints

1. **Aesthetics of maze and player**
   1. The aesthetics of the maze and player do not matter as much until the concept of the game, maze generation, scoring system etc have been considered and completed.
   2. The images used for the project are small and have been decided depending on the theme of the overall game which was pirates. The image for the character was used but for the maze no image was used.
2. **Background image**
   1. This also depends on the theme of the overall game.
3. **Multiple levels increasing in difficulty**
   1. There wasn’t time, so multiple levels were not considered with the size of the maze increasing with difficulty of each level.
4. **Theme of levels**
   1. Each level could have its own slightly different theme.
5. **Personalised settings**
   1. These could have been accessibility settings and other options which would’ve allowed the user to customise their game experience.
6. **Different game modes**
   1. There could be a time trial mode as well as a classic mode where instead of having a limited amount of moves, the user has a limited amount of time in which to complete the maze and answering questions correctly gives the user more time to complete the maze.
   2. There could also be a mode that only focuses on answering questions (an arcade mode) and doesn’t give bonus points for completing the maze in a small amount of moves.

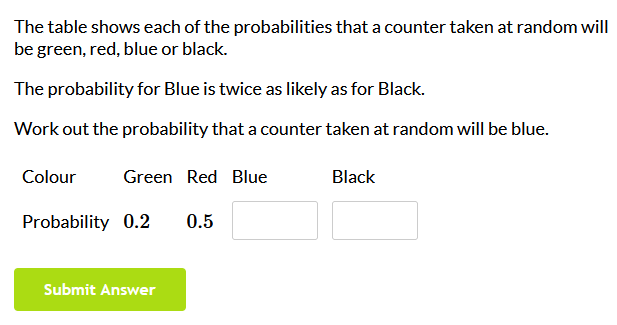
A4 - Question examples (from [Dr Frost Maths](https://www.drfrostmaths.com/)) [page 15]:

**GCSE:**

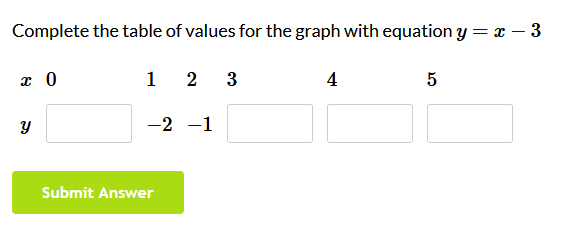
Angles and right-angled triangles



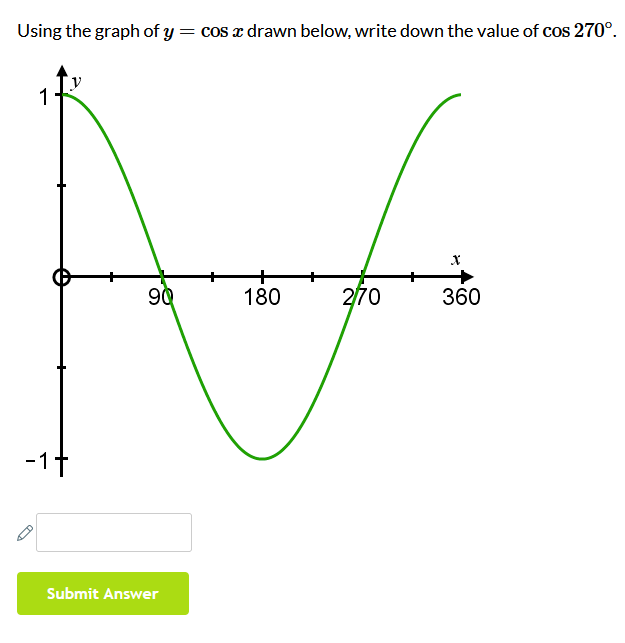
Probability



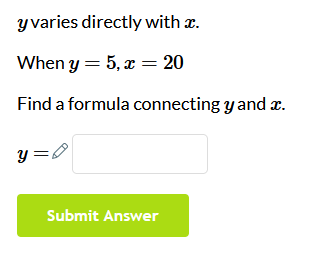
Graphs



Sine and cosine rules

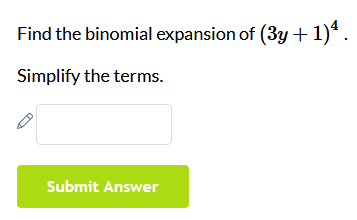


Proportion

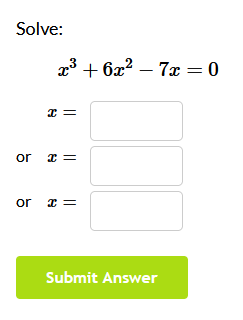


**Easy:**

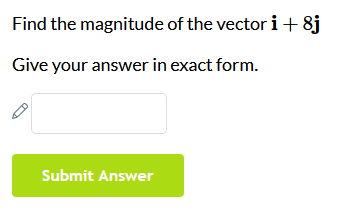
Binomial expansion



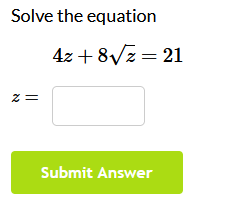
Algebra



Vectors

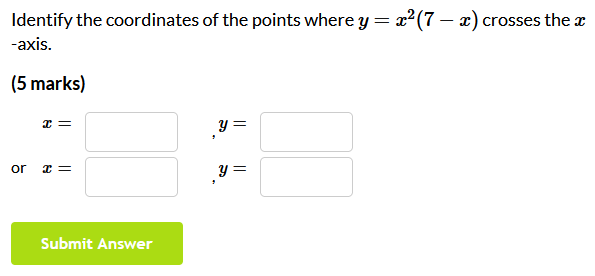


Quadratics

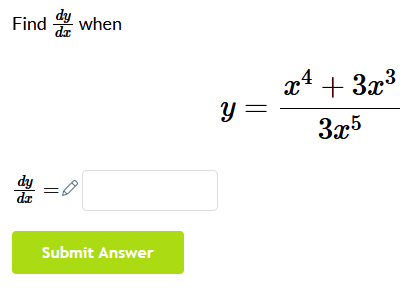


**Medium:**

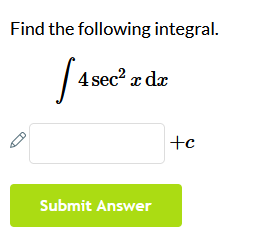
Graphs and transformations



Differentiation

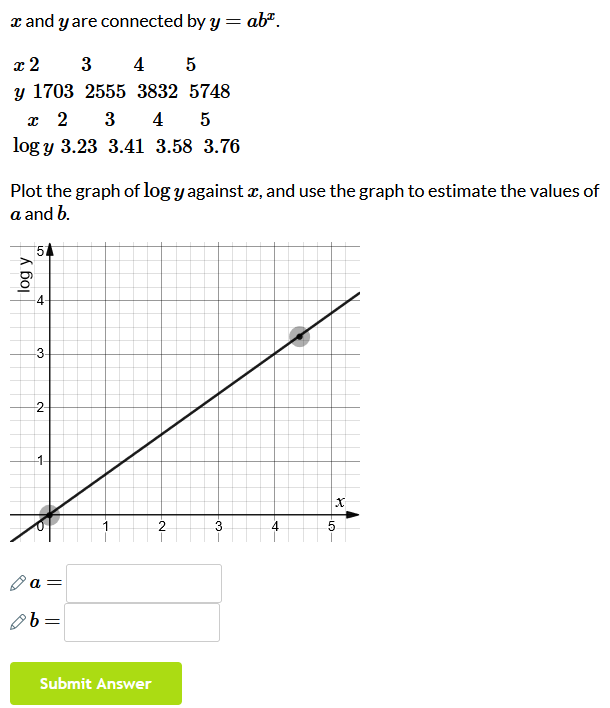


Integration

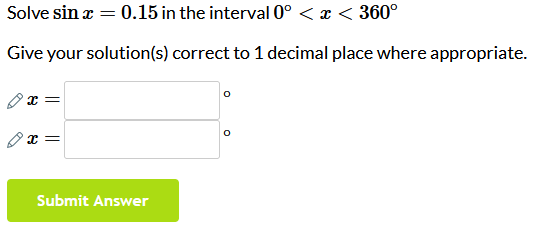


**Hard:**

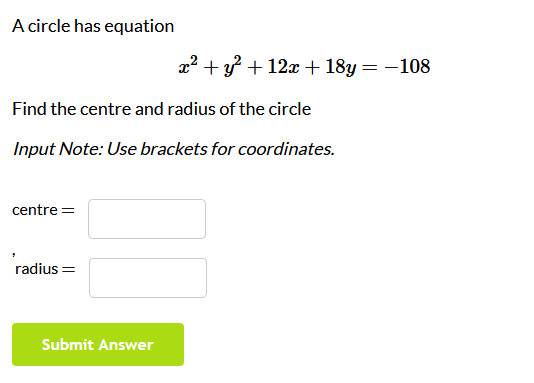
Logarithms



Trigonometry

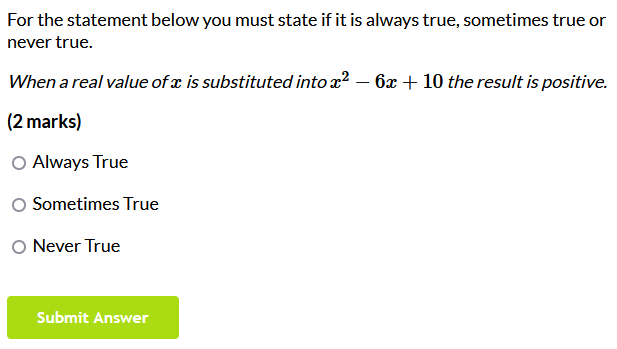


Circles

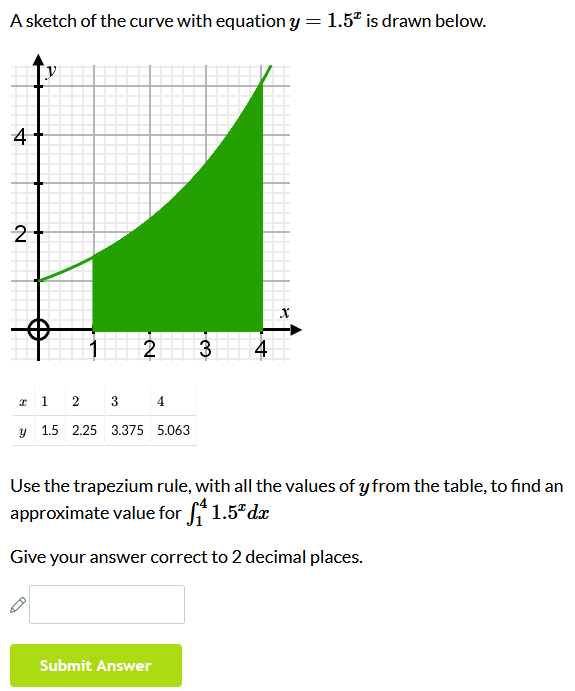


**Ninson’s specials:**

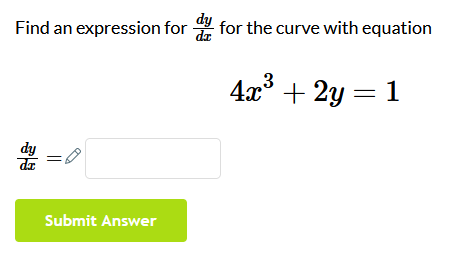
Mathematical proofs



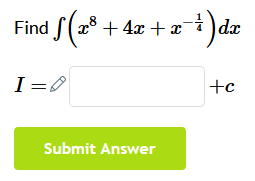
Problems involving areas between curves and lines



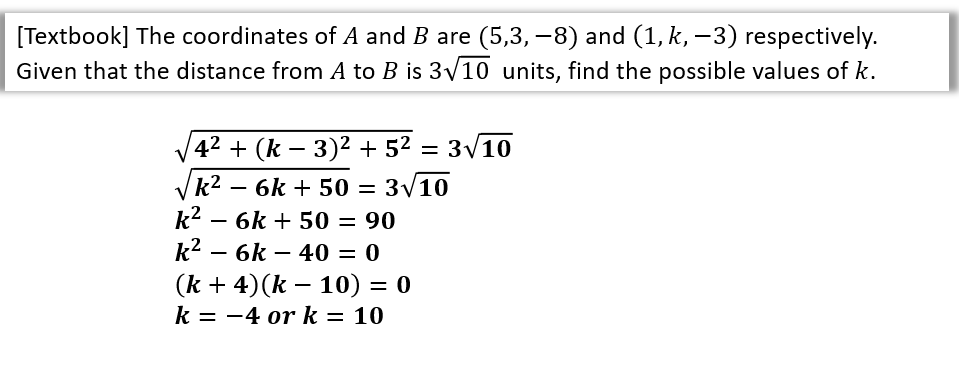
Implicit differentiation



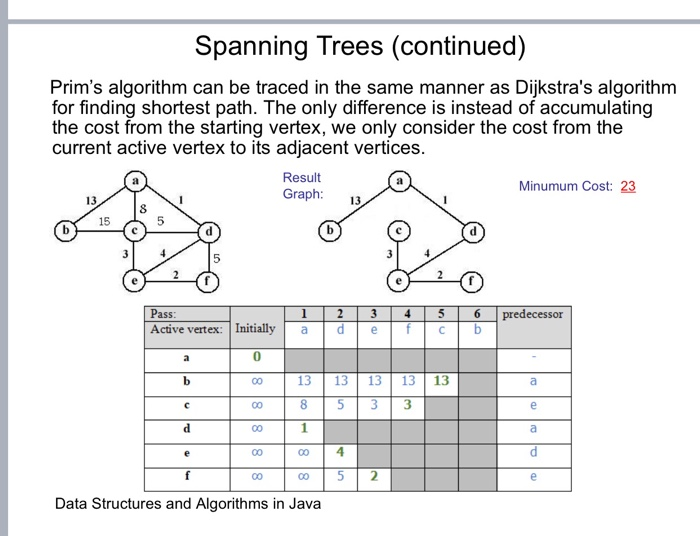
Integration



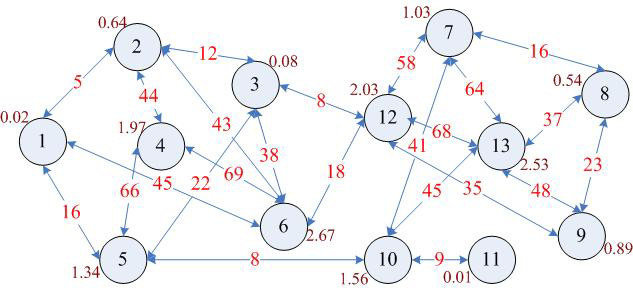
Vectors in 3D problems



A5 - Trace table for Prim’s algorithm on example graph [page 16]:



The GRAPH:



|  | Node | | | | | | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Vertex | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
| 1 | 0 |  |  |  |  |  |  |  |  |  |  |  |  |
| 2 | **5** |  | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 |
| 5 | 16 | ∞ | 22 | **66** |  | ∞ | ∞ | ∞ | ∞ | **8** |  |  |  |
| 6 | 45 | 43 | 38 | 69 | ∞ |  | ∞ | ∞ | ∞ | ∞ | ∞ | 18 |  |
| 3 | ∞ | **12** |  | ∞ | 43 | 38 | ∞ | ∞ | ∞ | ∞ | ∞ | **8** |  |
| 4 | ∞ | 44 | ∞ |  | 66 | 69 |  |  |  |  |  |  |  |
| 10 | ∞ | ∞ | ∞ | ∞ | **8** | ∞ | 41 | ∞ | ∞ |  | **9** | ∞ | 45 |
| 12 | ∞ | ∞ | **8** | ∞ | ∞ | **18** | 58 | ∞ | 35 | ∞ | ∞ |  | 68 |
| 11 | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | 9 |  |  |  |
| 7 | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |  | **16** | ∞ | 41 | ∞ | 58 | 64 |
| 8 | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | **16** |  | **23** | ∞ | ∞ | ∞ | **37** |
| 9 | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | 23 |  | ∞ | ∞ | 35 | 48 |
| 13 | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | 64 | 37 | 48 | 45 | ∞ | 68 |  |

A6 - More info on Recursive solving algorithm [page 18]:

Pusedocode was converted from Java which was found on wikipedia

<https://en.wikipedia.org/wiki/Maze-solving_algorithm>

A **maze-solving algorithm** is an [automated method](https://en.wikipedia.org/wiki/Algorithm) for solving a [maze](https://en.wikipedia.org/wiki/Maze). The random mouse, wall follower, Pledge, and Trémaux's [algorithms](https://en.wikipedia.org/wiki/Algorithm) are designed to be used inside the maze by a traveller with no prior knowledge of the maze, whereas the [dead-end](https://en.wikipedia.org/wiki/Cul-de-sac) filling and [shortest path algorithms](https://en.wikipedia.org/wiki/Shortest_path_algorithm) are designed to be used by a person or computer program that can see the whole maze at once.

