Heman Seegolam
Candidate Number: 4142

VISUALEARNING

Quiz Program

Centre Number: 16605
Computer Science NEA

3.1 Analysis of the Problem

3.1.1 Problem Identification:

My client, Mrs Parker, is a secondary school teacher who has trouble keeping up with the increasing workload that her job burdens upon her. This includes not only her working hours, but also during the time she could be spending looking after her children and spending time with her family as well as plan effective lessons for her students. As a maths teacher, she is trying to put together lessons which challenge her students no matter which ability group they may fall in and would like a quick way to monitor the progress of her pupils without having to mark pages of homework in her free time or inconveniently plan and carry out tests, in order to then maximise the time available to increase the effectiveness of her lessons. A program that can track progress in real time by answering quiz questions is already in the market, but, in order to maximise functionality for Mrs Parker, she requires a program that is able to successfully track the progress of answered questions by pupils, as well as store the progress of the students and present them in a way that allows her to easily monitor improvement and ability of students, over various quizzes and over a period of time. This display of the students' progress could come in different formats in order to maximise ease of reading for Mrs Parker, i.e. in tabular format or in graphical format.

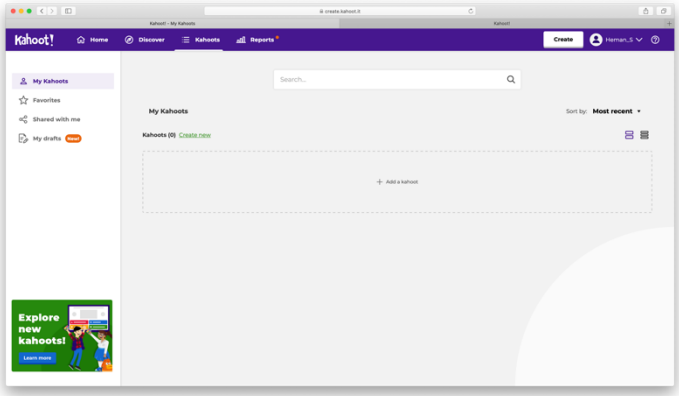
A potential alternative for tracking the progress of her students and easing the burden upon her would be to hire a teaching assistant that could help students in lower ability groups or help to do some of the marking, which could also be beneficial for the students if they need any questions answered or help in any other way. However, the problem with this is that it is may not be within the school's budget, much less her choice. Furthermore, Mrs Parker would like her colleagues to also be able to access the tool and overlook the child's total progress, perhaps when it comes to writing a report for the child and sending off grades to parents, or when the students progress through the years and their new teacher(s) would like a sort of progress report.

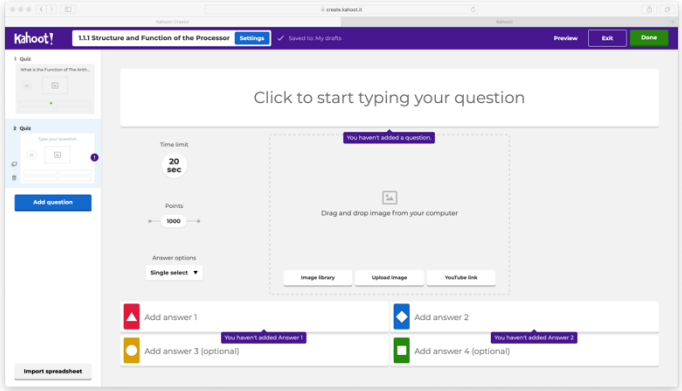
The marking of questions would be done automatically after a student's answer has come through, relieving my client of her busy workload, and also the storing of the data in which Mrs Parker and her colleagues are able to easily access. In addition, the progress of students may be a sensitive concern for some, and it is therefore important for it not to be openly available for other students to look at, as they may if their progress was marked on paper which lay openly on Mrs Parker's desk, or through test papers which have been marked and returned. Consequently, the progress of the students should be locked from access of students, but easily accessible to Mrs Parker and her colleagues.

3.1.2 Stakeholders and 3.1.3 Research:

As stated previously, there are various systems which allow children / students to answer and mark questions and track progress in real time.

Firstly, I investigated **Kahoot**. Kahoot has acquired over 1 billion cumulative users and have recorded and average of 50 million monthly users with an estimated worth of 300 million dollars which is why researching and analysing this application would prove to be useful for my project as I could acquire useful ideas to incorporate within it. In this system, a host and participants are required. From the host's perspective (kahoot.com), there is facility to create personalised multiple-choice questions, with personalised answers. From the perspective of participants (kahoot.it), there is a user-friendly platform which easily allows them to answer the questions and let them compete real-time against fellow students.





Here, hosts may create an account in order to create different personalised quizzes, with variable time limits, points and answer options. They may also add an image or YouTube link. They also have the ability to assign favourite quizzes under a separate page as well as create drafts of new quizzes and have quizzes shared with them.

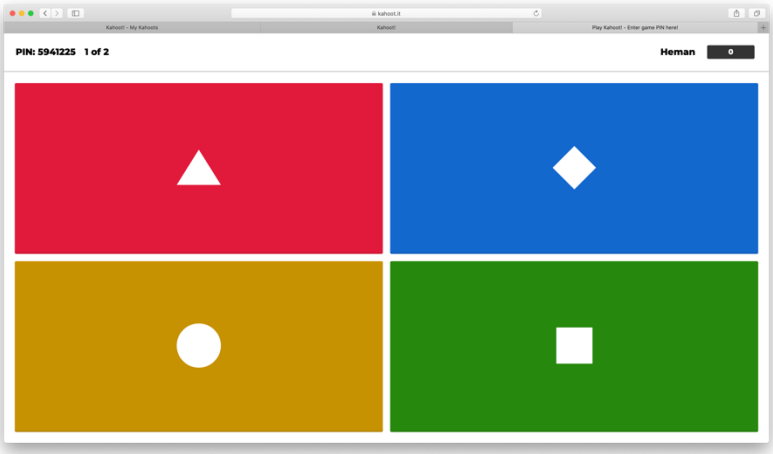
The facilities included in this website are satisfactory and exceedingly fit for purpose. For example, some aspects I may be able to use in my system include:

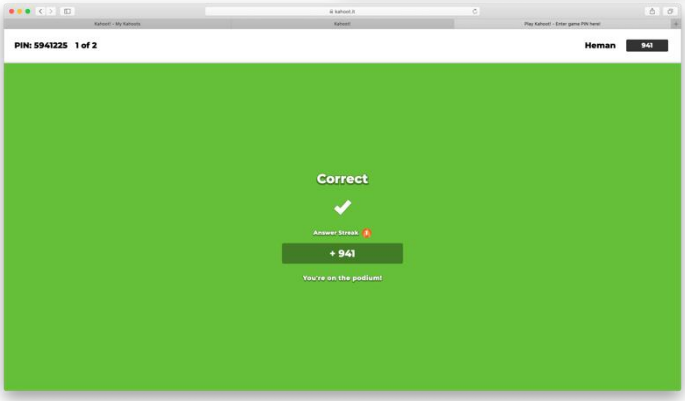
- a host login – to allow for the creation of (multiple) personalised quizzes
- a facility to share quizzes with other people – to allow colleagues to share quizzes
- a user-friendly platform to easily facilitate the creation of new questions within a quiz – the ability to set a question with corresponding multiple-choice answers and time limit, as well as points and perhaps and image.

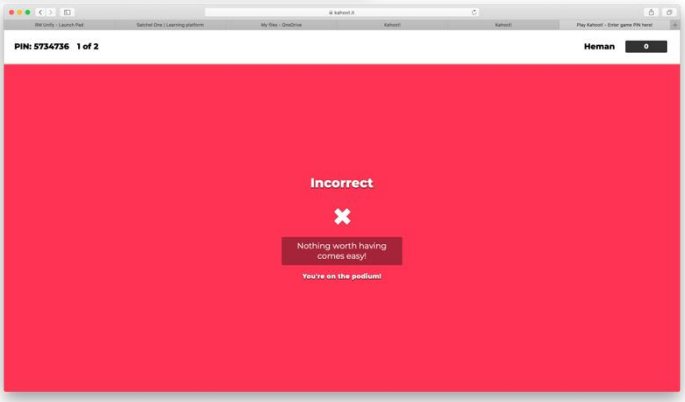
This is made amenable through computational methods such as decomposition and abstraction. Decomposition is the process by which a complex problem or system is broken down into parts that are easier to conceive, understand, program and maintain. Logins, as simple as they may seem, can be broken down into two main parts: logging in and signing up. When logging in, the program is required to match the user's input to the items in an array or database in or to find a match. It would then have to check a username has the correct corresponding password to allow them access to the program. If incorrect, it would have to allow them to try again up to a certain amount of times to prevent situations such as a brute force attack on the system. To sign up, details inputted by the user would have to be checked against given conditions before then being added to an array or database to allow them future access. Their account could also have to be verified by an admin user for security purposes. The use of decomposition creates a more efficient programming method, improving time and reducing the complexity of a once multifaceted problem.

Abstraction is the process of removing unnecessary details or attributes in a system to focus attention on details of greater importance, i.e. key elements of the program. This will be used when

creating an interface for the teachers so they can use what they need to use without concerning themselves with the intricacy of the actual program. This will also be used for students, where the teacher, already having made a test in advance, takes part in a quiz, not concerning themselves with how the test was made. By using abstraction, stakeholders are able to focus purely on the key elements of the programming, thus significantly reducing the complexity of the program for them. In addition, the source code of my program shall be hidden from users, restricting them from making any changes that may change the program.



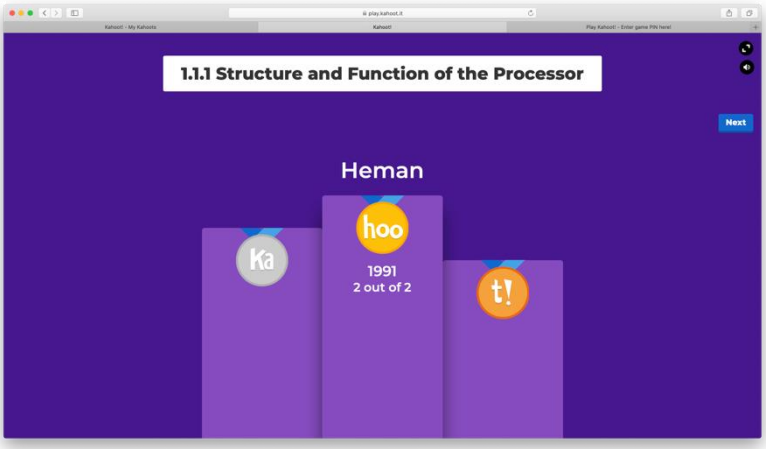




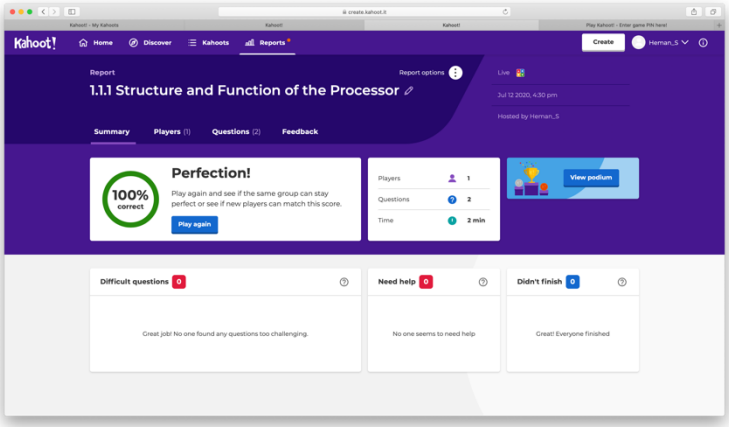
This is what the participants taking the quiz on Kahoot would see. It has an aesthetically pleasing interface to look at, utilising colours and shapes to capture the attention of a target audience with varying ages. This web-based application is was made to make a quiz system that more interactive than its predecessors by making players look up at a board that will have questions and answers projected onto them before turning back to their respective devices to enter one of the multiple-choice answers of their choosing by pressing the shape that corresponds with their answer. When everyone has then picked their answer, a new page comes up showing whether the chosen answer was correct, incorrect and how many points they receive. It also shows the question number they are currently on, out of how many there are as well as their name and the total accumulated number of points over the quiz. Furthermore, if incorrect, a little motivational message appears to help inspire the students; if correct, an answer streak appears, showing how many correct answers have been answered in a row. It is very popular among young students as it successfully engages players. It is definitely a popular application for teachers as it allows them to occasionally test students while retaining an enjoyable and interactive classroom environment and receive useful input onto the progress of their students.

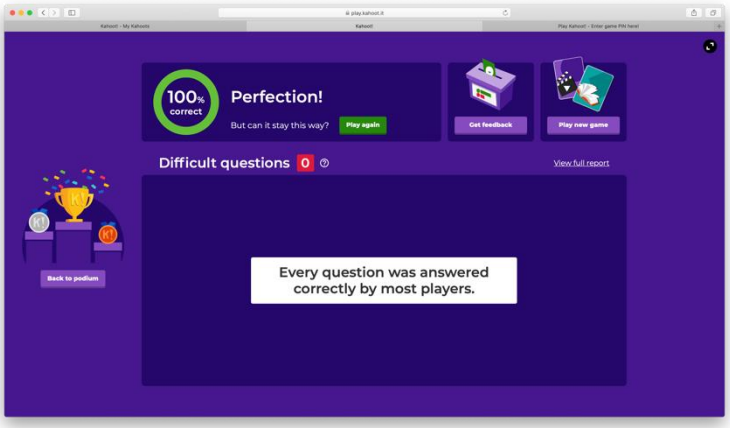
Some aspects I may be able to include are:

- Separate pages to indicate whether a student's answer is correct or incorrect
- A facility to assign and accumulate points over the duration of the quiz
- The ability to register answer streaks
- Little motivational messages if answers are incorrect



To conclude the quiz, a podium is shown of the top three participants and their corresponding points. This can act as a great incentive for students to perform as best they can during the test and may come up in a similar form in my software.





When finished taking a quiz, the host is provided with some data from the quiz. There is a summary page which indicates the number of players, the number of questions, the time taken, but also the accuracy of the answers and the questions that participants found the most difficult as well as participants that may need help or who didn't finish particular questions. This is extremely useful information for teachers since they are able to target less able students and identify particularly where the issue lies for the class or for individual students.

Some aspects I may be able to include are:

- The accuracy of the class as a whole
- Identification of specific questions that were particularly difficult
- Identification of students or topics that may need help

The identification of specific questions, topics or students that may require assistance is amenable to the use of computational methods, namely data mining. Data mining is the process of looking for general trends in a large set of data. In this case, once a quiz has been completed, data mining can be used to analyse the progress of the students as well as individual questions or students to

determine where they struggled. This would be useful in isolating particular difficult questions or topics or determine which students require additional assistance. With this information the teacher is in a better place to aid them with a more focused scope of attention. However, data mining would require fast processing and big databases (large amounts of storage) to work.

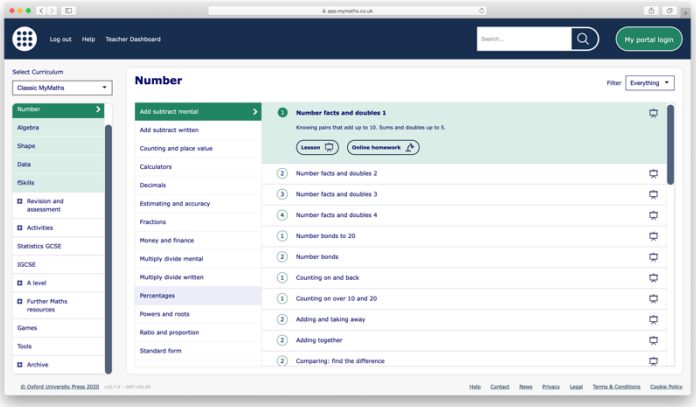
Kahoot fulfils its role as a fun educational quiz game as well as a useful tool for teachers, however, there are certain functionalities that can be extended to maximise its use to my client, which include:

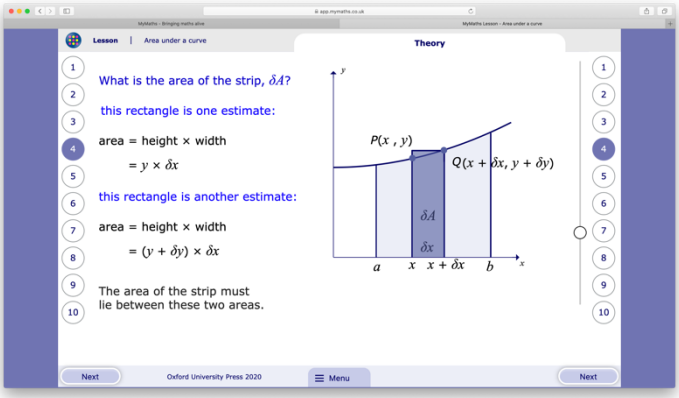
- an ability to store progress within classes of the host's login
- the accuracy of individual students after completing a quiz
- an ability to represent stored progress in graphical or tabular format

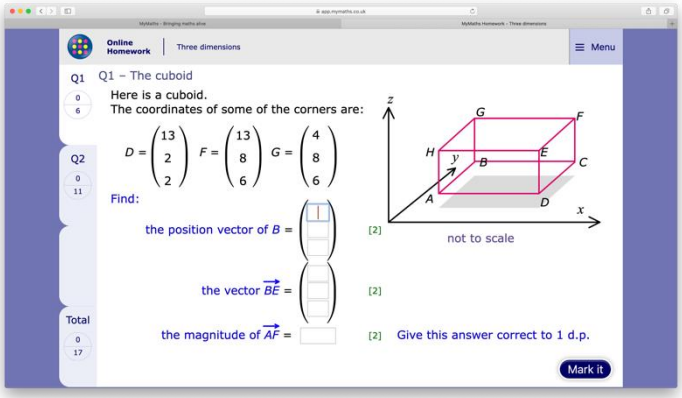
Here, I can use visualisation. Visualisation is the process of turning data in a visual representation which is easier for humans to understand and work with. Firstly, this can be done by creating a very engaging interface for the relevant stakeholders, providing a more interesting platform in which students can improve their educational abilities as well as providing an interface for teachers in which they can easily track progress of their students. As an extension into the use of visualisation, and into the presentation of student progress, by transforming the data into a graphical or tabular format, it will become so much easier to interpret the relevant data and patterns,

Outside of its current features, I can perhaps make a more engaging quiz system by the concept of incorporating more game-like elements such character creation and customization and roaming (e.g. RPG-based games). Therefore, I would like to include most functionalities of Kahoot but create a quiz system that will be popular amongst audiences of younger ages as well. I feel that an application with elements such as those aforementioned would be more popular than Kahoot in that regard.

The next system I investigated was **MyMaths** (mymaths.co.uk). This system works primarily on improving mathematical skill, from levels ranging from primary school to A-level. To access the MyMaths software, you must have an account which works as an annual subscription to the site which starts at £339 + VAT for primary schools, and £625 + VAT for secondary schools. However, they also provide free trials for 30 days to provide schools the satisfaction needed before paying for the service. As a popular and proven to work service, the site is entitled to price their service at such a price, although as it provides for an entire school, it equates to an affordable per person charge. As a new software, the prices for my system may initially be lower than that of MyMaths in order to incentivise the market to purchase it, however, a free trial system is a great way to allow potential users to try the system to see if it matches their requirements.



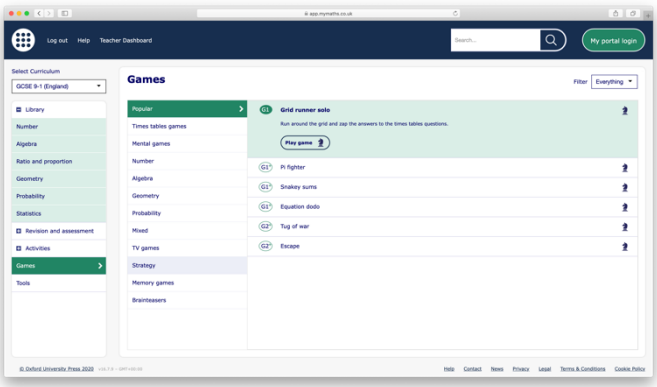


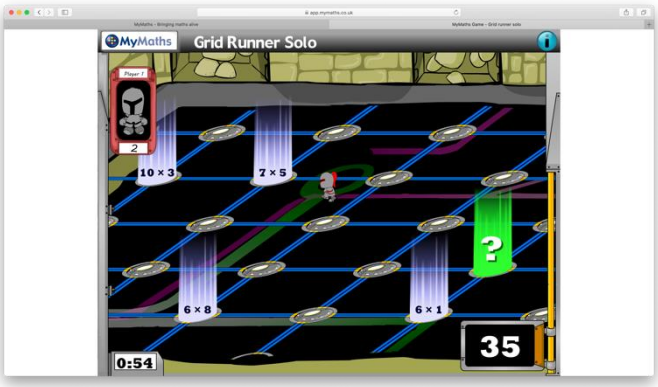


When logged in to the service, you are brought to a menu which has all options of topics, lessons and online homework / practice. The menu is fairly easy to follow and can be filtered to just the material you would like to use. The filters include the level of education, and the topics that are assigned to each grade. When a topic is chosen, the user is given the option to take a lesson on the topic, or alternatively do an online homework which teachers may set. The marks from the online homework are recorded and are seen by the teachers, which allow them to firstly see that it has been completed shown by a tick (or an empty space), and secondly track how well they have understood the given topic, shown by the mark received. They are able to redo the homework as many times as they would like or just simply practice a topic without having being set it specifically. In the lesson, there are images and animations which aid in the understanding of the content, as well as brief, written notes.

Although the facilities included in this website are primarily aimed to improve mathematical ability, they are extremely apt in doing so, in a way that is slightly different to the layout I wish to obtain for my system. However, there are various aspects I may be able to use in my system, such as:

- a menu that can be filtered – to organise quizzes, classes, students, etc.
- an ability for teachers to see whether students have answered questions and whether correct or not
- from inspiration of the lessons, I can implement a section if a student is incorrect when answering a question which shows them how and why they are incorrect, as well as an explanation of the correct answer



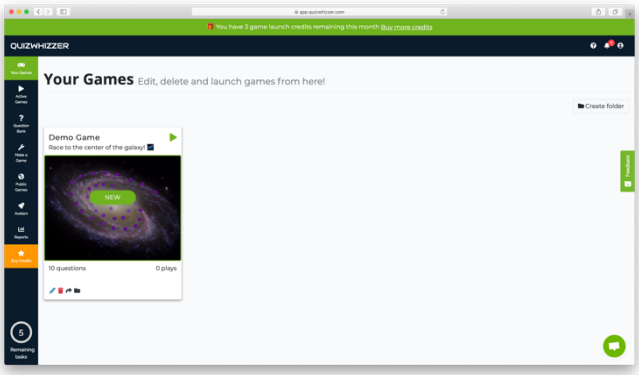


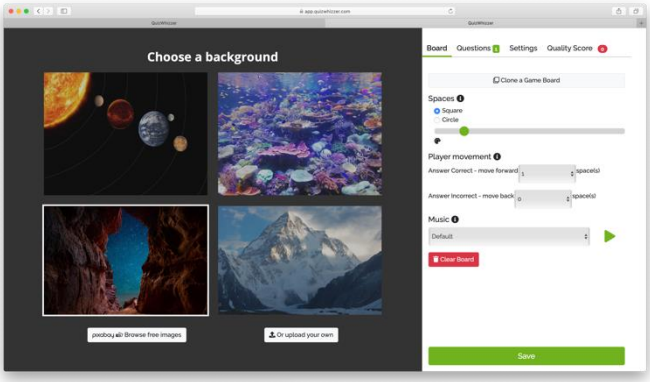
In addition to the great selection of lessons and online practice, MyMaths also provides games for children to play which can help improve their mathematical skill through a more interactive platform. There is a separate menu for the vast range of games included in the service. When starting a game, the user is presented with a leader-board taken from all schools subscribed to the service, and works as a good incentive for some kids to perform to the best of their ability during the game, whilst also including the fun aspect associated with games and vivid, visual images and animations. Although perhaps a great tool for children to progress through their early learning, it may seem inappropriate for my client who has an audience of adolescents. However, it may be applicable for those who may have intellectual disabilities or disorders such as the 'Peter Pan Syndrome', or instead a facility that can be exploited by primary school teachers if they wish to use the software.

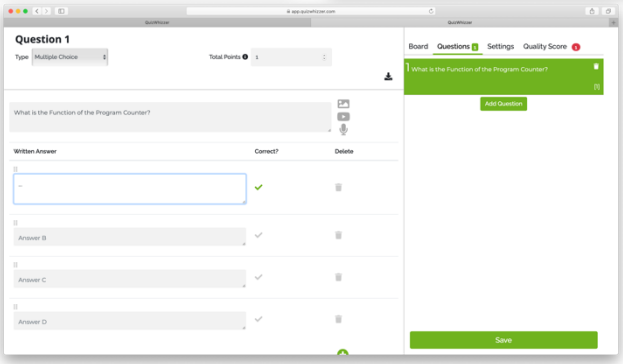
Generally, I found the platform and the layout of MyMaths extremely easy to use, and visually appealing which created a higher incentive for me to explore and use the service. I shall try to recreate a system which is not necessarily the same layout as MyMaths, but one which is very easy and user-friendly to use as well as aesthetically pleasing for the user. However, some aspects I may add to my system include:

- a wider range of subjects
- a facility to create personalised quizzes – perhaps I could include exemplar quizzes for core subjects
- a facility for students to complete the task in real-time (so my client would not have to spend her time chasing her students to complete the task, but simply devote a small amount of time doing the quiz with them, after having created or using one of the quizzes provided)
- logins for students and teachers as to allow teachers to track progress of all students within his or her class, and to allow students to track their own progress over individual quizzes and overall (perhaps by subject)
- a game aspect to better engage students with the quiz – ones, however, that are more appropriate for the audience of the level of the quiz

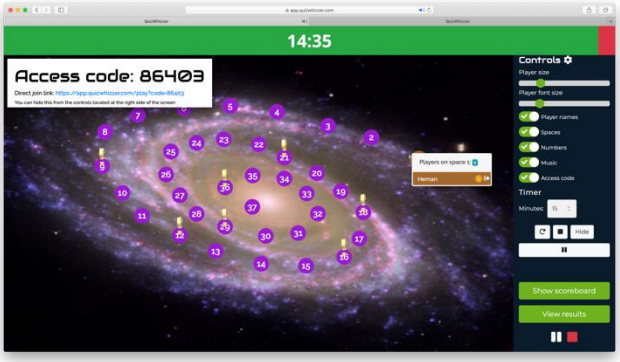
Finally, I investigated **QuizWhizzer** (quizwhizzer.com). Similar to Kahoot, QuizWhizzer is a platform that allows students to compete real-time with questions created and set by a teacher. However, it differs in the way the quiz is taken.



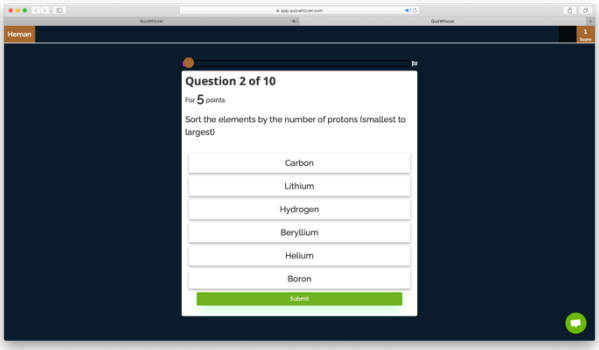


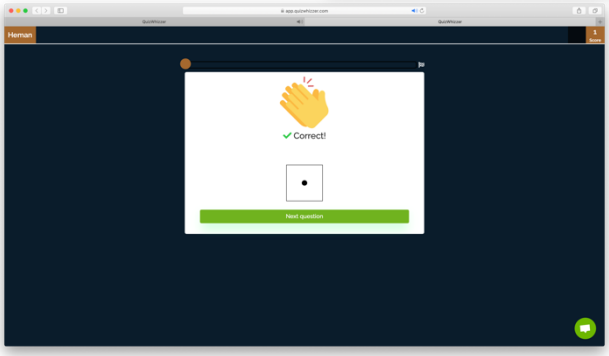


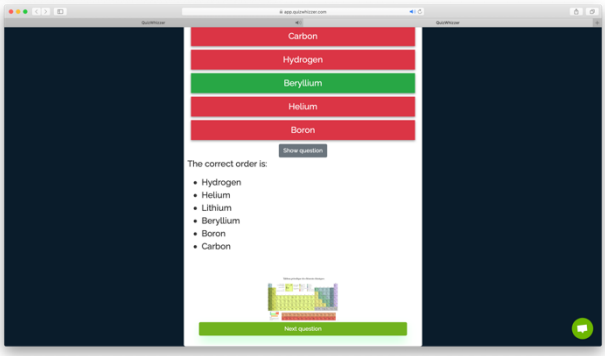
In QuizWhizzer, by creating questions and answers, teachers essentially create an online board game by uploading a background, adding spaces for where the players will move and choosing how much players move by. If players answer correctly, they are then given an online die to roll which indicates how many spaces they move forward. If incorrect, the game designer can choose if and by how many spaces to move back. In comparison to Kahoot, this system is more interactive, and perhaps more engaging for players, since they have a secondary objective of trying to take the lead in the game. However, the number from the die is not a reflection of the player's ability and could cause unintentional bias and an inaccurate data of progress.



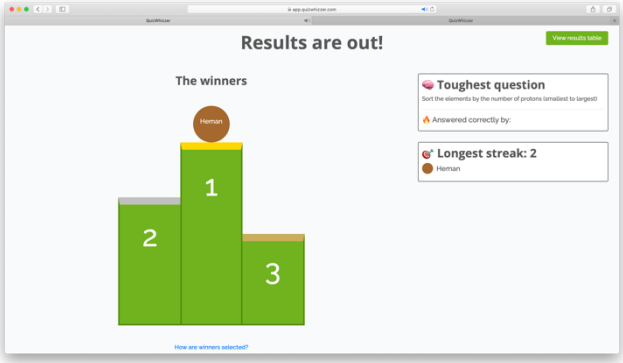
When the host starts a game, an access code appears on their screen, which allows players to join from their own device. A custom game board is shown and each player stands on their corresponding place on the board.

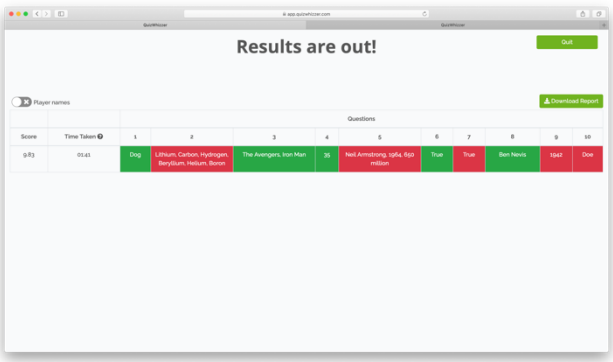






The players are presented with an easy to use platform which specifies the question, answer options and number of points available for each question. Unlike Kahoot, this system allows more flexibility with how questions are answered, for example through multiple choice options, boxes to input answers or sliders which can be used to position given answers in a correct order.





At the end of the quiz game, a podium is shown for the players with the highest points, as well as the toughest questions and the longest answer streaks achieved. The host is presented the questions answered by each student in a tabular format which is easy to understand and coloured in to provide an easier read. It also shows their total amount of points as well as the time taken to complete the quiz. There is also an option to download the report as an external file.

Although this system has great similarities to Kahoot, there are some prominent features that set it apart and some which I may be able to use in my system. Some features I may be able to incorporate are:

- a more engaging look to the quiz using visualisation– perhaps an option to upload their own image (whether for the whole quiz or for individual questions)
- a variety of question types, extending from just the typical multiple choice questions

- an ability to represent quiz data in a tabular format as well as an ability to create an external file containing the results.

To extend these features I may also be able to:

- represent data from the quiz in other forms such as in graphs or charts, etc.
- provide a login for students to track their own progress, and take practice quizzes of their own

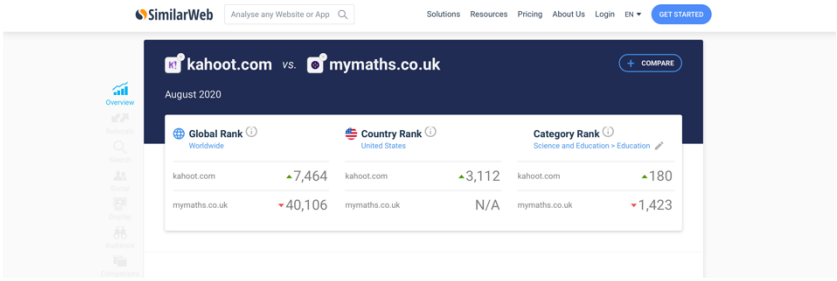
Research Conclusion:

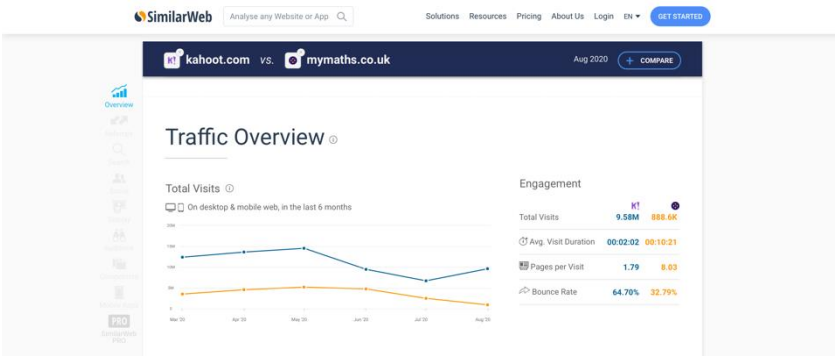
After researching three systems that are quite similar to the concept of how the program may function, I have kept certain things in mind. A multiplayer aspect of some sort will definitely boost the overall pull of the quiz application as being able to compete will real-life people tend to not only sustain a user base but also bring in new players. Therefore, I should look to make the program include as many social features as I can (might look into a group chat with censorship).

Another point is that there should be user access levels which more importantly allow administrators to manage login details, private information (e.g. students performance data) and so on. These admin accounts should be able to implement new quizzes so that students can always have a new quiz to play when they are already completed with the current quizzes.

There are some very noticeable differences between the three existing quiz web applications I have researched on top of the similarities that I have mentioned before. These differences are mainly oriented around playstyles and such. Kahoot and QuizWhizzer emanates a much more competitive and intense vibe in comparison to MyMaths' more passive style. Therefore, there are definitely preferences among players.

By exploring the SimilarWeb website (https://www.similarweb.com/website/) is ascertained the statistics provided below; comprehensive analysis of web apps are provided on this website.





From these screenshots, comparing Kahoot and MyMaths, on a global scale Kahoot Is ranked higher than MyMaths in terms of usage. Kahoot is also ranked higher in its specific field (science and education). Competitors for both of these quiz applications seem to be very different as the way they deliver their quizzes are fundamentally different meaning they are distinctly categorically distinguishable as is specified on the reference website I used.

Overall, Kahoot not only has a greater growth rate than MyMaths but also seems to be categorically linked to how the application that I will develop will most likely be. The statistics provided conveyed how Kahoot seems to be more popular in the present as it has grown faster and evidently seems more popular. Therefore, there is great potential in taking a similar route to Kahoot and making alterations along the way. There was no available data for QuizWhizzer (due to lack of data)

To tailor my program to my client, I found it beneficial to ask her several questions which may help to suit the system to her needs. To some replies in which further clarification was required, I followed up the initial questions with ones that allow me to pinpoint issues she may currently be facing, and find ways to discard them. These concern crucial features, aesthetics and privacy issues that may arise.

What is the current method of approach to the described problem, i.e. of test-taking and recording of progress for students?

Currently, the students are required to do physical examinations within the classroom, sometimes of which I have to write myself. As for tracking their progress – every time I am required to decide a grade to report back for my students, I have to shuffle through paperwork regarding all the test done and then confirm a final grade based on them.

Why not record grades on a digital platform as opposed to with paper and pen?

I simply can't use a computer! Perhaps clicking with the mouse and using the keyboard but applications like Excel? Not a chance!

What do you like about recording the grades on paper?

Simple to read; simple to record. It'd be great if it wasn't just boring tables and stacks of pages though!

Is there anything you would change to it?

Yeah, if possible find a way to record all the grades in one place. As I said before as well, tables can get a bit tedious to look at and can be very easy to misread. Also, it becomes harder to track actual progress of students over time without having to look at each page, each grade, for each student each time.

What operating systems do you currently use currently on your computer?

