

Rubik's Cube

Component 3

Computer Science Coursework

Name: Samuel James Flegg

Candidate Number: 9098

Centre Number: 37611

Table of Contents

Glossary	3
GitHub.....	3
Analysis.....	3
Problem Identification	3
Computational Approaches	4
Stakeholders	4
Research.....	5
Questionnaire.....	5
Useful technical approaches.....	9
Similar Programs.....	11
Features.....	16
Essential Features.....	16
Limitations	17
Requirements	17
Success Criteria.....	19
Design	20
Problem Decomposition	20
Table	20
Flowchart	23
Solutions	24
Sorting Algorithms	24
Searching Algorithms	26
Cube Algorithms	26
Usability Features.....	28
Data and Variables	29
Validation.....	33
Iterative-Development Test Data	34
Post-Development Test Data	40
Test plan	40
Quiz template	43
Implementation	45
Prototype One	45
Development.....	45
Testing.....	55
Improvements	57

Program Images	58
Prototype 2.....	59
Development.....	59
Testing.....	105
Improvements	110
Program Images	111
Prototype 3.....	113
Development.....	113
Testing.....	176
Improvements	185
Program Images	190
Evaluation	209
Post-development testing.....	209
Stakeholder Responses.....	209
Results	218
Success Criteria	220
Potential Improvements.....	221
Success Criteria	221
Limitations	222
Maintenance	225
Appendix	225
main.py.....	225
validation.py	236
cube.py.....	239
features.py	260
game_data.py	274
interface.py	280
user_data.py	286
tools.py.....	293

Glossary

Cuber(s) – a person (or group of people) who regularly play with a Rubik's Cube

Speed Cuber(s) – a person (or group of people) who enjoy solving a Rubik's Cube as fast as possible

Cubing – Solving, or attempting to solve, a Rubik's cube

GitHub

GitHub is an expansion upon the Git service which allow programs to be shared and collaborated on.

Git is a version history manager that allows you to edit expand upon your code without worry of breaking it, as you can simply rollback to a working version.

Whilst restrictions on school computers have made this difficult, I have tried my best to manage my Rubik's cube project with Git and GitHub, and as such my code can also be found at:

[s-flegg/Rubiks-cube: My A-Level coursework project.](#)

Analysis

Problem Identification

For my coursework project I am going to create a Rubik's Cube game – the core principle of a Rubik's "cube", regardless of if they are a cube or not, is dividing each face of the shape into coloured shapes, with each face having a different colour. The rows and columns of the shape can then be rotated, mixing up the colours. The goal is to return the shape to its original state, with each face being only one colour.

The main focus of my game will be the traditional Rubik's Cube, a 3x3x3 cube. Whilst some examples of this already exist, I believe many of the currently available options have clunky and confusing controls that make it harder to solve the Rubik's Cube. They also tend to be rather barebones – lacking any additional features. As such, my goal for the program will be to provide a highly functional and easy to use program, that will hopefully provide users more enjoyment than other programs.

Computational Approaches

I will be utilising abstraction to simplify the requirements into the barebones of what is required. This will allow for a more functional program with less unnecessary components that could confuse the user and/or make it harder to develop. For example, the first version of my cube will likely be a 2D net, as this can convey all the same information whilst being significantly easier to code. This makes it easier to make the code as problems are easier to understand and have less requirements. However, as I believe some less abstract features are crucial for the program – e.g. being able to show the cube in 3d – this abstraction will be removed in later versions. This will still make it easier to make the less abstract features as I can work up to them and have a reference as I make it.

Decomposition will also be used to significantly reduce the coding requirements. This involves breaking the code required to solve a problem into individual steps, or functions. This will allow for significantly less code as functions can be called multiple times. Additionally, it will make it easier to understand what each bit of code is doing in the case that it is later edited. For example, every possible way to interact with a cube can be achieved with 9 functions: 3 vertical turns, 3 horizontal turns, and 3 ways to rotate the entire cube. These functions can then build to other moves or be called directly.

I may also utilise concurrency. This means having multiple pieces of code running at the same time, for example the main Rubik's cube code and perhaps a save feature that automatically updates in case that the program crashes/ failures. This will allow for things to be processed quicker as multiple things can be done simultaneously.

I will also utilise backtracking, as this will allow me to repeatedly develop and improve my code, from the entire program down to individual functions, whilst ensuring no major errors are made. I may use Git to do this, as this will allow me to easily make backups of my code with comments so that I know what changes between each backup.

Stakeholders

Stakeholder	Represented Group	How do they represent this group?	Why have they been included?
James Bearly	Speed Cubers.	They regularly solve a Rubik's cube in under 2 minutes.	Speed cubers will likely be the second largest group of users and will have insights on the more competitive side of cubing.
Jake Elmer	Casual cubers.	They regularly attempt to solve a Rubik's cube, having spent at least one hour or more	Casual cubers represent a large group of the people likely to utilise my program and will

		attempting to solve one every week for over a year.	likely have insights about the convenience of the program.
Eddie Coulson	Beginner cubers.	They are starting to regularly invest time in cubing but have spent less than 5 hours doing so in total.	Beginner cubers can offer insight into the accessibility of the program for someone who knows none of the commonly used terms and perhaps doesn't know how to solve a Rubik's Cube
Connor Gilroy	One-off cubers/ non-cubers.	They have tried to solve a Rubik's cube less than 5 times in their life, for a sum total of less than 1 hour.	People who have no interest in repeated use of the program will have more interest and suggestions about how initially accessible the program is, with less focus of more advanced/later-use features.
Sam Flegg (myself)	Rubik's Cube program developers.	I am developing the program.	As the developer of the program, I will have access to the source code and can therefore offer code-related suggestions and improvements which end users will not be able to.

Research

Questionnaire

To find out people's opinions on a Rubik's cube program and what features I could implement I have conducted a questionnaire. Candidates were asked to give brief and concise answers.

Template

1. How many sessions of playing with a Rubik's Cube do you have per month?

<1 1-2 3-8 8+

2. How long does it typically take you to solve a Rubik's cube?

<2 minutes <10 minutes <1 hour >1 hour

3. On a scale of 1-10, with 1 being very easy, how difficult do you find solving a Rubik's cube?

4. What do you like about solving Rubik's Cubes?
5. What don't you like about solving Rubik's Cubes?
6. How do you think a Rubik's cube could be more fun?
7. What features would you like to see a Rubik's cube program have?
8. What concerns do you have about a Rubik's cube program?

Result 1: Speed Cuber

1. How many sessions of playing with a Rubik's Cube do you have per month?

<1 1-2 3-8 8+

2. How long does it typically take you to solve a Rubik's cube?

<2 minutes <10 minutes <1 hour >1 hour

3. On a scale of 1-10, with 1 being very easy, how difficult do you find solving a Rubik's cube?

1

4. What do you like about solving Rubik's Cubes?

They test my ability to think quickly and keep my mind engaged.

5. What don't you like about solving Rubik's Cubes?

Quick/rough turns can sometimes cause pieces to fall off.

6. How do you think a Rubik's cube could be more fun?

Various sizes and shapes

7. What features would you like to see a Rubik's cube program have?

A timer and leaderboard

8. What concerns do you have about a Rubik's cube program?

Slow response times and not being able to turn the cube and see all of it

Result 2: Casual Cuber

1. How many sessions of playing with a Rubik's Cube do you have per month?

<1 1-2 **3-8** 8+

2. How long does it typically take you to solve a Rubik's cube?

<2 minutes **<10 minutes** <1 hour >1 hour

3. On a scale of 1-10, with 1 being very easy, how difficult do you find solving a Rubik's cube?

3-4

4. What do you like about solving Rubik's Cubes?

They're engaging and a fun challenge

5. What don't you like about solving Rubik's Cubes?

They can be damaged or break apart

6. How do you think a Rubik's cube could be more fun?

Larger cubes

7. What features would you like to see a Rubik's cube program have?

The ability to change cube size

8. What concerns do you have about a Rubik's cube program?

Clunky interface

Result 3: Beginner Cuber

1. How many sessions of playing with a Rubik's Cube do you have per month?

<1 **1-2** 3-8 8+

2. How long does it typically take you to solve a Rubik's cube?

<2 minutes <10 minutes **<1 hour** >1 hour

3. On a scale of 1-10, with 1 being very easy, how difficult do you find solving a Rubik's cube?

5

4. What do you like about solving Rubik's Cubes?

Intelligence test

5. What don't you like about solving Rubik's Cubes?

Some scrambles can be really hard to solve

6. How do you think a Rubik's cube could be more fun?

Less hard scrambles

7. What features would you like to see a Rubik's cube program have?

Difficulty

8. What concerns do you have about a Rubik's cube program?

Weird interactions

Result 4: Non-Cuber

1. How many sessions of playing with a Rubik's Cube do you have per month?

<1 1-2 3-8 8+

2. How long does it typically take you to solve a Rubik's cube?

<2 minutes <10 minutes <1 hour >1 hour

Normally give up

3. On a scale of 1-10, with 1 being very easy, how difficult do you find solving a Rubik's cube?

9

4. What do you like about solving Rubik's Cubes?

It's a fun puzzles

5. What don't you like about solving Rubik's Cubes?

They're very difficult

6. How do you think a Rubik's cube could be more fun?

If it was easier

7. What features would you like to see a Rubik's cube program have?

A guide

8. What concerns do you have about a Rubik's cube program?

Too difficult

Result 5: Program Developer

1. How many sessions of playing with a Rubik's Cube do you have per month?

<1 1-2 3-8 8+

2. How long does it typically take you to solve a Rubik's cube?

<2 minutes <10 minutes <1 hour >1 hour

3. On a scale of 1-10, with 1 being very easy, how difficult do you find solving a Rubik's cube?

4. What do you like about solving Rubik's Cubes?

Fun, engaging

5. What don't you like about solving Rubik's Cubes?

Getting stuck/not knowing needed techniques

6. How do you think a Rubik's cube could be more fun?

More options

7. What features would you like to see a Rubik's cube program have?

Fully interactive cube, scrambler, solver, leaderboard, history, accounts, etc.

8. What concerns do you have about a Rubik's cube program?

Slow, inaccurate, doesn't 'feel' like a Rubik's cube.

Conclusion

After reviewing the questionnaire, people who solve many cubes a month and/or solve them quickly enjoy the challenge of solving the cube but due to their practice they find it easy. Therefore, I will investigate ways to make the cube harder, be it a guaranteed hard scramble, a larger cube, or some other technique. However, these people's main concerns about their software are its functionality, suggesting they would be happy with a normal cube as long as it is at least as functional as a physical cube.

For more intermediate and beginner people who find it harder and solve less cubes, their main feature request is a way to reduce the difficulty, possibly by implementing a guide. As I will very likely be implementing a solver, I may implement a hint button that can tell you the next move, or possibly moves, you should make.

Useful technical approaches

To create my program, I will utilise Python as this is the language I know. Python has many existing libraries that I can utilise to reduce the code requirements for this program. As Python is a very popular language and often utilised by new coders it is very well documented, allowing me to easily learn how to use any library I may need.

I will use Pygame for many – if not all – of the visual aspects of the game. Pygame is a popular library for making games and displaying elements. It allows for easily creating a window. It works primarily by having a game loop and events which you get() and handle yourself.

```

import pygame

pygame.init()
screen = pygame.display.set_mode((1600, 900))

while True:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()

    screen.fill((255, 255, 255))

    pygame.display.flip()

```

An example is:

The above image shows code that uses pygame to create a white window. It also checks for the quit button being pressed, in which case the window closes.

If I create a login system, I will most likely use Tkinter as it has many features, making it easy to create a simple graphical login system. A highly useful aspect are the widgets, which standardise the process for displaying various aspects such as text and entry boxes.

An example of the code used for a simple window is shown below.

```

self.root = tk.Tk()
self.root.title(name)

# prevent resizing
self.root.resizable = (False, False)

# for deleting everything
self.main_frame = ttk.Frame(self.root)

# window dimensions
self.window_width = 400
self.window_height = 450

# screen dimensions
self.screen_width = self.root.winfo_screenwidth()
self.screen_height = self.root.winfo_screenheight()

# centre points, left and top pos to centre window
self.centre_x = int(self.screen_width / 2 - self.window_height / 2)
self.centre_y = int(self.screen_height / 2 - self.window_height / 2)

# position
# widthxheight+x_pos+y_pos
self.root.geometry(
    f"{self.window_width}x{self.window_height}+"
    f"{self.centre_x}+{self.centre_y}"
)

```

The above code, whilst simple, handles the creation of an entire resizable window.

I will also utilise object-orientated programming. This means creating classes that can contain data and functions shared between the objects made from them. Objects are instances of a class and have access to the same data and functions. This makes it easy to create many similar things, e.g. A button – they look and function the same but do something different when pressed.

Similar Programs

The following are 3 online Rubik's cube programs that I have analysed. I have taken noted the good and bad features of each to ensure my program has all the necessary and good features and minimise the unnecessary functions.

Ruwix

[Online Rubik's Cube Simulator \(ruwix.com\)](http://ruwix.com)

The screenshot shows the Ruwix website's navigation menu. At the top is a logo of a Rubik's cube and the word "Ruwix". Below it is the text "Twisty Puzzle Wiki". The menu items are:

- Puzzle Simulators » (highlighted with a yellow box)
- 2x2x2
- 3x3x3
- 3x3x3 (3D) (highlighted with a blue box)
- 4x4x4
- 5x5x5
- 6x6x6
- 7x7x7
- 8x8x8
- 9x9x9
- 10x10x10
- 11x11x11
- Square-1
- Rubik's Clock
- Megaminx
- Pyraminx
- Skewb
- Hungarian Rings
- 15-Puzzle

Main pages »

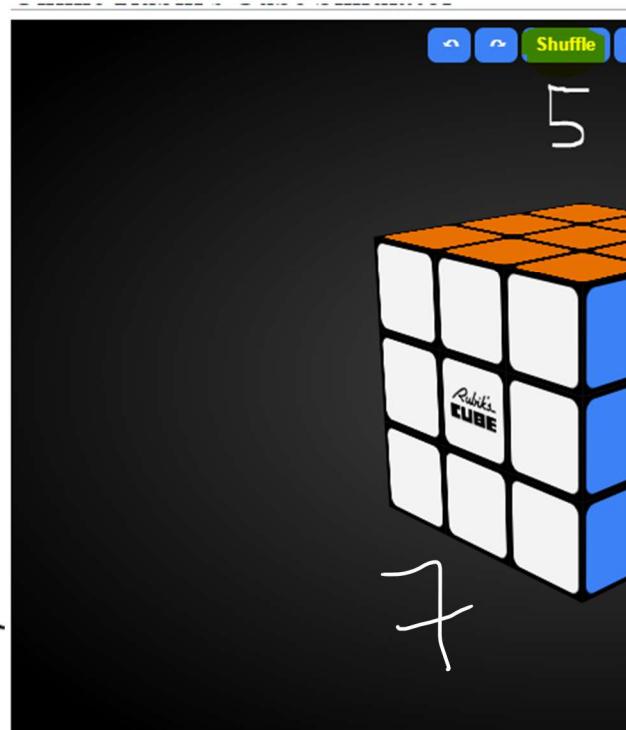
- Home page
- Programs
- Rubik's Cube
- Twisty Puzzles

Rubik's Programs

- Cube Solvers
- Stopwatch Timer
- Puzzle Scrambler
- Mosaic Generator
- Puzzle Simulators
- Widgets

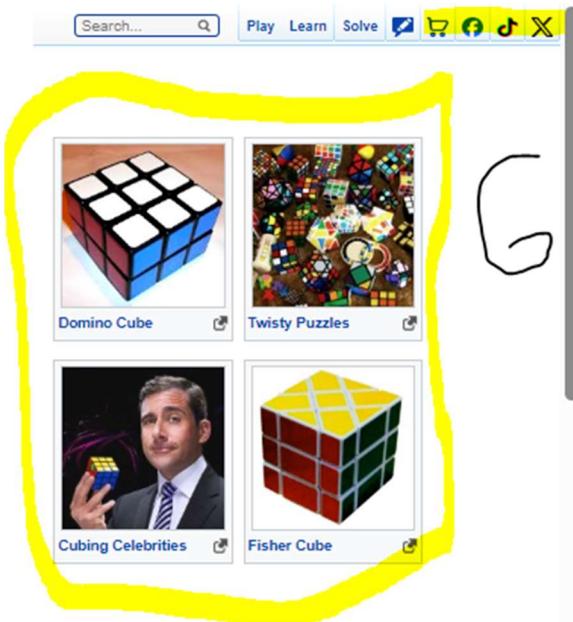
Rubik's Cube

- Cubing Terminology
- FAQ
- Notation
- Beginner's Solution



Play with the 3D Rubik's Cube simulator online. Press the scramble button and try to figure
Use your keyboard: the buttons on your keyboard are assigned to each face, according to the [note](#)

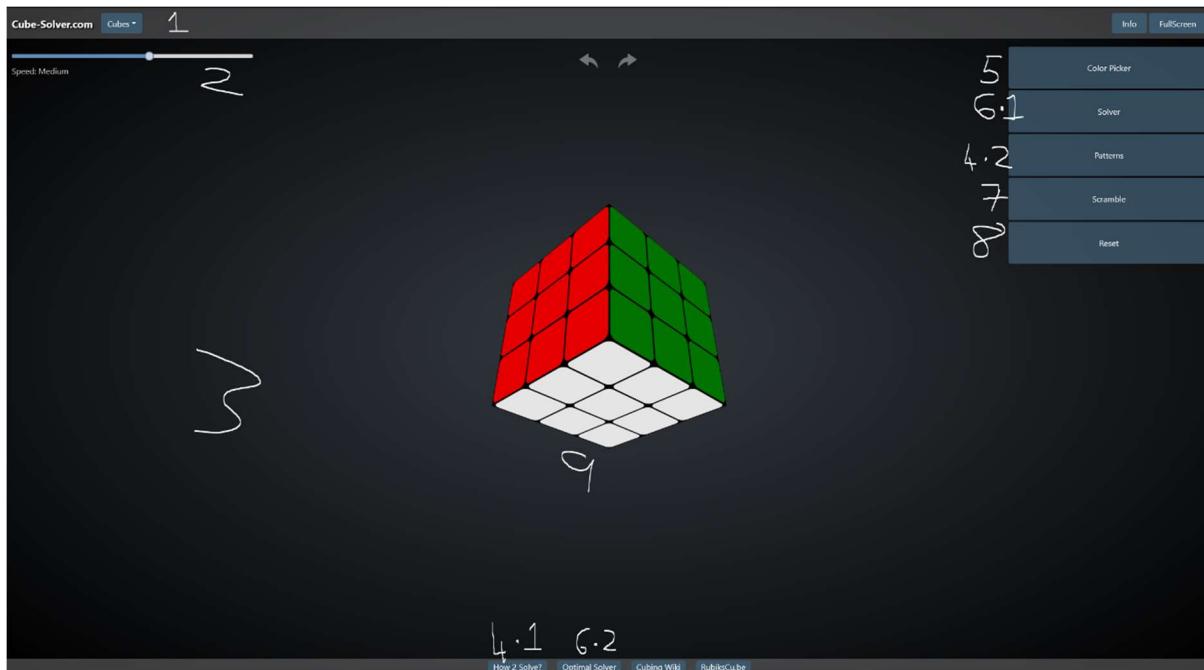
- Face: F R U B L D
- Slice: M E S
- Whole: X Y Z



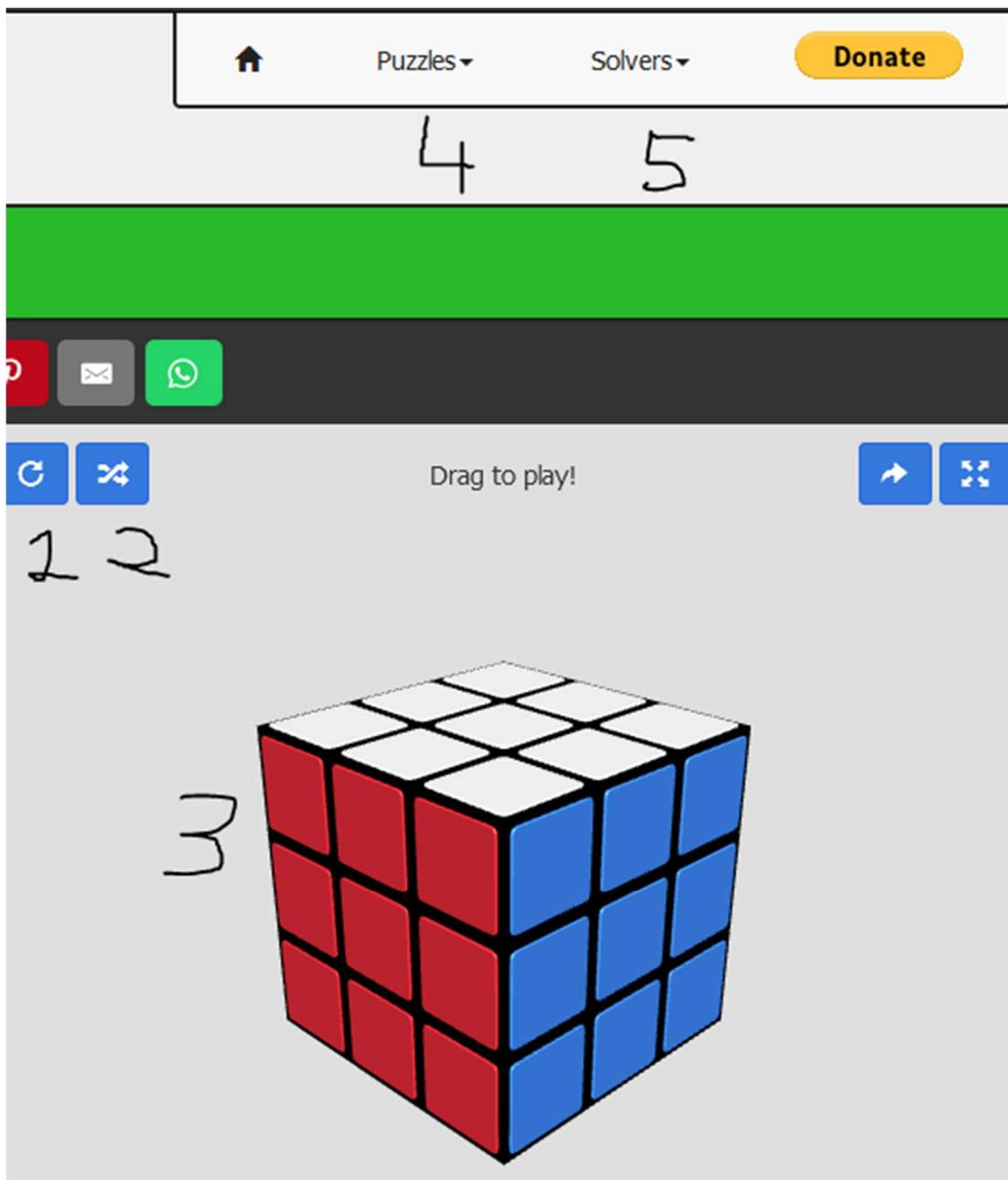
1. Ruwix has an option for all variety of cube types. This is a feature I would like to incorporate into my program; however, I believe it may greatly increase the time requirements. As such this will be one of the last features I will add if I add it at all.
2. A solve function is a staple of a Rubik's cube program and is a feature I will definitely add to my program. However, Ruwix have incorporated it by having you make the cube layout, allowing you to solve cubes that are not a part of the program. I will likely have my solve function solve your current cube, removing the requirement to enter the cube layout but stopping you from entering non-program cubes.
3. A stopwatch is useful for people curious about their solve time and especially speed cubers. This is an additional feature I hope to add to my program.
4. Ruwix has a simple-looking controls guide. However, it requires users to know the correct terminology. I believe this creates an unnecessary barrier to entry for non and new cubers. Therefore, I will likely create a more intuitive guide for my program, perhaps a visual one to make it obvious what each button press does.
5. A shuffle/scramble function is another staple of Rubik's cube program, and will be one of, if not the first feature I add to my program.
6. Ruwix has links to their other pages and social media. I believe this creates unnecessary visual clutter and creates an ugly UI. I will endeavour to avoid this in my program.
7. Ruwix also allows for rotating the cube with the cursor. Whilst I like the idea of this it can often lead to the cube being slanted which is very hard to undo. Due to this I will likely only allow for fixed rotation.

Cube Solver

Online NxN Rubik's Cube Solver and Simulator (cube-solver.com)



1. This is an option for other cube types/sizes. I believe this is a feature I should include in my program.
2. This is an option to modify the animation speed. This offers more convenience for the users, and I would also like to include this, however this is not important overall.
3. The black background is much easier to look at than the Ruwix's one and something I would like to include. However, as this is largely down to user preference I will try to make it an option for the user.
4. 1) This is a link to a guide on how to solve a Rubik's cube. Whilst I do not like the idea of linking to another website, I do the idea of an easily accessible guide.
2) This provides steps to achieve certain patterns. This alone is a reason to consider adding this feature, but it also serves to help solve the cube without doing too much of it for you. For this reason, I am definitely going to add this.
5. This allows you to recreate cube positions from other cubes/games. I will consider adding this feature.
6. 1) This shows steps to solve the cube and can do it automatically. Whilst I like it being automatic, it is incredibly unoptimized and can take hundreds of steps to solve a 5 move scramble. I will likely add a more optimised version of this.
2) This is a link to an optimised solver; however, it requires you to input your cube position as it is a link to another website. I will not be implementing this.
7. This scrambles the cube; I will definitely add this feature.
8. This allows you to reset the cube to its original position, also being solved. I had not considered this but will add this feature.
9. This program also has a 3d interactive cube, which I find to be much more responsive than the previous solution's version. I will attempt to implement this.



1. This resets the cube. As mentioned on the last program, I had not considered this, but it is a useful feature I will include.
2. A scramble button, a basic feature I will definitely include.
3. Like with the first program, it has a 3d interactive cube that I find to be clunky and would rather have a non-interactive cube.
4. This provides options for many different types of cubes, which is something I will try to implement.
5. This provides solvers for each type of cube. Again, this requires manually creating the cube position which is something I want to avoid. I do however want a solver for each cube type I implement.

Features

Essential Features

The Cube(s)

The most important part of a Rubik's program, the actual cube. I will implement a 3d cube which uses the keyboard as a controller. This will allow for having a 3d cube whilst avoiding the clunky interface that other programs had.

In addition to the standard 3 by 3 cube, I will add additional cube sizes. I will add 2 by 2 and 4 by 4 cubes, as well as non-cube variations. This allows for more variation in the program, helping prevent people getting bored.

Scrambler

Perhaps the most useful feature of the program, this allows people to have a cube they have to solve without having to scramble it themselves, meaning they can't just undo the moves done to scramble it, as they don't know them. This will utilise lots of randomness to ensure each scramble is different.

Solver

A common feature of Rubik's cube programs, this will solve the Rubik's cube. I will have this solve the current cube position instead of having the user input one, as I feel this makes it more accessible and easier to use. This can help people learn to solve cubes by showing them how to do it.

Tips/hints

I will implement a simple multi-stage guide to tell users what they should be attempting to do next to solve the cube. These stages will consist of multiple moves the user has to do by themselves. This will help the user learn how to solve the cube.

In the case a user gets stuck, there will also be a hint feature that utilises the solver to tell/show the user the next move they should do. This will help to ensure they do not get stuck and give up.

Timer

I will also implement an optional timer that starts at the first turn of cube and stops when the cube is solved. This will appeal to speed cubes allowing them to easily time themselves, thereby increasing the amount of people this program will appeal to.

Reset

A reset function will reset the cube to its default solved position and orientation, allowing users to quickly start again if they do give up on a cube.

Online leaderboard

I will implement an online leaderboard to encourage competition and giving users goals to strive for. I will have a leaderboard for each cube that shows the number of moves and the time taken.

Login System

A login system will allow for user scores to be saved as well as other user data. I will also add a guest login.

Guide

I will also need a guide, so players know how to play.

Save

I will add a save feature that will record and store the cube position.

Mouse Controls

I will attempt to add easy-to-use mouse controls.

Game History

This will allow users to track their progress and have measurable improvements.

Limitations

3D

Due to the limited amount of time and my unfamiliarity with 3D software in python, I am going to be unable to implement a true 3D interactive cube.

Multiple Cubes

As each cube would require its own logic and interactions, it would take an inordinate amount of time to implement multiple cubes. As such I am only going to implement a 3x3 cube.

Explained Tips/Hints

Due to the number of possible scenarios and my limited knowledge on cubing techniques, it is outside of my capabilities to explain to the user why the next move is best.

Online Leaderboard

As I do not have the resources or money to host a server, I will have to limit this to a local leaderboard.

Mouse Controls

As I am not sure how I would add this whilst avoiding the ‘clunky’ experience I had on other programs and am concerned about time restrictions I will not add this.

Requirements

Hardware

- 1.5 Ghz CPU or faster
- On-board graphics or any GPU
- 1 GB RAM
- 1080p display (recommended)

Software

- Python 3.11 or newer
- Pygame
- Tkinter
- Windows 7 or newer

Standard Cube

A standard 3 by 3 cube, that at the very least looks 3d if it isn't, is a must. It is the centre point of the program and there is no point in the program without it.

Scrambler

This is a standard Rubik's cube program feature, and my program would be incomplete without it.

Solver

Another common feature of a Rubik's cube program, mine will implement an algorithm to ensure efficient solves.

Tips/Hints

As the solver will be implemented this will be able to utilise that, making telling the user the next best move trivial.

Timer

This is relatively simple in all aspects and thus there is little downside to adding it.

Reset

Another feature that is trivial to add, only requiring a default state I can revert the cube to and wiping any move history.

Local Leaderboard

A local leaderboard should be included so people on the same device may compete with each other.

Login System

As I have made a login system in lesson it will be easy to adapt for my program

Guide

This is an important feature to add as otherwise users would have to discover how the program works by themselves.

Save

Utilising the login system, this shouldn't be too hard to add and will make the program more accessible as games can be stopped and continued later.

Game History

This should be reasonably simple to add, and I consider it an important feature for user retention.

Success Criteria

- There should be a professional looking visual representation of a 3-dimesional Rubik's cube as this is the core of the program and a non-standard cube layout (e.g. a net) may confuse users.
- The cube should have the correct logic - the result of any moves should match the result of performing the move on a real Rubik's cube.
- There should be a scramble feature able to produce a scramble for the user to solve. The scramble must be possible to solve.
- A solver feature should be included. This should be able to solve the cube move by move, allowing the user to see the steps required to solve it so they may learn from it.
- A hint feature should tell the user the next move they should make if they require help, as to decrease the chance that the user simply gives up.
- A timer function should be included to incentivise competitiveness. In line with this the timer must be as easy to use as possible, it should not cause any delays. It will automatically start upon the user's first move and automatically stop when the cube is solved.
- A local leaderboard will be included to allow people to compete. This should display, at a minimum: the ten quickest solve times, the respective usernames, and number of moves required.
- The leaderboard should only include solves that did not utilise the hint or solve functions.
- There should be a straightforward login system that utilises encryption and/or hashing for security.
- There should be a clear and concise guide to using the program to prevent any confusion.
- There should be a simple to use save function to allow users with limited time to play whenever they wish without worrying if they have enough time for a complete solve.
- The save function should be able to run automatically to prevent users from losing progress should they forgot to save or something unexpected happens – e.g. power loss.
- A game history function should be included to allow users to see how they have progressed overtime.

Design

Problem Decomposition

Table

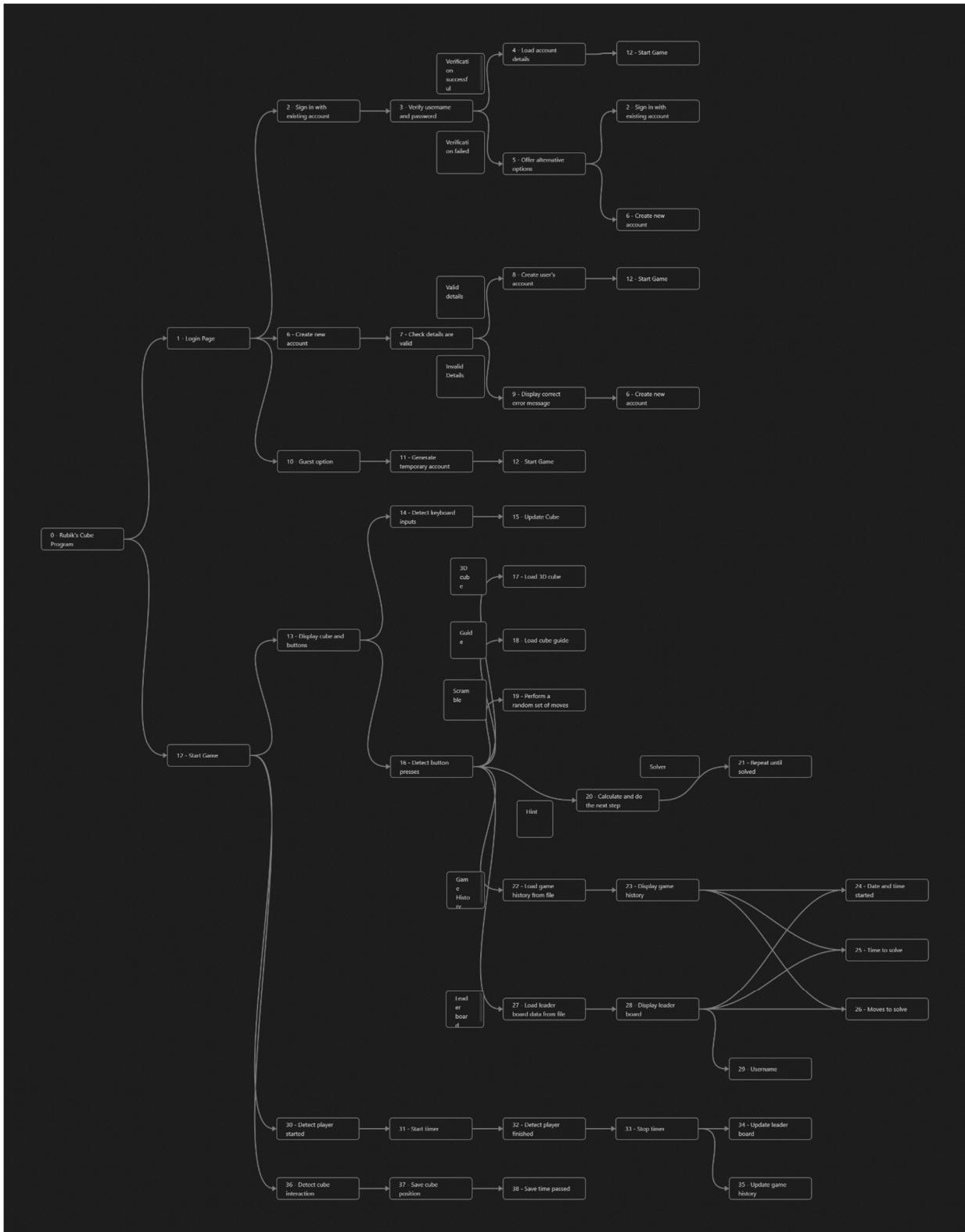
Sub-problem	Why this is suitable	Parent problem(s)	Child problem(s)
0 – Rubik's Cube Program	It is the entire program and requires decomposing	N/a	1, 12
1 – Login page	This is important to allow users to have their own accounts and has many complicated aspects	1	2, 6, 10
2 – Sign in with existing account	Users need to be able to access their account	1	3
3 – Verify username and password	The entered details must be checked against stored encrypted and/or hashed details	2	4, 5
4 – Load account details	The account details must be retrieved from a file and prepared for that user	3	12
5 – Offer alternative options	The user must be able to rectify a mistake when logging in	3	2, 6
6 – Create new account	New users must be able to create an account to have access to all the features an account offers	1, 5	7
7 – Check details are valid	The details entered must be able to be handled by all the code that may interact with it, including the code for logging in. E.g. the username must be unique	6	8, 9
8 – Create user's account	The new account must be created, with files in the appropriate format showing no game history, saves, etc.	7	12
9 – Display correct error message	The user must be informed why the details are invalid and how to fix their mistake	7	6
10 – Guest option	For new users who may not yet want to create an account there must be a way for them to still use the program	1	11
11 – Generate temporary account	A temporary account with no previous details that will only exist while in use must be created	10	12
12 – Start game	The core of the program, containing all the features of the game	0	13, 31, 38

13 – Display cube and buttons	The cube and button icons must be loaded and displayed for the user to see	12	14, 16
14 – Detect keyboard inputs	Any inputs from the keyboard must be detected and handled, as this is the user's primary way of interacting with the program	13	15
15 – Update Cube	The turns/rotations indicated by the user via the keyboard must be executed	14	N/a
16 – Detect button presses	The user's secondary method of interacting with the program, this is how they will access the additional features	13	17, 18, 19, 20, 22, 27
17 – Load 3D cube	The core program should be loaded for the user to interact with	16	N/a
18 – Load cube guide	A simple guide for the user should be loaded, allowing them to learn how to use the program	16	N/a
19 – Perform a random set of moves	Multiple random moves need to be executed to scramble the cube, providing the user a challenge to try and solve	16	N/a
20 – Calculate the next step to solve the cube	This will require a large algorithm to evaluate what the next step should be	16	21
21 – Repeat until solved	The algorithm in step 20 needs to be rerun until the cube is solved, meaning this process must also be able to recognise when the cube is solved	20	N/a
22 – Load game history from file	The user's username must be searched from a file, found, and then all the corresponding data must be parsed into the program, all whilst being reasonably fast	16	23
23 – Display game history	The user's game history must be displayed in a user-friendly manner without affecting the user's ability to use the program	22	24, 25, 26
24 – Date and time started	The date and time are crucial for the user to be able to track their progress	23, 28	N/a
25 – Time to solve	The time to solve is the main way to measure someone's skill at cubing	23, 28	N/a
26 – Moves to solve	Moves to solve is another method to measures someone's skill at cubing, and whilst not as popular, it is a more accessible for people	23, 28	N/a
27 – Load leaderboard data from file	The ten lowest times to solve and ten lowest moves to solve must be loaded from a file, meaning the entire file must be searched, but this must be done quickly	16	28
28 – Display leaderboard	Multiple pieces of data need to be presented in a reasonable manner which allows the user to see all the important information without crowding the screen and limiting the user's ability to interact with the program	27	24, 25, 26, 29

29 – Username	The username is important so that people can see who they are competing against and be inclined to challenge them	28	N/a
30 – Detect player started	The timer function must start running immediately after the user's first cube interaction, meaning this function must either always be checking for a cube interaction	12	31
31 – Start timer	The timer must be started in a manner that will persist while the program runs and will not be affected by things like frame rate	30	32
32 – Detect player finished	The cube state must be checked after every move by the user to see if it is solved	31	33
33 – Stop timer	The timer must be stopped, and the time taken recorded	32	34, 35
34 – Update leaderboard	The leaderboard must be updated with the date and time, time to solve, moves to solve, and user's username.	33	N/a
35 – Update game history	The user's game history must be updated with the time taken, moves taken, and the data and time.	33	N/a
36 – Detect cube interaction	After each move on the cube the save function should be run	12	37
37 – Save cube position	The complete cube position must be saved in a file so that the exact cube position can be recalled at a later date	36	38
38 – Save time passed	The amount of time passed and move count must also be saved so that they can continue being updated when the game continues	27	N/a

Flowchart

The following flowchart has been designed using decomposition to help show the logical steps of the program and to see where similar steps exist so one function can be coded to complete each task, decreasing the code requirement.



Solutions

Sorting Algorithms

A sort algorithm moves each element in a list so that they follow an order. Each algorithm described will assume sorting from smallest to largest, although this may not be the case in my program.

I will most likely use insertion and merge sort, although bubble and quicksort are options I will consider when coding. Insertion sort (and bubble sort) is very fast for small, near-sorted lists so I will most likely use this for the leaderboard. Merge sort (and quicksort) are best suited for larger lists as they are $O(n \log n)$ avg. time complexity. I have chosen merge sort over quicksort as its worst-case is still $O(n \log n)$ whereas quicksort is $O(n^2)$.

Bubble

Bubble sort involves going to each element and repeatedly swapping it with the element after it if the element after it is smaller. Time complexity $O(n^2)$, space complexity $O(1)$.

```
def sort_bubble(lst):
    swapped = False
    for i in range(len(lst) - 1):
        if lst[i] > lst[i+1]:
            lst[i], lst[i+1] = lst[i+1], lst[i]
            swapped = True
    if swapped:
        sort_bubble(lst)
```

Insertion

Insertion sort works by creating a sub-list at the start of the list with initial length 0 and then moving adding each element that is out of the sub-list into the sub-list, placing it in order as this is done, until the sub-list is the entire list and in order. Time complexity $O(n^2)$, space complexity $O(1)$.

```
def sort_insertion(lst):
    for i in range(len(lst)):
        item = lst.pop(i)
        for j in range(i+1):
            if lst[j] > item: # not j+1 as currently popped
                lst.insert(j, item)
                break
            elif j == i:
                lst.insert(j, item)
```

Merge

Merge sort works by recursively splitting the list until each list has length one and then combining each list and placing the elements in order as this is done, until all lists are sorted back into one list. Time complexity $O(n \log n)$, space complexity $O(n)$.

```
def sort_merge(lst):
    if len(lst) <= 1:
        return

    mid = len(lst) // 2
    left = lst[:mid]
    right = lst[mid:]

    sort_merge(left)
    sort_merge(right)

    i, j, k = 0, 0, 0
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            lst[k] = left[i]
            i += 1
        else:
            lst[k] = right[j]
            j += 1
        k += 1

    while i < len(left):
        lst[k] = left[i]
        i += 1
        k += 1

    while j < len(right):
        lst[k] = right[j]
        j += 1
        k += 1
```

Quick

Quicksort works by picking any element of the list as a pivot and then moving all smaller elements to the left of the pivot and all larger elements to the right of the pivot. Then each side of the pivot is considered as a sub-list and recursively quick sorted, until each sub-list has length one, when the list will be sorted. Time complexity $O(n \log n)$, space complexity $O(\log n)$.

```
def sort_quick(lst, low, high):

    def sort(lst, low, high):
        pivot = lst[high]
        i = low - 1
        for j in range(low, high):
            if lst[j] <= pivot:
                i += 1
                lst[i], lst[j] = lst[j], lst[i]

        lst[i+1], lst[high] = lst[high], lst[i+1]
        return i+1

    if low < high:
        pivot_index = sort(lst, low, high)
        sort_quick(lst, low, pivot_index - 1)
        sort_quick(lst, pivot_index + 1, high)
```

Searching Algorithms

A search algorithm finds the position of an element in a list. The 2 I may be using have space complexity of $O(1)$.

Linear search is best suited for small lists, although I cannot think of a need for this. Binary search is best suited for larger lists, and I will likely use it for searching for users in the saves file.

Linear

Linear search works by starting at position 0 of the list and moving to the next element until it matches the desired element. Time complexity $O(n)$

```
def search_linear(lst, item):
    for i in range(len(lst)):
        if lst[i] == item:
            return i
    return -1
```

Binary

Binary search works by repeatedly splitting an ordered list. It starts by looking at the middle of the list. If the element is found, the middle of the list is its position. If the element being looked at is lower than the desired element (assuming the list is ordered smallest to largest), then it considers everything to the right of the list as a sub-list (using low and high points), and then repeats the process. Time complexity $O(\log n)$.

```
def search_binary(lst, item):
    low = 0
    high = len(lst)
    while high - low > 0:
        mid = (high - low) // 2 + low
        if lst[mid] == item:
            return mid
        elif lst[mid] < item:
            low = mid
        elif lst[mid] > item:
            high = mid
    return -1
```

Cube Algorithms

Image

I will need to have an algorithm to create the image of cube. It may draw the cube as well, or if it does not it will return the image of it. I will decide which it does when I have started developing my program and have a better idea of how it will work, although I imagine it will work via return. The image will be gotten by drawing each individual square on the cube onto a surface, with their colour being dictated by a 3d array.

Turns

When the user inputs their move I will need to have a function to process this. It will need to update a 3d array that stores the cube's state, correctly swapping or maintaining the colour

stored in each position. I may use the NumPy library to help with this, as it has many functions for working with arrays, which my Rubik's cube will use.

Rotations

When the user inputs a rotation the cube array needs to be updated so that each face is moved to do this rotation. However, it needs to appear to the user as the cube is unchanged, and that they are simply looking at a different part. The turns function may be useful for this, although I will have to ensure there is nothing to indicate to the user that the turns function has been used (e.g. if the number of moves is recorded, this should not be updated)

Scramble

There should be a function that randomises the cube state to provide a cube for the user to solve. As such, the scramble function should also ensure the cube state produced is one that is possible to solve. I will use a series of turns to scramble the cube to ensure that the cube is possible to solve as simply undoing each turn would solve the cube. This will likely utilise the random library to help ensure the cube state produced is truly random and not biased by me.

Solver

The program is also meant to have a solver. After some research I have found the best method for this is likely a long algorithm consisting of many if statements that is run repeatedly until the cube is solved. However, this would take a long time and be tedious to implement. As such, I may initially use a stack consisting of every move the user has done and what moves were done to scramble the cube, and then I could pop elements from the stack and do the opposite of them to solve the cube. This method would be inefficient and may take much longer to solve the cube than required, but it would be a lot easier to implement initially, and I could look into implementing the 'proper' algorithm in later versions.

Hints

The hint function may be able to utilise the solver and simply get the next move from that function, although this may require editing the solve function slightly to allow a single move to be gotten.

Timer

The timer needs to be automatically triggered when the users start, although starting could be defined as either when the scramble function finishes or when the user makes their first move. I believe it would be easier to start the timer when the scramble function finishes so this is what I will do initially, however the World Cube Association allows contestants some time to inspect the cube scramble before they must start solving it, so I believe the correct thing to do would be to start the timer when the user makes their first move. As such I will endeavour to implement this in my later versions. However, this may not be something that is appropriate to implement in the timer itself, and instead the automatic starts and stops may be managed by a different section of code. The timer itself needs to manage creating a timestamp of when the timer is started, and tracking the amount of time passed since then, and stopping the timer when required.

Leaderboard

The leaderboard function will need to check all users solves to sort them from quickest to slowest, and check if they are eligible to be on the leaderboard (they mustn't have used the solve or hint feature). As more and more solves are recorded the time taken to do this will increase, potentially eventually taking an unreasonable amount of time. As such, I will create a sorted list of the top ten quickest eligible solves (which may be changed to a leaderboard of every solve in a later version) which every solve will be checked against at the time of solve to see if the list needs to be updated. This way loading the leaderboard will take the same amount of time regardless of how many users solves there are as all that needs to be done is a list must be loaded.

Login

A login system must also be implemented to manage users and their associated data. From some of my previous coding projects I have a premade login system which I can import and utilise instead of creating another login system. However, this login system is far from perfect, so in my later versions of the project I may improve this login system or simply find a better one.

Initially I will use my premade one which will call a function with a username as a parameter, so I will need to create user system that has an appropriate function to load the user's data into the program and will store the user's game related data, as my login system only handles login data.

Guide

The program will require a guide to instruct users on how to use the cube. To achieve this, I will get the image a default cube by reusing the function for getting the actual used cubes image and then add arrows to the image to show what turns and rotations will be done for corresponding key presses.

Save

A save function should save users progress so that they may return to a solve they are in the middle of if they have to leave for any reason/they run into any computer problems. The save function should run automatically and not run too often so as to not cause performance issues. To do this I may use threading to allow the save function to run (and sleep) in the background so as to allow the main program to run uninterrupted.

Game history

There should be a game history function to allow users to see their past games and progress. This must get data from a user's data file and parse it out to display data about past games in a manner the user understands. This therefore also necessitates saving data about every attempted solve, which will require getting the time taken, moves taken, cube state (which will be a 3d array), and possibly the original scramble.

Usability Features

Usability features are important to ensure the game is fun and accessible to everyone. If users have to spend too long learning how to use the program, they may get bored and stop using the program, or have severely decreased enjoyment.

As such, I will be using simple keyboard controls for the main part of the game, as this is the simplest and most accessible method of interaction for many inputs, and unlikely to prevent anyone from being able to play or causing any confusion.

I will also be utilising simple point and click navigation of the program and large, sometimes graphical, buttons, are these are also simple to understand, and point and click navigation is the standard that most people will be familiar with.