Mazes containing no loops are known as "simply connected", or "perfect" mazes, and are equivalent to a [*tree*](https://en.wikipedia.org/wiki/Tree_(graph_theory)) in graph theory. Maze-solving algorithms are closely related to [graph theory](https://en.wikipedia.org/wiki/Graph_theory). Intuitively, if one pulled and stretched out the paths in the maze in the proper way, the result could be made to resemble a tree.

If given an omniscient view of the maze, a simple recursive algorithm can tell one how to get to the end. The algorithm will be given a starting X and Y value. If the X and Y values are not on a wall, the method will call itself with all adjacent X and Y values, making sure that it did not already use those X and Y values before. If the X and Y values are those of the end location, it will save all the previous instances of the method as the correct path.

This is in effect a depth-first search expressed in terms of grid points. The omniscient view prevents entering loops by memorization.

Appendix B (from Documented Design section [page 30 - ])

B0 - Pseudocode syntax/format [page 59]:

KEYWORDS (Highlighted in Yellow in python) are written in CAPITALS, e.g. FOR, IF, AND





FOR, IF, AND, NOT, WHILE, OR, FUNCTION, ELIF, ELSE

Module (library) names (and when used) are written in **Bold** and *Italics* e.g. ***sleep()***

Booleans and some operations are written in **Bold** e.g. **True**, **False**, **Append**, **Add**

Other *keywords* (Highlighted in Purple in python) are written in *Italics* e.g *Int*, *Str*, *Float, Print*

**



*Int*, *Str*, *Float*

The main syntax of the pseudocode is AQA standard and the following websites have been used to refer to it throughout the code (also mentioned in References section)

[Pseudocode — Isaac Computer Science](https://isaaccomputerscience.org/concepts/prog_pas_pseudocode?examBoard=all&stage=all)

[Notes and guidance: Pseudo-code (AQA)](https://filestore.aqa.org.uk/resources/computing/AQA-8525-NG-PC.PDF)

[Pseudocode conventions — Isaac Computer Science](https://isaaccomputerscience.org/concepts/isaac_pseudocode?examBoard=all&stage=all)

B1 - Pseudocode for the Player class [page 60]:

IMPORT ***pygame***

from ***pygame***.locals IMPORT \*

CLASS Player inherits sprite

PRIVATE surf

PRIVATE rect

PUBLIC PROCEDURE new(image\_file, location)

SUPER.Sprite()

surf ← pygame.image.load(image\_file).convert()

surf.set\_colorkey ((0,0,255), RLEACCEL)

rect ← surf.get\_rect(center=(location))

ENDPROCEDURE

PUBLIC PROCEDURE move(pressed\_keys)

IF pressed\_keys[K\_UP] OR pressed\_keys[K\_w] THEN

Player.rect.move\_ip(0, -10)

ENDIF

IF pressed\_keys[K\_DOWN] OR pressed\_keys[K\_s] THEN

Player.rect.move\_ip(0, 10)

ENDIF

IF pressed\_keys[K\_LEFT] OR pressed\_keys[K\_a] THEN

Player.rect.move\_ip(-10, 0)

ENDIF

IF pressed\_keys[K\_RIGHT] OR pressed\_keys[K\_d] THEN

Player.rect.move\_ip(10, 0)

ENDIF

IF Player.rect.left < 0 THEN

Player.rect.left ← 0

ENDIF

IF Player.rect.right > 1900 THEN

Player.rect.right ← 1900

ENDIF

IF Player.rect.top <= 0 THEN

Player.rect.top ← 0

ENDIF

IF Player.rect.bottom. >= 1000 THEN

Player.rect.bottom ← 1000

ENDIF

ENDPROCEDURE

ENDCLASS

B2 - Pseudocode for the Maze\_objects:

IMPORT ***pygame***

CLASS Wall *inherits* sprite

PUBLIC surf

PUBLIC rect

PUBLIC PROCEDURE new(location)

super.Sprite()

surf ← ***pygame***.Surface((25, 25))

surf.*fill*((255,255,255))

rect ← surf.get\_rect(center=(location))

ENDPROCEDURE

ENDCLASS

CLASS Cell *inherits* sprite

PUBLIC surf

PUBLIC rect

PRIVATE rectLocation

PUBLIC PROCEDURE new(location)

SUPER.Sprite()

surf ← ***pygame***.Surface((25, 25))

surf.*fill*((0,0,255))

rectLocation ← location

rect ← surf.get\_rect(center=(rectLocation))

ENDPROCEDURE

PUBLIC METHOD getLocation

RETURN rectLocation

ENDMETHOD

ENDCLASS

CLASS Empty *inherits* sprite

PUBLIC surf

PUBLIC rect

PUBLIC PROCEDURE new(location)

SUPER.Sprite()

surf ← ***pygame***.Surface((25, 25))

surf.*fill*((0,0,0))

rect ← surf.get\_rect(center=(location))

ENDPROCEDURE

ENDCLASS

CLASS Key *inherits* sprite

PUBLIC surf

PUBLIC rect

PUBLIC PROCEDURE new(location)

SUPER.Sprite()

surf ← ***pygame***.Surface((25, 25))

surf.*fill*((0,0,0))

rect ← surf.get\_rect(center=(location))

ENDPROCEDURE

ENDCLASS

B3 - Pseudocode for the Information\_objects:

IMPORT ***pygame***

CLASS Scoreboard *inherits* sprite

PUBLIC CurrentScore

PUBLIC score

PUBLIC font

PUBLIC textSurf

PUBLIC surf

PUBLIC rect

PUBLIC W

PUBLIC H

PUBLIC PROCEDURE new(size, color, width, height, locationX, locationY)

CurrentScore ← 0

score ← "Score: "+*str*(CurrentScore)

SUPER.Sprite()

font ← ***pygame***.font.SysFont("Arial", size)

textSurf ← self.font.*render*(score, 1, color)

surf ← ***pygame***.Surface((width, height))

surf.*fill*((0,0,255))

rect ← surf.get\_rect(center=(locationX,locationY))

W ← textSurf.get\_width()

H ← textSurf.get\_height()

surf.blit(textSurf, [width/2 - W/2, height/2 - H/2])

ENDPROCEDURE

PUBLIC METHOD update(CurrentScore, size, color, width, height)

score ← "Score: "+*str*(CurrentScore)

font ← ***pygame***.font.SysFont("Arial", size)

textSurf ← self.font.*render*(score, 1, color)

surf ← ***pygame***.Surface((width, height))

surf.fill((0,0,255))

W ← textSurf.get\_width()

H ← textSurf.get\_height()

surf.blit(textSurf, [width/2 - W/2, height/2 - H/2])

ENDMETHOD

ENDCLASS

CLASS LivesC *inherits* sprite

PUBLIC CurrentScore

PUBLIC score

PUBLIC font

PUBLIC textSurf

PUBLIC surf

PUBLIC rect

PUBLIC W

PUBLIC H

PUBLIC PROCEDURE new(size, color, width, height, locationX, locationY)

Lives ← 3

lives ← "Lives: "+*str*(Lives)

SUPER.Sprite()

font ← ***pygame***.font.SysFont("Arial", size)

textSurf ← self.font.*render*(score, 1, color)

surf ← ***pygame***.Surface((width, height))

surf.*fill*((0,0,255))

rect ← surf.get\_rect(center=(locationX,locationY))

W ← textSurf.get\_width()

H ← textSurf.get\_height()

surf.blit(textSurf, [width/2 - W/2, height/2 - H/2])

ENDPROCEDURE

PUBLIC METHOD update(Lives, size, color, width, height)