My school laptop is using Windows 7, whereas my laptop at home uses Windows 10.

What key features would you want the proposed system to have?

Ideally, the system would allow me to create a quick, efficient quiz quickly and efficiently; it would quickly mark answers and provide me with an overall test score all in one place in a way I can easily understand. Perhaps it could have this data represented in charts and graphs too to make it so much easier to analyse progress, so I can quickly glance at them and have a general idea of the progress of the student, and also quickly write my reports! Not necessarily a key point, but it'd be great if the quiz was perhaps more engaging than its alternatives which I always find students quickly get bored of and just mess around.

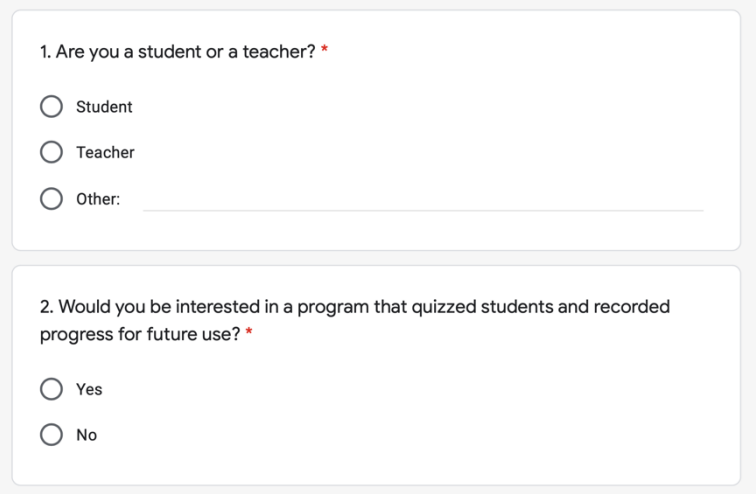
Are there any secondary features you may want?

So apart from being generally more engaging for students, I need everything to be self-explanatory – available to use to a monkey who could read! I'd prefer it if I could access the data at home, and a facility that allows it to be secure from other students for example. It could be beneficial if students had access to their own progress for their own reference or for their parents. Also, it would be great to be able to share the progress with other colleagues who may need to refer to the data.

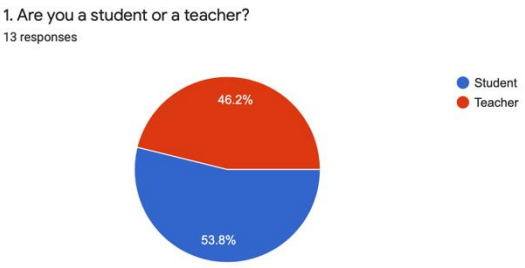
Would you prefer a simple, modern interface or a very detailed, less visually appealing one?

For me, someone who gets confused as soon as the computer shows me options that I have no idea what mean, it would be best to have a simple, modern interface that although incorporates a lot of detail, in terms of what each button does and where everything is, shows me the most important information in a format that is easy to read and follow. My main concern when it comes to the interface, is how the progress of the students is shown. For example, I don't want to be bombarded with loads of meaningless numbers on a page!

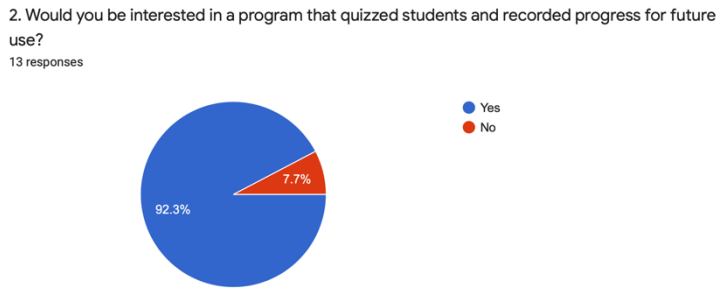
From this response from my client, I am able to guide the design of my program to fit the criteria of which she requires. However, it is also beneficial for me to target a wider audience by identifying my stakeholders, and also found out various opinions from other people. Stakeholders for my project include teachers – who the program is designed for to aid them in efficiently providing a platform in which tests their students, but also easily track their progress – and also students – who will also be using the program to take part in quizzes and check their progress, etc. To take in the opinion of my stakeholders, I found it best to create a survey for various people to complete using Google Forms.



This first question has the aim of identifying the type of stakeholder that answers the questions. From this information, I can easily see what common answers correspond with which type of stakeholder: student or teacher, and make decisions accordingly when designing the interface for each stakeholder. The results for this question is roughly half for each. The second question has the intention of identifying which of the people taking the survey would be interested in the proposed idea. From this question, we can see that the response is very high in favour of the program which makes it a good idea and a high demand program to make.



The third question (next page) determines in which way(s) I should make the data of the students representable. Over a third of people were in favour of presenting data in a more visual way, in graphs or charts as an extension to just a table. It may be useful to include these features in my program. The aim of the fourth question was to determine whether presenting data in an external file would be helpful for my stakeholders. Just over half of the responses came back negative, therefore it may be useful to add it in as an extra feature if needed.



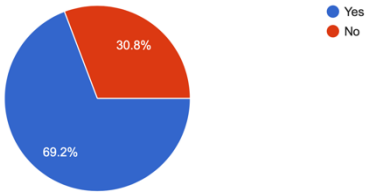
3. Would a graph or chart be more beneficial than just presenting progress of taken quizzes than just a table? *

- Yes
- No

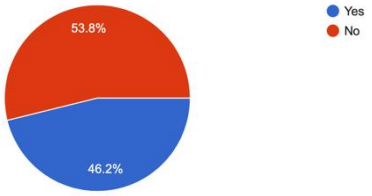
4. Would storing data in an external file be an efficient way of presenting your data? *

- Yes
- No

3. Would a graph or chart be more beneficial than just presenting progress of taken quizzes than just a table?
13 responses



4. Would storing data in an external file be an efficient way of presenting your data?
13 responses



5. As a teacher or student, would you find it useful to be able to share your progress with other users? *

- Yes
- No

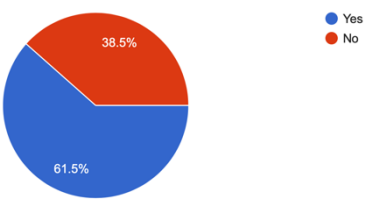
6. What would be the most important aspects of a quiz program that recorded progress *

- [ ] An ability to create personalised quizzes
- [ ] An ability to access previously created quizzes
- [ ] An ability to represent data in graphs or charts
- [ ] An ability to store data in an external file
- [ ] An ability to share data with others
- [ ] Clear and easy-to-use buttons and instructions on how to guide yourself around the program
- [ ] Other:

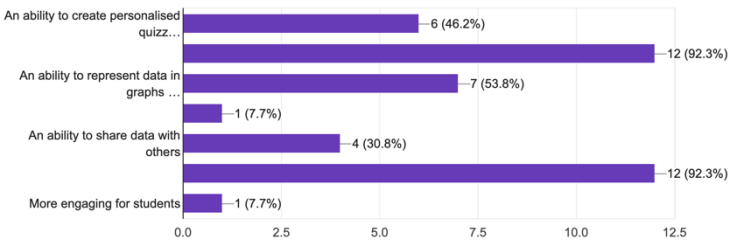
The main aim of question 5 is to determine whether stakeholders would like to be able to share their progress or the progress of their students to other users. This can be for students to share with their parents or for teachers to share with their colleagues. The majority of responses came back positive, making it a worthwhile feature to include.

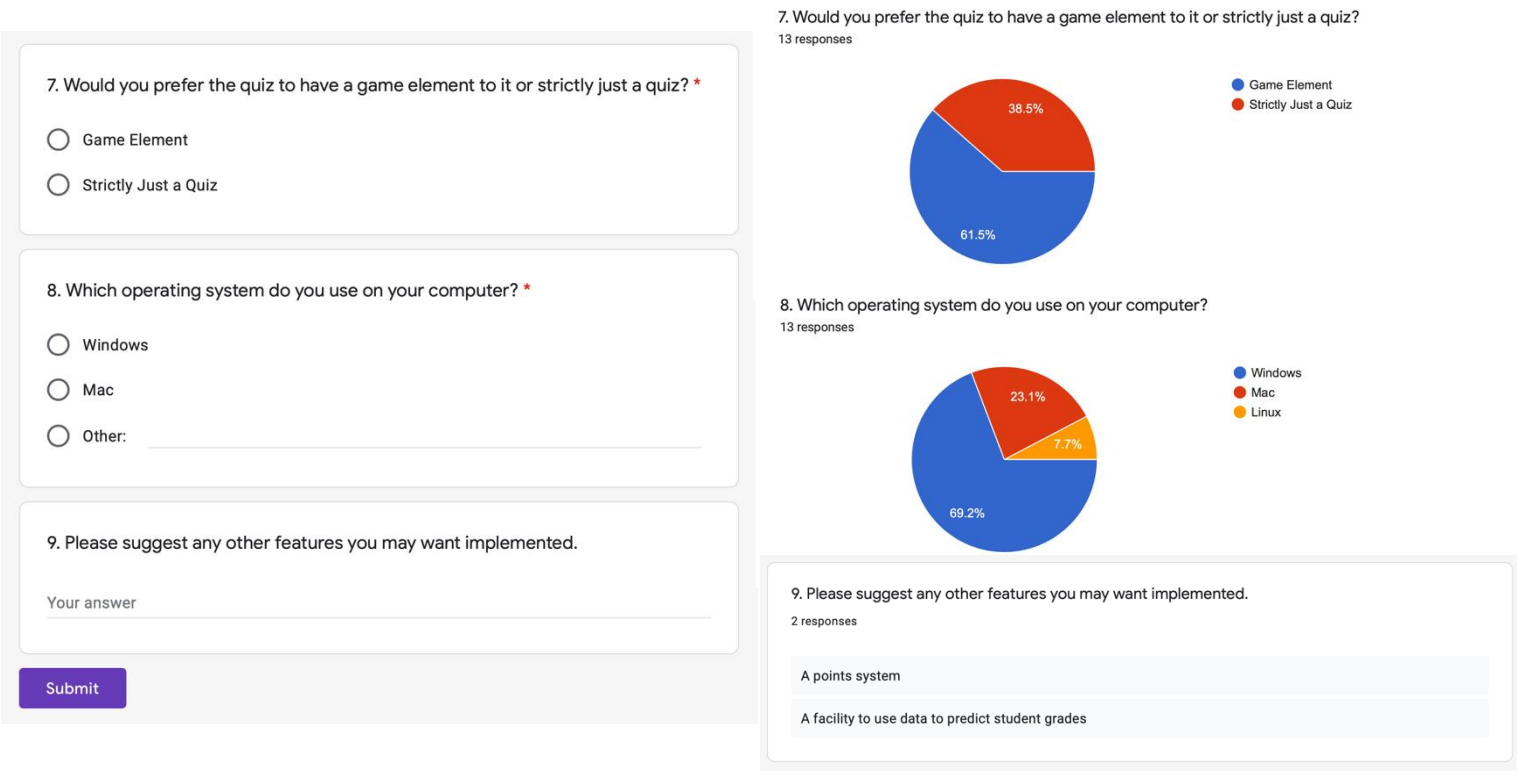
Question 6 allowed me to see which features would be the most important to students and teachers. The two most important features according to the survey were the ability to access previously created quizzes and clear and easy-to-use buttons and instructions on how to guide yourself around the program. These shall form a crucial part of my program. Also very important for stakeholders was the ability to create new, personalised quizzes. Therefore, I shall make it priority to implement these features into my program.

5. As a teacher or student, would you find it useful to be able to share your progress with other users?
13 responses



6. What would be the most important aspects of a quiz program that recorded progress
13 responses





With question 7, I tried to determine whether stakeholders would prefer the system to be as simple as possible for them to use, or instead if they preferred a more interactive, a more engaging format to it. From the response, I can see that the majority of the stakeholders prefer a game element to be included in my program.
Question 8 gives me a guide as to what system requirements I may need for my program – the majority of the people used Windows, with a few Mac users and one Linux user.
Question 9 didn't get many responses regarding an extension of the features I had to offer, but one user suggested a point user to be put in place which I agree is a good idea for my program. Another idea was to allow prediction of student grades based on analysis of progress throughout quizzes. This would make use of performance modelling, which allows us to build models or simulations in order to best predict the outcomes of a scenario.

3.1.4 Specify the Proposed Idea:

The Proposed Idea:

The proposition is a tool for my client and other stakeholders to overlook the students' skill of quick thinking and subject knowledge in the form of a quiz. This quiz may be single player or multiplayer and should allow for the tracking of progress of singular students, and the comparison between a number of them. This can be done simply using a tabular format or as an extension through graphs and charts using visualisation. With continuous stimulation of the brain by the quiz, there should be an improvement in both quick thinking and subject knowledge, fuelled by the incentive of accumulating points that increase with the complexity of the questions and the competitive nature of a multiplayer game. But also, uniquely and primarily should have the ability to monitor and record the students' progress in a convenient format for my client. Improvements to the subject knowledge would be achieved by displaying the correct answer if the child's answer is incorrect. The time

element introduced in the game will allow the child to practice how long it takes for them to answer the questions. It has been shown that taking a quiz improves long-term retention more than spending an equivalent amount of time restudying the material (as stated by Henry Roediger and Jeffrey Karpicke – psychology researchers into the memory of the human brain). The quiz would be presented to the user through a graphical user interface, in which it is made easy to read, understand and answer several questions on a given topic or subject. The user would simply have to click a button (if playing single player) or move an avatar through keys on the keyboard (if playing multiplayer). This can be achieved through abstraction and decomposition to allow users to focus on the key elements of the program, presented to them through a user-friendly interface, and by breaking down the larger issue of 'a program that is able to successfully track the progress of answered questions by pupils, as well as store the progress of the students and present them in a way that allows her to easily monitor improvement and ability of students'. This can be broken down into sub-problems such as question-making, question-answering, progress storing, etc. Whether the selected answer(s) are correct or incorrect, it must be displayed in a friendly and encouraging way, perhaps with an explanation if incorrect. The progress of the students, the subject, topics, questions, answers, and explanations can all be compiled within a database (or multiple), or alternatively through lists. To ensure the security of the students' progress, the program could be locked by a login of which only Mrs Parker and her colleagues are aware of. The use of buttons or the movement of an avatar supersedes the use of pen or pencil on paper, and avoids confusion with accidental markings or a change of the mind. However, the program would be prone to mis-clicks and wouldn't be effective without a fully operational keyboard or mouse. To combat this, it may be beneficial to have a confirm button, in order for the student to reconsider their answer or not be punished for mis-clicking. The use of databases and / or lists makes it infinitely easier for Mrs Parker and her colleagues to use, since they are not bombarded with many sheets of paper, and with loads of numbers across different pages. Instead, the data can be presented in tabular or graphical format and would be a lot clearer to read and interpret. This proposition assists the concerned parties in their day to day life: Mrs Parker to track and record progress of her students, her colleagues to monitor the progress of the students, and the students in improving their retention of subject content and their recall speed of the content, encouraged through the competitiveness of the quiz, and also allowing their teacher, Mrs Parker, to better tailor and organise lessons and tasks for them.

Although not explicitly asked by my client, a significant number of stakeholders expressed an interest in the option of having pre-created quizzes ready for them. This can be prepared for core subjects. For example, for subjects requiring mathematical testing, questions can have a general format in which they follow, but use random numbers which differ every time you attempt the quiz. By incorporating the various, relevant equations to each question, I am able to simply input the randomly chosen numbers into the equation to receive an answer that is available to the user. When doing this, I can make use of heuristics - solving a problem more quickly when classic methods are too slow, or for finding an approximate solution when classic methods fail to find any exact solution. The objective of a heuristic is to produce a solution in a reasonable time frame that is good enough for solving the problem at hand. This solution may not be the best of all the solutions to this problem, or it may simply approximate the exact solution. But it is still valuable because finding it does not require a prohibitively long time. This can be used for questions that do not have a assigned equation but instead a general method of approach. Here, I could also use backtracking which allows for more than one solution to be found - if a solution is found/not found the program will 'backtrack' and explore other paths/possibilities to try to find alternatives and incrementally builds to the solution.

Software and Hardware Requirements:

Software

- OS: Windows 7 or later at 64-bit / Macintosh:
    - The user will require a Windows 7 or later device or a Macintosh since the program will be programmed to work with these operating systems – Windows was the most used operating system by my target market but Python is also accessible on Mac, therefore allowing Mac users to make use of the program. By ensuring their device runs at 64-bits, they can fully maximise the use of the program.
- Python 3.7; SQLite 3 (Recommended):
    - These programs may help out when there is a fatal error/wrong question added, there is a possibility to remove it

Hardware

- Processor: Intel Core i5, 2GHz or better:
    - For processing the instructions that my program will produce, weaker processors may not be able to run the game smoothly. All Kahoot, MyMaths and QuizWhizzer are web-based, but optimised for browsers such as Google Chrome which recommends at least 1GHz for Celeron or 2GHz for Duron, therefore 2GHz should provide users the best experience
- RAM: At least 500MB; 1GB + is optimal:
    - Chrome recommends a minimum of 128MB, however users suggest that a higher memory is best for optimal performance so I will use a higher memory for my minimum requirements.
- Storage: At least 5GB:
    - For Python, the program and data storage. The 5GB will be used to store the large quantities of data for each user – needed to store the progress for students and other data. This is required for long term storage since RAM is volatile.
- Graphics card is optional:
    - The graphics card can assist rendering the program's graphical user interface.
- Power source:
    - A power source is needed to supply the components of the system its ran on.
- Motherboard: Any which can support previous components:
    - It's necessary for components to communicate with each other to allow the program to run.
- Keyboard:
    - The in game controls and login screen require an input of letters and numbers, which is most easily input via a keyboard.
- Mouse:
    - This is required for navigations between panels in the menu
- Monitor (at least 920 x 720):
    - To output the GUI of the application to the user

Project Limitations:

There are several limitations of my software as outlined below:

- Online accessibility - in the time frame provided, I will not be able to implement adequate security measures to allow online access such as for logins and data protection of users. Overall, this limits the access and features of the software, but it does result in stronger security and privacy which is important to comply to the Data Protection Act 1998.
- Independent work – since I am working alone, as opposed to working in a team of programmers, my solution will probably not be has high quality as current systems such as Kahoot and MyMaths because they would have been developed and maintained by a group of professionals. This, along with the time constraints I have, may mean my solution has limited functionality compared to competitors.
- Pre-created questions – since questions in pre-created quizzes are to be randomised, they must follow a strict format in which the question remains the same, but involves different mathematical values, therefore restricting the types of questions that will be available to the user. These questions will solve the randomised questions using equations and therefore will have to be extremely simple for the machine to understand. Moreover, the randomised values generated from the computer may be 'mean' for particular questions, and lead to messy calculations which may be unrealistic in comparison to an examination paper.
- Multiplayer – the program will be significantly limited in its multiplayer functionality since it will be too complex to implement in the given time frame. Multiplayer access will have to be limited to a single machine since the program will be offline, therefore restricting it to two players on one machine. For my client, that would have to mean several machines must have the application installed, and players would have to share machines and play separately which could increase the time it takes to complete a quiz.
- Databases – If the databases involved were to store large amounts of data, the speed of the program would begin to suffer since a longer time period is involved in searching and sorting through the various tables. It also requires careful programming to maintain referential integrity and is as consistent and efficient as possible.
- Offline – since the program will be offline, the software will be stored on the school's system because the program will be written specifically for my client and his students. This means it cannot be accessed elsewhere, such as via the internet, other than in the school. Although this is a limit in access, it will mean the security of the database is better as the data is not sent across the internet. This means the accessibility of the program is reduced but the security perhaps improved.

Success Criteria:

1. A clean, modern, user-friendly graphical user interface - both my stakeholder and target audience wanted a clean, modern interface which provides extra details when needed. To do this I will have a main dashboard with key information (to make it easier to use and guide themselves around the program) and options to go on other relevant pages, through clear, labelled button. There shall be a range of modern and user-friendly colours for the pages and all buttons shall be appropriately labelled by text or identifiable with an appropriate

logo. There shall be clear headings (and sub-headings) to indicate the content of each page and relevant images if needed. A menu will be provided to access the different features of the program.

2. Use of user access levels with secure authentication - I will allow for different users to create accounts and these will each have their own logins to ensure user privacy. I will make sure users have their own passwords and will include other authentication methods such as questions to ensure security which was a major concern for my stakeholder. I will use databases to store data for each individual user. These accounts should be categorised in terms of administration level, i.e. teachers have a higher priority than students, etc. Administrator accounts should allow the user to edit questions as well as view student statistics; students will only be able to view their own statistics and answer quizzes.

3. Login lock - Users will be able to attempt entering a password 3 times before their account is locked (admins will be able to revert this) meaning they will be unable to log on to that account – this is to avoid brute force attacks.

4. Creation of quizzes – my client and stakeholders should be able to create personalised quizzes for end users to answer. This would require administrator accounts the ability to add, edit, remove various questions within one quiz, as well as create new quizzes and sort them in an appropriate manner. Teachers should be able to set questions and time limits.

5. Pre-created quizzes – a high response of stakeholders as well as my client expressed an interest in having access to quizzes that had already been made. Questions can be randomised and perhaps worked out using equations or other relevant methods. This can be done for a variety of core subjects and could be subject to change by admin accounts.

6. Points system - students will be able to compete and earn points for correct answers. Also, there should be a facility to develop answer streaks and therefore accumulate higher points for higher streaks. This provides an incentive for students to maximise their learning.

7. Database storage - accounts and quiz data will be stored on a separate database, allowing more efficient storage. Quiz data will also have the ability to be represented in various formats such as tables, graphs and charts. There also should be the ability to store data in an external file.

8. Identification of difficulties – there should be the facility for teachers to determine which questions/quizzes/topics/subjects are poorly answered to allow them to scope the progress and the potential learning areas that need to be explored.

3.2 Design of the Solution

3.2.1 Decompose the problem:

To solve the problem more efficiently, I will split the problem into multiple smaller modules, using abstraction to isolate the key elements of the problem and decomposition to break down the information and requirements for each task and therefore make it easier to create a solution in a logical way. Each module will cover one of the system objectives that were identified in the analysis section. To make it easier in the implementation stage, I will include flowcharts and/or pseudocode where I think it's applicable in addition to explaining how I intend on solving that particular problem.

Here are the key elements of my program:

- User access levels with secure authentication –
  - Logging in has three choices of either admin login (for the teacher), student login (for the students) or parent login, all of which require a username and password and give different accessibilities to the user
  - Username and passwords shall be checked against a database to be authenticated
  - Users are locked if an incorrect password is entered three times; this is reversible through admin accounts

  - Signing up; users should be able to sign up by entering a username not already in the database and a password which is secure, i.e. meets a character limit/has special characters, etc.
  - New usernames and passwords should be added to the existing database.

- User-friendly user interface –
  - Separate menus for admin users, students, parents; admin users should have extra accessibilities such as the creation of quizzes, tracking students' progress; students will be able to access teacher-created quizzes or pre-created quizzes as well as their own progress; parent users will be able to view the progress of their child
  - Clear headings and sub-headings to indicate the content of each page and relevant images if needed.
  - Use of modern, user-friendly colours for pages and buttons
  - Clear labels or logos for buttons

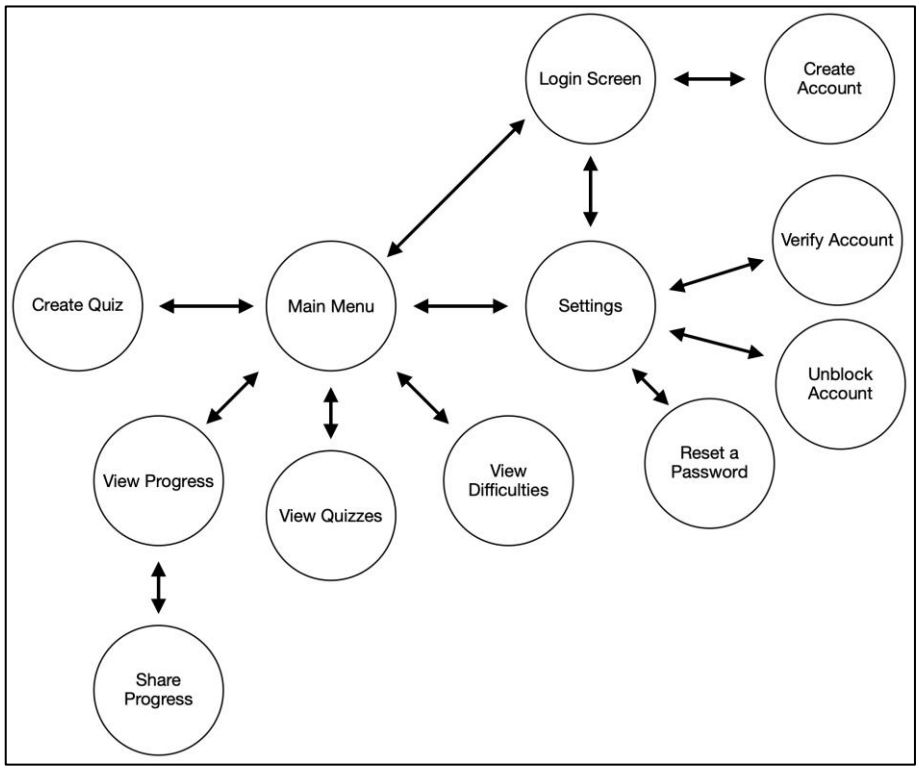
- Quiz Creation –
  - Administrator accounts should have the ability to create a quiz by adding, editing or removing different questions.
  - Teachers should be able to set questions and time limits.

- Pre-created quizzes –
  - Random questions on an exclusive topic or subject shall be created in the form of a quiz of which teachers and students have access to.
  - Questions with a mathematical aspect (of a pre-determined format) shall randomise numbers and consequently calculate answers using equations which shall be incorporated into the program.

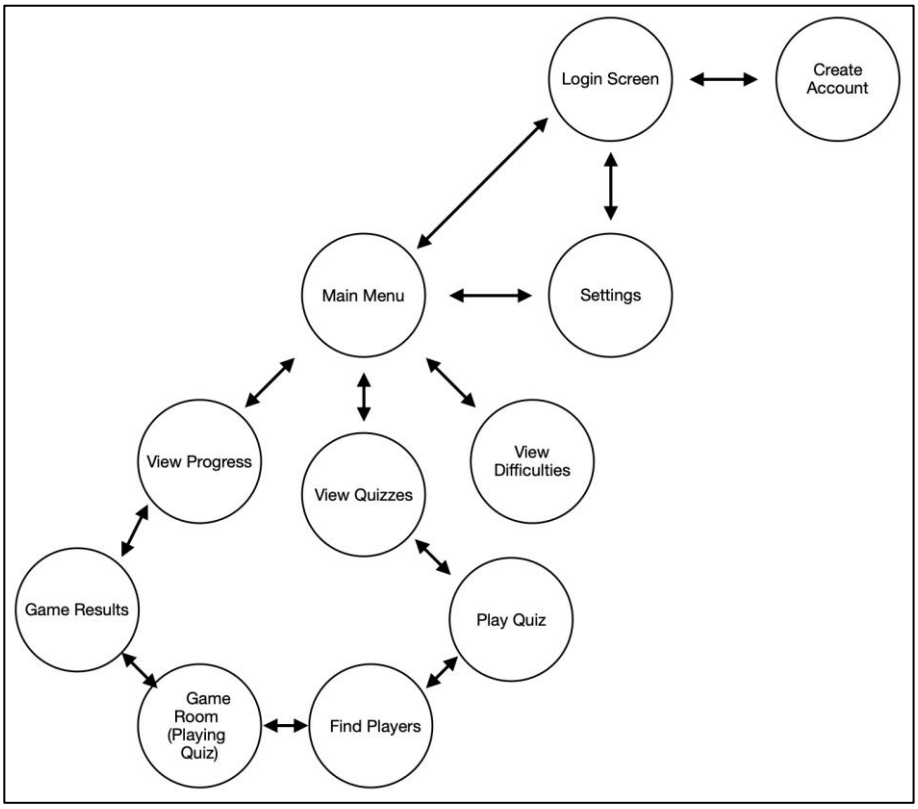
- Points system –
  - Students will earn points for answering a question and shall accumulate them over a quiz.
  - Students will earn an incremental number of points based on an answer streak that develops over a quiz.

- Progress tracking –
  - The progress of students from quizzes taken shall be recorded for teachers to see.
  - The progress of students shall be stored in a database.
  - Teachers will have the ability to see progress in the form of a table, chart or graph.
  - Progress will be able to be represented in an external file.
  - Difficult questions / topics shall be identified when students partake in quizzes and available to see to the teachers.

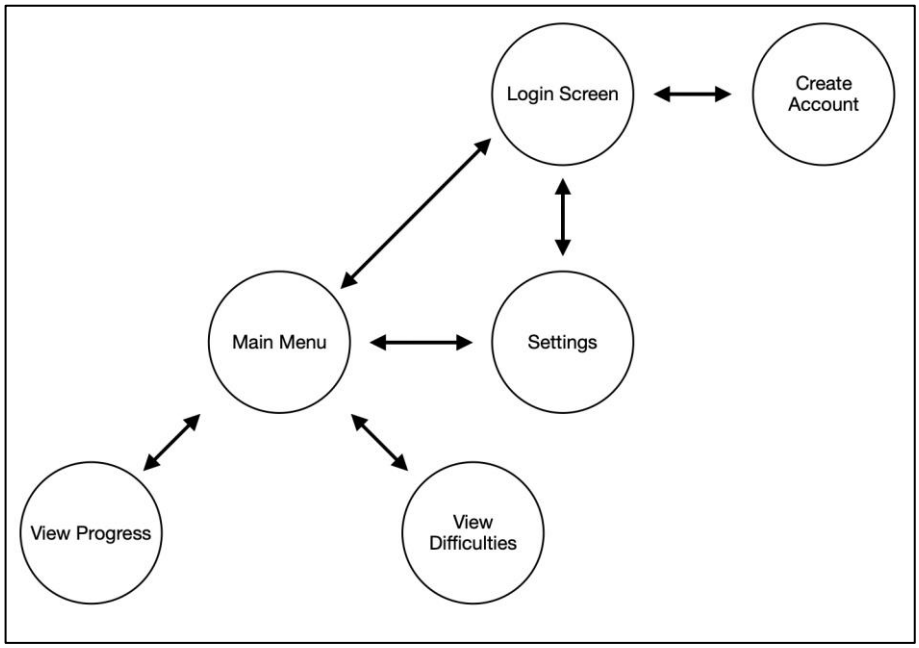
These are the main functionalities that an admin user will be entitled to :



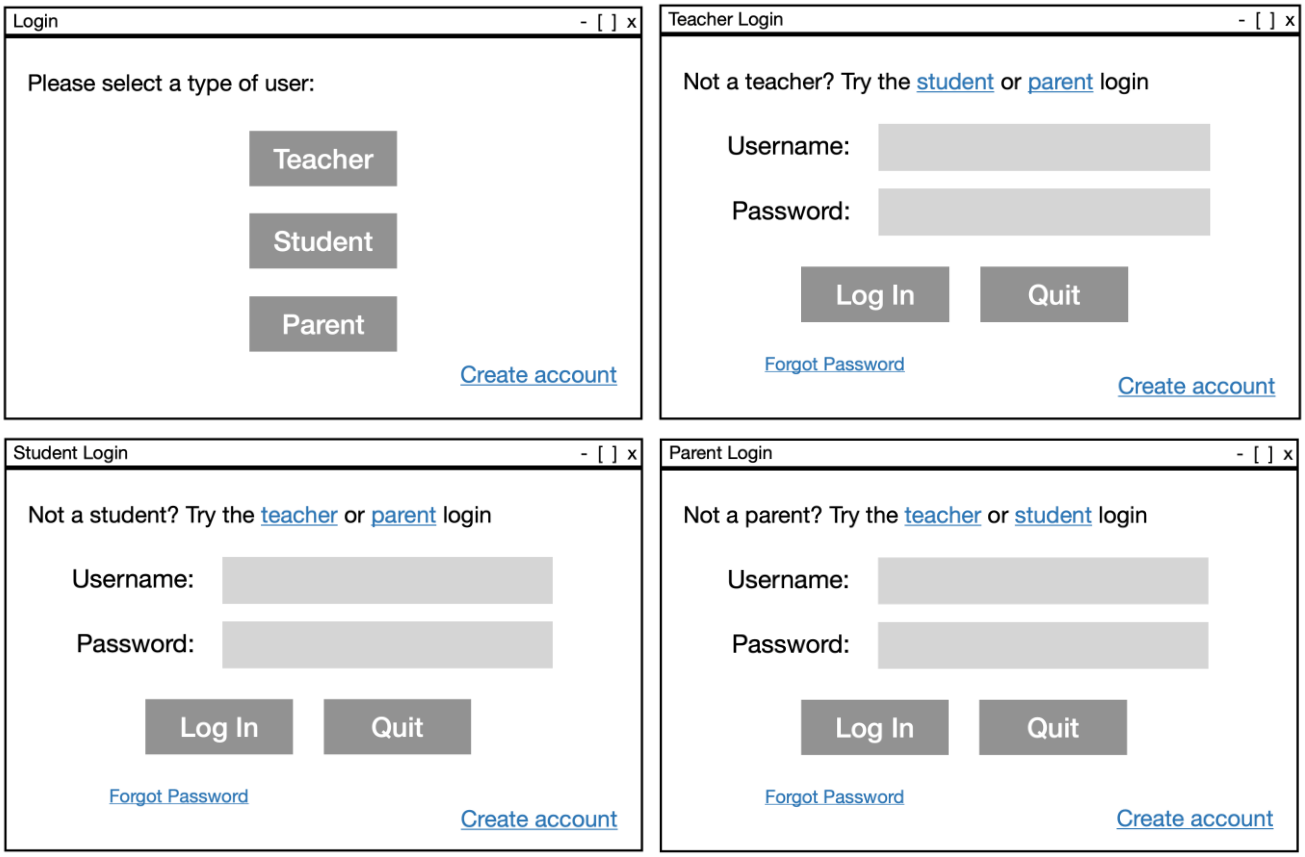
These are the main functionalities that a student user will be entitled to :



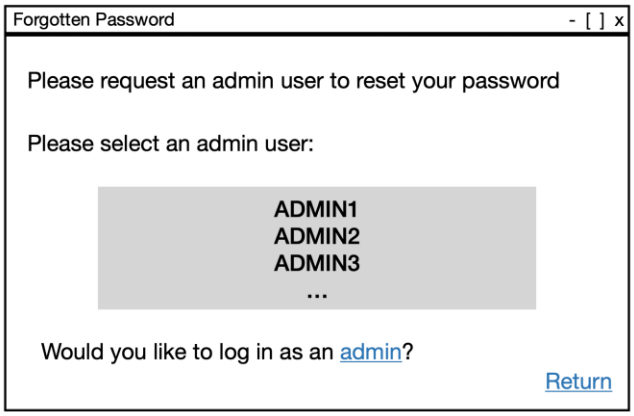
These are the main functionalities that a parent user will be entitled to :



LOGIN SCREEN:



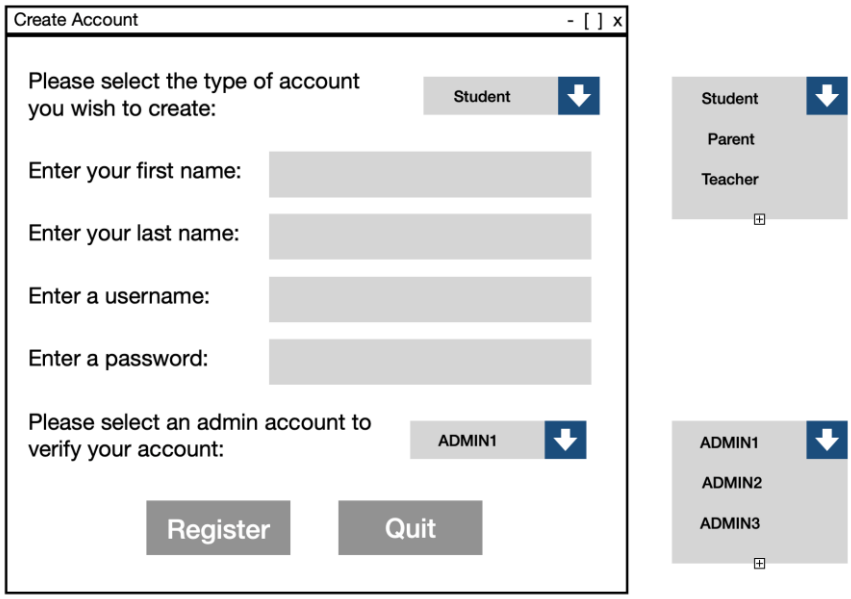
When the application is opened, the user will be shown a page in which they verify what type of user they are: teacher, student or parent. This will then bring them onto their own corresponding login page in which they are able to access their account. Once the user enters their credentials they can attempt to log in (entry fields are provided for entering the necessary details) to give them access to the main menu. The credentials are checked through an encrypted database (done by the program) to check if correct: if the entered credentials are wrong student users will have 2 more attempts before their account is locked (3 attempts in total). This can be undone by administrators. Admins are not subjected to this limit. If the user has forgotten their password, they can ask an admin to reset their password which the 'forgot password' link redirects you to. If a user mis-clicks on the type of user on the first page, they are able to change pages on any page they are directed to at the top of each page.