Data and Variables

Variable Name	Type	Description	Sample
width	Int	The screen width, useful for creating the game window and positioning elements to be placed on it	1600
height	Int	The screen width, useful for creating the game window and positioning elements to be placed on it	900
screen	Pygame.Surface	The displayed screen on which every item to be displayed must be placed	N/a

used_cube	3D array	This array will represent the cube and store the colour at each position.	<pre>used_cube = [[[ORANGE, ORANGE, ORANGE], [ORANGE, ORANGE, ORANGE], [ORANGE, ORANGE, ORANGE],], [[GREEN, GREEN, GREEN], [GREEN, GREEN, GREEN], [GREEN, GREEN, GREEN],], [[RED, RED, RED], [RED, RED, RED], [RED, RED, RED],], [[BLUE, BLUE, BLUE], [BLUE, BLUE, BLUE], [BLUE, BLUE, BLUE],], [[WHITE, WHITE, WHITE], [WHITE, WHITE, WHITE], [WHITE, WHITE, WHITE],], [[YELLOW, YELLOW, YELLOW], [YELLOW, YELLOW, YELLOW], [YELLOW, YELLOW, YELLOW],],]</pre>
BLACK	Tuple	This will store the RGB value of the colour to be used throughout the program. As this is a colour it will be stored as a constant.	(0, 0, 0)
WHITE	Tuple	This will store the RGB value of the colour to be used throughout the program. As this is a colour it will be stored as a constant.	(255, 255, 255)

YELLOW	Tuple	This will store the RGB value of the colour to be used throughout the program. As this is a colour it will be stored as a constant.	(255, 255, 0)
ORANGE	Tuple	This will store the RGB value of the colour to be used throughout the program. As this is a colour it will be stored as a constant.	(255, 165, 0)
RED	Tuple	This will store the RGB value of the colour to be used throughout the program. As this is a colour it will be stored as a constant.	(255, 0, 0)
GREEN	Tuple	This will store the RGB value of the colour to be used throughout the program. As this is a colour it will be stored as a constant.	(0, 255, 0)
BLUE	Tuple	This will store the RGB value of the colour to be used throughout the program. As this is a colour it will be stored as a constant.	(0, 0, 255)

GREY	Tuple	This will store the RGB value of the colour to be used throughout the program. As this is a colour it will be stored as a constant.	(169, 169, 169)
Cube3D	Class	This will manage creating the image of the 3D cube and drawing it, as well as updating the image and storing the position it should be drawn to (as this is unlikely to change in my program)	<pre>class Cube3D: def __init__(self, screen, pos): self.surf = screen self.pos = pos def update(self): ... def get_image(self): ...</pre>
CubeGuide	Class	Similar to Cube3D, CubeGuide will manage the drawing and updating the guide cube, and storing data, although instead of making the guide cube image from scratch this class will utilise inheritance and polymorphism.	<pre>class CubeGuide: def __init__(self, screen, pos): self.surf = screen self.pos = pos def update(self): ... def get_image(self): base_image = Cube3D_var.get_image() ...</pre>

default_font	Pygame.freetype.SysFont	default_font will hold the main font I will use throughout my program to display text, to ensure it is consistent throughout.	default_font = pygame.freetype.SysFont("calibri", 15)
moves	Stack (list)	moves will store every move the player makes so that the solve function can pop and undo moves repeatedly to solve the cube for the user.	

Validation

Throughout the program I will need to validate various aspects.

One thing I will validate is screen positions. A position that an image is going to be blitted to, should be validated to ensure it is on the screen. This will raise an error in case an invalid screen position is calculated.

Another form of validation is handling invalid key presses. Pygame itself handles this, as all key presses are added to the events list with their key recorded as type. You can then simply check for events with the key type of keys that are supposed to be used, and any other key presses will simply be ignored.

The last form of validation will be used with the image of the cube used for the guide cube. I want to display the default cube for the guide image_ and I don't want it to be able to be changed. To achieve this, I will check if the guide cube image is being displayed and ignore any cube interactions if it is.

Iterative-Development Test Data

Due to how difficult it would be to ensure that the changes to a 3D array correctly match a real Rubik's cube, and how reliant on visual output my program is, am going to test some elements based on their display to a pygame window.

Test No.	What is being tested	Description	Method	Expected Output	Pass/Fail
1	3D cube algorithm	There should be an image algorithm that either returns a pygame.Surface containing an image of the cube or display the cube to the screen.	Blit the image returned by the function to the screen or call the function, inside the main game loop. Repeat whilst making changes to used_cube.	An image should be displayed, and it should match used_cube.	
2	Cube turn – vertical, left, up	Executing the turns function with the correct parameters for the given turn should result in the leftmost column being rotated upwards.	Execute the turns function with the parameters for the turn. Run the cube display, the display should show the new cube state.	The image of both cubes should be exactly the same.	
3	Cube turns – vertical, middle, up	Executing the turns function with the correct parameters for the given turn should result in the middle column being rotated upwards.	The same rotation should be done to a real Rubik's cube.		
4	Cube turns – vertical, right, up	Executing the turns function with the correct parameters for the given turn should result in the rightmost column being rotated upwards.			
5	Cube turns – vertical, left, down	Executing the turns function with the correct parameters for the given turn should result in the leftmost column being rotated downwards.			

6	Cube turns – vertical, middle, down	Executing the turns function with the correct parameters for the given turn should result in the middle column being rotated downwards.			
7	Cube turns – vertical, right, down	Executing the turns function with the correct parameters for the given turn should result in the rightmost column being rotated downwards.			
8	Cube turns – horizontal, top, right	Executing the turns function with the correct parameters for the given turn should result in the upper row being rotated right.			
9	Cube turns – horizontal, middle, right	Executing the turns function with the correct parameters for the given turn should result in the middle row being rotated right.			
10	Cube turns – horizontal, bottom, right	Executing the turns function with the correct parameters for the given turn should result in the lower row being rotated right.			
11	Cube turns – horizontal, top, left	Executing the turns function with the correct parameters for the given turn should result in the upper row being rotated left.			
12	Cube turns – horizontal, middle, left	Executing the turns function with the correct parameters for the given turn should result in the middle row being rotated left.			

13	Cube turns – horizontal, bottom, left	Executing the turns function with the correct parameters for the given turn should result in the lower row being rotated left.			
14	Scramble	The scramble function should randomly position the individual squares whilst still ensuring that the cube is solvable.	Add a delay between each move in the scramble function. Run the scramble function and follow along with a real Rubik's cube.	Each move done by the scrambler should be possible on the real Rubik's cube.	
15	Solver - solving	The solve function should solve the cube, showing the user each step, ensuring that each move is possible and not simply changing the used_cube to fit as needed.	Manually scramble the cube, doing each move to a real Rubik's cube as well. Run the solver and follow the moves on the Real Rubik's cube.	Each move done by the solver should be possible on the real Rubik's cube and at the end the cube should be solved.	
16	Solver – stop solving	If used_cube reaches a solved state, the solver should stop solving, regardless of if there something such as a moves list indicates there are more moves to do to solve the cube.	Manually scramble the cube, ensuring that you return to a solved state at least once then scramble from there. Run the solver.	The solver should stop when it reaches the first solved state.	
17	Check solved	There should be a function to check if a cube state is solved or not.	Test the function using multiple different cube states, some of which are manually or automatically scrambled.	The outputs should match the given cube state.	

18	Hints	The hint function should complete one move towards the solve. It must only be one move, and it must help solve the cube.	Scramble the cube then run the runt the hint function. Note the move that it makes. Undo that move and then run the solver (test 15 must have passed).	Only one move should be completed by the hint function. The move should match the one done by the solver.	
19	Timer – time elapsed	The timer should correctly record the amount of timer that has passed since it started	Start the timer. Wait for 10 seconds (counted via a trusted, real, timer). Print the time elapsed. Repeat a few times with various amounts of time waited.	The trusted timer and the timer being tested should have a matching (or very similar, to account for human error) times.	
20	Timer – auto start	The timer should automatically start upon scramble.	Scramble the cube. Solve the cube. Scramble the cube, use hint function. Scramble the cube. Use the solve function. Scramble the cube.	Each time the cube is scrambled the timer should start,	
21	Timer – auto stop	The timer should automatically stop upon being solved.	Monitor the time elapsed during this.	Upon being solved and when the solver is used, the timer should stop. The timer should not stop if these do not occur.	
22	Leaderboard – Eligibility check	Each entry should be checked to see if they are faster than the slowest time on the leaderboard, to see if they have made it onto the leaderboard.	Submit a completion with a slower completion time than the slowest. Submit a completion with a faster time than the slowest. Submit a completion time identical to the slowest.	Only the completion with the faster time should be considered for updating the leaderboard.	

23	Leaderboard – add entry	If the leaderboard isn't full any completion should be added to the leaderboard. If the leaderboard is full and entry is eligible, the new entry should replace the slowest time on the leaderboard.	Add an entry when the leaderboard is empty. Add an entry when the leaderboard is half full. Add an entry when the leaderboard is full.	The first two entries should be automatically added to the leaderboard. The last entry should replace the slowest entry on the leaderboard.	
24	Leaderboard – sort leaderboard	When a new completion is added to the leaderboard, the leaderboard needs to be sorted to ensure that completion ends up in the correct position. The list should be ordered by ascending times.	Sort the leaderboard when it is already in order. Sort the leaderboard when it is in descending order. Sort the leaderboard when it is randomised. Sort the leaderboard when 2 identical times exist.	Each leaderboard should end up sorted. Manually check this.	
25	Leaderboard – save times	The ordered list of leaderboard times should be able to be saved to a text file so that they are kept even when the program ends.	Save the leaderboard when it is empty. Save the leaderboard when it has no completions. Save the leaderboard when it is half full. Save the leaderboard when it is full.	In each case the text file should be updated with the leaderboard.	
26	Leaderboard – load saved times	The saved leaderboard times need to be able to be loaded so they can be displayed and be checked against for any new records.	In each case close the program and start it again, attempting to load these saves.	In each case the leaderboard should be updated to match the text file.	

27	Guide algorithm - image	<p>The should either be a function that returns a pygame.Surface or a procedure that draws the image to the screen.</p> <p>The image should show how to use the cube.</p>	<p>Either blit the pygame.Surface to the screen or call the procedure inside a game loop.</p>	An image should be displayed.	
28	Guide algorithm – prevent moves	The cube image should only display the default cube and as such it should not allow cube interactions to happen when the guide is being displayed.	When displaying the cube, try the following: turns, rotations, scrambling and solving.	<p>None of the functions should work.</p> <p>The cube should remain unchanged.</p>	
29	Save – to file	The save algorithm should be able to save all the key data to a text file.	<p>Use the save function with a variety of different key information, including extreme test data such as the timer being at 0.0 seconds and the cube already being solved.</p> <p>In each case attempt loading the program with this saved data, ensuring it is loaded as current game data.</p>	<p>The save file should be updated to include the key data for the user.</p>	
30	Save – load form file	The save algorithm should be manage loading the data from the text file and updating game values so it is as if the saved state has been achieved in the current session.			
31	Save – autosave	The save function should automatically run periodically.	<p>Start a game and make some changes to the cube. Then wait the amount of time set between saves.</p> <p>Once this time has passed closer and reopen the program.</p>	<p>The cube should be in the same state as it was when the program was closed,</p>	

32	Game history	When a solve is complete, either by the solve function being used, the scrambler being used, or the cube being solved, the data about that solve should be saved to a list of solves.	Finish a solve using the solver, scrambler, and by solving manually.	These three solves should be added to game history list.	
----	--------------	---	--	--	--

Post-Development Test Data

Post development tests will focus on stakeholder reviews.

Test plan

Test No.	What is being tested	Type	Description	Pass criteria	Stakeholder responses.	Pass / Fail
33	3D cube – image.	Function	There is a 3D representation of a cube.	All stakeholders must answer yes		
34	3D cube - professionalism	Usability	The 3D cube must look professional.	Avg. score >= 70%		
35	Logic – possible states.	Function	It must be possible to reach every possible cube state.	All stakeholders must answer yes.		
36	Logic – possible moves.	Function / Robustness	All moves able to be done to the cube must be possible on a real Rubik's cube.			
37	Controls - cube.	Usability	The controls for interacting with the cube must be simple and intuitive.	Avg. score >= 70%		

38	Controls – program.	Useability	The controls for interacting with the program must be simple and intuitive.			
39	Scramble.	Function	There must be scramble function to scramble the cube.	All stakeholders must answer yes.		
40	Solver – solves cube.	Function	There must be a solve function that solves the cube.			
41	Solver – showcase moves.	Function, Useability	The solve function must show each move being done to solve the cube in an understandable manner.	Avg. score >= 70%		
42	Hints.	Function	A hint feature should show the user the next move to make.	All stakeholders must answer yes.		
43	Timer – timing.	Function	A timer should be available to time solves.			
44	Timer – auto start.	Useability	The timer automatically starts.			
45	Timer – auto stop.	Useability	The timer automatically stops.			
46	Leaderboard – image.	Function	There is a leaderboard.			
47	Leaderboard – solves.	Function	The leaderboard should display the ten quickest solve times or more, in ascending order.			
48	Leaderboard – details.	Function	Each entry must display: username, the number of moves required, the time taken.			

49	Login System – logs in.	Function	There should be a login system that allows users to login in.			
50	Login System – straightforward.	Useability	The login system should be straightforward and easy to use.	Avg. score >= 70%		
51	Save – loading.	Function	Upon logging in user data should be loaded.	All stakeholders must answer yes.		
52	Save – automatic.	Useability	The save function should run automatically.	All stakeholders must answer yes.		
53	Guide – exists.	Function	There must be a guide that shows how to use the program.			
54	Guide – user display.	Useability	The guide must be clear and concise.	Avg. score >= 70%		
55	Game history – exists.	Function	There must be a game history function that displays previous game history,	All stakeholders must answer yes.		
56	Game history – usefulness.	Useability	The game history function must make it easy to see how you have progressed over time.	Avg. score >= 70%		

Quiz template

Stakeholder name:			
Test no.	Question	Answer type	Stakeholder Answer
33	Is there a representation of a 3D cube?	Yes or No	
34	How professional does the 3D cube look?	1 to 10	
35	To your knowledge, is every move that's possible on a real Rubik's cube possible on the program's Rubik's cube?	Yes or No	
36	To your knowledge, is every move possible on the program possible on a real Rubik's cube?		
37	How well do the cube controls meet the description: simple and intuitive?	1 to 10	
38	How well do the program controls meet the description: simple and intuitive?		
39	Is there a scramble function?	Yes or No	
40	Is there a solve function?		
41	How well does the solve function show each move done (in regards to you being able to understand it)?	1 to 10	
42	Is there a hint function that shows you the next move to make?	Yes or No	
43	Is there a timer to track how long solves take?		
44	Does the timer start automatically?		
45	Does the timer stop automatically?		

46	Is there a leaderboard?		
47	Does the leaderboard show the ten quickest solves in ascending order?		
48	Does each leaderboard entry display: the username, the time taken, the moves taken?		
49	Is there a login system?		
50	How straightforward to use is the login system?	1 to 10	
51	Is your user and game data loaded when you log in?	Yes or No	
52	Does the save function run automatically?		
53	Is there a guide to use the program?		
54	How clear and concise is the guide?	1 to 10	
55	Is there a function to see your game history?	Yes or No	
56	How easy does the game history function make it to see how you have progressed?	1 to 10	

Implementation

Prototype One

To begin I will create a simple cube net with the purpose of testing the cube logic. This will be a very graphically simple cube with few features, but it will allow me to test the most important part of my program.

Development

Cube Display (Net, not 3D):

- [Window](#) for display
- Cube [image](#)

Cube Logic:

- Data [storage](#)
- Data [manipulation](#)

Validation:

- Validation [function](#)
- Function [usage](#)
- [Ignore invalid key presses](#)

```

1 # black, isort and flake8 used for formatting
2 import copy
3 import sys
4
5 import numpy
6 import pygame
7
8 # colours
9 BLACK = (0, 0, 0)
10 WHITE = (255, 255, 255)
11 YELLOW = (255, 255, 0)
12 ORANGE = (255, 165, 0)
13 RED = (255, 0, 0)
14 GREEN = (0, 255, 0)
15 BLUE = (0, 0, 255)
16
17 # cube design
18 # split into sides as easier to write
19 up = [
20     [WHITE, WHITE, WHITE],
21     [WHITE, WHITE, WHITE],
22     [WHITE, WHITE, WHITE],
23 ]
24 down = [
25     [YELLOW, YELLOW, YELLOW],
26     [YELLOW, YELLOW, YELLOW],
27     [YELLOW, YELLOW, YELLOW],
28 ]
29
30 left = [
31     [ORANGE, ORANGE, ORANGE],
32     [ORANGE, ORANGE, ORANGE],
33     [ORANGE, ORANGE, ORANGE],
34 ]
35
36 right = [
37     [RED, RED, RED],
38     [RED, RED, RED],
39     [RED, RED, RED],
40 ]
41
42 front = [
43     [GREEN, GREEN, GREEN],
44     [GREEN, GREEN, GREEN],
45     [GREEN, GREEN, GREEN],
46 ]
47
48 back = [
49     [BLUE, BLUE, BLUE],
50     [BLUE, BLUE, BLUE],
51     [BLUE, BLUE, BLUE],
52 ]

```

```

53
54     # default cube may always be shown and used |to check against for solves
55     default_cube = [
56         left,
57         front,
58         right,
59         back,
60         up,
61         down,
62     ]
63     # deepcopy passes by value, not reference, ensuring default_cube is not changed
64     used_cube = copy.deepcopy(default_cube)
65

```

In this section of code, I have initially imported the necessary libraries for later use in the program and then defined some colour's RGB values. The variable names of the colours have been written in all caps to signify that they are constants. Additionally, they have been written as tuples instead of lists as tuples are immutable.

I have then manually defined the default cube, creating a 2D array for each face, and then combined these into a 3D array. This could have been achieved by using loops; however this method provides a more visual representation of the cube and reduces the risk of an error during this section, allowing me to focus on developing the logic and functions required for the cube. I have called this variable 'default_cube' as it is the starting (solved) position of the cube and will be used for checking if the cube is solved – which will be important for the solver, leaderboard and game history. I have then created a deep copy of the cube called 'used_cube', which is the cube that will be updated and changed by user actions. A deep copy was used as this creates a new array by value instead of reference, ensuring default-cube is not changed by actions done to 'used_cube'.

```

66
67     # Window
68     width = 1600
69     height = 900
70     screen = pygame.display.set_mode((width, height)) # creates screen object
71     default_colour = BLACK
72

```

In this section I have created a pygame window for displaying the cube. I have defined width and height as variables as these will be used for positioning images later. I also created the 'default_colour' variable so that I can easily change the background colour without majorly altering the program, which may help with adding it as a feature if that is something I wish to-do at a later date. This is crucial for displaying the cube and all features associated with it.

```
73
74     def validate_screen_positions(pos): 3 usages
75         """
76             Ensures a position is within the confines of a standard 4k resolution screen
77
78             This function is designed so that it can be used
79             in the same place as the pos parameter.
80             It should simply be placed where the pos wants to be used.
81             It will quit the program if the pos is invalid.
82
83             :param pos: the position to check
84             :type pos: tuple[int, int]
85             :return: tuple[int, int]
86             """
87
88         if pos[0] < 0 or pos[0] > 3840 or pos[1] < 0 or pos[1] > 2160:
89             print("Error: invalid x or y value has been calculated.")
90             pygame.quit()
91             sys.exit()
92
93         else:
94             return pos
```

In this section I have designed a function validate screen positions, ensuring that if a screen position is outside the bounds of what is reasonably expected the program will end. This will be largely redundant in prototype one, due to the static nature of the window being displayed to, however it will become useful in later versions that implement resizable windows, and feature more calculations to display elements, increasing the likelihood of error. Additionally, as prototype one is a small program I could easily test this function without something else causing an error first.

I chose a 4k resolution (3840 x 2160) as this is the largest resolution commonly used by computer monitors – the intended display.

```

94
95     def cube(colour_3d_array):
96         """
97             Blits cube net to screen
98             :param colour_3d_array: cube colour
99             :type colour_3d_array: 3d array of tuples
100            """
101
102     def square(colour):
103         """
104             Creates the image of a single square
105             :param colour: RGB value
106             :type colour: tuple
107             :return: pygame.Surface
108            """
109
110         surf = pygame.Surface((50, 50))
111         surf.fill(colour)
112         return surf
113
114
115     def row(colour_list):
116         """
117             Creates the image of a row of 3 squares
118             :param colour_list: List of RGB values
119             :type colour_list: List len(3) of Tuples
120             :Return: pygame.Surface
121            """
122
123         surf = pygame.Surface((170, 50))
124         surf.fill(default_colour)
125         for i in range(3):
126             # iterates alongside the list of colours,
127             # getting a square with the respective colour and
128             # blitting it to calculated position
129             # i * 50 ensures the square is blitted after the previous one; not inside it
130             # i * 10 adds 10 spacing between the cubes
131             surf.blit(square(colour_list[i]), dest:(i * 50 + i * 10, 0))
132
133
134     def face(colour_array):
135         """
136             Creates one face (side) from 3 rows
137             :param colour_array: Array of RGB values [row1, row2, row3]
138             :type colour_array: 2D array of tuples
139             :return: pygame.Surface
140            """
141
142         surf = pygame.Surface((170, 170))
143         surf.fill(default_colour)
144         for i in range(3):
145             # iterates alongside the list of rows,
146             # getting and blitting the row image to calculated position
147             # i * 50 ensures the row is placed beneath, and not inside, the previous row
148             # i * 10 is for spacing between the rows
149             surf.blit(row(colour_array[i]), dest:(0, i * 50 + i * 10))
150
151
152

```

```

148 # 4 of the faces are placed next to each other so a loop can place them
149 for i in range(4):
150     screen.blit(
151         source: face(colour_3d_array[i]), # gets the image of the face
152         # width / 2, height / 2, i - 2 are for centering, -2 as 4 faces total
153         # * 200 for moving past the previous face and 30 spacing
154         # - 170 / 2 for centering
155         dest: validate_screen_positions(
156             (int(width / 2) + 10 + ((i - 2) * 200)), int(height / 2 - 170 / 2)
157         ),
158     )
159
160     screen.blit(
161         source: face(colour_3d_array[4]),
162         dest: validate_screen_positions(
163             (int(width / 2 - 200 + 10), int(height / 2 - 170 - 170 / 2 - 20) - 10)
164         ),
165     )
166     screen.blit(
167         source: face(colour_3d_array[5]),
168         dest: validate_screen_positions(
169             (int(width / 2 - 200 + 10), int(height / 2 + 170 / 2 + 20) + 10)
170         ),
171     )
172

```

This section serves to create and display a simple cube, although this one is not 3D.

In this section I have coded a function ‘cube’ that will blit the image of the cube onto the screen. The function has ‘colour_3d_array’ as a parameter, it is a 3D array of RGB tuples, wanting either ‘used_cube’ or ‘default_cube’ to be given. It works by breaking the 3D array down into single elements, creating the necessary image, and then recombining these images to form a cube. To do this I have created 3 local functions with ‘cube’: ‘square’, ‘row’, and ‘face’.

‘square’ takes in a single RGB colour and returns a 50 pixel by 50 pixel pygame image that is filled with that colour.

‘row’ takes in a list of 3 RGB tuples, creates an image and blits 3 squares to it in a row, with 10 pixels spacing between them, where the colours of the squares are taken from the list in order. It then returns the created image.

‘face’ takes in a 2D array of RGB tuples as a parameter, and for each list it creates a row. These rows are stacked vertically with 10 pixels spacing. It then returns this created image.

As this is a net design, 4 of the faces are in a line with a fixed y value. Therefore, I have used a loop to place these faces. The loop blits the images returned by the ‘face’ function, using the *i* iterator to give the correct 2D array from the ‘colour_3d_array’ as a parameter.

The iterator is also used for changing the x position of the images: width/2 centres the images, +10 is used for manually adjusting the position of the image to look more centred, i-2 ensures two faces go to the left of the centre and two go to the right, so the image is fully centred, and *200 ensures faces aren't placed on top of each other, as well as adding 30 pixels spacing. The y axis is fixed: height/2 places the images at the centre of the screen, and -2/170 removes half of the height of the faces, as the position given where the top left of the image is placed, not the centre.

The up and down faces were placed without a loop. Whilst a loop could be used, I felt this would unnecessary complicate something that can easily be achieved in a few lines of code. Like in the loop, the 'face' function is called and the image it returns blitted. For the x position the same logic as in the loop applies, with the iterator equalling 1 as this lines up the net design. For the y position height/2 centres the image, -170 moves the image up above the row of 4 images, + or - 170/2 accounts for the fact the position is of the top left of the image, + or - 20 is for spacing between the faces, and + or - 10 is for manual adjustment of the images.

```

173
174     def rotate(row_col, number, backwards=False):
175         """
176             Turn 1 row or column once
177             :type row_col: bool
178             :type number: int
179             :type backwards: bool
180             :param row_col: True=row, False=column
181             :param number: the number to do, left to right or top to bottom
182             :param backwards: do the opposite of the move/do the move 3 times if true
183             """
184
185         loop = 1
186         if backwards: # 3 right is used to achieve 1 left, 3 up to achieve 1 down
187             loop = 3
188         for j in range(loop):
189             face0 = copy.deepcopy(used_cube[0])
190             face1 = copy.deepcopy(used_cube[1]) # prevents pass by reference shenanigans
191             face2 = copy.deepcopy(
192                 used_cube[2]
193             ) # copy the value instead of creating a reference
194             face3 = copy.deepcopy(used_cube[2])
195             face3 = copy.deepcopy(used_cube[3])
196             face4 = copy.deepcopy(used_cube[4])
197             face5 = copy.deepcopy(used_cube[5])
198
199             n = number
200
201             if row_col: # rotate row
202                 used_cube[0][n], used_cube[1][n], used_cube[2][n], used_cube[3][n] = (
203                     face1[n],
204                     face2[n],
205                     face3[n],
206                     face0[n],
207                 )
208                 if number == 0: # rotate the top face
209                     used_cube[4] = numpy.rot90(m=used_cube[4], k=1, axes=(1, 0))
210                 elif number == 2: # rotate the bottom face
211                     used_cube[5] = numpy.rot90(m=used_cube[5], k=1, axes=(0, 1))
212             else: # column
213                 for i in range(3): # each loop only moves 1 square
214                     used_cube[1][i][n] = face5[i][n]
215                     # 2-i flips the row number for the back
216                     # 2 - n flips the column number for the back
217                     used_cube[5][2 - i][n] = face3[i][2 - n]
218                     used_cube[3][2 - i][2 - n] = face4[i][n]
219                     used_cube[4][i][n] = face1[i][n]
220
221                 if number == 0: # rotate left face
222                     used_cube[0] = numpy.rot90(m=used_cube[0], k=1, axes=(1, 0))
223                 elif number == 2: # rotate right face
224                     used_cube[2] = numpy.rot90(m=used_cube[2], k=1, axes=(1, 0))

```

This section allows interactions with the cube and is what must be correct to ensure the cube has the correct logic.

In this section of code, I have created a function to handle rotating each section of the cube. This required lots of iterative testing due to the complicated nature of Rubik's cube logic, and due to me using a trial-and-error approach to figuring out how the logic works. The procedure takes in Boolean 'row_col', int 'number' and Boolean 'backwards'. 'row_col' indicates whether the turn is to be done to a row (True) or column (False). 'number' indicates which row/column number needs to be done, 0, 1 or 2 along the face. 'backwards' allows for the reverse of a turn to be done.

First, I create a 'loop' variable set to one, then change it to three if 'backwards' is true. Everything in this function following this is in a loop that iterates based on the 'loop' variable. This is because a Rubik's cube works with 90 degree turns, so turning backwards is the same as turning forwards 3 times.

I then create a deep copy of each face. This is done so these new variables can serve as temp variables, allowing me to change each face without risking losing what the original position of the face was. A deep copy was used as it copies by value, instead of reference, which ensures these temp variables are not changed when the actual cube array is changed.

Next, I create a variable n equal to $number$ as this is quicker to type. I did not want to change the parameter name to n as this would be unclear if read at a later date by someone wanting to use the function.

After that, I have an if statement to check if the operation is to be done to a row or a column. In the row section, I update the horizontal section of 'used_cube' by changing each face to the one after it. I then check if the rotation was done to the top or bottom row and rotate the top or bottom face if they were, as this is what happens with a real Rubik's cube. For the column section, I used a loop to move each square one by one. This helps when it comes to reversing the order of the row and the column when the column rotates to/from the back face, which is achieved by the 2-l and the 2-n. It also allows for the row to be iterated. Then, similar to the row rotation, I check if the column being turned is one of the edge ones and rotate the left or right face.

```

226
227     pygame.init()
228
229     # game loop
230     while True:
231         for event in pygame.event.get():
232             if event.type == pygame.QUIT:
233                 pygame.quit()
234             if event.type == pygame.KEYDOWN:
235                 # row left
236                 if event.key == pygame.K_y:
237                     rotate( row_col: True, number: 0)
238                 elif event.key == pygame.K_h:
239                     rotate( row_col: True, number: 1)
240                 elif event.key == pygame.K_n:
241                     rotate( row_col: True, number: 2)
242                 # row right
243                 elif event.key == pygame.K_t:
244                     rotate( row_col: True, number: 0, backwards: True)
245                 elif event.key == pygame.K_g:
246                     rotate( row_col: True, number: 1, backwards: True)
247                 elif event.key == pygame.K_b:
248                     rotate( row_col: True, number: 2, backwards: True)
249                 # column up
250                 elif event.key == pygame.K_q:
251                     rotate( row_col: False, number: 0)
252                 elif event.key == pygame.K_w:
253                     rotate( row_col: False, number: 1)
254                 elif event.key == pygame.K_e:
255                     rotate( row_col: False, number: 2)
256                 # column down
257                 elif event.key == pygame.K_a:
258                     rotate( row_col: False, number: 0, backwards: True)
259                 elif event.key == pygame.K_s:
260                     rotate( row_col: False, number: 1, backwards: True)
261                 elif event.key == pygame.K_d:
262                     rotate( row_col: False, number: 2, backwards: True)
263
264         screen.fill(BLACK)
265
266         cube(used_cube)
267
268         pygame.display.flip()
269

```

In this final section of code, I have the main game loop. I first initialise pygame, then I create a while True loop. This loop will always run, only stopping when it encounters an error from trying to execute a pygame command for a pygame window that has been quit.

The majority of this loop is checking for events – the events list is gotten then looped through, and I check for different types of events. If the event is the quit button being pressed, pygame is quit and the program finishes. Every other event is for rotating the cube.

Then I fill the screen with the colour black to create a black background, then call the ‘cube’ function which will blit the cube net image. Then the display is refreshed so the actual image displayed is updated.

Testing

I am only going to test the implemented features.

Test No.	What is being tested	Description	Method	Expected Output	Pass/Fail
2	Cube turn – vertical, left, up	Executing the turns function with the correct parameters for the given turn should result in the leftmost column being rotated upwards.	Execute the turns function with the parameters for the turn. Run the cube display, the display should show the new cube state. The same rotation should be done to a real Rubik’s cube.	The image of both cubes should be exactly the same.	Pass
3	Cube turns – vertical, middle, up	Executing the turns function with the correct parameters for the given turn should result in the middle column being rotated upwards.			Pass
4	Cube turns – vertical, right, up	Executing the turns function with the correct parameters for the given turn should result in the rightmost column being rotated upwards.			Pass
5	Cube turns – vertical, left, down	Executing the turns function with the correct parameters for the given turn should result in the leftmost column being rotated downwards.			Pass

6	Cube turns – vertical, middle, down	Executing the turns function with the correct parameters for the given turn should result in the middle column being rotated downwards.			Pass
7	Cube turns – vertical, right, down	Executing the turns function with the correct parameters for the given turn should result in the rightmost column being rotated downwards.			Pass
8	Cube turns – horizontal, top, right	Executing the turns function with the correct parameters for the given turn should result in the upper row being rotated right.			Pass
9	Cube turns – horizontal, middle, right	Executing the turns function with the correct parameters for the given turn should result in the middle row being rotated right.			Pass
10	Cube turns – horizontal, bottom, right	Executing the turns function with the correct parameters for the given turn should result in the lower row being rotated right.			Pass

11	Cube turns – horizontal, top, left	Executing the turns function with the correct parameters for the given turn should result in the upper row being rotated left.			Fail
12	Cube turns – horizontal, middle, left	Executing the turns function with the correct parameters for the given turn should result in the middle row being rotated left.			Fail
13	Cube turns – horizontal, bottom, left	Executing the turns function with the correct parameters for the given turn should result in the lower row being rotated left.			Fail

Improvements

Tests 11, 12 and 13 should have passed, but did not. I mistakenly used keys y, h, n instead of the keys r, f, v, which are what I chose to use as the keys for those turns. The function itself worked correctly.

```

234 if event.type == pygame.KEYDOWN:
235     # row left
236     if event.key == pygame.K_y:
237         rotate( row_col: True, number: 0)
238     elif event.key == pygame.K_h:
239         rotate( row_col: True, number: 1)
240     elif event.key == pygame.K_n:
241         rotate( row_col: True, number: 2)

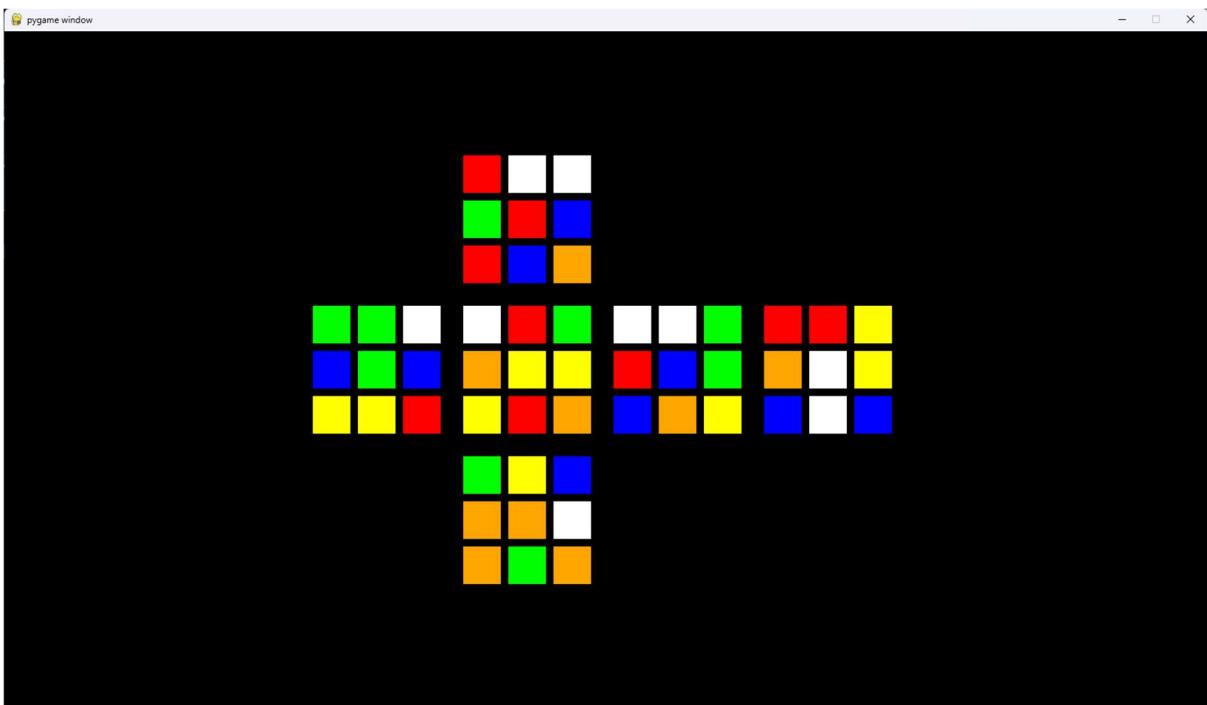
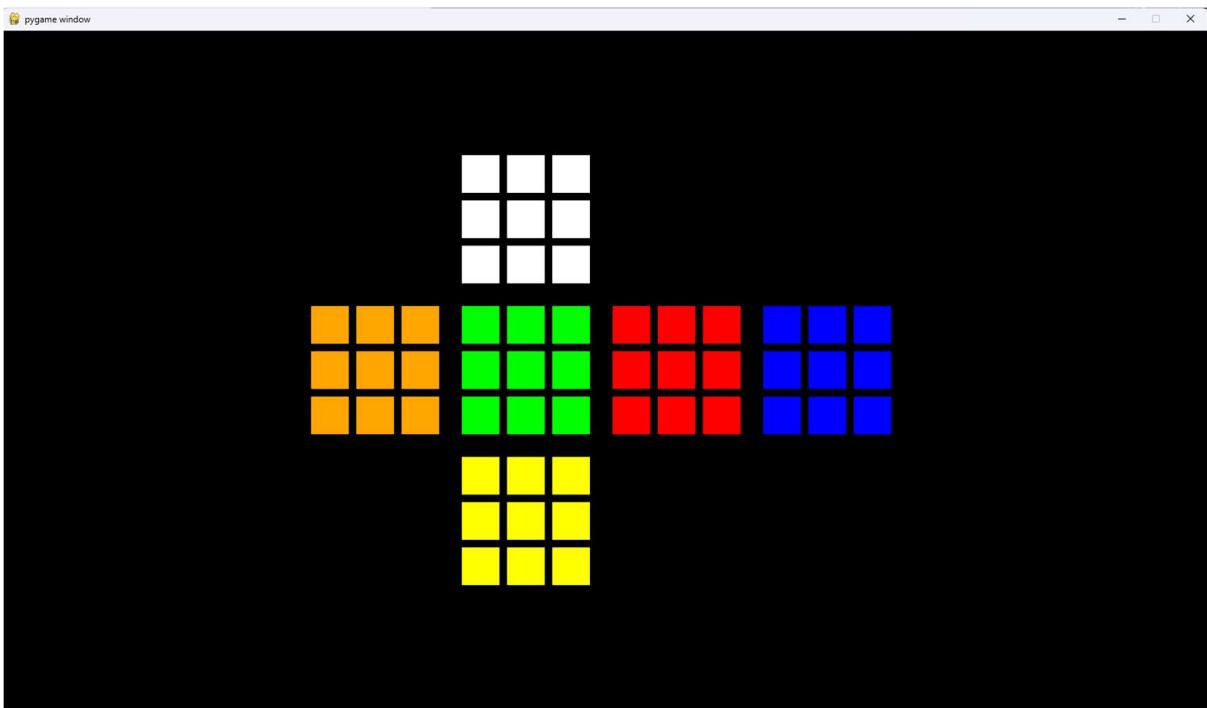
```

```

if event.type == pygame.KEYDOWN:
    # row left
    if event.key == pygame.K_r:
        rotate( row_col: True, number: 0)
    elif event.key == pygame.K_f:
        rotate( row_col: True, number: 1)
    elif event.key == pygame.K_v:
        rotate( row_col: True, number: 2)

```

Program Images



Prototype 2

Development

As the scope of prototype 2 was much greater and more complicated than prototype one, I initially split the code into 4 files: main – which is the file to run, and uses pygame to display the cube and features, and to record user inputs; cube – which manages any interactions with the cube, including getting the image; interface – which handles creating complicated visual elements to be displayed such as text and buttons; validation – which handles validating screen positions and anything I may find that needs validation. However, during development, I realised that I was copying some constants and passing the same variable repeatedly through all the files, so I created a data file to store these commonly used pieces of data. During this I also realised I had made some mistakes with the format of my docstrings, so I also fixed those, and some were unclear, so I fixed that.

Cube display:

- 3D methods [research](#)
- [Cube class](#)
- Cube [object](#) and screen creation
- Display [images](#)

Cube Logic:

- Data [storage](#)
- [Turns](#)
- [Rotations](#)

Scramble:

- [Function](#)
- [Usage](#)

Solver:

- [Storage](#) (added to by [Turns](#) and [Rotations](#))
- [Class](#)
- [Object](#)
- [Usage](#)

Hint:

- [Function](#) (of Solver)
- [Usage](#)

Timer:

- [Class](#)

- [Object](#)
- [Automatic start](#)
- [Automatic stop](#)
- [Usage](#)

Guide:

- [Class](#)
- [Object](#)
- [Usage](#)

Validation:

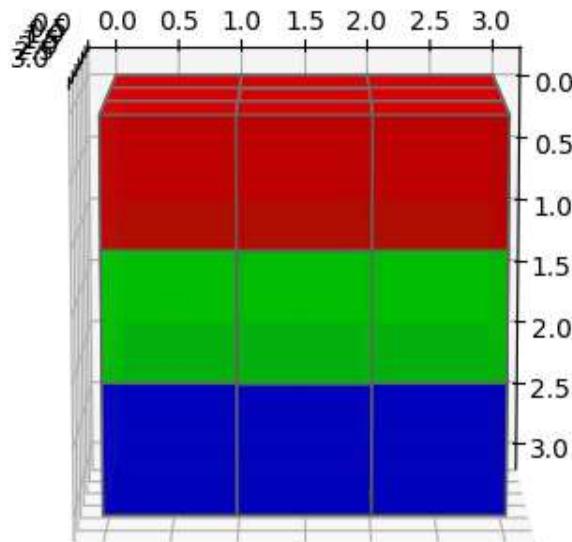
- [Class](#)
 - [Object](#)
 - Usage can be seen throughout [Main](#)
-
- [Ignore invalid key presses](#)
 - [Ignore cube interactions when on guide](#)

Cube method research

The first goal of my prototype two – past the fixes specified in review, was to have a 3D Rubik's cube. To achieve this, I first had to decide of a method of displaying a 3D cube, as I had not decided on an exact method to use during development, although I knew it was likely I would use pygame, I couldn't be completely confident until I tried coding it to see what worked.

```
 1 import matplotlib.pyplot as plt
 2 from mpl_toolkits.mplot3d import Axes3D
 3 import numpy
 4
 5 # create axis
 6 axes = [3, 3, 3] # dimesions
 7
 8 # create data
 9 data = numpy.ones(axes)
10
11 # control transparency
12 alpha = 0.9
13
14 # colour
15 colours = numpy.empty(axes + [4])
16
17 colours[0] = [1, 0, 0, alpha] # red
18 colours[1] = [0, 1, 0, alpha] # green
19 colours[2] = [0, 0, 1, alpha] # blue
20 #colours[3] = [1, 1, 0, alpha] # yellow
21 #colours[4] = [1, 1, 1, alpha] # grey
22
23 ##colours = numpy.empty(axes + [4], dtype=numpy.float32)
24 ##
25 ##colours[:] = [1, 0, 0, alpha] # red
26
27 # plot figure
28 fig = plt.figure() # create figure
29 ax = fig.add_subplot(111, projection='3d') # add 3d axis
30
31 # voxels is used to customise of the
32 # sizes, positions and colors.
33 ax.voxels(data, facecolors=colours, edgecolors='grey')
34
35 # it can be used to change the axes view
36 ax.view_init(100, 0)
37
38
39 plt.show()
40
```

Figure 1

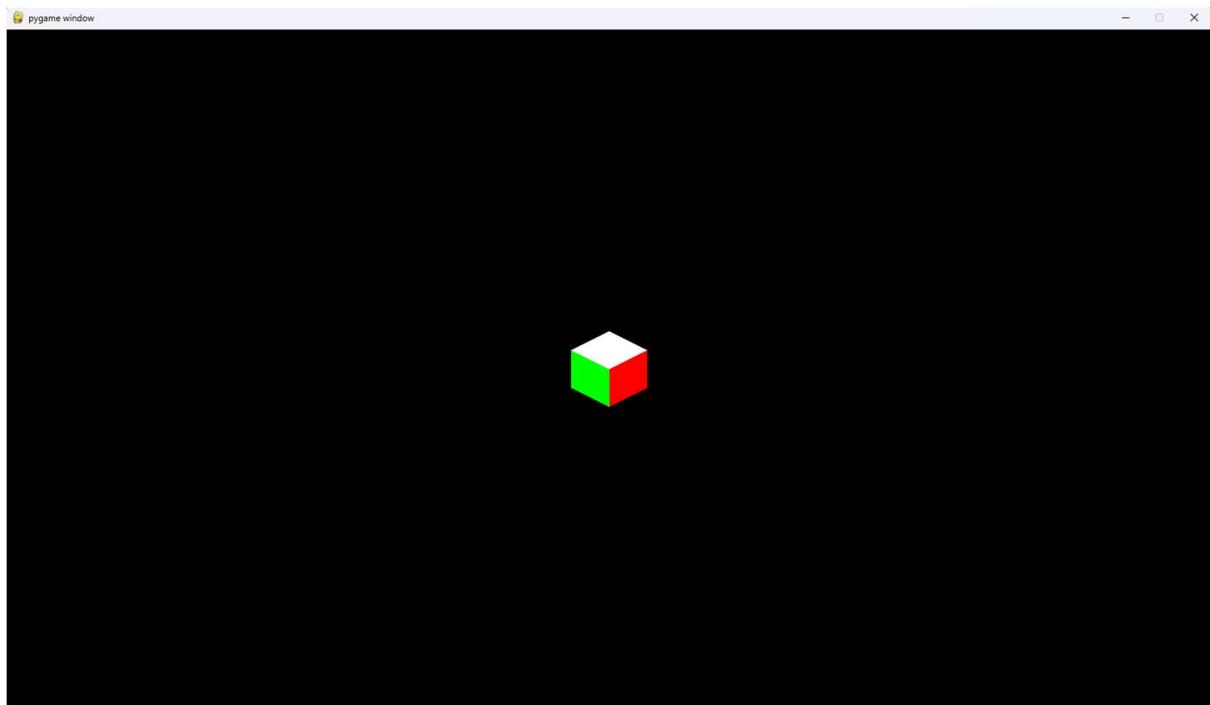


I looked into many options, one of which used matplotlib. I found and modified this code from [geeksforgeeks](https://www.geeksforgeeks.org/how-to-draw-3d-cube-using-matplotlib-in-python/) (<https://www.geeksforgeeks.org/how-to-draw-3d-cube-using-matplotlib-in-python/>) to get a look into how it might function, however I found I wasn't happy with the results.

```

62 import pygame
63
64 WHITE = (255, 255, 255)
65 GREEN = (0, 255, 0)
66 RED = (255, 0, 0)
67 BLUE = (0, 0, 255)
68
69 screen = pygame.display.set_mode((1600, 900))
70
71 # right = pygame.draw.polygon(screen, RED, (
72 #     (800, 450),
73 #     (850, 825),
74 #     (800, 750),
75 #     (850, 775)
76 # ))
77
78
79 def square():
80     surf = pygame.Surface((50, 50)) # (50, 25),
81
82
83 while True:
84     for event in pygame.event.get():
85         if event.type == pygame.QUIT:
86             pygame.quit()
87
88     right = pygame.draw.polygon(
89         surface=screen,
90         color=RED,
91         points=(
92             (800, 450),
93             (850, 425),
94             (850, 475),
95             (800, 500),
96         ),
97     )
98     left = pygame.draw.polygon(
99         surface=screen, color=GREEN, points=((800, 450), (750, 425), (750, 475), (800, 500))
100 )
101     top = pygame.draw.polygon(
102         surface=screen, color=WHITE, points=((800, 450), (850, 425), (800, 400), (750, 425))
103 )
104
105     # line = pygame.draw.line(screen, BLUE, (0, 500), (1600, 5))
106
107     pygame.display.flip()
108

```



In the end I did decide to use pygame, using stretched and angles cubes to give the impression of it being 3D, manually drawing them, without using any complicated 3D libraries. As I coded this myself, I was confident in how it worked and therefore comfortable expanding upon it to create a full Rubik's cube.

Main

```
1 """
2 This is the file to run to execute the program
3
4 This file handles the main loop of the program, it gets images and data from the other
5 files and displays them. It also handles user input within the game loop.
6
7 black, isort and flake8 used for formatting
8 """
9
10 import time
11
12 import cube
13 import interface
14 import pygame
15 from data import *
16 from validation import ValidateScreenPositions
17
18 # window
19 pygame.init()
20 width = 1600
21 height = 900
22 screen = pygame.display.set_mode(size: (width, height), flags: pygame.RESIZABLE)
23
24 # validation
25 val = ValidateScreenPositions(width: width, height: height)
26
27 # cubes
28 cube_net = cube.CubeNet(surface: screen, pos: val.run((width // 2, height // 2)))
29 cube_3d = cube.Cube3D(surface: screen, pos: val.run((width // 2, height // 2)))
30 cube_guide = cube.CubeGuide(surface: screen, pos: val.run((width // 2, height // 2)))
31
```

As I knew I had made mistakes (albeit small ones) when it came to documentation in prototype one, I endeavoured to have perfect documentation this time. As such, the main file (along with every other file) starts with a doc-string. Then, like in prototype one, importing the necessary libraries, although this time I'm also importing my own files. I chose to import all from data as this is the main file, and as such will end up needing most of the data.

I have also allowed the screen window to be resized by the user, however, most of the calculations to place images on the surface are only done once and will be done before the has the chance to change the screen size, resulting in changing screen size making little difference for the user.

I then create 3 objects – one of each cube type – from the imported cube classes from cube file. These are what draw the image of the cube to the screen, at the position specified, when their update method is ran.