Lives ← Lives - 1

lives ← "Lives: "+*str*(Lives)

font ← ***pygame***.font.SysFont("Arial", size)

textSurf ← self.font.*render*(score, 1, color)

surf ← ***pygame***.Surface((width, height))

surf.fill((0,0,255))

W ← textSurf.get\_width()

H ← textSurf.get\_height()

surf.blit(textSurf, [width/2 - W/2, height/2 - H/2])

RETURN Lives

ENDMETHOD

PUBLIC METHOD set(Lives, size, color, width, height)

lives ← "Lives: "+*str*(Lives)

font ← ***pygame***.font.SysFont("Arial", size)

textSurf ← self.font.*render*(score, 1, color)

surf ← ***pygame***.Surface((width, height))

surf.fill((0,0,255))

W ← textSurf.get\_width()

H ← textSurf.get\_height()

surf.blit(textSurf, [width/2 - W/2, height/2 - H/2])

ENDMETHOD

CLASS MovesC *inherits* sprite

PUBLIC CurrentScore

PUBLIC score

PUBLIC font

PUBLIC textSurf

PUBLIC surf

PUBLIC rect

PUBLIC W

PUBLIC H

PUBLIC PROCEDURE new(size, color, width, height, locationX, locationY)

Moves ← 5

moves ← "Moves Left: "+*str*(Moves)

SUPER.Sprite()

font ← ***pygame***.font.SysFont("Arial", size)

textSurf ← self.font.*render*(score, 1, color)

surf ← ***pygame***.Surface((width, height))

surf.*fill*((0,0,255))

rect ← surf.get\_rect(center=(locationX,locationY))

W ← textSurf.get\_width()

H ← textSurf.get\_height()

surf.blit(textSurf, [width/2 - W/2, height/2 - H/2])

ENDPROCEDURE

PUBLIC METHOD update(Moves, size, color, width, height)

moves ← "Moves Left: "+str(Moves)

font ← ***pygame***.font.SysFont("Arial", size)

textSurf ← self.font.*render*(score, 1, color)

surf ← ***pygame***.Surface((width, height))

surf.fill((0,0,255))

W ← textSurf.get\_width()

H ← textSurf.get\_height()

surf.blit(textSurf, [width/2 - W/2, height/2 - H/2])

ENDMETHOD

ENDCLASS

CLASS Timer *inherits* sprite

PUBLIC CurrentScore

PUBLIC score

PUBLIC font

PUBLIC textSurf

PUBLIC surf

PUBLIC rect

PUBLIC W

PUBLIC H

PUBLIC PROCEDURE new(size, color, width, height, locationX, locationY)

Time ← 0

time ← "Time: "+*str*(Time)

SUPER.Sprite()

font ← ***pygame***.font.SysFont("Arial", size)

textSurf ← self.font.*render*(score, 1, color)

surf ← ***pygame***.Surface((width, height))

surf.*fill*((0,0,255))

rect ← surf.get\_rect(center=(locationX,locationY))

W ← textSurf.get\_width()

H ← textSurf.get\_height()

surf.blit(textSurf, [width/2 - W/2, height/2 - H/2])

ENDPROCEDURE

PUBLIC METHOD update(Time, size, color, width, height)

time ← "Time: "+*str*(Time)

font ← ***pygame***.font.SysFont("Arial", size)

textSurf ← self.font.*render*(score, 1, color)

surf ← ***pygame***.Surface((width, height))

surf.fill((0,0,255))

W ← textSurf.get\_width()

H ← textSurf.get\_height()

surf.blit(textSurf, [width/2 - W/2, height/2 - H/2])

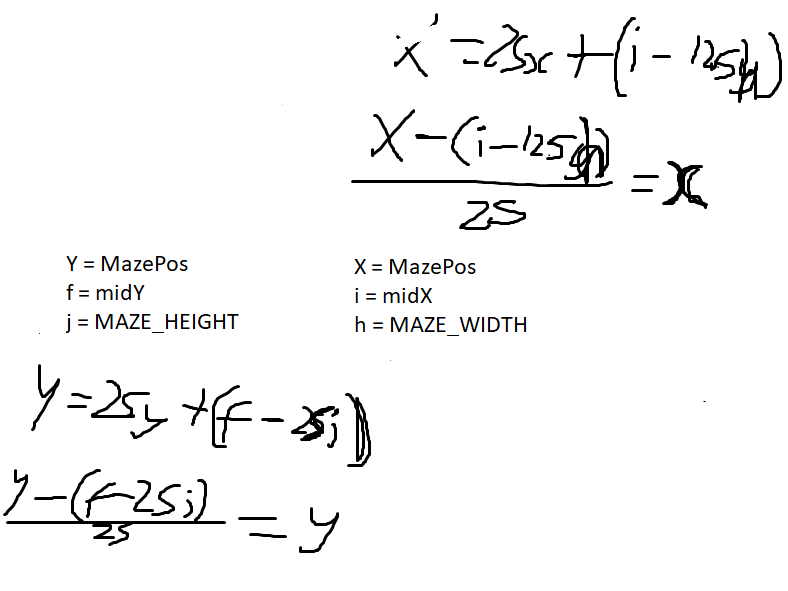
RETURN Time

ENDMETHOD

ENDCLASS

B5 - Calculations from Technical solution:

Below is an example of the calculation used to reverse the calculation used to place cells/walls in their positions when generating the maze to get the player position in the maze.

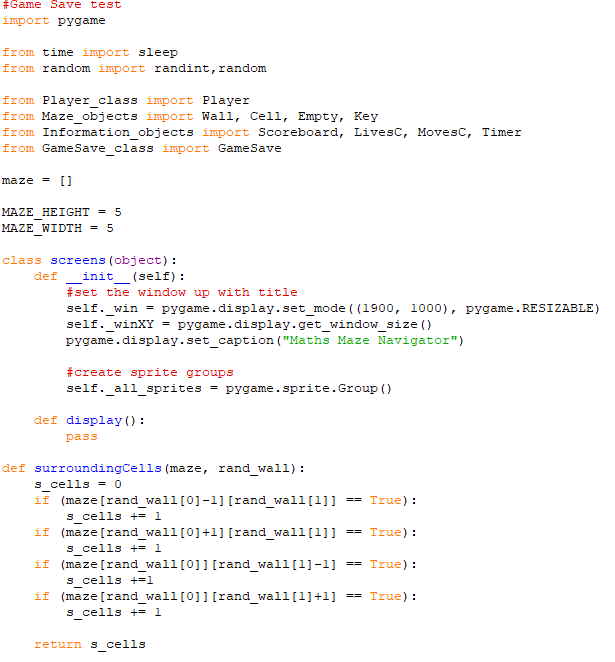


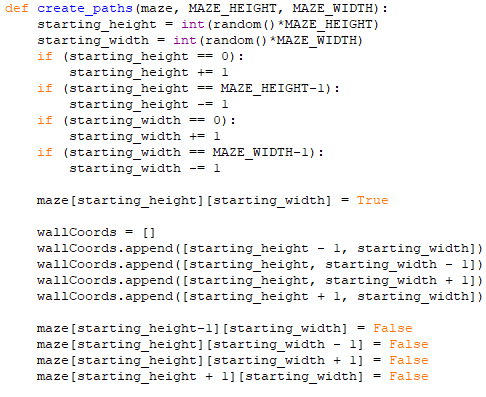
Appendix C (from Technical Solution section [page ])