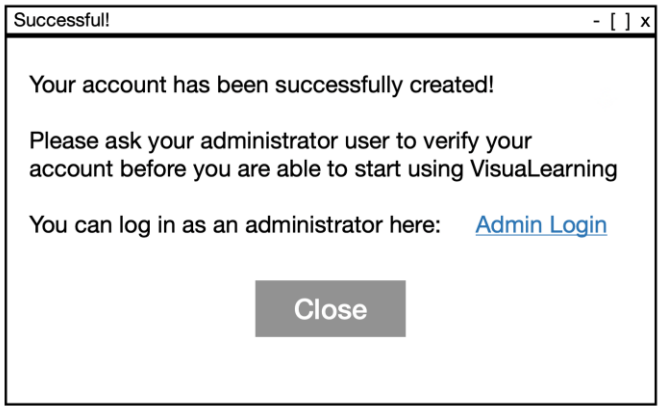


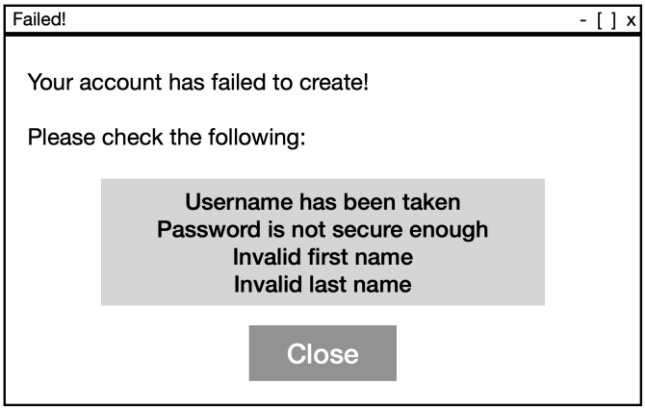
Here, a student is able to request to change their password. They do this by selecting an admin user from the box provided which allows a notification to be sent to the admin user account that a password needs to be reset. The idea here is that a student or parent will select their own teacher. This will maintain the security of the users and will also make it quicker to retrieve data.

There is a link to the admin login for ease of admins if they happen to be present at the time.
There is also a link to return to the log in home page.
If a user does not already have an account, they are able to create account from the 'Create account' button on each login page.

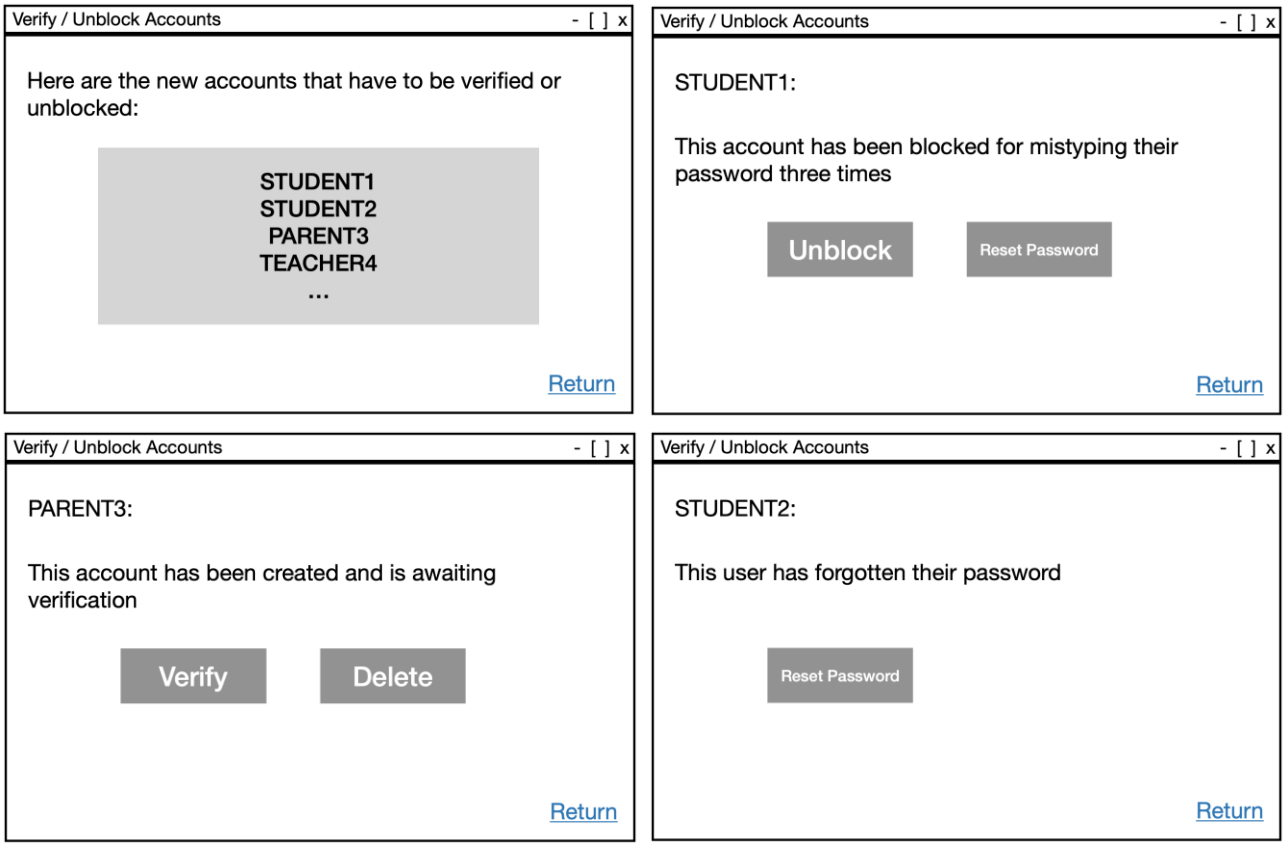
When a user creates an account, they are taken to a separate page which confirms what type of account they are making (student, parent or teacher) by use of a dropdown menu, their first name, last name, username and password.
When entering their password, characters entered shall appear as asterisks to maximise the security of the user.
They also must choose an admin account which will have the option to verify the new account before they start to use it once logged in.







Once the user presses the 'Register' button after completing the form, one of two pages may appear. If all fields are correctly filled out, a 'Success' page will appear to inform the user that the account has been created yet is waiting for verification from an administrator. For ease of access, if an admin user is present, there is a link to the admin login page. Alternatively, if the form has not been filled in correctly, a 'Failed' page will appear informing the user that the account has failed to create and which fields have been incorrectly done. In the example above, there is a list of reasons for why an account may fail to create, however, it is not an exhaustive list – there may be other reasons for failure such as an empty field, etc.



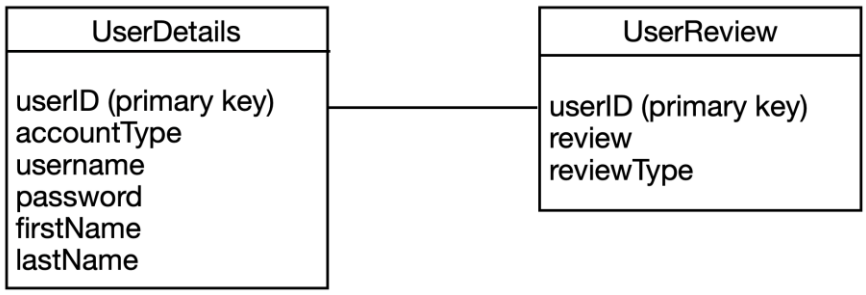
When an account has been blocked, newly created or a user has forgotten their password, they shall appear in the settings of an admin user under 'Verify / Unblock Accounts'. This allows them to see all the accounts that need reviewing and allows them to press each one to identify the problem and resolve it. This includes unblocking blocked accounts or resetting their password, and verifying or deleting a newly created account.

The process of logging in, signing up, verifying, unblocking and resetting passwords will make use of one database. This database will contain a table that will hold the data of all existing accounts including fields for their username, password, first name, last name, their account type, whether their account is under review and if so, why the account is under review. Each account will also have a unique user ID (the primary key of this table) to allow for easy recognition of each account – this avoids confusion between two accounts which may share first names and last names, etc. The 'review' field will contain a Boolean value – True or False – to determine whether admins will have to review the account.

AccountInfo
userID (primary key) accountType username password firstName lastName review reviewType

AccountInfo			
Field Name	**Data Type**	**Sample Data**	**Key Field (PK / FK)**
userID	Integer	1	Primary Key
accountType	String	Teacher Student Parent	None
username	String	heman_seego1	None
password	String	Pass-word123	None
firstName	String	Heman	None
lastName	String	Seegolam	Secondary Key
review	Boolean	True False	None
reviewType	String	NewAccount ForgottenPassword BlockedAccount	None

Although this database is usable in my program, it can be made a lot more efficient. This can be done through normalisation, i.e. the process of trying to come up with the best possible design for a relational database. The flat-file database above (consists of one table), is currently in first normal form. This is because it contains no repeating (group of) attributes, there is a unique primary key and the data is atomic, i.e. data is split into its simplest form across fields. This database could cease to be in first normal form if for example the firstName, lastName, review, or reviewType field was the primary key since values may not be unique. Also, if for example the firstName and lastName fields were merged to simple a name field, then the data would no longer be atomic.
This database is also in second normal form. For a database to be in second normal form, the existing data must first be in first normal form. Secondly, it must have no partial dependencies – everything must rely on the primary key.
This database can be modified into third normal form. To do this, the existing data must be in second normal form, which it is, and secondly, there must be no transitive dependencies, i.e. a non-primary key field should not be dependent on another non-primary key value. In the current model, the reviewType is dependent on whether the review field is True or False.

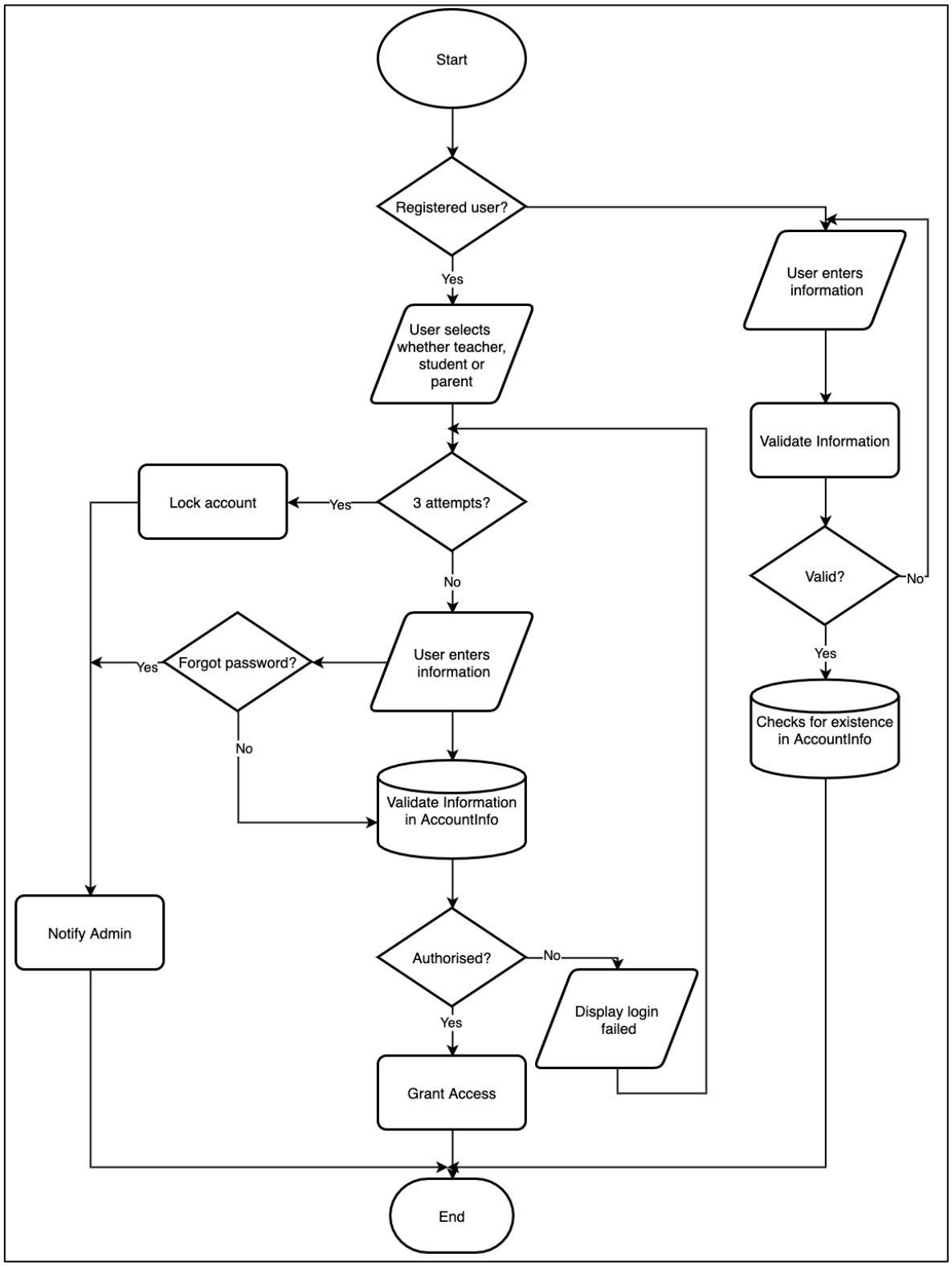


In this new model, the reviewType field was dependent on the review field. Both being non-primary key fields, they had to be separated out into a table which has a one to one relationship with the original table. In doing so, all transitive dependencies have been removed.

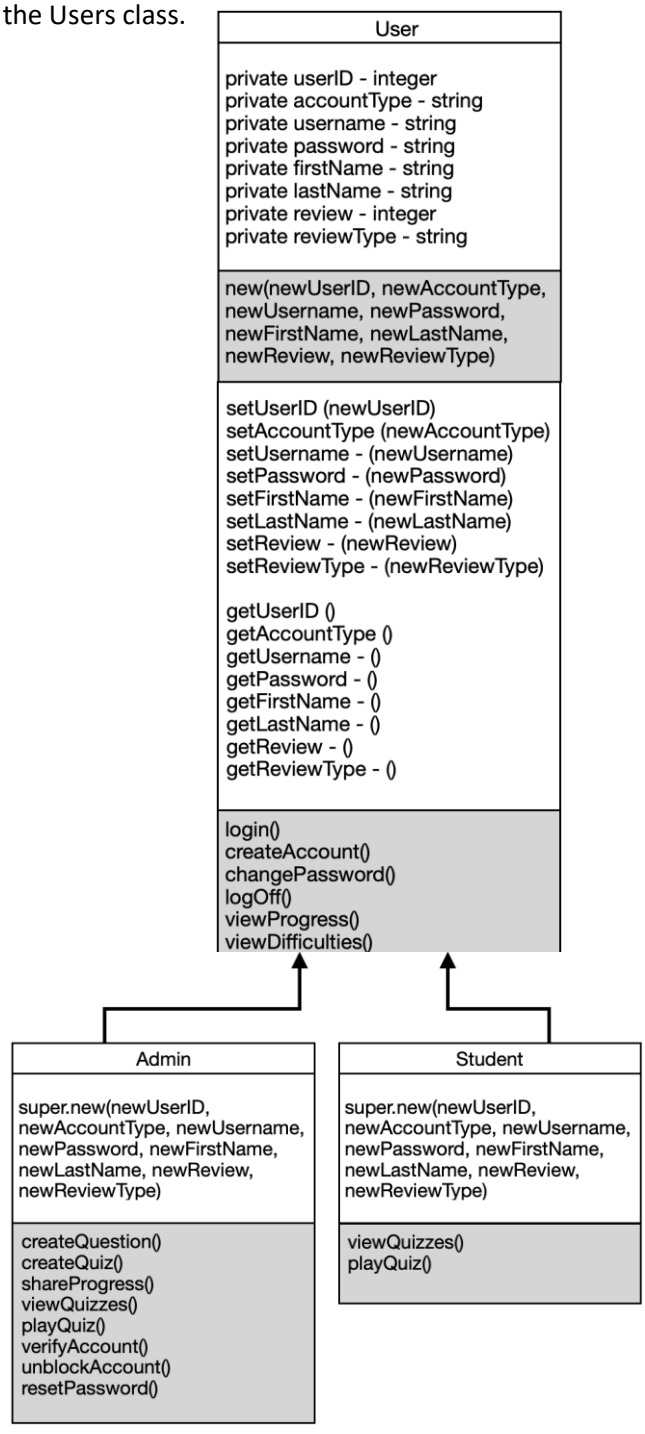
The advantage of normalising databases is that there is no data redundancy, i.e. no repeated data which can cause inefficiencies. It also become easier to maintain & change normalised databases, data integrity is maintained and sorting and searching becomes faster since there is no unnecessary duplication of data, producing smaller tables with fewer fields. This also results in saved storage space. Finally, normalised databases with correctly defined relationships between tables will not allow records in a table on the “one” side of a one to many relationship to be deleted accidentally.

To open the database –
Pseudocode:

AccountInfo.open()
…
AccountInfo.close()



This program will make use of three main classes, all inheriting the same set of core information, however with individual additional privileges. Objects in the User class will have generic methods and attributes that will be inherited by the Admin and Student classes. All objects of the class User (the users) will have general attributes of a user ID, the account type (admin, student or parent), a username, a password, their first name and last name, whether their account is under review and if so, what type of review. These users are able to make use of various methods such as logging in, creating an account, changing their password, logging off, viewing their progress and viewing their difficulties (or that of their child or students). Objects within the Admin class are disposed to more functionalities such as creating questions, creating quizzes, sharing progress, viewing quizzes, playing quizzes, verifying newly created accounts, unblocking accounts and resetting passwords of other accounts. Objects within the Student class are able to view quizzes and play quizzes in addition to the methods included within the Users class.



To create the classes –
Pseudocode:

```
// declare class named User and create its attributes
class User
        private userID
        private accountType
        private username
        private password
        private firstName
        private lastName
        private review
        private reviewType

// create constructor for User class
        public procedure new(newUserID, newAccountType, newUsername, newPassword,
        newFirstName, newLastName, newReview, newReviewType)
                userID = newUserID
                accountType = newAccountType
                username = newUsername
                password = newPassword
                firstName = newFirstName
                lastName = newLastName
                review = newReview
                reviewType = newReviewType
        endprocedure

// encapsulation of the class
                public procedure setUserID(newUserID)
                        userID = newUserID
                endprocedure

                public procedure setAccountType(newAccountType)
                        accountType = newAccountType
                endprocedure



                public procedure setUsername(newUsername)
                        username = newUsername
                endprocedure

                public procedure setPassword(newPassword)
                        password = newPassword
                endprocedure

                public procedure setFirstName(newFirstName)
                        firstName = newFirstName
                endprocedure
```

```
                public procedure setLastName (newLastName)
                        lastName = newLastName
                endprocedure

                public procedure setReview(newReview)
                        review = newReview
                endprocedure

                public procedure setReviewType(newReviewType)
                        review = newReviewType
                endprocedure
endclass

// declare child class named Admin and create its attributes
class Admin inherits User

// create constructor for Admin class
        public procedure new(newUserID, newAccountType, newUsername, newPassword,
        newFirstName, newLastName, newReview, newReviewType)
                super.new(newUserID, newAccountType, newUsername, newPassword,
                newFirstName, newLastName, newReview, newReviewType)
        endprocedure
endclass

// declare child class named Student and create its attributes
class Student inherits User

// create constructor for Student class
        public procedure new(newUserID, newAccountType, newUsername, newPassword,
        newFirstName, newLastName, newReview, newReviewType)
                super.new(newUserID, newAccountType, newUsername, newPassword,
                newFirstName, newLastName, newReview, newReviewType)
        endprocedure
endclass
```

To hide the passwords of the users from other users, I can make use of either hashing or encryption. If users were to receive the key for the encryption of passwords, they would be able to reverse the process to decrypt the cipher and retrieve the original password. However, once text has been hashed, this process cannot be reversed to retrieve the original data, therefore proving to be an ideal approach for hiding the passwords of users from other users. I shall provide the facility within my program, abstracted from user eyes, to hash inputted passwords and store them in the database. This would mean every time the user inputs their password to log in, their input would run through a function to return a hashed value of that input, and then be checked against the value in the database to see if correct.

To create the login –
Pseudocode:

```
// create two boolean variables to determine whether log in is successful and whether user wants to return
return = FALSE
success = FALSE

// checks to see whether user would like to continue and if log in is not yet successful to start while loop
WHILE success == FALSE and return == FALSE DO
        // when user presses log in button, each input from the textboxes are placed in a variable
        IF (Event -> user clicks on login button) THEN
                username <- User's input in the "Username" textbox
                password <- User's input in the "Password" textbox
                // checks the values inputted against the database
                FOR value IN (Database: AccountInfo username Field) DO
                        IF username == AccountInfo username THEN
                                password <- Hashed Password using hashPassword() module
                                IF password == (Database: Corresponding User Password) THEN
                                        Success = TRUE
                                        Load the User Menu Interface
                                ELSE
                                        DISPLAY Incorrect Password
                                ENDIF
                        ELSE IF value == length(Database: AccountInfo username Field)
                                DISPLAY Username doesn't exist
                        ENDIF
                ENDFOR
        ELSE IF EVENT button "Return" is clicked THEN
                return = TRUE
        ENDIF
ENDWHILE
```

In this pseudocode, two boolean variables are created: 'return' and 'success'. The 'return' variable determines whether the 'Return' button has been pressed on the login page. If found to be True, this stops the while loop from functioning and therefore stops the login process. Whilst False, the while loop is able to proceed. The 'success' variable determines whether the login of the user has been approved and therefore stops the loop whilst True. Inside the loop, the click of the log in button triggers the selection to begin, i.e. the if-else statements. When the user presses the button,

the input to the username and password textboxes in the GUI will be recorded into their own variables. This allows the code to then check whether the username given matches a username in the database 'AccountInfo'. If so, the corresponding password is checked against the hashed version of the inputted password (since that is what is stored in the database). If all credentials match, 'success' is then turned to True, therefore stopping the loop and the user menu interface is loaded. If the password is incorrect, a message is displayed informing the user of this. If the username entered matches against none in the database, then a message is displayed informing the user of this.

This pseudocode can be extended to carry out this functionality for each of the individual pages mentioned earlier, i.e. the admin login, the student login and the parent login. To do so, the user's account type, obtained from the database field accountType, can be checked against the type of login that is being used. This can be done by:

For an admin login:

```
IF username == AccountInfo username THEN
        password <- Hashed Password using hashPassword() module
        accountType <- Database: Corresponding User Account Type
        IF accountType == 'admin' THEN
                IF password == (Database: Corresponding User Password) THEN
                        Success = TRUE
                        Load the User Menu Interface
                ELSE
                        DISPLAY Incorrect Password
                ENDIF
        ELSE
                DISPLAY Please use dedicated login
        ENDIF
ELSE
        DISPLAY Username doesn't exist
ENDIF
```

For a student login:
The line stating 'IF accountType == 'admin' THEN' can be modified to read 'IF accountType == 'student' THEN'

For a parent login:
The line stating 'IF accountType == 'admin' THEN' can be modified to read 'IF accountType == 'parent' THEN'

To create an account –
Pseudocode:

```
// …using User class
// creates two boolean variables that determines whether the inputted username has been taken and if the password is strong enough
takenUsername <- False
strongPassword <- False

// storing inputs into own variables
newUsername <- User's input in the "newUsername" textbox
// checking whether username exists by comparing it to each field in the username column of the database
FOR VALUE IN (Database: AccountInfo username Field) DO
        IF username == AccountInfo username THEN
                takenUsername <- True
                DISPLAY Username Taken
        ENDIF
ENDFOR

IF takenUsername == False THEN
        // allows user to change password until is matches criteria for strong password
        WHILE strongPassword <- False DO
                newPassword <- User's input in the "newPassword" textbox
                // calls separate function to check whether password is strong enough
                lenPassword, numInPassword, symbolInPassword, caseInPassword <-
                passwordCheck(newPassword)
                // if all conditions from the function are true, password is strong, else not
                IF lenPassword == True AND numInPassword == True AND symbolInPassword ==
                True AND caseInPassword == True THEN
                        strongPassword <- True
                ENDIF
                IF strongPassword == True THEN // returned from passwordCheck function
                        // hash the inputted password to match database format
                        newPassword <- hashPassword(newPassword)
                ELSE
                        DISPLAY Password Not Strong Enough
                ENDIF
        ENDWHILE
        // places each attribute inputted by the user into its own variable
        newUserID <- (Database: AccountInfo userID from (length(userID Field))+1)
        newAccountType <- User's input from "newAccountType" textbox
        newFirstName <- User's input from "newFirstName" textbox
        newLastName <- User's input from "newLastName" textbox
        // creates a new instance of the class User to create a new user
        newUser <- new User(newUserID, newAccountType, newUsername, newPassword,
        newFirstName, newLastName, True, 'NewAccount' )
        // adds new user into database
        Database: AccountInfo – Add newUser as a record
ENDIF
```

To check the strength of the password–
Pseudocode:

```
function passwordCheck(newPassword)
        lenPassword <- False
        numInPassword <- False
        symbolInPassword <- False
        caseInPassword <- False

        // checks to see if password is greater than seven characters long
        IF length(newPassword) > 7 THEN
                lenPassword <- True
        ENDIF
        // checks to see if there is a number in the password
        IF newPassword.isalpha() == False THEN
                numInPassword <- True
        // checks to see if there is a symbol in the password
        IF newPassword.isalpha() == False AND newPassword.isdigit() == False  AND
        newPassword.isalnum() = False THEN
                symbolInPassword <- True
        ENDIF
        // checks to see if there is both upper and lower case letters in the password
        IF newPassword.lower() != newPassword and newPassword.upper() != newPassword THEN
                caseInPassword <- True
        ENDIF

        return lenPassword
        return numInPassword
        return symbolInPassword
        return caseInPassword
ENDFUNCTION
```

With this pseudocode, an issue arises. When the function checks whether a symbol is present in the password, 'newPassword' is checked against another (pre-created) function to see if all characters are alphabet characters or if they are all numerical digits and finally if it is alphanumerical. If it doesn't consist purely of letters or number and is not alphanumerical, the function decides that there must therefore be a symbol in the password. However, it disregards the fact that the password may include spaces and therefore could provide a False output for all three conditions if numbers, letters and a space character was used. To combat this, in Python, the re module can be imported by 'import re' and we can use re.search to detect whether any symbols are in the password.

```
IF ['!#$%&'()*+,-./[\\\]^_`{|}~"'] in newPassword THEN
        symbolInPassword <- True
```

Similar to this problem, when the function attempts to detect whether a number is included, it simply does this by checking whether all characters are letters and if not, deciding that there must be a number. But some or all of the characters could be symbols, therefore not necessarily including a number. This can also be combatted with re.search.

```
IF re.search(r'[0-9]', newPassword) THEN
    numInPassword = True
```

Here, one boolean variable is created, named ‘taken’ to determine whether the username entered by the user creating a new account is taken. This variable is initially set to False. The program then takes the input of the user for their desired username and password and uses the input for the username to match it against the username field in the database ‘AccountInfo’. If the inputted username matches one already in the database, ‘taken’ is turned to True, therefore restricting the subsequent IF statement to run. Given that ‘taken’ is False, the password inputted by the user is then hashed from the hashPassword() function with parameter ‘newPassword’. Now, the program then will move to create a unique incremental user ID for the new user as well as attain all other relevant information from the input boxes of the user interface such as their name and account type. Using this information, a new instance of the class Users is created. The ‘review’ attribute will be set to True for the new user and the ‘reviewType’ to ‘NewAccount’. This object is then uploaded to the database (awaiting for verification by an admin).

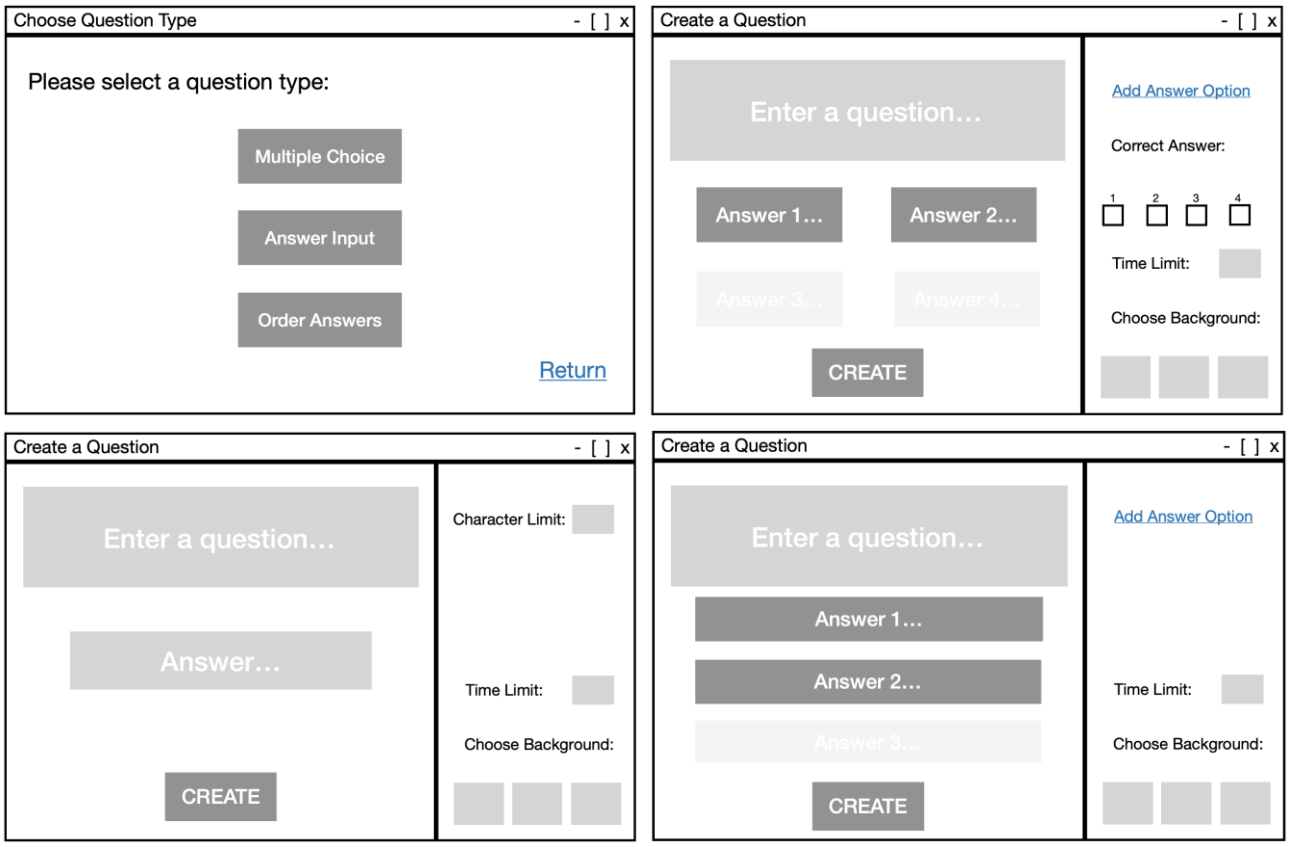
Links to success criteria:

- Use of user access levels with secure authentication
  - ability for different users to have their own logins to ensure user privacy.
  - ability for users to have their own corresponding passwords
  - ability for passwords to be encrypted and stored
  - accounts should be categorised in terms of administration level

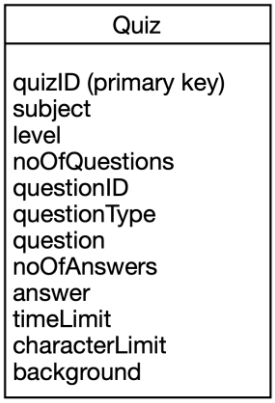
- Login lock
  - users will be able to attempt entering a password three times before their account is locked
  - admins will be able to revert this

- Account creation
  - new users will be able to create an account
  - new accounts will be uploaded to the database

QUESTION CREATION:

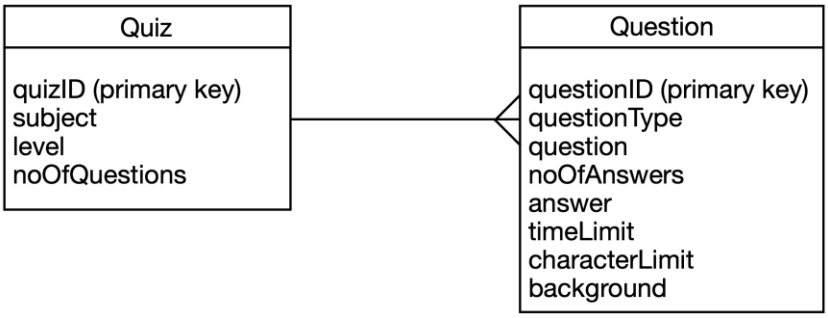


When an admin user selects the 'Create Question' button when creating a quiz from their menu, a 'Choose Question Type' page will appear. This allows the user to choose the type of question they would like to set, options being a multiple choice question, a question in which the user is required to input their own answer or a question with predetermined answers which are in need of reordering. Alternatively, they can return to the previous page. Once the user has selected their desirable question format, they are taken to its corresponding page. For a multiple choice question, the user is able to input a question, set two default answers, but also add an answer option. This will be to a maximum of 4. The user then has to tick a relevant checkbox relating to the correct answer of the question and set the time limit for the question. Since some participants of the questionnaire had previously said they would appreciate a more engaging look to the program, the user also has a choice of one of three set backgrounds. Although it could've been more beneficial for a user to be able to upload their own image, the image file would have to be in a relevant directory and would not be able to be moved by the user to elsewhere on their computer or be deleted. Therefore, I concluded it would be significantly simpler to set backgrounds but allow a choice of three different ones so whilst not being able to fully exploit the privilege, as is done on Kahoot, there is still a facility which allows for a more aesthetic appeal to the program.
For the answer input type questions, the user is able to enter a question and answer but also set a character limit that will limit how much the students are able to write as an answer. Again, users are able to set a time limit and one of three backgrounds.
For the order answer type questions, the user can enter a question and two default answers with an option to add more. The user is required to write the answers in correct order and will be automatically randomised when students start the quiz. Again, they can set a time limit and one of three backgrounds.



The process of creating questions and quizzes will make use of one database. This database will contain a table that will hold the data of all existing quizzes including fields for their quiz ID, subject, level, number of questions, question type, questions, number of answer options, answers, time limit, character limit, and finally the chosen background. For questions which have multiple answer choices, the answer choices will be stored as a list in the answer field, having the correct answer as the first item in the list. This database is currently in first normal form (contains no repeating (group of) attributes, unique primary key and atomic data). However, it is not in second normal form since there are partial dependencies and not all fields relate to the primary key, quizID, for example the answer, time limit, character limit, etc. which is independent to the question.

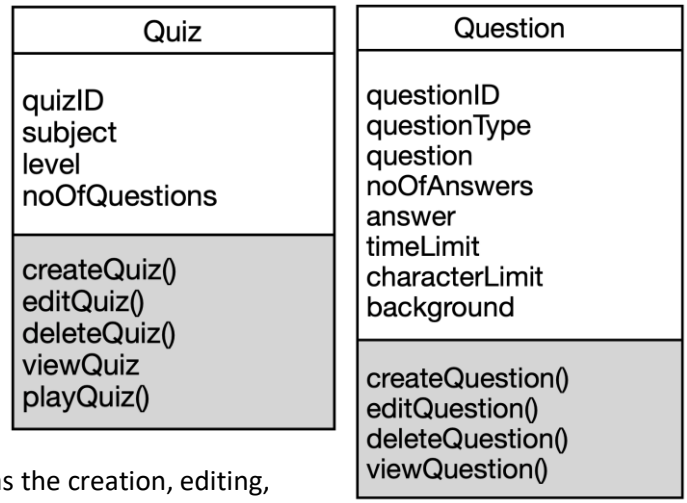
This database can be made more efficient, through normalisation. This database can be made into second normal form.