The aforementioned positions are validated by being run through the ValidateScreenPositions class, or more specifically, the class's run function. This ensures that the positions are within the bounds of the screen and a 4k resolution. Every other screen position in the main file is also validated this way.

```
32
33 class Buttons: 10 usages
34 """
35     This class handles the rendering of the buttons
36
37     This class is largely self-contained, the only usage should be to run
38     .update as this automatically updates the buttons
39 """
40
41 cube_option = interface.DisplayOption(
42     image_function: lambda: cube_3d.get_image(),
43     display_surf: screen,
44     pos: val.run([10, 0]),
45     size: [100, 100],
46     mult: 1.5,
47     action: lambda: Buttons.display_swap("3d"),
48     bg_col: default_colour,
49 )
50 net_option = interface.DisplayOption(
51     image_function: lambda: cube_net.get_image(),
52     display_surf: screen,
53     pos: val.run([10, 100]),
54     size: [100, 100],
55     mult: 1.5,
56     action: lambda: Buttons.display_swap("net"),
57     bg_col: default_colour,
58 )
59 guide_option = interface.DisplayOption(
60     image_function: lambda: cube_guide.get_image(),
61     display_surf: screen,
62     pos: val.run([10, 200]),
63     size: [100, 100],
64     mult: 1.5,
65     action: lambda: Buttons.display_swap("guide"),
66     bg_col: BLACK,
67 ) # should be default colour,
68 # but this causes the background of the hovered button to be black.
69 # May be an error with pygame.smoothscale in interface file
70 # this works as a solution
71 cube_option_bar = interface.DisplayBar( # update with any new options
72     object_list: [cube_option, net_option, guide_option], row: False
73 )
74 display_option = "3d"
```

```

75
76     @staticmethod 3 usages
77     def display_swap(option):
78         """
79             Updates display_option variable within the class
80
81             This provides a function for interface.DisplayOption objects
82             to update the display_option variable which is saved with this class
83
84             :param option: the new display_option: 3d, net or guide
85             :type option: str
86             """
87
88         Buttons.display_option = option
89
90     @staticmethod
91     def update(mouse_pos, mouse_up):
92         """
93             Updates each button in the class
94
95             :param mouse_pos: the x,y position of the mouse
96             :param mouse_up: whether the mouse button has been clicked
97             :type mouse_pos: tuple[int, int] or list[int, int]
98             :type mouse_up: bool
99             :rtype: None
100
101        Buttons.cube_option_bar.update( mouse_pos=mouse_pos, mouse_up=mouse_up)

```

Next, I have created a class that manages the buttons for swapping between the different types of cubes that can be displayed (3D, net, guide). As this class only has one purpose and only serves to encapsulate the buttons, it has no constructor, and the two functions are static, meaning that they do the exact same thing regardless of the state of the object they are in.

The buttons are object of the class `DisplayOption` from the interface file, and they are placed in a list that is a parameter to `DisplayBar`, which was specifically designed to work with objects of `DisplayOption`. The buttons are placed in a column in the top left corner of the screen, and this position is validated. The buttons are given the `get_image` function of their respective cubes and a size of [100, 100], meaning the image they display is what just a shrunk-down version of what will actually be displayed to the screen as the main cube. When a button is hovered, it will be enlarged by 1.5x in each direction, and the `cube_bar` will account for this and move the images to ensure they do not overlap.

The update function is somewhat unneeded, as running `Buttons.update()` has the exact same effect as running `Buttons.cube_option_bar.update()`, however I felt it resulted in cleaner code and helped abstract away unneeded elements.

```
102  
103     # used for solving the cube  
104     solve_cube = False  
105     solver = cube.Solver()  
106  
107     timer = cube.Timer()  
108
```

Then I have defined a variable for whether or not the cube is actively being solved, and 2 for the objects of Solver and Timer respectively.

```

109 # game loop
110 while True:
111     mouse_pos = pygame.mouse.get_pos()
112     mouse_up = False
113     val.update_size(pygame.display.get_surface().get_size())
114
115     for event in pygame.event.get():
116         if event.type == pygame.QUIT:
117             pygame.quit()
118         elif event.type == pygame.MOUSEBUTTONUP:
119             mouse_up = True
120         # prevent any moves made whilst on guide cube
121         elif event.type == pygame.KEYDOWN and Buttons.display_option != "guide":
122             # row right
123             if event.key == pygame.K_t:
124                 cube.turn( row_col: True, number: 0)
125             elif event.key == pygame.K_g:
126                 cube.turn( row_col: True, number: 1)
127             elif event.key == pygame.K_b:
128                 cube.turn( row_col: True, number: 2)
129             # row left
130             elif event.key == pygame.K_r:
131                 cube.turn( row_col: True, number: 0, backwards: True)
132             elif event.key == pygame.K_f:
133                 cube.turn( row_col: True, number: 1, backwards: True)
134             elif event.key == pygame.K_v:
135                 cube.turn( row_col: True, number: 2, backwards: True)
136
137             # column up
138             elif event.key == pygame.K_q:
139                 cube.turn( row_col: False, number: 0)
140             elif event.key == pygame.K_w:
141                 cube.turn( row_col: False, number: 1)
142             elif event.key == pygame.K_e:
143                 cube.turn( row_col: False, number: 2)
144             # column down
145             elif event.key == pygame.K_a:
146                 cube.turn( row_col: False, number: 0, backwards: True)
147             elif event.key == pygame.K_s:
148                 cube.turn( row_col: False, number: 1, backwards: True)
149             elif event.key == pygame.K_d:
150                 cube.turn( row_col: False, number: 2, backwards: True)
151
152             # rotations
153             elif event.key == pygame.K_x:
154                 cube.rotate("x")
155             elif event.key == pygame.K_y:
156                 cube.rotate("y")
157             elif event.key == pygame.K_z:
158                 cube.rotate("z")
159
160             elif event.key == pygame.K_k: # solve
161                 solve_cube = True
162                 if timer.running:
163                     timer.delete()
164             elif event.key == pygame.K_m: # scramble
165                 cube.scramble()
166                 timer.start() # start timer
167             elif event.key == pygame.K_h: # hint
168                 solver.pop_move()

```

I then start the game loop. At the start if the game loop I update some key information – mouse_pos and mouse_up, as these are important for the event checks that are about to happen. Mouse up is set to False in case it was being pressed last game loop but isn't anymore, and it will be one of the first events checked so it can be updated.

I also update the screen size for the validation object in case the user has resized the screen, but as mentioned earlier this has little effect on what and where things are displayed. However, adding this update in now ensures that I don't forget it if I add more support for resizing the screen at a later date.

I iterate through all the new events, checking for expected inputs. The 'and ...' portion of line 121 ensures that no cube rotations or turns happen when the cube guide is being displayed, as the cube guide displays the default cube image which isn't supposed to be changed. I have updated the keys used for left turns so that they now match what is expected. In the portion for the solve key being pressed the timer is deleted as a completion time shouldn't be awarded for using the solve function. The scramble portion starts the timer as the timer is supposed to start automatically and be as easy to use as possible. The hint feature simply uses the solve function that gets the next move to do, as the hint feature is supposed to just do one move.

```

169
170     screen.fill(default_colour) # background colour
171
172     if solve_cube:
173         # ensures each solve take 5 sections, assuming no hardware limitations
174         time.sleep(solver.sleep_time)
175         solve_cube = solver.solve() # solves one move
176     else:
177         solver.first = True # so next solve it is set to true
178
179     if timer.running and solver.check_solved(): # ends timer on solve
180         timer.stop()
181
182     if Buttons.display_option == "3d":
183         display_cube = cube_3d
184     elif Buttons.display_option == "net":
185         display_cube = cube_net
186     elif Buttons.display_option == "guide":
187         # also prevents cube interact as uses default
188         display_cube = cube_guide
189         # actions text
190         screen.blit(
191             source: interface.text(
192                 text="Scramble: M",
193                 font=guide_font,
194                 foreground_colour=BLACK,
195                 background_colour=default_colour,
196             ),
197             dest: val.run((1100, 300)),
198         )
199         screen.blit(
200             source: interface.text(
201                 text="Solve: K",
202                 font=guide_font,
203                 foreground_colour=BLACK,
204                 background_colour=default_colour,
205             ),
206             dest: val.run((1100, 350)),
207         )
208         screen.blit(
209             source: interface.text(
210                 text="Hint: H",
211                 font=guide_font,
212                 foreground_colour=BLACK,
213                 background_colour=default_colour,
214             ),
215             dest: val.run((1100, 400)),
216         )
217         display_cube.update() # actually update cube
218
219     if timer.exists: # display timer
220         screen.blit( source: timer.display_elapsed(), dest: val.run((1400, 200)))
221         timer.update()
222
223     # update buttons
224     Buttons.update( mouse_pos: mouse_pos, mouse_up: mouse_up)
225
226     pygame.display.flip()
227
228     pygame.quit()

```

In the final part of the game loop, I get and draw all the images to the screen. The if solve_cube section works to do a single move of the solve and wait the calculated amount of time between each move. The else ensures that after each solve the first attribute of solve is reset to True, for the next solve.

The if and elifs on lines 181, 184, and 186 change display_cube to the correct object so that display_cube.update() only has to be written once. The guide portion also writes additional text to the screen. Ideally, this should be a part of the guide_image, but as the classes responsible for the images place the image based on a centre position, adding them would change the position of the cube so it would no longer be centred. As I want to avoid this, I have simply added the text directly to the screen like this.

Validation

```
1  """
2  This file contains validation functions and error handling
3
4  black, isort and flake8 used for formatting
5  """
6
7  import time
8
9
10 class InvalidScreenPosition(Exception):
11     """This exception is raised when the screen position is invalid"""
12
13     def __init__(self, pos):
14         """
15             :param pos: the x,y position that is invalid
16             :type pos: tuple[int, int] or list[int]
17         """
18         super().__init__(f"Invalid Screen Position: {pos}")
19         with open("error.txt", "a") as f:
20             error_time = time.time()
21             f.write(f"{error_time} Invalid Screen Position: {pos}\n")
22
```

In a large project, particularly one that will be used by others, error handling is very important of the code, ensuring that any unexpected errors are recorded so that they can be fixed, and so that anyone working with the code knows what went wrong and what to avoid doing. Important sections of code would be within try except statements to handle any errors that do occur.

As my code is relatively simple and static (there is not much that can be changed, so if it works once, it will likely always work), this is far less important for me. However, I wanted to learn more about error handling, and there is nothing to prevent me from using this file in other projects I

end up working on, so I decided to do validation in the correct manner. As such, I have created a custom exception for invalid screen positions, which can be raised as an exception, and will document itself in an error file with a timestamp and the position that caused the error, so that if one does occur, I can hopefully identify what caused it and fix it.

```

23
24     class ValidateScreenPositions: 2 usages
25
26         """
27             This class contains a screen position validation function
28
29             It will ensure a screen position is valid based on a 4k resolution screen
30             and the size of the window being displayed to
31         """
32
33     def __init__(self, width, height):
34
35         """
36             :param width: the width of the screen window
37             :param height: the height of the screen window
38             :type width: int
39             :type height: int
40         """
41
42     def run(self, pos):
43
44         """
45             Ensures a position is within the confines of the screen and 4k resolution
46
47             This function will throw an error if the position is invalid.
48             An invalid position is one that is outside the confines of the screen based
49             width and height, or a 4k resolution screen.
50
51             :param pos: the x,y position to check
52             :type pos: tuple[int, int] or list[int]
53             :return: the x,y screen position if it is valid, else raises an exception
54             :rtype: tuple[int, int] or list[int]
55         """
56
57         if (
58             pos[0] < 0
59             or pos[0] > self.width
60             or pos[0] > 3840
61             or pos[1] < 0
62             or pos[1] > self.height
63             or pos[1] > 2160
64         ):
65             raise InvalidScreenPosition(pos)
66         else:
67             return pos
68
69     def update_size(self, size): 1 usage
70
71         """
72             Update the screen width and height
73
74             :param size: the width and height of the window
75             :type size: tuple[int, int] or list[int]
76         """
77
78         self.width = size[0]
79         self.height = size[1]

```

I've also updated the validation function to check if it's within the expected screen size, although I kept the 4k resolution checks in case the screen size ends up incorrect. I also changed it to raise the exception instead of printing an error message and quitting the program.

To prevent having to pass the width and height of the screen each time the validation function is run, I made it into a class that stores the width and height of the screen, and created a function to update that information should it change.

Cube

```
1  """  
2  This file contains the code for the cube as well as game features that reference the cube  
3  
4  This file handles all the data to do with the cube: its state, moves made, etc.  
5  It also handles features that make use of the cube such as the solver and timer.  
6  
7  black, isort and flake8 used for formatting  
8  """  
9  
10 import copy  
11 import time  
12 from random import randint  
13  
14 import data  
15 import interface  
16 import numpy  
17 import pygame  
18  
19 # colours  
20 from data import BLACK, BLUE, GREEN, GREY, ORANGE, RED, WHITE, YELLOW, default_colour  
21
```

I first import all the required libraries and files, including all the colour constants I have defined in the data file.

```

22     # cube design
23     # split into sides as easier to write
24     up = [
25         [WHITE, WHITE, WHITE],
26         [WHITE, WHITE, WHITE],
27         [WHITE, WHITE, WHITE],
28     ]
29     down = [
30         [YELLOW, YELLOW, YELLOW],
31         [YELLOW, YELLOW, YELLOW],
32         [YELLOW, YELLOW, YELLOW],
33     ]
34
35     left = [
36         [ORANGE, ORANGE, ORANGE],
37         [ORANGE, ORANGE, ORANGE],
38         [ORANGE, ORANGE, ORANGE],
39     ]
40
41     right = [
42         [RED, RED, RED],
43         [RED, RED, RED],
44         [RED, RED, RED],
45     ]
46
47     front = [
48         [GREEN, GREEN, GREEN],
49         [GREEN, GREEN, GREEN],
50         [GREEN, GREEN, GREEN],
51     ]
52
53     back = [
54         [BLUE, BLUE, BLUE],
55         [BLUE, BLUE, BLUE],
56         [BLUE, BLUE, BLUE],
57     ]
58
59     # so a default cube may always be shown and to check against for solves
60     default_cube = [
61         left,
62         front,
63         right,
64         back,
65         up,
66         down,
67     ]
68     # deepcopy passes by value, not reference, ensuring default_cube is not changed
69     used_cube = copy.deepcopy(default_cube)
70
71     # used for tracking moves and 'solving' the cube
72     moves = []
73

```

I then create the cube and the copy, as well as create a moves list that will store all the moves made. This is vital for the Solve feature, as it works by undoing all of the completed moves. And for the hint feature, as that uses the solve function.

```

74
75     @class Cubenet: 2 usages
76     """Handles the display of the cube as a net to a fixed position on the screen"""
77
78     def __init__(self, surface, pos):
79         """
80             :param surface: The surface that this cube is to be blitted to
81             :param pos: The centre position that this cube is to be blitted to: x,y
82             :type surface: pygame.Surface
83             :type pos: list[int] or tuple[int, int]
84         """
85
86         # pos is centre
87         self.screen = surface
88         self.pos = pos
89
90     def update(self):
91         """
92             Updates the cube image and re-blits it to the surface
93
94             :rtype: None
95         """
96
97         image = self.get_image()
98         self.screen.blit(source=image, dest=image.get_rect(center=self.pos))
99
100    @staticmethod
101    def get_image(default=False):
102        """
103            Creates the image of the cube from the current state of the cube
104
105            :param default: if True, uses the default image instead of the current state
106            :type default: bool
107
108            :return: the image of the cube as a 720x540 surface
109            :rtype: pygame.Surface
110        """
111
112        surf = pygame.Surface((720, 540))
113        surf.fill(default_colour)
114        colour_3d_array = used_cube
115        if default:
116            colour_3d_array = default_cube
117
118        def square(colour):
119            """
120                Creates a single square with the given colour
121
122                :param colour: the RGB values of the colour
123                :type colour: tuple[int, int, int]
124
125                :return: the square image, 50x50
126                :rtype: pygame.Surface
127            """
128
129            surf = pygame.Surface((50, 50))
130            surf.fill(colour)
131
132            return surf

```

```

127     def row(colour_list):
128         """
129             Creates the image of a row of 3 squares
130
131             :param colour_list: List len(3) of tuples, where each tuple is an RGB value
132             :type colour_list: list[tuple[int, int, int]]
133             :return: the row image, 170*50
134             :rtype: pygame.Surface
135
136             surf = pygame.Surface((170, 50))
137             surf.fill(default_colour)
138             for i in range(3):
139                 # iterates alongside the list of colours,
140                 # getting a square with the respective colour and
141                 # blitting it to calculated position
142                 # i * 50 ensures the square is blitted after the previous one;
143                 # not inside it
144                 # i * 10 adds 10 spacing between the cubes
145                 surf.blit(source= square(colour_list[i]), dest=(i * 50 + i * 10, 0))
146
147             return surf
148
149     def face(colour_array):
150         """
151             Creates one face (side) from 3 rows
152
153             :param colour_array: 2D array (3x3)(row x col) of tuples,
154             # where each tuple is an RGB value
155             :type colour_array: list[list[tuple[int, int, int]]]
156             :return: the face image, 170*170
157             :rtype: pygame.Surface
158
159             surf = pygame.Surface((170, 170))
160             surf.fill(default_colour)
161             for i in range(3):
162                 # iterates alongside the list of rows,
163                 # getting and blitting the row image to calculated position
164                 # i * 50 ensures the row is placed beneath,
165                 # and not inside, the previous row
166                 # i * 10 is for spacing between the rows
167                 surf.blit(source= row(colour_array[i]), dest=(0, i * 50 + i * 10))
168
169             # 4 of the faces are placed next to each other so a loop can place them
170             for i in range(4):
171                 surf.blit(
172                     source= face(colour_3d_array[i]), # gets the image of the face
173                     # 180 * i includes 10 pixels spacing
174                     # placed 180 down to allow top to be placed above with 10 pixels spacing
175                     dest=(180 * i, 180),
176                 )
177
178             # 180 x val aligns with front face
179             surf.blit(source= face(colour_3d_array[4]), dest=(180, 0))
180             # 360 is below face image with 10 pixels spacing
181             surf.blit(source= face(colour_3d_array[5]), dest=(180, 360))
182
183             return surf

```

I then create a class for the net design of the cube, so that it can have an update function that will display the cube to the screen at the saved position. The get_image function is very similar to the cube function in prototype one, as it creates the same image. It is a static method as it creates the image based solely on the used_cube variable available in that file. I am able to use the used_cube variable like this as I have no intentions of having any cube states available past the state of the user's cube and the default image.

```
184
185  @4 class Cube3D(CubeNet): 2 usages
186      """
187      Handles the display of the 3d cube to a fixed position on the screen
188
189      This class is a child of CubeNet, only changing the get_image method
190      """
191
192      @staticmethod
193  @1@T def get_image(default=False):
194          """
195          Creates the image of the cube from its current state
196
197          :param default: if True, uses the default image instead of the current state
198          :type default: bool
199          :return: the cube image, 365*335
200          :rtype: pygame.Surface
201          """
202
203      surf = pygame.Surface((365, 335))
204      surf.fill(default_colour)
205      colour_3d_array = used_cube
206      if default:
207          colour_3d_array = default_cube
208
209      def right():
210          """
211              Draws the right face of the cube to the surf, it is a slanted square
212
213          :rtype: None
214          """
215
216      def square(colour):
217          """
218              Creates a slanted cube of solid colour
219
220              :param colour: RGB values
221              :type colour: tuple[int, int, int]
222              :return: the square image, 50*75
223              :rtype: pygame.Surface
224
225      surf = pygame.Surface((50, 75))
226      surf.fill(default_colour)
227      pygame.draw.polygon(surface=surf, color=colour, points=((0, 25), (50, 0), (50, 50), (0, 75)))
228      surf.set_colorkey(default_colour) # make background transparent
229      return surf
```

```

230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265

    def row(colour_list):
        """
        Creates a slanted row of 3 squares

        :param colour_list: List len(3) of tuples of RGB values
        :type colour_list: list[tuple[int, int, int]]
        :return: the row image, 165*135
        :rtype: pygame.Surface
        """

        surf = pygame.Surface((165, 135))
        surf.fill(default_colour)
        for i in range(3):
            # 55 includes 5 pixels spacing
            # -30 includes 5 pixels spacing
            surf.blit( source: square(colour_list[i]), dest: (55 * i, 60 - (30 * i)))
        surf.set_colorkey(default_colour)
        return surf

    def face(colour_array):
        """
        Stacks 3 row images to create a face of 9 squares

        :param colour_array: 2D (3x3)(row x col) array of tuples of RGB values
        :type colour_array: list[list[tuple[int, int, int]]]
        :return: the row image, 165*250
        :rtype: pygame.Surface
        """

        surf = pygame.Surface((165, 250))
        surf.fill(default_colour)
        for i in range(3):
            surf.blit( source: row(colour_array[i]), dest: (0, 55 * i))
        return surf

        # positioned to the right of front face with 5 pixels spacing
        surf.blit( source: face(colour_3d_array[2]), dest: (205, 90))

```

I then create a class for the 3D version of the cube, which inherits the CubeNet class, allowing me to reuse the constructor and update methods. The get image function is polymorphed, meaning that its name stays the same, but it works differently than the get image function in the parent class. In this case I have changed it so that it creates and returns a psudo-3D image, using slanted images to give the impression of a 3D cube.

It works by using subfunctions with subfunctions to break the image down into individual cubes, then slowly constructing it, into rows then a face. The surf.colorkey makes the background of the surface transparent, which is important as these slanted cubes can end up overlapping, and would break the image if the not for the transparency,

```

266     def front():
267         """
268             Draws the front face of the cube to the surf, it's a slanted square
269
270             :rtype: None
271         """
272
273     def square(colour):
274         """
275             Creates a slanted cube of solid colour
276
277             :param colour: RGB values
278             :type colour: tuple[int, int, int]
279             :return: the square image
280             :rtype: pygame.Surface
281         """
282
283         surf = pygame.Surface((50, 75))
284         surf.fill(default_colour)
285         pygame.draw.polygon( surface=surf, color=colour, points=((0, 0), (50, 25), (50, 75), (0, 50)))
286         surf.set_colorkey(default_colour)
287         return surf
288
289     def row(colour_list):
290         """
291             Creates the image of a row of 3 squares
292
293             :param colour_list: List len(3) of tuples of RGB values
294             :type colour_list: list[tuple[int, int, int]]
295             :return: the row image
296             :rtype: pygame.Surface
297         """
298
299         surf = pygame.Surface((165, 135))
300         surf.fill(default_colour)
301         for i in range(3):
302             # 55 includes 5 pixels spacing
303             # 30 includes 5 pixels spacing
304             surf.blit( source=square(colour_list[i]), dest=(55 * i, 30 * i))
305         surf.set_colorkey(default_colour)
306         return surf

```

```

306     def face(colour_array):
307         """
308             Stacks 3 row images to create a face of 9 squares
309
310             :param colour_array: 2D array (3x3)(row x col) of tuples of RGB values
311             :type colour_array: list[list[tuple[int, int, int]]]
312             :return: the face image
313             :rtype: pygame.Surface
314
315         surf = pygame.Surface((165, 250))
316         surf.fill(default_colour)
317         for i in range(3):
318             # 55 includes 5 pixels spacing
319             surf.blit( source=row(colour_array[i]), dest=(0, 55 * i))
320         surf.set_colorkey(default_colour)
321         return surf
322
323         # positioned below top of front face with 5 pixels spacing
324         surf.blit( source=face(colour_3d_array[1]), dest=(40, 90))
325

```

```

326     def top():
327         """
328             Draws the top face of the cube to the surf,
329             it is a horizontally stretched square
330
331         :rtype: None
332         """
333
334     def square(colour):
335         """
336             Creates a horizontally stretched cube of solid colour
337
338             :param colour: RGB values
339             :type colour: tuple[int, int, int]
340             :return: the square image
341             :rtype: pygame.Surface
342             """
343
344         surf = pygame.Surface((100, 50))
345         surf.fill(default_colour)
346         pygame.draw.polygon(
347             surface=surf, color=colour, points=((50, 0), (100, 25), (50, 50), (0, 25)))
348
349         surf.set_colorkey(default_colour)
350         return surf
351
352     def row(colour_list):
353         """
354             Creates the image of a row of 3 squares
355
356             :param colour_list: List len(3) of tuples of RGB values
357             :type colour_list: list[tuple[int, int, int]]
358             :return: the row image
359             :rtype: pygame.Surface
360             """
361
362         surf = pygame.Surface((215, 120))
363         surf.fill(default_colour)
364         for i in range(3):
365             # 55 includes 5 pixels spacing
366             # 30 includes 5 pixels spacing
367             surf.blit(source=square(colour_list[i]), dest=(55 * i, 30 * i))
368
369         surf.set_colorkey(default_colour)
370         return surf

```

```
368
369     def face(colour_array):
370         """
371             Stacks 3 row images to create a face of 9 squares
372
373             :param colour_array: 2D array (3x3)(row x col) of tuples of RGB values
374             :type colour_array: list[list[tuple[int, int, int]]]
375             :return: the face image
376             :rtype: pygame.Surface
377         """
378
379         surf = pygame.Surface((370, 315))
380         surf.fill(default_colour)
381         for i in range(3):
382             # 150 - to place bottom to top
383             # 55 includes 5 pixels spacing
384             # 30 includes 5 pixels spacing
385             surf.blit(source= row(colour_array[i]), dest=(150 - (55 * i), 30 * i))
386         surf.set_colorkey(default_colour)
387         return surf
388
389     surf.blit(source= face(colour_3d_array[4]), dest=(0, 0))
```

Unfortunately, despite the similarity of the right, front and top images they are not similar enough that using a single function instead could meaningfully decrease the amount of code required.

```
388     surf.blit(source= face(colour_3d_array[4]), dest=(0, 0))
389
390     right()
391     front()
392     top()
393
394     return surf
395
```

Finally, the 3 functions can be called to create the image.

```

395
396     class CubeGuide(Cube3D):
397         """ 1 usage
398
399         Class that handles the guide cube and adds instructions
400
401         This class inherits from Cube3D and overrides the get_image method
402         """
403
404     @classmethod
405     def get_image(cls):
406         """
407             Creates the image of the default cube with added instructions
408
409             :return: the cube image, 600*600
410             :rtype: pygame.Surface
411             """
412
413         surf = pygame.Surface((600, 600))
414         surf.fill(default_colour)
415         colour = data.guide_arrow_colour
416         if default_colour == colour:
417             # arrows will blend into background
418             print("BAD idea, change guide arrow colour first")

```

Next, I create the class for the cube guide – an annotated version of the 3D cube which serves as a guide for using the program. This class inherits from Cube3D, again reusing the constructor and update methods. However, whilst I still want to change the get_image function, I also want to be able to use the get_image function from Cube3D. As such, I have used `@classmethod` instead of `@staticmethod`. This means that, unlike static methods, it is not completely independent of the class, but it does require the object-specific self, it instead only requires a pointer to the class itself. This allows super() to be used in the function to access methods from the parent class.

I have also added a very basic warning in case the background colour ends up the same colour as the guide arrows, making them impossible to see. Whilst far from a perfect solution, as this is a simple problem, a small error message such as this should be sufficient for anyone to identify the problem should one occur.

```
419     def arrow_top(text, angle=0):
420         """
421             Draws an arrow aligned with the cubes slant on the top edge
422
423             :param text: text to draw above the arrow
424             :param angle: clockwise angle to rotate the arrow, 0 is upwards
425             :type text: str
426             :type angle: int
427             :return: the image of the arrow, 100*100
428             :rtype: pygame.Surface
429         """
430
431         surf = pygame.Surface((100, 100))
432         surf.fill(default_colour)
433         pygame.draw.polygon(
434             surface=surf,
435             color=colour,
436             points=((13, 13), (50, 0), (63, 63), (50, 50), (50, 93), (25, 80), (25, 25)),
437         )
438         surf.set_colorkey(default_colour)
439         # angle
440         surf = pygame.transform.rotate(surface=surf, angle=angle)
441         # letter
442         surf.blit(
443             source=interface.text(
444                 text=text,
445                 font=data.guide_font,
446                 foreground_colour=BLACK,
447                 background_colour=data.default_colour,
448             ),
449             dest=(15, 0),
450         )
451
452         return surf
```

```
452     def arrow_right(text, angle=0):
453         """
454             Draws an arrow aligned with the cubes slant on the right edge
455
456             :param text: text to draw above the arrow
457             :param angle: clockwise angle to rotate the arrow, 0 is right
458             :type text: str
459             :type angle: int
460             :return: the image of the arrow, 100*100
461             :rtype: pygame.Surface
462         """
463         surf = pygame.Surface((100, 100))
464         surf.fill(default_colour)
465         pygame.draw.polygon(
466             surface=surf,
467             color=colour,
468             points=(
469                 (38, 50),
470                 (63, 25),
471                 (63, 38),
472                 (100, 38),
473                 (100, 63),
474                 (63, 63),
475                 (63, 75),
476             ),
477         )
478         surf.set_colorkey(default_colour)
479         # angle
480         surf = pygame.transform.rotate(surface=surf, angle=angle)
481         # letter
482         surf.blit(
483             source=interface.text(
484                 text=text,
485                 font=data.guide_font,
486                 foreground_colour=BLACK,
487                 background_colour=data.default_colour,
488             ),
489             dest=(35, 25),
490         )
491     return surf
492
```

```

493     def arrow_rotate(text, angle=0):
494         """
495             Draws a large straight arrow
496
497             :param text: text to draw above the arrow
498             :param angle: clockwise angle to rotate the arrow, 0 is right
499             :type text: str
500             :type angle: int
501             :return: the image of the arrow, 100*100
502             :rtype: pygame.Surface
503         """
504
505         surf = pygame.Surface((200, 100))
506         surf.fill(default_colour)
507         pygame.draw.polygon(
508             surface=surf,
509             color=colour,
510             points=(
511                 (200, 50),
512                 (150, 100),
513                 (150, 75),
514                 (0, 75),
515                 (0, 25),
516                 (150, 25),
517                 (150, 0),
518             ),
519         )
520         surf.set_colorkey(default_colour)
521         # angle
522         surf = pygame.transform.rotate(surface=surf, angle=angle)
523         # letter
524         surf.blit(
525             source=interface.text(
526                 text=text,
527                 font=data.guide_font,
528                 foreground_colour=BLACK,
529                 background_colour=data.default_colour,
530             ),
531             dest=(0, 0),
532         )
533         return surf

```

Inside the get_image function I created 3 sub-functions. arrow_top creates an arrow aligned to the top slant of the cube that can display text (to instruct the user on what key press will achieve that rotation), the rotation parameter allows any degree of rotation, but its intended usage was specifically angle=180 to get an arrow for the downwards rotation. arrow_right is similar to arrow_top, just aligned to the right side of the cube. arrow_rotate has no alignment to the cube, instead creating a large arrow to show the direction of rotation.

```

534     # offsets allow moving cube and arrows
535     # whilst maintaining thier relative position to each other
536     cube_offset_x = 100
537     cube_offset_y = 50
538
539     # cube
540     surf.blit( source: super().get_image(True), dest: (cube_offset_x, cube_offset_y))
541
542     # up
543     surf.blit( source: arrow_top("Q"), dest: (cube_offset_x + 30, cube_offset_y + 13))
544     surf.blit( source: arrow_top("W"), dest: (cube_offset_x + 90, cube_offset_y + 45))
545     surf.blit( source: arrow_top("E"), dest: (cube_offset_x + 150, cube_offset_y + 73))
546
547     # left
548     surf.blit( source: arrow_right("R"), dest: (cube_offset_x + 100, cube_offset_y + 150))
549     surf.blit( source: arrow_right("F"), dest: (cube_offset_x + 100, cube_offset_y + 200))
550     surf.blit( source: arrow_right("V"), dest: (cube_offset_x + 100, cube_offset_y + 250))
551
552     # right
553     surf.blit( source: arrow_right( text: "T", angle: 180), dest: (cube_offset_x + 205, cube_offset_y + 152))
554     surf.blit( source: arrow_right( text: "G", angle: 180), dest: (cube_offset_x + 205, cube_offset_y + 202))
555     surf.blit( source: arrow_right( text: "B", angle: 180), dest: (cube_offset_x + 205, cube_offset_y + 252))
556
557     # down
558     surf.blit( source: arrow_top( text: "A", angle: 180), dest: (cube_offset_x, cube_offset_y + 250))
559     surf.blit( source: arrow_top( text: "S", angle: 180), dest: (cube_offset_x + 50, cube_offset_y + 277))
560     surf.blit( source: arrow_top( text: "D", angle: 180), dest: (cube_offset_x + 100, cube_offset_y + 305))
561
562     # rotate
563     surf.blit( source: arrow_rotate("X"), dest: (cube_offset_x + 50, cube_offset_y + 400))
564     surf.blit( source: arrow_rotate( text: "Y", angle: 90), dest: (cube_offset_x - 100, cube_offset_y + 100))
565     surf.blit( source: arrow_rotate( text: "Z", angle: 335), dest: (cube_offset_x + 250, cube_offset_y - 50))
566
567     return surf
568

```

At the end of the get_image function I first define variables cube_offset_x and cube_offset_y. These are used to help move all the images on the surf whilst maintaining their positions relative to each other. This was important as when I started placing the images I had started with the cube too far to the left, and didn't have enough space for Y rotation arrow. Thanks to the offset variables I only had to update 2 variables to move everything to the right to make space.

To get the image of the cube, I used the super() function so that I could access Cube3D's get_image function. I passed True as a parameter as I wanted the guide cube to always show the default image.

Then all the arrows were created and blitted to the surf before it is returned.

```
569
570     def turn(row_col, number, backwards=False, ignore_moves=False):
571         """
572             Turn 1 row or column once in a given direction, default is right/up
573
574             :param row_col: row is True, column is False
575             :param number: the number to do, left to right or top to bottom
576             :param backwards: do the opposite of the move/do the move 3 times if true
577             :param ignore_moves: don't add the move to the moves list
578             :type row_col: bool
579             :type number: int
580             :type backwards: bool
581             :type ignore_moves: bool
582             :rtype: None
583         """
584
585         if not ignore_moves:
586             # add the move to the moves list
587             # ignoring is useful for solving
588             moves.append({"direction": row_col, "number": number, "backwards": backwards})
589
590         # loop to turn the row or column the correct number of times
591         loop = 1
592         if backwards: # 3 right is used to achieve 1 left, 3 up to achieve 1 down
593             loop = 3
594
595         for _ in range(loop):
596             # make copies of the faces of the cube so the original state isn't lost
597             # deepcopy prevents pass by reference shenanigans
598             # by copying the value instead of creating a reference
599             face0 = copy.deepcopy(used_cube[0])
600             face1 = copy.deepcopy(used_cube[1])
601             face2 = copy.deepcopy(used_cube[2])
602             face3 = copy.deepcopy(used_cube[3])
603             face4 = copy.deepcopy(used_cube[4])
604             face5 = copy.deepcopy(used_cube[5])
605
606             n = number
```

```

607
608     if row_col: # turn the row
609         used_cube[2][n], used_cube[3][n], used_cube[0][n], used_cube[1][n] = (
610             face1[n],
611             face2[n],
612             face3[n],
613             face0[n],
614         )
615     if number == 0: # rotate the top face
616         used_cube[4] = numpy.rot90(m=used_cube[4], k=1, axes=(0, 1))
617     elif number == 2: # rotate the bottom face
618         used_cube[5] = numpy.rot90(m=used_cube[5], k=1, axes=(1, 0))
619     else: # turn the column
620         for i in range(3):
621             used_cube[1][i][n] = face5[i][n]
622             # 2-i flips the row number for the back
623             # 2 - n flips the column number for the back
624             used_cube[5][2 - i][n] = face3[i][2 - n]
625             used_cube[3][2 - i][2 - n] = face4[i][n]
626             used_cube[4][i][n] = face1[i][n]
627
628     if number == 0: # rotate left face
629         used_cube[0] = numpy.rot90(m=used_cube[0], k=1, axes=(0, 1))
630     elif number == 2: # rotate right face
631         used_cube[2] = numpy.rot90(m=used_cube[2], k=1, axes=(1, 0))

```

The turn function is a slightly updated and renamed version of the rotate function from prototype one. The 2 coding changes are the addition of recording moves and changing the iterator of the loop. Apart from that some small changes were made to the docstring and comments.

The addition of recording moves involved adding ignore_moves as a parameter and lines 585-588 to record the moves. Ignore moves allows moves to be undone by the solver without getting readded to the list.

The iterator was changed to _ from l as _ is conventionally used as a throwaway variable, and the loop doesn't need to know the iteration it is on for any reason, which this indicates.

```

633
634     def rotate(axis, ignore_moves=False):
635         """
636             Rotates the view of the cube without changing layout
637
638             :param axis: x, y, z
639             :type axis: str
640             :param ignore_moves: whether to add the move to the moves list, defaults to False
641             :type ignore_moves: bool or optional
642             :rtype: None
643             """
644
645         # make copies of the faces of the cube so the original state isn't lost
646         # deepcopy prevents pass by reference shenanigans
647         # by copying the value instead of creating a reference
648         face0 = copy.deepcopy(used_cube[0])
649         face1 = copy.deepcopy(used_cube[1])
650         face2 = copy.deepcopy(used_cube[2])
651         face3 = copy.deepcopy(used_cube[2])
652         face4 = copy.deepcopy(used_cube[4])
653         face5 = copy.deepcopy(used_cube[5])
654
655         if not ignore_moves: # add the move to the moves list
656             # ignoring is useful for solving
657             moves.append({"rotation": True, "direction": axis})
658
659         if axis == "x":
660             for i in range(3): # equivalent to a rotation along the x axis
661                 turn( row_col=True, number=i, ignore_moves=True)
662         elif axis == "y":
663             for i in range(3): # equivalent to a rotation along the y axis
664                 turn( row_col=False, number=i, ignore_moves=True)
665         elif axis == "z": # equivalent to a rotation along the z axis
666             # rotate the front and back faces
667             used_cube[1] = numpy.rot90( m: used_cube[1], k=1, axes=(1, 0))
668             used_cube[3] = numpy.rot90( m: used_cube[3], k=1, axes=(0, 1))
669
670             # required a lot of manual testing
671             # carefully test any changes
672             for j in range(3):
673                 for i in range(3):
674                     used_cube[0][j][2 - i] = face5[i][j]
675                     used_cube[4][j][2 - i] = face0[i][j]
676                     used_cube[2][j][2 - i] = face4[i][j]
677                     used_cube[5][j][2 - i] = face2[i][j]
678

```

The rotate function does a rotation to the cube and like the turn function, it adds the move to the moves list. Rotations in the x or y direction are achieved by turning each row or column once. Rotations in the z axis were far more complicated. Whilst I knew what needed to be done physically to the cube, I couldn't think of exactly how to that whilst using the 3D array. However, I was confident that whatever the solution was it would involve the rotations of face 1 and 3, and face0=face5, face5=face2, etc. As such, I chose to use a trial-and-error approach until I found something that worked.

```
679     def scramble():
680         """
681             Randomly scrambles the cube by making between 15 and 25 moves randomly
682
683         :rtype: None
684         """
685
686         for _ in range(randint(a=15, b=25)):
687             # randomise every aspect of the turn
688             direction = bool(randint(a=0, b=1))
689             number = randint(a=0, b=2)
690             backwards = bool(randint(a=0, b=1))
691
692             turn(row_col=direction, number=number, backwards=backwards)
693
```

To scramble the cube whilst ensuring that it was possible to solve, I chose to make many turns, knowing that if the turn function worked correctly it couldn't result in an impossible position. I used the random library to randomise each turn, and have many turns are made. I arbitrarily chose between 15 and 25 turns, finding this resulted in a sufficiently difficult position. Each move is also added to the moves list so that the solve function can undo them.

```
694
695     class Solver:
696         """
697             Solve the cube, one turn per game loop
698
699             The solve function must be called once per game loop
700             until it returns False
701             to completely solve the cube
702
703             The attribute first should be updated to True before each complete solve
704
705             A solve can optionally be made to take 5 seconds. To do this, implement a
706             time.sleep(this_object.sleep_time) before the this_object.solve() call
707             """
708
709         def __init__(self):
710             self.first = True
711             """If it is the first move of the solve
712             :type: bool"""
713             self.sleep_time = 0.2
714             """The amount of time to wait between each move
715             :type: float"""
716
```

For the solve function I created a class, with sleep_time being used in the game loop to ensure each solve takes 5 seconds, assuming there's no hardware limitations.

```
717     def solve(self):
718         """
719             Does the reverse of the last done move and removes it from the moves list
720
721             :return: False if the cube is solved, True otherwise
722             :rtype: bool
723         """
724
725         # guard clause
726         if len(moves) == 0 or self.check_solved():
727             return False
728
729         # calculate time to wait between move
730         if self.first:
731             if len(moves) > 0:
732                 # every solve should take 5 seconds regardless of moves required,
733                 # although this can be affected by hardware limitations
734                 self.sleep_time = 5 / len(moves)
735                 self.first = False
736             else:
737                 # wait upon every button press so the user knows it has 'worked'
738                 # even when the cube is already solved
739                 self.sleep_time = 1
740
741         return self.pop_move()
```

The solve function itself first checks if the cube is solved as a guard clause. This prevents the solve function from un-solving a cube just so it can completely empty the moves list. Doing this as a guard clause means checking and returning before the main block of code, so the main block isn't reached. Guard clauses can be treated as if all the following code is in an else block, but without the need for indentation.

The solve function calculates the time to sleep for that move, before returning pop.move(), which is what will undo the move.

```

742     def check_solved(self): 2 usages
743         """
744             Checks whether the cube is in a solved state
745
746             :return: True if the cube is solved, False otherwise
747             :rtype: bool
748         """
749
750         not_solved = False
751         for i in range(6): # face
752             for j in range(3): # row
753                 for k in range(3): # column
754                     # checks for any square not the same colour
755                     # as the middle square on the same face
756                     # numpy.all handles it being a tuple comparison
757                     if not numpy.all(used_cube[i][j][k] == used_cube[i][1][1]):
758                         not_solved = True
759
760         return not not_solved

```

The check_solve function checks whether all squares on a face match the centre square, as this is the case for a solved cube and if this is true for every face, then it must be a solved cube. The not_solved variable will be updated to True if any one of these checks fails, as then the cube can't be solved. The return includes a not as I want the output to be True if the cube is solved.

```

760     @staticmethod 2 usages
761     def pop_move():
762         """
763             Removes a move from the moves list and does the reverse
764
765             :return: False if the cube is solved, True otherwise
766             :rtype: bool
767         """
768
769         # guard clause
770         if len(moves) == 0:
771             return False
772
773         move = moves.pop() # get the move dictionary
774         if "rotation" in move.keys(): # check if the move was a rotation
775             # rotate does not have a backwards parameter,
776             # so achieve via 3 'forward' turns
777             for _ in range(3):
778                 # ignore move as it is part of the solve, not the user or scramble
779                 rotate(axis=move["direction"], ignore_moves=True)
780             else: # if not rotation must be turn
781                 # not move["backwards"] to always undo the move
782                 # ignore move as part of solve
783                 turn(row_col=move["direction"], number=move["number"], backwards=not move["backwards"], ignore_moves=True)
784         if len(moves) == 0: # must be solved
785             return False
786         else:
787             return True # continue solving

```

The pop_move function is static as it only uses the move list. It gets the last done move from the move list and undoes it, before returning if the moves list is empty or not. It is possible the cube may be solved even when the moves list still has moves in it, but this is checked for in the solve() function, which is its primary usage and where this issue is most likely to occur.

The moves list should ideally be stored in a class, requiring moves to be added by a function, to ensure that all moves added are stored in the correct format, however due to the small scale of this project I felt I could safely access the moves list directly and just ensure I was using the correct format every time I did.

```
788
789 class Timer: 1usage
790     """This class handles timing how long it takes the user to complete a solve"""
791
792     def __init__(self):
793         self.start_time = 0.0
794         """The time since epoch that the timer was started
795         :type: float"""
796         self.end = 0.0
797         """The time since epoch that the timer was stopped
798         :type: float"""
799         self.elapsed = 0.0
800         """The amount of time that has elapsed since the timer was started
801         :type: float"""
802         self.exists = False
803         """Whether the timer has ever been started for this solve
804         :type: bool"""
805         self.running = False
806         """Whether the timer is actively running
807         :type: bool"""
808
809     def start(self):
810         """Starts the timer and marks it as running"""
811         self.exists = True
812         self.running = True
813         self.start_time = time.time()
814
815     def stop(self):
816         """Gets the final time elapsed and stops the timer"""
817         self.update()
818         self.running = False
819
820     def delete(self):
821         """Marks the timer as not having run for the current solve"""
822         self.exists = False
823
824     def update(self):
825         """Updates the time elapsed if the timer is running"""
826         if self.running:
827             self.end = time.time()
828             self.elapsed = self.end - self.start_time
```

```
830     def display_elapsed(self): 1 usage
831         """
832             Creates a text image displaying the time elapsed
833
834         :return: The text image
835         :rtype: pygame.Surface
836         """
837
838         # if time is less than a minute
839         if self.elapsed < 60: # display time as seconds and milliseconds
840             image = interface.text(
841                 text str(round(self.elapsed, 3)) + " seconds", # round to milliseconds
842                 font: data.default_font,
843                 foreground_colour: BLACK,
844                 background_colour: default_colour,
845             )
846         else: # display time as minutes and seconds
847             image = interface.text(
848                 text str(int(self.elapsed / 60)) # minutes
849                 + "m "
850                 + str(int(self.elapsed % 60)) # seconds
851                 + "s ",
852                 font: data.default_font,
853                 foreground_colour: BLACK,
854                 background_colour: default_colour,
855             )
856
857         return image
```

The timer class manages recording the time taken for a solve so far, and in total. It also creates an image with the time taken to display to the user. If the time taken is less than a minute the image displays the time taken rounded to the millisecond, otherwise it displays the time taken in mutes and seconds.

The minutes are obtained by dividing by 60, and this value gets rounded down when it is converted to python. The seconds are obtained using modulus 60, and this also gets converted to an integer, as I did not feel milliseconds were important after the time taken exceeds a minute.

Interface

```
1  """
2  This file contains some key elements of the interface to be used by other files
3
4  This file handles creating visual elements and user interface
5  to be displayed to the screen for the user.
6  DisplayOption and DisplayBar should be used together.
7
8  black, isort and flake8 used for formatting
9  """
10
11
12 import pygame
13
14
```

```

15  class DisplayOption:
16      """
17          Creates a button with an image that changes size when hovered
18
19          This class should be used with DisplayBar
20      """
21
22      def __init__(self, image_function, display_surf, pos, size, mult, action, bg_col):
23          """
24              :param image_function: the function to get the image to use as the button
25              :param display_surf: the surface to display the button to
26              :param pos: the position to display the button from the top left
27              :param size: the x length and y length of the button
28              :param mult: how much to increase the image size when hovered
29              :param action: the function to run when the button is clicked
30              :param bg_col: the RGB value of the background colour
31              :type image_function: function
32              :type display_surf: pygame.Surface
33              :type pos: list[int] or tuple[int, int]
34              :type size: list[int]
35              :type mult: float
36              :type action: function
37              :type bg_col: tuple[int, int, int]
38          """
39
40          self.image_function = image_function
41          self.display_surf = display_surf
42          self.pos = pos
43          self.size = size
44          self.last_size = size
45          self.mult = mult
46          self.act = action
47          self.bg_col = bg_col
48          self.image = self.get_image()
49
50          self.last_size = self.size
51          """The last x,y size of the button. Used for checking if the button is hovered
52          :type last_size: list[int]"""
53
54      def get_image(self):
55          """
56              Gets the image of the button in its current state
57
58              :return: the image of the button
59              :rtype: pygame.Surface
60          """
61
62          surf = pygame.Surface(self.size)
63          cube = self.image_function()
64          cube = pygame.transform.smoothscale(surface=cube, size=self.size)
65          cube.set_colorkey(self.bg_col)
66          surf.blit(source=cube, dest=(0, 0))
67
68          return surf

```

The DisplayOption class creates a button that enlarges when hovered over. It takes a function as a parameter to get an image to display. This allows it to have an updating image. The `get_image` function gets the image and scales it to the size of the button based on `self.size`. The background is also made transparent.

```

67     def update(self, mouse_pos, offset, mouse_up):
68         """
69             Update the button, checking if it is hovered or clicked
70
71             :param mouse_pos: the x,y position of the mouse
72             :param offset: the width and height to offset the button ensures its enlarged
73             size does not overlap anything
74             :param mouse_up: whether the mouse button has been clicked
75             :type mouse_pos: tuple[int, int]
76             :type offset: list[int]
77             :type mouse_up: bool
78             :return: whether the button is hovered
79             :rtype: bool
80         """
81
82         # calculate the position of the button with its offset
83         pos = [0, 0]
84         pos[0] = self.pos[0] + offset[0]
85         pos[1] = self.pos[1] + offset[1]
86
87         # calculate the centre of the button accounting for possible enlargement
88         # and offset
89         width = self.last_size[0]
90         height = self.last_size[1]
91         centre = width // 2 + pos[0], height // 2 + pos[1]
92
93         if self.image.get_rect(center=centre).collidepoint(mouse_pos): # if hovered
94             if mouse_up: # if pressed
95                 self.act()
96                 # save same size so it can be restored
97                 temp = self.size.copy()
98                 # enlarge the button
99                 self.size[0], self.size[1] = (
100                     self.size[0] * self.mult,
101                     self.size[1] * self.mult,
102                 )
103                 self.last_size = self.size
104                 # get the enlarged image
105                 self.image = self.get_image()
106                 # restore size to the original state so it can be displayed
107                 self.size = temp
108
109                 self.display_surf.blit( source= self.image, dest= pos)
110             return True
111         else: # if not hovered
112             self.image = self.get_image()
113             self.last_size = self.size
114             self.display_surf.blit( source= self.image, dest= pos)
115             return False

```

The update function will display the button to the screen. It includes an offset as when many buttons are next to each other, and one is enlarged the rest need to move or there will be

overlapping. The last size is used as the user will see the image with its last size, not the image about to be gotten, and thus that is the size of the button they expect.