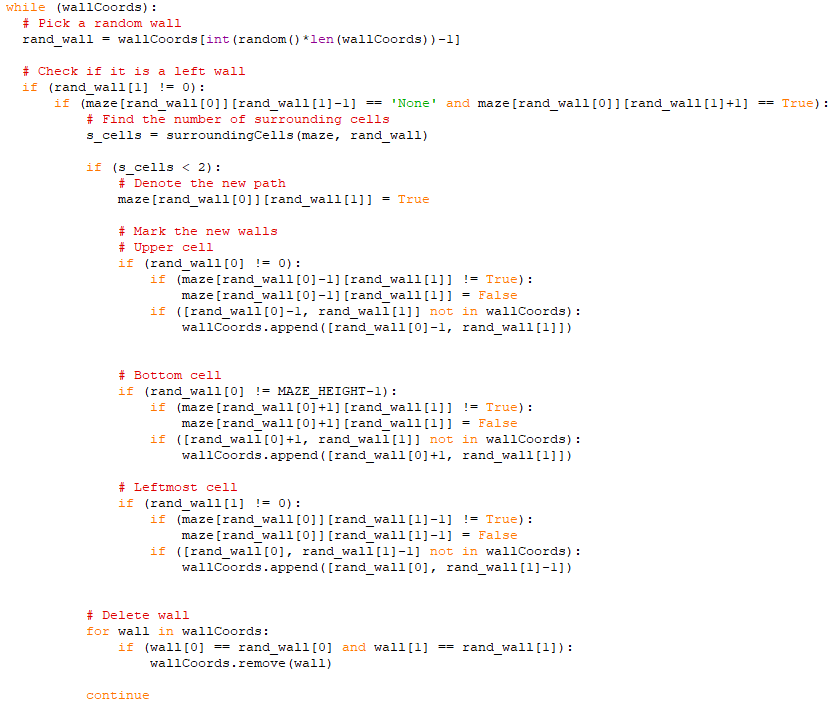
The program will be included in full here with line numbers and comments

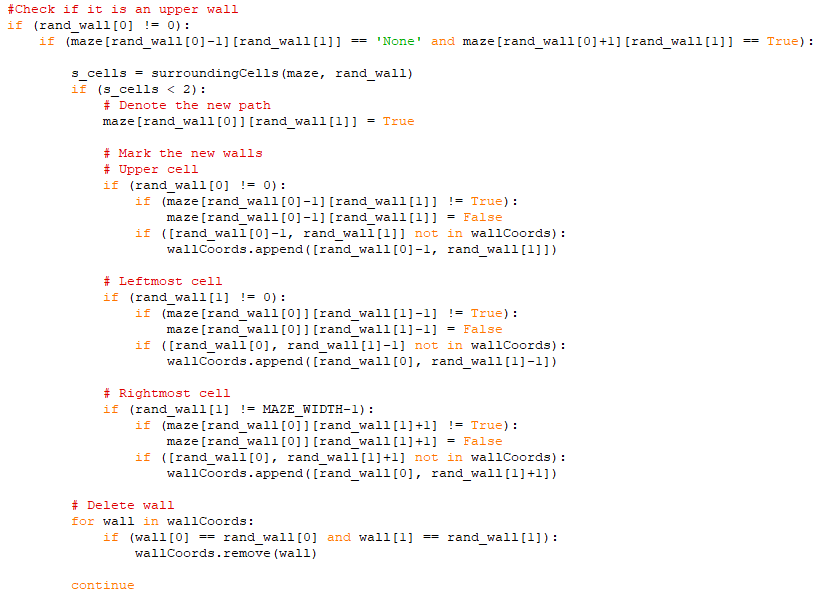
However, before that, extra code which was required for modular testing will be detailed here:

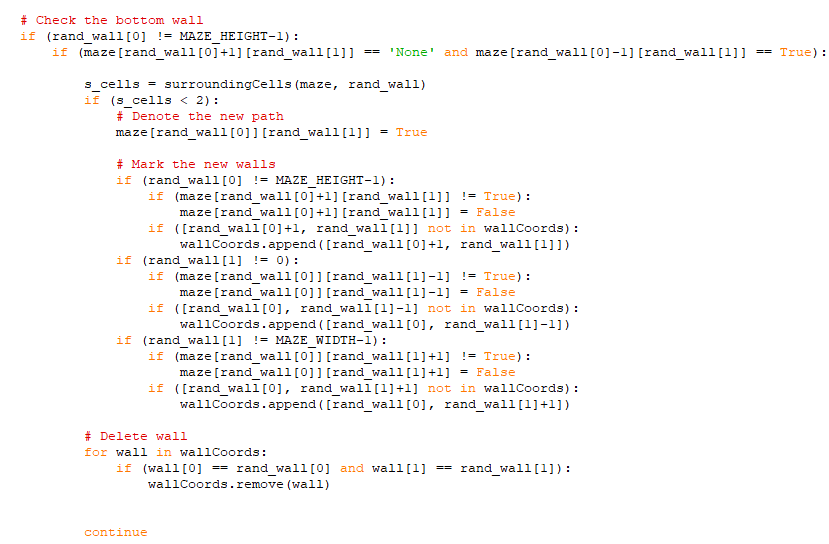
**C1 - GAMESAVE\_TEST:**

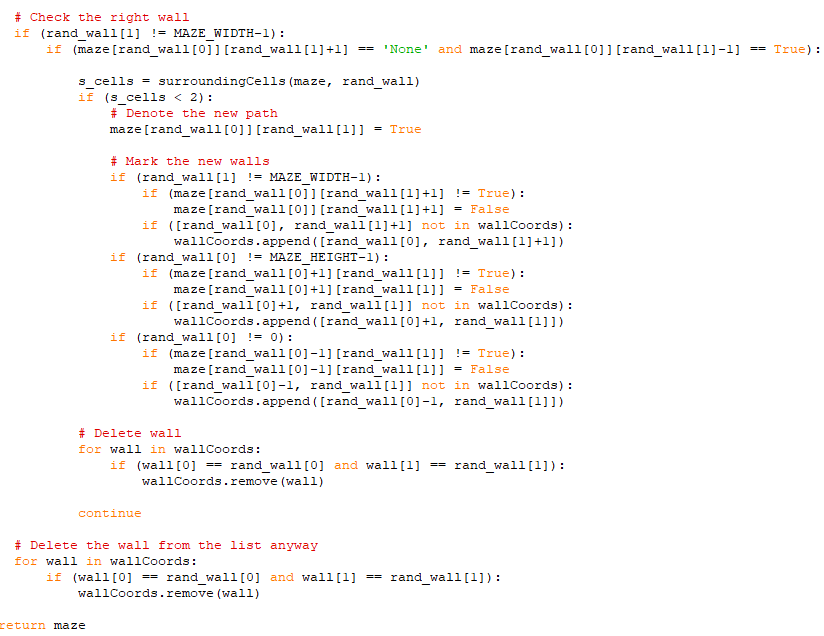
****

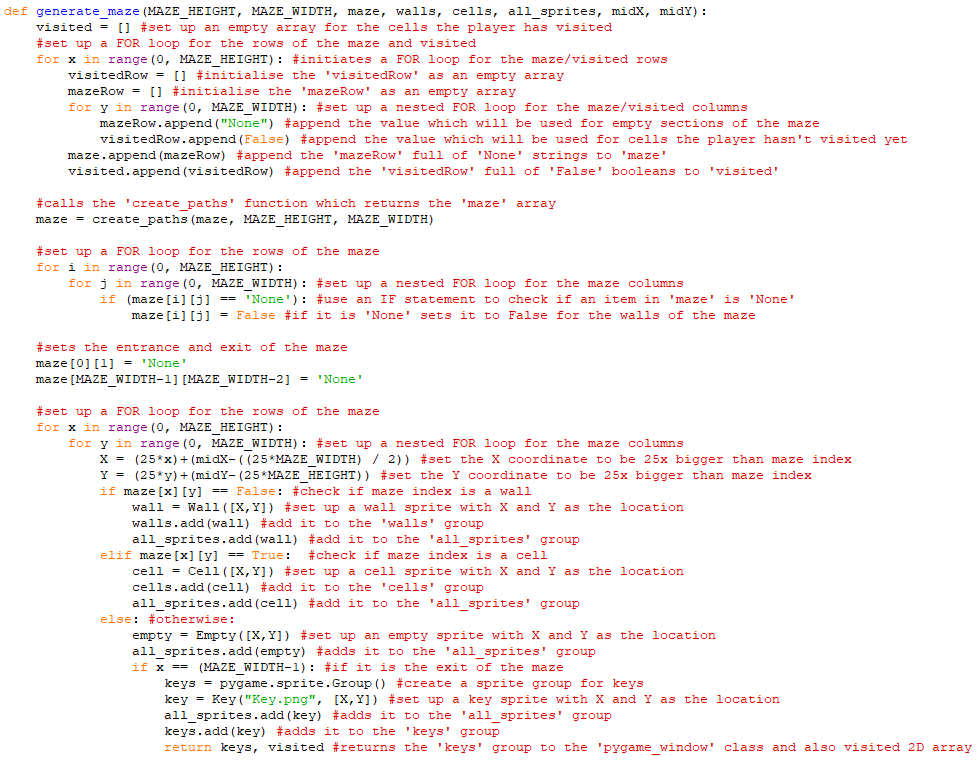
****

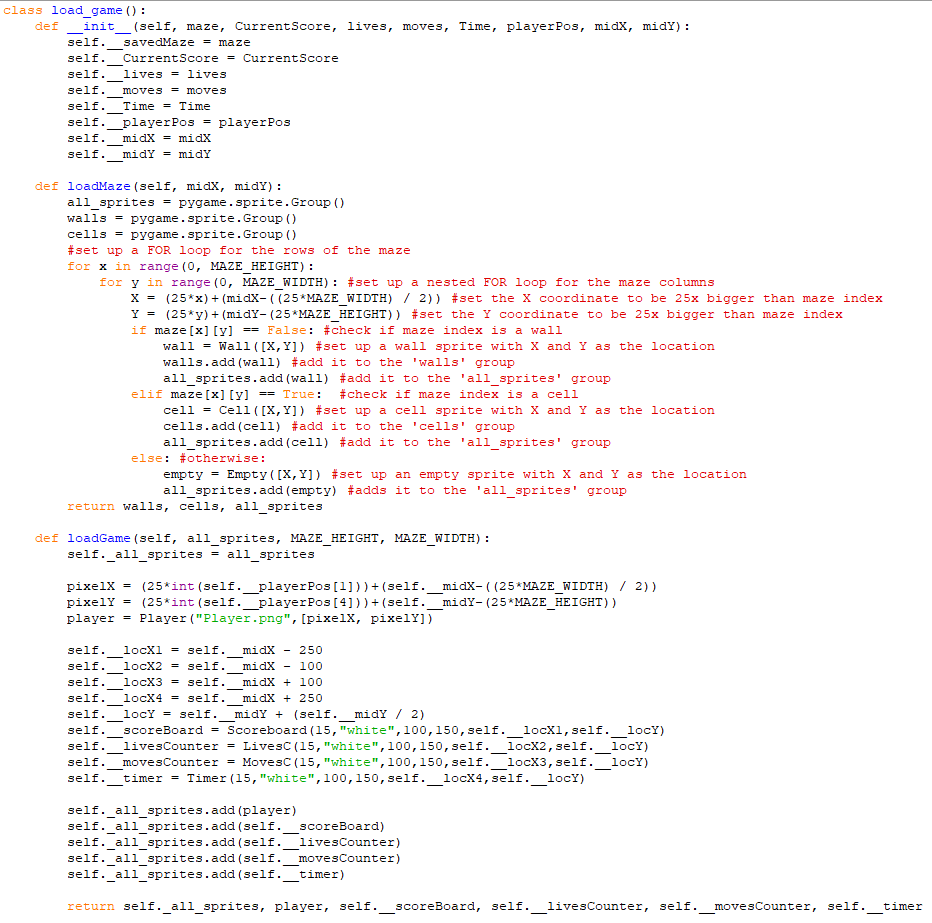
****

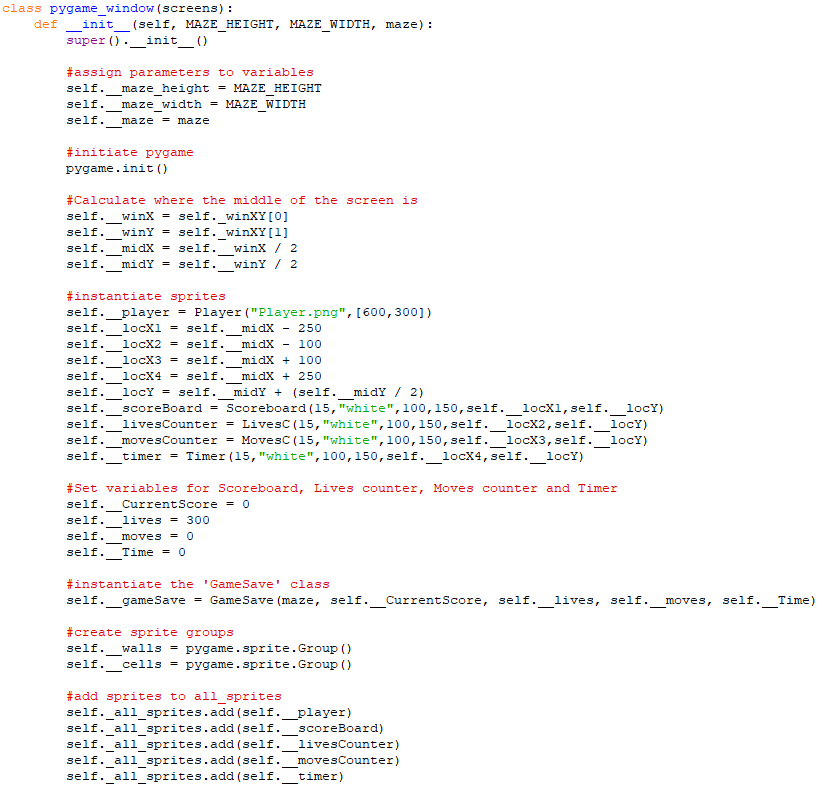
****

****

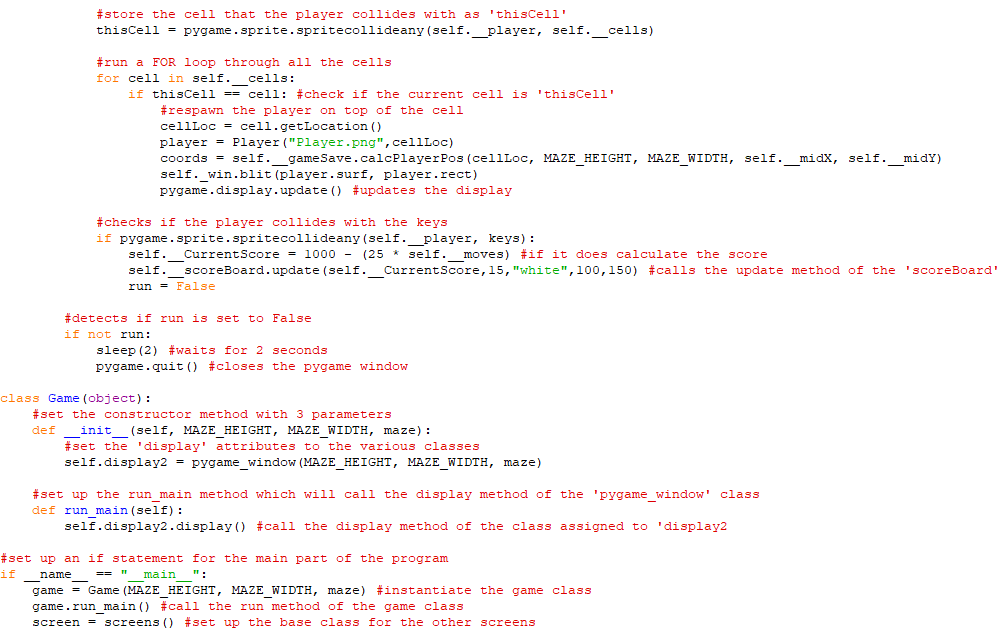
****

****

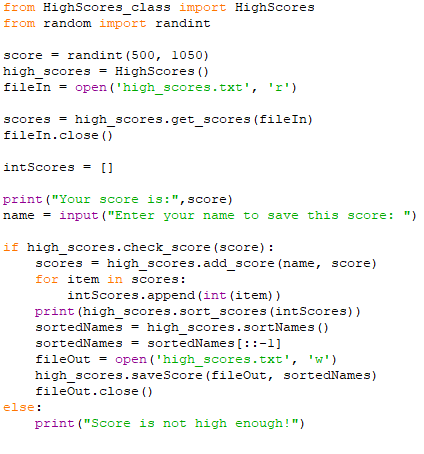
****

****

****

****

**C2 - HIGHSCORES\_TEST:**

****

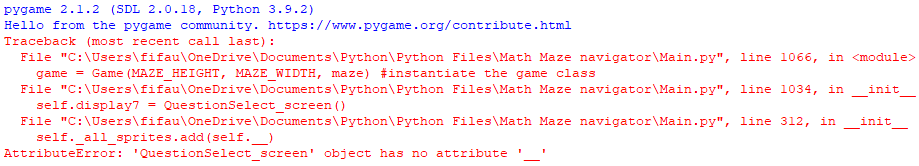
Then the Main code was to be provided in full starting with the Player class