In this model, the attributes for a quiz and the attributes for a question have been separated out into two separate tables, with a one to many relationship from quiz to questions. Here, all attributes link to its respective primary key, hence meaning there are no partial dependencies. This database model is also in third normal form since there are no transitive dependencies – no non-primary key fields are dependent on another non-primary key field.

QuizInfo			
Quiz			
Field Name	**Data Type**	**Sample Data**	**Key Field (PK / FK)**
quizID	Integer	1	Primary Key
subject	String	Maths Physics, …	None
level	String	A-Level	None
noOfQuestions	Integer	10	None
Question			
questionID	Integer	1	Primary Key
questionType	String	Multiple Choice Answer Input Order Answers	None
question	String	What is a Program Counter?	None
noOfAnswers	Integer	1	None
answer	String	A register that stores the address of the next instruction to be fetched	None
timeLimit	Integer	20	None
characterLimit	Integer	50	None
background	Integer	1 (2/3)	None

This process will make use of two classes: Quiz and Question. All object of the class Quiz (the quizzes) will have general attributes of a quiz ID, subject, level and a number of questions. All object of the class Question (the questions) will have general attributes of a question ID, question type, question, number of answers, answer(s), time limit, character limit and the desired background. These classes can make use of various methods. The Quiz class can make use of methods such as the creation, editing, viewing, deletion and playing of a quiz. The Question class can make use of methods such as the creation, editing, deletion and viewing of a question.



To create the classes –
Pseudocode:

```
// declare class named Quiz and create its attributes
class Quiz
        private quizID
        private subject
        private level
        private noOfQuestions

// create constructor for Quiz class
        public procedure new(newQuizID, newSubject, newLevel, newNoOfQuestions)
                quizID = newQuizID
                subject = newSubject
                level = newLevel
                noOfQuestions = newNoOfQuestions
        endprocedure

// encapsulation of the class
                public procedure setQuizID(newQuizID)
                        quizID = newQuizID
                endprocedure

                public procedure setSubject(newSubject)
                        subject = newSubject
                endprocedure

                public procedure setLevel(newLevel)
                        level = newLevel
                endprocedure

                public procedure setNoOfQuestions(newNoOfQuestions)
                        noOfQuestions = newNoOfQuestions
                endprocedure
end class
```

```
// declare class named Quiz and create its attributes
class Question
        private questionID
        private questionType
        private question
        private noOfAnswers
        private answer
        private timeLimit
        private characterLimit
        private background

// create constructor for Quiz class
        public procedure new(newQuestionID, newQuestionType, newQuestion, newNoOfAnswers,
        newAnswer, newTimeLimit, newCharacterLimit, newBackground)
                questionID = newQuestionID
                questionType = newQuestionType
                question = newQuestion
                noOfAnswers = newNoOfAnswers
                answer = newAnswer
                timeLimit = newTimeLimit
                characterLimit = newCharacterLimit
                background = newBackground
        endprocedure

// encapsulation of the class
                public procedure setQuestionID(newQuestionID)
                        questionID = newQuestionID
                endprocedure

                public procedure setQuestionType(newQuestionType)
                        questionType = newQuestionType
                endprocedure

                public procedure setQuestion(newQuestion)
                        question = newQuestion
                endprocedure

                public procedure setNoOfAnswers (newNoOfAnswers)
                        noOfAnswers = newNoOfAnswers
                endprocedure

                public procedure setAnswer(newAnswer)
                        answer = newAnswer
                endprocedure

                public procedure setTimeLimit(newTimeLimit)
                        timeLimit = newTimeLimit
                endprocedure

                public procedure setCharacterLimit(newCharacterLimit)
                        characterLimit = newCharacterLimit
```

```
        endprocedure

        public procedure setBackground(newBackground)
            background = newBackground
        endprocedure
endclass
```

To choose the type of question:
Pseudocode:

```
procedure createQuestion
    IF (Event -> user clicks on “Multiple Choice" button) THEN
        createMultipleChoiceQuestion()
    ELSE IF (Event -> user clicks on “Answer Input” button) THEN
        createAnswerInputQuestion()
    ELSE IF (Event -> user clicks on “Order Answers" button) THEN
        createOrderAnswersQuestion()
    ENDIF
endprocedure
```

To create a multiple choice question –
Pseudocode:

```
procedure createMultipleChoiceQuestion
    // sets two default answer options
    answerOptions <- 2
    // allow the user to increase the number of answer options
    IF (Event -> user clicks ‘Add Answer' button) THEN
        answerOptions += 1
        Add Answer Option Box
    ENDIF

    IF (Event -> user clicks ‘Create' button) THEN
        question <- user's input from ‘question’ textbook
        // for an increasing value of 1, take the answer from the relevant textbox and place
        it into a corresponding variable
        FOR x <- 0 to answerOptions DO
            answer(x) <- user's input from ‘Answer (x)’ textbox
        ENDFOR
        newQuestionID <- (Database: Quiz questionID from (length(questionID Field))+1)
        questionType <- ‘Multiple Choice’
        // appends all answers into one array
        answer <- arraySize(x)
        FOR x <- 0 to answerOptions DO
            answer.append(answer(x))
        ENDFOR
        timeLimit <- user's input from ‘Time Limit’ textbox
        characterLimit <- None
        background <- user's input from background checkboxes
```

```
            // creates new instance of Question class
            newQuestion <- new Question(newQuestionID, questionType, question,
            answerOptions, answer, timeLimit, characterLimit, background)
            // add new question into Question database
            Database: Question – Add newQuestion as a record
endprocedure
```

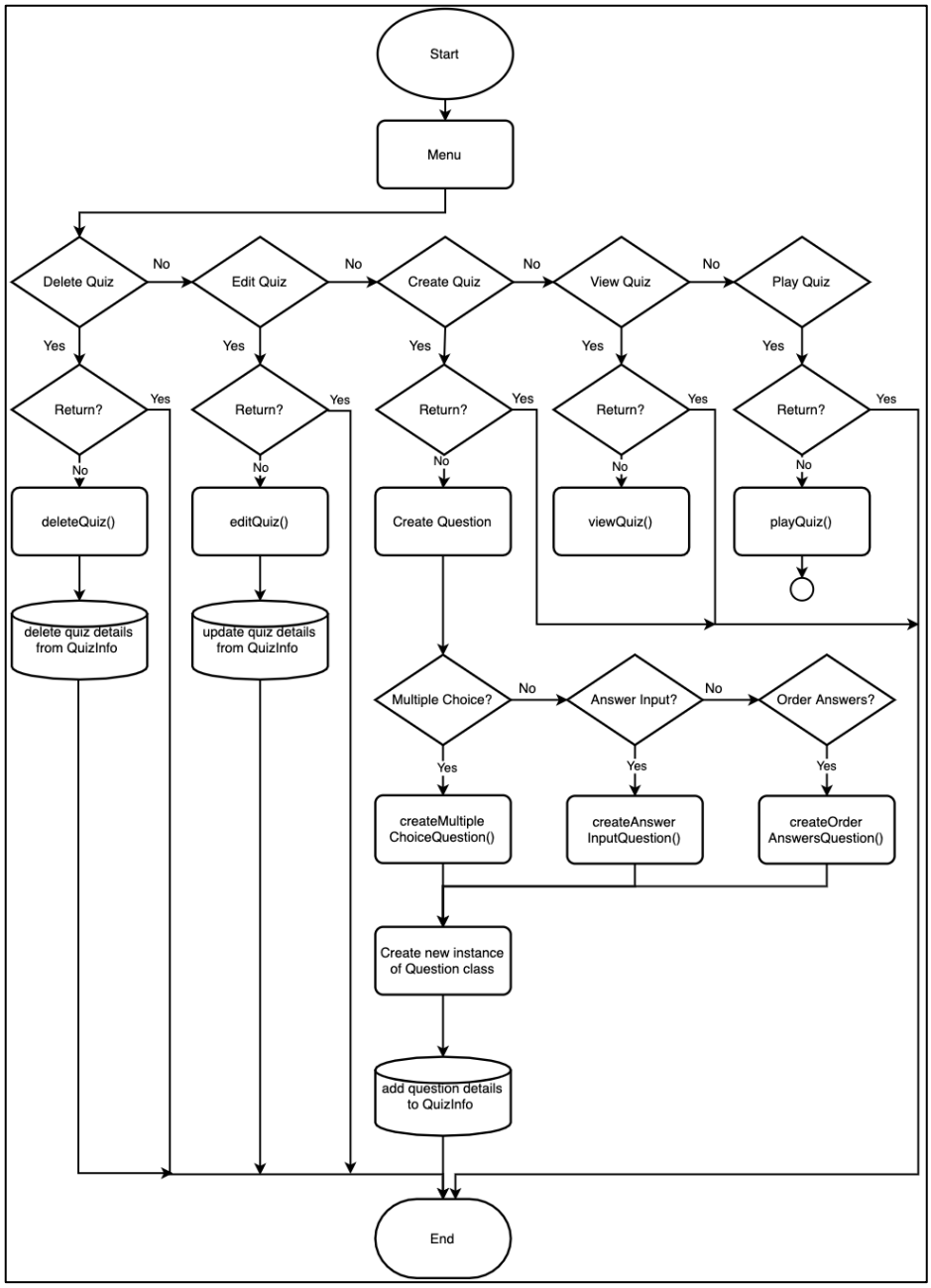
To create an answer input question –
Pseudocode:

```
procedure createAnswerInputQuestion
    IF (Event -> user clicks ‘Create’ Button) THEN
        question <- user's input in ‘questions' textbox
        characterLimit <- user's input from ‘characterLimit’ textbox
    ENDIF

    // verifies that correct answer is below the character limit
    IF length(answer) > characterLimit THEN
        DISPLAY error message
    ELSE
        newQuestionID <- (Database: Quiz questionID from (length(questionID Field))+1)
        questionType <- ‘Answer Input
        answerOptions <- 1
        answer <- user's input from ‘answer’ textbox
        timeLimit <- user's input from ‘Time Limit’ textbox
        background <- user's input from background checkboxes
        // creates new instance of Question class
        newQuestion <- new Question(newQuestionID, questionType, question,
        answerOptions, answer, timeLimit, characterLimit, background)
        // add new question into Question database
        Database: Question – Add newQuestion as a record
    ENDIF
endprocedure
```

To create an order answers question –
Pseudocode:

```
procedure createOrderAnswersQuestion
    // sets two default answer options
    answerOptions <- 2
    // allow the user to increase the number of answer options
    IF (Event -> user clicks ‘Add Answer' button) THEN
        answerOptions += 1
        Add Answer Option Box
    ENDIF

    IF (Event -> user clicks ‘Create' button) THEN
        question <- user's input from ‘question’ textbook
```

```
                // for an increasing value of 1, take the answer from the relevant textbox and place
                it into a corresponding variable
                FOR x <- 0 to answerOptions DO
                        answer(x) <- user's input from 'Answer (x)' textbox
                ENDFOR
                newQuestionID <- (Database: Quiz questionID from (length(questionID Field))+1)
                questionType <- 'Order Answers'
                // appends all answers into one array
                answer <- arraySize(x)
                FOR x <- 0 to answerOptions DO
                        answer.append(answer(x))
                ENDFOR
                timeLimit <- user's input from 'Time Limit' textbox
                characterLimit <- None
                background <- user's input from background checkboxes
                // creates new instance of Question class
                newQuestion <- new Question(newQuestionID, questionType, question,
                answerOptions, answer, timeLimit, characterLimit, background)
                // add new question into Question database
                Database: Question – Add newQuestion as a record
        ENDIF
endprocedure
```



Links to success criteria -

- Quiz Creation –

- Administrator accounts should have the ability to create a quiz by adding, editing or removing different questions.

- Teachers should be able to set questions and time limits.

PRE-CREATED QUIZZES:

For the pre-created quizzes that the program will provide to users, I shall create quizzes for a few core / popular subjects, with pre-prepared questions and other settings. However, by creating set questions for a quiz, this significantly limits the utility of the function, and therefore has limited value to the user. Therefore, I have decided that, for subjects with mathematical aspects which make use of general equations, I will be able to create questions of a pre-determined format, but with randomised values within the question to then consequently produce a corresponding answer. Although values can be randomised, this still may become repetitive and potentially boring for a user. Perhaps then, I should create a range of question formats, which can make use of randomised values. The issue then raised, is the one that certain randomised values may significantly and unnecessarily increase the difficulty of the question. On the other hand, since I shall be working only with integers, it may become overly easy for the user at certain levels. In addition, for subjects without a mathematical aspect, such as English or modern languages, this form of questioning would be inappropriate and therefore would require the need to create a long list of different questions manually, and store them together to allow them to be randomised when playing a quiz.
Having considered all these problems and solutions, along with the designated time frame and complexity level, I have concluded that it would be best to create various question formats, which make use of random integer values, and using an equation stored in the program, calculates and returns the correct answer to the question to the user, and checks to see if their input is correct. Since admin users have the ability to create their own specific questions, it is acceptable to include none or little questions or quizzes for other non-mathematical subjects.

Maths –
(where k is a randomised value)

Area of a circle	The radius of a circle is k. Calculate the area of the circle.	$A = \pi r^2$
Volume of sphere	The radius of a sphere is k. Calculate the volume of the sphere.	$\frac{4}{3}\pi r^3$
Surface area of sphere	The radius of a sphere is k. Calculate the surface area of the sphere.	$4\pi r^2$
Volume of cone or pyramid	The base area of a cone is k_1 and the height of it is k_2. Calculate the volume of the cone.	$\frac{1}{3} \times base\ area \times height$
Area of curved surface of cone	The perpendicular height of a cone is k_1. The radius of the cone is k_2. Calculate the curved surface area of the cone.	$\pi r^2 \times slant\ height$
Arc length of sector	The radius of a circle is k_1. A sector is created from this circle of k_2 radians. Calculate the arc length of the sector.	$r\theta$ (θ in radians)
Area of sector of circle	The radius of a circle is k_1. A sector is created from this circle of k_2 radians. Calculate the area of the sector.	$\frac{1}{2}r^2\theta$ (θ in radians)
Area of a triangle	Two sides of a triangle are $k_1$and $k_2$cm respectively. The angle between these two sides is k_3°. Calculate the area of the triangle.	$\frac{1}{2}ab\sin C$

Pythagoras' Theorem	Two sides of a right-angled triangle are $k_1$and $k_2$cm respectively. Calculate the length of the hypotenuse.	$a^2 + b^2 = c^2$
Quadratic Formula	Calculate the roots for the quadratic equation $k_1x^2 + k_2x + k_3$.	$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$
Discriminant	Determine how many roots, if any the quadratic equation $k_1x^2 + k_2x + k_3$ has.	$b^2 - 4ac$
Dot Product	Two vectors **a** and **b** are equal to $k_1i + k_2j + k_3k$ and $k_4i + k_5j + k_6k$ respectively. Calculate the dot product of **a** and **b.** Calculate the size of the acute angle between **a** and **b.**	$\boldsymbol{a}.\boldsymbol{b} = a_1b_1 + a_2b_2 + a_3b_3$ $= \lvert\boldsymbol{a}\rvert\lvert\boldsymbol{b}\rvert \cos\theta$
Velocity	The initial speed of a particle is $k_1\ ms^{-1}$ and its acceleration is $k_2\ ms^{-2}$. If it takes $k_3\ s$ to reach point B from an initial point A, calculate the terminal velocity that the particle reaches.	$v = u + at$
Displacement	The initial speed of a particle is $k_1\ ms^{-1}$ and its terminal velocity is $k_2\ ms^{-1}$. If it takes $k_3\ s$ to reach point B from an initial point A, calculate the total displacement done by the particle.	$s = \frac{1}{2}(u + v)t$
Displacement	The initial speed of a particle is $k_1\ ms^{-1}$, with an acceleration of $k_2\ ms^{-2}$. If it takes $k_3\ s$ to reach point B from an initial point A, calculate the total displacement done by the particle.	$s = ut + \frac{1}{2}at^2$
Velocity	The initial speed of a particle is $k_1\ ms^{-1}$ and its acceleration is $k_2\ ms^{-2}$. If the particle travels a total of $k_3\ m$, calculate the terminal velocity that the particle reaches.	$v^2 = u^2 + 2as$

I shall break down a few of these equations into a form which is amenable by computational methods and demonstrate how it would be solved.

Quadratic Formula –

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

For values from the form $ax^2 + bx + c$, where a is not equal to 0, the quadratic formula can be used. To make it easier to work with, it can be decomposed, e.g.

function quadraticFormula(a, b, c)
…

1. b^2

   bSquared <- b*b

2. $4ac$

4ac <- 4*a*c

3. $b^2 - 4ac$

discriminant <- bSquared – 4ac

4. $\sqrt{b^2 - 4ac}$

rootDiscriminant <- sqrt(discriminant)

5. $-b \pm \sqrt{b^2 - 4ac}$

plusNumerator <- (-b) + rootDiscriminant
minusNumerator <- (-b) - rootDiscriminant

6. $\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$

higherAnswer <- plusNumerator / 2a
lowerAnswer <- minusNumerator / 2a

…
return higherAnswer
return lowerAnswer
endfunction

Although this function will initially use a considerable amount of space, due to the excessive use of variables, since it is a function, it will all be wiped after the function is concluded, therefore not permanently taking up much space for the program.

Dot Product –
$\boldsymbol{a}.\boldsymbol{b} = a_1b_1 + a_2b_2 + a_3b_3 = |\boldsymbol{a}||\boldsymbol{b}| \cos\theta$

For vectors $\boldsymbol{a} = a_1i + a_2j + a_3k$ and $\boldsymbol{b} = b_1i + b_2j + b_3k$, the dot product equation can be used. Another version of the dot product equation can also be used to calculate the size of the acute angle between the vectors **a** and **b**.

1. $a_1b_1 + a_2b_2 + a_3b_3$

// for a and b in the form [x(i), y(j), z(k)]
function dotProduct(a, b)

answer <- a[0]*b[0] + a[1]*b[1] + a[2]*b[2]
return answer

endfunction

2. $|\boldsymbol{a}|$

```
function modulus(a)

    squareValues <- a[0]*a[0] + a[1]*a[1] + a[2]*a[2]
    modA <- sqrt(squareValues)
    return modA

endfunction
```

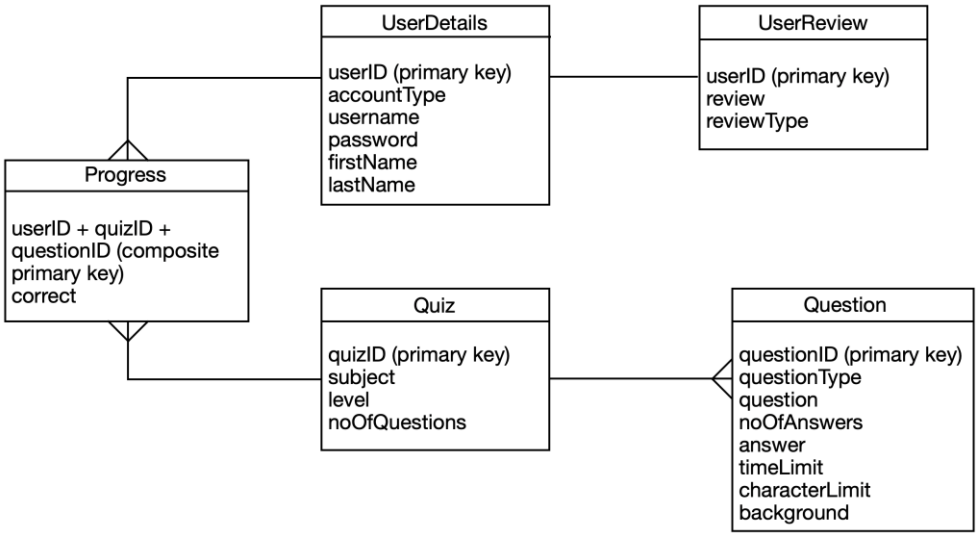
3. $|\boldsymbol{a}||\boldsymbol{b}| \cos \theta$

```
function angleBetweenVectors

    answer <- dotProductAnswer(a, b)
    modA <- modulus(a)
    modB <- modulus(b)
    cosTheta <- (answer / modA) / modB
    // from importing math library
    radianAnswer <- math.acos (cosTheta)
    degreeAnswer <- math.degrees (math.acos (cosTheta))
    // since the dot product always calculates the acute angle from the vectors, it is unnecessary
    to modify and add values of cos θ

    return radianAnswer
    return degreeAnswer

endfunction
```

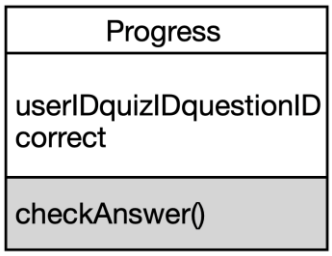
Links to success criteria –

- Pre-created quizzes –
  - Random questions on an exclusive topic or subject shall be created in the form of a quiz of which teachers and students have access to.
  - Questions with a mathematical aspect (of a pre-determined format) shall randomise numbers and consequently calculate answers using equations which shall be incorporated into the program.

PLAYING A QUIZ / PROGRESS TRACKING –
All progress of students will be stored in a database. Since the recorded progress will relate to both the user and the quiz and questions, it is best to combine all the tables mentioned into an individual database, in which relevant relationships can be maintained.

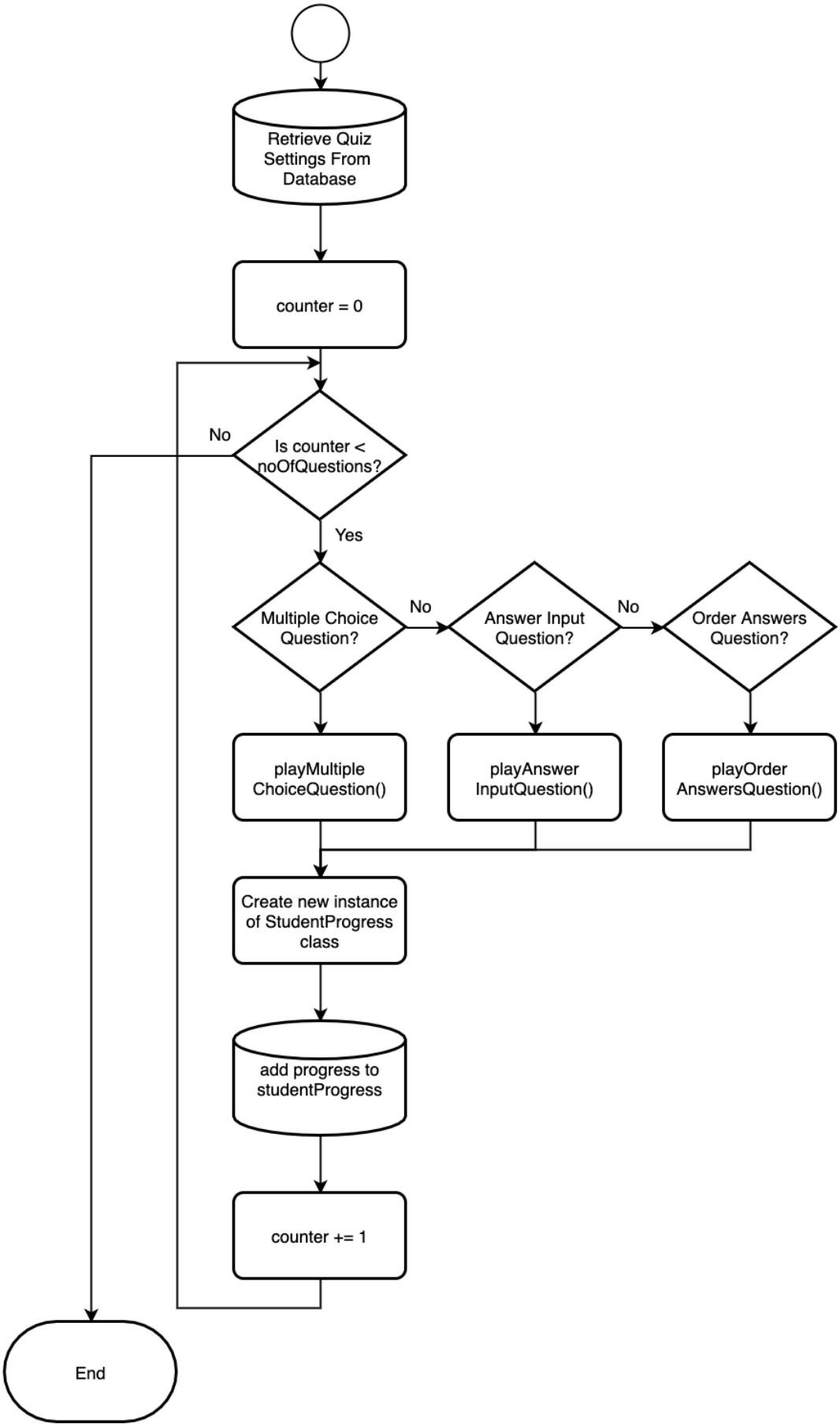


The progress of the students can be recorded in a class.



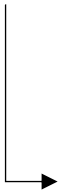
To create the class –
Pseudocode:

```
// declare class named Progress and create its attributes
class Progress
        private userIDquizIDquestionID
        private correct

// create constructor for Progress class
        public procedure new(newUserIDQuizIDQuestionID, newCorrect)
                quizID = newUserIDQuizIDQuestionID
                subject = newCorrect
        endprocedure

// encapsulation of the class
                public procedure setUserIDQuizIDQuestionID (newUserIDQuizIDQuestionID)
                        userIDquizIDquestionID = newUserIDQuizIDQuestionID
                endprocedure

                public procedure setSubject(newSubject)
                        subject = newSubject
                endprocedure
end class
```


Retrieve Quiz Settings From Database
counter = 0
Is counter < noOfQuestions?
No
Yes
Multiple Choice Question?
No
Answer Input Question?
No
Order Answers Question?
playMultiple ChoiceQuestion()
playAnswer InputQuestion()
playOrder AnswersQuestion()
Create new instance of StudentProgress class
add progress to studentProgress
counter += 1
End

To play a quiz:
Pseudocode –

```
IF (Event -> user presses 'Play' button) THEN
        function playQuiz(quizID)
ENDIF

function playQuiz(quizID)
        // ...using Quiz and Question class and tables from database
        // Retrieve quiz settings from database QuizInfo
        subject <- Database: Corresponding subject from quizID
        level <- Database: Corresponding level from quizID
        noOfQuestions <- Database: Corresponding noOfQuestions from quizID
        points <- 0

        // …by importing sqlite3
        SELECT quizID
        FROM Quiz
        INNER JOIN Question
        ON Quiz.quizID = Question.questionID

        next <- True
        // declare a counter variable that determines what question it is
        currentQuestionCounter <- 1
        // determine whether user would like to continue to next question
        WHILE next = True and currentQuestionCounter < noOfQuestions DO
                next <- False
                Load Answer Question Interface
                function retrieveQuestion(quizID, noOfQuestions)
```



```
function retrieveQuestion(quizID, noOfQuestions)
        // joins quizID and questionID column into one table (for easy reference)
        SELECT questionID
        FROM QuizQuestion  // newly joined table
        WHERE QuizQuestion.quizID = quizID  // …that was passed as a parameter

        // all questionID's for relevant quizID are stored in an array
        questionIDList <- arraySize(noOfQuestions)
        FOR each in SQL SELECT STATEMENT DO
                questionIDList.append(questionID)
                next each
        ENDFOR
        return questionIDList
endprocedure
```

```
                // checks the type of question and calls appropriate function
                currentQuestionID <- questionIDList[currentQuestionCounter]
                IF (questionType From Database of currentQuestionID) = 'Multiple Choice' THEN
                        answerMultipleChoiceQuestion(questionID)
                ELSE IF (questionType From Database of currentQuestionID) = 'Answer Input' THEN
```

```
                answerAnswerInputQuestion(questionID)
            ELSE IF (questionType From Database of currentQuestionID) = 'Order Answers' THEN
                answerOrderAnswersQuestion(questionID)
            ENDIF
            checkAnswer(currentQuestionID, userAnswer) // shown below
            displayAnswerAndPoints(checkAnswer(currentQuestionID, userAnswer), points)
            // shown below
            next <- True
            currentQuestionCounter += 1
        ENDWHILE
```

To check an answer:
Pseudocode –

```
function checkAnswer(currentQuestionID, userAnswer)
    IF (questionType From Database of currentQuestionID) = 'Multiple Choice' THEN
        correctAnswer <- answer[1] From Database of currentQuestionID
    ELSE IF (questionType From Database of currentQuestionID) = 'Answer Input' THEN
        correctAnswer <- answer From Database of currentQuestionID
    ELSE IF (questionType From Database of currentQuestionID) = 'Order Answers' THEN
        correctAnswer <- answer From Database of currentQuestionID
    ENDIF

    IF userAnswer = correctAnswer THEN
        return True
    ELSE
        Return False
    ENDIF
endfunction


function displayAnswerAndPoints(checkAnswer(currentQuestionID, userAnswer), points)
    IF checkAnswer(currentQuestionID, userAnswer) = True THEN
        DISPLAY Correct
        points += 1000
    ELSE
        DISPLAY Incorrect

    return points
    ENDIF
endfunction
```

This function can be extended to increase the factor of points for achieving an answer streak:

```
function displayAnswerAndPoints(checkAnswer(currentQuestionID, userAnswer), points)
        addPoints <- 1000
        multiplier <- 1
        answer <- (Database QuizInfo: answer field for corresponding questionID)
        IF userAnswer = answer THEN
                DISPLAY correct
                IF answerStreak() = True THEN
                x += 0.1
                points += (addPoints * x)
        ELSE
                DISPLAY incorrect
                x <- 1
        ENDIF

        return points
endfunction


function answerStreak
        IF correct = True
                return True
        ELSE
                return False
        ENDIF
endfunction
```

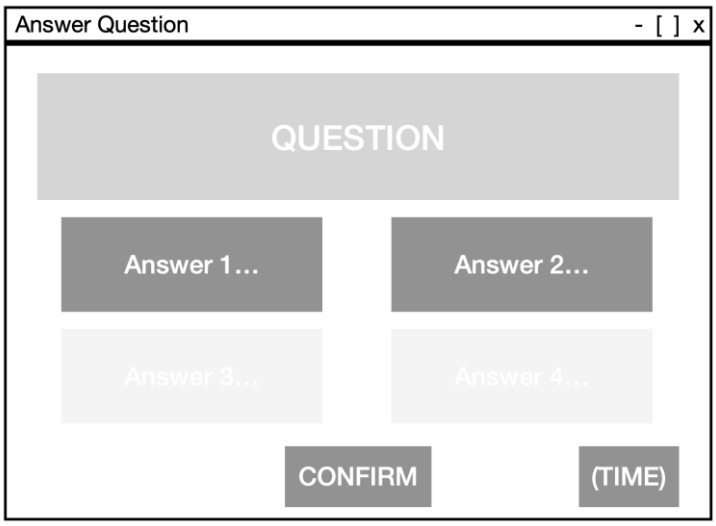
Links to success criteria –

- Points system –
  - Students will earn points for answering a question and shall accumulate them over a quiz.
  - Students will earn an incremental number of points based on an answer streak that develops over a quiz.

```
function answerMultipleChoiceQuestion(questionID)
    // Retrieve settings for Question From Database
    question <- Database: Corresponding question from questionID
    noOfAnswers <- Database: Corresponding noOfAnswers from questionID
    answer <- Database: Corresponding answer from questionID
    timeLimit <- Database: Corresponding timeLimit from questionID
    background <- Database: Corresponding background from questionID

    // randomise order of displayed answers
    // ...by importing random
    random.shuffle(answer)
    DISPLAY question as label
    FOR each in answer DO
        DISPLAY answer[each] as button
        next each
    ENDFOR
    // ...by importing time
    DISPLAY countdown() as label
```

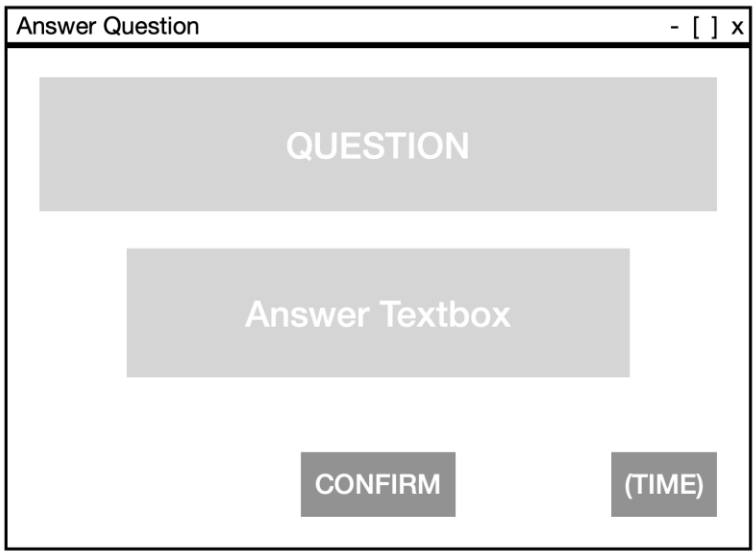
```
function countdown(timeLimit)
    WHILE timeLimit > 0 DO
        DISPLAY timeLimit
        timeLimit -= 1
        time.sleep(1)
    ENDWHILE
endfunction
```

```
    WHILE timeLimit > 0 DO
        IF (Event -> User presses 'Confirm' button) THEN
            userAnswer <- Chosen Answer
        ELSE
            userAnswer <- None
        ENDIF
    ENDWHILE
    return userAnswer
```



This is the interface for a single user playing a quiz for a multiple choice question. When the user presses on an answer option, it will become highlighted to show the user it has been pressed. The user will then have to press the 'Confirm' button to confirm their answer to prevent mis-clicks. This is useful in my program since the points are not allocated based on the speed of the user like it is on Kahoot. The time is also shown as a countdown to allow the user to see how long they have left. The background shall be incorporated within the coding of the program.

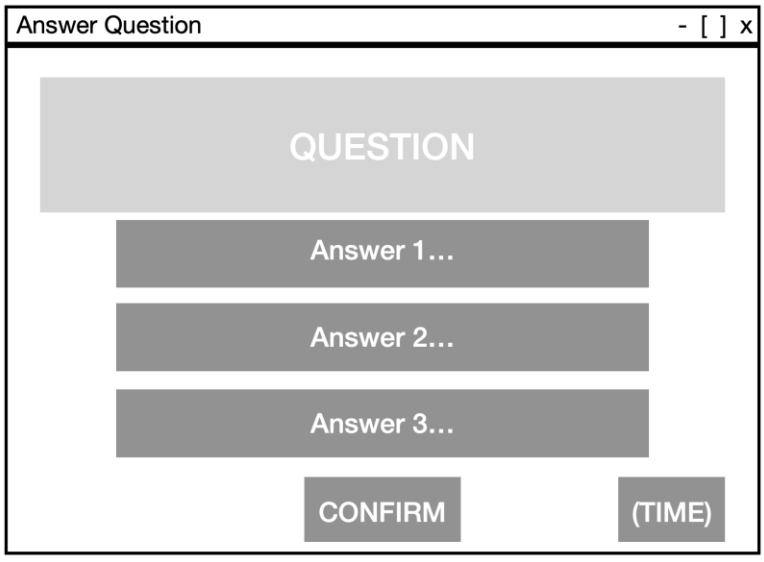
```
function answerAnswerInputQuestion(questionID)
        // Retrieve settings for Question From Database
        question <- Database: Corresponding question from questionID
        noOfAnswers <- Database: Corresponding noOfAnswers from questionID
        answer <- Database: Corresponding answer from questionID
        timeLimit <- Database: Corresponding timeLimit from questionID
        characterLimit <- Database: Corresponding characterLimit from questionID
        background <- Database: Corresponding background from questionID

        DISPLAY question as label
        FOR each in answer DO
                DISPLAY answer[each] as button
                next each
        ENDFOR
        DISPLAY countdown() as label

        WHILE timeLimit > 0 DO
                IF (Event -> User presses 'Confirm' button) AND length(Inputted Answer) <
                characterLimit THEN
                        userAnswer <- Chosen Answer
                ELSE IF  length(Inputted Answer) > characterLimit THEN
                        DISPLAY Character Limit Exceeded
                        userAnswer <- None
                ELSE
                        userAnswer <- None
                ENDIF
        ENDWHILE
        return userAnswer
```



This is the interface for a single user playing a quiz for an answer input question. Here, the question is displayed for the user, and they are required to type in an answer themselves within the time and character limit. The user will then have to press the 'Confirm' button to confirm their answer to allow them to check their answer. The time is also shown. The background shall be incorporated within the coding of the program.

```
function answerOrderAnswersQuestion(questionID)
        // Retrieve settings for Question From Database
        question <- Database: Corresponding question from questionID
        noOfAnswers <- Database: Corresponding noOfAnswers from questionID
        answer <- Database: Corresponding answer from questionID
        timeLimit <- Database: Corresponding timeLimit from questionID
        background <- Database: Corresponding background from questionID

        // shuffle list of answers
        random.shuffle(answer)

        DISPLAY question as label
        FOR each in answer DO
                DISPLAY answer[each] as button
                next each
        ENDFOR
        DISPLAY countdown() as label

        // If textbox is considered as a general container and the answers as moveable boxes...
        IF (Event -> User moves answer(x) box to container with answer(y)) THEN
                // creates a temporary variable to successfully swap the two answers
                temp <- answer(y)
                answer(y) <- answer(x)
                answer(x) <- temp

        WHILE timeLimit > 0 DO
                IF (Event -> User presses 'Confirm' button) THEN
                        userAnswer <- arraySize(noOfAnswers)
                        FOR x <- 0 to noOfAnswers DO
                                userAnswer.append(answer(z))
                        ENDFOR
                ELSE
                        userAnswer <- None
                ENDIF
        ENDWHILE
        return userAnswer
```

This is the interface for a single user playing a quiz for an order answers question. Here, the question is displayed for the user, and they are required to order the answer within the time limit, by dragging the answers across. The user will then have to press the 'Confirm' button to confirm their answer to allow them to check their answer. The time is also shown. The background shall be incorporated within the coding of the program.