The button checks if it is being hovered by checking if the mouse position collides with its image. If it is, its checked if the mouse button has been pressed. If it has the button action is done. If not, self.size is stored in a temp variable, before self.size is updated to the enlarged size, and then get_image is called. As get image uses self.size, it will return the enlarged image. Self.size is then returned to its original size by using the temp value.

```
116
117     class DisplayBar:
118         """For creating a bar of DisplayObject in a row/column"""
119
120         def __init__(self, object_list, row):
121             """
122                 :param object_list: list of DisplayOption in sequential order
123                 :param row: if the buttons are in a row(True) or column(False)
124                 :type object_list: list[DisplayOption]
125                 :type row: bool
126
127             self.object_list = object_list
128             self.row = row
129
130         def update(self, mouse_pos, mouse_up):
131             """
132                 Updates each button in the bar and offsets them if one is hovered
133
134                 :param mouse_pos: the x,y position of the mouse
135                 :param mouse_up: whether the mouse button has been clicked
136                 :type mouse_pos: tuple[int, int] or list[int]
137                 :type mouse_up: bool
138                 :rtype: None
139
140             offset = [0, 0]
141             for i in range(len(self.object_list)):
142                 # update the button and check if it is hovered
143                 if self.object_list[i].update(mouse_pos=mouse_pos, offset=offset, mouse_up=mouse_up):
144                     if self.row:
145                         # set offset to the difference in size
146                         offset[0] = (
147                             self.object_list[i].last_size[0] - self.object_list[i].size[0]
148                         )
149                     else: # column
150                         offset[1] = (
151                             self.object_list[i].last_size[1] - self.object_list[i].size[1]
152                         )
```

DisplayBar was created to help manage DisplayOption objects. It allows for a bar of vertical or horizontal buttons to be created. With the objects stored sequentially in order from the edge of what they can touch, a loop can be used to check if an object is being hovered and offset all the following buttons by the change that object's image size, preventing overlap. This also updates all of the DisplayOptions.

The `text` function returns the image created by rendering the text. The surface's `Rect` is returned by `.render` but it is not needed so I assign it to the throwaway variable. I also run `.convert_alpha()` on the image as this optimises the image for faster blitting to the screen.

Data

```
1  """
2  This file contains global data and settings information
3
4  This data is used by multiple files in the program. It may be edited here or it could be
5  provided to the user as settings for them to change.
6
7  black, isort and flake8 used for formatting
8  """
9
10 import pygame
11 from pygame import freetype
12
13 pygame.font.init()
14 pygame.freetype.init()
15
16 # colours
17 BLACK = (0, 0, 0)
18 WHITE = (255, 255, 255)
19 YELLOW = (255, 255, 0)
20 ORANGE = (255, 165, 0)
21 RED = (255, 0, 0)
22 GREEN = (0, 255, 0)
23 BLUE = (0, 0, 255)
24 GREY = (169, 169, 169)
25
26 default_colour = GREY
27 guide_arrow_colour = BLACK
28
29
30 # fonts
31 default_font = pygame.freetype.SysFont(name: "calibri", size: 20)
32 guide_font = pygame.freetype.SysFont(name: "calibri", size: 20, bold=True)
33
```

The data file holds key data used by every file, so that I can be easily accessed and updated as needed.

Testing

I am only going to test the implemented features.

Test No.	What is being tested	Description	Method	Expected Output	Pass/Fail
1	3D cube algorithm	There should be an image algorithm that either returns a pygame.Surface containing an image of the cube or display the cube to the screen.	Blit the image returned by the function to the screen or call the function, inside the main game loop. Repeat whilst making changes to used_cube.	An image should be displayed, and it should match used_cube.	Pass
2	Cube turn – vertical, left, up	Executing the turns function with the correct parameters for the given turn should result in the leftmost column being rotated upwards.	Execute the turns function with the parameters for the turn. Run the cube display, the display should show the new cube state. The same rotation should be done to a real Rubik's cube.	The image of both cubes should be exactly the same.	Pass
3	Cube turns – vertical, middle, up	Executing the turns function with the correct parameters for the given turn should result in the middle column being rotated upwards.			Pass
4	Cube turns – vertical, right, up	Executing the turns function with the correct parameters for the given turn should result in the rightmost column being rotated upwards.			Pass

5	Cube turns – vertical, left, down	Executing the turns function with the correct parameters for the given turn should result in the leftmost column being rotated downwards.			Pass
6	Cube turns – vertical, middle, down	Executing the turns function with the correct parameters for the given turn should result in the middle column being rotated downwards.			Pass
7	Cube turns – vertical, right, down	Executing the turns function with the correct parameters for the given turn should result in the rightmost column being rotated downwards.			Pass
8	Cube turns – horizontal, top, right	Executing the turns function with the correct parameters for the given turn should result in the upper row being rotated right.			Pass
9	Cube turns – horizontal, middle, right	Executing the turns function with the correct parameters for the given turn should result in the middle row being rotated right.			Pass

10	Cube turns – horizontal, bottom, right	Executing the turns function with the correct parameters for the given turn should result in the lower row being rotated right.			Pass
11	Cube turns – horizontal, top, left	Executing the turns function with the correct parameters for the given turn should result in the upper row being rotated left.			Pass
12	Cube turns – horizontal, middle, left	Executing the turns function with the correct parameters for the given turn should result in the middle row being rotated left.			Pass
13	Cube turns – horizontal, bottom, left	Executing the turns function with the correct parameters for the given turn should result in the lower row being rotated left.			Pass
14	Scramble	The scramble function should randomly position the individual squares whilst still ensuring that the cube is solvable.	Add a delay between each move in the scramble function. Run the scramble function and follow along with a real Rubik's cube.	Each move done by the scrambler should be possible on the real Rubik's cube.	Pass

15	Solver - solving	The solve function should solve the cube, showing the user each step, ensuring that each move is possible and not simply changing the used_cube to fit as needed.	Manually scramble the cube, doing each move to a real Rubik's cube as well. Run the solver and follow the moves on the Real Rubik's cube.	Each move done by the solver should be possible on the real Rubik's cube and at the end the cube should be solved.	Fail
16	Solver – stop solving	If used_cube reaches a solved state, the solver should stop solving, regardless of if there something such as a moves list indicates there are more moves to do to solve the cube.	Manually scramble the cube, ensuring that you return to a solved state at least once then scramble from there. Run the solver.	The solver should stop when it reaches the first solved state.	Pass
17	Check solved	There should be a function to check if a cube state is solved or not.	Test the function using multiple different cube states, some of which are manually or automatically scrambled.	The outputs should match the given cube state.	Pass
18	Hints	The hint function should complete one move towards the solve. It must only be one move, and it must help solve the cube.	Scramble the cube then run the runt the hint function. Note the move that it makes. Undo that move and then run the solver (test 15 must have passed).	Only one move should be completed by the hint function. The move should match the one done by the solver.	Pass

19	Timer – time elapsed	The timer should correctly record the amount of timer that has passed since it started	Start the timer. Wait for 10 seconds (counted via a trusted, real, timer). Print the time elapsed. Repeat a few times with various amounts of time waited.	The trusted timer and the timer being tested should have a matching (or very similar to account for human error) times.	Pass
20	Timer – auto start	The timer should automatically start upon scramble.	Scramble the cube. Solve the cube. Scramble the cube, use hint function.	Each time the cube is scrambled the timer should start,	Pass
21	Timer – auto stop	The timer should automatically stop upon being solved.	Scramble the cube. Use the solve function. Scramble the cube. Monitor the time elapsed during this.	Upon being solved and when the solver is used, the timer should stop. The timer should not stop if these do not occur.	Fail
27	Guide algorithm	The should either be a function that returns a pygame.Surface or a procedure that draws the image to the screen. The image should show how to use the cube.	Either blit the pygame.Surface to the screen or call the procedure inside a game loop.	An image should be displayed.	Pass

28	Guide algorithm – prevent moves	The cube image should only display the default cube and as such it should not allow cube interactions to happen when the guide is being displayed.	When displaying the cube, try the following: turns, rotations, scrambling and solving.	None of the functions should work. The cube should remain unchanged.	Pass
----	---------------------------------	--	--	--	------

Improvements

Test 15 – The solve function results in a key error when rotations are in the move list.

This was due to a small spelling mistake in ‘rotation’ in line 773 which resulted in rotations being treated as turns.

```
773 if "rotatiopn" in move.keys(): # check if the move was a rotation
774     # rotate does not have a backwards parameter,
775     # so achieve via 3 'forward' turns
776     for _ in range(3):
777         # ignore move as it is part of the solve, not the user or scramble
778         rotate(axis=move["direction"], ignore_moves=True)
779     else: # if not rotation must be turn
780         # not move["backwards"] to always undo the move
781         # ignore move as part of solve
782         turn(row_col=move["direction"], number=move["number"], backwards=not move["backwards"], ignore_moves=True)
```

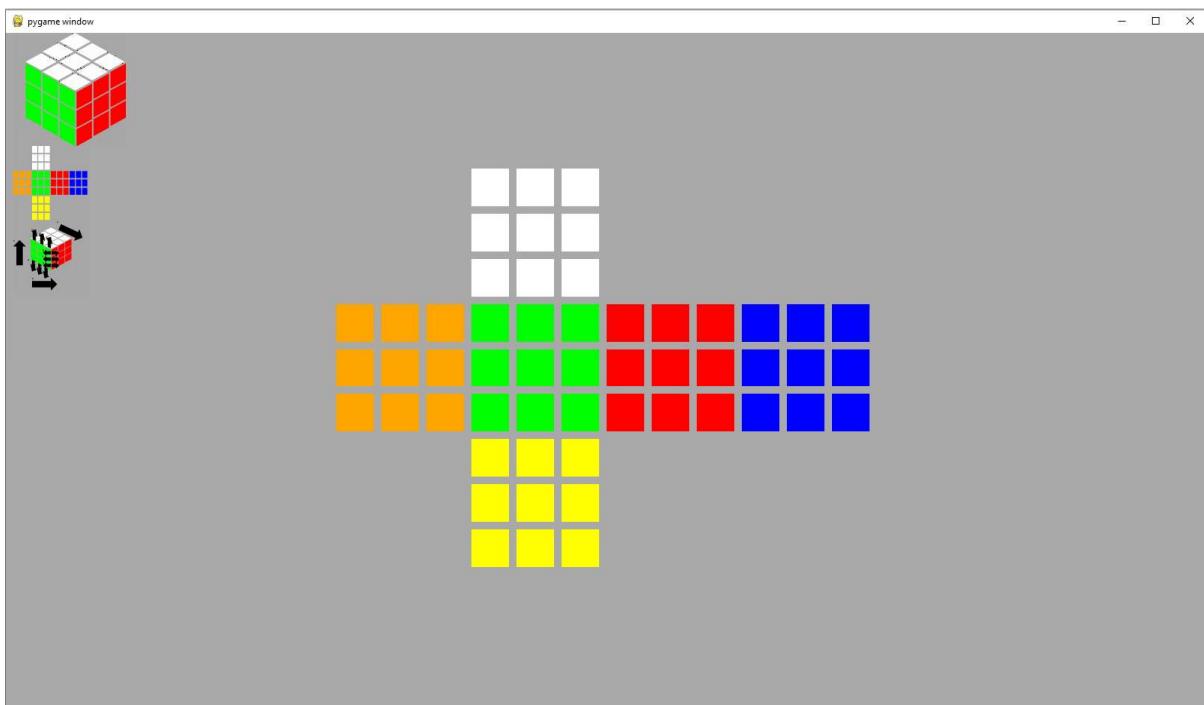
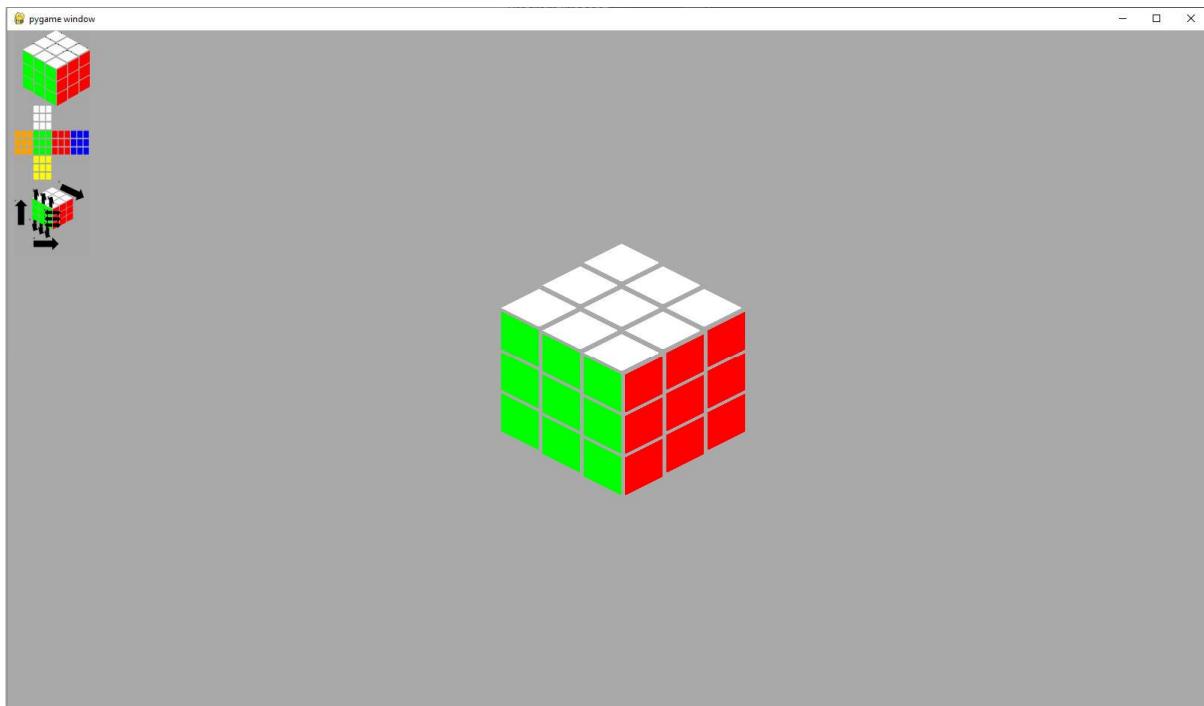
This was easily solved by simply fixing the spelling mistake.

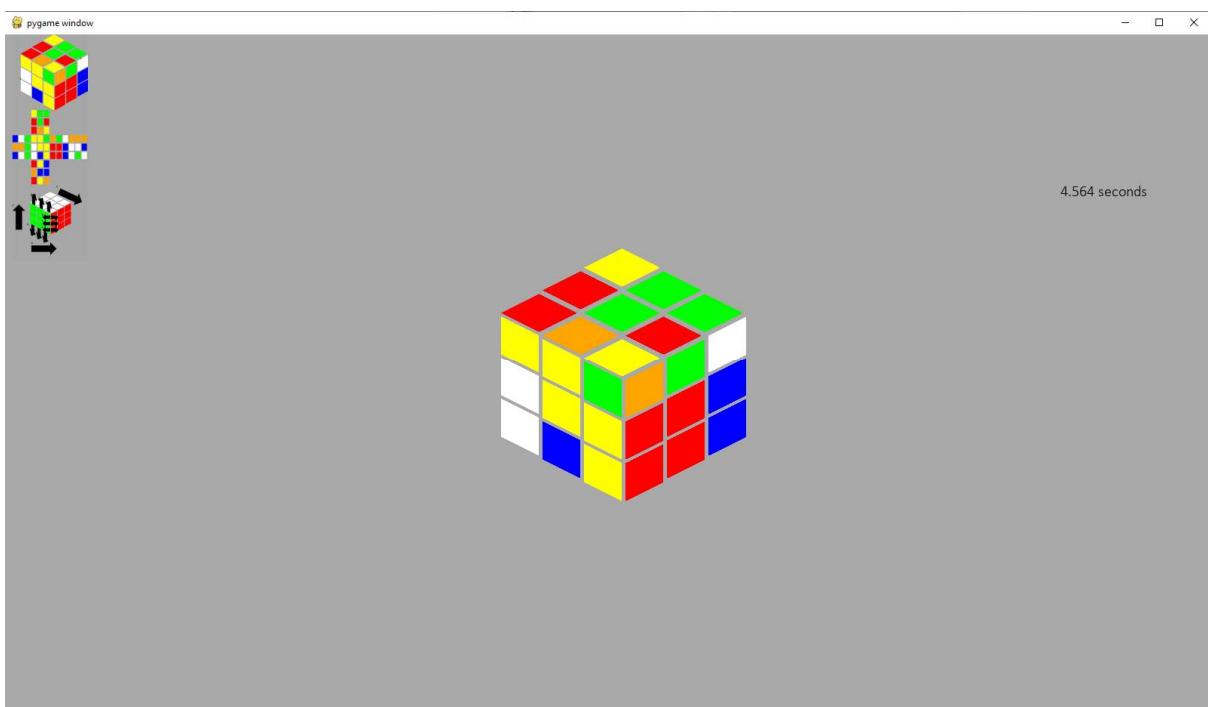
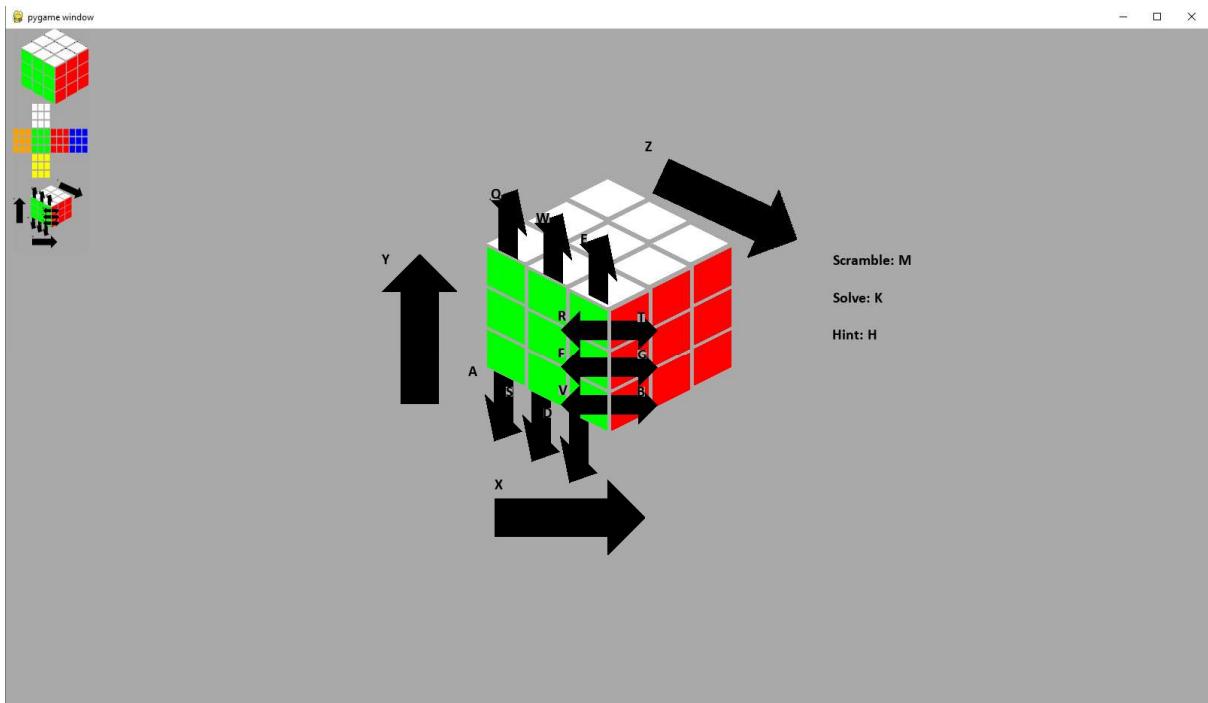
```
773 if "rotation" in move.keys(): # check if the move was a rotation
774     # rotate does not have a backwards parameter,
775     # so achieve via 3 'forward' turns
776     for _ in range(3):
777         # ignore move as it is part of the solve, not the user or scramble
778         rotate(axis=move["direction"], ignore_moves=True)
779     else: # if not rotation must be turn
780         # not move["backwards"] to always undo the move
781         # ignore move as part of solve
782         turn(row_col=move["direction"], number=move["number"], backwards=not move["backwards"], ignore_moves=True)
```

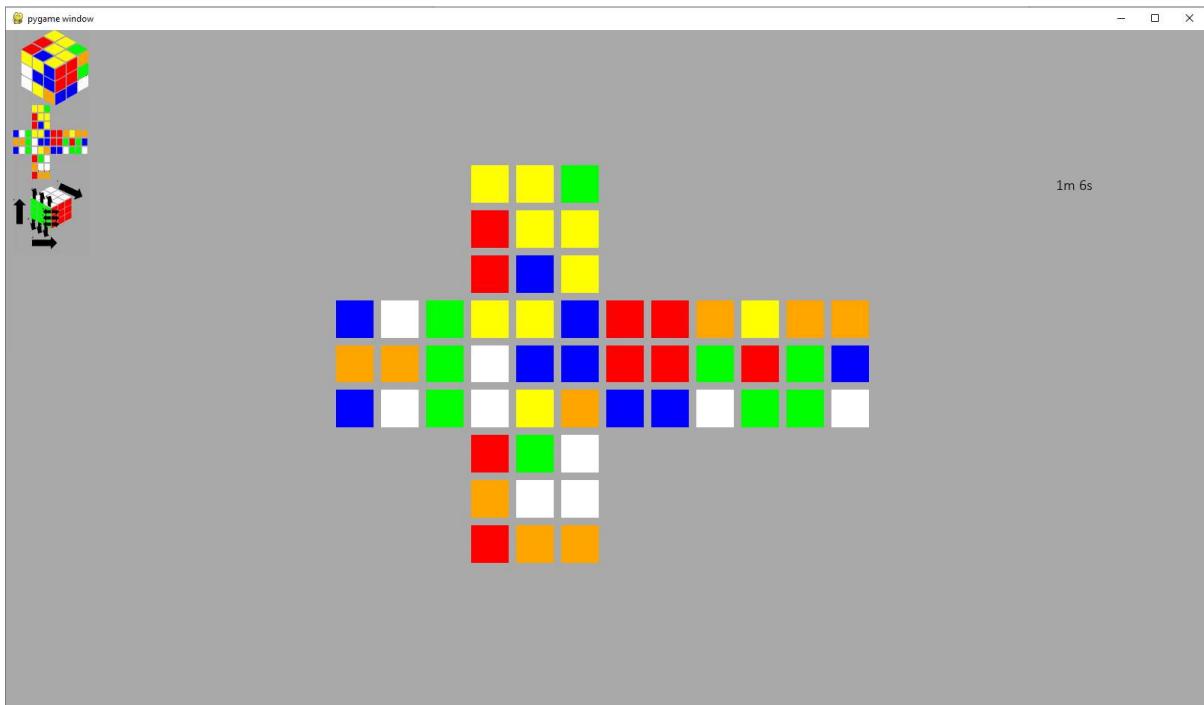
Test 21 – If the scramble key is pressed before the solver ends, the timer will start whilst the solver continues to solve the cube. This was solved by making the solver stop when the scramble key (which also starts the timer) is pressed.

```
elif event.key == pygame.K_m: # scramble
    cube.scramble()
    # prevent the timer from being started whilst the solver runs
    # was achieved by scrambling whilst the timer ran
    solve_cube = False
    timer.start() # start timer
```

Program Images







Prototype 3

Development

Cube display:

- [Class](#)
- [Object](#)
- [Image](#)

Cube logic:

- [Data storage](#)
- [Turns](#)
- [Rotations](#)

Scramble:

- [Function](#)
- [Usage](#)

Solver:

- Moves [class](#)
- Solver [class](#)

- [Usage](#)

Hint:

- [Function](#)
- [Usage](#)

Timer:

- [Class](#)
- [Object](#)
- [Automatic start](#)
- [Automatic stop](#)

Leaderboard:

- [Class](#)
- [Object](#)
- [Image](#)

Login:

- [Usage](#)

Guide:

- [Class](#)
- [Object](#)
- [Usage](#)

Save:

- File [storage](#)
- Manager [class](#)
- User [class](#)
- [Automatic](#)

History:

- [Data](#)
- [Class](#)
- [Object](#)
- [Image](#)

Validation:

- [Class](#)

- [Object](#)
- Usage throughout [Main](#)

- [Ignore invalid key presses](#)
- [Ignore cube interactions when on guide](#)

Main

```

1  """
2  This is the file to run to execute the program
3
4  This file handles the main loop of the program, it gets images and data from the other
5  files and displays them. It also handles user input within the game loop.
6
7  black, isort and flake8 used for formatting
8  """
9
10 import sys
11 import time
12
13 import cube
14 import features
15 import game_data # for changing variables in data file
16 import interface
17 import pygame
18 import user_data
19 from game_data import *
20 from Login import login_window
21 from validation import ValidateScreenPositions
22
23 # window
24 pygame.init()
25 width = 1600
26 height = 900
27 screen = pygame.display.set_mode(size: (width, height), flags: pygame.RESIZABLE)
28 pygame.display.set_caption("Rubik's Cube")
29
30 # validation
31 val = ValidateScreenPositions(width: width, height: height)
32
33 # cubes and visuals
34 cube_net = cube.CubeNet(surface: screen, pos: val.run((width // 2, height // 2)))
35 cube_3d = cube.Cube3D(surface: screen, pos: val.run((width // 2, height // 2)))
36 cube_guide = cube.CubeGuide(surface: screen, pos: val.run((width // 2, height // 2)))
37 display_history = features.DisplayHistory(screen: screen, pos: val.run((width // 2, height // 2)))
38 display_leaderboard = features.Leaderboard(screen: screen, pos: val.run((width // 2, height // 2)))

```

The additional files have been imported and the new display elements – history and leaderboard – have been added to the display section. Game_data has been imported as well as had all imported from it, as to change variables in other files you must use file.variable = ..., as from file import variable gets the variables value, instead of a pointer to it.

```

39
40
41 class Buttons: 15 usages
42 """
43     This class handles the rendering of the buttons
44
45     This class is largely self-contained, the only usage should be to run
46     .update as this automatically updates the buttons
47 """
48
49 cube_option = interface.DisplayOption(
50     image_function: lambda: cube_3d.get_image(),
51     display_surf: screen,
52     pos: val.run([10, 0]),
53     size: [100, 100],
54     mult: 1.5,
55     action: lambda: Buttons.display_swap("3d"),
56     bg_col: default_colour,
57 )
58 net_option = interface.DisplayOption(
59     image_function: lambda: cube_net.get_image(),
60     display_surf: screen,
61     pos: val.run([10, 100]),
62     size: [100, 100],
63     mult: 1.5,
64     action: lambda: Buttons.display_swap("net"),
65     bg_col: default_colour,
66 )

67 guide_option = interface.DisplayOption(
68     image_function: lambda: cube_guide.get_image(),
69     display_surf: screen,
70     pos: val.run([10, 200]),
71     size: [100, 100],
72     mult: 1.5,
73     action: lambda: Buttons.display_swap("guide"),
74     bg_col: BLACK,
75 ) # should be default colour,
76 # but this causes the background of the hovered button to be black.
77 # May be an error with pygame.smoothscale in interface file
78 # this works as a solution
79
80 history_option = interface.DisplayOption(
81     image_function: lambda: interface.text(
82         text: "HISTORY",
83         font: default_font,
84         foreground_colour: BLACK,
85         background_colour: default_colour
86     ),
87     display_surf: screen,
88     pos: val.run([10, 300]),
89     size: [100, 25],
90     mult: 1.5,
91     action: lambda: Buttons.display_swap("history"),
92     bg_col: BLACK, # same problem as guide
93 )
94

```

```

95     leaderboard_option = interface.DisplayOption(
96         image_function: lambda: interface.text(
97             text: "LEADERBOARD",
98             font: default_font,
99             foreground_colour: BLACK,
100            background_colour: default_colour
101        ),
102        display_surf: screen,
103        pos: val.run([10, 325]),
104        size: [100, 25],
105        mult: 1.5,
106        action: lambda: Buttons.display_swap("leaderboard"),
107        bg_col: BLACK, # same problem as guide
108    )
109
110    cube_option_bar = interface.DisplayBar( # update with any new options
111        object_list: [cube_option, net_option, guide_option, history_option, leaderboard_option],
112        row: False,
113    )
114    display_option = "3d"
115
116    @staticmethod 5 usages
117    def display_swap(option):
118        """
119            Updates display_option variable within the class
120
121            This provides a function for interface.DisplayOption objects
122            to update the display_option variable which is saved with this class
123
124            :param option: the new display_option: 3d, net, guide, history or leaderboard
125            :type option: str
126        """
127        Buttons.display_option = option
128
129    @staticmethod
130    def update(mouse_pos, mouse_up):
131        """
132            Updates each button in the class
133
134            :param mouse_pos: the x,y position of the mouse
135            :param mouse_up: whether the mouse button has been clicked
136            :type mouse_pos: tuple[int, int] or list[int, int]
137            :type mouse_up: bool
138            :rtype: None
139        """
140        Buttons(cube_option_bar).update(mouse_pos=mouse_pos, mouse_up=mouse_up)
141

```

The buttons class is largely the same, just with display_history and display_leaderboard getting their buttons added.

```

142
143     # used for solving the cube
144     solve_cube = False
145     """If the cube is being solved
146     :type solve_cube: bool"""
147     solver = features.Solver()
148
149     timer = features.Timer()
150     last_save = time.time()
151     """The timestamp of the last save, used for calculating time since last save
152     :type last_save: float"""
153
154
155     # login
156     def load(username):
157         """Desinged to be called by the login window, this function will load the users data
158
159         Uses Manager.load to load the users data and then checks the game state, updating
160         details about the timer and solver is nessessary
161
162         :param username: the unique username of the user
163         :type username: str
164         """
165         user_data.Manager.load(username)
166
167         if game_data.time_taken > 0: # timer is running
168             # manually start timer to avoid changing start time
169             timer.exists = True
170             timer.running = True
171             timer.start_time = (
172                 time.time() - game_data.time_taken
173             ) # act as if timer has just started
174             if game_data.solver_used: # solver is runnnning
175                 # finish solving cube
176                 solver.first = False
177                 solve_cube = True
178
179
180         login_window.Window(lambda u: load(u))
181
182

```

The load function is designed to be called by the login_window, which give it the user's username. The save functions in the user data file will use this to load their data. This function then checks if the timer or solver are supposed to be running and updates their objects variables to start them in a way that makes it seem as if they never stopped.

The login function is then called. This starts the login window, which checks the user's username and password before calling the load function.

```

183 # game loop
184 while True:
185     mouse_pos = pygame.mouse.get_pos()
186     mouse_up = False
187     val.update_size(pygame.display.get_surface().get_size())
188
189     for event in pygame.event.get():
190         if event.type == pygame.QUIT:
191             pygame.quit()
192             sys.exit()
193         elif event.type == pygame.MOUSEBUTTONUP:
194             mouse_up = True
195         elif event.type == pygame.MOUSEWHEEL and Buttons.display_option == "history":
196             display_history.scroll(event.y * 25)
197         # prevent any moves made whilst on guide cube
198         elif event.type == pygame.KEYDOWN and Buttons.display_option != "guide":
199             # row right
200             if event.key == pygame.K_t:
201                 cube.turn( row_col: True, number: 0)
202             elif event.key == pygame.K_g:
203                 cube.turn( row_col: True, number: 1)
204             elif event.key == pygame.K_b:
205                 cube.turn( row_col: True, number: 2)
206             # row left
207             elif event.key == pygame.K_r:
208                 cube.turn( row_col: True, number: 0, backwards: True)
209             elif event.key == pygame.K_f:
210                 cube.turn( row_col: True, number: 1, backwards: True)
211             elif event.key == pygame.K_v:
212                 cube.turn( row_col: True, number: 2, backwards: True)
213
214             # column up
215             elif event.key == pygame.K_q:
216                 cube.turn( row_col: False, number: 0)
217             elif event.key == pygame.K_w:
218                 cube.turn( row_col: False, number: 1)
219             elif event.key == pygame.K_e:
220                 cube.turn( row_col: False, number: 2)
221             # column down
222             elif event.key == pygame.K_a:
223                 cube.turn( row_col: False, number: 0, backwards: True)
224             elif event.key == pygame.K_s:
225                 cube.turn( row_col: False, number: 1, backwards: True)
226             elif event.key == pygame.K_d:
227                 cube.turn( row_col: False, number: 2, backwards: True)
228
229             # rotations
230             elif event.key == pygame.K_x:
231                 cube.rotate("x")
232             elif event.key == pygame.K_y:
233                 cube.rotate("y")
234             elif event.key == pygame.K_z:
235                 cube.rotate("z")

```

The first section of the events portion in nearly the same, with only the one event being added for mouse scrolling. This allows the game history image to be scrolled.

The quit section has been updated to include sys.exit(), which will close the program, as it was previously stopping due to an error occurring when trying to run a pygame function when pygame had been quit.

```
237     elif event.key == pygame.K_k: # solve
238         game_data.solver_used = True
239         if timer.running: # ensures the attempt was started
240             # failed attempts should be recorded
241             game_data.solved = False
242             user_data.game_history.add_game()
243             timer.delete()
244
245             solve_cube = True
246             elif event.key == pygame.K_m: # scramble
247                 if timer.running: # ensures the attempt was started
248                     # failed attempts should be recorded
249                     user_data.game_history.add_game()
250
251             # reset key data
252             game_data.moves.clear()
253             game_data.move_count = 0
254             game_data.scrambler_count = 0
255             game_data.hints_used = False
256             game_data.solver_used = False
257             game_data.solved = False
258             game_data.time_taken = 0
259             game_data.start_time = time.time()
260
261             features.scramble()
262             # prevent the timer from being started whilst the solver runs
263             # was achieved by scrambling whilst the timer ran
264             solve_cube = False
265             timer.start() # start timer
266             elif event.key == pygame.K_h: # hint
267                 game_data.hints_used = True
268                 solver.pop_move()
269
```

The second section has statements added to update the information in game_data when key events happen. It will also add completed games to the user's game history.

```

270     screen.fill(default_colour) # background colour
271
272     if solve_cube:
273         # ensures each solve take 5 sections, assuming no hardware limitations
274         time.sleep(solver.sleep_time)
275         solve_cube = solver.solve() # solves one move
276     else:
277         solver.first = True # so next solve it is set to true
278
279     if timer.running and solver.check_solved(): # on a solve
280         timer.stop()
281         game_data.solved = True
282         user_data.game_history.add_game()
283         display_leaderboard.update_list(
284             time=game_data.time_taken,
285             moves=game_data.move_count
286         )
287
288     if Buttons.display_option == "3d":
289         display_cube = cube_3d
290     elif Buttons.display_option == "net":
291         display_cube = cube_net
292     elif Buttons.display_option == "guide":
293         # also prevents cube interact as uses default
294         display_cube = cube_guide
295         # actions text
296         screen.blit(
297             source=interface.text(
298                 text="Scramble: M",
299                 font=guide_font,
300                 foreground_colour=BLACK,
301                 background_colou
302             ),
303             dest=val.run((1100, 300)),
304         )
305         screen.blit(
306             source=interface.text(
307                 text="Solve: K",
308                 font=guide_font,
309                 foreground_colour=BLACK,
310                 background_colou
311             ),
312             dest=val.run((1100, 350)),
313         )
314         screen.blit(
315             source=interface.text(
316                 text="Hint: H",
317                 font=guide_font,
318                 foreground_colour=BLACK,
319                 background_colou
320             ),
321             dest=val.run((1100, 400)),
322         )
323     elif Buttons.display_option == "history":
324         display_cube = display_history
325     elif Buttons.display_option == "leaderboard":
326         display_cube = display_leaderboard
327         display_cube.update() # actually update cube

```

```

328     if timer.exists: # display timer
329         screen.blit( source: timer.display_elapsed(), dest: val.run((1400, 200)))
330         timer.update()
331
332
333     # update buttons
334     Buttons.update(mouse_pos: mouse_pos, mouse_up: mouse_up)
335
336     # save every 5 seconds
337     if time.time() - last_save > 5:
338         time_since_save = time.time()
339         user_data.Manager.save()
340
341     pygame.display.flip()
342

```

The game loop is largely the same as well, with features added to update the user's game history on a completed solve, display the new history and leaderboard screens, and to run the save function every 5 seconds.

Validation

This file is unchanged from prototype 2.

```

1  """
2  This file contains validation functions and error handling
3
4  black, isort and flake8 used for formatting
5  """
6
7  import time
8
9
10 class InvalidScreenPosition(Exception):
11     """This exception is raised when the screen position is invalid"""
12
13     def __init__(self, pos):
14         """
15             :param pos: the x,y position that is invalid
16             :type pos: tuple[int, int] or list[int]
17         """
18         super().__init__(f"Invalid Screen Position: {pos}")
19         with open("error.txt", "a") as f:
20             error_time = time.time()
21             f.write(f"{error_time} Invalid Screen Position: {pos}\n")
22

```

```
24 class ValidateScreenPositions: 4 usages
25 """
26     This class contains a screen position validation function
27
28     It will ensure a screen position is valid based on a 4k resolution screen
29     and the size of the window being displayed to
30 """
31
32     def __init__(self, width, height):
33         """
34             :param width: the width of the screen window
35             :param height: the height of the screen window
36             :type width: int
37             :type height: int
38         """
39         self.width = width
40         self.height = height
41
42     def run(self, pos):
43         """
44             Ensures a position is within the confines of the screen and 4k resolution
45
46             This function will throw an error if the position is invalid.
47             An invalid position is one that is outside the confines of the screen based
48             width and height, or a 4k resolution screen.
49
50             :param pos: the x,y position to check
51             :type pos: tuple[int, int] or list[int]
52             :return: the x,y screen position if it is valid, else raises an exception
53             :rtype: tuple[int, int] or list[int]
54         """
55         if (
56             pos[0] < 0
57             or pos[0] > self.width
58             or pos[0] > 3840
59             or pos[1] < 0
60             or pos[1] > self.height
61             or pos[1] > 2160
62         ):
63             raise InvalidScreenPosition(pos)
64         else:
65             return pos
66
67     def update_size(self, size): 1 usage
68         """
69             Update the screen width and height
70
71             :param size: the width and height of the window
72             :type size: tuple[int, int] or list[int]
73         """
74         self.width = size[0]
75         self.height = size[1]
```

Cube

The cube file has been split across 3 files: cube, features, and game_data.

The 3 cube classes that display an image remain in the cube file, along with the turn and rotations functions. The code for these is largely the same, with only small adjustments made as to account for data structures such as the moves list being moved into game_data.

```
1  """
2      This file contains the code for the cube as well as the turn and rotation functions
3
4      This file handles creating the images to display the cube
5      and the basic turn and rotation functions for interacting with the cube
6
7      black, isort and flake8 used for formatting
8      """
9
10     import copy
11
12     import game_data as gd
13     import interface
14     import numpy
15     import pygame
16     from game_data import BLACK, default_colour, default_cube
17
18
19     @lru_cache(2)
20     class CubeNet:
21         """Handles the display of the cube as a net to a fixed position on the screen"""
22
23         def __init__(self, surface, pos):
24             """
25                 :param surface: The surface that this cube is to be blitted to
26                 :param pos: The centre position that this cube is to be blitted to: x,y
27                 :type surface: pygame.Surface
28                 :type pos: list[int] or tuple[int, int]
29             """
30
31             # pos is centre
32             self.screen = surface
33             self.pos = pos
34
35         def update(self):
36             """
37                 Updates the cube image and re-blits it to the surface
38
39                 :rtype: None
40             """
41
42             image = self.get_image()
43             self.screen.blit( source= image, dest= image.get_rect( center= self.pos))
```

```

42     @staticmethod
43     def get_image(default=False):
44         """
45             Creates the image of the cube from the current state of the cube
46
47             :param default: if True, uses the default image instead of the current state
48             :type default: bool
49             :return: the image of the cube as a 720x540 surface
50             :rtype: pygame.Surface
51         """
52
53         surf = pygame.Surface((720, 540))
54         surf.fill(default_colour)
55         colour_3d_array = gd.used_cube
56         if default:
57             colour_3d_array = default_cube
58
59         def square(colour):
60             """
61                 Creates a single square with the given colour
62
63                 :param colour: the RGB values of the colour
64                 :type colour: tuple[int, int, int]
65                 :return: the square image, 50x50
66                 :rtype: pygame.Surface
67
68             """
69             surf = pygame.Surface((50, 50))
70             surf.fill(colour)
71             return surf
72
73         def row(colour_list):
74             """
75                 Creates the image of a row of 3 squares.
76
77                 :param colour_list: List len(3) of tuples, where each tuple is an RGB value
78                 :type colour_list: list[tuple[int, int, int]]
79                 :return: the row image, 170*50
80                 :rtype: pygame.Surface
81
82             """
83
84             surf = pygame.Surface((170, 50))
85             surf.fill(default_colour)
86             for i in range(3):
87                 # iterates alongside the list of colours,
88                 # getting a square with the respective colour and
89                 # blitting it to calculated position
90                 # i * 50 ensures the square is blitted after the previous one;
91                 # not inside it
92                 # i * 10 adds 10 spacing between the cubes
93                 surf.blit(source=square(colour_list[i]), dest=(i * 50 + i * 10, 0))
94
95             return surf

```

```

92     def face(colour_array):
93         """
94             Creates one face (side) from 3 rows
95
96             :param colour_array: 2D array (3x3)(row x col) of tuples,
97             | where each tuple is an RGB value
98             :type colour_array: list[list[tuple[int, int, int]]]
99             :return: the face image, 170*170
100            :rtype: pygame.Surface
101
102        """
103
104        surf = pygame.Surface((170, 170))
105        surf.fill(default_colour)
106        for i in range(3):
107            # iterates alongside the list of rows,
108            # getting and blitting the row image to calculated position
109            # i * 50 ensures the row is placed beneath,
110            # and not inside, the previous row
111            # i * 10 is for spacing between the rows
112            surf.blit(source=colour_array[i], dest=(0, i * 50 + i * 10))
113
114        # 4 of the faces are placed next to each other so a loop can place them
115        for i in range(4):
116            surf.blit(
117                source=face(colour_3d_array[i]), # gets the image of the face
118                # 180 * i includes 10 pixels spacing
119                # placed 180 down to allow top to be placed above with 10 pixels spacing
120                dest=(180 * i, 180),
121            )
122
123        # 180 x val aligns with front face
124        surf.blit(source=face(colour_3d_array[4]), dest=(180, 0))
125        # 360 is below face image with 10 pixels spacing
126        surf.blit(source=face(colour_3d_array[5]), dest=(180, 360))
127        return surf

```

```
128
129 ④ class Cube3D(CubeNet): 2 usages
130
131     """
132     Handles the display of the 3d cube to a fixed position on the screen
133
134     This class is a child of CubeNet, only changing the get_image method
135
136     """
137 ④⑤ @staticmethod
138     def get_image(default=False):
139         """
140             Creates the image of the cube from its current state
141
142             :param default: if True, uses the default image instead of the current state
143             :type default: bool
144             :return: the cube image, 365*335
145             :rtype: pygame.Surface
146
147         surf = pygame.Surface((365, 335))
148         surf.fill(default_colour)
149         colour_3d_array = gd.used_cube
150         if default:
151             colour_3d_array = default_cube
152
153     def right():
154         """
155             Draws the right face of the cube to the surf, it is a slanted square
156
157             :rtype: None
158
159         def square(colour):
160             """
161                 Creates a slanted cube of solid colour
162
163                 :param colour: RGB values
164                 :type colour: tuple[int, int, int]
165                 :return: the square image, 50*75
166                 :rtype: pygame.Surface
167
168             surf = pygame.Surface((50, 75))
169             surf.fill(default_colour)
170             pygame.draw.polygon(surface=surf, color=colour, points=((0, 25), (50, 0), (50, 50), (0, 75)))
171             surf.set_colorkey(default_colour) # make background transparent
172
173             return surf
```

```

174     def row(colour_list):
175         """
176             Creates a slanted row of 3 squares
177
178             :param colour_list: List len(3) of tuples of RGB values
179             :type colour_list: list[tuple[int, int, int]]
180             :return: the row image, 165*135
181             :rtype: pygame.Surface
182         """
183
184         surf = pygame.Surface((165, 135))
185         surf.fill(default_colour)
186         for i in range(3):
187             # 55 includes 5 pixels spacing
188             # -30 includes 5 pixels spacing
189             surf.blit(source=square(colour_list[i]), dest=(55 * i, 60 - (30 * i)))
190         surf.set_colorkey(default_colour)
191         return surf
192
193     def face(colour_array):
194         """
195             Stacks 3 row images to create a face of 9 squares
196
197             :param colour_array: 2D (3x3)(row x col) array of tuples of RGB values
198             :type colour_array: list[list[tuple[int, int, int]]]
199             :return: the row image, 165*250
200             :rtype: pygame.Surface
201
202         surf = pygame.Surface((165, 250))
203         surf.fill(default_colour)
204         for i in range(3):
205             surf.blit(source=row(colour_array[i]), dest=(0, 55 * i))
206         return surf
207
208         # positioned to the right of front face with 5 pixels spacing
209         surf.blit(source=face(colour_3d_array[2]), dest=(205, 90))

```

```

210
211     def front():
212         """
213             Draws the front face of the cube to the surf, it's a slanted square
214
215         :rtype: None
216
217     def square(colour):
218         """
219             Creates a slanted cube of solid colour
220
221         :param colour: RGB values
222         :type colour: tuple[int, int, int]
223         :return: the square image
224         :rtype: pygame.Surface
225
226         surf = pygame.Surface((50, 75))
227         surf.fill(default_colour)
228         pygame.draw.polygon(surf, colour, points=((0, 0), (50, 25), (50, 75), (0, 50)))
229         surf.set_colorkey(default_colour)
230         return surf
231
232     def row(colour_list):
233         """
234             Creates the image of a row of 3 squares
235
236         :param colour_list: list len(3) of tuples of RGB Values
237         :type colour_list: list[tuple[int, int, int]]
238         :return: the row image
239         :rtype: pygame.Surface
240
241         surf = pygame.Surface((165, 135))
242         surf.fill(default_colour)
243         for i in range(3):
244             # 55 includes 5 pixels spacing
245             # 30 includes 5 pixels spacing
246             surf.blit(source=square(colour_list[i]), dest=(55 * i, 30 * i))
247         surf.set_colorkey(default_colour)
248         return surf
249
250     def face(colour_array):
251         """
252             Stacks 3 row images to create a face of 9 squares
253
254         :param colour_array: 2D array (3x3)(row x col) of tuples of RGB values
255         :type colour_array: list[list[tuple[int, int, int]]]
256         :return: the face image
257         :rtype: pygame.Surface
258
259         surf = pygame.Surface((165, 250))
260         surf.fill(default_colour)
261         for i in range(3):
262             # 55 includes 5 pixels spacing
263             surf.blit(source=row(colour_array[i]), dest=(0, 55 * i))
264         surf.set_colorkey(default_colour)
265         return surf
266
267     # positioned below top of front face with 5 pixels spacing
268     surf.blit(source=face(colour_3d_array[1]), dest=(40, 90))
269

```

```
270
271     def top():
272         """
273             Draws the top face of the cube to the surf,
274             it is a horizontally stretched square
275
276         :rtype: None
277
278     def square(colour):
279         """
280             Creates a horizontally stretched cube of solid colour
281
282             :param colour: RGB values
283             :type colour: tuple[int, int, int]
284             :return: the square image
285             :rtype: pygame.Surface
286
287             surf = pygame.Surface((100, 50))
288             surf.fill(default_colour)
289             pygame.draw.polygon(
290                 surface=surf, color=colour, points=((50, 0), (100, 25), (50, 50), (0, 25)))
291             )
292             surf.set_colorkey(default_colour)
293             return surf
294
295     def row(colour_list):
296         """
297             Creates the image of a row of 3 squares
298
299             :param colour_list: List len(3) of tuples of RGB values
300             :type colour_list: list[tuple[int, int, int]]
301             :return: the row image
302             :rtype: pygame.Surface
303
304             surf = pygame.Surface((215, 120))
305             surf.fill(default_colour)
306             for i in range(3):
307                 # 55 includes 5 pixels spacing
308                 # 30 includes 5 pixels spacing
309                 surf.blit(source=square(colour_list[i]), dest=(55 * i, 30 * i))
310             surf.set_colorkey(default_colour)
311             return surf
312
```

```
313     def face(colour_array):
314         """
315             Stacks 3 row images to create a face of 9 squares
316
317             :param colour_array: 2D array (3x3)(row x col) of tuples of RGB values
318             :type colour_array: list[list[tuple[int, int, int]]]
319             :return: the face image
320             :rtype: pygame.Surface
321         """
322
323         surf = pygame.Surface((370, 315))
324         surf.fill(default_colour)
325         for i in range(3):
326             # 150 - to place bottom to top
327             # 55 includes 5 pixels spacing
328             # 30 includes 5 pixels spacing
329             surf.blit(source=colour_array[i], dest=(150 - (55 * i), 30 * i))
330         surf.set_colorkey(default_colour)
331         return surf
332
333     surf.blit(source=face(colour_3d_array[4]), dest=(0, 0))
334
335     right()
336     front()
337     top()
338
339     return surf
```

```

340
341     class CubeGuide(Cube3D): 	usage
342         """
343             Class that handles the guide cube and adds instructions
344
345             This class inherits from Cube3D and overrides the get_image method
346         """
347
348     @classmethod
349     def get_image(cls):
350         """
351             Creates the image of the default cube with added instructions
352
353             :return: the cube image, 600*600
354             :rtype: pygame.Surface
355         """
356
357         surf = pygame.Surface((600, 600))
358         surf.fill(default_colour)
359         colour = gd.guide_arrow_colour
360         if default_colour == colour:
361             # arrows will blend into background
362             print("BAD idea, change guide arrow colour first")
363
364     def arrow_top(text, angle=0):
365         """
366             Draws an arrow aligned with the cubes slant on the top edge
367
368             :param text: text to draw above the arrow
369             :param angle: clockwise angle to rotate the arrow, 0 is upwards
370             :type text: str
371             :type angle: int
372             :return: the image of the arrow, 100*100
373             :rtype: pygame.Surface
374
375         surf = pygame.Surface((100, 100))
376         surf.fill(default_colour)
377         pygame.draw.polygon(
378             surface=surf,
379             color=colour,
380             points=((13, 13), (50, 0), (63, 63), (50, 50), (50, 93), (25, 80), (25, 25)),
381         )
382         surf.set_colorkey(default_colour)
383         # angle
384         surf = pygame.transform.rotate(surface=surf, angle=angle)
385         # letter
386         surf.blit(
387             source=interface.text(
388                 text=text,
389                 font=gd.guide_font,
390                 foreground_colour=BLACK,
391                 background_colour=gd.default_colour,
392             ),
393             dest=(15, 0),
394         )
395         return surf