**C3 - PLAYER CLASS:**

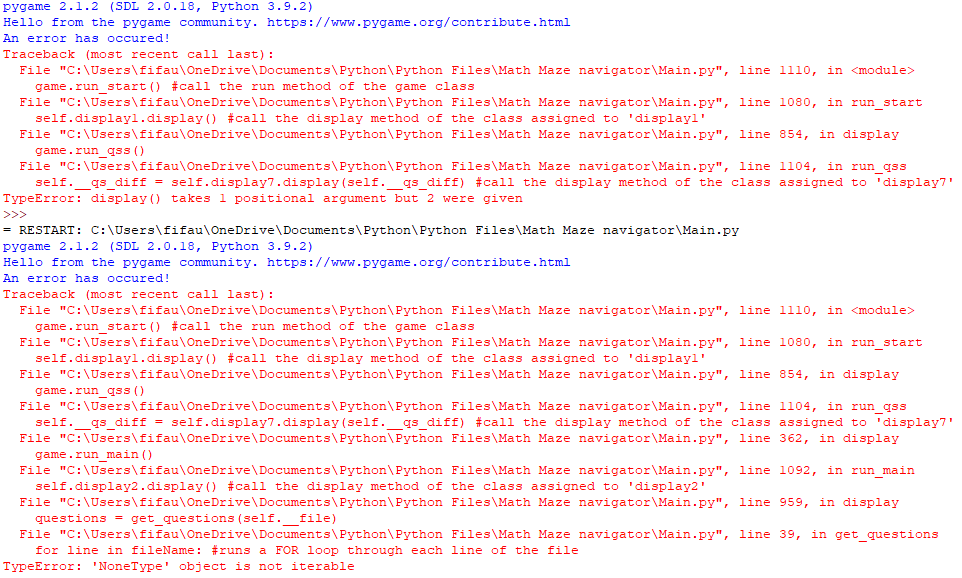
Appendix D (from Testing section [page ])

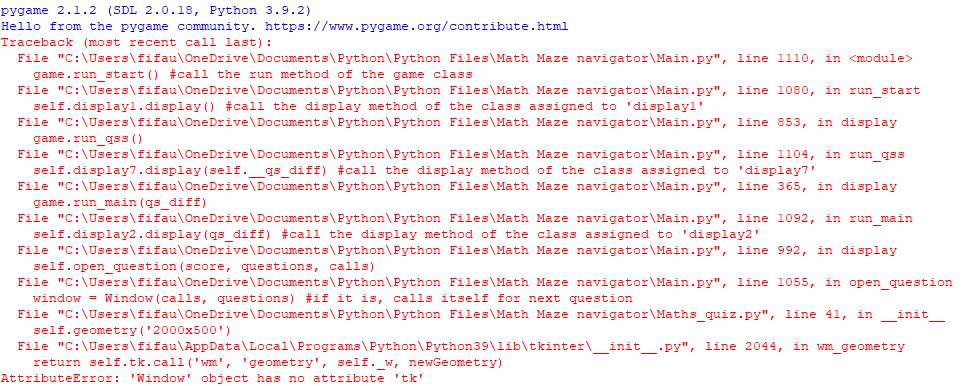
D1 - MODULAR TESTING:

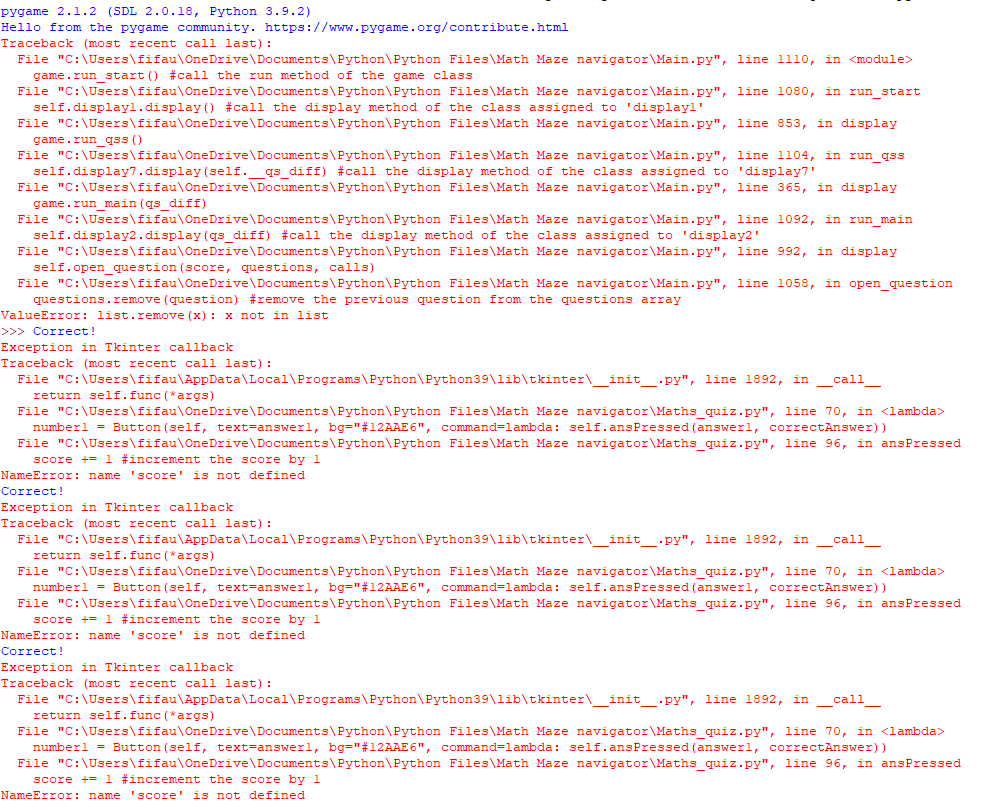
Earlier screenshots were unfortunately not captured. The first screenshot is from Phase 5 of the project.