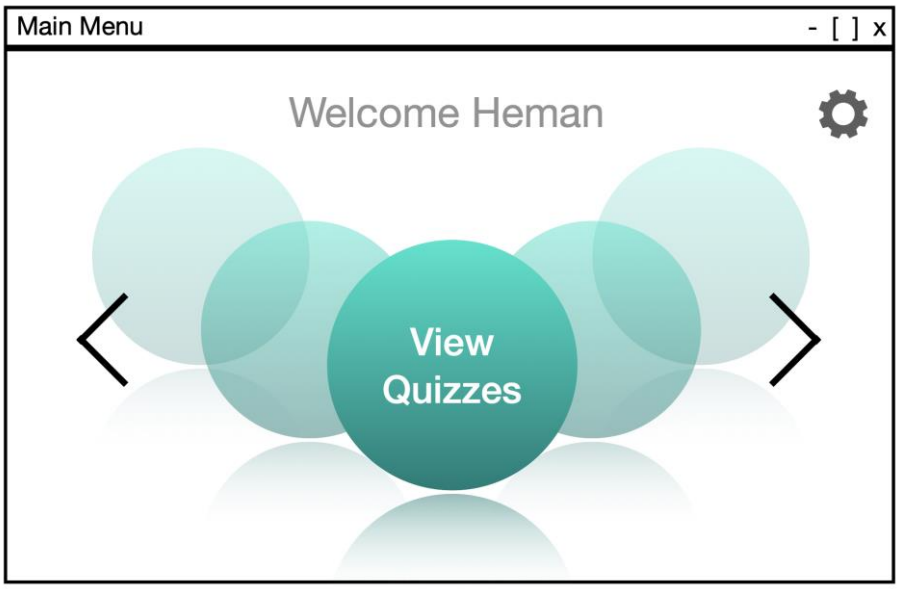


Links to success criteria –

- Quiz
  - Users are able to partake in quizzes

Usability Features –

The usability features that I have considered make sure that the program is easy to use for any type of user. On the interfaces that I have designed, all buttons and labels are clearly visible, maximising the space of the window, with clear indications of the purpose of each. Backgrounds will be available for the user when answering questions to improve the overall aesthetic of the program as an alternative to personal imported images.



The program is fairly self-explanatory to use and no past experience is required to work your way around it. Although it may take some time for the users to get used to the functions and other privileges available. In this general main menu page, a modern and clean interface is provided which welcomes you to the program, easily allows you to browse through and select different options including some settings for the program, i.e. where admin users get notifications, etc. Although the final design may have an altogether different or improved colour scheme, the text will be large and visible through the chosen colours.

The usage of the program needs to be easy, which involves a simplistic nature for the menu, preventing the possibility of the user becoming lost within the program. All different pages from the main menu will have a return option to allow the user to easily navigate back to the main menu if necessary.

Links to success criteria –

- User-friendly user interface –
  - Clear headings and sub-headings to indicate the content of each page and relevant images if needed.
  - Use of modern, user-friendly colours for pages and buttons
  - Clear labels or logos for buttons

INPUTS / PROCESSES / OUTPUTS + TESTING:

PROBLEM	INPUTS	PROCESSES	TEST	OUTPUTS	STORAGE
LOGIN	User selects button indicating whether teacher, student or parent	adminLogin /studentLogin / parentLogin function	Whether button initiates correct function	Relevant login page is brought up	-
	Username	Check if username exists Check if username is valid	Whether username is in database Whether username is valid	Username already exists Username is valid or invalid	UserDetails table within database
	Password	Check whether password has been incorrectly inputted three times Check password in database after hashPassword() Check if password is valid in validatePassword() function	Whether password hasn't been incorrectly entered three times and (once hashed) corresponds to given username Whether password is valid	'Account blocked' if inputted incorrectly three times	UserDetails table within database
	Forgot Password (- Pick Admin User)	forgotPassword() resetPassword()	Account is placed under review and sent to an admin user to change Admin users must be able to reset a password within the database	Notification is shown to relevant admin user to address this issue	-
	Create Account	createAccount()	Whether credentials are acceptable (expanded below)	Notification is shown to relevant admin user to verify the account	-
	Login	login()	Whether username and (hashed) password exist and correspond from the database with corresponding user type	Menu interface is loaded	-
	Quit	Closes the program	Page closes	-	-

ACCOUNT CREATION	Account Type	User chooses account type from drop down menu	Check if drop down menu gives correct selection	-	Stored in UserDetails table within database
	First Name	User types in first name	Whether valid or invalid	Invalid if not valid	Stored in UserDetails table within database
	Last Name	User types in last name	Whether valid or invalid	Invalid if not valid	Stored in UserDetails table within database
	Username	User types in new username	Checks database to see whether inputted username already exists Check if username is valid	('Username Taken') if taken Invalid if not valid	Stored in UserDetails table within database
	Password	User types in new password passwordCheck() hashPassword()	Whether password is valid, i.e strong enough to then be hashed	'Password not strong enough' if strongPassword returns False	Stored in UserDetails table within database
	Three incorrect password attempts	blockUser() unblockUser()	Whether user loses facility to access account Whether facilities are restored when admin user unblocks an account	'Account Blocked. Go see an admin user'	Stored in UserDetails table within database
	Admin User to Verify	User chooses admin account from drop down menu	Account is placed under review and sent to an admin user	Notification is shown to relevant admin user to verify the account	-
	Register	createAccount() - retrieve username and password from interface - check existence of username and strength of password - add new instance of User class and add it to the database Admin must verify user	Whether credentials are acceptable, i.e. username doesn't already exist and is valid; password is valid and strong enough; first name and last name are valid	Unsuccessful if any of username, password, first name, last name are invalid Unverified if not yet verified	New record of UserDetails is created and stored in the database if successful

			Whether users can access account after verification		
	userID	Uniquely incremented for each user	Whether userID is unique for each user	-	Stored in UserDetails table within database
	accountType	Assigned to a user from the create account page	-	-	Stored in UserDetails table within database
	review	True when a user has created a new account, forgotten their password, or has been blocked False otherwise	-	Restrict access if review is True	Stored in UserDetails table within database
	reviewType	Describes the type of review Should implement the correct function accordingly verifyUser() unblockUser() resetPasssword()	Whether review statues changes Whether user gains access once verified Whether password is reset Whether access is restored when unblocked	'Account Blocked' 'Ask admin user to verify your account'	Stored in UserDetails table within database
	Quit	Closes the program	Page closes	-	-
QUESTION CREATION	User selects button indicating whether multiple choice, answer input or order answers	createMulitpleChoiceQuestion(), createAnswerInputQuestion(), createOrderAnswersQuestion()	-	Relevant create question page is brought up	-
	Question	User enters a question	-	Displayed when answering question	Stored in Question table within database
	Answer	User enters an answer	For an answer input question – the length answer must be greater than the character limit	Displayed when answering question	Stored in Question table within database
	Time Limit	User enters a time limit in seconds	Whether valid, i.e. a number between 1 and 60	Displayed when answering question	Stored in Question table within database
	Character Limit	User enters a character limit	For an answer input question –	-	Stored in Question table

			the length answer must be greater than the character limit Whether valid, i.e. a number between 1 and 100		within database
	Background	User picks one of three background choice	Whether correct background appears for each question	Displayed when answering question	Stored in Question table within database
PLAYING A QUIZ	- Subject - Level - Number of questions	These settings are retrieved from the database using the relevant questionID	-	-	Question table within database
	Points	Initially set at 0, increases for every correct answer. More points are awarded for answer streaks	Points are awarded as multiples of 1000 with the multiplier beginning at 1 and increasing by 0.1 for every answer streak	Displayed after each question and at end of quiz	Stored in Progress table within database
	Play	answerMulitpleChoiceQuestion(), answerAnswerInputQuestion(), answerOrderAnswersQuestion()	Redirects to separate function for each question type which is checked when questions are retrieved from database	Loads Answer Question Interface	-
	Confirm	Takes inputted answer and verifies it against checkAnswer()	Whether the answer is correct or not – determined from whether the function returns a True, for correct, or False, for incorrect, value.	Displays whether user was correct or incorrect	-
PROGRESS TRACKING	Correct answers	Determines whether a user has inputted or selected the correct answer	Compare input against database	Displays progress to teacher and student	Stored in Progress table within database
	Time	Determines when a user has played a certain quiz	-	-	Stored in Progress table within database

Test Strategies:

- Black Box Testing - where the software is tested without the testers being aware of the internal structure of the software and can be carried out both within the company and by end-users. The test plan traces through inputs and outputs within the software.
    - This will be used in the post development phase for when I test the program interface as when I run the interface, the code may contain some bugs or errors, so the outcome will be unknown. When testing the program in this way, if something appears that was unintended, or an intended part of it is otherwise missing, there is a clear indication of which part of the code needs improvement. This will be done during the development stage of the program.

- White Box Testing - where the test plan is based on the internal structure of the program, all of the possible routes through the program are tested.
    - This will be used during the iterative development of the program to test processes that are not immediately obvious to the end user, such as transactions within the database, creating or manipulating instances of classes, etc. This allows me to focus not only on the ostensible aspect of the program, as black box testing restricts me to, but also identify any bugs or errors within the hidden processes inside the code.

- Alpha Testing - carried out in-house by the software development teams within the company, bugs are pinpointed and fixed.
    - This post development method of testing will be used by my client and her colleagues after the other forms of testing have been completed. Any errors identified or tweaks required can then be passed on to me for examination and resolution

Test Plans:
Create Account Function:

Test Number	Test	Input Data	Expected Outcome
SELECTING TYPE OF ACCOUNT			
1	User can only select one from Teacher, Student or Parent	Select ‘Teacher’, ‘Student’ or ‘Parent’	Drop-down box displays selected option and passes it on to next function
FIRST NAME / LAST NAME ENTRY			
2	Field empty	‘’	Error message
3	Contains only white space	‘ ‘	Error message
4	Contains invalid character: number	‘Heman1’ ‘Seegolam2’	Error message
5	Contains invalid character: symbol	‘Heman!’ ‘See-golam’	Error message
6	Contains invalid space character	‘He man’ ‘See golam’	Error message
USERNAME ENTRY			
7	Contains invalid space character	‘heman 123’	Error message
8	Invalid username	‘username’	Error message

9	Correct username	‘hemanseego01’	Creates account if password is valid
10	Taken username	‘hemanseego01’	Error message
PASSWORD ENTRY			
11	Field empty	‘’	Error message
12	Contains only white space	‘ ‘	Error message
13	Short password	‘Pass1!’	Error message
14	No upper/lower case	‘password1!’	Error message
15	No number	‘Pass-word’	Error message
16	No symbol	‘Password1’	Error message
17	Correct password	‘Pass-Word123’	Creates account if username is valid
SELECTING ADMIN USER			
18	User can only select one from all admin users	Select ‘AdminUsername1’, ‘AdminUsername2’,…	Drop-down box displays selected option and passes it on to next function

Logging in:

Test Number	Test	Input Data	Expected Outcome
1	Whether username exists	‘hemanseego01’, ‘hemanseego02’	Checks matching password, Error message
2	Incorrect password	‘incorrectPassword’	Error message
3	Correct password	‘Pass-Word123’	Access granted to main menu
4	Whether access denied when account is blocked	‘hemanseego1’ as username; ‘Pass-Word123’ as password when account is blocked	Access denied; Error message
5	Whether account has been blocked	Three consecutive incorrect log ins	Message stating account has been blocked
6	Whether account has been verified	Log in from newly unverified created user	Message stating account hasn’t been verified

Question Creation:

Test Number	Test	Input Data	Expected Outcome
TIME LIMIT ENTRY			
1	Invalid value: symbol, space, letter	‘@’, ‘1 2‘ ‘a’	Error message
2	Valid integer value	‘30’	Successfully passes it onto next function
CHARACTER LIMIT ENTRY			
3	Invalid value: symbol, space, letter	‘£’, ‘3 0‘	Error message

		‘C’	
4	Valid integer value	‘50’	Successfully passes it onto next function
BACKGROUND ENTRY			
5	Invalid value: symbol, space, letter	‘%’, ‘ ‘ ‘g’	Error message
6	Valid integer value between 1 and 3	‘2’	Successfully passes it onto next function

3.3 Developing the solution

Database:
My program makes use of one crucial database. This database contains all account information that needs to be stored to be used for use within the program. This includes tables for the user details, the status of the user review, the quizzes, the questions and the progress of students.

To create the database, I have used python by importing the sqlite3 library and consequently making use of SQL statements to form each table, assign the different fields and their data type and identified the primary key.

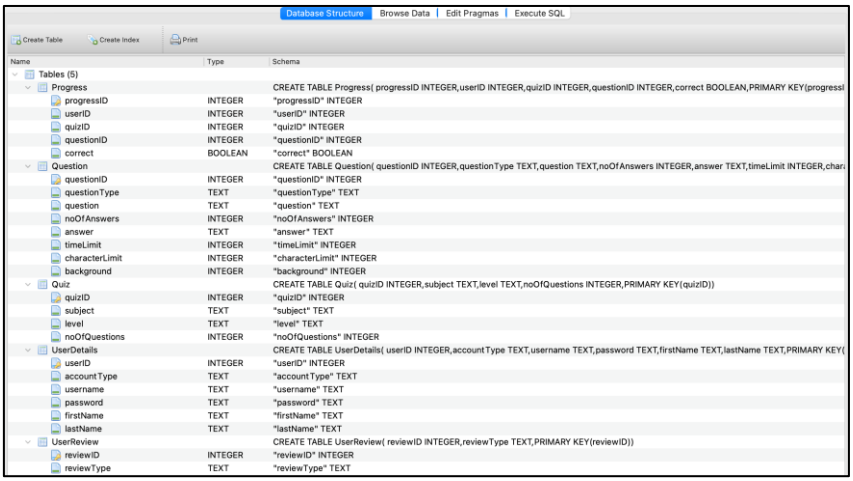
```
## CREATE ACCOUNTINFO DATABASE

import sqlite3

## Initiates the database with the name "AccountInfo.db" along with its cursor
conn = sqlite3.connect('AccountInfo.db')
c = conn.cursor()

## Creates a table for all user details
c.execute(""" CREATE TABLE UserDetails(
            userID INTEGER,
            accountType TEXT,
            username TEXT,
            password TEXT,
            firstName TEXT,
            lastName TEXT,
            PRIMARY KEY(userID)) """)

## Creates a table for the status of the review of users
c.execute(""" CREATE TABLE UserReview(
            reviewID INTEGER,
            reviewType TEXT,
            PRIMARY KEY(reviewID))""")

## Creates a table for all quizzes
c.execute(""" CREATE TABLE Quiz(
            quizID INTEGER,
            subject TEXT,
            level TEXT,
            noOfQuestions INTEGER,
            PRIMARY KEY(quizID))""")
```

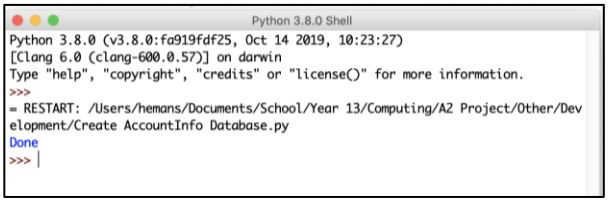
```
## Creates a table for all questions
c.execute(""" CREATE TABLE Question(
            questionID INTEGER,
            questionType TEXT,
            question TEXT,
            noOfAnswers INTEGER,
            answer TEXT,
            timeLimit INTEGER,
            characterLimit INTEGER,
            background INTEGER,
            PRIMARY KEY(questionID)) """)

## Creates a table for the progress of the students
c.execute(""" CREATE TABLE Progress(
            progressID INTEGER,
            userID INTEGER,
            quizID INTEGER,
            questionID INTEGER,
            correct BOOLEAN,
            PRIMARY KEY(progressID)) """)

print('Done')

## Closes connection with the database
conn.commit()
conn.close()
```

By importing the sqlite3 library in python, it permitted the use of SQL statements to allow functionality of databases: a critical aspect for efficient handling of data in my program. The database allows my program to store all necessary data in a separate file that can be accessed and modified accordingly to fulfil different aspects of the program. This means that all relevant data doesn't have to manually be inputted at the start of the program which reduces the complexity of the program for the end user and makes it faster for the user to use. It also keeps data private from users which improves the security of the program and abides by the Data Protection Act 2018 – [information must be] *handled in a way that ensures appropriate security, including protection against unlawful or unauthorised processing, access, loss, destruction or damage* (https://www.gov.uk/data-protection)

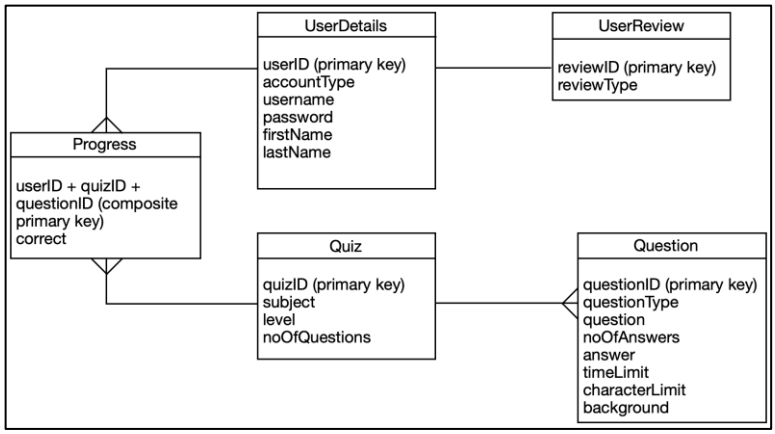
After importing the sqlite3 library, it was important to establish a connection between the library and the relevant database: the AccountInfo Database. This process was assigned to the variable conn, which then allowed a cursor to be created (named c)

to make use of the library. This then permitted SQL statements to be made to create the relevant tables. I created a UserDetails table to include a user's user ID, account type (Teacher, Student or Parent), username, password (which will be hashed for extra security), first name, last name and reviewID. I also identified the user ID as the primary key since it's unique. I created a UserReview table to include the review ID and the review type (which will include 'None', 'ForgotPassword', 'Blocked Account', etc.). In this table, the review ID is the primary key of the table. I created a Quiz table to include a quiz's ID, subject, level (i.e. A-Level, GCSE, etc.) and number of questions, identifying the quiz ID as the unique primary key. I created a Question table to include a question's ID, type (Multiple Choice, Answer Options or Order Answers), number of answers (relevant if multiple choice or order answer question to aid with the layout of the page), answer(s), the time limit, the character limit and a chosen background (integer value of 1, 2 or 3 taken from a check box when user creates question. The questionID is the unique primary key. Finally, I created a Progress table to record the progress of students, acquired from the quizzes that they take. This table includes the progress ID (primary key), the user ID, the quiz ID and question ID. It also contains a 'correct' field – a boolean value indicating whether a question has been answered correctly or incorrectly. To end this code, I printed out 'Done' to ensure the whole process had completed and thereby closed the connection between the database and sqlite3. Since the file AccountInfo.db had not previously existed in the given directory, the code created a new database file with the relevant data. This only has to be done once to create the database.





The database is set in the following format:



This demonstrates the relationship between the five entities. UserDetails has a one-to-one relationship with UserReview since for every one user, there will be a corresponding review. Alternatively, Quiz has a one-to-many relationship with Question since in every quiz there a various questions. Likewise, UserDetails and Quiz have a one-to-many relationship with Progress since for the progress stored, there is one related quiz and one related user. There may be various incidents of progress of the same user and perhaps even the same quiz. The use of a composite primary key here allows it to be unique and therefore differentiate between them. By creating relationships between the entities, the data hence becomes related and allows the maintenance of referential integrity within the database.

To simplify the use of the database functions within the program, I created a class for them. This class simplifies the process of viewing the database, adding to the database, updating the database and deleting from the database. By doing this, I have overall minimised the amount of code by making these important functions reusable, therefore saving me from rewriting the same code multiple times and making the program less prone to errors and allows for easier maintenance.

```
## DATABASE CLASS

import sqlite3

## Initiates the database with the name "AccountInfo.db" along with its cursor
    conn = sqlite3.connect('AccountInfo.db')
    c = conn.cursor()

## Creates a parent class for all database functions
class DatabaseFunction:
    ## Assigns attributes for database functions
    def __init__(self, newType):
        self.__type = newType

    ## Encapsulation of DatabaseFunction Class (Set Methods)
    def setType(self, newType):
        self.__type = newType

    ## Encapsulation of DatabaseFunction Class (Get Methods)
    def getType(self):
        return self.__type

## Creates a new instance of the DatabaseFunction class for the different functions
addToDatabase = DatabaseFunction('Add')
updateDatabase = DatabaseFunction('Update')
deleteFromDatabase = DatabaseFunction('Delete')
viewFromDatabase = DatabaseFunction('View')
```

The class firstly identifies the type of the relevant database function. They are recognised simply by 'Add', 'View', 'Update' and 'Delete'. To begin, sqlite3 had to be imported to allow SQL functions to be used and a connection had to be established with the AccountInfo database. This is for later when creating the functions for these relevant class objects. To create the class, I constructed and encapsulated the 'type' attribute, allowing each object of the DatabaseFunction class to have its own 'type' attribute and allowing it to be set and retrieved when necessary.

ADD TO DATABASE FUNCTION:

```
def addRecordToDatabase(record, table):
    c.execute("INSERT INTO table VALUES record")
```

Initially, for this function, I wrote the code above. However, I immediately faced some issues: the python variables were not recognised within the SQL statement. To test it, I used this code:

```
def addRecordToDatabase(record, table):
    c.execute("INSERT INTO table VALUES record")

heman = [1, 'Admin', 'hemanseego01', 'Pass-word123', 'Heman', 'Seegolam']
table = 'UserDetails'

addRecordToDatabase(heman, table)

conn.commit()
conn.close()
```

Which met this error:

```
    c.execute("INSERT INTO table VALUES record")
sqlite3.OperationalError: near "table": syntax error
>>>
```

This was because instead of taking the python variables 'table' and 'record', it was searching the database for a table named 'table'. I went on to try this alternative:

```
def addRecordToDatabase(record, table):
    c.execute("INSERT INTO" + table + "VALUES" + record)

heman = [1, 'Admin', 'hemanseego01', 'Pass-word123', 'Heman', 'Seegolam']
table = 'UserDetails'

addRecordToDatabase(heman, table)

conn.commit()
conn.close()
```

Which met this error:

```
    c.execute("INSERT INTO" + table + "VALUES" + record)
TypeError: can only concatenate str (not "list") to str
>>>
```

Firstly, since concatenation in python joins together two pieces of data without spaces, the SQL statement read 'INSERT INTOtableVALUESrecord'.
Secondly, it was unable to concatenate a list which contained data types which were not strings – here the userID field of 1. Also, when trying to add a list to the database, the syntax isn't immediately compatible with the SQL syntax since it has the [ ] brackets around the values which is not recognisable by SQL. Hence, I had to find a way to remove the square brackets from the list.

I then tried:

```
def addRecordToDatabase(record, table):
    recordString = ','.join(record)
    # print(recordString)
    c.execute("INSERT INTO " + table + " VALUES " + recordString)

heman = [1, 'Admin', 'hemanseego01', 'Pass-word123', 'Heman', 'Seegolam']
table = 'UserDetails'

addRecordToDatabase(heman, table)

conn.commit()
conn.close()
```

Here, I inserted spaces inside the SQL statement to retain the appropriate format even after concatenation. I also tried to make another variable to hold all the values of the record list but as a simple string – this was to get rid of the square brackets when passing it on to the SQL statement.

It met this error:

```
    recordString = ','.join(record)
TypeError: sequence item 0: expected str instance, int found
>>>
```

When it tried to concatenate the contents of the record list into a separate variable, it immediately had an issue with the integer value placed in record[0].

To combat this I wrote:

```
def addRecordToDatabase(record, table):
    print (record)
    for each in range(len(record)):
        record[each] = str(record[each])
    print (record)
    recordString = ','.join(record)
    print(recordString)
    #c.execute("INSERT INTO " + table + " VALUES " + recordString)

heman = [1, 'Admin', 'hemanseego01', 'Pass-word123', 'Heman', 'Seegolam']
table = 'UserDetails'

addRecordToDatabase(heman, table)

conn.commit()
conn.close()
```

Here, it seemed that all values had to be strings for it to properly be added to the database. As such, I converted all values in record into strings using a for loop. I had to use for each in range(len(record)) as opposed to just for each in record since this error was produced:

```
    record[each] = str(record[each])
TypeError: list indices must be integers or slices, not str
>>>
```

After doing this, I joined all values from the new record (all values now strings) into one string variable

This produced:

```
Python 3.8.0 Shell
Python 3.8.0 (v3.8.0:fa919fdf25, Oct 14 2019, 10:23:27)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: /Users/hemans/Documents/School/Year 13/Computing/A2 Project/Other/Development
/Database Class Code/Add to Database/3.py
[1, 'Admin', 'hemanseego01', 'Pass-word123', 'Heman', 'Seegolam']
['1', 'Admin', 'hemanseego01', 'Pass-word123', 'Heman', 'Seegolam']
1,Admin,hemanseego01,Pass-word123,Heman,Seegolam
>>>
```

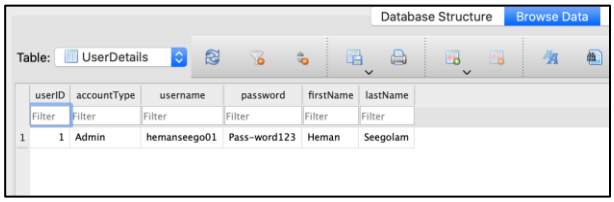
However, when then proceeding to execute the SQL statement and add it into the database, there was again the issue of data types since where the database table expected an integer, boolean, etc., it received a string.

To get around the problem of data types, I investigated a different approach:

```
def addRecordToDatabase(record, table):
    sqlAdd = ("INSERT INTO " + table + " VALUES (?,?,?,?,?,?)")
    c.execute(sqlAdd, record)

heman = [1, 'Admin', 'hemanseego01', 'Pass-word123', 'Heman', 'Seegolam']
table = 'UserDetails'

addRecordToDatabase(heman, table)
print('Done')

conn.commit()
conn.close()
```

In this method, the values that are to be added to the database are generally defined in the statement, but are actually saturated later on. Here, I made an SQL statement for the insertion of the general values, and then executed it with the particular values of record.

```
Python 3.8.0 Shell
Python 3.8.0 (v3.8.0:fa919fdf25, Oct 14 2019, 10:23:27)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: /Users/hemans/Documents/School/Year 13/Computing/A2 Project/Other
/Development/Database Class Code/Add to Database/4.py
Done
>>>
```



And whilst this method initially seems to work effectively, there is actually an underlying issue. This issue concerns the general use of this addRecordToDatabase function. Since not only the UserDetails table will make use of this function, the number of general placeholders included will need to vary depending on the number of fields in the table. To demonstrate:

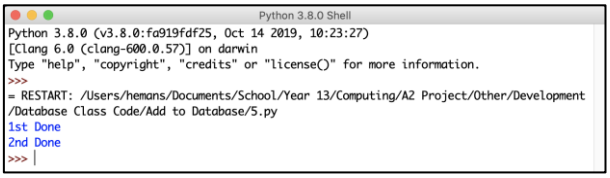
```
def addRecordToDatabase(record, table):
    sqlAdd = ("INSERT INTO " + table + " VALUES (?,?,?,?,?,?)")
    c.execute(sqlAdd, record)

#heman = [1, 'Admin', 'hemanseego01', 'Pass-word123', 'Heman', 'Seegolam']
#table = 'UserDetails'

hemanReview = [1, False, 'None']
table = 'UserReview'

addRecordToDatabase(hemanReview, table)
print('Done')
```

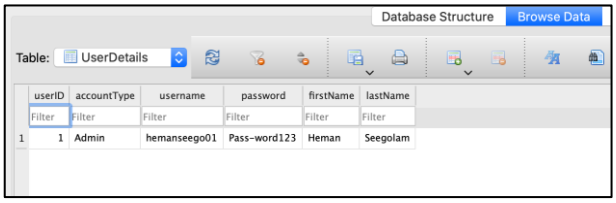
```
    c.execute(sqlAdd, record)
sqlite3.OperationalError: table UserReview has 3 columns but 6 values were supplied
>>>
```

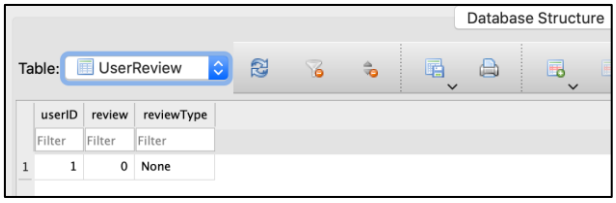
Finally, I settled with this method of approach:

```
def addRecordToDatabase(record, table):
    ## If to be added to UserDetails table
    if len(record) == 6:
        sqlAdd = ("INSERT INTO " + table + " VALUES (?,?,?,?,?,?)")
    ## If to be added to UserReview table
    elif len(record) == 3:
        sqlAdd = ("INSERT INTO " + table + " VALUES (?,?,?)")
    ## If to be added to Quiz table
    elif len(record) == 4:
        sqlAdd = ("INSERT INTO " + table + " VALUES (?,?,?,?)")
    ## If to be added to Question table
    elif len(record) == 8:
        sqlAdd = ("INSERT INTO " + table + " VALUES (?,?,?,?,?,?,?,?)")
    ## If to be added to Progress table
    elif len(record) == 2:
        sqlAdd = ("INSERT INTO " + table + " VALUES (?,?)")
    c.execute(sqlAdd, record)

heman = [1, 'Admin', 'hemanseego01', 'Pass-word123', 'Heman', 'Seegolam']
table1 = 'UserDetails'

hemanReview = [1, False, 'None']
table2 = 'UserReview'

addRecordToDatabase(heman, table1)
print('1st Done')

addRecordToDatabase(hemanReview, table2)
print('2nd Done')

conn.commit()
conn.close()
```

In this function, it first determines the length of the record list. From this, it is then able to execute the correct SQL statement for the correct table (a different number of question marks is required for the values). This method is safe against SQL injection attacks since all variables will be verified (shown later) before being added as an object of a class, which will then be used to add data to the database. For example, the names will contain only letters, the account type will be modifiable only by private methods, etc.







To produce an efficient database, all transactions must be atomic, consistent, in isolation and durable.

Atomicity requires a transaction to be processed in its entirety or not at all. To do this, I produced this code:

```
try:
    # Execute SQL command
    c.execute(sqlAdd, record)
    # Commit changes to database
    conn.commit()
except:
    # Rollback in case of error (reverts database to earlier state)
    conn.rollback()
```

In this block of code, there is a try except statement. This means that the program first attempts to run the code under try, so attempts to execute the given statement and if successful, commit the changes into the database. However, if there is any problem doing so, the program will then run the code under except and therefore rollback the database to its previous state. Here, the transaction is either done completely, with a commitment to the database, or not done at all, reverting the database back to its original state if any problems are encountered.

However, there are limitations to this method. This function accounts for mishaps within the program itself, and so if an internal problem is experienced, it reverts any changed effects. What it fails to account for is any external situations such as a power cut or computer crash, and so if a situation like this were to occur, the program would fail to revert any changes made by the code under try, since it may not reach the except part of the code. (Solution given when exploring durability below)

Consistency ensures a transaction maintains the validation and referential integrity rules between linked tables. To do this, I would have to create foreign keys between tables in my database to create a relational link between them, and therefore allowing the database to take care of the rules. This requires me to modify the code shown previously which created the database. I modified three parts of the previous code: the UserDetails, Question and Progress tables. This is because they all related to another table and should therefore have used foreign keys.

```
reviewID INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT UNIQUE,
```

```
quizID INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT UNIQUE,
```

Firstly, I extended the statement to ensure that all primary keys were integers that couldn't be zero, were unique and incremented automatically. This reduces any chances of conflicts between different records.
To identify the foreign keys within the tables and recognise which primary keys they related to:

```
## Creates a table for all user details
c.execute(""" CREATE TABLE UserDetails(
            userID INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT UNIQUE,
            accountType TEXT,
            username TEXT,
            password TEXT,
            firstName TEXT,
            lastName TEXT,
            reviewID INTEGER,
            FOREIGN KEY(reviewID) REFERENCES UserReview(reviewID)) """)
```

Here, I added the column reviewID as a foreign key to link it to the UserReview table

```
## Creates a table for all questions
c.execute(""" CREATE TABLE Question(
            questionID INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT UNIQUE,
            quizID INTEGER
            questionType TEXT,
            question TEXT,
            noOfAnswers INTEGER,
            answer TEXT,
            timeLimit INTEGER,
            characterLimit INTEGER,
            background INTEGER,
            FOREIGN KEY(quizID) REFERENCES Quiz(quizID)) """)
```

```
## Creates a table for the progress of the students
c.execute(""" CREATE TABLE Progress(
            progressID INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT UNIQUE,
            userID INTEGER,
            quizID INTEGER,
            questionID INTEGER,
            correct BOOLEAN,
            time DATETIME,
            FOREIGN KEY(userID) REFERENCES UserDetails(userID)) """)
```

I also added a time column in the progress table with format DATETIME to prevent conflicts when users wish to play a quiz multiple times.

This now allows consistency to be maintained within the database.

Isolation requires simultaneous executions of transactions to produce the same output as a concurrent execution of those transactions. Since my program will not be extended for internet and multi user usage, there will be no simultaneous executions of transactions which may interfere with one another. Whilst perhaps available to a wider audience if extended to include online functionality, since my program does not need to modified to ensure isolation, there is a reduced risk of deadlocks or any other potential conflicts such as overwritten data.

Durability ensures that once a transaction has been processed, it will remain so regardless of the circumstances surrounding it, such as in the event of the program crashing or the computer shutting down. In the code shown above (for atomicity), this is partially achieved since a command is either completed in its entirety or not at all. However, it doesn't account for program or computer crashes, in which case, a transaction request is lost

To make a transaction (more) durable, so as to allow it to be completed in the case of any error, I have used this code:

```
## Transaction Durability
def saveTransacation(self, sqlCommand, conn, c):
    # Save SQL command to external text file
    with open('sqlCommands.txt', w) as file:
        file.write(sqlCommand)

## Check whether any pending transactions (and execute)
def readTransaction(self, conn, c):
    # Read SQL command from external text file (if any)
    with open('sqlCommands.txt', r) as file:
        sqlCommand = file.read()

    # If file isn't blank, execute instruction
    if sqlCommand != '':
        atomicTransacation(sqlCommand, conn, c)
```

Here, before executing a transaction, the SQL statement will first be saved into an external text file. When the statement has been successfully completed, it is then removed from the text file. This will be done by:

```
## Transaction Atomicity
def atomicTransacation(self, sqlCommand, conn, c):
    try:
        # Execute SQL command
        c.execute(sqlCommand)
        # Commit changes to database
        conn.commit()
        with open('sqlCommands.txt', w) as file:
            file.write('')
    except:
        # Rollback in case of error (reverts database to earlier state)
        conn.rollback()
```

Using this code, I can also combat the aforementioned issue of atomicity and external crashes. I shall create a backup file for my database. This backup will be updated after every transaction. When starting the program, it shall first check the external file (using the readTransaction function showed above) to see whether there are any SQL statements to be executed. If not, it shall assume that there have been no issues with database transactions and open the original database file. If there is, it will assume that there has been an issue and instead open the backup file after replacing the original file with it. It will then continue to execute the statement in the file. I shall do this by extending the readTransaction function:

```
## Check whether any pending transactions (and execute)
def readTransaction(self, conn, c):
    # Read SQL command from external text file (if any)
    with open('sqlCommands.txt', r) as file:
        sqlCommand = file.read()

    # If file isn't blank, open backup database file and execute instruction
    if sqlCommand != '':
        databaseFile = 'AccountInfo2.db'
        conn, c = openingConnection.openConnection(databaseFile)
        atomicTransacation(sqlCommand, conn, c)
    # If file is blank, open original database file
    else:
        databaseFile = 'AccountInfo.db'
        conn, c = openingConnection.openConnection(databaseFile)
```

There is also a limitation to this method however, since if the program or hardware crashes before the program has a chance to write the SQL statement into the external file, whether partially or at all, the instruction will not be carried out, even after reboot.