```

```
396     def arrow_right(text, angle=0):
397         """
398             Draws an arrow aligned with the cubes slant on the right edge
399
400             :param text: text to draw above the arrow
401             :param angle: clockwise angle to rotate the arrow, 0 is right
402             :type text: str
403             :type angle: int
404
405             :return: the image of the arrow, 100*100
406             :rtype: pygame.Surface
407
408
409         surf = pygame.Surface((100, 100))
410         surf.fill(default_colour)
411         pygame.draw.polygon(
412             surface=surf,
413             color=colour,
414             points=[
415                 (38, 50),
416                 (63, 25),
417                 (63, 38),
418                 (100, 38),
419                 (100, 63),
420                 (63, 63),
421                 (63, 75),
422             ],
423         )
424         surf.set_colorkey(default_colour)
425         # angle
426         surf = pygame.transform.rotate(surface=surf, angle=angle)
427         # letter
428         surf.blit(
429             source=interface.text(
430                 text=text,
431                 font=gd.guide_font,
432                 foreground_colour=BLACK,
433                 background_colour=gd.default_colour,
434             ),
435             dest=(35, 25),
436         )
437
438         return surf
```

```
437     def arrow_rotate(text, angle=0):
438         """
439             Draws a large straight arrow
440
441             :param text: text to draw above the arrow
442             :param angle: clockwise angle to rotate the arrow, 0 is right
443             :type text: str
444             :type angle: int
445             :return: the image of the arrow, 100*100
446             :rtype: pygame.Surface
447         """
448
449         surf = pygame.Surface((200, 100))
450         surf.fill(default_colour)
451         pygame.draw.polygon(
452             surface=surf,
453             color=colour,
454             points=(
455                 (200, 50),
456                 (150, 100),
457                 (150, 75),
458                 (0, 75),
459                 (0, 25),
460                 (150, 25),
461                 (150, 0),
462             ),
463             )
464         surf.set_colorkey(default_colour)
465         # angle:
466         surf = pygame.transform.rotate(surface=surf, angle=angle)
467         # letter
468         surf.blit(
469             source=interface.text(
470                 text=text,
471                 font=gd.guide_font,
472                 foreground_colour=BLACK,
473                 background_colour=gd.default_colour,
474             ),
475             dest=(0, 0),
476         )
477         return surf
```

```
478     # offsets allow moving cube and arrows
479     # whilst maintaining thier relative position to each other
480     cube_offset_x = 100
481     cube_offset_y = 50
482
483     # cube
484     surf.blit( source: super().get_image(True), dest: (cube_offset_x, cube_offset_y))
485
486     # up
487     surf.blit( source: arrow_top("Q"), dest: (cube_offset_x + 30, cube_offset_y + 13))
488     surf.blit( source: arrow_top("W"), dest: (cube_offset_x + 90, cube_offset_y + 45))
489     surf.blit( source: arrow_top("E"), dest: (cube_offset_x + 150, cube_offset_y + 73))
490
491     # left
492     surf.blit( source: arrow_right("R"), dest: (cube_offset_x + 100, cube_offset_y + 150))
493     surf.blit( source: arrow_right("F"), dest: (cube_offset_x + 100, cube_offset_y + 200))
494     surf.blit( source: arrow_right("V"), dest: (cube_offset_x + 100, cube_offset_y + 250))
495
496     # right
497     surf.blit( source: arrow_right( text: "T", angle: 180), dest: (cube_offset_x + 205, cube_offset_y + 152))
498     surf.blit( source: arrow_right( text: "G", angle: 180), dest: (cube_offset_x + 205, cube_offset_y + 202))
499     surf.blit( source: arrow_right( text: "B", angle: 180), dest: (cube_offset_x + 205, cube_offset_y + 252))
500
501     # down
502     surf.blit( source: arrow_top( text: "A", angle: 180), dest: (cube_offset_x, cube_offset_y + 250))
503     surf.blit( source: arrow_top( text: "S", angle: 180), dest: (cube_offset_x + 50, cube_offset_y + 277))
504     surf.blit( source: arrow_top( text: "D", angle: 180), dest: (cube_offset_x + 100, cube_offset_y + 305))
505
506     # rotate
507     surf.blit( source: arrow_rotate("X"), dest: (cube_offset_x + 50, cube_offset_y + 400))
508     surf.blit( source: arrow_rotate( text: "Y", angle: 90), dest: (cube_offset_x - 100, cube_offset_y + 100))
509     surf.blit( source: arrow_rotate( text: "Z", angle: 335), dest: (cube_offset_x + 250, cube_offset_y - 50))
510
511     return surf
512
```

```
513
514     def turn(row_col, number, backwards=False, ignore_moves=False):
515         """
516             Turn 1 row or column once in a given direction, default is right/up
517
518             :param row_col: row is True, column is False
519             :param number: the number to do, left to right or top to bottom
520             :param backwards: do the opposite of the move/do the move 3 times if true
521             :param ignore_moves: don't add the move to the moves list
522             :type row_col: bool
523             :type number: int
524             :type backwards: bool
525             :type ignore_moves: bool
526             :rtype: None
527         """
528
529         if not ignore_moves:
530             # add the move to the moves list
531             # ignoring is useful for solving
532             gd.moves.push({"direction": row_col, "number": number, "backwards": backwards})
533             gd.move_count += 1
534         else:
535             gd.move_count -= 1
536
537         # loop to turn the row or column the correct number of times
538         loop = 1
539         if backwards: # 3 right is used to achieve 1 left, 3 up to achieve 1 down
540             loop = 3
```

```

542     for _ in range(loop):
543         # make copies of the faces of the cube so the original state isn't lost
544         # deepcopy prevents pass by reference shenanigans
545         # by copying the value instead of creating a reference
546         face0 = copy.deepcopy(gd.used_cube[0])
547         face1 = copy.deepcopy(gd.used_cube[1])
548         face2 = copy.deepcopy(gd.used_cube[2])
549         face3 = copy.deepcopy(gd.used_cube[3])
550         face4 = copy.deepcopy(gd.used_cube[4])
551         face5 = copy.deepcopy(gd.used_cube[5])
552
553         n = number
554
555         if row_col: # turn the row
556             (
557                 gd.used_cube[2][n],
558                 gd.used_cube[3][n],
559                 gd.used_cube[0][n],
560                 gd.used_cube[1][n],
561             ) = (
562                 face1[n],
563                 face2[n],
564                 face3[n],
565                 face0[n],
566             )
567
568         if number == 0: # rotate the top face
569             gd.used_cube[4] = numpy.rot90(m: gd.used_cube[4], k=1, axes=(0, 1))
570         elif number == 2: # rotate the bottom face
571             gd.used_cube[5] = numpy.rot90(m: gd.used_cube[5], k=1, axes=(1, 0))
572         else: # turn the column
573             for i in range(3):
574                 gd.used_cube[1][i][n] = face5[i][n]
575                 # 2-i flips the row number for the back
576                 # 2 - n flips the column number for the back
577                 gd.used_cube[5][2 - i][n] = face3[i][2 - n]
578                 gd.used_cube[3][2 - i][2 - n] = face4[i][n]
579                 gd.used_cube[4][i][n] = face1[i][n]
580
581         if number == 0: # rotate left face
582             gd.used_cube[0] = numpy.rot90(m: gd.used_cube[0], k=1, axes=(0, 1))
583         elif number == 2: # rotate right face
584             gd.used_cube[2] = numpy.rot90(m: gd.used_cube[2], k=1, axes=(1, 0))

```

```

585
586     def rotate(axis, ignore_moves=False):
587         """
588             Rotates the view of the cube without changing layout
589
590             :param axis: x, y, z
591             :type axis: str
592             :param ignore_moves: whether to add the move to the moves list, defaults to False
593             :type ignore_moves: bool or optional
594             :rtype: None
595         """
596
597         # make copies of the faces of the cube so the original state isn't lost
598         # deepcopy prevents pass by reference shenanigans
599         # by copying the value instead of creating a reference
600         face0 = copy.deepcopy(gd.used_cube[0])
601         face1 = copy.deepcopy(gd.used_cube[1])
602         face2 = copy.deepcopy(gd.used_cube[2])
603         face3 = copy.deepcopy(gd.used_cube[3])
604         face4 = copy.deepcopy(gd.used_cube[4])
605         face5 = copy.deepcopy(gd.used_cube[5])
606
607         if not ignore_moves: # add the move to the moves list
608             # ignoring is useful for solving
609             gd.moves.push({"rotation": True, "direction": axis})
610             gd.move_count += 1
611         else:
612             gd.move_count -= 1
613
614         if axis == "x":
615             for i in range(3): # equivalent to a rotation along the x axis
616                 turn( row_col=True, number=i, ignore_moves=True)
617             elif axis == "y":
618                 for i in range(3): # equivalent to a rotation along the y axis
619                     turn( row_col=False, number=i, ignore_moves=True)
620             elif axis == "z": # equivalent to a rotation along the z axis
621                 # rotate the front and back faces
622                 gd.used_cube[1] = numpy.rot90( m: gd.used_cube[1], k=1, axes=(1, 0))
623                 gd.used_cube[3] = numpy.rot90( m: gd.used_cube[3], k=1, axes=(0, 1))
624
625                 # required a lot of manual testing
626                 # carefully test any changes
627                 for j in range(3):
628                     for i in range(3):
629                         gd.used_cube[0][j][2 - i] = face5[i][j]
630                         gd.used_cube[4][j][2 - i] = face0[i][j]
631                         gd.used_cube[2][j][2 - i] = face4[i][j]
632                         gd.used_cube[5][j][2 - i] = face2[i][j]

```

Features

The features move from the old cube file to this file remain largely the same, with only small changes to them to due to not being in the same file as the data structures. However, there are two new classes at the end of the file for displaying the game history and leaderboard.

```
1  """
2  This file contains all the features of the program available to the user
3
4  These provide additional functionality
5  beyond the basic turn and rotation functions of the cube
6
7  black, isort and flake8 used for formatting
8  """
9
10 import time
11 from random import randint
12
13 import game_data as gd
14 import interface
15 import numpy
16 import pygame
17 import tools
18 import user_data as ud
19 from cube import rotate, turn
20 from game_data import BLACK, default_colour, default_cube, default_font
21 from validation import ValidateScreenPositions
22
23 val = ValidateScreenPositions( width: 1600, height: 900)
24
25
26 def scramble():
27     """
28         Randomly scrambles the cube by making between 15 and 25 moves randomly
29
30     :rtype: None
31     """
32     count = randint( a: 15, b: 25)
33     gd.scrambler_count = count
34     for _ in range(count):
35         # randomise every aspect of the turn
36         direction = bool(randint( a: 0, b: 1))
37         number = randint( a: 0, b: 2)
38         backwards = bool(randint( a: 0, b: 1))
39
40         turn( row_col: direction, number: number, backwards: backwards)
41
```

```

42
43     class Solver:
44
45         """
46
47             Solve the cube, one turn per game loop
48
49             The solve function must be called once per game loop
50             until it returns False
51             to completely solve the cube
52
53             The attribute first should be updated to True before each complete solve
54
55             A solve can optionally be made to take 5 seconds. To do this, implement a
56             time.sleep(this_object.sleep_time) before the this_object.solve() call
57             """
58
59         def __init__(self):
60             self.first = True
61             """If it is the first move of the solve
62             :type: bool"""
63             self.sleep_time = 0.2
64             """The amount of time to wait between each move
65             :type: float"""
66
66         def solve(self):
67             """
68                 Does the reverse of the last done move and removes it from the moves list
69
70                 :return: False if the cube is solved, True otherwise
71                 :rtype: bool
72             """
73
74             # guard clause
75             if gd.moves.size() == 0 or self.check_solved():
76                 return False
77
78             # calculate time to wait between move
79             if self.first:
80                 if gd.moves.size() > 0:
81                     # every solve should take 5 seconds regardless of moves required,
82                     # although this can be affected by hardware limitations
83                     self.sleep_time = 5 / gd.moves.size()
84                     self.first = False
85                 else:
86                     # wait upon every button press so the user knows it has 'worked'
87                     # even when the cube is already solved
88                     self.sleep_time = 1
89
90             return self.pop_move()

```

```

90     @staticmethod
91     def check_solved():
92         """
93             Checks whether the cube is in a solved state
94
95             :return: True if the cube is solved, False otherwise
96             :rtype: bool
97         """
98
99         not_solved = False
100        for i in range(6): # face
101            for j in range(3): # row
102                for k in range(3): # column
103                    # checks for any square not the same colour
104                    # as the middle square on the same face
105                    # numpy.all handles it being a tuple comparison
106                    if not numpy.all(gd.used_cube[i][j][k] == gd.used_cube[i][1][1]):
107                        not_solved = True
108
109        # sys.exit()
110        return not not_solved
111
112    @staticmethod: 2 usages
113    def pop_move():
114        """
115            Removes a move from the moves list and does the reverse
116
117            :return: False if the cube is solved, True otherwise
118            :rtype: bool
119        """
120
121        # guard clause
122        if gd.moves.size() == 0:
123            return False
124
125        move = gd.moves.pop() # get the move dictionary
126        if "rotation" in move.keys(): # check if the move was a rotation
127            # rotate does not have a backwards parameter,
128            # so achieve via 3 'forward' turns
129            for _ in range(3):
130                # ignore move as it is part of the solve, not the user or scramble
131                rotate( axis=move["direction"], ignore_moves=True)
132        else: # if not rotation must be turn
133            # not move["backwards"] to always undo the move
134            # ignore move as part of solve
135            turn(
136                row_col=move["direction"],
137                number=move["number"],
138                backwards=not move["backwards"],
139                ignore_moves=True
140            )
141        if gd.moves.size() == 0: # must be solved
142            return False
143        else:
144            return True # continue solving

```

```

143
144     class Timer: 1usage
145         """This class handles timing how long it takes the user to complete a solve"""
146
147     def __init__(self):
148         self.start_time = 0.0
149             """The time since epoch that the timer was started
150             :type: float"""
151         self.end = 0.0
152             """The time since epoch that the timer was stopped
153             :type: float"""
154         self.elapsed = 0.0
155             """The amount of time that has elapsed since the timer was started
156             :type: float"""
157         self.exists = False
158             """Whether the timer has ever been started for this solve
159             :type: bool"""
160         self.running = False
161             """Whether the timer is actively running
162             :type: bool"""
163
164     def start(self):
165         """Starts the timer and marks it as running"""
166         self.exists = True
167         self.running = True
168         self.start_time = time.time()
169         gd.start_time = self.start_time
170
171     def stop(self):
172         """Gets the final time elapsed and stops the timer"""
173         self.update()
174         self.running = False
175
176     def delete(self):
177         """Marks the timer as not having run for the current solve"""
178         self.exists = False
179         self.running = False
180         gd.time_taken = 0.0
181         gd.start_time = 0.0
182
183     def update(self):
184         """Updates the time elapsed if the timer is running"""
185         if self.running:
186             self.end = time.time()
187             self.elapsed = self.end - self.start_time
188             gd.time_taken = self.elapsed

```

```
190     def display_elapsed(self): 1 usage
191     """
192         Creates a text image displaying the time elapsed
193
194         :return: The text image
195         :rtype: pygame.Surface
196     """
197
198     # if time is less than a minute
199     if self.elapsed < 60: # display time as seconds and milliseconds
200         image = interface.text(
201             text: str(round(self.elapsed, 3)) + " seconds", # round to milliseconds
202             font: gd.default_font,
203             foreground_colour: BLACK,
204             background_colour: default_colour,
205         )
206     else: # display time as minutes and seconds
207         image = interface.text(
208             text: str(int(self.elapsed / 60)) # minutes
209             + "m"
210             + str(int(self.elapsed % 60)) # seconds
211             + "s",
212             font: gd.default_font,
213             foreground_colour: BLACK,
214             background_colour: default_colour,
215         )
216
217     return image
```

```

218
219     class DisplayHistory: 1usage
220         """This class manages fetching and displaying the user's game history"""
221
222     def __init__(self, screen, pos):
223         """
224             :param screen: The screen that this is to be blitted to
225             :param pos: The top-left position that this is to be blitted to: x,y
226             :type screen: pygame.Surface
227             :type pos: list[int] or tuple[int, int]
228         """
229         self.screen = screen
230         self.pos = pos
231
232         self.history = []
233         """A 2D array where each row is a game and each column is text to display
234             :type: list[list]"""
235
236         self.y_offset = 0
237         """The amount the image should be offset vertically
238             - the amount it has been scrolled
239             :type: int"""
240
241     def format_history(self): 1usage
242         """Formats the user's game history into a 2D array
243             that contains elements to be displayed"""
244         history_data = vd.game_history.get_history()
245
246         for i in range(len(history_data)): # one game
247             game = history_data[i]
248             game_array = []
249
250             # date of solve
251             game_array.append(str(
252                 time.strftime( format: "%d/%m/%Y", time_tuple: time.localtime(game[5])))
253             ))
254
255             # solved or unsolved
256             if game[6]:
257                 game_array.append("SOLVED")
258             else:
259                 game_array.append("UNSOLVED")
260
261             # move count for the user
262             game_array.append(str(game[1] - game[3]))
263
264             # time taken
265             game_array.append(str(
266                 time.strftime( format: "%H:%M:%S", time_tuple: (time.gmtime(game[4]))))
267             ))
268
269             # hints used
270             game_array.append(str(game[7]))
271
272             self.history.append(game_array)
273
274

```

The display history class manages getting the user's game history and displaying it to the screen. The image it creates is scrollable.

The format_history function creates a 2D array, where the outer index is games, and the inner index is specific information. Only certain information has been taken from the user's game history, and it all has been converted to a format the user will understand (e.g. time into hours, minutes, seconds) and then to a string that can be rendered.

I would have liked to make the time displayed include milliseconds, however the time library I had been using for times did not support this and I unfortunately did not have time to figure a way to implement this.

```

275     def get_image(self):
276         """
277             Creates a text image displaying the user's game history
278
279             This will get and format the user's game history before creating the image
280         """
281
282         self.format_history()
283
284         img_height = 0 # will vary on size of history
285         img_list = [] # will vary on size of history, to be added to returned surf
286
287         # header
288         img = interface.text(
289             text=" DATE | STATE | MOVES | TIME | HINTS USED ",
290             font=default_font,
291             foreground_colour=BLACK,
292             background_colour=default_colour,
293         )
294         img_height += img.get_height()
295         img_width = img.get_width()
296         img_list.append(img)
297
298         for i in range(len(self.history)):
299             img = interface.text(
300                 text=self.history[i][0]
301                 + " | "
302                 + self.history[i][1]
303                 + " | "
304                 + self.history[i][2]
305                 + " | "
306                 + self.history[i][3]
307                 + " | "
308                 + self.history[i][4],
309                 font=default_font,
310                 foreground_colour=BLACK,
311                 background_colour=default_colour,
312             )
313             img_height += img.get_height()
314             img_list.append(img)
315
316         surf = pygame.Surface((img_width, img_height))
317         surf.fill(default_colour)
318         for i in range(len(img_list)):
319             surf.blit(source=img_list[i], dest=(0, i * img_list[i].get_height()))
320
321         return surf

```

The `get_image` function uses the formatted list to create the image. It creates an image for each game that is just a rendered string and uses the `get_height` function and a variable to ensure the `surf` created is big enough, as the number of images being drawn to it is variable.

As the images are being created before the `surf` is, they are stored in a list, then iteratively blitted to the `surf` once it is created.

```
322     def update(self):
323         """
324             Updates the history image and blits it to the screen
325
326             This takes into account the y_offset (amount scrolled) and adjusts it vertically
327         """
328         img = self.get_image()
329         self.screen.blit(
330             source= img, dest=[
331                 self.pos[0] - img.get_width() // 2,
332                 self.pos[1] + self.y_offset
333             ]
334         )
335
336     def scroll(self, amount): 1 usage
337         """
338             Changes the y position the image is blitted to,
339             which allows it to be scrolled
340
341             :param amount: The amount to scroll by, positive or negative
342             :type amount: int
343         """
344         self.y_offset += amount
345
```

The update function doesn't used validation, despite the fact it has a changing and thus non-validated screen position it blits to, as it is intended that the surf will be blitted offscreen if scrolled far enough.

The scroll function updates the y offset so that the image can be scrolled.

```

346
347     class Leaderboard: 4 usages
348     """This class manages creating and displaying the leaderboard"""
349
350     class Entry:
351         """
352             This class represents an entry in the leaderboard
353
354             This is designed to be used by tools.File
355             and as such all its attributes must also be parameters
356         """
357
358         def __init__(self, id, name, time, moves):
359             """
360                 :param id: a unique identifier for each object
361                 :type id: int
362                 :param name: the username of the player
363                 :type name: str
364                 :param time: the time taken to solve the cube
365                 :type time: float
366                 :param moves: the number of moves the user did to solve the cube
367                 :type moves: int
368             """
369
370             self.id = id
371             self.name = name
372             self.time = time
373             self.moves = moves
374
375             leaderboard_file = tools.File(file="leaderboard.txt", cls=Entry)
376
377         def __init__(self, screen, pos):
378             """
379                 :param screen: the screen that this is to be blitted to
380                 :type screen: pygame.Surface
381                 :param pos: the centre position that this is to be blitted to: x,y
382                 :type pos: list[int] or tuple[int, int]
383             """
384
385             self.screen = screen
386             self.pos = pos
387
388             self.entries = self.leaderboard_file.get_list()
389             """The top ten quickest solve times, should be kept in in order
390             :type: list[Entry]"""
391             self.sort()

```

The leaderboard class manages creating and displaying the leaderboard. It contains another class that holds a single entry to the leaderboard, and that class also conforms to the specifications in the tools.File class that is being used to store the top ten scores, which includes having a unique, if meaningless id. Ideally the id would have some significance however its possible for multiple entries to be from the same user, or have the same time taken, or have the same number of moves done, so I had to create a separate variable for the id.

The entries are sorted after being obtained as the tools.File will store them according to their id, not their times.

```
391     def update_list(self, time, moves):  # usage
392         """
393             Checks if the user has a leaderboard worthy time and updates the ordered list
394
395             :param time: the time taken to solve the cube
396             :type time: float
397             :param moves: the number of moves the user did to solve the cube
398             :type moves: int
399
400             ...
401
402             if len(self.entries) < 10: # add new entry
403                 self.entries.append(
404                     Leaderboard.Entry(
405                         id: len(self.entries),
406                         name: ud.Manager.username,
407                         time: time,
408                         moves: moves
409                     )
410
411             elif self.entries[-1].time <= time: # if no new entry, stop
412                 return
413
414             elif self.entries[-1].time > time: # if new entry replace slowest
415                 self.entries[-1] = Leaderboard.Entry(
416                     id: self.entries[-1].id,
417                     name: ud.Manager.username,
418                     time: time,
419                     moves: moves
420
421             self.sort()
422             self.leaderboard_file.replace_list(self.entries)
423             self.leaderboard_file.save()
```

The update list function manages checking new scores against the existing ones and adding or replacing scores if necessary. If the leaderboard isn't full the solve is guaranteed to be added. When this is done the entry is given the id equal to the length entries list as this id cannot exist yet unless an entry is deleted without another being added, which shouldn't happen.

If the slowest entry is quicker than or equal to the solve being checked the function exits as the entries aren't going to change. This is \leq as if two solves have taken the same amount of time I believe the solve done first should be the one to stay, if there not going to both be on the leaderboard. As this is checking against the slowest entry, only one of them can be on the leaderboard.

The final elif could have been an else, but I used an elif to make it clear when this will execute. It creates a new entry to replace the slowest one. It uses the same id as entry its replacing to ensure the list doesn't end up with duplicate ids.

```
424     def sort(self):
425         """Sorts the entries list by time"""
426         new_entries = []
427         for i in range(len(self.entries)):
428             j = 0
429             added = False
430             while j <= len(new_entries):
431                 if self.entries[i].time < new_entries[j].time:
432                     # will move slower to end
433                     new_entries.insert(j, self.entries[i])
434                     added = True
435                     break
436                 j += 1
437
438             if not added: # ensures the Entry is added,
439                 # this will always be the slowest in new_entries
440                 new_entries.append(self.entries[i])
441
442         self.entries = new_entries
443
```

```
424     def sort(self):
425         """Sorts the entries list by time"""
426         self.entries.sort(key=lambda Entry: Entry.time)
427
```

To sort the list, I had originally created the above insertion sort style code, however I then realised the built-in sort function could be passed a keyword argument 'key' which would allow me to use it to sort by the time attribute of the entry classes. As this is more efficient than my function, I used this instead, although I placed it within my sort function so I did not have to replace any of the self.sort calls.

```

428     def update(self):
429         """Updates the leaderboard image and blits it to the screen"""
430         img = self.get_image()
431         self.screen.blit(source=img, dest=img.get_rect(center=self.pos))
432
433     def get_image(self):
434         """Creates the image of the leaderboard"""
435         img_height = 0
436         img_list = []
437
438         # header
439         img = interface.text(
440             text=" POSITION | NAME | TIME | MOVES ",
441             font=default_font,
442             foreground_colour=BLACK,
443             background_colour=default_colour,
444         )
445         img_height += img.get_height() # ensures the surd isn't too small
446         img_width = img.get_width()
447         img_list.append(img)
448
449         for i in range(len(self.entries)):
450             img = interface.text(
451                 text=str(i + 1)
452                 + " | "
453                 + self.entries[i].name
454                 + " | "
455                 + str(round(self.entries[i].time, 3))
456                 + " | "
457                 + str(self.entries[i].moves),
458                 font=default_font,
459                 foreground_colour=BLACK,
460                 background_colour=default_colour,
461             )
462             img_height += img.get_height()
463             img_list.append(img)
464
465         surf = pygame.Surface((img_width, img_height))
466         surf.fill(default_colour)
467         for i in range(len(img_list)):
468             surf.blit(source=img_list[i], dest=(0, i * img_list[i].get_height()))
469
470         return surf
471

```

The `get_image` function works similar to the `DisplayHistory`'s `get_image` method, but it isn't scrollable, and values used are different.

Game Data

The game_data file exists to hold and manage global data.

```
1  """
2  This file contains global data and settings information
3
4  This data is used by multiple files in the program. It may be edited here, or it may be
5  provided to the user as settings for them to change.
6
7  black, isort and flake8 used for formatting
8  """
9  import copy
10
11 import pygame
12 from pygame import freetype
13
14 pygame.font.init()
15 pygame.freetype.init()
16
17 # colours
18 BLACK = (0, 0, 0)
19 WHITE = (255, 255, 255)
20 YELLOW = (255, 255, 0)
21 ORANGE = (255, 165, 0)
22 RED = (255, 0, 0)
23 GREEN = (0, 255, 0)
24 BLUE = (0, 0, 255)
25 GREY = (169, 169, 169)
26
27 default_colour = GREY
28 guide_arrow_colour = BLACK
29
30
31 # fonts
32 default_font = pygame.freetype.SysFont(name: "calibri", size: 20)
33 guide_font = pygame.freetype.SysFont(name: "calibri", size: 20, bold=True)
34
```

```

35
36     # cube design
37     # split into sides as easier to write
38     up = [
39         [WHITE, WHITE, WHITE],
40         [WHITE, WHITE, WHITE],
41         [WHITE, WHITE, WHITE],
42     ]
43     down = [
44         [YELLOW, YELLOW, YELLOW],
45         [YELLOW, YELLOW, YELLOW],
46         [YELLOW, YELLOW, YELLOW],
47     ]
48
49     left = [
50         [ORANGE, ORANGE, ORANGE],
51         [ORANGE, ORANGE, ORANGE],
52         [ORANGE, ORANGE, ORANGE],
53     ]
54
55     right = [
56         [RED, RED, RED],
57         [RED, RED, RED],
58         [RED, RED, RED],
59     ]
60
61     front = [
62         [GREEN, GREEN, GREEN],
63         [GREEN, GREEN, GREEN],
64         [GREEN, GREEN, GREEN],
65     ]
66
67     back = [
68         [BLUE, BLUE, BLUE],
69         [BLUE, BLUE, BLUE],
70         [BLUE, BLUE, BLUE],
71     ]
72
73     # so a default cube may always be shown and to check against for solves
74     default_cube = [
75         left,
76         front,
77         right,
78         back,
79         up,
80         down,
81     ]
82     # deepcopy passes by value, not reference, ensuring default_cube is not changed
83     used_cube = copy.deepcopy(default_cube)
84

```

```

85
86     # used for tracking moves and 'solving' the cube
87     class MoveStack:
88         """A stack for managing the moves made by the user and scrambler"""
89
90         def __init__(self):
91             self.stack = []
92
93         def push(self, move):
94             """
95                 Pushes a move onto the stack
96
97                 :param move: move should be in the format
98                 {
99                     "direction": True for row, False for column,
100                     "number": row or column number,
101                     "backwards": If the move was backwards (left or down)
102                 }
103                 for a turn or the following for a rotation:
104                 {
105                     "rotation": True,
106                     "direction": "x" or "y" or "z"
107                 }
108                 :type move: dict
109             """
110
111             if move.keys() == {"direction", "number", "backwards"} or move.keys() == {
112                 "rotation",
113                 "direction",
114             }:
115                 self.stack.append(move)
116             else:
117                 raise ValueError("Invalid dict keys")

```

I decided to make a class for the move stack so I could ensure that any moves added do have the correct structure. As such, the docstring specifies the two acceptable dictionary format and checks the given parameter matches one of these, and raises an error if it doesn't.

```
118     def pop(self):
119         """
120             Pops a move off the stack
121
122             :return: move
123             :rtype: dict
124             """
125
126         return self.stack.pop()
127
128     def clear(self):
129         """Clears the stack"""
130         self.stack = []
131
132     def size(self):
133         """
134             :return: size of the stack
135             :rtype: int
136             """
137
138         return len(self.stack)
139
140     def get_stack(self): 4 usages
141         """
142             :return: the list of moves stored as dictionaries
143             :rtype: list[dict]
144             """
145
146     def set_stack(self, stack): 1 usage
147         """
148             Replaces the current stack with the one provided
149
150             :param stack: the list of moves stored as dictionaries
151             :type stack: list[dict]
152             """
153
154         self.stack = stack
```

The stack also contains some basic stack functions.

```
154
155     moves = MoveStack()
156     """The MoveStack of moves that have been made by the user and the scrambler in order
157     :type: MoveStack"""
158     move_count = 0
159     """The amount of moves made by the user and scrambler.
160     These will be in order in the moves list, but will be preceded by scrambler moves
161     :type: int"""
162     scrambler_count = 0
163     """The amount of moves made by the scrambler
164     :type: int"""
165
166     # used for tracking time
167     start_time = 0.0
168     """The time since epoch that the user started the solve/ started the scrambler
169     :type: float"""
170     time_taken = 0.0
171     """The amount of time that has elapsed since the user started the solve
172     :type: float"""
173
174     # used for seeing if the solve is eligible for the leaderboard and for users knowledge
175     hints_used = False
176     """Whether the user has used hints
177     :type: bool"""
178     solver_used = False
179     """Whether the user has used the solver
180     :type: bool"""
181     solved = False
182     """Whether the cube is solved
183     :type: bool"""
184
```

Interface

The only changes to the interface file were minor docstring edits on lines 37, 73, 75, 165, 166.

```

1  """
2     This file contains some key elements of the interface to be used by other files
3
4     This file handles creating visual elements and user interface
5     to be displayed to the screen for the user.
6     DisplayOption and DisplayBar should be used together.
7
8     black, isort and flake8 used for formatting
9 """
10
11
12 import pygame
13
14
15 class DisplayOption:
16     """
17         Creates a button with an image that changes size when hovered
18
19         This class should be used with DisplayBar
20     """
21
22     def __init__(self, image_function, display_surf, pos, size, mult, action, bg_col):
23         """
24             :param image_function: the function to get the image to use as the button
25             :param display_surf: the surface to display the button to
26             :param pos: the position to display the button from the top left
27             :param size: the x length and y length of the button
28             :param mult: how much to increase the image size when hovered
29             :param action: the function to run when the button is clicked
30             :param bg_col: the RRB value of the background colour
31             :type image_function: function
32             :type display_surf: pygame.Surface
33             :type pos: list[int] or tuple[int, int]
34             :type size: list[int]
35             :type mult: float
36             :type action: function
37             :type bg_col: tuple[int, int, int] or list[int]
38         """
39
40         self.image_function = image_function
41         self.display_surf = display_surf
42         self.pos = pos
43         self.size = size
44         self.last_size = size
45         self.mult = mult
46         self.act = action
47         self.bg_col = bg_col
48         self.image = self.get_image()
49
50         self.last_size = self.size
51         """The last x,y size of the button. Used for checking if the button is hovered
52         :type last_size: list[int]"""

```

```
53     def get_image(self):
54         """
55             Gets the image of the button in its current state
56
57         :return: the image of the button
58         :rtype: pygame.Surface
59         """
60
61         surf = pygame.Surface(self.size)
62         cube = self.image_function()
63         cube = pygame.transform.smoothscale(surface=cube, size=self.size)
64         cube.set_colorkey(self.bg_col)
65         surf.blit(source=cube, dest=(0, 0))
66
67     return surf
```

```

67     def update(self, mouse_pos, offset, mouse_up):
68         """
69             Update the button, checking if it is hovered or clicked
70
71             :param mouse_pos: the x,y position of the mouse
72             :param offset: the width and height to offset the button ensures its enlarged
73                 size does not overlap anything
74             :param mouse_up: whether the mouse button has been clicked
75             :type mouse_pos: tuple[int, int] or list[int]
76             :type offset: list[int]
77             :type mouse_up: bool
78             :return: whether the button is hovered
79             :rtype: bool
80         """
81
82         # calculate the position of the button with its offset
83         pos = [0, 0]
84         pos[0] = self.pos[0] + offset[0]
85         pos[1] = self.pos[1] + offset[1]
86
87         # calculate the centre of the button accounting for possible enlargement
88         # and offset
89         width = self.last_size[0]
90         height = self.last_size[1]
91         centre = width // 2 + pos[0], height // 2 + pos[1]
92
93         if self.image.get_rect(center=centre).collidepoint(mouse_pos): # if hovered
94             if mouse_up: # if pressed
95                 self.act()
96                 # save same size so it can be restored
97                 temp = self.size.copy()
98                 # enlarge the button
99                 self.size[0], self.size[1] = (
100                     self.size[0] * self.mult,
101                     self.size[1] * self.mult,
102                 )
103                 self.last_size = self.size
104                 # get the enlarged image
105                 self.image = self.get_image()
106                 # restore size to the original state so it can be displayed
107                 self.size = temp
108
109                 self.display_surf.blit(source=self.image, dest=pos)
110                 return True
111             else: # if not hovered
112                 self.image = self.get_image()
113                 self.last_size = self.size
114                 self.display_surf.blit(source=self.image, dest=pos)
115             return False

```

```
116
117     class DisplayBar:
118         """For creating a bar of DisplayObject in a row/column"""
119
120     def __init__(self, object_list, row):
121         """
122             :param object_list: list of DisplayOption in sequential order
123             :param row: if the buttons are in a row(True) or column(False)
124             :type object_list: list[DisplayOption]
125             :type row: bool
126         """
127
128         self.object_list = object_list
129         self.row = row
130
131     def update(self, mouse_pos, mouse_up):
132         """
133             Updates each button in the bar and offsets then if one is hovered
134
135             :param mouse_pos: the x,y position of the mouse
136             :param mouse_up: whether the mouse button has been clicked
137             :type mouse_pos: tuple[int, int] or list[int]
138             :type mouse_up: bool
139             :rtype: None
140         """
141
142         offset = [0, 0]
143         for i in range(len(self.object_list)):
144             # update the button and check if it is hovered
145             if self.object_list[i].update(mouse_pos=mouse_pos, offset=offset, mouse_up=mouse_up):
146                 if self.row:
147                     # set offset to the difference in size
148                     offset[0] = (
149                         self.object_list[i].last_size[0] - self.object_list[i].size[0]
150                     )
151                 else: # column
152                     offset[1] = (
153                         self.object_list[i].last_size[1] - self.object_list[i].size[1]
154                     )
```

```
154 def text(text, font, foreground_colour, background_colour):
155     """
156     Returns an image of the text
157
158     :param text: the text to display
159     :param font: the font to use
160     :param foreground_colour: the RGB value of the foreground colour
161     :param background_colour: the RGB value of the background colour
162     :type text: str
163     :type font: pygame.freetype.Font
164     :type foreground_colour: tuple[int, int, int] or list[int]
165     :type background_colour: tuple[int, int, int] or list[int]
166     :return: the image of the text
167     :rtype: pygame.Surface
168     """
169
170     # render returns surface, rect so we only need surface
171     surface, _ = font.render(
172         text=text, fgcolor=foreground_colour, bgcolor=background_colour
173     )
174     image = surface.convert_alpha() # optimisation
175
176     return image
```

User Data

The user data file was created to manage storing a user's data, including creating a list of their game history.

```
1  """
2  This file handles loading and saving user data
3
4  This file handles a user's game history, details about their current game,
5  as well as loading and saving data to a file
6
7  black, isort and flake8 used for formatting
8  """
9
10 import game_data as gd
11 import tools
12
13
14 # game history
15 class History:
16     """This class manages the game history of the user"""
17
18     def __init__(self):
19         # do not change these as they are used for saving
20         # they are directly acquired by self.__dict__ in the add method
21         # even changing their order will break things
22         self.game_state = gd.used_cube
23
24         """The 3D array of the cube state at the last move
25         :type: list"""
26         self.move_count = gd.move_count
27
28         """The amount of moves made by the user. These will be in order in the moves list,
29         but will be preceded by scrambler moves
30         :type: int"""
31         self.moves = gd.moves.get_stack()
32
33         """The list of moves that have been made by the user and the scrambler in order
34         :type: list"""
35         self.scrambler_count = gd.scrambler_count
36
37         """The amount of moves made by the scrambler
38         :type: int"""
39         self.time_taken = gd.time_taken
40
41         """The amount of time that has elapsed since the user started the solve
42         :type: float"""
43         self.time_started = gd.start_time
44
45         """The time since epoch that the user started the solve/ started the scrambler
46         :type: float"""
47         self.solved = gd.solved
48
49         """Whether the cube is solved
50         :type: bool"""
51         self.hints_used = gd.hints_used
52
53         """Whether the user has used hints
54         :type: bool"""
55         self.solver_used = gd.solver_used
56
57         """Whether the user has used the solver
58         :type: bool"""
59
60         self.history_list = []
61
62         """The list of all history records
63         :type: list"""
64
```

As the `.__dict__` method is used the names, order, and amount of attributes cannot be changed without risking breaking things. This is not a good thing to do but it does greatly simplify the add function later on.

```
55     def add_game(self): 3 usages
56         """Adds the current game to game history using the game_data"""
57         self.game_state = gd.used_cube
58         self.move_count = gd.move_count
59         self.moves = gd.moves.get_stack()
60         self.scrambler_count = gd.scrambler_count
61         self.time_taken = gd.time_taken
62         self.time_started = gd.start_time
63         self.solved = gd.solved
64         self.hints_used = gd.hints_used
65         self.solver_used = gd.solver_used
66         # add attributes to history list
67         # excluding history list itself
68         self.history_list.append(list(self.__dict__.values())[:-1])
69
70     def replace_history(self, history_list): 1 usage
71         """
72             Replaces the history list, useful for when initialising with user's saved data
73
74             :param history_list: the new history list
75             :type history_list: list
76             """
77         self.history_list = history_list
78
79     def get_history(self): 3 usages
80         """
81             :return: the game history list
82             :rtype: list
83             """
84         return self.history_list
85
86
87     game_history = History()
88     """The class containing the history of the user's game
89     :type: History"""
90
```

The aforementioned add function – it sets its attributes to the value of their respective variable in `game_data`, effectively updating them. Then they are added to the `history_list`. Instead of manually creating the list of data to add to the `history_list`, I have used `self.__dict__.values()` to obtain the values of the attributes, then converted it to a list. This includes the `history_list` attribute itself, so that is removed by restricting the list to everything but its last value (`history_list`) with `[:-1]`. This means even changing the order of attributes would break things but, as I am the only one working on this project, and as adding or removing any variables would cause problems anyway unless every user's history is wiped, I do not think it is too severe an issue.

```
91
92     class User:
93         """
94             A class containing the user data and methods to update it
95
96             Designed for use with the Manager class and tools.File
97             """
98
99         def __init__(
100             self,
101             username=None,
102             cube_state=gd.used_cube,
103             start_time=gd.start_time,
104             time_taken=gd.time_taken,
105             moves=gd.moves.get_stack(),
106             move_count=gd.move_count,
107             scrambler_count=gd.scrambler_count,
108             hints_used=gd.hints_used,
109             solver_used=gd.solver_used,
110             history=game_history.get_history(),
111         ):
112             """
113                 :param username: the unique identifier of the user
114                 :type username: str
115                 :param cube_state: the 3D array of the cube
116                 :type cube_state: list[list[list]]
117                 :param start_time: the time since epoch when the user started the solve
118                 :type start_time: float
119                 :param time_taken: the time elapsed during the solve
120                 :type time_taken: float
121                 :param moves: the list of moves that have been made
122                 :type moves: list[dict]
123                 :param move_count: the amount of moves that has been made
124                 :type move_count: int
125                 :param scrambler_count: the amount of scrambler moves that have been made
126                 :type scrambler_count: int
127                 :param hints_used: whether the user has used hints
128                 :type hints_used: bool
129                 :param solver_used: whether the user has used the solver
130                 :type solver_used: bool
131                 :param history: the game history of the user
132                 :type history: game_data.History
133             """
134
135             # due to the way the user data is saved and loaded
136             # self. must match init param and username must be first
137             self.username = username
138             self(cube_state = cube_state
139             self.start_time = start_time
140             self.time_taken = time_taken
141             self.moves = moves
142             self.move_count = move_count
143             self.scrambler_count = scrambler_count
144             self.hints_used = hints_used
145             self.solver_used = solver_used
146             self.history = history
```

```
147     def save(self, username=None):
148         """
149             Updates this class's attributes to the current game data
150
151             Optionally updates the username
152
153             :param username: the unique identifier of the user, defaults to None (no change)
154             :type username: str, optional
155         """
156
157         if username is not None:
158             self.username = username
159             self.cube_state = gd.used_cube
160             self.start_time = gd.start_time
161             self.time_taken = gd.time_taken
162             self.moves = gd.moves.get_stack()
163             self.move_count = gd.move_count
164             self.scrambler_count = gd.scrambler_count
165             self.hints_used = (gd.hints_used,)
166             self.solver_used = gd.solver_used
167             self.history = game_history.get_history()
168
169     def load(self):
170         """Updates the current game data to this class's attributes"""
171         gd.used_cube = self.cube_state
172         gd.start_time = self.start_time
173         gd.time_taken = self.time_taken
174         gd.moves.set_stack(self.moves)
175         gd.move_count = self.move_count
176         gd.scrambler_count = self.scrambler_count
177         gd.hints_used = (self.hints_used,)
178         gd.solver_used = self.solver_used
179         game_history.replace_history(self.history)
```

The user class is designed to work with the tools.File class which also uses the `__dict__` method that history does, so it works very similarly.

```

180
181     class Manager: 22 usages
182         """This class handles data in a txt file"""
183
184         user_file = tools.File(file="saves_data.txt", cls=User)
185         username = None
186         obj = None
187
188         @staticmethod
189         def load(username):
190             """
191                 Load the user data for the given username, or create a new user
192
193                 :param username: the unique username of the user
194                 :type username: str
195             """
196
197             Manager.username = username
198             try:
199                 Manager.obj = Manager.user_file.get_object(Manager.username)
200             except tools.ObjectNotFound:
201                 Manager.obj = User(Manager.username)
202
203             Manager.obj.load()
204
205         @staticmethod
206         def save(username=None):
207             """
208                 Save the user data, and optionally change the username
209
210                 :param username: the new username, defaults to None
211                 :type username: str, optional
212             """
213
214             if username is not None:
215                 # replace the username
216                 Manager.obj = User(username)
217                 Manager.user_file.update_object(identifier=Manager.username, obj=Manager.obj)
218                 Manager.username = username
219
220                 # save the data
221                 Manager.obj.save(Manager.username)
222                 Manager.user_file.update_object(identifier=Manager.username, obj=Manager.obj)
223                 Manager.user_file.save()

```

The manager class serves to encapsulate the functions used for saving and loading data. It handles using the tools.File class as well as the User class to load and save data. The load function uses a try except statement to check if a user exists and create an account for them if they don't.

Tools

When developing the save feature, I realised I needed a way to save users to a file, and be able to retrieve that specific user. As I have had to do something similar on multiple projects, I decided to make a general solution that can work for saving to any file. I also decided to put this in a tools file, something I am going to continue developing so I can use it on any project where it may help.

```
1  """
2  This file contains useful tools for any program
3
4  As this file has been designed to work with any program all its functions are generic.
5
6  black, isort and flake8 used for formatting
7  """
8
9  from os.path import isfile
10
11
12 class ObjectNotFound(Exception): 4 usages
13     """Indicates that an object was not found when searched for within a file"""
14
15     def __init__(self, identifier, file):
16         """
17             :param identifier: the identifier of the object that was not found
18             :type identifier: any
19             :param file: the file that was searched
20             :type file: str
21         """
22         super().__init__(f"Object not found | Identifier: {identifier} | File: {file}")
```

I created a custom exception to raise in case an object being searched for is not found. The exception gives the identifier that couldn't be found, and the name of the file being looked in.

```

24
25     class File:
26         """
27             This class manages a list of objects in a file
28
29             The list should be updated using the get_list and update_list functions.
30             It should be saved with the save function.
31             It contains the class objects.
32
33             Individual objects can be got with the get_object function.
34             Objects can be replaced with the update_object function.
35             Objects can be removed with the remove_object function.
36
37             Either the entire list should be modified or only single objects should be modified.
38             These should not be used together.
39             """
40
41     def __init__(self, file, cls):
42         """
43             :param file: the name of the file to store the data in, must be .txt
44             :type file: str
45             :param cls: the class of the data stored in the file, not an object
46             :type cls: class
47                 cls(arg0, arg1, etc.) must call the constructor
48                 arg0 must be a unique identifier.
49                 the attributes self.'s must be the exact same as the parameters
50             :type cls: class
51             """
52
53         self.name = file
54         self.cls = cls
55
56         self.list = []
57         """The list of all data in the file
58         :type list: list[object]"""
59
60         # check file exists, create if it doesn't
61         if not isfile(self.name):
62             f = open(self.name, "w")
63             f.close()
64         self.read()

```

As this is a general-purpose class, I have provided many methods. Some of these may cause issues if used together, as they update the data in different ways and thus could end up overwriting each other, so I have noted in the class's documentation to not do that.

If the file that is to be used for saving doesn't already exist, the class will handle creating it.

```

64 @staticmethod 3 usages
65 def get_identifier(obj):
66     """
67     Returns the first key in the object's dictionary as a string
68
69     The first key is considered the identifier, it is converted to a string to
70     ensure there are no errors during comparison
71
72     :param obj: the object to get the identifier of
73     :type obj: object
74
75     :return: the identifier of the object
76     :rtype: str
77     """
78     keys = list(obj.__dict__.keys())
79     identifier = obj.__dict__[keys[0]]
80     return str(identifier)
81
82 def read(self):
83     """
84     Reads the file and updates self.list
85
86     :rtype: None
87     """
88     f = open(self.name, "r")
89     file_str = f.read()
90     f.close()
91
92     # check file isn't empty
93     if file_str == "":
94         return
95
96     objects = file_str.split("\n")
97     objects.pop() # get rid of newline at end of file
98     for obj in objects:
99         self.list.append(
100             self.cls(**eval(obj)) # convert string to dict and pass as kwargs
101         )
102

```

As the file stores classes where the first attribute is the identifier, the identifier can be obtained by getting the data associated with the first key in the classes dictionary.

To recreate the classes from the stored dictionaries, each line of the file is iterated through. Each line is read as a whole as a string and evaluated, which converts it back into a dictionary. The dictionary is then given to the class to be reconstructed as a set of keyword arguments, where the keys of the dictionary are the keywords. As the dictionary is in order of the class's arguments this works the same as `*(eval(obj).values())`, which would get the values of the keys of the dictionary and pass them as many arguments – instead of keywords and arguments.