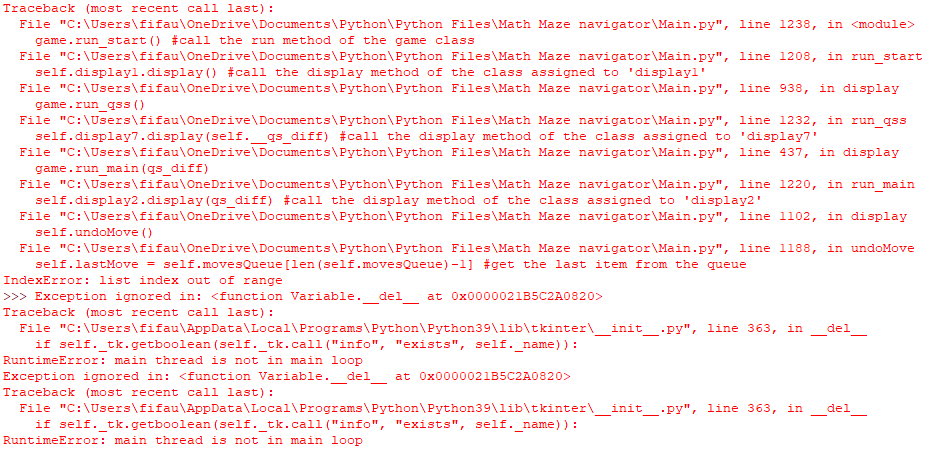


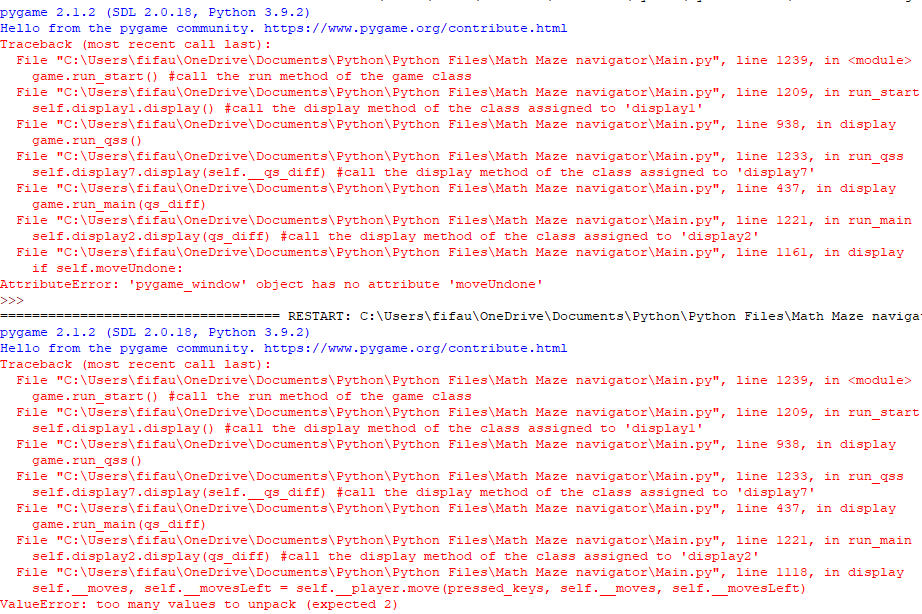


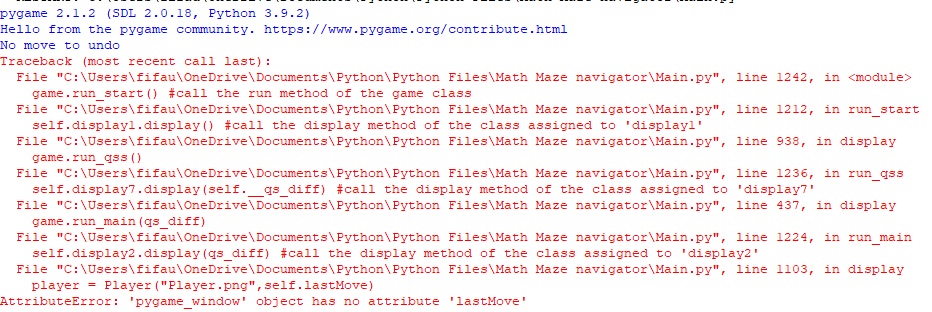
****

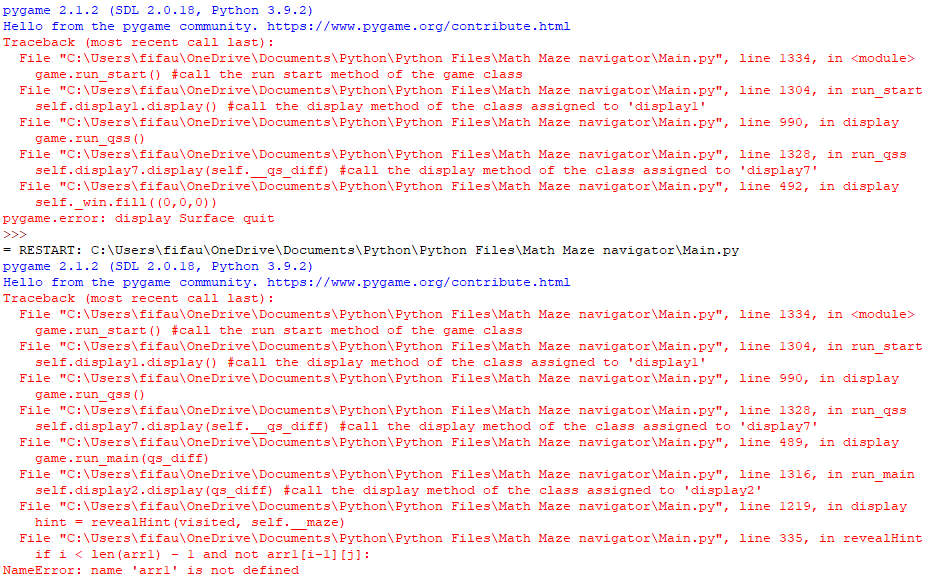
****

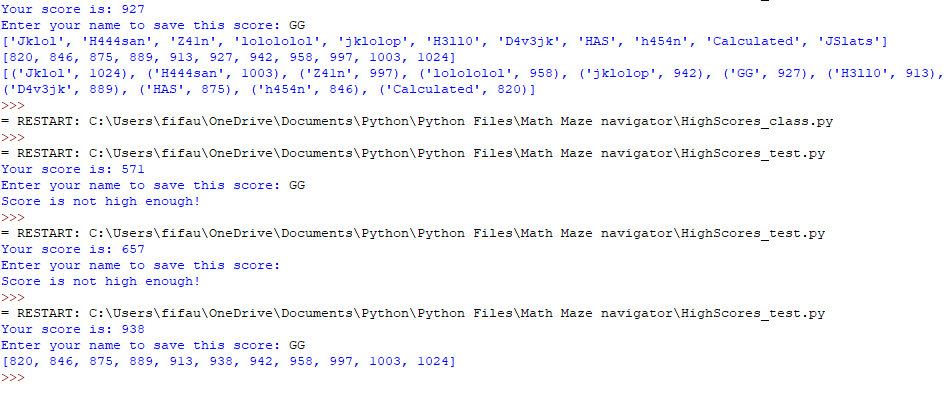
****

****

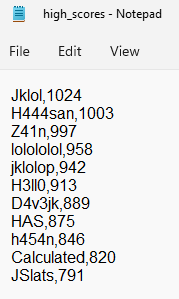
****

****

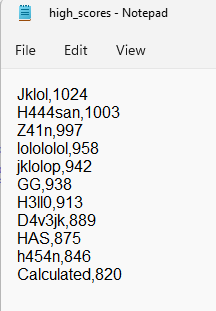
****

**D2 - Testing for High Scores class:**

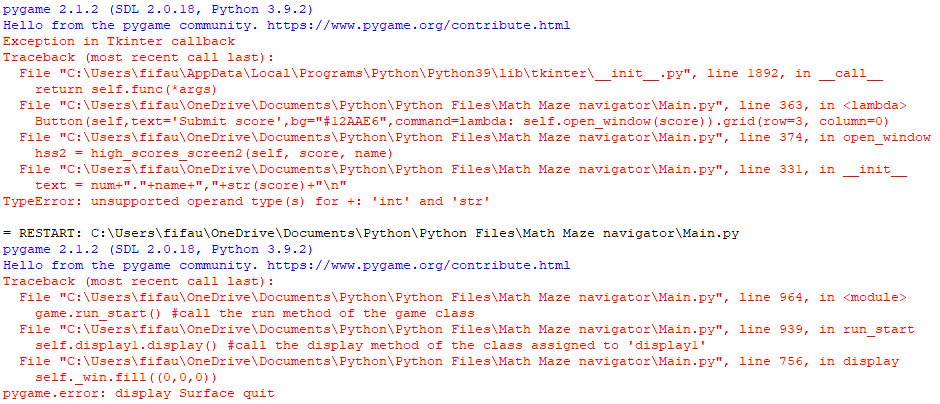
**Before:**

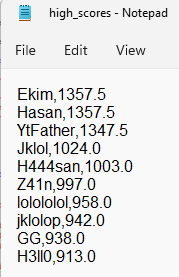
****

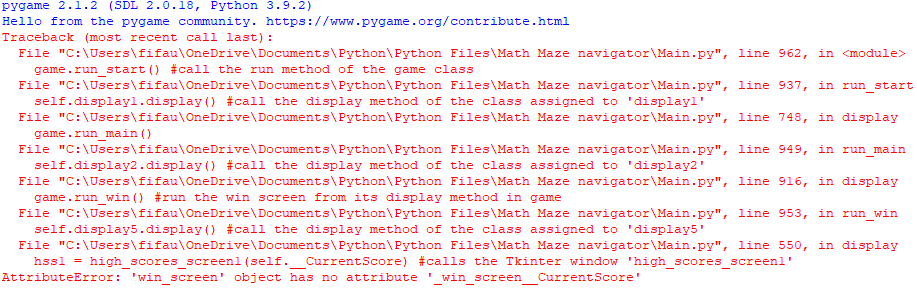
**After:**

****

****

****

****

****