When testing the atomicTransaction function with the addToDatabase function, I encountered a few errors. Firstly, I modified the addToDatabase function to look like this:

```
## Add To Database Function
def addToDatabase(self, conn, c, record, table):
    ## If to be added to UserDetails table
    if len(record) == 7:
        # Establish SQL statement
        sqlAdd = ("INSERT INTO " + table + " VALUES (?,?,?,?,?,?,?)")
    ## If to be added to UserReview table
    elif len(record) == 2:
        sqlAdd = ("INSERT INTO " + table + " VALUES (?,?)")
    ## If to be added to Quiz table
    elif len(record) == 4:
        sqlAdd = ("INSERT INTO " + table + " VALUES (?,?,?,?)")
    ## If to be added to Question table
    elif len(record) == 9:
        sqlAdd = ("INSERT INTO " + table + " VALUES (?,?,?,?,?,?,?,?,?)")
    ## If to be added to Progress table
    elif len(record) == 6:
        sqlAdd = ("INSERT INTO " + table + " VALUES (?,?,?,?,?,?)")

    sqlCommand = sqlAdd, record
    self.saveTransacation(sqlCommand, conn, c)
    self.atomicTransaction(sqlCommand, conn, c)
```

I modified the code in terms of the length of records, since I had changed the length of the UserReview table to four columns and the Question table to nine. In addition, I created a variable sqlCommand which took the values of sqlAdd and record so to be passed on as one and execute faster. I concluded the function by calling the saveTransaction function and the atomicTransaction function so the SQL statement first saves to the external file and then executes (and then removes the statement from the file once successful).

I tested it with these variables:

```
table = 'Quiz'
record = [1, 'Computer Science', 'A-Level', 10]

addingToDatabase.addToDatabase(conn, c, record, table)
closingConnection.closeConnection()
```

It produced this error:

```
    file.write(sqlCommand)
TypeError: write() argument must be str, not tuple
>>>
```

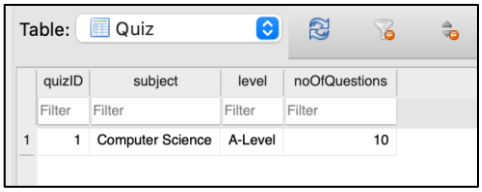
This is because when the program attempted to execute, it recognised the statement as a tuple since the brackets pass on to it, in the place of where it was supposed to receive a string.

To combat this:

```
## Transaction Atomicity
def atomicTransacation(self, sqlCommand, record, conn, c):
    try:
        # Execute SQL command
        c.execute(sqlCommand, record)
```

```
self.saveTransacation(sqlAdd, record, conn, c)
self.atomicTransaction(sqlAdd, record, conn, c)
```

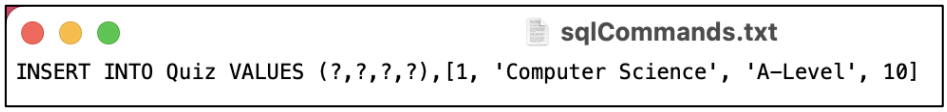
I passed on both the SQL statement and the record separately to the saveTransaction and atomicTransaction function and modified the code which executed the statement.



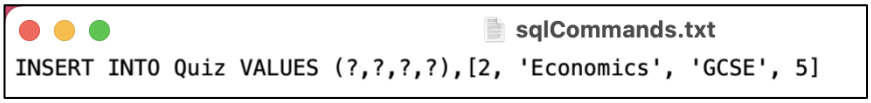
When executing just the top line, there was initially a small syntax error:

```
    with open('sqlCommands.txt', w) as file:
NameError: name 'w' is not defined
>>>
```

However this was easily rectifiable by putting quotation marks around the file modes, i.e. ‘w’ and ‘r’. This then produced:



When the program starts, it will first check the text file to see if empty or not. If not it will execute the written instruction. This occurs under the readTransaction function. To test it, I wrote this into the external text file:



To make testing easier, I included print statements before and after crucial parts of the program, such as opening the file, when the file is blank, when the file is not blank, the text read from the file, when the try statement is executed, when the SQL statement is executed, when changes to the database are committed and finally one for the except statement.

```
open file
INSERT INTO Quiz VALUES (?,?,?,?),[2, 'Economics', 'GCSE', 5]
file not blank
try statement
except
```

Here, we see that from this code, it first attempts to complete the try statement, however it is unable to execute the SQL statement successfully and thus resorts to the except statement. This is because, in the atomicTransaction function, where the SQL statement is executed, it takes in two inputs: the SQL statement and the record. However, since the entire statement was read from the file, ‘record’ is not defined, therefore the program is unable to complete the execution.

```
c.execute(sqlCommand, record)
```

I then modified the code to define record as a blank value. I first defined it as None which didn't work since instead of providing the statement with no input, it provided it with the string value 'None'. Alternatively, I defined record as '' so no input was passed on to the SQL statement. However, the same above output was produced. To identify the problem, I isolated the relevant code to outside of the try except statement to pinpoint the error:

```
sqlite3.OperationalError: near "[2, 'Economics', 'GCSE', 5]": syntax error
```

I decided to use a different approach. In the text file, I wrote the SQL statement on one line and underneath, the record list that I wanted to add to the database, as such:

sqlCommands.txt

```
INSERT INTO Quiz VALUES (?,?,?,?)
[2, 'Economics', 'GCSE', 5]
```

And modified the code under the readTransaction function to:

```
with open('sqlCommands.txt', 'r') as file:
    print('open file')
    sqlCommand = file.readline()
    sqlCommand = sqlCommand.rstrip('\n')
    record = file.readline()
```

Here, I read the first line of the text file using file.readline() and assigned it to a variable sqlCommand. However, I had to use the rstrip() function to get rid of the extra line that comes with the readline() function. I also assigned the second line to a variable record. It returned:

```
sqlite3.ProgrammingError: Incorrect number of bindings supplied. The current
statement uses 4, and there are 27 supplied.
```

To try understand the problem, I attempted to append an item to the record list:

```
    record.append('value')
AttributeError: 'str' object has no attribute 'append'
```

From this, it was evident that the program recognised the text in the text file not as a list, but as a string. Because of this, I tried an alternative approach. In the text file, I left the first line as it was, but spread the values of my record list over many lines:

sqlCommands.txt

```
INSERT INTO Quiz VALUES (?,?,?,?)
2
'Economics'
'GCSE'
5
```

By doing this, I hoped to append each value to a record list pre-defined by the program so the program will be able to identify it as a list rather than a string of values.

```
with open('sqlCommands.txt', 'r') as file:
            print('1')
            sqlCommand = file.readline()
            sqlCommand = sqlCommand.rstrip('\n')
            record = [line.rstrip() for line in file]
            print(sqlCommand)
            print(record)
```

This code produced this output:

```
open file
INSERT INTO Quiz VALUES (?,?,?,?)
['2', "'Economics'", "'GCSE'", '5']
file not blank
try statement
execute statement
commit changes
>>>
```

sqlCommands.txt

Which in turn resulted in:

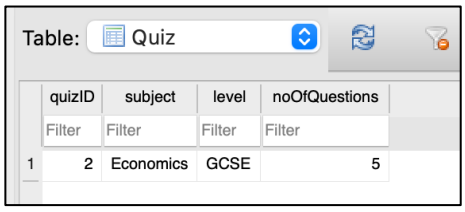


Table: Quiz

	quizID	subject	level	noOfQuestions
	Filter	Filter	Filter	Filter
1	2	Economics	GCSE	5

It appeared that SQL automatically dealt with the data structure of an integer when it was identified in the program as a string. To test this hypothesis, I entered this into the text file instead:

sqlCommands.txt
```
INSERT INTO Quiz VALUES (?,?,?,?)
TEST
'Economics'
'GCSE'
5
```

Which consequently produced:

```
    c.execute(sqlCommand,(record))
sqlite3.IntegrityError: datatype mismatch
```

This error represents the conflicting data structures that were identified. It failed to represent 'TEST' as the desired integer defined for its field in the database, thereby producing the error. This meant that I didn't have to first change any integers in the text file to integers in the program. However, it requires the code to deal with any alien values that may appear, for example conflicting data structures (as in the case above) or identical primary keys, etc. This is done with the try except statement since if the program experiences any errors, the except statement kicks in and any changes are reverted. I may consider an interface with an appropriate message when I code them.

By changing the way the program reads from the text file, it was crucial for me to modify the code in which transactions are saved to the external file to create the desired format.

```
## Transaction Durability
def saveTransaction(self, sqlCommand, record, conn, c):
    # Save SQL command to external text file
    with open('sqlCommands.txt', 'w') as file:
            file.write(sqlCommand + '\n')
            for each in record:
                    file.write(str(each) + '\n')
```

I tested this function with these variables:

```
sqlCommand = 'INSERT INTO Quiz VALUES (?,?,?,?)'
record = [3, 'Maths', 'AS-Level', 7]
```

To produce:

sqlCommands.txt
```
INSERT INTO Quiz VALUES (?,?,?,?)
3
Maths
AS-Level
7
```

as desired.

To create a backup file for my database within my program, I used the shutil library. There are various functions within this library that allow for different privileges:

Function	Copies metadata	Copies permissions	Can use buffer	Destination may be directory
shutil.copy	No	Yes	No	Yes
shutil.copyfile	No	No	No	No
shutil.copy2	Yes	Yes	No	Yes
shutil.copyfileobj	No	No	Yes	No

When assessing my options, I'd like any permissions and metadata in the original database file to be preserved, therefore making the copy2 function the best option:

```
import shutil
```

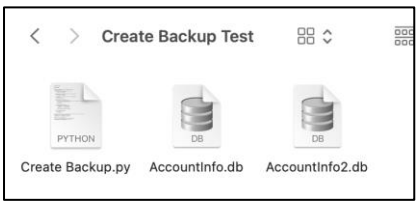
```
## Create Backup Database File
def createBackupDBFile():
    originalFile = 'AccountInfo.db'
    backupFile = 'AccountInfo2.db'
    shutil.copy2(originalFile, backupFile)
```

This function determines the file that is to be copied, and the file (name) that is to contain the copy and copies it across.
However, the program may not always take the AccountInfo database file as the opened file, since if any errors are experienced, it resorts to the backup. Hence I changed it to:

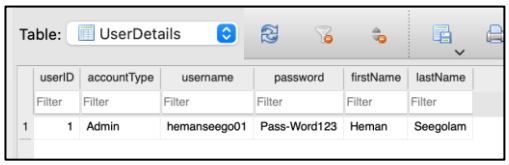
```
## Create Backup Database File
def createBackupDBFile(self, databaseFile):
    if databaseFile == 'AccountInfo.db':
        backupFile = 'AccountInfo2.db'
    else:
        backupFile = 'AccountInfo.db'
    shutil.copy2(databaseFile, backupFile)
```

This modified function takes in the parameter databaseFile which contains the name of the database file in use and then proceeds to assign the other file as the backup. It then allows the database file in use to be copied to the relevant backup. This is necessary since when the backup file is first opened by the program after encountering an error, the original file is copied with its contents since it may be corrupted.

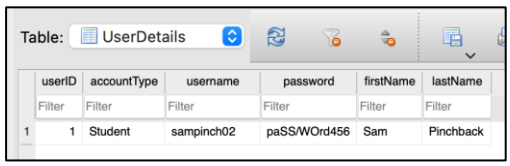
To test this function, I isolated it from the rest of the program and assigned the original file to databaseFile. I moved this new python file and only the original database to a new folder to explore its effects.



In the AccountInfo file:



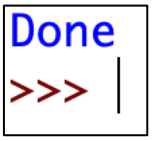
In the AccountInfo2 file:

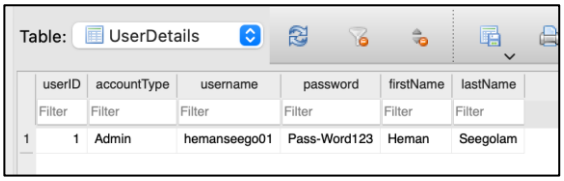


To test it:

```
databaseFile = 'AccountInfo.db'
createBackupFile.createBackupDBFile(databaseFile)
print('Done')
```

And it produced:



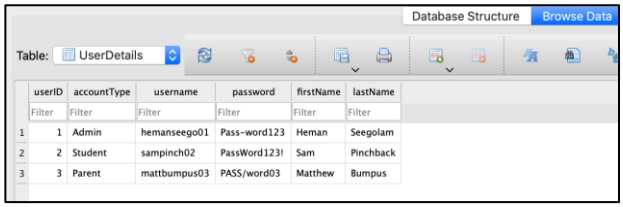


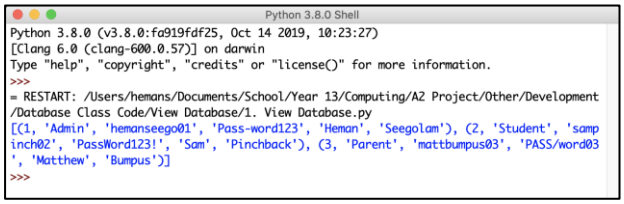
In the AccountInfo2 file.

VIEW FROM DATABASE FUNCTION:

For this function, it must be able to view all or some items stored in the database for a given table, field and record. To retrieve and display all items from a table:

```
def viewFromDatabase(field, record, table):
    if field == 'All' and record == 'All':
        c.execute("SELECT * FROM " + table)
        return c.fetchall()

field = 'All'
record = 'All'
table = 'UserDetails'
print(viewFromDatabase(field, record, table))
```

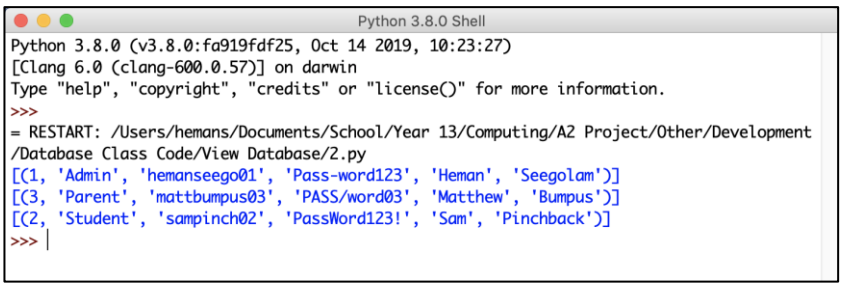




To retrieve and display all items from a record:

```
def viewFromDatabase(allFields, allRecords, recordPrimaryKey, recordPrimaryKeyValue, table):
    if allFields == True and allRecords == False:
        c.execute("SELECT * FROM " + table + " WHERE " + recordPrimaryKey + " = " + str(recordPrimaryKeyValue))
        return c.fetchall()

allFields = True
allRecords = False
recordPrimaryKey = 'userID'
recordPrimaryKeyValue1 = 1
recordPrimaryKeyValue2 = 2
recordPrimaryKeyValue3 = 3
table = 'UserDetails'
print(viewFromDatabase(allFields, allRecords, recordPrimaryKey, recordPrimaryKeyValue1, table))
print(viewFromDatabase(allFields, allRecords, recordPrimaryKey, recordPrimaryKeyValue3, table))
print(viewFromDatabase(allFields, allRecords, recordPrimaryKey, recordPrimaryKeyValue2, table))
```

In this code, I have firstly swapped the 'field' and 'record' variables to 'allFields' and 'allRecords' boolean variables to make it easier to determine whether all records and fields are desired. Since the record primary key value can take an integer, I have first made it a string before executing. This is because you cannot concatenate an integer to a string. However, when comparing a string (i.e. '1') to the same integer (1), they are identified as the same, thus allowing it to work in this way. Since the table and the record primary key name never takes on an integer value, I can leave those as they are. When testing, I retrieved and displayed multiple values from my table, but in a different order to confirm that that program was retrieving the correct data:



To retrieve and display all items from a field:

```
def viewFromDatabase(allFields, field, allRecords, table):
    if allFields == False and allRecords == True:
        c.execute("SELECT " + field + " FROM " + table)
        return c.fetchall()

allFields = False
allRecords = True
field = 'firstName'
table = 'UserDetails'

print(viewFromDatabase(allFields, field, allRecords, table))
```

```
Python 3.8.0 Shell
Python 3.8.0 (v3.8.0:fa919fdf25, Oct 14 2019, 10:23:27)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: /Users/hemans/Documents/School/Year 13/Computing/A2 Project/Other/Development
/Database Class Code/View Database/3.py
[('Heman',), ('Sam',), ('Matthew',)]
>>>
```

To retrieve and display an item from a field from a specific record:

```
def viewAllFromDatabase(allFields, allRecords, field, recordPrimaryKey, recordPrimaryKeyValue, table):
    if allFields == False and allRecords == False:
        c.execute("SELECT " + field + " FROM " + table + " WHERE " + recordPrimaryKey + " = " + str(recordPrimaryKeyValue))
        return c.fetchall()

allFields = False
allRecords = False
table = 'UserDetails'
field = 'firstName'
recordPrimaryKey = 'userID'
recordPrimaryKeyValue = 1

print(viewAllFromDatabase(allFields, allRecords, field, recordPrimaryKey, recordPrimaryKeyValue, table))
```

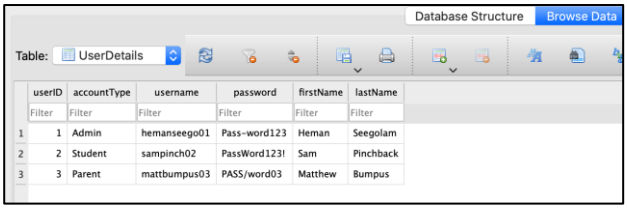
```
Python 3.8.0 Shell
Python 3.8.0 (v3.8.0:fa919fdf25, Oct 14 2019, 10:23:27)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: /Users/hemans/Documents/School/Year 13/Computing/A2 Project/Other/Dev
elopment/Database Class Code/View Database/4.py
[('Heman',)]
>>>
```

Full function:

```
## VIEW FROM DATABASE

import sqlite3

## Initiates the database with the name "AccountInfo.db" along with its cursor
conn = sqlite3.connect('AccountInfo.db')
c = conn.cursor()

def viewFromDatabase(allFields, allRecords, field, recordPrimaryKey, recordPrimaryKeyValue, table):
    ## Retrieve and display all items from a table
    if allFields == True and allRecords == True:
        c.execute("SELECT * FROM " + table)
        return c.fetchall()
    ## Retrieve and display all items from a record
    elif allFields == True and allRecords == False:
        c.execute("SELECT * FROM " + table + " WHERE " + recordPrimaryKey + " = " + str(recordPrimaryKeyValue))
        return c.fetchall()
    ## Retrieve and display all items from a field
    elif allFields == False and allRecords == True:
        c.execute("SELECT " + field + " FROM " + table)
        return c.fetchall()
    ## Retrieve and display an item from a field from a specific record
    if allFields == False and allRecords == False:
        c.execute("SELECT " + field + " FROM " + table + " WHERE " + recordPrimaryKey + " = " + str(recordPrimaryKeyValue))
        return c.fetchall()

conn.commit()
conn.close()
```

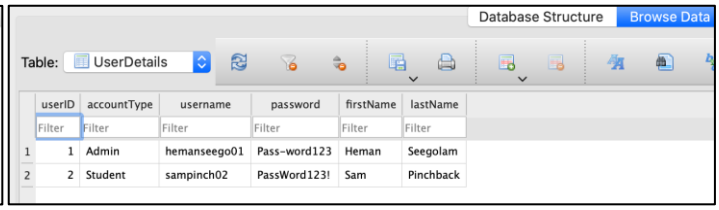
DELETE FROM DATABASE FUNCTION:

To delete all fields for a specified record:



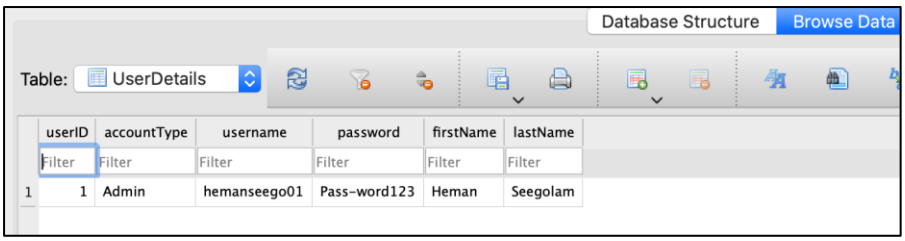

```
def deleteFromDatabase(allFields, recordPrimaryKey, recordPrimaryKeyValue, table):
    if allFields == True:
        c.execute("DELETE from " + table + " WHERE " + recordPrimaryKey + " = " + str(recordPrimaryKeyValue))

allFields = True
table = 'UserDetails'
recordPrimaryKey = 'userID'
recordPrimaryKeyValue = 3
deleteFromDatabase(allFields, recordPrimaryKey, recordPrimaryKeyValue, table)
print('Done')
```

```
Python 3.8.0 Shell
Python 3.8.0 (v3.8.0:fa919fdf25, Oct 14 2019, 10:23:27)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: /Users/hemans/Documents/School/Year 13/Computing/A2 Project/Other/Dev
elopment/Database Class Code/Delete From Database/1.py
Done
>>>
```



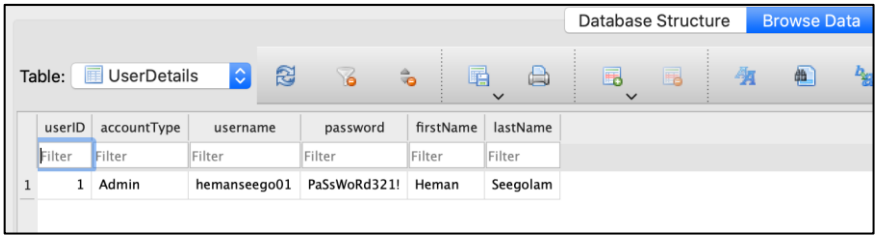
Although possible, other forms of deletion such as whole table deletion, singular field deletion, etc. are irrelevant to my program and hence do not need to be programmed.

UPDATE DATABASE FUNCTION:




```
def updateFromDatabase(field, updatedData, recordPrimaryKey, recordPrimaryKeyValue, table):
    c.execute("UPDATE " + table + " SET " + field + " = '" + updatedData + "' WHERE " + recordPrimaryKey + " = " + str(recordPrimaryKeyValue))

table = 'UserDetails'
field = 'password'
updatedData = 'PaSsWoRd321!'
recordPrimaryKey = 'userID'
recordPrimaryKeyValue = 1

updateFromDatabase(field, updatedData, recordPrimaryKey, recordPrimaryKeyValue, table)
print('Done')
```

```
Python 3.8.0 Shell
Python 3.8.0 (v3.8.0:fa919fdf25, Oct 14 2019, 10:23:27)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: /Users/hemans/Documents/School/Year 13/Computing/A2 Project/Other/Dev
elopment/Database Class Code/Update Database/1.py
Done
>>>
```



User Class:

To facilitate easy transactions within my database, I have set up the functions shown above to take in a list of all the relevant fields. To create these record lists, I shall create a class for all the relevant users in my program.

First, I declared a class for all users:

```
## Creates a parent class for all users
class User:
    def __init__(self, newUserID, newAccountType, newUsername, newPassword, newFirstName, newLastName, newReviewID):
        self.__userID = newUserID
        self.__accountType = newAccountType
        self.__username = newUsername
        self.__password = newPassword
        self.__firstName = newFirstName
        self.__lastName = newLastName
        self.__reviewID = newReviewID
```

Here, I declared the class, and constructed all the various attributes relevant to the users, namely the user ID, account type, username, password, first name, last name and review ID

I then defined the set and get methods for this class:

```
## Encapsulation of User Class (Set Methods)
    def setUserID(self, newUserID):
        self.__userID = newUserID

    def setAccountType(self, newAccountType)
        self.__accountType = newAccountType

    def setUsername(self, newUsername):
        self.__username = newUsername

    def setPassword(self, newPassword):
        self.__password = newPassword

    def setFirstName(self, newFirstName):
        self.__firstName = newFirstName

    def setLastName(self, newLastName):
        self.__lastName = newLastName

    def setReview(self, newReviewID):
        self.__reviewID = newReviewID
```

```
## Encapsulation of User Class (Get Methods)
    def getUserID(self):
        return self.__userID

    def getAccountType(self):
        return self.__accountType

    def getUsername(self):
        return self.__username

    def getPassword(self):
        return self.__password

    def getFirstName(self):
        return self.__firstName

    def getLastName(self):
        return self.__lastName

    def getReviewID(self):
        return self.__reviewID
```

To retrieve all details from an instance of the class:

```
def getAllDetails(self):
    return [User.getUserID(self), User.getAccountType(self), User.getUsername(self), User.getPassword(self),
            User.getFirstName(self), User.getLastName(self), User.getReviewID(self)]
```

```
heman = User(1, 'Admin', 'hemanseego01', 'Pass-word123', 'Heman', 'Seegolam', 1)
sam = User(2, 'Student', 'sampinch02', 'Pass-word123', 'Sam', 'Pinchback', 1)
matt = User(3, 'Parent', 'mattbumpus03', 'Pass-word123', 'Matt', 'Bumpus', 2)
print(heman.getAllDetails())
print(sam.getAllDetails())
print(matt.getAllDetails())
```

```
[1, 'Admin', 'hemanseego01', 'Pass-word123', 'Heman', 'Seegolam', 1]
[2, 'Student', 'sampinch02', 'Pass-word123', 'Sam', 'Pinchback', 1]
[3, 'Parent', 'mattbumpus03', 'Pass-word123', 'Matt', 'Bumpus', 2]
```

Previously when designing this class, I had created two additional child classes for admins and for students. However, to effectively retrieve values from the database and declare them as an object of this class, it is easier and more efficient to instead make the account type an attribute of all users and retrieve it from the database like that.

To retrieve these values from the database and identify them as the attributes of an instance of the User class, I shall use the viewFromDatabase function with allFields as True and other predetermined variables:

```
allFields = True
allRecords = False
field = None
recordPrimaryKey = 'userID'
recordPrimaryKeyValue = 1
table = 'UserDetails'
```

```
[(1, 'Admin', 'hemanseego01', 'password123', 'Heman', 'Seegolam', 1)]
```

Here, we can see that the program takes all the values from the database and puts it into a tuple. This tuple is then put into a list. To add this as an object of the User class:

```
record = viewingFromDatabase.viewFromDatabase(c, allFields, allRecords, field, recordPrimaryKey, recordPrimaryKeyValue, table)
userOne = User(*record[0])
print(userOne.getAllDetails())
```

Here * retrieves all values from record[0]. Record[0] refers to the tuple inside the list retrieved from the database.
This outputs:

```
[1, 'Admin', 'hemanseego01', 'password123', 'Heman', 'Seegolam', 1]
```

I can put this all under its own function in the database class shown above:

```
## Create Class Object from Database Record
def createObjectOfClassFromDatabase(self, c, allFields, allRecords, field, recordPrimaryKey, recordPrimaryKeyValue, table):
    record = self.viewFromDatabase(c, allFields, allRecords, field, recordPrimaryKey, recordPrimaryKeyValue, table)
    userOne = User(*record[0])
```

FULL DATABASE CLASS FUNCTION:

```
## DATABASE CLASS

##from UserClass import *
import sqlite3
import shutil

## Creates a parent class for all database functions
class DatabaseFunction:
    ## Assigns attributes for database functions
    def __init__(self, newType):
        self.__type = newType

    ## Encapsulation of DatabaseFunction Class (Set Methods)
    def setType(self, newType):
        self.__type = newType

    ## Encapsulation of DatabaseFunction Class (Get Methods)
    def getType(self):
        return self.__type

    ## Open Connection Function
    def openConnection(self, databaseFile):
        ## Initiates the database with the name stored in databaseFile along with its cursor
        conn = sqlite3.connect(databaseFile)
        c = conn.cursor()
        return conn, c
```

```
    ## Add To Database Function
    def addToDatabase(self, conn, c, record, table):
        ## If to be added to UserDetails table
        if len(record) == 8:
            # Establish SQL statement
            sqlAdd = ("INSERT INTO " + table + " VALUES (?,?,?,?,?,?,?,?)")
        ## If to be added to UserReview table
        elif len(record) == 2:
            sqlAdd = ("INSERT INTO " + table + " VALUES (?,?)")
        ## If to be added to Quiz table
        elif len(record) == 4:
            sqlAdd = ("INSERT INTO " + table + " VALUES (?,?,?,?)")
        ## If to be added to Question table
        elif len(record) == 9:
            sqlAdd = ("INSERT INTO " + table + " VALUES (?,?,?,?,?,?,?,?,?)")
        ## If to be added to Progress table
        elif len(record) == 6:
            sqlAdd = ("INSERT INTO " + table + " VALUES (?,?,?,?,?,?)")

        self.saveTransaction(sqlAdd, record, conn, c)
        self.atomicTransaction(sqlAdd, record, conn, c)

    ## View From Database Function
    def viewFromDatabase(self, c, allFields, allRecords, field, recordPrimaryKey, recordPrimaryKeyValue, table):
        ## Retrieve and display all items from a table
        if allFields == True and allRecords == True:
            sqlView = ("SELECT * FROM " + table)
        ## Retrieve and display all items from a record
        elif allFields == True and allRecords == False:
            sqlView = ("SELECT * FROM " + table + " WHERE " + recordPrimaryKey + " = " + str(recordPrimaryKeyValue))
        ## Retrieve and display all items from a field
        elif allFields == False and allRecords == True:
            sqlView = ("SELECT " + field + " FROM " + table)
        ## Retrieve and display an item from a field from a specific record
        elif allFields == False and allRecords == False:
            sqlView = ("SELECT " + field + " FROM " + table + " WHERE " + recordPrimaryKey + " = " + str(recordPrimaryKeyValue))
        c.execute(sqlView)
        return c.fetchall()
```

```
    ## Update In Database Function
    def updateInDatabase(self, conn, c, field, updatedData, recordPrimaryKey, recordPrimaryKeyValue, table):
        sqlUpdate = ("UPDATE " + table + " SET " + field + " = '" + updatedData + "' WHERE " + recordPrimaryKey + " = " + str(recordPrimaryKeyValue))
        record = []
        self.saveTransaction(sqlUpdate, record, conn, c)
        self.atomicTransaction(sqlUpdate, record, conn, c)

    ## Delete From Database Function
    def deleteFromDatabase(self, conn, c, recordPrimaryKey, recordPrimaryKeyValue, table):
        sqlDelete = ("DELETE from " + table + " WHERE " + recordPrimaryKey + " = " + str(recordPrimaryKeyValue))
        record = []
        self.saveTransaction(sqlDelete, record, conn, c)
        self.atomicTransction(sqlDelete, record, conn, c)

    ## Create Class Object from Database Record
    def createObjectOfClassFromDatabase(self, c, allFields, allRecords, field, recordPrimaryKey, recordPrimaryKeyValue, table):
        record = self.viewFromDatabase(c, allFields, allRecords, field, recordPrimaryKey, recordPrimaryKeyValue, table)
        userOne = User(*record[0])
```

```
    ## Transaction Atomicity
    def atomicTransaction(self, sqlCommand, record, conn, c):
        try:
            ## For updating and deleting
            if record == []:
                # Execute SQL command
                c.execute(sqlCommand)
            ## For adding
            else:
                c.execute(sqlCommand, record)
            # Commit changes to database
            conn.commit()
            with open('sqlCommands.txt', 'w') as file:
                file.write('')
            self.createBackupDBFile(databaseFile)
        except:
            # Rollback in case of error (reverts database to earlier state)
            conn.rollback()

    ## Transaction Durability
    def saveTransaction(self, sqlCommand, record, conn, c):
        # Save SQL command to external text file
        with open('sqlCommands.txt', 'w') as file:
                file.write(sqlCommand + '\n')
                for each in record:
                        file.write(str(each) + '\n')
```

```
    ## Check whether any pending transactions (and execute)
    def readTransaction(self):
        # Read SQL command from external text file (if any)
        with open('sqlCommands.txt', 'r') as file:
            sqlCommand = file.readline()
            sqlCommand = sqlCommand.rstrip('\n')
            record = [line.rstrip() for line in file]

        # If file isn't blank, open backup database file and execute instruction
        if sqlCommand != '':
            databaseFile = 'AccountInfo2.db'
            conn, c = self.openConnection(databaseFile)
            self.atomicTransaction(sqlCommand, record, conn, c)
            self.createBackupDBFile(databaseFile)
        # If file is blank, open original database file
        else:
            databaseFile = 'AccountInfo.db'
            conn, c = self.openConnection(databaseFile)

        return conn, c

    ## Create Backup Database File
    def createBackupDBFile(self, databaseFile):
        if databaseFile == 'AccountInfo.db':
            backupFile = 'AccountInfo2.db'
        else:
            backupFile = 'AccountInfo.db'
        shutil.copy2(databaseFile, backupFile)

    ## Close Connection Function
    def closeConnection(self):
        conn.commit()
        conn.close()
```

```
## Creates a new instance of the DatabaseFunction class for the different functions
openingConnection = DatabaseFunction('Open')
addingToDatabase = DatabaseFunction('Add')
updatingDatabase = DatabaseFunction('Update')
deletingFromDatabase = DatabaseFunction('Delete')
saveTransactionToFile = DatabaseFunction('Save')
readTransactionFromFile = DatabaseFunction('Read')
viewingFromDatabase = DatabaseFunction('View')
createBackupFile = DatabaseFunction('Backup')
closingConnection = DatabaseFunction('Close')

conn, c = readTransactionFromFile.readTransaction()

##allFields =
##allRecords =
##field =
##recordPrimaryKey =
##recordPrimaryKeyValue =
##table =

##addingToDatabase.addToDatabase(conn, c, record, table)
##print(viewingFromDatabase.viewFromDatabase(c, allFields, allRecords, field, recordPrimaryKey, recordPrimaryKeyValue, table))
##viewingFromDatabase.createObjectOfClassFromDatabase(c, allFields, allRecords, field, recordPrimaryKey, recordPrimaryKeyValue, table)
##updatingDatabase.updateInDatabase(conn, c, field, updatedData, recordPrimaryKey, recordPrimaryKeyValue, table)
##deletingFromDatabase.deleteFromDatabase(conn, c, recordPrimaryKey, recordPrimaryKeyValue, table)

#closingConnection.closeConnection()
```

In this final code for the database class, I have linked all the methods, i.e. the database functions (adding, viewing, updating, deleting, atomicity, durability, object creation, backup creation) to the class – this was done by passing the 'self' variable into each function and applying the functions to a particular object in the class. In addition, I have created two extra methods to open and close the connection with the database. By doing so, I have made the code more reusable and overall, more efficient. By assigning the name of my database file to a variable named databaseFile and passing that into the open connection statement, theoretically, this variable could be changed and the class should work equally as well for another database file. To enable the connection that had been opened and the cursor to be accessible within the other functions, I passed the cursor to all functions requiring SQL statement execution and the connection to the final close method in order to commit the connection and finally close it at the end of the program. The add, delete and update function all passed their relevant SQL statements and records to the transaction atomicity function where they were carried out. To get around the issue of (sqlCommand, record) for all commands, I assigned a value of [] (a blank list) to record for the delete and update functions and used an if statement to allow all functions to be carried out.

Since my UserReview table will not be subject to any changes, I can feed into it the relevant details:

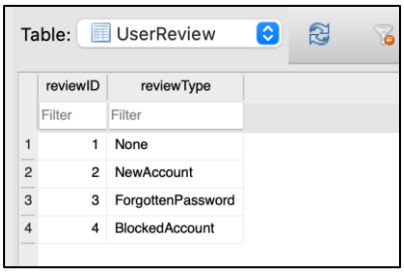


Table: UserReview

	reviewID	reviewType
	Filter	Filter
1	1	None
2	2	NewAccount
3	3	ForgottenPassword
4	4	BlockedAccount

These are the various reasons for why an account may be under review: it is a newly created account (so needs verification from an admin user), the user has forgotten their password (so needs to be reset by an admin user), the account is blocked (in the case of mistyping their password three times – can be unblocked by admin user) or none, where no action needs to be taken.