I originally found the out about the eval function from [this](#) stack overflow comment, and when I then went looking into how I could use the dictionary I found [this](#).

```

103     def sort(self):
104         """
105             Sorts the list
106
107         :rtype: None
108
109         """
110
111     lst = self.list
112
113     def merge_sort(lst):
114         """
115             Recursive merge sort, uses the identifier of the object
116
117         :rtype: None
118         """
119
120         # split into lists of 1
121         if len(lst) > 1:
122             # split list
123             mid = len(lst) // 2
124             left = lst[:mid]
125             right = lst[mid:]
126
127             # recursively sort list
128             merge_sort(left)
129             merge_sort(right)
130
131             # combine lists
132             # left, right, sorted
133             i, j, k = 0, 0, 0
134             # while unsorted elements remain
135             while i < len(left) and j < len(right):
136                 # if left is lower add left to sorted list
137                 if self.get_identifier(left[i]) < self.get_identifier(right[j]):
138                     lst[k] = left[i]
139                     # increase i as position i has been added to sorted list
140                     i += 1
141                 # else add right to sorted list
142                 else:
143                     lst[k] = right[j]
144                     # increase j as j has been added to sorted list
145                     j += 1
146                     # increase k as position k is now filled
147                     k += 1
148             # cleanup
149             while i < len(left):
150                 lst[k] = left[i]
151                 i += 1
152                 k += 1
153             while j < len(right):
154                 lst[k] = right[j]
155                 j += 1
156                 k += 1
157
158         merge_sort(lst)

```

```
103     def sort(self):
104         """
105             Sorts the list
106
107             :rtype: None
108         """
109         self.list.sort(key=lambda obj: self.get_identifier(obj))
110
```

I had originally used a merge sort to sort the list based on its identifier, but after learning I could use the .sort method with a key whilst developing the leaderboard, I switched it to using that instead.

```
111     def search(self, target):
112         """
113             Finds the position of the target in the list, automatically orders the list
114
115             :param target: the target to search for
116             :type target: object
117             :return: the position of the target, -1 if not found
118             :rtype: int
119         """
120
121     def binary_search(lst, start_pos=0):
122         """
123             Recursive binary search using the identifier
124
125             :param lst: the list section to search
126             :type lst: list
127             :param start_pos: the start position of the list section
128             :type start_pos: int
129             :return: the position of the target, -1 if not found
130             :rtype: int
131         """
132
133         # empty list
134         if len(lst) == 0:
135             return -1
136
137         mid = len(lst) // 2
138         identifier = self.get_identifier(lst[mid])
139
140         if identifier == target:
141             return mid + start_pos
142         elif identifier > target:
143             return binary_search( lst: lst[:mid], start_pos: start_pos)
144         else:
145             return binary_search( lst: lst[mid + 1 :], start_pos: start_pos + mid + 1 )
146
147         self.sort() # order the list
148         target = str(target) # for comparison to prevent errors
149         return binary_search(self.list)
```

I used a binary search for the search function as it is easy to code and efficient. The pos variable tracks what position the start of the current sub-list is in the main self.list, as this means when a position is returned it is the useful and meaningful position of the searched item in the accessible self.list.

```
150 def get_list(self): 3 usages
151     """Returns the list of objects"""
152     return self.list
153
154 def replace_list(self, lst): 1 usage
155     """
156     Replaces the list of objects
157
158     :param lst: the new list
159     :type lst: list
160     :rtype: None
161     """
162     self.list = lst
163
164 def save(self):
165     """Sorts self.list then replaces the file with it."""
166     self.sort()
167     f = open(self.name, "w")
168     for obj in self.list:
169         f.write(str(obj.__dict__) + "\n")
170     f.close()
171
172 def get_object(self, identifier): 1 usage
173     """
174     Gets the object with the given identifier
175
176     :param identifier: a unique identifier
177     :type identifier: any
178     :return: the object or raises exception ObjectNotFound if the object is not found
179     :rtype: object
180     """
181
182     pos = self.search(identifier)
183     if pos == -1:
184         raise ObjectNotFound(identifier=identifier, file=self.name)
185     return self.list[pos]
```

The save function works by ensuring the list is sorted then overwriting the file with it, where it writes the dictionary of the class in the list, one dictionary/class per line.

```
186     def add_object(self, obj):
187         """
188             Adds the object to the list
189
190         :param obj: the object to add
191         :type obj: object
192         :rtype: None
193         """
194         self.list.append(obj)
195         self.sort()
196
197     def update_object(self, identifier, obj): 2 usages
198         """
199             Replaces the object with identifier with the given object.
200
201         :param identifier: the identifier of the object to replace
202         :type identifier: any
203         :param obj: the new object to replace the old one
204         :type obj: object
205         :return: None or raises exception ObjectNotFound if the object is not found
206         :rtype: None
207         """
208
209         pos = self.search(identifier)
210         if pos == -1:
211             raise ObjectNotFound( identifier=identifier, file=self.name)
212             self.list[pos] = obj
213
214     def remove_object(self, identifier):
215         """
216             Removes the object with the given identifier
217
218         :param identifier: the identifier of the object to remove
219         :type identifier: any
220         :return: None or raises exception ObjectNotFound if the object is not found
221         :rtype: None
222         """
223
224         pos = self.search(identifier)
225         if pos == -1:
226             raise ObjectNotFound( identifier=identifier, file=self.name)
227             self.list.pop(pos)
```

```
227
228 # testing
229 ▶ if __name__ == "__main__":
230
231     class Test: 1 usage
232         def __init__(self, arg0=None, arg1=None, arg2=None):
233             self.arg0 = arg0
234             self.arg1 = arg1
235             self.arg2 = arg2
236
237         def output(self):
238             return str(self.arg0) + " " + str(self.arg1) + " " + str(self.arg2)
239
240         file = File( file: "test.txt", cls: Test)
241         file.sort()
242         lst = file.get_list()
243         # lst.append(Test('{a: b}', 3, 5))
244         file.update_list(lst)
245         file.save()
246         print(lst)
247         for cls in file.get_list():
248             print(cls.output())
249             pos = file.search("{")
250             print(pos)
251
```

As I do not have tests for this file, I did some smalls tests at the end of the file, using line 229 to ensure the tests do not execute when the file is imported.

Testing

29	Save – to file	The save algorithm should be able to save all the key data to a text file.	Use the save function with a variety of different key information, including extreme test data such as the timer being at 0.0 seconds and the cube already being solved. In each case attempt loading the program with this saved data, ensuring it is loaded as current game data.	The save file should be updated to include the key data for the user.	Fail
----	----------------	--	--	---	------

```
tools.ObjectNotFound: Object not found | Identifier: dev | File: saves_data.txt
```

The first issue I encountered, before even being able to use my tests, is a user not being found when the Manager.save function is run. This shouldn't be an issue as the Manager.load function should have run before the save function ever could, and it was designed to create a user if they didn't exist. After investigating, I found that the problem was as simple as me never actually adding the created user to the File class.

```
197
198     try:
199         Manager.obj = Manager.user_file.get_object(Manager.username)
200     except tools.ObjectNotFound:
201         Manager.obj = User(Manager.username)
202
203         Manager.obj.load()
```

Fixed by

```
197
198     try:
199         Manager.obj = Manager.user_file.get_object(Manager.username)
200     except tools.ObjectNotFound:
201         Manager.obj = User(Manager.username)
202         Manager.user_file.add_object(Manager.obj)
203
204         Manager.obj.load()
```

Test No.	What is being tested	Description	Method	Expected Output	Pass/Fail
1	3D cube algorithm	<p>There should be an image algorithm that either returns a pygame.Surface containing an image of the cube or display the cube to the screen.</p>	<p>Blit the image returned by the function to the screen or call the function, inside the main game loop.</p> <p>Repeat whilst making changes to used_cube.</p>	An image should be displayed, and it should match used_cube.	Pass
2	Cube turn – vertical, left, up	Executing the turns function with the correct parameters for the given turn should result in the leftmost column being rotated upwards.	Execute the turns function with the parameters for the turn. Run the cube display, the display should show the new cube state. The same rotation should be done to a real Rubik's cube.	The image of both cubes should be exactly the same.	Pass
3	Cube turns – vertical, middle, up	Executing the turns function with the correct parameters for the given turn should result in the middle column being rotated upwards.			Pass
4	Cube turns – vertical, right, up	Executing the turns function with the correct parameters for the given turn should result in the rightmost column being rotated upwards.			Pass

5	Cube turns – vertical, left, down	Executing the turns function with the correct parameters for the given turn should result in the leftmost column being rotated downwards.			Pass
6	Cube turns – vertical, middle, down	Executing the turns function with the correct parameters for the given turn should result in the middle column being rotated downwards.			Pass
7	Cube turns – vertical, right, down	Executing the turns function with the correct parameters for the given turn should result in the rightmost column being rotated downwards.			Pass
8	Cube turns – horizontal, top, right	Executing the turns function with the correct parameters for the given turn should result in the upper row being rotated right.			Pass
9	Cube turns – horizontal, middle, right	Executing the turns function with the correct parameters for the given turn should result in the middle row being rotated right.			Pass

10	Cube turns – horizontal, bottom, right	Executing the turns function with the correct parameters for the given turn should result in the lower row being rotated right.			Pass
11	Cube turns – horizontal, top, left	Executing the turns function with the correct parameters for the given turn should result in the upper row being rotated left.			Pass
12	Cube turns – horizontal, middle, left	Executing the turns function with the correct parameters for the given turn should result in the middle row being rotated left.			Pass
13	Cube turns – horizontal, bottom, left	Executing the turns function with the correct parameters for the given turn should result in the lower row being rotated left.			Pass
14	Scramble	The scramble function should randomly position the individual squares whilst still ensuring that the cube is solvable.	Add a delay between each move in the scramble function. Run the scramble function and follow along with a real Rubik's cube.	Each move done by the scrambler should be possible on the real Rubik's cube.	Pass

15	Solver - solving	The solve function should solve the cube, showing the user each step, ensuring that each move is possible and not simply changing the used_cube to fit as needed.	Manually scramble the cube, doing each move to a real Rubik's cube as well. Run the solver and follow the moves on the Real Rubik's cube.	Each move done by the solver should be possible on the real Rubik's cube and at the end the cube should be solved.	Fail
16	Solver – stop solving	If used_cube reaches a solved state, the solver should stop solving, regardless of if there something such as a moves list indicates there are more moves to do to solve the cube.	Manually scramble the cube, ensuring that you return to a solved state at least once then scramble from there. Run the solver.	The solver should stop when it reaches the first solved state.	Pass
17	Check solved	There should be a function to check if a cube state is solved or not.	Test the function using multiple different cube states, some of which are manually or automatically scrambled.	The outputs should match the given cube state.	Pass
18	Hints	The hint function should complete one move towards the solve. It must only be one move, and it must help solve the cube.	Scramble the cube then run the runt the hint function. Note the move that it makes. Undo that move and then run the solver (test 15 must have passed).	Only one move should be completed by the hint function. The move should match the one done by the solver.	Fail

19	Timer – time elapsed	The timer should correctly record the amount of timer that has passed since it started	Start the timer. Wait for 10 seconds (counted via a trusted, real, timer). Print the time elapsed. Repeat a few times with various amounts of time waited.	The trusted timer and the timer being tested should have a matching (or very similar to account for human error) times.	Pass
20	Timer – auto start	The timer should automatically start upon scramble.	Scramble the cube. Solve the cube. Scramble the cube, use hint function.	Each time the cube is scrambled the timer should start,	Pass
21	Timer – auto stop	The timer should automatically stop upon being solved.	Scramble the cube. Use the solve function. Scramble the cube. Monitor the time elapsed during this.	Upon being solved and when the solver is used, the timer should stop. The timer should not stop if these do not occur.	Fail
22	Leaderboard – Eligibility check	Each entry should be checked to see if they are faster than the slowest time on the leaderboard, to see if they have made it onto the leaderboard.	Submit a completion with a slower completion time than the slowest. Submit a completion with a faster time than the slowest. Submit a completion time identical to the slowest.	Only the completion with the faster time should be considered for updating the leaderboard.	Pass

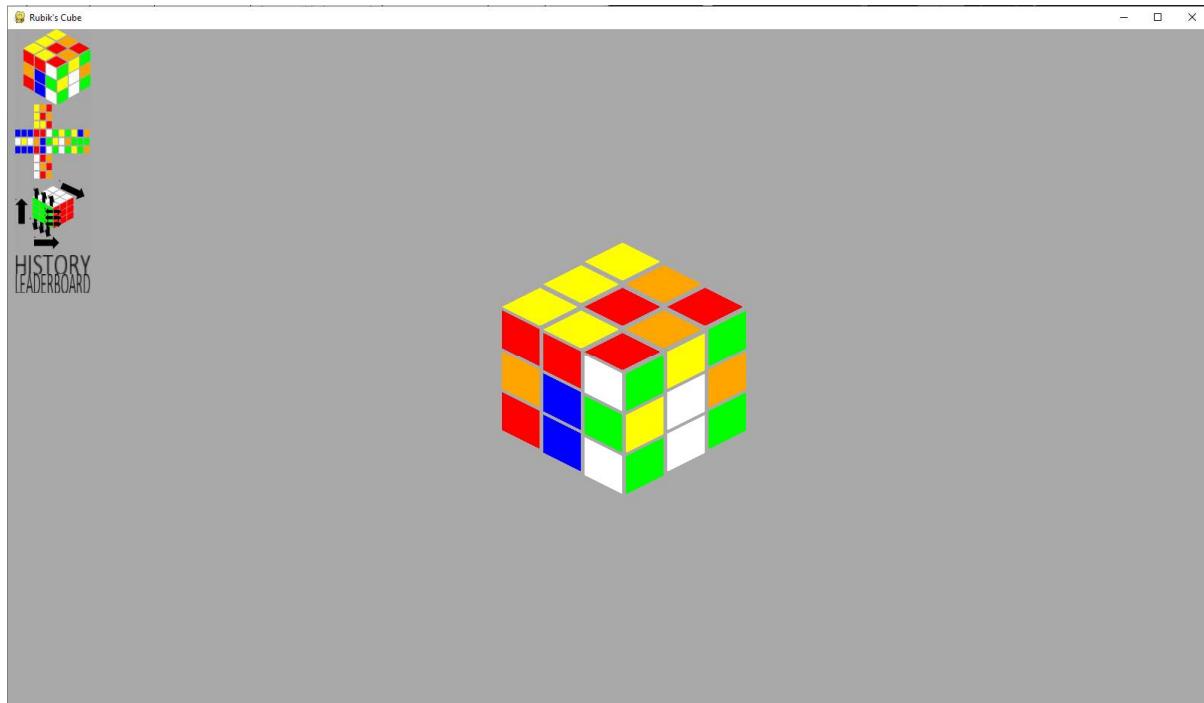
23	Leaderboard – add entry	If the leaderboard isn't full any completion should be added to the leaderboard. If the leaderboard is full and entry is eligible, the new entry should replace the slowest time on the leaderboard.	Add an entry when the leaderboard is empty. Add an entry when the leaderboard is half full. Add an entry when the leaderboard is full.	The first two entries should be automatically added to the leaderboard. The last entry should replace the slowest entry on the leaderboard.	Pass
24	Leaderboard – sort leaderboard	When a new completion is added to the leaderboard, the leaderboard needs to be sorted to ensure that completion ends up in the correct position. The list should be ordered by ascending times.	Sort the leaderboard when it is already in order. Sort the leaderboard when it is in descending order. Sort the leaderboard when it is randomised. Sort the leaderboard when 2 identical times exist.	Each leaderboard should end up sorted. Manually check this.	Pass
25	Leaderboard – save times	The ordered list of leaderboard times should be able to be saved to a text file so that they are kept even when the program ends.	Save the leaderboard when it is empty. Save the leaderboard when it has no completions. Save the leaderboard	In each case the text file should be updated with the leaderboard.	Pass

26	Leaderboard – load saved times	The saved leaderboard times need to be able to be loaded so they can be displayed and be checked against for any new records.	when it is half full. Save the leaderboard when it is full. In each case close the program and start it again, attempting to load these saves.	In each case the leaderboard should be updated to match the text file.	Pass
27	Guide algorithm	The should either be a function that returns a pygame.Surface or a procedure that draws the image to the screen. The image should show how to use the cube.	Either blit the pygame.Surface to the screen or call the procedure inside a game loop.	An image should be displayed.	Pass
28	Guide algorithm – prevent moves	The cube image should only display the default cube and as such it should not allow cube interactions to happen when the guide is being displayed.	When displaying the cube, try the following: turns, rotations, scrambling and solving.	None of the functions should work. The cube should remain unchanged.	Pass

30	Save – load form file	The save algorithm should be manage loading the data from the text file and updating game values so it is as if the saved state has been achieved in the current session.	Use the save function with a variety of different key information, including extreme test data such as the timer being at 0.0 seconds and the cube already being solved. In each case attempt loading the program with this saved data, ensuring it is loaded as current game data.	The save file should be updated to include the key data for the user.	Fail
31	Save – autosave	The save function should automatically run periodically.	Start a game and make some changes to the cube. Then wait the amount of time set between saves. Once this time has passed closer and reopen the program.	The cube should be in the same state as it was when the program was closed,	Pass
32	Game history	When a solve is complete, either by the solve function being used, the scrambler being used, or the cube being solved, the data about that solve should be saved to a list of solves.	Finish a solve using the solver, scrambler, and by solving manually.	These three solves should be added to game history list.	Pass

Improvements

Tests 15 and 18 – Both the solver and hint feature work if the scramble function is used once, and then the cube solved. However, if the scramble feature is used more than once the solver and hint feature seems to have a limit for how much of the cube they can solve.



Both these features share the function Solver.pop_move, so I investigated that function. I could find no errors with it, so I knew it had something to do with game_data.moves stack. As the problem occurred when the scramble function was called, I checked when the function was called in the game loop.

```
246 elif event.Key == pygame.K_m: # scramble
247     if timer.running: # ensures the attempt was started
248         # failed attempts should be recorded
249         user_data.game_history.add_game()
250
251     # reset key data
252     game_data.moves.clear()
253     game_data.move_count = 0
254     game_data.scrambler_count = 0
255     game_data.hints_used = False
256     game_data.solver_used = False
257     game_data.solved = False
258     game_data.time_taken = 0
259     game_data.start_time = time.time()
260
261     features.scramble()
262     # prevent the timer from being started whilst the solver runs
263     # was achieved by scrambling whilst the timer ran
264     solve_cube = False
265     timer.start() # start timer
```

The problem was then immediately obvious, the moves list was being cleared each time the scramble function was called, and the move_count set to 0, but the moves themselves where still done and needed to be know so the solver can undo them.

I chose to fix this by making the scramble function reset the cube to default before scrambling it, which ensures this error won't occur again.

Original:

```
10 import time
11 from random import randint
12
13 import game_data as gd
14 import interface
15 import numpy
16 import pygame
17 import tools
18 import user_data as ud
19 from cube import rotate, turn
20 from game_data import BLACK, default_colour, default_cube, default_font
21 from validation import ValidateScreenPositions
22
23
24 def scramble():
25     """
26     Randomly scrambles the cube by making between 15 and 25 moves randomly
27
28     :rtype: None
29     """
30
31     count = randint(a=15, b=25)
32     gd.scrambler_count = count
33     for _ in range(count):
34         # randomise every aspect of the turn
35         direction = bool(randint(a=0, b=1))
36         number = randint(a=0, b=2)
37         backwards = bool(randint(a=0, b=1))
38
39         turn(row_col=direction, number=number, backwards=backwards)
```

Fixed:

```

10 import copy
11 import time
12 from random import randint
13
14 import game_data as gd
15 import interface
16 import numpy
17 import pygame
18 import tools
19 import user_data as ud
20 from cube import rotate, turn
21 from game_data import BLACK, default_colour, default_cube, default_font
22 from validation import ValidateScreenPositions
23

```

```

27 def scramble():
28     """
29     Randomly scrambles the cube by making between 15 and 25 moves randomly
30
31     :rtype: None
32     """
33
34     # reset the cube
35     gd.used_cube = copy.deepcopy(default_cube)
36
37     count = randint(a=15, b=25)
38     gd.scrambler_count = count
39     for _ in range(count):
40         # randomise every aspect of the turn
41         direction = bool(randint(a=0, b=1))
42         number = randint(a=0, b=2)
43         backwards = bool(randint(a=0, b=1))
44
45         turn(row_col=direction, number=number, backwards=backwards)

```

Tests 21 and 30 – The issue was that the timer function sometimes stopped automatically, and sometimes it continued when the cube was solved. I quickly determined that the Solver.check_solved() function was the problem, as it returned False even when the cube was solved. However, when checking the function, everything appeared to be in order.

However, in the saves_data.txt file, the cube state had an interesting anomaly. Some of the colours were being saved as tuples, whilst some were saved as lists. The save files was also being broken and has ‘array’ for it for some reason. This broken file format had been preventing me from loading saves, however I had simply been deleting the save file (so a new one was created, which worked) before starting the program each time, as I was focusing on the this (and previous) errors, waiting to fix the loading error until I reached the save test.

```

1  {'username': 'dev', 'cube_state': [array([[255, 165, 0],
2      [255, 165, 0],
3      [255, 165, 0]], 
4
5      [[255, 165, 0],
6      [255, 165, 0],
7      [255, 165, 0]], 
8
9      [[255, 165, 0],
10     [255, 165, 0],
11     [255, 165, 0]]]), [[(0, 0, 255), (0, 255, 0), (0, 255, 0)]]

```

Unfortunately, I still had little idea what could be causing this, so I decided to test each possible turn one by one. I found only edge turns resulted in this, but middle turns did not. This did help, as I knew the main coding difference between middle turns and edge turns was that edge turns involved using `numpy.rot90` on the face the edge touched. After checking, I found that `numpy.rot90` did return an array like I expected, but instead a ndarray. I fixed this by using `.tolist()`, a method of ndarrays which returns them as a normal list. I made sure to include this for the z axis rotation as well.

Original:

```

567         if number == 0: # rotate the top face
568             gd.used_cube[4] = numpy.rot90(m: gd.used_cube[4], k=1, axes=(0, 1))
569         elif number == 2: # rotate the bottom face
570             gd.used_cube[5] = numpy.rot90(m: gd.used_cube[5], k=1, axes=(1, 0))
571         else: # turn the column
572             for i in range(3):
573                 gd.used_cube[1][i][n] = face5[i][n]
574                 # 2-i flips the row number for the back
575                 # 2 - n flips the column number for the back
576                 gd.used_cube[5][2 - i][n] = face3[i][2 - n]
577                 gd.used_cube[3][2 - i][2 - n] = face4[i][n]
578                 gd.used_cube[4][i][n] = face1[i][n]
579
580             if number == 0: # rotate left face
581                 gd.used_cube[0] = numpy.rot90(m: gd.used_cube[0], k=1, axes=(0, 1))
582             elif number == 2: # rotate right face
583                 gd.used_cube[2] = numpy.rot90(m: gd.used_cube[2], k=1, axes=(1, 0))
584
620         elif axis == "z": # equivalent to a rotation along the z axis
621             # rotate the front and back faces
622             gd.used_cube[1] = numpy.rot90(m: gd.used_cube[1], k=1, axes=(1, 0))
623             gd.used_cube[3] = numpy.rot90(m: gd.used_cube[3], k=1, axes=(0, 1))

```

Fixed:

```

567     if number == 0: # rotate the top face
568         gd.used_cube[4] = numpy.rot90(
569             m: gd.used_cube[4], k=1, axes=(0, 1)
570         ).tolist()
571     elif number == 2: # rotate the bottom face
572         gd.used_cube[5] = numpy.rot90(
573             m: gd.used_cube[5], k=1, axes=(1, 0)
574         ).tolist()
575     else: # turn the column
576         for i in range(3):
577             gd.used_cube[1][i][n] = face5[i][n]
578             # 2-i flips the row number for the back
579             # 2 - n flips the column number for the back
580             gd.used_cube[5][2 - i][n] = face3[i][2 - n]
581             gd.used_cube[3][2 - i][2 - n] = face4[i][n]
582             gd.used_cube[4][i][n] = face1[i][n]
583
584     if number == 0: # rotate left face
585         gd.used_cube[0] = numpy.rot90(
586             m: gd.used_cube[0], k=1, axes=(0, 1)
587         ).tolist()
588     elif number == 2: # rotate right face
589         gd.used_cube[2] = numpy.rot90(
590             m: gd.used_cube[2], k=1, axes=(1, 0)
591         ).tolist()
592
593
594     elif axis == "z": # equivalent to a rotation along the z axis
595         # rotate the front and back faces
596         gd.used_cube[1] = numpy.rot90( m: gd.used_cube[1], k=1, axes=(1, 0)).tolist()
597         gd.used_cube[3] = numpy.rot90( m: gd.used_cube[3], k=1, axes=(0, 1)).tolist()

```

This fixed my saving issue, but still prevented check_sorted from working, as some tuples were still becoming lists. I could have created a small function to check to convert the RGB lists back into tuples, but this function would have had to be called every time check_solved was run, and any other function that relied on the colours being a consistent data type, so I instead decided to get rid of the tuples completely and just use lists. To do this I just converted the colours in game_data from tuples to lists.

Original

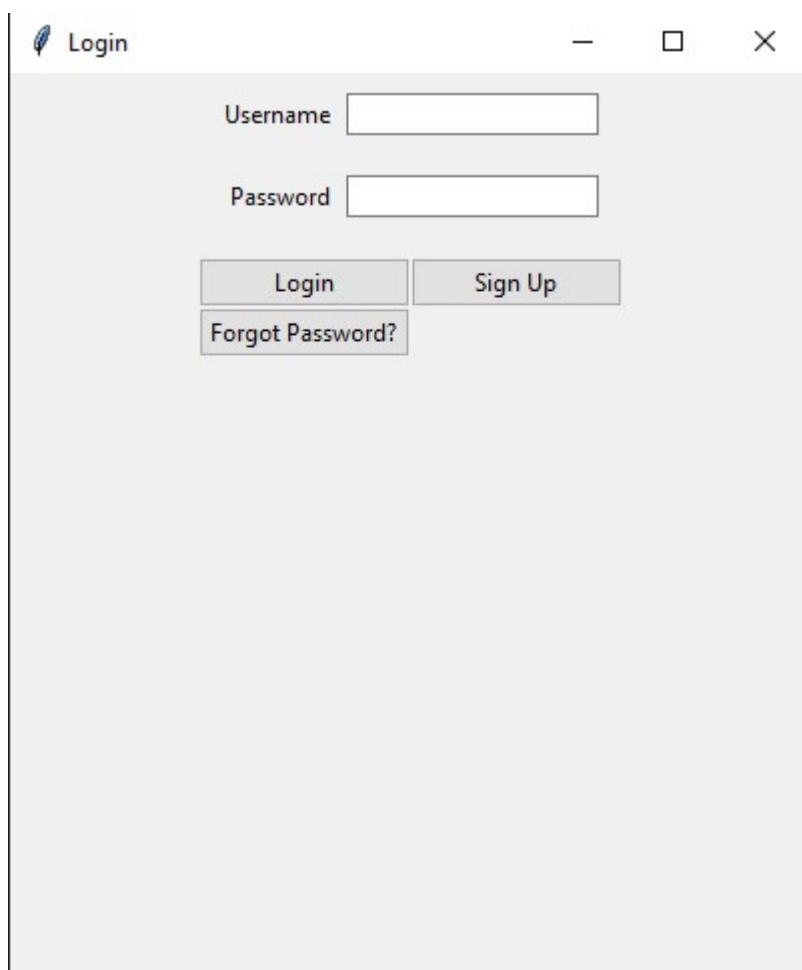
```
17 # colours  
18 BLACK = (0, 0, 0)  
19 WHITE = (255, 255, 255)  
20 YELLOW = (255, 255, 0)  
21 ORANGE = (255, 165, 0)  
22 RED = (255, 0, 0)  
23 GREEN = (0, 255, 0)  
24 BLUE = (0, 0, 255)  
25 GREY = (169, 169, 169)  
26
```

Fixed

```
17 # colours  
18 BLACK = [0, 0, 0]  
19 WHITE = [255, 255, 255]  
20 YELLOW = [255, 255, 0]  
21 ORANGE = [255, 165, 0]  
22 RED = [255, 0, 0]  
23 GREEN = [0, 255, 0]  
24 BLUE = [0, 0, 255]  
25 GREY = [169, 169, 169]  
26
```

This finally fixed the problem.

Program Images



>Login

— □ ×

Incorrect username or password

Username

Password

[Login](#)

[Sign Up](#)

[Forgot Password?](#)

 Login

— □ ×

No username given.

Username

Password

Re-enter password

Security Question

Answer

Re-enter answer

 Login

— □ ×

No password given.

Username

Password

Re-enter password

Security Question

Answer

Re-enter answer

>Login

— □ ×

Passwords do not match.

Username

Password

Re-enter password

Security Question

Answer

Re-enter answer

 Login

No security question given.

Username

Password

Re-enter password

Security Question

Answer

Re-enter answer



Login

— □ ×

No answer given.

Username

Password

Re-enter password

Security Question

Answer

Re-enter answer



Login



Answers do not match.

Username

Password

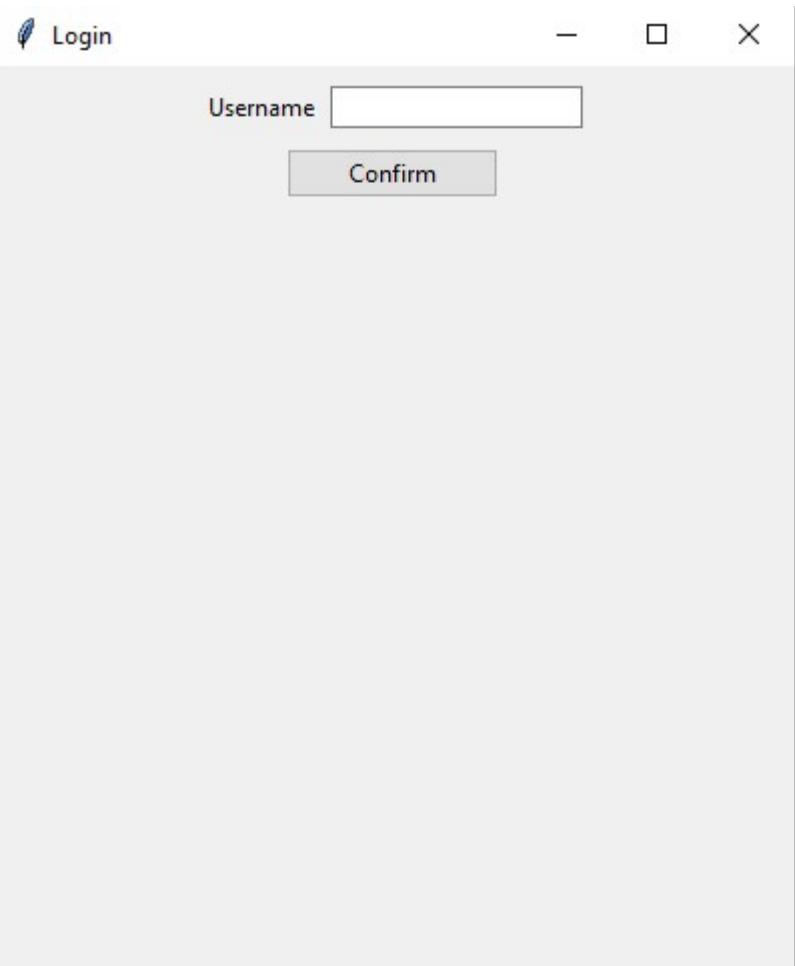
Re-enter password

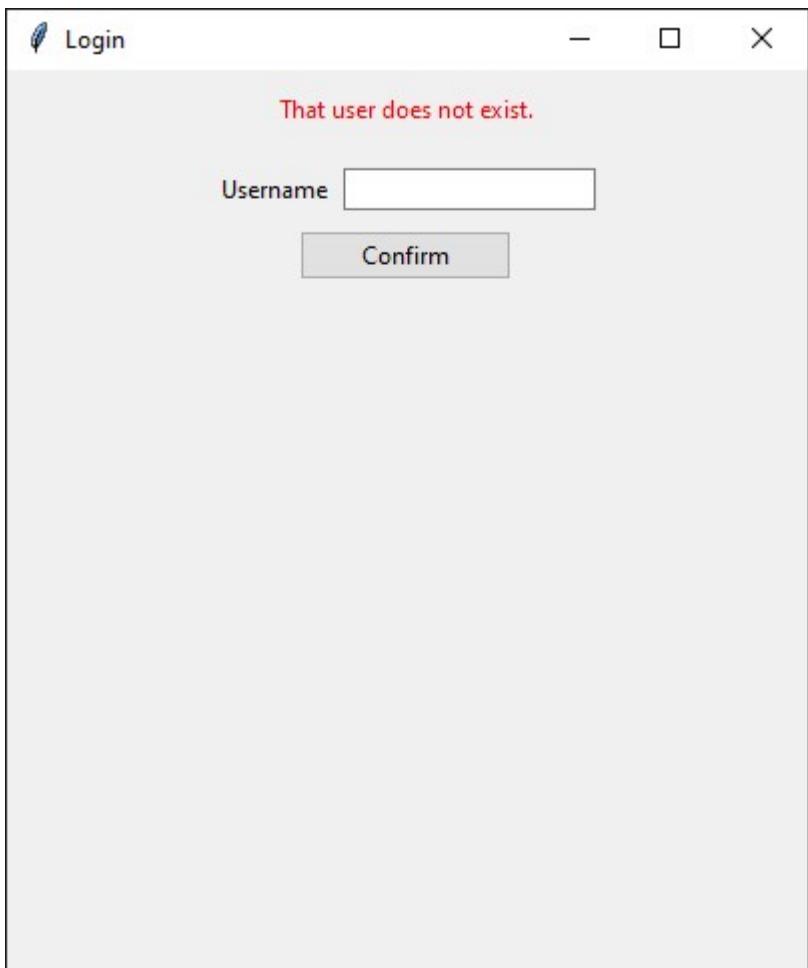
Security Question

Answer

Re-enter answer

Sign Up







Login

-

□

X

test

Answer

Enter

>Login

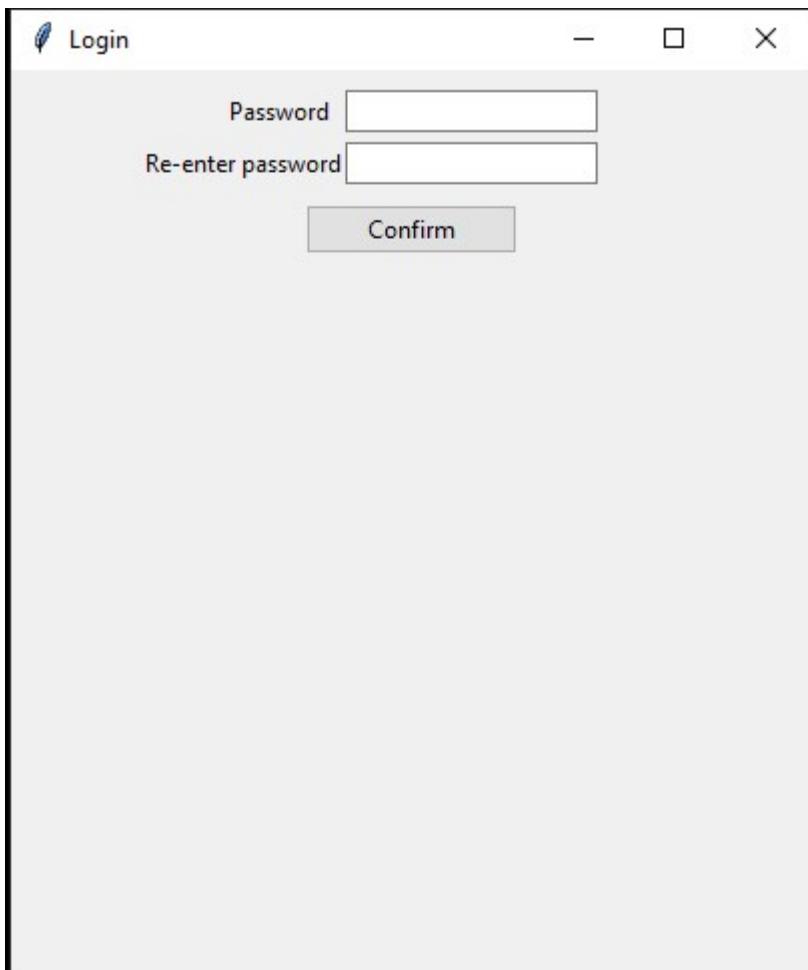
- □ ×

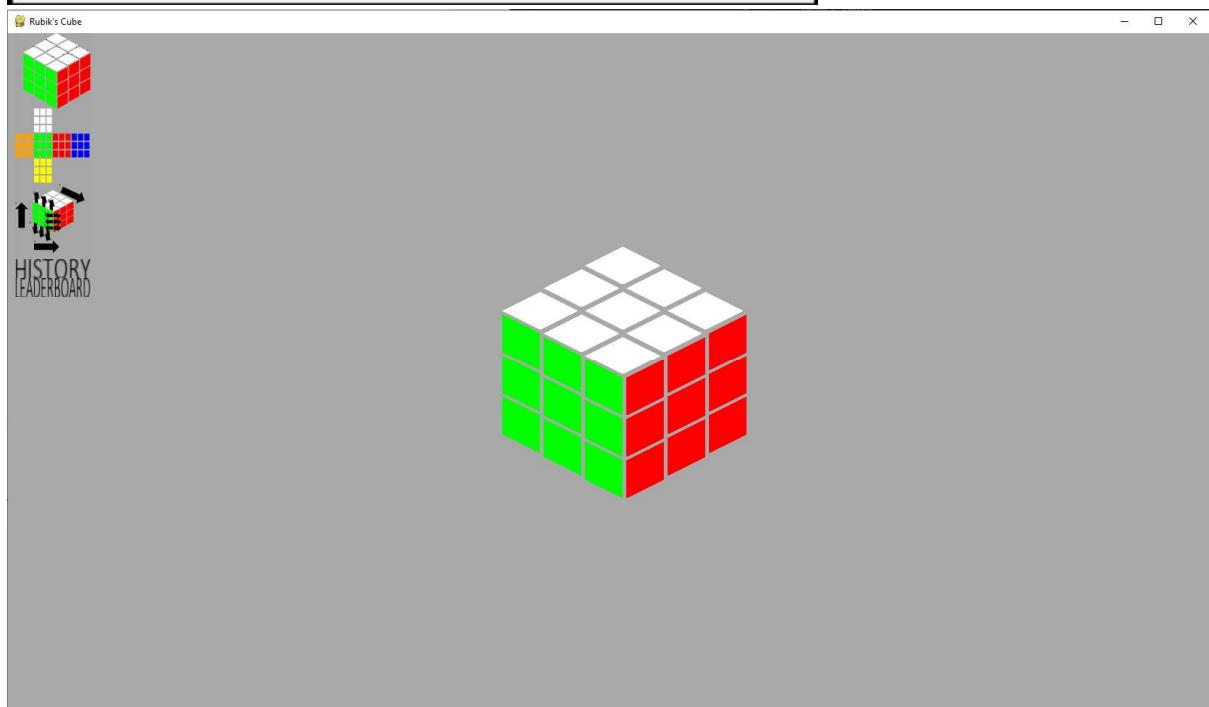
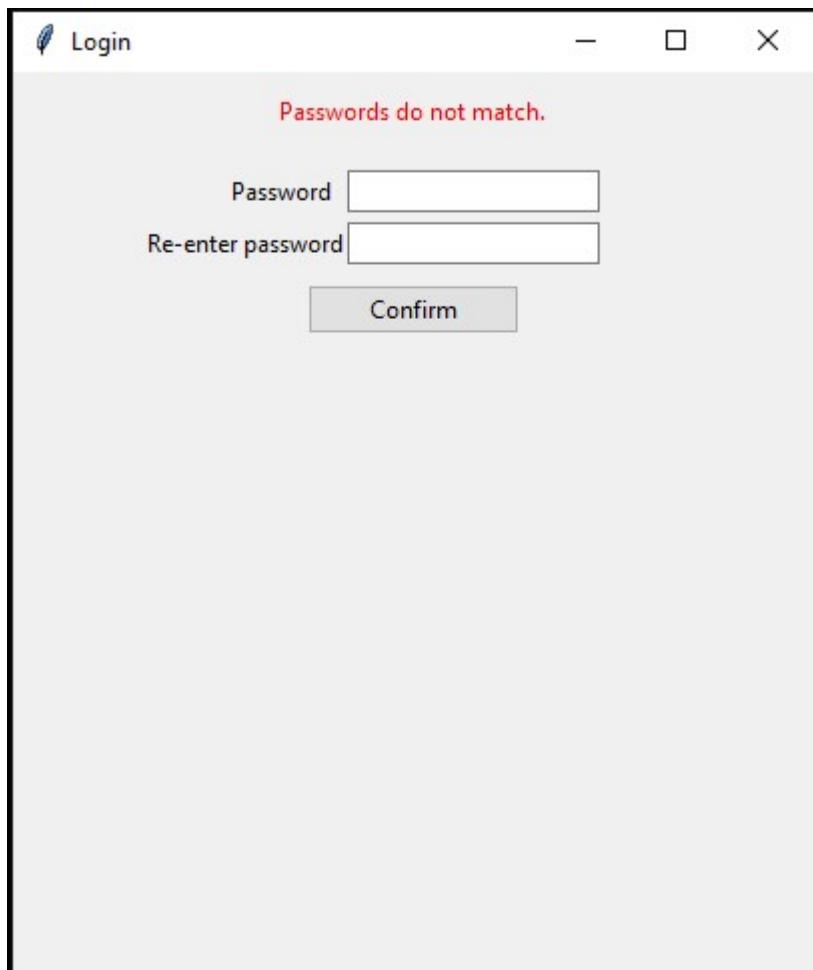
Incorrect

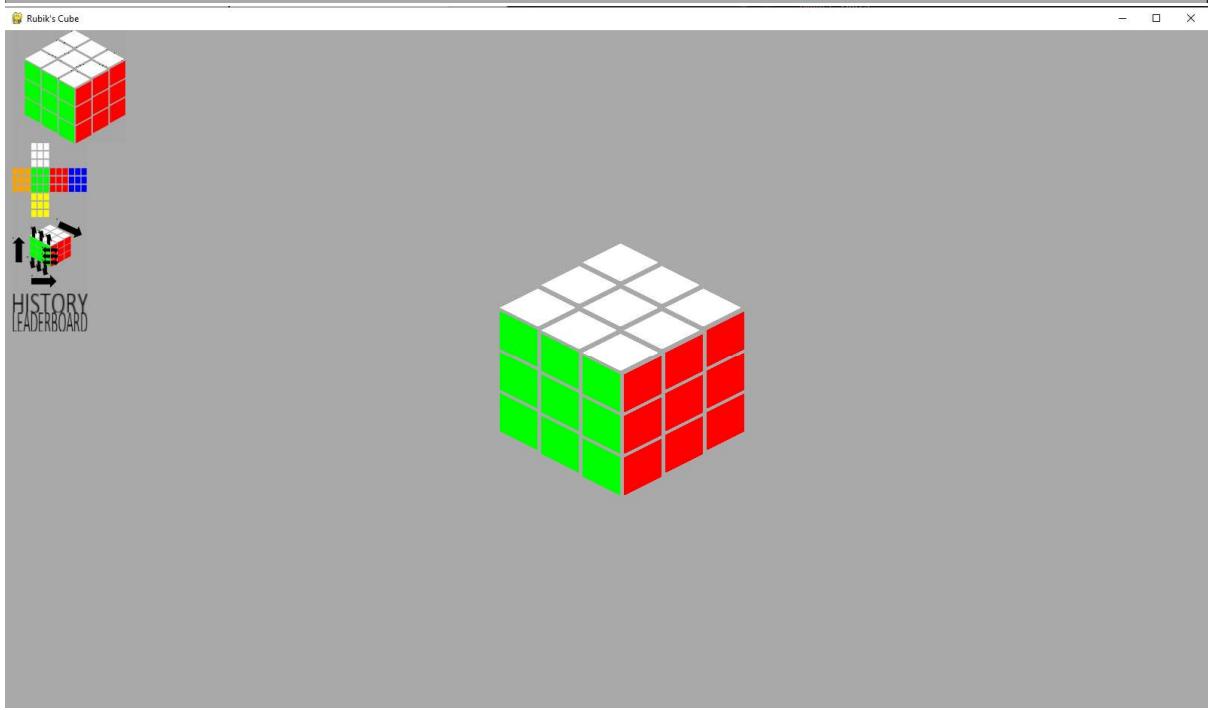
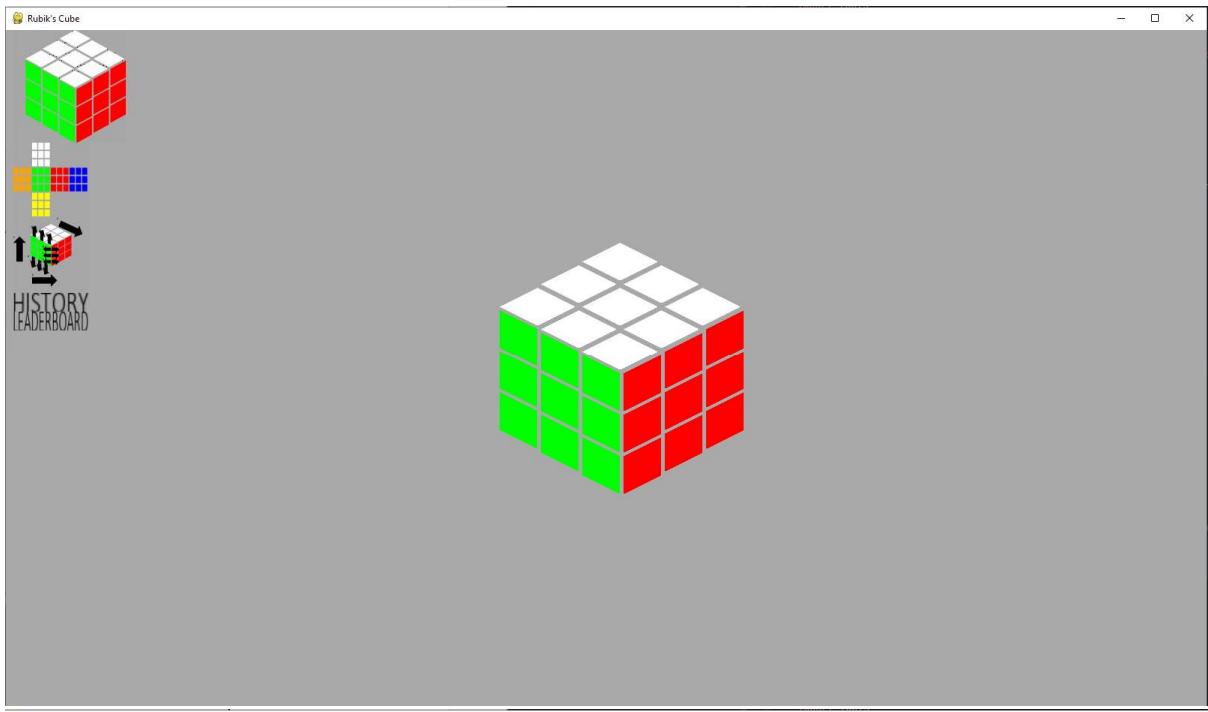
test