Interface Class:

```
## Creates a parent class for all interfaces
class Interface:
    ## Assigns attributes for all interfaces (height and width)
    def __init__(self, newScreenWidth, newScreenHeight):
        self.__screenWidth = newScreenWidth
        self.__screenHeight = newScreenHeight

    ## Encapsulation of Interface Class (Set Methods)
    def setScreenWidth(self, newScreenWidth):
        self.__screenWidth = newScreenWidth

    def setScreenHeight(self, newScreenHeight):
        self.__screenHeight = newScreenHeight

    ## Encapsulation of Interface Class (Get Methods)
    def getScreenWidth(self):
        return self.__screenWidth

    def getScreenHeight(self):
        return self.__screenHeight
```

Here, I have created a (parent) class for all interface pages. In this class, I constructed two attributes which determined the height of the screen of the page and the width of the page. I then created corresponding methods to set the height and width of the page and retrieve them for a particular instance of the class.

Login:

For the login, I created a child class of the Interface class for all of the login pages.

```
## Creates a child class for the login pages
class Login(Interface):
    def __init__(self, newScreenWidth, newScreenHeight, newType):
        ## Assigns attributes for login pages (height and width) from parent class Interface
        Interface.__init__(self, newScreenWidth, newScreenHeight)
        ## Assigns additional attribute unique to Login class (the type of login page, i.e. Admin, Student, Parent or the main login page)
        self.__type = newType

    ## Encapsulation of Login Class (Set Methods)
    def setNewType(self, newType):
        self.__type = newType

    ## Encapsulation of Login Class (Get Methods)
    def getType(self):
        return self.__type
```

In this child class, I inherited the attributes from the parent class above, and included a type attribute to differentiate between the pages, e.g. the admin, student, parent or main login page.

```
## Creates a new instance of the Login class for the main login page and the admin, student and parent login pages
mainLogin = Login(700, 500, 'Main')
adminLogin = Login(700, 500, 'Admin')
studentLogin = Login(700, 500, 'Student')
parentLogin = Login(700, 500, 'Parent')
```

Consequently, I created four instances of the child class Login mentioned above. I assigned them with a width of 700 pixels and a height of 500 pixels. I also mentioned the type of each of these pages. By using a class in this way, I am able to easily change the dimensions of the pages without modifying many different bits of the code and therefore makes it more efficient and easier to use.

To create the interfaces, I used the tkinter library within python, so firstly I had to import the library:

```
import tkinter as tk
```

By referencing it as 'tk', at every occasion where I have to call the tkinter library, I can do this by mentioning 'tk.', followed by any function within the library.
I then had to import the tkinter font library to make use of different fonts within my pages. This is to make my program more professional and presentable to my client and target audience.

```
from tkinter import font
```

Within the class, I created these variables:

```
## Sets the font and size for the title of the pages, the buttons and default writing on the mainframe
titleFont = ('Helvetica', 25)
buttonFont = ('Helvetica', 20)
defaultFont = ('Helvetica', 17)
```

These will be used later on in my interface methods to determine fonts of any headings, button text and any other writing on the page. By importing the font library from tkinter, I am able to make use of the 'Helvetica' font as I do here. I also determine the font size for each of these variables.

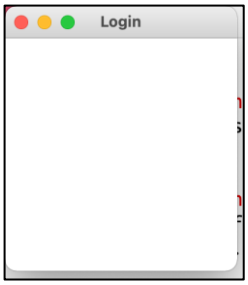
To create my main login page, I declared the function as a method of the login class.

```
## Creates the main login page interface
def mainLoginPage(self, titleFont, buttonFont, defaultFont):
    root = tk.Tk()
    root.title("Login")

    root.mainloop()
```

Here, I created a root window by calling the Tk function from tkinter and named the window 'Login'. root.mainloop() is a method on the main window which runs the program. This method will loop forever, waiting for events from the user, until the user exits the program – either by closing the window, or by terminating the program with a keyboard interrupt in the console. Running:

```
mainLogin.mainLoginPage(mainLogin.titleFont, mainLogin.buttonFont, mainLogin.defaultFont)
```

This produced:

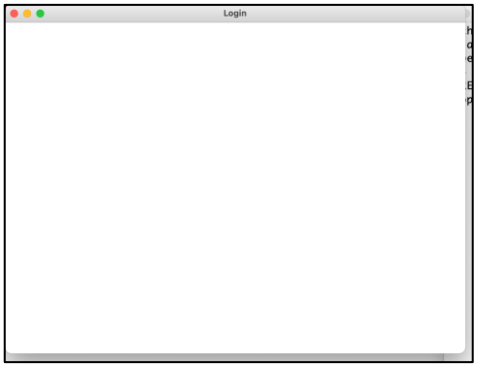


This creates a separate, blank window. It has the name declared above.

I extended the code to create the window with the dimensions set from the class instances:

```
## Creates the main login page interface
def mainLoginPage(self, titleFont, buttonFont, defaultFont):
    root = tk.Tk()
    root.title("Login")

    ## Retrieves the assigned width and height for the main login page
    mainScreenWidth = mainLogin.getScreenWidth()
    mainScreenHeight = mainLogin.getScreenHeight()

    canvas = tk.Canvas(root, height = mainScreenHeight, width = mainScreenWidth)
    canvas.pack()

    root.mainloop()
```

Here, I created a canvas for the window, and set the dimensions of the canvas by calling the get functions of the Interface class to assign the width and height of the main login page to separate variables. To place the canvas into the window, I used to the pack() method. By default, this method arranges widgets vertically inside their parent container, from the top down.
This produced:

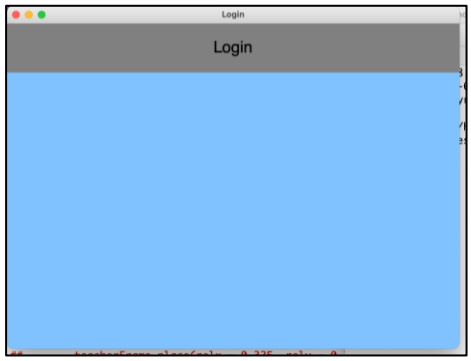


A window with size 700 x 500 as declared in the mainLogin instance of the Login class.

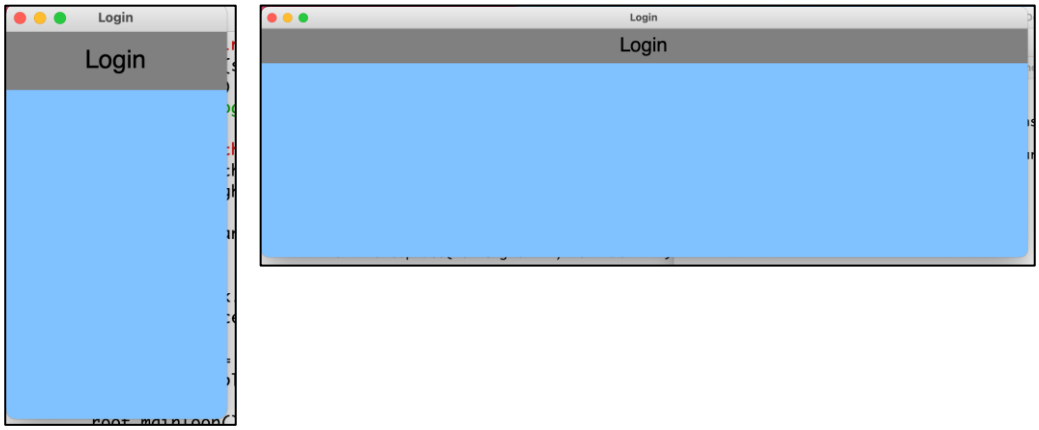
To better organise the different items that were to be placed on the window, I included a main frame for my window:

```
mainframe = tk.Frame(root, bg = '#80c1ff')
mainframe.place(relheight = 1, relwidth = 1)

welcomeLabel = tk.Label(mainframe, text = "Login", bg = 'gray', font = titleFont)
welcomeLabel.place(relx = 0, rely = 0, relheight = 0.15, relwidth = 1)

root.mainloop()
```

Whilst the canvas was just to predetermine the dimensions of the window when it was first opened, the use of a frame allows the window to be more flexible. Here, I assigned a background colour to the frame and used the place() method instead of the pack method. This is what allows for the flexibility of the window. By determining the relative height and relative width (to the window) as 1, it meant that the frame covered the entirety of the window.
For example, this is what appears when this code is run:



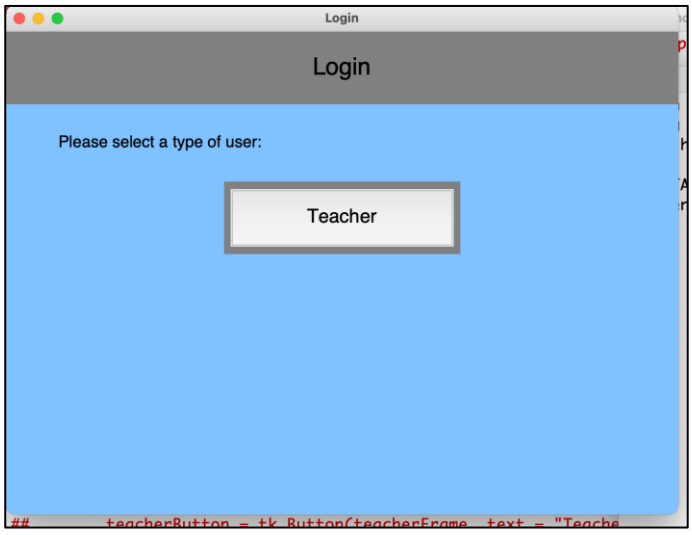
However, when the window is stretched by the user:



All information on the window is still present, whether contracted or extended. I decided to use this method of approach since it provide a degree of flexibility for the user and can modify the window size to match his or her preference.

```
selectUserLabel = tk.Label(mainframe, text = "Please select a type of user:", bg = '#80c1ff', font = defaultFont)
selectUserLabel.place(relx = 0.03, rely = 0.18, relheight = 0.1, relwidth = 0.4)

teacherFrame = tk.Frame(mainframe, bg = 'gray')
teacherFrame.place(relx = 0.325, rely = 0.31, relheight = 0.15, relwidth = 0.35)

teacherButton = tk.Button(teacherFrame, text = "Teacher", bg = '#E5E7E9', font = buttonFont)
teacherButton.place(relx = 0.03, rely = 0.1, relheight = 0.8, relwidth = 0.94)

root.mainloop()
```

Here, following the design I previously created for this page, I created a label which asks the user to select the type of user that they are. To make it blend in with the background, I also set its background to the same colour as above. By using relx and rely, I was able to place the label in a specific position relative to the size of the window where 0.03 in this case represents 3% of the screen size, in the x or y direction. This also aids the flexibility of the window. I also created a separate frame which went into the main frame for the teacher button. This was mainly for a more aesthetic look in which the button had a background separate to that of the main frame, in this case of the colour grey. Within that frame, I placed the teacher button. Within the selectUserLabel and the teacherButton I also fed into it the correct respective font. This produced:



Currently, the button has no function since no command had been integrated into it. However, I created a separate function for if the teacher button was clicked and fed it into the button:

```
teacherButton = tk.Button(teacherFrame, text = "Teacher", bg = '#E5E7E9', font = buttonFont, command = self.teacherButtonClicked)
teacherButton.place(relx = 0.03, rely = 0.1, relheight = 0.8, relwidth = 0.94)
```

```
def teacherButtonClicked(self):
    print('Teacher Clicked')
```

Which when clicked, produced:

```
Teacher Clicked
```

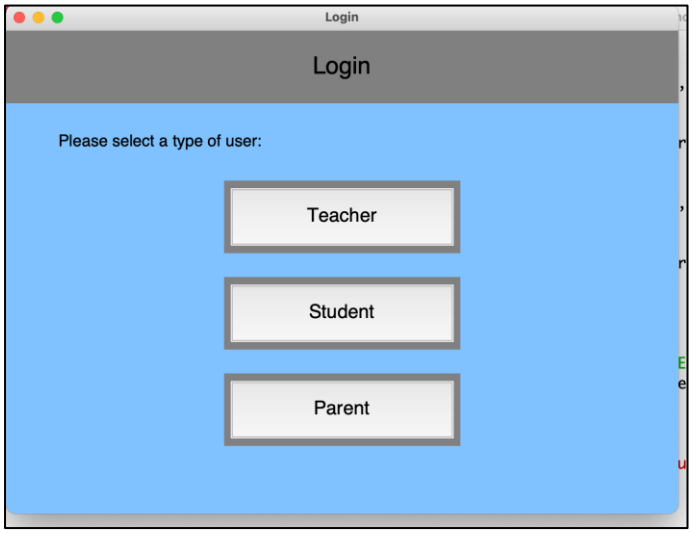
I proceeded to do the other frames and buttons:

```
studentFrame = tk.Frame(mainframe, bg = 'gray')
studentFrame.place(relx = 0.325, rely = 0.51, relheight = 0.15, relwidth = 0.35)

studentButton = tk.Button(studentFrame, text = "Student", bg = '#E5E7E9', font = buttonFont, command = self.studentButtonClicked)
studentButton.place(relx = 0.03, rely = 0.1, relheight = 0.8, relwidth = 0.94)

parentFrame = tk.Frame(mainframe, bg = 'gray')
parentFrame.place(relx = 0.325, rely = 0.71, relheight = 0.15, relwidth = 0.35)

parentButton = tk.Button(parentFrame, text = "Parent", bg = '#E5E7E9', font = buttonFont, command = self.parentButtonClicked)
parentButton.place(relx = 0.03, rely = 0.1, relheight = 0.8, relwidth = 0.94)

root.mainloop()
```

As above, I created a frame for the student and parent button and embedded the button within it, changing the rely values as I go to place the frames in a lower position than the last.

```
def studentButtonClicked(self):
    print('Student Clicked')

def parentButtonClicked(self):
    print('Parent Clicked')
```

And as above, I created functions for the student and parent buttons within the class.

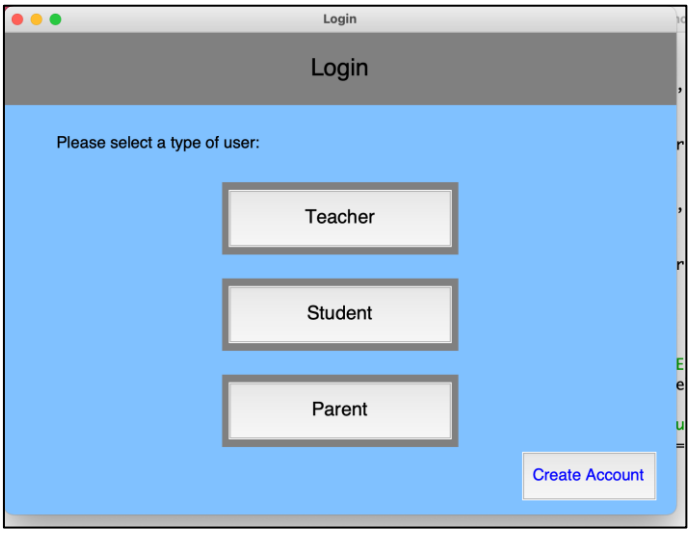



```
Teacher Clicked
Student Clicked
Parent Clicked
```

Finally, I created a label to create an account:

```
createAccountButton = tk.Button(mainframe, text = "Create Account", fg = 'blue', font = defaultFont, command = self.createAccountButtonClicked)
createAccountButton.place(relx = 0.77, rely = 0.87, relheight = 0.1, relwidth = 0.2)

root.mainloop()
```

```
def createAccountButtonClicked(self):
    print('Create Account Clicked')
```




```
Create Account Clicked
```

Admin login page:

```
def adminLoginPage(self, titleFont, buttonFont, defaultFont):
    aLogin = tk.Tk()
    aLogin.title("Teacher Login")

    ## Retrieves the assigned width and height for the admin login page
    adminScreenWidth = adminLogin.getScreenWidth()
    adminScreenHeight = adminLogin.getScreenHeight()

    canvas = tk.Canvas(aLogin, height = adminScreenHeight, width = adminScreenWidth)
    canvas.pack()

    mainframe = tk.Frame(aLogin, bg = '#80c1ff')
    mainframe.place(relheight = 1, relwidth = 1)

    welcomeLabel = tk.Label(mainframe, text = "Login", bg = 'gray', font = titleFont)
    welcomeLabel.place(relx = 0, rely = 0, relheight = 0.1, relwidth = 1)

    differentLoginLabel = tk.Label(mainframe, text = "Not a teacher? Try the student or parent login", bg = '#80c1ff', fg = 'blue', font = defaultFont)
    differentLoginLabel.place(relx = 0.05, rely = 0.17, relheight = 0.1, relwidth = 0.55)

    usernameFrame = tk.Frame(mainframe, bg = 'gray')
    usernameFrame.place(relx = 0.2, rely = 0.35, relheight = 0.1, relwidth = 0.65)

    usernameLabel = tk.Label(usernameFrame, text = "Username: ", font = defaultFont)
    usernameLabel.place(relx = 0.02, rely = 0.1, relheight = 0.8, relwidth = 0.35)

    usernameEntry = tk.Entry(usernameFrame)
    usernameEntry.place(relx = 0.4, rely = 0.1, relheight = 0.8, relwidth = 0.58)

    passwordFrame = tk.Frame(mainframe, bg = 'gray')
    passwordFrame.place(relx = 0.2, rely = 0.5, relheight = 0.1, relwidth = 0.65)

    passwordLabel = tk.Label(passwordFrame, text = "Password: ", font = defaultFont)
    passwordLabel.place(relx = 0.02, rely = 0.1, relheight = 0.8, relwidth = 0.35)

    passwordEntry = tk.Entry(passwordFrame)
    passwordEntry.place(relx = 0.4, rely = 0.1, relheight = 0.8, relwidth = 0.58)
```

```
    loginButton = tk.Button(mainframe, text = "Log in", bg = 'gray', font = buttonFont, command = self.loginClicked(usernameEntry.get(), passwordEntry.get()))
    loginButton.place(relx = 0.25, rely = 0.7, relheight = 0.1, relwidth = 0.25)

    quitButton = tk.Button(mainframe, text = "Quit", bg = 'gray', font = buttonFont, command = self.quitClicked)
    quitButton.place(relx = 0.55, rely = 0.7, relheight = 0.1, relwidth = 0.25)

    forgotPasswordLabel = tk.Label(mainframe, text = "Forgot password", bg = '#80c1ff', fg = 'blue', font = defaultFont)
    forgotPasswordLabel.place(relx = 0.23, rely = 0.83, relheight = 0.1, relwidth = 0.2)

    createAccountLabel = tk.Label(mainframe, text = "Create Account", bg = '#80c1ff', fg = 'blue', font = defaultFont)
    createAccountLabel.place(relx = 0.77, rely = 0.87, relheight = 0.1, relwidth = 0.2)

    aLogin.mainloop()
```

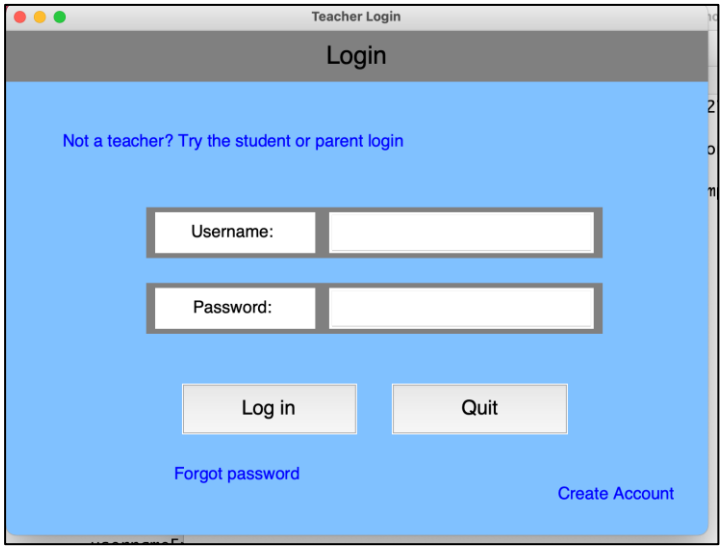
The commands of the login button and the quit button are as follows:

```
def loginClicked(self, usernameEntry, passwordEntry):
    print('Login')
    username = usernameEntry
    password = passwordEntry
    print (username + password)

def quitClicked(self):
    print('Quit')
```

The login command takes in the inputs from the two entry boxes (using the .get() function), and passes it into the loginClicked function as parameters. For the purpose of testing to see whether the button click is registered, I simply create this function to output the two inputted values.

For the quit button, I initially set the command as aLogin.destroy() to terminate the window, however I later changed it when considering the possibility of a quit confirmation page in the event of mis-clicks.

This produced:



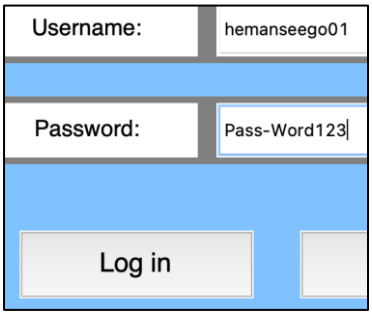
Whilst the interface came up as desired, there was an issue with the functions of the buttons. Namely, the loginClicked function seemed to execute without pressing the button.

```
Login

>>>
```

Upon further research, it became clear that tkinter had no option for specifying parameters to be passed to button commands as in the case with the loginClicked function, which required the usernameEntry and passwordEntry parameters. To combat this, I made use of the lambda function. The lambda function allowed the program to send specific data to the loginClicked function within tkinter. I modified part of the code to read this:

```
command = lambda: self.loginClicked(usernameEntry.get(), passwordEntry.get()))
```

This then allowed me to make use of the entry boxes and subsequently the login button.




```
Login
hemanseego01Pass-Word123
```

I shall show the validation of the entries and the actual functions of the buttons later on once the design of all login interface pages are complete.

Since the labels for the student and parent pages to access a different login were different to that of the admin page, I found that I had to create separate pages with that small adjustment. With the rest of the code the same, the differentLoginLabel for each read –
For the student login page:

```
differentLoginLabel = tk.Label(mainframe, text = "Not a student? Try the teacher or parent login", bg = '#80c1ff', fg = 'blue', font = defaultFont)
differentLoginLabel.place(relx = 0.05, rely = 0.17, relheight = 0.1, relwidth = 0.55)
```

For the parent login page:

```
differentLoginLabel = tk.Label(mainframe, text = "Not a parent? Try the teacher or student login", bg = '#80c1ff', fg = 'blue', font = defaultFont)
differentLoginLabel.place(relx = 0.05, rely = 0.17, relheight = 0.1, relwidth = 0.55)
```

This was extremely inefficient since there were large sets of repeated code. If any adjustment or update had to be made to the interface, it would take a significant amount of time since many parts of the code would have to be modified. It would also be very long and difficult to read and trace if need be. Therefore, I modified the code. Firstly, I changed the functions of the main login page buttons to look like this:

```
def teacherButtonClicked(self):
    print('Teacher Clicked')
    ## Retrieves the assigned width, height and type for the admin login page
    loginType = adminLogin.getType()
    if loginType == 'Admin':
        screenWidth = adminLogin.getScreenWidth()
        screenHeight = adminLogin.getScreenHeight()
        title = 'Teacher Login'
        differentLogin = 'Not a teacher? Try the student or parent login'
    loginPages(screenWidth, screenHeight, title, differentLogin, titleFont, buttonFont, defaultFont)

def studentButtonClicked(self):
    print('Student Clicked')
    ## Retrieves the assigned width, height and type for the student login page
    loginType = studentLogin.getType()
    if loginType == 'Student':
        screenWidth = studentLogin.getScreenWidth()
        screenHeight = studentLogin.getScreenHeight()
        title = 'Student Login'
        differentLogin = 'Not a student? Try the teacher or parent login'
    loginPages(screenWidth, screenHeight, title, differentLogin, titleFont, buttonFont, defaultFont)

def parentButtonClicked(self):
    print('Parent Clicked')
    ## Retrieves the assigned width, height and type for the student login page
    loginType = parentLogin.getType()
    if loginType == 'Parent':
        screenWidth = parentLogin.getScreenWidth()
        screenHeight = parentLogin.getScreenHeight()
        title = 'Parent Login'
        differentLogin = 'Not a parent? Try the teacher or student login'
    loginPages(screenWidth, screenHeight, title, differentLogin, titleFont, buttonFont, defaultFont)
```

In this modified code, the button functions fetch the details of their relevant pages through the get functions of the class and assign all the various variables that would be used in the loginPages function (previously what were separately the adminLoginPage, studentLoginPage and parentLoginPage functions). However even this modified code showed a lot of inefficiency due to excessive repetition so I further modified it to look like this:

```
def teacherButtonClicked(self):
    print('Teacher Clicked')
    ## Retrieves the assigned type for the admin login page
    loginType = adminLogin.getType()
    screenWidth, screenHeight, title, differentLogin = self.loginPageDetails(loginType)
    self.loginPages(screenWidth, screenHeight, title, differentLogin, self.titleFont, self.buttonFont, self.defaultFont)

def studentButtonClicked(self):
    print('Student Clicked')
    ## Retrieves the assigned type for the student login page
    loginType = studentLogin.getType()
    screenWidth, screenHeight, title, differentLogin = self.loginPageDetails(loginType)
    self.loginPages(screenWidth, screenHeight, title, differentLogin, self.titleFont, self.buttonFont, self.defaultFont)

def parentButtonClicked(self):
    print('Parent Clicked')
    ## Retrieves the assigned type for the student login page
    loginType = parentLogin.getType()
    screenWidth, screenHeight, title, differentLogin = self.loginPageDetails(loginType)
    self.loginPages(screenWidth, screenHeight, title, differentLogin, self.titleFont, self.buttonFont, self.defaultFont)
```

Which made use of a separate function loginPageDetails which decided all the details of the login page:

```
def loginPageDetails(self, loginType):
    ## Retrieves the assigned width, height and type for the relevant login page and accordingly assigns certain text to variables for login page
    if loginType == 'Admin':
        screenWidth = adminLogin.getScreenWidth()
        screenHeight = adminLogin.getScreenHeight()
        title = 'Teacher Login'
        differentLogin = 'Not a teacher? Try the student or parent login'
    elif loginType == 'Student':
        screenWidth = studentLogin.getScreenWidth()
        screenHeight = studentLogin.getScreenHeight()
        title = 'Student Login'
        differentLogin = 'Not a student? Try the teacher or parent login'
    elif loginType == 'Parent':
        screenWidth = parentLogin.getScreenWidth()
        screenHeight = parentLogin.getScreenHeight()
        title = 'Parent Login'
        differentLogin = 'Not a parent? Try the teacher or student login'
    return screenWidth, screenHeight, title, differentLogin
```

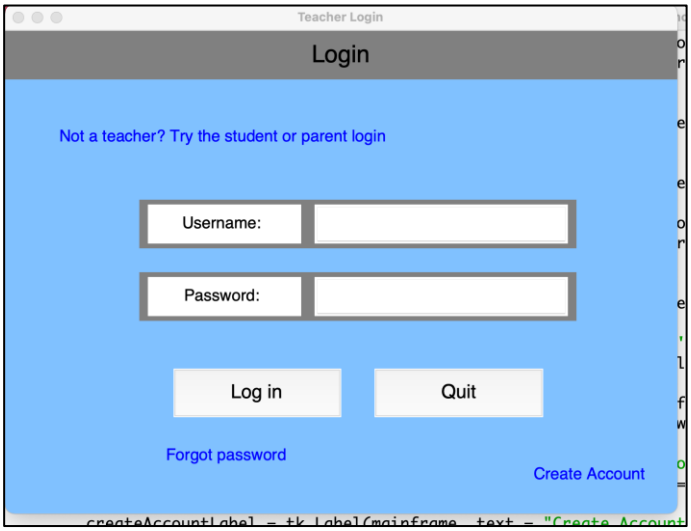
The loginPages function:

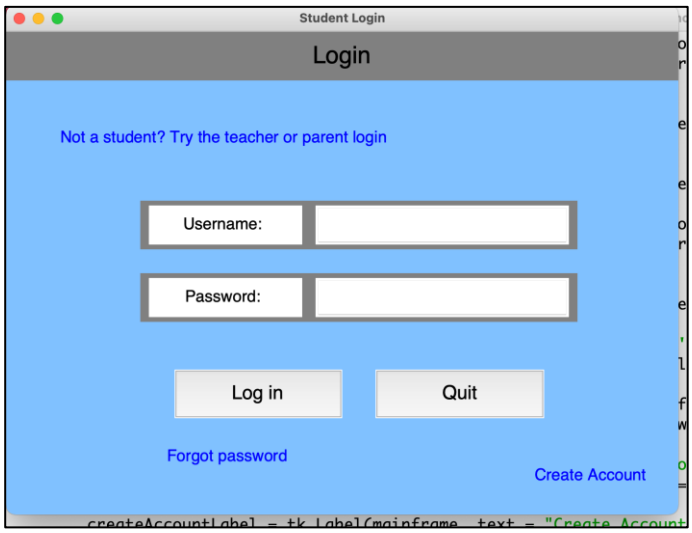
```
def loginPages(self, screenWidth, screenHeight, title, differentLogin, titleFont, buttonFont, defaultFont):
    loginPage = tk.Tk()
    loginPage.title(title)

    canvas = tk.Canvas(loginPage, height = screenHeight, width = screenWidth)
    canvas.pack()

    mainframe = tk.Frame(loginPage, bg = '#80c1ff')
    mainframe.place(relheight = 1, relwidth = 1)

    welcomeLabel = tk.Label(mainframe, text = "Login", bg = 'gray', font = titleFont)
    welcomeLabel.place(relx = 0, rely = 0, relheight = 0.1, relwidth = 1)

    differentLoginLabel = tk.Label(mainframe, text = differentLogin, bg = '#80c1ff', fg = 'blue', font = defaultFont)
    differentLoginLabel.place(relx = 0.05, rely = 0.17, relheight = 0.1, relwidth = 0.55)

    usernameFrame = tk.Frame(mainframe, bg = 'gray')
    usernameFrame.place(relx = 0.2, rely = 0.35, relheight = 0.1, relwidth = 0.65)

    usernameLabel = tk.Label(usernameFrame, text = "Username: ", font = defaultFont)
    usernameLabel.place(relx = 0.02, rely = 0.1, relheight = 0.8, relwidth = 0.35)

    usernameEntry = tk.Entry(usernameFrame)
    usernameEntry.place(relx = 0.4, rely = 0.1, relheight = 0.8, relwidth = 0.58)

    passwordFrame = tk.Frame(mainframe, bg = 'gray')
    passwordFrame.place(relx = 0.2, rely = 0.5, relheight = 0.1, relwidth = 0.65)

    passwordLabel = tk.Label(passwordFrame, text = "Password: ", font = defaultFont)
    passwordLabel.place(relx = 0.02, rely = 0.1, relheight = 0.8, relwidth = 0.35)

    passwordEntry = tk.Entry(passwordFrame)
    passwordEntry.place(relx = 0.4, rely = 0.1, relheight = 0.8, relwidth = 0.58)
```

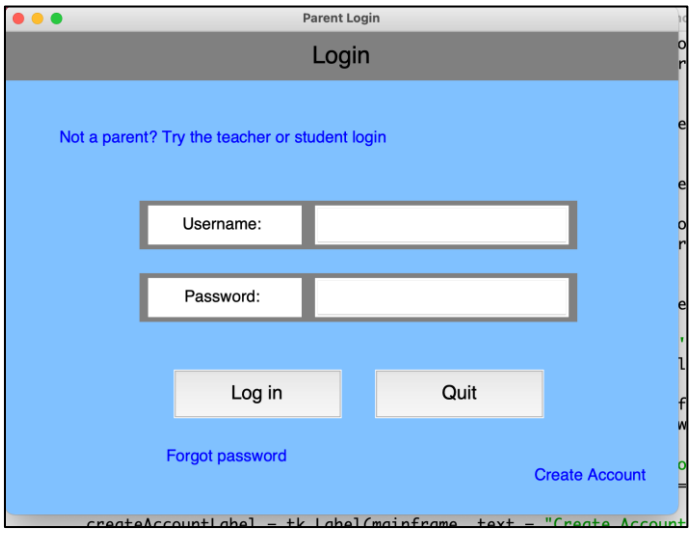
```
    loginButton = tk.Button(mainframe, text = "Log in", bg = 'gray', font = buttonFont, command = lambda: self.loginClicked(usernameEntry.get(),
                                                                                                                              passwordEntry.get()))
```

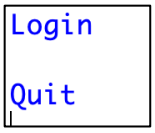
```
loginButton.place(relx = 0.25, rely = 0.7, relheight = 0.1, relwidth = 0.25)

quitButton = tk.Button(mainframe, text = "Quit", bg = 'gray', font = buttonFont, command = self.quitClicked)
quitButton.place(relx = 0.55, rely = 0.7, relheight = 0.1, relwidth = 0.25)

forgotPasswordLabel = tk.Label(mainframe, text = "Forgot password", bg = '#80c1ff', fg = 'blue', font = defaultFont)
forgotPasswordLabel.place(relx = 0.23, rely = 0.83, relheight = 0.1, relwidth = 0.2)

createAccountLabel = tk.Label(mainframe, text = "Create Account", bg = '#80c1ff', fg = 'blue', font = defaultFont)
createAccountLabel.place(relx = 0.77, rely = 0.87, relheight = 0.1, relwidth = 0.2)

loginPage.mainloop()
```

Produced:







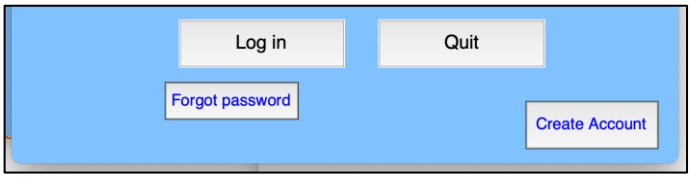


Initially, for aesthetic purposes, I wanted to keep certain links as labels since it looks more professional such as the forgot password label, the create account label and the different login label. However, to simplify the code and make it easier to read and understand, I prevented this where I could. I changed the forgot password label and the create account label into respective buttons.

```
forgotPasswordButton = tk.Button(mainframe, text = "Forgot password", highlightbackground = '#424949', fg = 'blue', font = defaultFont,
forgotPasswordButton.place(relx = 0.23, rely = 0.83, relheight = 0.08, relwidth = 0.2)
                                                                                    command = self.forgotPasswordClicked)
```

```
createAccountButton = tk.Button(mainframe, text = "Create Account", highlightbackground = '#424949', fg = 'blue', font = defaultFont,
createAccountButton.place(relx = 0.77, rely = 0.87, relheight = 0.1, relwidth = 0.2)
                                                                                    command = self.createAccountClicked)
```

```
def forgotPasswordClicked(self):
    print('Forgot Password Clicked')

def createAccountClicked(self):
    print('Create Account Clicked')
```




```
Forgot Password Clicked
Create Account Clicked
```

Since I didn't want to add buttons in the middle of the label as I would need to do to allow 'teacher', 'student' or 'parent' to act as a link to another page (as it would look extremely unprofessional), I instead decided to leave that as a label. This caused issues since firstly, I didn't want the whole label to link to another page and secondly, there were two links that would have to be included in the label. In order to make it work as I wanted, I had to separate the label into different parts. To do this I modified the loginPageDetails function:

```
def loginPageDetails(self, loginType):
    ## Creates variables for fixed parts of message used in loginPages function
    differentLogin1 = 'Not a student? Try the'
    differentLogin3 = ' or '
    differentLogin5 = ' login'
    ## Retrieves the assigned width, height and type for the relevant login page and accordingly assigns certain text to variables for login page
    if loginType == 'Admin':
        screenWidth = adminLogin.getScreenWidth()
        screenHeight = adminLogin.getScreenHeight()
        title = 'Teacher Login'
        ## Creates variables for parts of different login message in loginPages function which change with the type of login page
        differentLogin2 = 'Student'
        differentLogin4 = 'Parent'
    elif loginType == 'Student':
        screenWidth = studentLogin.getScreenWidth()
        screenHeight = studentLogin.getScreenHeight()
        title = 'Student Login'
        differentLogin2 = 'Teacher'
        differentLogin4 = 'Parent'
    elif loginType == 'Parent':
        screenWidth = parentLogin.getScreenWidth()
        screenHeight = parentLogin.getScreenHeight()
        title = 'Parent Login'
        differentLogin2 = 'Teacher'
        differentLogin4 = 'Student'
    return screenWidth, screenHeight, title, differentLogin1, differentLogin2, differentLogin3, differentLogin4, differentLogin5
```

Here, I separated the different login label into five parts, three of which were constant throughout the different pages so were determined beforehand, outside the if statement. Then, based on the type of login page, the relevant parts of the different login label were assigned. To make this change consistent throughout the rest of the code, I had to modify a few lines.

```
def loginPages(self, screenWidth, screenHeight, title, differentLogin1, differentLogin2, differentLogin3, differentLogin4, differentLogin5,
               titleFont, buttonFont, defaultFont):
```

Where the loginPages function took in just the singular differentLogin variable, it now takes all five parts of the label.

```
def studentButtonClicked(self):
    print('Student Clicked')
    ## Retrieves the assigned type for the student login page
    loginType = studentLogin.getType()
    screenWidth, screenHeight, title, differentLogin1, differentLogin2, differentLogin3, differentLogin4, differentLogin5 = self.loginPageDetails(loginType)
    self.loginPages(screenWidth, screenHeight, title, differentLogin1, differentLogin2, differentLogin3, differentLogin4, differentLogin5, self.titleFont,
                    self.titleFont, self.buttonFont, self.defaultFont)
```

I had to modify the button functions since they didn't account for the new variables created. I did this for all the teacher, student and parent button.