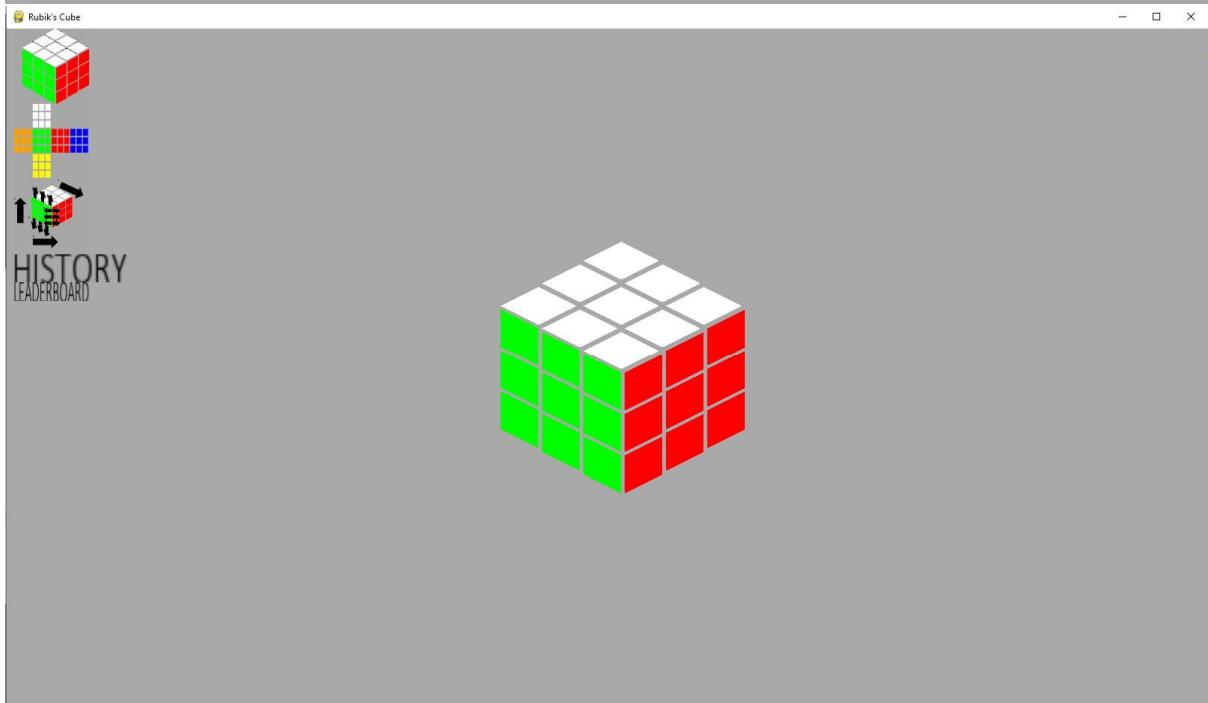
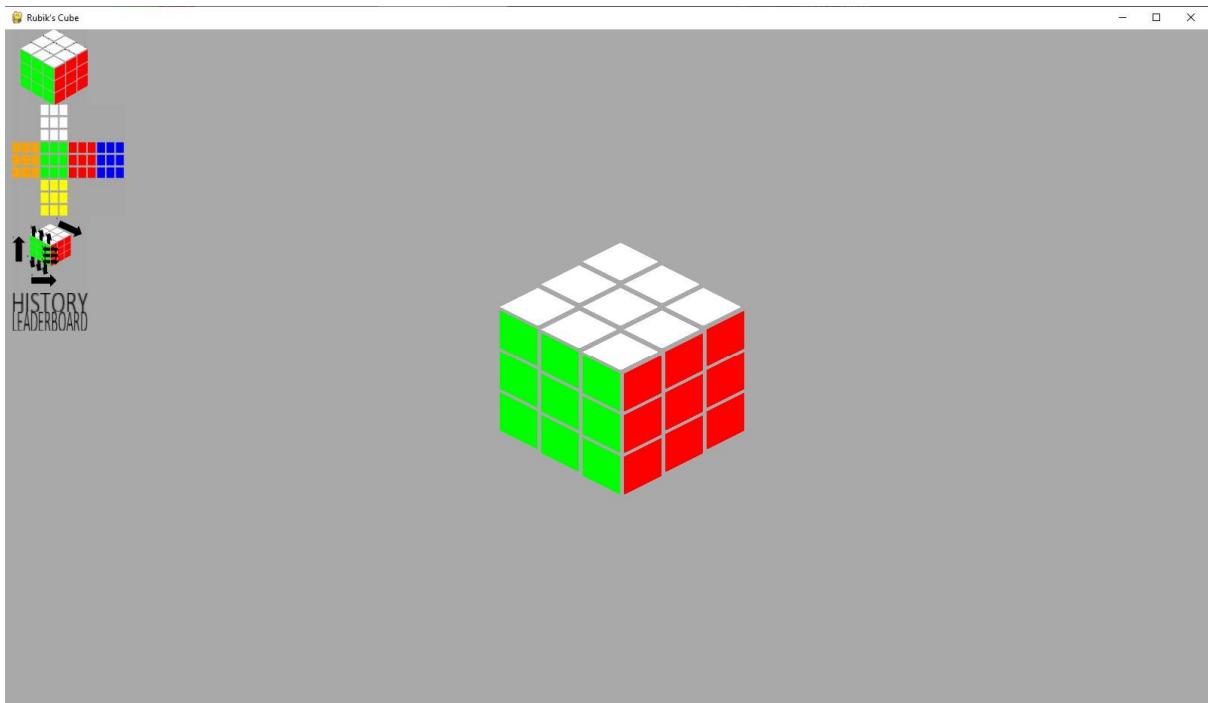
Answer

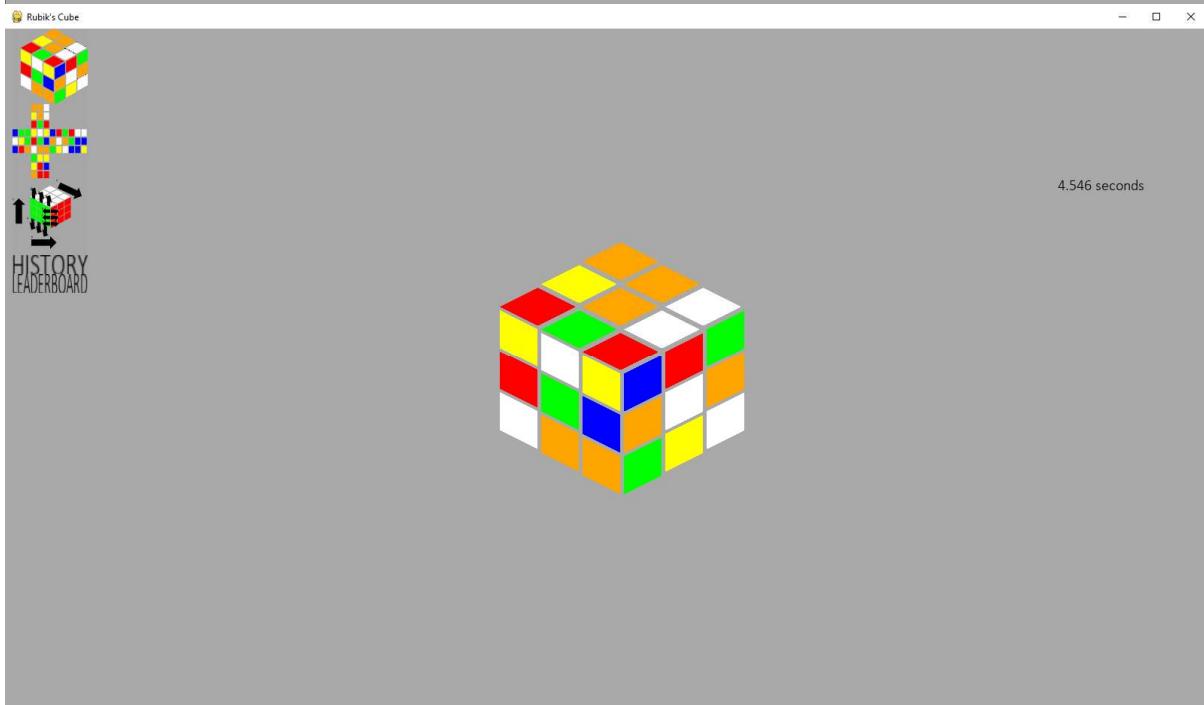
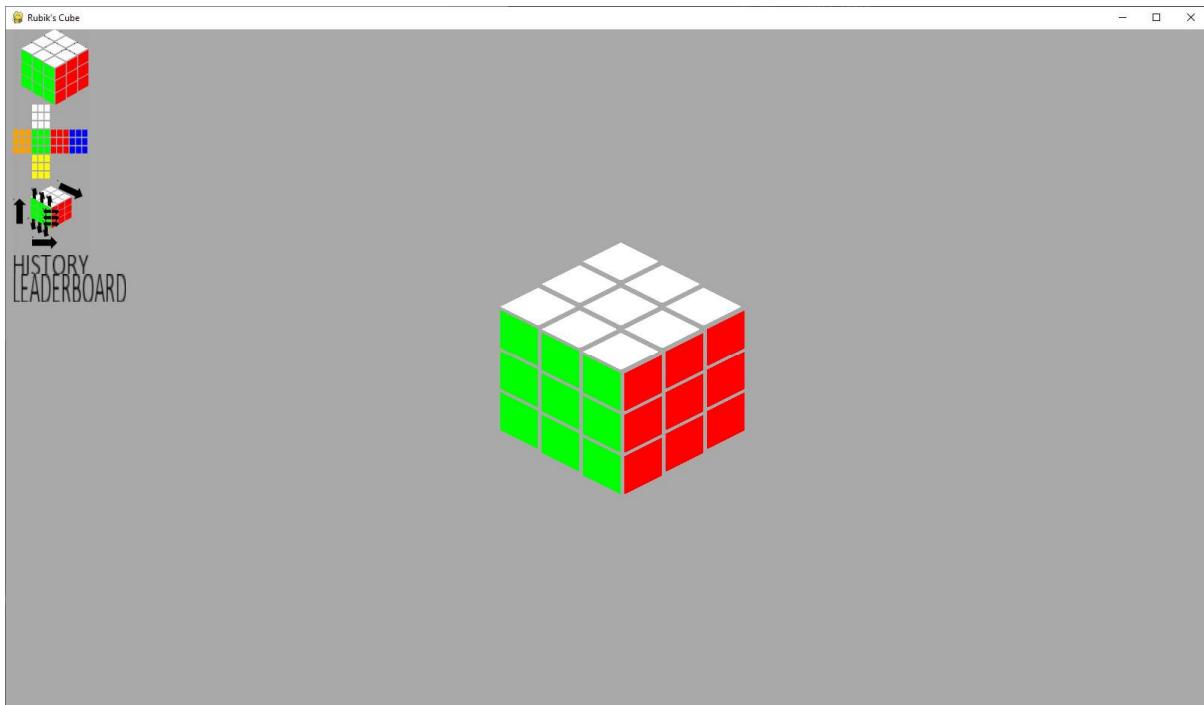
Enter

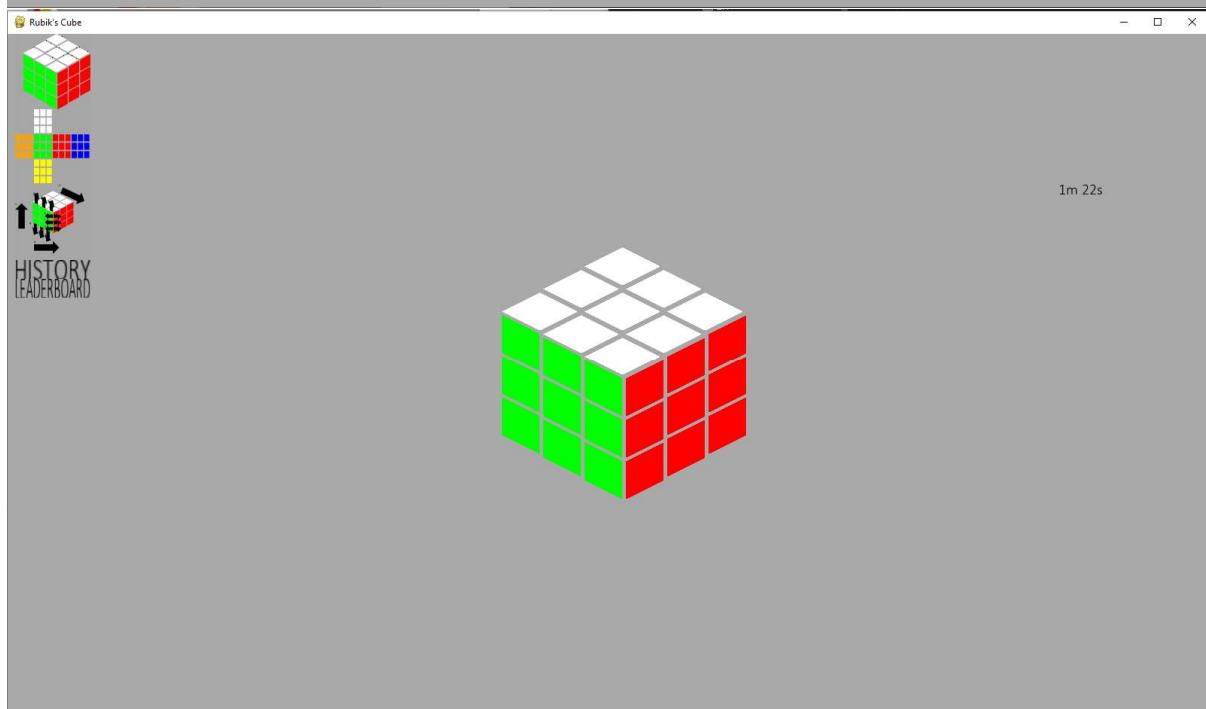
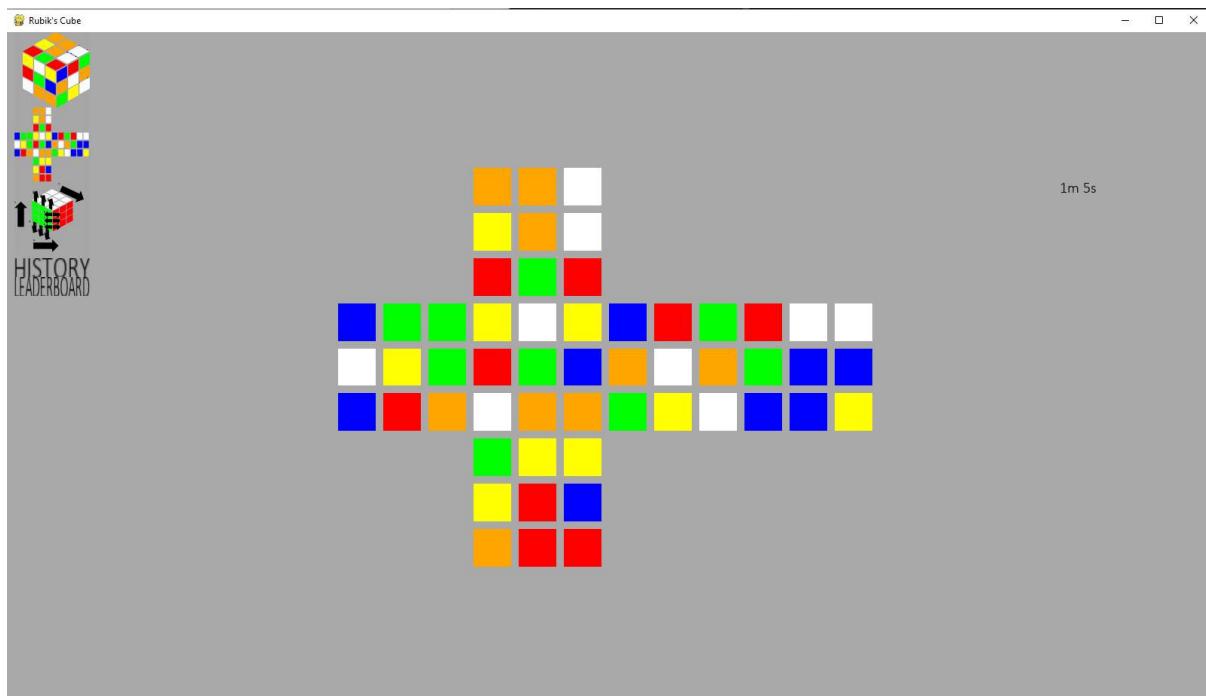


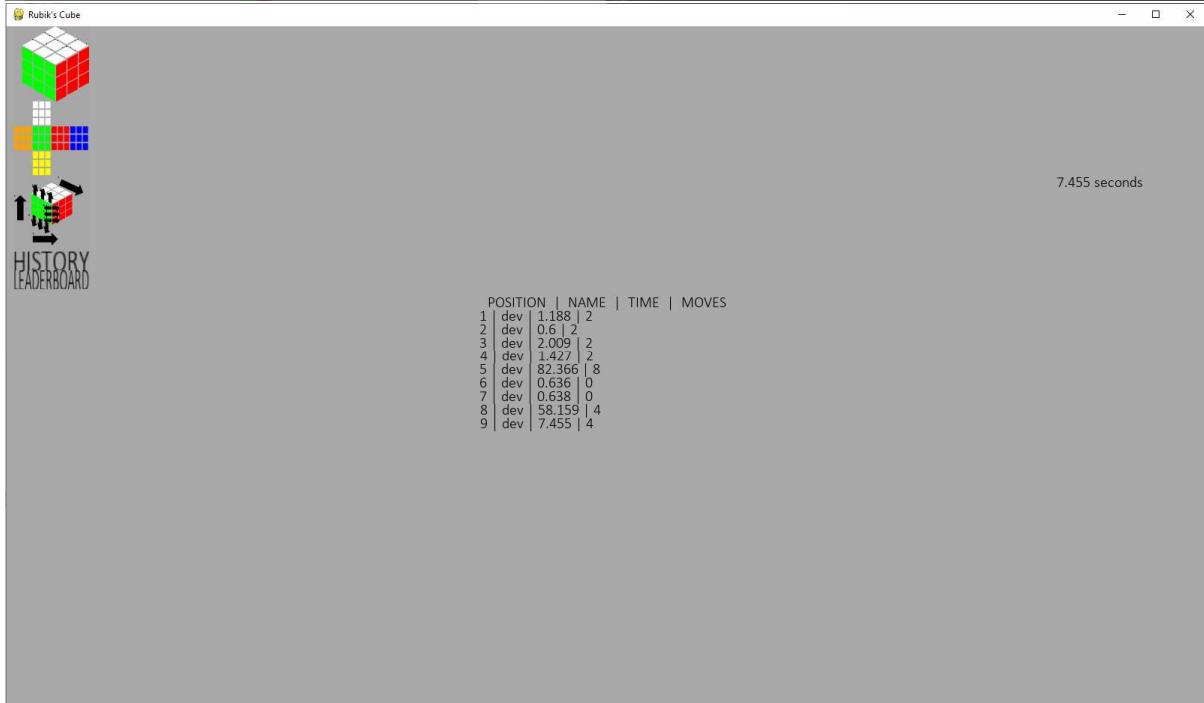












Evaluation

Post-development testing

Stakeholder Responses

Stakeholder name:	Sam Flegg		
Test no.	Question	Answer type	Stakeholder Answer
33	Is there a representation of a 3D cube?	Yes or No	Yes
34	How professional does the 3D cube look?	1 to 10	8
35	To your knowledge, is every move that's possible on a real Rubik's cube possible on the program's Rubik's cube?	Yes or No	Yes
36	To your knowledge, is every move possible on the program possible on a real Rubik's cube?		Yes
37	How well do the cube controls meet the description: simple and intuitive?	1 to 10	10
38	How well do the program controls meet the description: simple and intuitive?		10
39	Is there a scramble function?	Yes or No	Yes
40	Is there a solve function?		Yes
41	How well does the solve function show each move done (in regards to you being able to understand it)?	1 to 10	6
42	Is there a hint function that shows you the next move to make?	Yes or No	Yes
43	Is there a timer to track how long solves take?		Yes

44	Does the timer start automatically?		Yes
45	Does the timer stop automatically?		Yes
46	Is there a leaderboard?		Yes
47	Does the leaderboard show the ten quickest solves in ascending order?		Yes
48	Does each leaderboard entry display: the username, the time taken, the moves taken?		Yes
49	Is there a login system?		Yes
50	How straightforward to use is the login system?	1 to 10	9
51	Is your user and game data loaded when you log in?	Yes or No	Yes
52	Does the save function run automatically?		Yes
53	Is there a guide to use the program?		Yes
54	How clear and concise is the guide?	1 to 10	10
55	Is there a function to see your game history?	Yes or No	Yes
56	How easy does the game history function make it to see how you have progressed?	1 to 10	8

Stakeholder name:	Eddie Coulson		
Test no.	Question	Answer type	Stakeholder Answer
33	Is there a representation of a 3D cube?	Yes or No	Yes
34	How professional does the 3D cube look?	1 to 10	10
35	To your knowledge, is every move that's	Yes or No	Yes

	possible on a real Rubik's cube possible on the program's Rubik's cube?		
35	To your knowledge, is every move possible on the program possible on a real Rubik's cube?		Yes
37	How well do the cube controls meet the description: simple and intuitive?	1 to 10	10
38	How well do the program controls meet the description: simple and intuitive?		8
39	Is there a scramble function?	Yes or No	Yes
40	Is there a solve function?		Yes
41	How well does the solve function show each move done (in regards to you being able to understand it)?	1 to 10	5
42	Is there a hint function that shows you the next move to make?	Yes or No	Yes
43	Is there a timer to track how long solves take?		Yes
44	Does the timer start automatically?		Yes
45	Does the timer stop automatically?		Yes
46	Is there a leaderboard?		Yes
47	Does the leaderboard show the ten quickest solves in ascending order?		Yes
48	Does each leaderboard entry display: the username, the time taken, the moves taken?		Yes

49	Is there a login system?		Yes
50	How straightforward to use is the login system?	1 to 10	10
51	Is your user and game data loaded when you log in?	Yes or No	Yes
52	Does the save function run automatically?		Yes
53	Is there a guide to use the program?		Yes
54	How clear and concise is the guide?	1 to 10	7
55	Is there a function to see your game history?	Yes or No	Yes
56	How easy does the game history function make it to see how you have progressed?	1 to 10	9

Stakeholder name:	Connor Gilroy		
Test no.	Question	Answer type	Stakeholder Answer
33	Is there a representation of a 3D cube?	Yes or No	Yes
34	How professional does the 3D cube look?	1 to 10	7
35	To your knowledge, is every move that's possible on a real Rubik's cube possible on the program's Rubik's cube?	Yes or No	Yes
35	To your knowledge, is every move possible on the program possible on a real Rubik's cube?		Yes
37	How well do the cube controls meet the description: simple and intuitive?	1 to 10	6

38	How well do the program controls meet the description: simple and intuitive?		6
39	Is there a scramble function?	Yes or No	Yes
40	Is there a solve function?		Yes
41	How well does the solve function show each move done (in regards to you being able to understand it)?	1 to 10	5
42	Is there a hint function that shows you the next move to make?	Yes or No	Yes
43	Is there a timer to track how long solves take?		Yes
44	Does the timer start automatically?		Yes
45	Does the timer stop automatically?		Yes
46	Is there a leaderboard?		Yes
47	Does the leaderboard show the ten quickest solves in ascending order?		Yes
48	Does each leaderboard entry display: the username, the time taken, the moves taken?		Yes
49	Is there a login system?		Yes
50	How straightforward to use is the login system?	1 to 10	7
51	Is your user and game data loaded when you log in?	Yes or No	Yes
52	Does the save function run automatically?		Yes
53	Is there a guide to use the program?		Yes

54	How clear and concise is the guide?	1 to 10	7
55	Is there a function to see your game history?	Yes or No	Yes
56	How easy does the game history function make it to see how you have progressed?	1 to 10	4

Stakeholder name:	James Brearly		
Test no.	Question	Answer type	Stakeholder Answer
33	Is there a representation of a 3D cube?	Yes or No	Yes
34	How professional does the 3D cube look?	1 to 10	9
35	To your knowledge, is every move that's possible on a real Rubik's cube possible on the program's Rubik's cube?	Yes or No	Yes
35	To your knowledge, is every move possible on the program possible on a real Rubik's cube?		Yes
37	How well do the cube controls meet the description: simple and intuitive?	1 to 10	6
38	How well do the program controls meet the description: simple and intuitive?		8
39	Is there a scramble function?	Yes or No	Yes
40	Is there a solve function?		Yes
41	How well does the solve function show each move done (in regards to you being able to understand it)?	1 to 10	4

42	Is there a hint function that shows you the next move to make?	Yes or No	Yes
43	Is there a timer to track how long solves take?		Yes
44	Does the timer start automatically?		Yes
45	Does the timer stop automatically?		Yes
46	Is there a leaderboard?		Yes
47	Does the leaderboard show the ten quickest solves in ascending order?		Yes
48	Does each leaderboard entry display: the username, the time taken, the moves taken?		Yes
49	Is there a login system?		Yes
50	How straightforward to use is the login system?		8
51	Is your user and game data loaded when you log in?	Yes or No	Yes
52	Does the save function run automatically?		Yes
53	Is there a guide to use the program?		Yes
54	How clear and concise is the guide?	1 to 10	5
55	Is there a function to see your game history?	Yes or No	Yes
56	How easy does the game history function make it to see how you have progressed?	1 to 10	6

Stakeholder name:	Jake Elmer		
Test no.	Question	Answer type	Stakeholder Answer
33	Is there a representation of a 3D cube?	Yes or No	Yes
34	How professional does the 3D cube look?	1 to 10	7
35	To your knowledge, is every move that's possible on a real Rubik's cube possible on the program's Rubik's cube?	Yes or No	Yes
35	To your knowledge, is every move possible on the program possible on a real Rubik's cube?		Yes
37	How well do the cube controls meet the description: simple and intuitive?	1 to 10	6
38	How well do the program controls meet the description: simple and intuitive?		6
39	Is there a scramble function?	Yes or No	Yes
40	Is there a solve function?		Yes
41	How well does the solve function show each move done (in regards to you being able to understand it)?	1 to 10	6
42	Is there a hint function that shows you the next move to make?	Yes or No	Yes
43	Is there a timer to track how long solves take?		Yes
44	Does the timer start automatically?		Yes
45	Does the timer stop automatically?		Yes
46	Is there a leaderboard?		Yes

47	Does the leaderboard show the ten quickest solves in ascending order?		Yes
48	Does each leaderboard entry display: the username, the time taken, the moves taken?		Yes
49	Is there a login system?		Yes
50	How straightforward to use is the login system?	1 to 10	9
51	Is your user and game data loaded when you log in?	Yes or No	Yes
52	Does the save function run automatically?		Yes
53	Is there a guide to use the program?		Yes
54	How clear and concise is the guide?	1 to 10	6
55	Is there a function to see your game history?	Yes or No	Yes
56	How easy does the game history function make it to see how you have progressed?	1 to 10	6

Results

Test No.	What is being tested	Type	Description	Pass criteria	Stakeholder responses.	Pass / Fail
33	3D cube – image.	Function	There is a 3D representation of a cube.	All stakeholders must answer yes	Yes / Yes / Yes / Yes / Yes	Pass
34	3D cube - professionalism	Useability	The 3D cube must look professional.	Avg. score >= 70%	82%	Pass
35	Logic – possible states.	Function	It must be possible to reach every possible cube state.	All stakeholders must answer yes.	Yes / Yes / Yes / Yes / Yes	Pass
36	Logic – possible moves.	Function / Robustness	All moves able to be done to the cube must be possible on a real Rubik's cube.		Yes / Yes / Yes / Yes / Yes	Pass
37	Controls - cube.	Useability	The controls for interacting with the cube must be simple and intuitive.	Avg. score >= 70%	76%	Pass
38	Controls – program.	Useability	The controls for interacting with the program must be simple and intuitive.		76%	Pass
39	Scramble.	Function	There must be scramble function to scramble the cube.	All stakeholders must answer yes.	Yes / Yes / Yes / Yes / Yes	Pass
40	Solver – solves cube.	Function	There must be a solve function that solves the cube.		Yes / Yes / Yes / Yes / Yes	Pass
41	Solver – showcase moves.	Function, Useability	The solve function must show each move being done to solve the cube in an understandable manner.	Avg. score >= 70%	52%	Fail

42	Hints.	Function	A hint feature should show the user the next move to make.	All stakeholders must answer yes.	Yes / Yes / Yes / Yes / Yes	Pass
43	Timer – timing.	Function	A timer should be available to time solves.		Yes / Yes / Yes / Yes / Yes	Pass
44	Timer – auto start.	Useability	The timer automatically starts.		Yes / Yes / Yes / Yes / Yes	Pass
45	Timer – auto stop.	Useability	The timer automatically stops.		Yes / Yes / Yes / Yes / Yes	Pass
46	Leaderboard – image.	Function	There is a leaderboard.		Yes / Yes / Yes / Yes / Yes	Pass
47	Leaderboard – solves.	Function	The leaderboard should display the ten quickest solve times or more, in ascending order.		Yes / Yes / Yes / Yes / Yes	Pass
48	Leaderboard – details.	Function	Each entry must display: username, the number of moves required, the time taken.		Yes / Yes / Yes / Yes / Yes	Pass
49	Login System – logs in.	Function	There should be a login system that allows users to login in.		Yes / Yes / Yes / Yes / Yes	Pass
50	Login System – straightforward.	Useability	The login system should be straightforward and easy to use.	Avg. score >= 70%	86%	Pass
51	Save – loading.	Function	Upon logging in user data should be loaded.	All stakeholders must answer yes.	Yes / Yes / Yes / Yes / Yes	Pass
52	Save – automatic.	Useability	The save function should run automatically.		Yes / Yes / Yes / Yes / Yes	Pass
53	Guide – exists.	Function	There must be a guide that shows how to use the program.		Yes / Yes / Yes / Yes / Yes	Pass
54	Guide – user display.	Useability	The guide must be clear and concise.	Avg. score >= 70%	70%	Pass

55	Game history – exists.	Function	There must be a game history function that displays previous game history.	All stakeholders must answer yes.	Yes / Yes / Yes / Yes / Yes	Pass
56	Game history – usefulness.	Usability	The game history function must make it easy to see how you have progressed over time.	Avg. score >= 70%	66%	Fail

Success Criteria

No.	Criteria	Associated Tests	Pass / Partial / Fail
1	There should be a professional looking visual representation of a 3-dimesional Rubik's cube as this is the core of the program and a non-standard cube layout (e.g. a net) may confuse users.	1, 33, 34	Pass
2	The cube should have the correct logic - the result of any moves should match the result of performing the move on a real Rubik's cube.	2-13, 35, 36	Pass
3	There should be a scramble feature able to produce a scramble for the user to solve. The scramble must be possible to solve.	14, 39	Pass
4	A solver feature should be included. This should be able to solve the cube move by move, allowing the user to see the steps required to solve it so they may learn from it.	15-17, 40, 41	Partial
5	A hint feature should tell the user the next move they should make if they require help, as to decrease the chance that the user simply gives up.	18, 42	Pass
6	A timer function should be included to incentivise competitiveness. In line with this the timer must be as easy to use as possible, it should not cause any delays. It will automatically start upon the user's first move and automatically stop when the cube is solved.	19-21, 43-45	Pass
7	A local leaderboard will be included to allow people to compete. This should display, at a minimum: the ten quickest solve times, the respective usernames, and number of moves required.	23-26, 46-48	Pass
8	The leaderboard should only include solves that did not utilise the hint or solve functions.	22	Pass

9	There should be a straightforward login system that utilises encryption and/or hashing for security.	49, 50	Partial
10	There should be a clear and concise guide to using the program to prevent any confusion.	27, 28, 53, 54	Pass
11	There should be a simple to use save function to allow users with limited time to play whenever they wish without worrying if they have enough time for a complete solve.	29, 30, 51	Pass
12	The save function should be able to run automatically to prevent users from losing progress should they forgot to save or something unexpected happens – e.g. power loss.	31, 52	Pass
13	A game history function should be included to allow users to see how they have progressed overtime.	32, 55, 56	Partial

Potential Improvements

Success Criteria

Success Criteria 4

Success criteria 4 relates to the solver. Whilst the solver functions, test 41 failed. This is because the solver “must show each move being done to solve the cube in an understandable manner.” As the solver displays moves by simply doing them, it can be difficult to see the goal of the move. Additionally, as the time between each move is inversely proportional to the number of moves required, it can complete some solves too fast for people to see.

To fix this problem a new sub-feature could be added, with the purpose of displaying all the moves done by the solver in a written format, showing all the moves at the same time. This would allow users to analyse the moves being made and see their overall purpose.

Success Criteria 9

Success criteria 9 relates to the login function, and the aspect this improvement relates to is that states that the login system “utilises encryption and/or hashing for security.” Technically, this criterion has been met, however the purpose of the criteria is to ensure a high level of security. Despite the login function being treated as an unrelated module, I was the one to create it, and I know it only uses a simple Caesar cypher.

To fix this, the login could be updated to use a well-developed login library, or a new login system could be created, using a more secure encryption or hashing function.

Success Criteria 13

Success criteria 13 relates the game history function. The feature does function, however the failure of test 56 indicates that it is not easy to see your progress over time.

To fix this the display of the game history should be changed to a cleaner, easier to read, display. Additionally, metrics could also be displayed on graphs and charts, such as a line graph showing how the user's solve time has varied over time.

Limitations

3D cube

Due to the significantly higher difficulty and associated time requirements of using and learning a 3D engine, I have not implemented a true 3D cube. This means that users cannot freely rotate the cube or see moves being made – they just instantly happen. Introducing a 3D cube would fix this.

To introduce a 3D cube whilst keeping the code limited to Python an engine such as Panda3D could be used, or, if different languages can be used, the more popular Unity or Godot engines would be good engines to use. To allow users to rotate the cube an interface method would need to be added. If the mouse is used this would be very easily as Pygame supports mouse inputs, and this feature is already used in the program.

Multiple cubes

Originally, I believed that implementing multiple cubes would be very difficult and time consuming, and as such I did not implement them. This may reduce how appealing my program is to users, as anyone wanting to quickly try their hand at a more unique Rubik's cube would have to use another program. Additionally, the niche of potentially users who mainly use unique cube types are completely lost.

Now that I am more familiar with the coding requirements, I do not believe this would be as hard to implement as I originally thought. A parent class should be created that is inherited by all the cube types to standardise the methods used to interact with them. Whilst any non-cube would need its own class, all cubes could be managed by one class with the only additional information required being the side length of the cube. The turns and rotations functions could easily be updated to support this, with only the fixed values such as 3 and 2 needing to be changed to side length and side length – 1 respectively. For example, changing this:

```
567     if number == 0: # rotate the top face
568         gd.used_cube[4] = numpy.rot90(
569             m: gd.used_cube[4], k=1, axes=(0, 1)
570         ).tolist()
571     elif number == 2: # rotate the bottom face
572         gd.used_cube[5] = numpy.rot90(
573             m: gd.used_cube[5], k=1, axes=(1, 0)
574         ).tolist()
575     else: # turn the column
576         for i in range(3):
577             gd.used_cube[1][i][n] = face5[i][n]
578             # 2-i flips the row number for the back
579             # 2 - n flips the column number for the back
580             gd.used_cube[5][2 - i][n] = face3[i][2 - n]
581             gd.used_cube[3][2 - i][2 - n] = face4[i][n]
582             gd.used_cube[4][i][n] = face1[i][n]
583
584         if number == 0: # rotate left face
585             gd.used_cube[0] = numpy.rot90(
586                 m: gd.used_cube[0], k=1, axes=(0, 1)
587             ).tolist()
588         elif number == 2: # rotate right face
589             gd.used_cube[2] = numpy.rot90(
590                 m: gd.used_cube[2], k=1, axes=(1, 0)
591             ).tolist()
592
```

To this:

```

567     if number == 0: # rotate the top face
568         gd.used_cube[4] = numpy.rot90(
569             m: gd.used_cube[4], k=1, axes=(0, 1)
570         ).tolist()
571     elif number == side_length: # rotate the bottom face
572         gd.used_cube[5] = numpy.rot90(
573             m: gd.used_cube[5], k=1, axes=(1, 0)
574         ).tolist()
575     else: # turn the column
576         for i in range(side_length):
577             gd.used_cube[1][i][n] = face5[i][n]
578             # 2-i flips the row number for the back
579             # 2 - n flips the column number for the back
580             gd.used_cube[5][side_length - 1 - i][n] = face3[i][2 - n]
581             gd.used_cube[3][side_length - 1 - i][side_length - 1 - n] = face4[i][n]
582             gd.used_cube[4][i][n] = face1[i][n]
583
584     if number == 0: # rotate left face
585         gd.used_cube[0] = numpy.rot90(
586             m: gd.used_cube[0], k=1, axes=(0, 1)
587         ).tolist()
588     elif number == side_length - 1: # rotate right face
589         gd.used_cube[2] = numpy.rot90(
590             m: gd.used_cube[2], k=1, axes=(1, 0)
591         ).tolist()
592

```

Should allow the turns function to work with any `side_length*side_length*side_length` cube, although this is untested.

Explained tips/hints

Due to my limited knowledge on solving Rubik's cubes and the number of possible cube states, I did not implement hint explanations. This would be useful to help users improve, instead of simply doing the move for them.

Implementing this, however, would still be very difficult. The solver would need to be overhauled so that it follows an algorithm, such as the Advanced Fridrich (CFOP) algorithm, instead of undoing the moves that have already been done to the cube. This would break solving the cube down into 4 steps, and each move can be explained by how they help complete that step. The tip could either tell users the step it is helping to solve or fully explain its purpose in solving that step. However, the latter option would require someone writing an explanation for every single move. Once this is done, the explanation can be displayed whenever the hint feature is used.

Online Leaderboard

I did not implement an online leaderboard as I do not have access to a server. However, an online leaderboard would allow for more competition between users as they could see other's scores regardless of where they played, instead of each device having its own leaderboard.

The leaderboard could be shared across devices by making an API with Flask to allow the leaderboard to be accessed from anywhere online. This would require the leaderboard being redone to make it fetch the leaderboard from online instead of loading the file. By using things such an API key, this could also be made secure and therefore it would be harder for people to cheat. Currently, anyone can create any leaderboard entry they want by simply editing the leaderboard.txt file.

Mouse controls

I did not implement mouse controls as I was not sure how to stop them from being as clunky as they were in similar programs. Unfortunately, I still am unsure about how to prevent this. The corner positions of each square on the cube would need to be recorded, as well as the corner positions of the cube itself, to check where the user is trying to interact with. It would also need to be checked if the user is actively holding the left mouse button. The change in x and y position of the cursor would need to be recorded whilst the button is held will need to be measure. If this is going to be implemented more research will need to be done.

Maintenance

Currently, whilst the program can be further developed, the unorganized file structure may make it confusing and errors more likely. To combat this, the files should be broken down and restructured. For example, the features.py file could be separated into a file for every feature, that all get stores in a features folder. The saves.txt file and the new save.py file cold themselves be bundled into a saves folder inside the feature folder.

Appendix

main.py

.....

This is the file to run to execute the program

This file handles the main loop of the program, it gets images and data from the other files and displays them. It also handles user input within the game loop.

black, isort and flake8 used for formatting

:::::

```
import sys
import time

import cube
import features
import game_data # for changing variables in data file
import interface
import pygame
import user_data
from game_data import *
from Login import login_window
from validation import ValidateScreenPositions

# window
pygame.init()
width = 1600
height = 900
screen = pygame.display.set_mode((width, height), pygame.RESIZABLE)
pygame.display.set_caption("Rubik's Cube")

# validation
val = ValidateScreenPositions(width, height)

# cubes and visuals
cube_net = cube.CubeNet(screen, val.run((width // 2, height // 2)))
cube_3d = cube.Cube3D(screen, val.run((width // 2, height // 2)))
cube_guide = cube.CubeGuide(screen, val.run((width // 2, height // 2)))
```

```
display_history = features.DisplayHistory(screen, val.run((width // 2, height // 2)))  
display_leaderboard = features.Leaderboard(screen, val.run((width // 2, height // 2)))
```

class Buttons:

"""

This class handles the rendering of the buttons

This class is largely self-contained, the only usage should be to run
.update as this automatically updates the buttons

"""

```
cube_option = interface.DisplayOption(  
    lambda: cube_3d.get_image(),  
    screen,  
    val.run([10, 0]),  
    [100, 100],  
    1.5,  
    lambda: Buttons.display_swap("3d"),  
    default_colour,  
)  
  
net_option = interface.DisplayOption(  
    lambda: cube_net.get_image(),  
    screen,  
    val.run([10, 100]),  
    [100, 100],  
    1.5,  
    lambda: Buttons.display_swap("net"),  
    default_colour,  
)  
  
guide_option = interface.DisplayOption(
```

```

lambda: cube_guide.get_image(),
screen,
val.run([10, 200]),
[100, 100],
1.5,
lambda: Buttons.display_swap("guide"),
BLACK,
) # should be default colour,
# but this causes the background of the hovered button to be black.
# May be an error with pygame.smoothscale in interface file
# this works as a solution

```

```

history_option = interface.DisplayOption(
    lambda: interface.text(
        "HISTORY",
        default_font,
        BLACK,
        default_colour
    ),
    screen,
    val.run([10, 300]),
    [100, 25],
    1.5,
    lambda: Buttons.display_swap("history"),
    BLACK, # same problem as guide
)

```

```

leaderboard_option = interface.DisplayOption(
    lambda: interface.text(
        "LEADERBOARD",
        default_font,

```

```

        BLACK,
        default_colour
    ),
    screen,
    val.run([10, 325]),
    [100, 25],
    1.5,
    lambda: Buttons.display_swap("leaderboard"),
    BLACK, # same problem as guide
)

cube_option_bar = interface.DisplayBar( # update with any new options
[cube_option, net_option, guide_option, history_option, leaderboard_option],
False,
)
display_option = "3d"

@staticmethod
def display_swap(option):
    """
    Updates display_option variable within the class
    This provides a function for interface.DisplayOption objects
    to update the display_option variable which is saved with this class
    :param option: the new display_option: 3d, net, guide, history or leaderboard
    :type option: str
    """
    Buttons.display_option = option

@staticmethod

```

```

def update(mouse_pos, mouse_up):
    """
    Updates each button in the class

    :param mouse_pos: the x,y position of the mouse
    :param mouse_up: whether the mouse button has been clicked
    :type mouse_pos: tuple[int, int] or list[int, int]
    :type mouse_up: bool
    :rtype: None
    """

    Buttons(cube_option_bar).update(mouse_pos, mouse_up)

# used for solving the cube
solve_cube = False
"""If the cube is being solved
:type solve_cube: bool"""

solver = features.Solver()

timer = features.Timer()
last_save = time.time()
"""The timestamp of the last save, used for calculating time since last save
:type last_save: float"""

# login
def load(username):
    """Desgined to be called by the login window, this function will load the users data

    Uses Manager.load to load the users data and then checks the game state, updating
    details about the timer and solver is nessesar
```

```

:param username: the unique username of the user
:type username: str
"""

user_data.Manager.load(username)

if game_data.time_taken > 0: # timer is running
    # manually start timer to avoid changing start time
    timer.exists = True
    timer.running = True
    timer.start_time = (
        time.time() - game_data.time_taken
    ) # act as if timer has just started

if game_data.solver_used: # solver is runnning
    # finish solving cube
    solver.first = False
    solve_cube = True

login_window.Window(lambda u: load(u))

# game loop
while True:
    mouse_pos = pygame.mouse.get_pos()
    mouse_up = False
    val.update_size(pygame.display.get_surface().get_size())

    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()

```

```

    sys.exit()

elif event.type == pygame.MOUSEBUTTONUP:

    mouse_up = True

elif event.type == pygame.MOUSEWHEEL and Buttons.display_option == "history":

    display_history.scroll(event.y * 25)

# prevent any moves made whilst on guide cube

elif event.type == pygame.KEYDOWN and Buttons.display_option != "guide":

    # row right

    if event.key == pygame.K_t:

        cube.turn(True, 0)

    elif event.key == pygame.K_g:

        cube.turn(True, 1)

    elif event.key == pygame.K_b:

        cube.turn(True, 2)

    # row left

    elif event.key == pygame.K_r:

        cube.turn(True, 0, True)

    elif event.key == pygame.K_f:

        cube.turn(True, 1, True)

    elif event.key == pygame.K_v:

        cube.turn(True, 2, True)

    # column up

    elif event.key == pygame.K_q:

        cube.turn(False, 0)

    elif event.key == pygame.K_w:

        cube.turn(False, 1)

    elif event.key == pygame.K_e:

        cube.turn(False, 2)

    # column down

    elif event.key == pygame.K_a:

```

```

        cube.turn(False, 0, True)

    elif event.key == pygame.K_s:
        cube.turn(False, 1, True)

    elif event.key == pygame.K_d:
        cube.turn(False, 2, True)

    # rotations

    elif event.key == pygame.K_x:
        cube.rotate("x")

    elif event.key == pygame.K_y:
        cube.rotate("y")

    elif event.key == pygame.K_z:
        cube.rotate("z")

    elif event.key == pygame.K_k: # solve
        game_data.solver_used = True
        if timer.running: # ensures the attempt was started
            # failed attempts should be recorded
            game_data.solved = False
            user_data.game_history.add_game()
            timer.delete()

        solve_cube = True

    elif event.key == pygame.K_m: # scramble
        if timer.running: # ensures the attempt was started
            # failed attempts should be recorded
            user_data.game_history.add_game()

        # reset key data
        game_data.moves.clear()
        game_data.move_count = 0

```

```

game_data.scrambler_count = 0
game_data.hints_used = False
game_data.solver_used = False
game_data.solved = False
game_data.time_taken = 0
game_data.start_time = time.time()

features.scramble()
# prevent the timer from being started whilst the solver runs
# was achieved by scrambling whilst the timer ran
solve_cube = False
timer.start() # start timer
elif event.key == pygame.K_h: # hint
    game_data.hints_used = True
    solver.pop_move()

screen.fill(default_colour) # background colour

if solve_cube:
    # ensures each solve take 5 sections, assuming no hardware limitations
    time.sleep(solver.sleep_time)
    solve_cube = solver.solve() # solves one move
else:
    solver.first = True # so next solve it is set to true

if timer.running and solver.check_solved(): # on a solve
    timer.stop()
    game_data.solved = True
    user_data.game_history.add_game()
    display_leaderboard.update_list(
        game_data.time_taken,

```

```

    game_data.move_count
)

if Buttons.display_option == "3d":
    display_cube = cube_3d
elif Buttons.display_option == "net":
    display_cube = cube_net
elif Buttons.display_option == "guide":
    # also prevents cube interact as uses default
    display_cube = cube_guide
# actions text
screen.blit(
    interface.text(
        text="Scramble: M",
        font=guide_font,
        foreground_colour=BLACK,
        background_colour=default_colour,
    ),
    val.run((1100, 300)),
)
screen.blit(
    interface.text(
        text="Solve: K",
        font=guide_font,
        foreground_colour=BLACK,
        background_colour=default_colour,
    ),
    val.run((1100, 350)),
)
screen.blit(
    interface.text(

```

```

text="Hint: H",
font=guide_font,
foreground_colour=BLACK,
background_colour=default_colour,
),
val.run((1100, 400)),
)

elif Buttons.display_option == "history":
    display_cube = display_history

elif Buttons.display_option == "leaderboard":
    display_cube = display_leaderboard

display_cube.update() # actually update cube

if timer.exists: # display timer
    screen.blit(timer.display_elapsed(), val.run((1400, 200)))
    timer.update()

# update buttons
Buttons.update(mouse_pos, mouse_up)

# save every 5 seconds
if time.time() - last_save > 5:
    time_since_save = time.time()
    user_data.Manager.save()

pygame.display.flip()

```

validation.py

.....

This file contains validation functions and error handling

black, isort and flake8 used for formatting

:::::

import time

class InvalidScreenPosition(Exception):

"""This exception is raised when the screen position is invalid"""

def __init__(self, pos):

:::::

:param pos: the x,y position that is invalid

:type pos: tuple[int, int] or list[int]

:::::

super().__init__(f"Invalid Screen Position: {pos}")

with open("error.txt", "a") as f:

 error_time = time.time()

 f.write(f"{error_time} Invalid Screen Position: {pos} \n")

class ValidateScreenPositions:

:::::

This class contains a screen position validation function

It will ensure a screen position is valid based on a 4k resolution screen

and the size of the window being displayed to

:::::

def __init__(self, width, height):

:::::

```
:param width: the width of the screen window
:param height: the height of the screen window
:type width: int
:type height: int
"""
self.width = width
self.height = height

def run(self, pos):
"""

Ensures a position is within the confines of the screen and 4k resolution
```

This function will throw an error if the position is invalid.

An invalid position is one that is outside the confines of the screen based width and height, or a 4k resolution screen.

```
:param pos: the x,y position to check
:type pos: tuple[int, int] or list[int]
:return: the x,y screen position if it is valid, else raises an exception
:rtype: tuple[int, int] or list[int]
"""

if (
    pos[0] < 0
    or pos[0] > self.width
    or pos[0] > 3840
    or pos[1] < 0
    or pos[1] > self.height
    or pos[1] > 2160
):
    raise InvalidScreenPosition(pos)
else:
```

```
    return pos

def update_size(self, size):
    """
    Update the screen width and height

    :param size: the width and height of the window
    :type size: tuple[int, int] or list[int]
    """

    self.width = size[0]
    self.height = size[1]
```

[cube.py](#)

This file contains the code for the cube as well as the turn and rotation functions

This file handles creating the images to display the cube
and the basic turn and rotation functions for interacting with the cube

black, isort and flake8 used for formatting

```
import copy

import game_data as gd
import interface
import numpy
import pygame
from game_data import BLACK, default_colour, default_cube
```

```
class CubeNet:  
    """Handles the display of the cube as a net to a fixed position on the screen"""  
  
    def __init__(self, surface, pos):  
        """  
        :param surface: The surface that this cube is to be blitted to  
        :param pos: The centre position that this cube is to be blitted to: x,y  
        :type surface: pygame.Surface  
        :type pos: list[int] or tuple[int, int]  
        """  
        # pos is centre  
        self.screen = surface  
        self.pos = pos  
  
    def update(self):  
        """  
        Updates the cube image and re-blits it to the surface  
        :rtype: None  
        """  
        image = self.get_image()  
        self.screen.blit(image, image.get_rect(center=self.pos))  
  
    @staticmethod  
    def get_image(default=False):  
        """  
        Creates the image of the cube from the current state of the cube  
        :param default: if True, uses the default image instead of the current state  
        :type default: bool  
        :return: the image of the cube as a 720x540 surface
```

```
:rtype: pygame.Surface
"""

surf = pygame.Surface((720, 540))
surf.fill(default_colour)
colour_3d_array = gd.used_cube

if default:
    colour_3d_array = default_cube


def square(colour):
    """
Creates a single square with the given colour

:param colour: the RGB values of the colour
:type colour: tuple[int, int, int]
:return: the square image, 50x50
:rtype: pygame.Surface
"""

surf = pygame.Surface((50, 50))
surf.fill(colour)
return surf


def row(colour_list):
    """
Creates the image of a row of 3 squares

:param colour_list: List len(3) of tuples, where each tuple is an RGB value
:type colour_list: list[tuple[int, int, int]]
:return: the row image, 170*50
:rtype: pygame.Surface
"""

surf = pygame.Surface((170, 50))
```

```

surf.fill(default_colour)

for i in range(3):

    # iterates alongside the list of colours,
    # getting a square with the respective colour and
    # blitting it to calculated position

    # i * 50 ensures the square is blitted after the previous one;
    # not inside it

    # i * 10 adds 10 spacing between the cubes

    surf.blit(square(colour_list[i]), (i * 50 + i * 10, 0))

return surf

```

```

def face(colour_array):
    """
Creates one face (side) from 3 rows

:param colour_array: 2D array (3x3)(row x col) of tuples,
    where each tuple is an RGB value
:type colour_array: list[list[tuple[int, int, int]]]
:return: the face image, 170*170
:rtype: pygame.Surface
    """

surf = pygame.Surface((170, 170))
surf.fill(default_colour)

for i in range(3):

    # iterates alongside the list of rows,
    # getting and blitting the row image to calculated position

    # i * 50 ensures the row is placed beneath,
    # and not inside, the previous row

    # i * 10 is for spacing between the rows

    surf.blit(row(colour_array[i]), (0, i * 50 + i * 10))

return surf

```

```

# 4 of the faces are placed next to each other so a loop can place them
for i in range(4):
    surf.blit(
        face(colour_3d_array[i]), # gets the image of the face
        # 180 * i includes 10 pixels spacing
        # placed 180 down to allow top to be placed above with 10 pixels spacing
        (180 * i, 180),
    )

# 180 x val aligns with front face
surf.blit(face(colour_3d_array[4]), (180, 0))

# 360 is below face image with 10 pixels spacing
surf.blit(face(colour_3d_array[5]), (180, 360))

return surf

```

class Cube3D(CubeNet):

"""

Handles the display of the 3d cube to a fixed position on the screen

This class is a child of CubeNet, only changing the get_image method

"""

@staticmethod

def get_image(default=False):

"""

Creates the image of the cube from its current state

:param default: if True, uses the default image instead of the current state

:type default: bool

```

:rtype: the cube image, 365*335
:rtype: pygame.Surface

"""
surf = pygame.Surface((365, 335))
surf.fill(default_colour)
colour_3d_array = gd.used_cube
if default:
    colour_3d_array = default_cube

def right():

"""
Draws the right face of the cube to the surf, it is a slanted square

:rtype: None
"""

def square(colour):

"""
Creates a slanted cube of solid colour

:param colour: RGB values
:type colour: tuple[int, int, int]
:return: the square image, 50*75
:rtype: pygame.Surface

"""
surf = pygame.Surface((50, 75))
surf.fill(default_colour)
pygame.draw.polygon(surf, colour, ((0, 25), (50, 0), (50, 50), (0, 75)))
surf.set_colorkey(default_colour) # make background transparent
return surf

```

```

def row(colour_list):
    """
Creates a slanted row of 3 squares

:param colour_list: List len(3) of tuples of RGB values
:type colour_list: list[tuple[int, int, int]]
:return: the row image, 165*135
:rtype: pygame.Surface
"""

surf = pygame.Surface((165, 135))
surf.fill(default_colour)
for i in range(3):
    # 55 includes 5 pixels spacing
    # -30 includes 5 pixels spacing
    surf.blit(square(colour_list[i]), (55 * i, 60 - (30 * i)))
surf.set_colorkey(default_colour)
return surf


def face(colour_array):
    """
Stacks 3 row images to create a face of 9 squares

:param colour_array: 2D (3x3)(row x col) array of tuples of RGB values
:type colour_array: list[list[tuple[int, int, int]]]
:return: the row image, 165*250
:rtype: pygame.Surface
"""

surf = pygame.Surface((165, 250))
surf.fill(default_colour)
for i in range(3):
    surf.blit(row(colour_array[i]), (0, 55 * i))

```

```
    return surf

# positioned to the right of front face with 5 pixels spacing
surf.blit(face(colour_3d_array[2]), (205, 90))
```

```
def front():
```

```
    """
```

Draws the front face of the cube to the surf, it s a slanted square

:rtype: None

```
    """
```

```
def square(colour):
```

```
    """
```

Creates a slanted cube of solid colour

:param colour: RGB values

:type colour: tuple[int, int, int]

:return: the square image

:rtype: pygame.Surface

```
    """
```

```
surf = pygame.Surface((50, 75))
```

```
surf.fill(default_colour)
```

```
pygame.draw.polygon(surf, colour, ((0, 0), (50, 25), (50, 75), (0, 50)))
```

```
surf.set_colorkey(default_colour)
```

```
return surf
```

```
def row(colour_list):
```

```
    """
```

Creates the image of a row of 3 squares

```

:param colour_list: List len(3) of tuples of RGB values
:type colour_list: list[tuple[int, int, int]]
:return: the row image
:rtype: pygame.Surface
"""

surf = pygame.Surface((165, 135))
surf.fill(default_colour)
for i in range(3):
    # 55 includes 5 pixels spacing
    # 30 includes 5 pixels spacing
    surf.blit(square(colour_list[i]), (55 * i, 30 * i))
surf.set_colorkey(default_colour)
return surf

def face(colour_array):
"""

Stacks 3 row images to create a face of 9 squares

:param colour_array: 2D array (3x3)(row x col) of tuples of RGB values
:type colour_array: list[list[tuple[int, int, int]]]
:return: the face image
:rtype: pygame.Surface
"""

surf = pygame.Surface((165, 250))
surf.fill(default_colour)
for i in range(3):
    # 55 includes 5 pixels spacing
    surf.blit(row(colour_array[i]), (0, 55 * i))
surf.set_colorkey(default_colour)
return surf

```

```
# positioned below top of front face with 5 pixels spacing
surf.blit(face(colour_3d_array[1]), (40, 90))
```

```
def top():
```

```
"""
```

```
    Draws the top face of the cube to the surf,
    it is a horizontally stretched square
```

```
:rtype: None
```

```
"""
```

```
def square(colour):
```

```
"""
```

```
    Creates a horizontally stretched cube of solid colour
```

```
:param colour: RGB values
```

```
:type colour: tuple[int, int, int]
```

```
:return: the square image
```

```
:rtype: pygame.Surface
```

```
"""
```

```
surf = pygame.Surface((100, 50))
```

```
surf.fill(default_colour)
```

```
pygame.draw.polygon(
```

```
    surf, colour, ((50, 0), (100, 25), (50, 50), (0, 25))
```

```
)
```

```
surf.set_colorkey(default_colour)
```

```
return surf
```

```
def row(colour_list):
```

```
"""
```

```
    Creates the image of a row of 3 squares
```

```

:param colour_list: List len(3) of tuples of RGB values
:type colour_list: list[tuple[int, int, int]]
:return: the row image
:rtype: pygame.Surface
"""

surf = pygame.Surface((215, 120))
surf.fill(default_colour)
for i in range(3):
    # 55 includes 5 pixels spacing
    # 30 includes 5 pixels spacing
    surf.blit(square(colour_list[i]), (55 * i, 30 * i))
surf.set_colorkey(default_colour)
return surf

def face(colour_array):
"""

Stacks 3 row images to create a face of 9 squares

:param colour_array: 2D array (3x3)(row x col) of tuples of RGB values
:type colour_array: list[list[tuple[int, int, int]]]
:return: the face image
:rtype: pygame.Surface
"""

surf = pygame.Surface((370, 315))
surf.fill(default_colour)
for i in range(3):
    # 150 - to place bottom to top
    # 55 includes 5 pixels spacing
    # 30 includes 5 pixels spacing
    surf.blit(row(colour_array[i]), (150 - (55 * i), 30 * i))

```

```
    surf.set_colorkey(default_colour)

    return surf

    surf.blit(face(colour_3d_array[4]), (0, 0))

right()
front()
top()

return surf
```

```
class CubeGuide(Cube3D):
```

```
    """
```

Class that handles the guide cube and adds instructions

This class inherits from Cube3D and overrides the get_image method

```
    """
```

```
@classmethod
```

```
def get_image(cls):
```

```
    """
```

Creates the image of the default cube with added instructions

```
:return: the cube image, 600*600
```

```
:rtype: pygame.Surface
```

```
    """
```

```
surf = pygame.Surface((600, 600))
```

```
surf.fill(default_colour)
```

```
colour = gd.guide_arrow_colour
```

```
if default_colour == colour:
```

```

# arrows will blend into background
print("BAD idea, change guide arrow colour first")

def arrow_top(text, angle=0):
    """
    Draws an arrow aligned with the cubes slant on the top edge

    :param text: text to draw above the arrow
    :param angle: clockwise angle to rotate the arrow, 0 is upwards
    :type text: str
    :type angle: int
    :return: the image of the arrow, 100*100
    :rtype: pygame.Surface
    """

    surf = pygame.Surface((100, 100))
    surf.fill(default_colour)
    pygame.draw.polygon(
        surf,
        colour,
        ((13, 13), (50, 0), (63, 63), (50, 50), (50, 93), (25, 80), (25, 25)),
    )
    surf.set_colorkey(default_colour)
    # angle
    surf = pygame.transform.rotate(surf, angle)
    # letter
    surf.blit(
        interface.text(
            text=text,
            font=gd.guide_font,
            foreground_colour=BLACK,
            background_colour=gd.default_colour,
        )
    )

```

```

        ),
        (15, 0),
    )
return surf

def arrow_right(text, angle=0):
    """
    Draws an arrow aligned with the cubes slant on the right edge

    :param text: text to draw above the arrow
    :param angle: clockwise angle to rotate the arrow, 0 is right
    :type text: str
    :type angle: int
    :return: the image of the arrow, 100*100
    :rtype: pygame.Surface
    """

    surf = pygame.Surface((100, 100))
    surf.fill(default_colour)
    pygame.draw.polygon(
        surf,
        colour,
        (
            (38, 50),
            (63, 25),
            (63, 38),
            (100, 38),
            (100, 63),
            (63, 63),
            (63, 75),
        ),
    )

```

```

surf.set_colorkey(default_colour)

# angle

surf = pygame.transform.rotate(surf, angle)

# letter

surf.blit(
    interface.text(
        text=text,
        font=gd.guide_font,
        foreground_colour=BLACK,
        background_colour=gd.default_colour,
    ),
    (35, 25),
)
return surf

```

def arrow_rotate(text, angle=0):

"""

Draws a large straight arrow

:param text: text to draw above the arrow

:param angle: clockwise angle to rotate the arrow, 0 is right

:type text: str

:type angle: int

:return: the image of the arrow, 100*100

:rtype: pygame.Surface

"""

surf = pygame.Surface((200, 100))

surf.fill(default_colour)

pygame.draw.polygon(

surf,

colour,

```

(
    (200, 50),
    (150, 100),
    (150, 75),
    (0, 75),
    (0, 25),
    (150, 25),
    (150, 0),
),
)

surf.set_colorkey(default_colour)
# angle
surf = pygame.transform.rotate(surf, angle)
# letter
surf.blit(
    interface.text(
        text=text,
        font=gd.guide_font,
        foreground_colour=BLACK,
        background_colour=gd.default_colour,
    ),
    (0, 0),
)
return surf

```

```

# offsets allow moving cube and arrows
# whilst maintaining thier relative position to each other
cube_offset_x = 100
cube_offset_y = 50

# cube

```

```

surf.blit(super().get_image(True), (cube_offset_x, cube_offset_y))

# up
surf.blit(arrow_top("Q"), (cube_offset_x + 30, cube_offset_y + 13))
surf.blit(arrow_top("W"), (cube_offset_x + 90, cube_offset_y + 45))
surf.blit(arrow_top("E"), (cube_offset_x + 150, cube_offset_y + 73))

# left
surf.blit(arrow_right("R"), (cube_offset_x + 100, cube_offset_y + 150))
surf.blit(arrow_right("F"), (cube_offset_x + 100, cube_offset_y + 200))
surf.blit(arrow_right("V"), (cube_offset_x + 100, cube_offset_y + 250))

# right
surf.blit(arrow_right("T", 180), (cube_offset_x + 205, cube_offset_y + 152))
surf.blit(arrow_right("G", 180), (cube_offset_x + 205, cube_offset_y + 202))
surf.blit(arrow_right("B", 180), (cube_offset_x + 205, cube_offset_y + 252))

# down
surf.blit(arrow_top("A", 180), (cube_offset_x, cube_offset_y + 250))
surf.blit(arrow_top("S", 180), (cube_offset_x + 50, cube_offset_y + 277))
surf.blit(arrow_top("D", 180), (cube_offset_x + 100, cube_offset_y + 305))

# rotate
surf.blit(arrow_rotate("X"), (cube_offset_x + 50, cube_offset_y + 400))
surf.blit(arrow_rotate("Y", 90), (cube_offset_x - 100, cube_offset_y + 100))
surf.blit(arrow_rotate("Z", 335), (cube_offset_x + 250, cube_offset_y - 50))

return surf

```

```
def turn(row_col, number, backwards=False, ignore_moves=False):
```

....

Turn 1 row or column once in a given direction, default is right/up

```
:param row_col: row is True, column is False  
:param number: the number to do, left to right or top to bottom  
:param backwards: do the opposite of the move/do the move 3 times if true  
:param ignore_moves: don't add the move to the moves list  
:type row_col: bool  
:type number: int  
:type backwards: bool  
:type ignore_moves: bool  
:rtype: None
```

....

if not ignore_moves:

```
# add the move to the moves list  
# ignoring is useful for solving  
gd.moves.push({"direction": row_col, "number": number, "backwards": backwards})  
gd.move_count += 1
```

else:

```
gd.move_count -= 1
```

loop to turn the row or column the correct number of times

```
loop = 1
```

if backwards: # 3 right is used to achieve 1 left, 3 up to achieve 1 down

```
loop = 3
```

for _ in range(loop):

```
# make copies of the faces of the cube so the original state isn't lost  
# deepcopy prevents pass by reference shenanigans  
# by copying the value instead of creating a reference
```

```
face0 = copy.deepcopy(gd.used_cube[0])
face1 = copy.deepcopy(gd.used_cube[1])
face2 = copy.deepcopy(gd.used_cube[2])
face3 = copy.deepcopy(gd.used_cube[3])
face4 = copy.deepcopy(gd.used_cube[4])
face5 = copy.deepcopy(gd.used_cube[5])
```

```
n = number
```

```
if row_col: # turn the row
```

```
(  
    gd.used_cube[2][n],  
    gd.used_cube[3][n],  
    gd.used_cube[0][n],  
    gd.used_cube[1][n],  
) = (  
    face1[n],  
    face2[n],  
    face3[n],  
    face0[n],  
)
```

```
if number == 0: # rotate the top face
```

```
gd.used_cube[4] = numpy.rot90(  
    gd.used_cube[4], k=1, axes=(0, 1)  
)
```

```
elif number == 2: # rotate the bottom face
```

```
gd.used_cube[5] = numpy.rot90(  
    gd.used_cube[5], k=1, axes=(1, 0)  
)
```

```
else: # turn the column
```

```
for i in range(3):
```

```

gd.used_cube[1][i][n] = face5[i][n]
# 2-i flips the row number for the back
# 2 - n flips the column number for the back
gd.used_cube[5][2 - i][n] = face3[i][2 - n]
gd.used_cube[3][2 - i][2 - n] = face4[i][n]
gd.used_cube[4][i][n] = face1[i][n]

if number == 0: # rotate left face
    gd.used_cube[0] = numpy.rot90(
        gd.used_cube[0], k=1, axes=(0, 1)
    ).tolist()

elif number == 2: # rotate right face
    gd.used_cube[2] = numpy.rot90(
        gd.used_cube[2], k=1, axes=(1, 0)
    ).tolist()

def rotate(axis, ignore_moves=False):
    """
    Rotates the view of the cube without changing layout

    :param axis: x, y, z
    :type axis: str
    :param ignore_moves: whether to add the move to the moves list, defaults to False
    :type ignore_moves: bool or optional
    :rtype: None
    """

    # make copies of the faces of the cube so the original state isn't lost
    # deepcopy prevents pass by reference shenanigans
    # by copying the value instead of creating a reference
    face0 = copy.deepcopy(gd.used_cube[0])

```

```

face1 = copy.deepcopy(gd.used_cube[1])
face2 = copy.deepcopy(gd.used_cube[2])
face3 = copy.deepcopy(gd.used_cube[2])
face3 = copy.deepcopy(gd.used_cube[3])
face4 = copy.deepcopy(gd.used_cube[4])
face5 = copy.deepcopy(gd.used_cube[5])

if not ignore_moves: # add the move to the moves list
    # ignoring is useful for solving
    gd.moves.push({"rotation": True, "direction": axis})
    gd.move_count += 1
else:
    gd.move_count -= 1

if axis == "x":
    for i in range(3): # equivalent to a rotation along the x axis
        turn(True, i, ignore_moves=True)
elif axis == "y":
    for i in range(3): # equivalent to a rotation along the y axis
        turn(False, i, ignore_moves=True)
elif axis == "z": # equivalent to a rotation along the z axis
    # rotate the front and back faces
    gd.used_cube[1] = numpy.rot90(gd.used_cube[1], k=1, axes=(1, 0)).tolist()
    gd.used_cube[3] = numpy.rot90(gd.used_cube[3], k=1, axes=(0, 1)).tolist()

    # required a lot of manual testing
    # carefully test any changes
for j in range(3):
    for i in range(3):
        gd.used_cube[0][j][2 - i] = face5[i][j]
        gd.used_cube[4][j][2 - i] = face0[i][j]

```

```
gd.used_cube[2][j][2 - i] = face4[i][j]
gd.used_cube[5][j][2 - i] = face2[i][j]
```

features.py

:::::

This file contains all the features of the program available to the user

These provide additional functionality
beyond the basic turn and rotation functions of the cube

black, isort and flake8 used for formatting

:::::

```
import copy
import time
from random import randint

import game_data as gd
import interface
import numpy
import pygame
import tools
import user_data as ud
from cube import rotate, turn
from game_data import BLACK, default_colour, default_cube, default_font
from validation import ValidateScreenPositions
```

```
val = ValidateScreenPositions(1600, 900)
```

```
def scramble():
```

"""

Randomly scrambles the cube by making between 15 and 25 moves randomly

:rtype: None

"""

reset the cube

```
gd.used_cube = copy.deepcopy(default_cube)
```

```
count = randint(15, 25)
```

```
gd.scrambler_count = count
```

```
for _ in range(count):
```

```
    # randomise every aspect of the turn
```

```
    direction = bool(randint(0, 1))
```

```
    number = randint(0, 2)
```

```
    backwards = bool(randint(0, 1))
```

```
    turn(direction, number, backwards)
```

class Solver:

"""

Solve the cube, one turn per game loop

The solve function must be called once per game loop

until it returns False

to completely solve the cube

The attribute first should be updated to True before each complete solve

A solve can optionally be made to take 5 seconds. To do this, implement a

time.sleep(this_object.sleep_time) before the this_object.solve() call

```

"""
def __init__(self):
    self.first = True
    """If it is the first move of the solve
    :type: bool"""

    self.sleep_time = 0.2
    """The amount of time to wait between each move
    :type: float"""

def solve(self):
    """
    Does the reverse of the last done move and removes it from the moves list

    :return: False if the cube is solved, True otherwise
    :rtype: bool
    """

    # guard clause
    if gd.moves.size() == 0 or self.check_solved():
        return False

    # calculate time to wait between move
    if self.first:
        if gd.moves.size() > 0:
            # every solve should take 5 seconds regardless of moves required,
            # although this can be affected by hardware limitations
            self.sleep_time = 5 / gd.moves.size()
            self.first = False
    else:
        # wait upon every button press so the user knows it has 'worked'
        # even when the cube is already solved

```

```
    self.sleep_time = 1
```

```
    return self.pop_move()
```

```
@staticmethod
```

```
def check_solved():
```

```
    """
```

```
    Checks whether the cube is in a solved state
```

```
:return: True if the cube is solved, False otherwise
```

```
:rtype: bool
```

```
    """
```

```
    not_solved = False
```

```
    for i in range(6): # face
```

```
        for j in range(3): # row
```

```
            for k in range(3): # column
```

```
                # checks for any square not the same colour
```

```
                # as the middle square on the same face
```

```
                # numpy.all handles it being a tuple comparison
```

```
                if not numpy.all(gd.used_cube[i][j][k] == gd.used_cube[i][1][1]):
```

```
                    not_solved = True
```

```
# sys.exit()
```

```
    return not not_solved
```

```
@staticmethod
```

```
def pop_move():
```

```
    """
```

```
    Removes a move from the moves list and does the reverse
```

```
:return: False if the cube is solved, True otherwise
```

```
:rtype: bool
```

```

"""
# guard clause

if gd.moves.size() == 0:
    return False

move = gd.moves.pop() # get the move dictionary

if "rotation" in move.keys(): # check if the move was a rotation
    # rotate does not have a backwards parameter,
    # so achieve via 3 'forward' turns
    for _ in range(3):
        # ignore move as it is part of the solve, not the user or scramble
        rotate(move["direction"], ignore_moves=True)

else: # if not rotation must be turn
    # not move["backwards"] to always undo the move
    # ignore move as part of solve
    turn(move["direction"], move["number"], not move["backwards"], True)

if gd.moves.size() == 0: # must be solved
    return False
else:
    return True # continue solving

```

```

class Timer:
    """
    This class handles timing how long it takes the user to complete a solve
    """

```

```

def __init__(self):
    self.start_time = 0.0
    """
    The time since epoch that the timer was started
    :type: float
    """
    self.end = 0.0
    """
    The time since epoch that the timer was stopped
    """

```

```

:type: float"""
self.elapsed = 0.0

"""The amount of time that has elapsed since the timer was started
:type: float"""

self.exists = False

"""Whether the timer has ever been started for this solve
:type: bool"""

self.running = False

"""Whether the timer is actively running
:type: bool"""

def start(self):
    """Starts the timer and marks it as running"""
    self.exists = True
    self.running = True
    self.start_time = time.time()
    gd.start_time = self.start_time

def stop(self):
    """Gets the final time elapsed and stops the timer"""
    self.update()
    self.running = False

def delete(self):
    """Marks the timer as not having run for the current solve"""
    self.exists = False
    self.running = False
    gd.time_taken = 0.0
    gd.start_time = 0.0

def update(self):

```

```

"""Updates the time elapsed if the timer is running"""

if self.running:

    self.end = time.time()

    self.elapsed = self.end - self.start_time

    gd.time_taken = self.elapsed


def display_elapsed(self):
    """
    Creates a text image displaying the time elapsed

    :return: The text image
    :rtype: pygame.Surface
    """

    # if time is less than a minute

    if self.elapsed < 60: # display time as seconds and milliseconds

        image = interface.text(
            str(round(self.elapsed, 3)) + " seconds", # round to milliseconds
            gd.default_font,
            BLACK,
            default_colour,
        )

    else: # display time as minutes and seconds

        image = interface.text(
            str(int(self.elapsed / 60)) # minutes
            + "m "
            + str(int(self.elapsed % 60)) # seconds
            + "s",
            gd.default_font,
            BLACK,
            default_colour,
        )

```

```

return image

class DisplayHistory:

    """This class manages fetching and displaying the user's game history"""

    def __init__(self, screen, pos):
        """
        :param screen: The screen that this is to be blitted to
        :param pos: The top-left position that this is to be blitted to: x,y
        :type screen: pygame.Surface
        :type pos: list[int] or tuple[int, int]
        """

        self.screen = screen
        self.pos = pos

        self.history = []
        """A 2D array where each row is a game and each column is text to display
        :type: list[list]"""

        self.y_offset = 0
        """The amount the image should be offset vertically
        - the amount it has been scrolled
        :type: int"""

    def format_history(self):
        """Formats the user's game history into a 2D array
        that contains elements to be displayed"""
        history_data = ud.game_history.get_history()
        self.history = []

```

```

for i in range(len(history_data)): # one game
    game = history_data[i]
    game_array = []

    # date of solve
    game_array.append(str(time.strftime("%d/%m/%Y", time.localtime(game[5]))))

    # solved or unsolved
    if game[6]:
        game_array.append("SOLVED")
    else:
        game_array.append("UNSOLVED")

    # move count for the user
    game_array.append(str(game[1] - game[3]))

    # time taken
    game_array.append(str(time.strftime("%H:%M:%S", (time.gmtime(game[4])))))

    # hints used
    game_array.append(str(game[7]))

    self.history.append(game_array)

def get_image(self):
    ....

```

Creates a text image displaying the user's game history

This will get and format the user's game history before creating the image

....

```

self.format_history()

img_height = 0 # will vary on size of history
img_list = [] # will vary on size of history, to be added to returned surf

# header
img = interface.text(
    " DATE | STATE | MOVES | TIME | HINTS USED ",
    default_font,
    BLACK,
    default_colour,
)
img_height += img.get_height()
img_width = img.get_width()
img_list.append(img)

for i in range(len(self.history)):
    img = interface.text(
        self.history[i][0]
        + " | "
        + self.history[i][1]
        + " | "
        + self.history[i][2]
        + " | "
        + self.history[i][3]
        + " | "
        + self.history[i][4],
        default_font,
        BLACK,
        default_colour,
    )
    img_height += img.get_height()
    img_width = max(img_width, img.get_width())
    img_list.append(img)

```

```
    img_height += img.get_height()
    img_list.append(img)

surf = pygame.Surface((img_width, img_height))
surf.fill(default_colour)
for i in range(len(img_list)):
    surf.blit(img_list[i], (0, i * img_list[i].get_height()))

return surf
```

def update(self):

"""

Updates the history image and blits it to the screen

This takes into account the y_offset (amount scrolled) and adjusts it vertically

"""

```
    img = self.get_image()
    self.screen.blit(
        img, [self.pos[0] - img.get_width() // 2, self.pos[1] + self.y_offset]
    )
```

def scroll(self, amount):

"""

Changes the y position the image is blitted to,

which allows it to be scrolled

:param amount: The amount to scroll by, positive or negative

:type amount: int

"""

```
    self.y_offset += amount
```

```
class Leaderboard:  
    """This class manages creating and displaying the leaderboard"""
```

```
class Entry:  
    """  
    This class represents an entry in the leaderboard  
    """
```

```
This is deigned to be used by tools.File  
and as such all its attributes must also be parameters  
"""
```

```
def __init__(self, id, name, time, moves):  
    """  
    :param id: a unique identifier for each object  
    :type id: int  
    :param name: the username of the player  
    :type name: str  
    :param time: the time taken to solve the cube  
    :type time: float  
    :param moves: the number of moves the user did to solve the cube  
    :type moves: int  
    """
```

```
    self.id = id  
    self.name = name  
    self.time = time  
    self.moves = moves
```

```
leaderboard_file = tools.File("leaderboard.txt", Entry)
```

```
def __init__(self, screen, pos):
```

```
"""
:param screen: the screen that this is to be blitted to
:type screen: pygame.Surface
:param pos: the centre position that this is to be blitted to: x,y
:type pos: list[int] or tuple[int, int]
"""

self.screen = screen
self.pos = pos

self.entries = self.leaderboard_file.get_list()

"""The top ten quickest solve times, should be kept in in order
:type: list[Entry]"""

self.sort()
```

```
def update_list(self, time, moves):
```

```
"""
Checks if the user has a leaderboard worthy time and updates the ordered list
```

```
:param time: the time taken to solve the cube
:type time: float
:param moves: the number of moves the user did to solve the cube
:type moves: int
"""

if len(self.entries) < 10: # add new entry
```

```
    self.entries.append(
```

```
        Leaderboard.Entry(len(self.entries), ud.Manager.username, time, moves)
```

```
)
```

```
elif self.entries[-1].time <= time: # if no new entry, stop
```

```
    return
```

```
elif self.entries[-1].time > time: # if new entry replace slowest
```

```

        self.entries[-1] = Leaderboard.Entry(
            self.entries[-1].id, ud.Manager.username, time, moves
        )

    self.sort()
    self.leaderboard_file.replace_list(self.entries)
    self.leaderboard_file.save()

def sort(self):
    """Sorts the entries list by time"""
    self.entries.sort(key=lambda Entry: Entry.time)

def update(self):
    """Updates the leaderboard image and blits it to the screen"""
    img = self.get_image()
    self.screen.blit(img, img.get_rect(center=self.pos))

def get_image(self):
    """Creates the image of the leaderboard"""
    img_height = 0
    img_list = []

    # header
    img = interface.text(
        " POSITION | NAME | TIME | MOVES ",
        default_font,
        BLACK,
        default_colour,
    )
    img_height += img.get_height() # ensures the surd isn't too small
    img_width = img.get_width()

```

```

img_list.append(img)

for i in range(len(self.entries)):

    img = interface.text(
        str(i + 1)
        + " | "
        + self.entries[i].name
        + " | "
        + str(round(self.entries[i].time, 3))
        + " | "
        + str(self.entries[i].moves),
        default_font,
        BLACK,
        default_colour,
    )

    img_height += img.get_height()
    img_list.append(img)

surf = pygame.Surface((img_width, img_height))

surf.fill(default_colour)

for i in range(len(img_list)):

    surf.blit(img_list[i], (0, i * img_list[i].get_height()))

return surf

```

game_data.py

====

This file contains global data and settings information

This data is used by multiple files in the program. It may be edited here, or it may be provided to the user as settings for them to change.

black, isort and flake8 used for formatting

:::::

```
import copy
```

```
import pygame
```

```
from pygame import freetype
```

```
pygame.font.init()
```

```
pygame.freetype.init()
```

```
# colours
```

```
BLACK = [0, 0, 0]
```

```
WHITE = [255, 255, 255]
```

```
YELLOW = [255, 255, 0]
```

```
ORANGE = [255, 165, 0]
```

```
RED = [255, 0, 0]
```

```
GREEN = [0, 255, 0]
```

```
BLUE = [0, 0, 255]
```

```
GREY = [169, 169, 169]
```

```
default_colour = GREY
```

```
guide_arrow_colour = BLACK
```

```
# fonts
```

```
default_font = pygame.freetype.SysFont("calibri", 20)
```

```
guide_font = pygame.freetype.SysFont("calibri", 20, bold=True)
```

```
# cube design
```

```
# split into sides as easier to write
```

```
up = [
```

```
    [WHITE, WHITE, WHITE],
```

```
    [WHITE, WHITE, WHITE],
```

```
    [WHITE, WHITE, WHITE],
```

```
]
```

```
down = [
```

```
    [YELLOW, YELLOW, YELLOW],
```

```
    [YELLOW, YELLOW, YELLOW],
```

```
    [YELLOW, YELLOW, YELLOW],
```

```
]
```

```
left = [
```

```
    [ORANGE, ORANGE, ORANGE],
```

```
    [ORANGE, ORANGE, ORANGE],
```

```
    [ORANGE, ORANGE, ORANGE],
```

```
]
```

```
right = [
```

```
    [RED, RED, RED],
```

```
    [RED, RED, RED],
```

```
    [RED, RED, RED],
```

```
]
```

```
front = [
```

```
    [GREEN, GREEN, GREEN],
```

```
    [GREEN, GREEN, GREEN],
```

```
    [GREEN, GREEN, GREEN],
```

```
]
```

```
back = [
```

```

[BLUE, BLUE, BLUE],
[BLUE, BLUE, BLUE],
[BLUE, BLUE, BLUE],
]

# so a default cube may always be shown and to check against for solves
default_cube = [
    left,
    front,
    right,
    back,
    up,
    down,
]
# deepcopy passes by value, not reference, ensuring default_cube is not changed
used_cube = copy.deepcopy(default_cube)

# used for tracking moves and 'solving' the cube
class MoveStack:
    """A stack for managing the moves made by the user and scrambler"""

    def __init__(self):
        self.stack = []

    def push(self, move):
        """
        Pushes a move onto the stack

        :param move: move should be in the format
        {

```

```

    "direction": True for row, False for column,
    "number": row or column number,
    "backwards": If the move was backwards (left or down)
}

for a turn or the following for a rotation:

{
    "rotation": True,
    "direction": "x" or "y" or "z"
}

:type move: dict
"""

if move.keys() == {"direction", "number", "backwards"} or move.keys() == {
    "rotation",
    "direction",
}:
    self.stack.append(move)
else:
    raise ValueError("Invalid dict keys")

def pop(self):
"""

Pops a move off the stack

:return: move
:rtype: dict
"""

return self.stack.pop()

def clear(self):
"""Clears the stack"""

self.stack = []

```

```

def size(self):
    """
    :return: size of the stack
    :rtype: int
    """
    return len(self.stack)

def get_stack(self):
    """
    :return: the list of moves stored as dictionaries
    :rtype: list[dict]
    """
    return self.stack

def set_stack(self, stack):
    """
    Replaces the current stack with the one provided

    :param stack: the list of moves stored as dictionaries
    :type stack: list[dict]
    """
    self.stack = stack

moves = MoveStack()
"""
The MoveStack of moves that have been made by the user and the scrambler in order
:type: MoveStack"""
move_count = 0
"""
The amount of moves made by the user and scrambler.
These will be on order in the moves list, but will be preceded by scrambler moves

```

```

:type: int"""
scrambler_count = 0
"""The amount of moves made by the scrambler
:type: int"""

# used for tracking time
start_time = 0.0
"""The time since epoch that the user started the solve/ started the scrambler
:type: float"""

time_taken = 0.0
"""The amount of time that has elapsed since the user started the solve
:type: float"""

# used for seeing if the solve is eligible for the leaderboard and for users knowledge
hints_used = False
"""Whether the user has used hints
:type: bool"""

solver_used = False
"""Whether the user has used the solver
:type: bool"""

solved = False
"""Whether the cube is solved
:type: bool"""

```

interface.py

....

This file contains some key elements of the interface to be used by other files

This file handles creating visual elements and user interface

to be displayed to the screen for the user.

DisplayOption and DisplayBar should be used together.

black, isort and flake8 used for formatting

:::::

```
import pygame
```

```
class DisplayOption:
```

:::::

Creates a button with an image that changes size when hovered

This class should be used with DisplayBar

:::::

```
def __init__(self, image_function, display_surf, pos, size, mult, action, bg_col):
```

:::::

:param image_function: the function to get the image to use as the button

:param display_surf: the surface to display the button to

:param pos: the position to display the button from the top left

:param size: the x length and y length of the button

:param mult: how much to increase the image size when hovered

:param action: the function to run when the button is clicked

:param bg_col: the RGB value of the background colour

:type image_function: function

:type display_surf: pygame.Surface

:type pos: list[int] or tuple[int, int]

:type size: list[int]

:type mult: float

:type action: function

:type bg_col: tuple[int, int, int] or list[int]

```

"""
self.image_function = image_function
self.display_surf = display_surf
self.pos = pos
self.size = size
self.last_size = size
self.mult = mult
self.act = action
self.bg_col = bg_col
self.image = self.get_image()

self.last_size = self.size
"""The last x,y size of the button. Used for checking if the button is hovered
:type last_size: list[int]"""

def get_image(self):
"""
Gets the image of the button in its current state

:return: the image of the button
:rtype: pygame.Surface
"""

surf = pygame.Surface(self.size)
cube = self.image_function()
cube = pygame.transform.smoothscale(cube, self.size)
cube.set_colorkey(self.bg_col)
surf.blit(cube, (0, 0))
return surf

def update(self, mouse_pos, offset, mouse_up):
"""

```

Update the button, checking if it is hovered or clicked

```
:param mouse_pos: the x,y position of the mouse
:param offset: the width and height to offset the button ensures its enlarged
    size does not overlap anything
:param mouse_up: whether the mouse button has been clicked
:type mouse_pos: tuple[int, int] or list[int]
:type offset: list[int]
:type mouse_up: bool
:return: whether the button is hovered
:rtype: bool
"""

# calculate the position of the button with its offset
pos = [0, 0]
pos[0] = self.pos[0] + offset[0]
pos[1] = self.pos[1] + offset[1]

# calculate the centre of the button accounting for possible enlargement
# and offset
width = self.last_size[0]
height = self.last_size[1]
centre = width // 2 + pos[0], height // 2 + pos[1]

if self.image.get_rect(center=centre).collidepoint(mouse_pos): # if hovered
    if mouse_up: # if pressed
        self.act()
    # save same size so it can be restored
    temp = self.size.copy()
    # enlarge the button
    self.size[0], self.size[1] = (
        self.size[0] * self.mult,
```

```

        self.size[1] * self.mult,
    )

    self.last_size = self.size
    # get the enlarged image
    self.image = self.get_image()
    # restore size to the original state so it can be displayed
    self.size = temp

    self.display_surf.blit(self.image, pos)
    return True

else: # if not hovered
    self.image = self.get_image()
    self.last_size = self.size
    self.display_surf.blit(self.image, pos)
    return False

```

```

class DisplayBar:
    """For creating a bar of DisplayObject in a row/column"""

```

```

def __init__(self, object_list, row):
    """
    :param object_list: list of DisplayOption in sequential order
    :param row: if the buttons are in a row(True) or column(False)
    :type object_list: list[DisplayOption]
    :type row: bool
    """

    self.object_list = object_list
    self.row = row

```

```

def update(self, mouse_pos, mouse_up):

```

....

Updates each button in the bar and offsets then if one is hovered

```
:param mouse_pos: the x,y position of the mouse
:param mouse_up: whether the mouse button has been clicked
:type mouse_pos: tuple[int, int] or list[int]
:type mouse_up: bool
:rtype: None
.....
offset = [0, 0]
for i in range(len(self.object_list)):
    # update the button and check if it is hovered
    if self.object_list[i].update(mouse_pos, offset, mouse_up):
        if self.row:
            # set offset to the difference in size
            offset[0] = (
                self.object_list[i].last_size[0] - self.object_list[i].size[0]
            )
        else: # column
            offset[1] = (
                self.object_list[i].last_size[1] - self.object_list[i].size[1]
            )
```

def text(text, font, foreground_colour, background_colour):

....

Returns an image of the text

```
:param text: the text to display
:param font: the font to use
:param foreground_colour: the RGB value of the foreground colour
```

```
:param background_colour: the RGB value of the background colour
:type text: str
:type font: pygame.freetype.Font
:type foreground_colour: tuple[int, int, int] or list[int]
:type background_colour: tuple[int, int, int] or list[int]
:return: the image of the text
:rtype: pygame.Surface
"""

# render returns surface, rect so we only need surface
surface, _ = font.render(
    text=text, fgcolor=foreground_colour, bgcolor=background_colour
)
image = surface.convert_alpha() # optimisation
return image
```

[user_data.py](#)

```
"""
This file handles loading and saving user data
```

```
This file handles a user's game history, details about their current game.
as well as loading and saving data to a file
```

```
black, isort and flake8 used for formatting
```

```
"""
import game_data as gd
```

```
import tools
```

```
# game history
```

```
class History:
```

"""This class manages the game history of the user"""

```
def __init__(self):  
    # do not change these as they are used for saving  
    # they are directly aquired by self.__dict__ in the add method  
    # even changing thier order will break things  
    self.game_state = gd.used_cube  
  
    """The 3D array of the cube state at the last move  
    :type: list"""  
  
    self.move_count = gd.move_count  
  
    """The amount of moves made by the user. These will be on order in the moves list,  
    but will be preceded by scrambler moves  
    :type: int"""  
  
    self.moves = gd.moves.get_stack()  
  
    """The list of moves that have been made by the user and the scrambler in order  
    :type: list"""  
  
    self.scrambler_count = gd.scrambler_count  
  
    """The amount of moves made by the scrambler  
    :type: int"""  
  
    self.time_taken = gd.time_taken  
  
    """The amount of time that has elapsed since the user started the solve  
    :type: float"""  
  
    self.time_started = gd.start_time  
  
    """The time since epoch that the user started the solve/ started the scrambler  
    :type: float"""  
  
    self.solved = gd.solved  
  
    """Whether the cube is solved  
    :type: bool"""  
  
    self.hints_used = gd.hints_used  
  
    """Whether the user has used hints  
    :type: bool"""
```

```

self.solver_used = gd.solver_used
"""Whether the user has used the solver
:type: bool"""

self.history_list = []
"""The list of all history records
:type: list"""

def add_game(self):
    """Adds the current game to game history using the game_data"""
    self.game_state = gd.used_cube
    self.move_count = gd.move_count
    self.moves = gd.moves.get_stack()
    self.scrambler_count = gd.scrambler_count
    self.time_taken = gd.time_taken
    self.time_started = gd.start_time
    self.solved = gd.solved
    self.hints_used = gd.hints_used
    self.solver_used = gd.solver_used
    # add attributes to history list
    # excluding history list itself
    self.history_list.append(list(self.__dict__.values())[:-1])

```

```
def replace_history(self, history_list):
```

```
"""


```

Replaces the history list, useful for when initialising with user's saved data

:param history_list: the new history list

:type history_list: list

```
"""


```

self.history_list = history_list

```
def get_history(self):
    """
    :return: the game history list
    :rtype: list
    """
    return self.history_list
```

```
game_history = History()
"""The class containing the history of the user's game
:type: History"""
```

```
class User:
    """
    A class containing the user data and methods to update it
```

Designed for use with the Manager class and tools.File

```
def __init__(
    self,
    username=None,
    cube_state=gd.used_cube,
    start_time=gd.start_time,
    time_taken=gd.time_taken,
    moves=gd.moves.get_stack(),
    move_count=gd.move_count,
    scrambler_count=gd.scrambler_count,
    hints_used=gd.hints_used,
```

```

solver_used=gd.solver_used,
history=game_history.get_history(),
):

"""

:param username: the unique identifier of the user
:type username: str

:param cube_state: the 3D array of the cube
:type cube_state: list[list[list]]

:param start_time: the time since epoch when the user started the solve
:type start_time: float

:param time_taken: the time elapsed during the solve
:type time_taken: float

:param moves: the list of moves that have been made
:type moves: list[dict]

:param move_count: the amount of moves that has been made
:type move_count: int

:param scrambler_count: the amount of scrambler moves that have been made
:type scrambler_count: int

:param hints_used: whether the user has used hints
:type hints_used: bool

:param solver_used: whether the user has used the solver
:type solver_used: bool

:param history: the game history of the user
:type history: game_data.History

"""

# due to the way the user data is saved and loaded
# self. must match init param and username must be first

self.username = username
self(cube_state = cube_state
self.start_time = start_time
self.time_taken = time_taken

```

```

self.moves = moves
self.move_count = move_count
self.scrambler_count = scrambler_count
self.hints_used = hints_used
self.solver_used = solver_used
self.history = history

def save(self, username=None):
    """
    Updates this class's attributes to the current game data

    Optionally updates the username

    :param username: the unique identifier of the user, defaults to None (no change)
    :type username: str, optional
    """

    if username is not None:
        self.username = username
        self.cube_state = gd.used_cube
        self.start_time = gd.start_time
        self.time_taken = gd.time_taken
        self.moves = gd.moves.get_stack()
        self.move_count = gd.move_count
        self.scrambler_count = gd.scrambler_count
        self.hints_used = (gd.hints_used,)
        self.solver_used = gd.solver_used
        self.history = game_history.get_history()

    def load(self):
        """
        Updates the current game data to this class's attributes
        """
        gd.used_cube = self.cube_state

```

```
gd.start_time = self.start_time
gd.time_taken = self.time_taken
gd.moves.set_stack(self.moves)
gd.move_count = self.move_count
gd.scrambler_count = self.scrambler_count
gd.hints_used = (self.hints_used,)
gd.solver_used = self.solver_used
game_history.replace_history(self.history)
```

```
class Manager:
```

```
    """This class handles data in a txt file"""


```

```
    user_file = tools.File("saves_data.txt", User)
    username = None
    obj = None
```

```
@staticmethod
```

```
def load(username):
    """

```

```
    Load the user data for the given username, or create a new user
```

```
:param username: the unique username of the user
```

```
:type username: str
```

```
"""


```

```
    Manager.username = username
```

```
    try:
```

```
        Manager.obj = Manager.user_file.get_object(Manager.username)
```

```
    except tools.ObjectNotFound:
```

```
        Manager.obj = User(Manager.username)
```

```
        Manager.user_file.add_object(Manager.obj)
```

```
Manager.obj.load()

@staticmethod
def save(username=None):
    """
    Save the user data, and optionally change the username

    :param username: the new username, defaults to None
    :type username: str, optional
    """

    if username is not None:
        # replace the username
        Manager.obj = User(username)
        Manager.user_file.update_object(Manager.username, Manager.obj)
        Manager.username = username

    # save the data
    Manager.obj.save(Manager.username)
    Manager.user_file.update_object(Manager.username, Manager.obj)
    Manager.user_file.save()
```

tools.py

"""

This file contains useful tools for any program

As this file has been designed to work with any program all its functions are generic.

black, isort and flake8 used for formatting

"""

```
from os.path import isfile

class ObjectNotFound(Exception):
    """Indicates that an object was not found when searched for within a file"""

    def __init__(self, identifier, file):
        """
        :param identifier: the identifier of the object that was not found
        :type identifier: any
        :param file: the file that was searched
        :type file: str
        """

        super().__init__(f"Object not found | Identifier: {identifier} | File: {file}")
```

class File:

....

This class manages a list of objects in a file

The list should be updated using the get_list and update_list functions.

It should be saved with the save function.

It contains the class objects.

Individual objects can be got with the get_object function.

Objects can be replaced with the update_object function.

Objects can be removed with the remove_object function.

Either the entire list should be modified or only single objects should be modified.

These should not be used together.

....

```

def __init__(self, file, cls):
    """
    :param file: the name of the file to store the data in, must be .txt
    :type file: str
    :param cls: the class of the data stored in the file, not an object
        cls(arg0, arg1, etc.) must call the constructor
        arg0 must be a unique identifier.
        the attributes self.'s must be the exact same as the parameters
    :type cls: class
    """

    self.name = file
    self.cls = cls

    self.list = []
    """The list of all data in the file
    :type list: list[object]"""

# check file exists, create if it doesn't
if not isfile(self.name):
    f = open(self.name, "w")
    f.close()
    self.read()

@staticmethod
def get_identifier(obj):
    """
    Returns the first key in the object's dictionary as a string
    """

The first key is considered the identifier, it is converted to a string to
ensure there are no errors during comparison

```

```

:param obj: the object to get the identifier of
:type obj: object

:return: the identifier of the object
:rtype: str
"""

keys = list(obj.__dict__.keys())
identifier = obj.__dict__[keys[0]]
return str(identifier)

def read(self):
"""

Reads the file and updates self.list

:rtype: None
"""

f = open(self.name, "r")
file_str = f.read()
f.close()

# check file isn't empty
if file_str == "":
    return

objects = file_str.split("\n")
objects.pop() # get rid of newline at end of file
for obj in objects:
    self.list.append(
        self.cls(**eval(obj)) # convert string to dict and pass as kwargs
    )

```

```
def sort(self):
    """
    Sorts the list

    :rtype: None
    """

    self.list.sort(key=lambda obj: self.get_identifier(obj))

def search(self, target):
    """
    Finds the position of the target in the list, automatically orders the list

    :param target: the target to search for
    :type target: object
    :return: the position of the target, -1 if not found
    :rtype: int
    """

def binary_search(lst, start_pos=0):
    """
    Recursive binary search using the identifier

    :param lst: the list section to search
    :type lst: list
    :param start_pos: the start position of the list section
    :type start_pos: int
    :return: the position of the target, -1 if not found
    :rtype: int
    """

    # empty list
```

```

if len(lst) == 0:
    return -1

mid = len(lst) // 2
identifier = self.get_identifier(lst[mid])

if identifier == target:
    return mid + start_pos
elif identifier > target:
    return binary_search(lst[:mid], start_pos)
else:
    return binary_search(lst[mid + 1 :], start_pos + mid + 1)

self.sort() # order the list
target = str(target) # for comparison to prevent errors
return binary_search(self.list)

def get_list(self):
    """Returns the list of objects"""
    return self.list

def replace_list(self, lst):
    """
    Replaces the list of objects

    :param lst: the new list
    :type lst: list
    :rtype: None
    """

    self.list = lst

```

```

def save(self):
    """Sorts self.list then replaces the file with it."""
    self.sort()
    f = open(self.name, "w")
    for obj in self.list:
        f.write(str(obj.__dict__) + "\n")
    f.close()

def get_object(self, identifier):
    """
    Gets the object with the given identifier

    :param identifier: a unique identifier
    :type identifier: any
    :return: the object or raises exception ObjectNotFound if the object is not found
    :rtype: object
    """

    pos = self.search(identifier)
    if pos == -1:
        raise ObjectNotFound(identifier, self.name)
    return self.list[pos]

def add_object(self, obj):
    """
    Adds the object to the list

    :param obj: the object to add
    :type obj: object
    :rtype: None
    """

    self.list.append(obj)

```

```

self.sort()

def update_object(self, identifier, obj):
    """
    Replaces the object with identifier with the given object.

    :param identifier: the identifier of the object to replace
    :type identifier: any

    :param obj: the new object to replace the old one
    :type obj: object

    :return: None or raises exception ObjectNotFound if the object is not found
    :rtype: None
    """

    pos = self.search(identifier)
    if pos == -1:
        raise ObjectNotFound(identifier, self.name)
    self.list[pos] = obj


def remove_object(self, identifier):
    """
    Removes the object with the given identifier

    :param identifier: the identifier of the object to remove
    :type identifier: any

    :return: None or raises exception ObjectNotFound if the object is not found
    :rtype: None
    """

    pos = self.search(identifier)
    if pos == -1:
        raise ObjectNotFound(identifier, self.name)
    self.list.pop(pos)

```

```
# testing

if __name__ == "__main__":

    class Test:

        def __init__(self, arg0=None, arg1=None, arg2=None):
            self.arg0 = arg0
            self.arg1 = arg1
            self.arg2 = arg2

        def output(self):
            return str(self.arg0) + " " + str(self.arg1) + " " + str(self.arg2)

    file = File("test.txt", Test)
    file.sort()
    lst = file.get_list()
    # lst.append(Test({'a': b}, 3, 5))
    file.update_list(lst)
    file.save()
    print(lst)

    for cls in file.get_list():
        print(cls.output())
    pos = file.search("{")
    print(pos)
```