I replaced the differentLoginLabel code shown above with this:

```
differentLoginLabel1 = tk.Label(mainframe, text = differentLogin1, bg = 'gray', font = defaultFont)
differentLoginLabel1.place(relx = 0.05, rely = 0.17, relheight = 0.1, relwidth = 0.25)

differentLoginLabel2 = tk.Label(mainframe, text = differentLogin2, bg = 'gray', fg = 'blue', font = defaultFont)
differentLoginLabel2.place(relx = 0.3, rely = 0.17, relheight = 0.1, relwidth = 0.09)
differentLoginLabel2.bind("<Button-1>", lambda x: self.changeLogin(differentLogin2))

differentLoginLabel3 = tk.Label(mainframe, text = differentLogin3, bg = 'gray', font = defaultFont)
differentLoginLabel3.place(relx = 0.39, rely = 0.17, relheight = 0.1, relwidth = 0.03)

differentLoginLabel4 = tk.Label(mainframe, text = differentLogin4, bg = 'gray', fg = 'blue', font = defaultFont)
differentLoginLabel4.place(relx = 0.42, rely = 0.17, relheight = 0.1, relwidth = 0.09)
differentLoginLabel4.bind("<Button-1>", lambda x: self.changeLogin(differentLogin4))

differentLoginLabel5 = tk.Label(mainframe, text = differentLogin5, bg = 'gray', font = defaultFont)
differentLoginLabel5.place(relx = 0.51, rely = 0.17, relheight = 0.1, relwidth = 0.07)
```

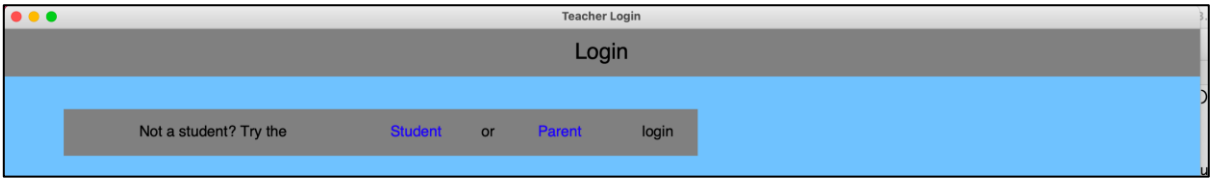
Here I placed the different parts of the label side by side to create the illusion that it is one. I also changed the background of the labels and the foreground of the labels with links for aesthetic purposes. For the labels which were to be linked (differentLoginLabel2 and differentLoginLabel4), I created a bind to an event – “<Button-1>”. This meant that this label behaved like a button, but with the properties of a label. I have used “<Button-1>” here to represent a left click of the mouse. When creating a bind as such, tkinter passes the event as an argument to the desired function which required me to use lambda x, x representing the passed on argument. This prevented this error from appearing:

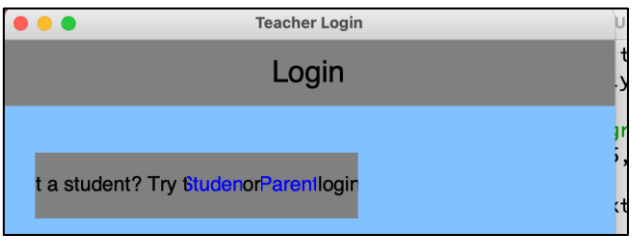
```
TypeError: <lambda>() takes 0 positional arguments but 1 was given
```

To successfully change the login page, I created a changeLogin function which took the label itself as a parameter since the label read the login type desired:

```
def changeLogin(self, labelText):
    if labelText == 'Teacher':
        print(labelText)
        self.teacherButtonClicked()
    elif labelText == 'Student':
        print(labelText)
        self.studentButtonClicked()
    elif labelText == 'Parent':
        print(labelText)
        self.parentButtonClicked()
```

Which successfully allowed the user to change login pages. However, there was a huge disadvantage with this method. Since I used the tkinter place method to organise the widgets on my pages, I permitted a degree of flexibility within them. Yet by placing the labels in this way, when stretched, they lost their side by side placement:





To extend the login page, I created a bind to the username and password entries, which would allow the enter button to trigger the login button:

```
usernameEntry = tk.Entry(usernameFrame)
usernameEntry.place(relx = 0.4, rely = 0.1, relheight = 0.8, relwidth = 0.58)
usernameEntry.bind("<Return>", lambda x: self.loginClicked(usernameEntry.get(), passwordEntry.get()))
```

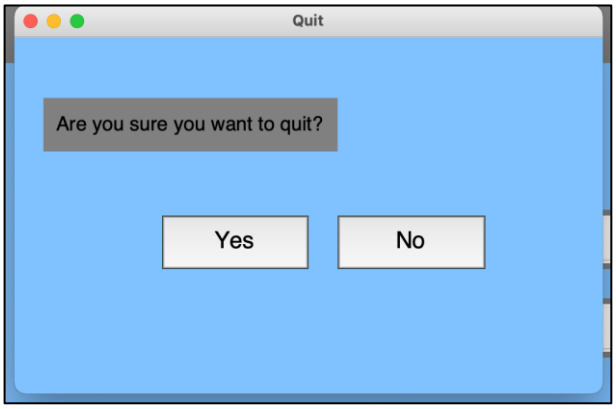
```
passwordEntry = tk.Entry(passwordFrame)
passwordEntry.place(relx = 0.4, rely = 0.1, relheight = 0.8, relwidth = 0.58)
passwordEntry.bind("<Return>", lambda x: self.loginClicked(usernameEntry.get(), passwordEntry.get()))
```

This makes the program easier for the user to use.

Although the code produced thus far allows me to successfully open the main login page and navigate between the teacher, student and parent login pages, it doesn't close any prior windows that have been in use – a very unprofessional aspect of the program. Hence, I looked at closing windows, both manually and automatically. Firstly, I created a quit confirmation page for when the quit button is clicked:

```
def quitClicked(self, previousPage, titleFont, buttonFont, defaultFont):
    ## Quit confirmation page
    ## Retrieves the assigned width and height for the quit confirmation page
    mainScreenWidth = quitConfirmation.getScreenWidth()
    mainScreenHeight = quitConfirmation.getScreenHeight()

    quitPage = tk.Tk()
    quitPage.title("Quit")

    canvas = tk.Canvas(quitPage, height = mainScreenHeight, width = mainScreenWidth)
    canvas.pack()

    mainframe = tk.Frame(quitPage, bg = '#80c1ff')
    mainframe.place(relheight = 1, relwidth = 1)

    confirmationLabel = tk.Label(mainframe, text = 'Are you sure you want to quit?', bg = 'gray', font = defaultFont)
    confirmationLabel.place(relx = 0.05, rely = 0.17, relheight = 0.15, relwidth = 0.5)

    yesButton = tk.Button(mainframe, text = "Yes", highlightbackground = '#424949', font = buttonFont, command = lambda: (previousPage.destroy(), quitPage.destroy()))
    yesButton.place(relx = 0.25, rely = 0.5, relheight = 0.15, relwidth = 0.25)

    noButton = tk.Button(mainframe, text = "No", highlightbackground = '#424949', font = buttonFont, command = quitPage.destroy)
    noButton.place(relx = 0.55, rely = 0.5, relheight = 0.15, relwidth = 0.25)

    quitPage.mainloop()
```

I modified the quitClicked function to create the following interface:



Here, the No button prompts the program to close just this confirmation page, whereas the Yes button prompts the program to close both this page and the previous page passed into this function as a parameter.

To create the dimensions of this page, I created a new instance of the Interface class:

```
quitConfirmation = Interface(500, 300)
```

When considering how to automatically close the previous windows when opening a new one, I stumbled upon an inefficiency in my code where there was a lot of repeated code. This was on the main login page where each button initially retrieved the type of login page from their respective class objects, however, the rest of the code that followed was the same. To get rid of this inefficiency, I created an additional function and modified the button functions:

```
def teacherButtonClicked(self, root):
    print('Teacher Clicked')
    ## Retrieves the assigned type for the admin login page
    loginType = adminLogin.getType()
    self.newLoginPage(loginType, root)

def studentButtonClicked(self, root):
    print('Student Clicked')
    ## Retrieves the assigned type for the student login page
    loginType = studentLogin.getType()
    self.newLoginPage(loginType, root)

def parentButtonClicked(self, root):
    print('Parent Clicked')
    ## Retrieves the assigned type for the student login page
    loginType = parentLogin.getType()
    self.newLoginPage(loginType, root)

def newLoginPage(self, loginType, root):
    screenWidth, screenHeight, title, differentLogin1, differentLogin2, differentLogin3, differentLogin4, differentLogin5 = self.loginPageDetails(loginType)
    root.destroy()
    self.loginPages(screenWidth, screenHeight, title, differentLogin1, differentLogin2, differentLogin3, differentLogin4, differentLogin5, self.titleFont, self.bu
```

Since the buttons all had to perform the same tasks anyways, it was more efficient to create an additional function for this purpose. The root variable has been passed along each button function and the newly created newLoginPage function to allow me to destroy the previous window (done by root.destroy(), root being the previous page). Because of this, I had to modify the code in the main login page function under command, to allow for a lambda function since a parameter is now being passed onto these functions.

```
command = lambda: self.teacherButtonClicked(root))
```

```
command = lambda: self.studentButtonClicked(root))
```

```
command = lambda: self.parentButtonClicked(root))
```

I then had to change the changeLogin function to take in the previous login page as a parameter:

```
def changeLogin(self, labelText, loginPage):
    if labelText == 'Teacher':
        print(labelText)
        self.teacherButtonClicked(loginPage)
    elif labelText == 'Student':
        print(labelText)
        self.studentButtonClicked(loginPage)
    elif labelText == 'Parent':
        print(labelText)
        self.parentButtonClicked(loginPage)
```

In turn, I had to modify parts of the loginPages function:

```
differentLoginLabel2.bind("<Button-1>", lambda x: self.changeLogin(differentLogin2, loginPage))
```

```
differentLoginLabel4.bind("<Button-1>", lambda x: self.changeLogin(differentLogin4, loginPage))
```

To allow the loginPage variable to be taken as a parameter, to allow it to be destroyed.
By modifying the code in this way, I was able to destroy a window upon opening a new one.

Next, I went on to create the create account page:

```
def createAccountClicked(self, titleFont, buttonFont, defaultFont):
    print('Create Account Clicked')
    ## Retrieves the assigned width and height for the create account page
    mainScreenWidth = createAccount.getScreenWidth()
    mainScreenHeight = createAccount.getScreenHeight()

    createAccountPage = tk.Tk()
    createAccountPage.title("Create Account")

    canvas = tk.Canvas(createAccountPage, height = mainScreenHeight, width = mainScreenWidth)
    canvas.pack()

    mainframe = tk.Frame(createAccountPage, bg = '#80c1ff')
    mainframe.place(relheight = 1, relwidth = 1)

    welcomeLabel = tk.Label(mainframe, text = "Create Account", bg = 'gray', font = titleFont)
    welcomeLabel.place(relx = 0, rely = 0, relheight = 0.1, relwidth = 1)

    selectTypeLabel = tk.Label(mainframe, text = 'Please select the type of account you wish to create:', bg = 'gray', font = defaultFont)
    selectTypeLabel.place(relx = 0.05, rely = 0.15, relheight = 0.07, relwidth = 0.6)

    userTypeList = ttk.Combobox(mainframe, values = ["Teacher", "Student", "Parent"], font = buttonFont)
    userTypeList.place(relx = 0.7, rely = 0.16, relheight = 0.04, relwidth = 0.15)
    userTypeList.current(0)

    enterFirstNameLabel = tk.Label(mainframe, text = 'Enter your first name:', bg = 'gray', font = defaultFont)
    enterFirstNameLabel.place(relx = 0.05, rely = 0.27, relheight = 0.07, relwidth = 0.26)

    firstNameEntry = tk.Entry(mainframe)
    firstNameEntry.place(relx = 0.38, rely = 0.27, relheight = 0.07, relwidth = 0.55)

    enterLastNameLabel = tk.Label(mainframe, text = 'Enter your last name:', bg = 'gray', font = defaultFont)
    enterLastNameLabel.place(relx = 0.05, rely = 0.39, relheight = 0.07, relwidth = 0.26)

    lastNameEntry = tk.Entry(mainframe)
    lastNameEntry.place(relx = 0.38, rely = 0.39, relheight = 0.07, relwidth = 0.55)
```

```
    enterUsernameLabel = tk.Label(mainframe, text = 'Enter a username:', bg = 'gray', font = defaultFont)
    enterUsernameLabel.place(relx = 0.05, rely = 0.51, relheight = 0.07, relwidth = 0.22)

    usernameEntry = tk.Entry(mainframe)
    usernameEntry.place(relx = 0.38, rely = 0.51, relheight = 0.07, relwidth = 0.55)

    enterPasswordLabel = tk.Label(mainframe, text = 'Enter a password:', bg = 'gray', font = defaultFont)
    enterPasswordLabel.place(relx = 0.05, rely = 0.63, relheight = 0.07, relwidth = 0.22)

    passwordEntry = tk.Entry(mainframe)
    passwordEntry.place(relx = 0.38, rely = 0.63, relheight = 0.07, relwidth = 0.55)

    selectAdminLabel = tk.Label(mainframe, text = 'Please select an admin account to verify your account', bg = 'gray', font = defaultFont)
    selectAdminLabel.place(relx = 0.05, rely = 0.75, relheight = 0.07, relwidth = 0.6)

    adminUsersList = ttk.Combobox(mainframe, values = ["Admin1", "Admin2", "Admin3"], font = buttonFont)
    adminUsersList.place(relx = 0.7, rely = 0.76, relheight = 0.04, relwidth = 0.15)
    adminUsersList.current(0)

    createAccountButton = tk.Button(mainframe, text = "Create Account", highlightbackground = '#424949', fg = 'blue', font = defaultFont, command = lambda:
    createAccountButton.place(relx = 0.35, rely = 0.87, relheight = 0.07, relwidth = 0.25)
                                                                                                    self.createUserAccount(userTypeList.get(),
    returnButton = tk.Button(mainframe, text = "Return", highlightbackground = '#424949', fg = 'blue', font = defaultFont)
    returnButton.place(relx = 0.83, rely = 0.92, relheight = 0.05, relwidth = 0.14)                 firstNameEntry.get(), lastNameEntry.get(),
                                                                                                    usernameEntry.get(), passwordEntry.get(),
    createAccountPage.mainloop()                                                                                                    adminUsersList.get()))
```

For this code to work, I had to import another library within tkinter: ttk, to allow me to make use of comboboxes, i.e. drop down lists where the user can choose options.

```
from tkinter import ttk
```

For the comboboxes, I had to assign the values that were to be included in the list, and determine which value would show when the page was first opened (done by …List.current(0)).

Since this page had a lot more information to fit on, I had to regulate them by creating a new instance of the Interface class:

```
createAccount = Interface(700, 800)
```

I created a new function for this new create account button:

```
def createUserAccount(self, userTypeList, firstNameEntry, lastNameEntry, usernameEntry, passwordEntry, adminUsersList):
    print('Create User Account')

    userType = userTypeList
    firstName = firstNameEntry
    lastName = lastNameEntry
    username = usernameEntry
    password = passwordEntry
    adminUser = adminUsersList

    print(userType)
    print(firstName)
    print(lastName)
    print(username)
    print(password)
    print(adminUser)
```

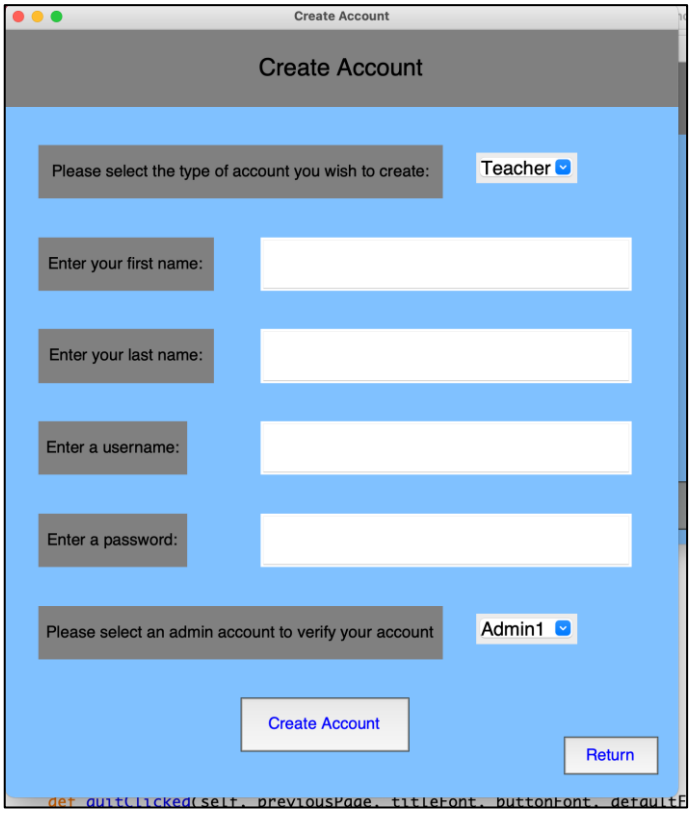
Simply as a matter of testing, I created this function to take in the inputs of the entries in the above page and display them. I shall extend this later on to fully create an account using these details and my database.

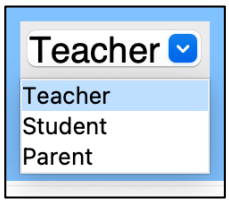
When first running this code, when just trying to run the main login page, both the main login page and the create account page appeared. To allow this to work correctly, I had to change the commands of the create account buttons on the main login page and the login pages to this:

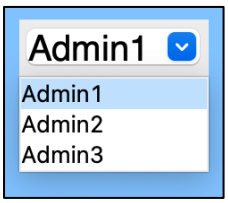
```
command = lambda: self.createAccountClicked(self.titleFont, self.buttonFont, self.defaultFont))
```

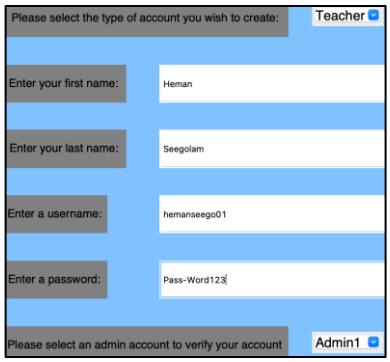
To allow the font variables to be passed on as parameters, and allow the page only to come up upon the clicks of the relevant button.

This now produced:










```
Teacher Clicked
Create Account Clicked
Create User Account
Teacher
Heman
Seegolam
hemanseego01
Pass-Word123
Admin1
```

However, there was a limitation to this method. The combo-boxes allowed the user to enter their own input as well as choose the options provided and this could cause a lot of issues within my program. Since these values would not have to be validated, since predetermined, if they can be changed in any way it can compromise the data of all users, especially when making transactions with the database, it could lead to incidents of SQL injection.
For example:

```
accountType = "accountType"
sqlView = "SELECT * FROM Users WHERE accountType = " + accountType
```

If a user wished to retrieve all the stored data about them (as permitted under the General Data Protection Regulation), they may be allowed to retrieve all data related to them in the database. If instead of Admin, Student or Parent, if 'accountType' was passed in as the account type to the SQL statement, the data of all users would show since accountType would be equal to accountType for all users.

```
accountType = "Teacher OR 1=1"
sqlView = "SELECT * FROM Users WHERE accountType = " + accountType
```

Similarly, if they extended it in this way, the program would read:

```
"SELECT * FROM Users WHERE accountType = Teacher OR 1=1"
```

This allows the database to choose between two conditions, whether accountType = Teacher or 1=1. If indeed the account type is not Teacher, then the program would submit to the latter condition, again producing the data of all users. Teacher does not necessarily have to be written in this example and would provide fool proof access to all data. Even 1=1 is interchangeable with other expressions which produce equivalent results such as "" = "", etc. I will also have to investigate and ensure this problem is completely inoperable at later stages within my program such as when coding the login function.

As well as gaining unauthorised access to data, users may also be able to corrupt data in this way:

```
accountType = "INJECTION; DROP TABLE UserDetails;"
sqlView = "SELECT * FROM Users WHERE accountType = " + accountType
```

Which would look like:

```
"SELECT * FROM Users WHERE accountType = INJECTION; DROP TABLE UserDetails;"
```

And would consequently delete the whole UserDetails table. In fact this transaction can be filled with any other and should, in theory, be executed. This can be avoided in my login function by creating a character limit, preventing certain characters or words, etc.

Even without the necessity that SQL injection has been committed, it may also be the case that values are altered in a way in which the program doesn't recognise and is therefore unable to deal with.

To combat this issue, I configured the combo-box to be in a read-only state:

```
userTypeList = ttk.Combobox(mainframe, values = ["Teacher", "Student", "Parent"], font = buttonFont, state = 'readonly')
```

By doing so, the user could not alter the options included in the combo-box

To destroy the previous page upon opening this page, I extended the function:

```
def createAccountClicked(self, previousPage, titleFont, buttonFont, defaultFont):
    print('Create Account Clicked')
    # Destroys previous page
    previousPage.destroy()
```

I made this change consistent throughout the code:
For the main login page:

```
command = lambda: self.createAccountClicked(root, self.titleFont, self.buttonFont, self.defaultFont))
```

For the login pages:

```
command = lambda: self.createAccountClicked(loginPage, self.titleFont, self.buttonFont, self.defaultFont))
```

I chose this method in favour of lambda (previousPage.destroy(), self.createAccountClicked…), since I would have to repeat this code a lot throughout for every link to the create account page. Therefore I elected the above method since less repetition would be required.
By passing on the previous page as a parameter, I also tried to create the return button function.

I created a command for the return button:

```
command = lambda: self.returnPage(previousPage))
```

Along with the relevant function:

```
def returnPage(self, previousPage):
    print('Return')
```

I initially wrote this function just to test whether the command was successful and it produced:

```
Create Account Clicked
Return
```

As desired.

However, there wasn't a way in which the previous page could be restored once closed. As a result, in favour of simplicity, I wrote:

```
def returnPage(self, page):
    print('Return')
    page.destroy()
    self.mainLoginPage(self.titleFont, self.buttonFont, self.defaultFont)
```

```
command = lambda: self.returnPage(createAccountPage))
```

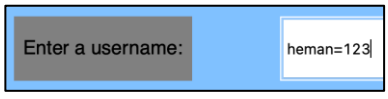
To allow the user to return to the main login page. The obvious limitation here was that I couldn't return to the last specific page that the user was on (if not the main login page), in the case of mis-clicks, and this could be inconvenient for the user.

Now linking the create account page to the database class and file, I modified the create account function. From above, I commented out all the print statements now that they indeed proved to work, all except the username. This is because I decided to test whether the inputted username could be checked against all usernames already stored in the database to show if it was taken and also to validate the username.

```
takenUsername = False
invalidUsername = False
weakPassword = False

## Check whether inputted username is already taken
## Fetch all usernames from the database and put it in a list
c.execute("SELECT username FROM UserDetails")
takenUsernames = c.fetchall()
print(takenUsernames)
## Check inputted username against list of current usernames
for each in takenUsernames:
    if each[0] == username:
        print('Yes')
        takenUsername = True
    else:
        print('No')
print(takenUsername)

print('\n')

## Check whether inputted username is valid (to prevent SQL injection)
invalidUsername = '=' in username
print(invalidUsername)
if invalidUsername == False:
    invalidUsername = ';' in username
print(invalidUsername)
if invalidUsername == False:
    invalidUsername = ' ' in username
print(invalidUsername)
if username == 'username':
    invalidUsername = True
print(invalidUsername)
```

The way this function is to work is as follows. There will be a number of determining variables all assigned to initially be False. After all checks, i.e. whether the username is taken, whether the username is valid, whether the password is strong enough, etc. (to come later), if any variables are shown to be true, then there is an issue with creating the account and the issue can be shown through which variable is true.

In this function, the program retrieves all the usernames from the database and assigns them to a variable. As shown previously, when retrieving values from a database in this way, the values are stored in tuples, within a list. For this reason, the first value of each tuple (each[0]) is taken and matched against the inputted username. If any match, the takenUsername variable is set to True. I have chosen to display Yes and No to represent whether the usernames match or not for testing purposes in favour of clarity. For this reason I have also created a space between the takenUsername variable and the invalidUsername variable using '\n'.
The invalid username test follows the examples shown above of SQL injection. The main characters that would have to be avoided in the username included an '=', ';' and a space. By checking for these characters, I hope to avoid any instances of 'OR x=x', or 'example; DROP table exampleTable'. Also by checking for spaces, it avoids any confusion SQL may face when using that username in transactions. Finally, I have provided a check for whether the inputted username is 'username' which, as shown above, can compromise the security of all users' data.
When creating this latter part of the function, I had to make sure that the invalidUsername variable was not overwritten by different parts of the tests. For example, it shouldn't first turn True after the first check but then turn False again after the last check. This obviously wouldn't work as needed. Therefore, I carried out the first check as required (and displayed the result of the test). From there, I then used if statements for the rest of the checks, requiring that the invalidUsername must first be False for them to be executed. This ensures that once this variable turns True, it doesn't revert back to False due to a different part of validation.



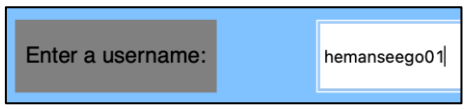

```
Create Account Clicked
Create User Account
heman=123
[('hemanseego01',), ('sampinch02',)]
No
No
False


True
True
True
True
```


```
Create User Account
heman;123
[('hemanseego01',), ('sampinch02',)]
No
No
False


False
True
True
True
```

```
Create User Account
heman 123
[('hemanseego01',), ('sampinch02',)]
No
No
False


False
False
True
True
```


```
Create User Account
username
[('hemanseego01',), ('sampinch02',)]
No
No
False


False
False
False
True
```

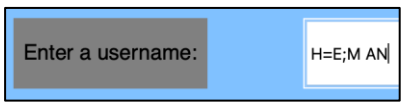
```
Create User Account
hemanseego01
[('hemanseego01',), ('sampinch02',)]
Yes
No
True


False
False
False
False
```



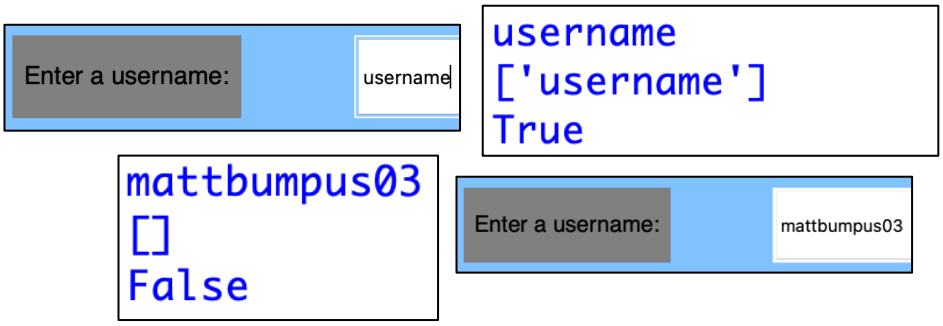
Although this method works in the simplest notion of its purpose, it fails to recognise explicitly where the user has failed to meet the requirements. Hence I modified the code to create a list of errors.

```
## Check whether inputted username is valid (to prevent SQL injection)
## Create list of all errors
errorList = []
if '=' in username:
    errorList.append('=')
if ';' in username:
    errorList.append(';')
if ' ' in username:
    errorList.append('Space')
if username == 'username':
    errorList.append('username')
print(errorList)
if len(errorList) != 0:
    invalidUsername = True
print(invalidUsername)
```

In this modified version, for every error experienced, it gets added to a list. If the length of the list exceeds 0 after all checks are complete, the invalidUsername variable then turns to True and all errors are stored.




```
H=E;M AN
['=', ';', 'Space']
True
```



To maintain the modularity of the program, I made these validation function separate from the create account function. This maximises the reusability of my functions.

```
def createUserAccount(self, userType, firstName, lastName, username, password, adminUser):
    print('Create User Account')
    print[userType, firstName, lastName, username, password, adminUser]
    takenUsername = self.checkTakenUsername(username)
    invalidUsername = self.checkValidUsername(username)

def checkTakenUsername(self, username):
    ## Check whether inputted username is already taken
    takenUsername = False
    ## Fetch all usernames from the database and put it in a list
    c.execute("SELECT username FROM UserDetails")
    takenUsernames = c.fetchall()
    ## Check inputted username against list of current usernames
    for each in takenUsernames:
        if each[0] == username:
            print('Yes')
            takenUsername = True
    return takenUsername

def checkValidUsername(self, username):
    ## Check whether inputted username is valid (to prevent SQL injection)
    invalidUsername = False
    ## Create list of all errors
    usernameErrorList = []
    if '=' in username:
        usernameErrorList.append('=')
    if ';' in username:
        usernameErrorList.append(';')
    if ' ' in username:
        usernameErrorList.append('Space')
    if username == 'username':
        usernameErrorList.append('username')
    if len(usernameErrorList) != 0:
        invalidUsername = True
    return invalidUsername
```

I also changed the name of the error list to usernameErrorList, since the password would also be having an error list.

Next I validated the inputted password.
In this function, I followed the same format as the above method. Any errors experienced are added into a passwordErrorList which is checked at the end for all the errors. These error lists will help me pass on the relevant issues to the user when creating their account. To locate certain characters within the inputted password, I made use of the re library in python. Therefore I had to first import it:

```
import re
```

```
def checkValidPassword(self, password):
    ## Check whether inputted password is valid (i.e. length > 7, contains number, character, symbol, upper and lower case)
    weakPassword = False
    passwordErrorList = []

    ## Checks to see whether password is greater than 7 characters
    if len(password) < 8:
        passwordErrorList.append('short')

    ## Check to see whether (lower case) character in password
    if not(re.search(r'[a-z]', password)):
        passwordErrorList.append('lowerCase')

    ## Check to see whether (upper case) character in password
    if not(re.search(r'[A-Z]', password)):
        passwordErrorList.append('upperCase')

    ## Check to see whether number in password
    if not(re.search(r'[0-9]', password)):
        passwordErrorList.append('number')

    ## Check to see whether symbol in password
    if not(re.search("[!@£$%^&*()_+-={};:‘“\|`~,<.>/?']", password)):
        passwordErrorList.append('symbol')

    ## Check to see whether input is 'password' or blank (to prevent SQL injection or other errors)
    if password == 'password':
        passwordErrorList.append('password')
    if password == '':
        passwordErrorList.append('empty')

    print(passwordErrorList)

    if len(passwordErrorList) != 0:
        weakPassword = True
    return weakPassword, passwordErrorList
```

Initially, when looking for symbols in the password, an error persisted when running:

```
## Check to see whether symbol in password
if not(re.search("[!@£$%^&*()_+-=[]{};:‘“\|`~,<.>/?']", password)):
    passwordErrorList.append('symbol')
```

This was because I included ‘[‘ and ‘]’ which interfered with the intended brackets for the set and would therefore not recognise any symbols in the password. However, upon taking these brackets out of the symbol selection, the program ran as desired.
Whilst I started to consider the possibility of SQL injection through the password (by disallowing it to simply be ‘password’), I hadn’t considered the effects of ‘=’ and ‘;’. I therefore extended the function slightly.

```
## Check to see whether symbol in password
if (re.search("[=;]", password)):
    passwordErrorList.append('=;')
```

If the program recognises a ‘=’ or ‘;’ in the password, it shall inform the user that these such symbols are not permitted. Whilst the username validation also disallowed spaces, I decided that for the password, they may be used since before entering the database, it will be hashed for security. These validation checks are necessary before storing the data in the database since for the password, admin users will not be allowed to see them to verify since they are private to the user.

However, there was another error. On some occasions where a symbol had not been entered, it wasn’t present in the error list. This was because of ‘-‘. In the re library, this is used to determine a range between two characters – a case of operator overloading. To fix this, I simply put the ‘-‘ at the end of the list, so it didn’t get recognised as a range of two characters.

```
if not(re.search("[!@£$%^&*()_+={};:‘“\|`~,<.>/?'-]", password)):
```

Upon finishing the validation for the password, I noticed that the username validation function wasn't equipped to deal with blank usernames, and so I extended it to firstly disallow blank usernames and to return the username error list:

```
if username == '':
    usernameErrorList.append('empty')
```

```
return invalidUsername, usernameErrorList
```

When testing the function:

```
p
['short', 'upperCase', 'number', 'symbol']


pA
['short', 'number', 'symbol']


pAss1
['short', 'symbol']


pAsswOrd123
['symbol']


pAss-wOrd123
[]


pAss=wOrd123
['=;']
```

```
pAss;wOrd123
['=;']


password
['upperCase', 'number', 'symbol', 'password']


['short', 'lowerCase', 'upperCase', 'number', 'symbol', 'empty']


password; DROP TABLE UserDetails
['number', '=;']


example OR 1=1
['=;']
```

Where firstly the inputted password is displayed, followed by its corresponding error list.

Now I have concluded the testing for my password validation, I am now able to hide the password when it is entered into the page, for maximum security for the user.

```
passwordEntry = tk.Entry(mainframe, show = '*')
passwordEntry.place(relx = 0.38, rely = 0.63, relheight = 0.07, relwidth = 0.55)
passwordEntry.bind("<Return>", lambda x: self.createUserAccount(userTypeList.get(), firstNameEntry.get(), lastNameEntry.get(), usernameEntry.ge
```

I also created a bind in which the enter button triggered the create account function, as for the rest of the entries:

```
firstNameEntry.bind("<Return>", lambda x: self.createUserAccount(userTypeList.get(), firstNameEntry.get(), lastNameEntry.get(), usernameEntry.g
```

```
lastNameEntry.bind("<Return>", lambda x: self.createUserAccount(userTypeList.get(), firstNameEntry.get(), lastNameEntry.get(), usernameEntry.ge
```

```
usernameEntry.bind("<Return>", lambda x: self.createUserAccount(userTypeList.get(), firstNameEntry.get(), lastNameEntry.get(), usernameEntry.ge
```

The password field now showed:

To check for a valid first name and last name:

```
def checkValidName(self, name):
    invalidName = not(name.isalpha())
    print(invalidName)
    return invalidName
```

```
Heman
False

See golam
True

Seegolam
False

Heman1
True

Heman!
True

Seegolam2
True
```

For testing purposes, the admin users for the create account page and the validation function have been Admin1, Admin2 and Admin3. These were to represent all the admin users involved within the program, i.e. all the teachers of the students. I therefore had to change these to show all the actual admin users within the database.
I first had to retrieve all admin users from the database. To test, I filled the database with the following values:

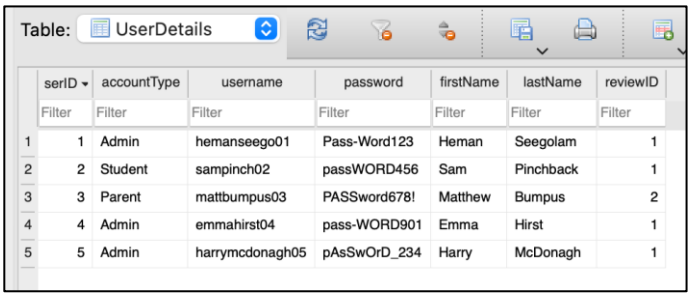


Table: UserDetails

	serID	accountType	username	password	firstName	lastName	reviewID
	Filter	Filter	Filter	Filter	Filter	Filter	Filter
1	1	Admin	hemanseego01	Pass-Word123	Heman	Seegolam	1
2	2	Student	sampinch02	passWORD456	Sam	Pinchback	1
3	3	Parent	mattbumpus03	PASSword678!	Matthew	Bumpus	2
4	4	Admin	emmahirst04	pass-WORD901	Emma	Hirst	1
5	5	Admin	harrymcdonagh05	pAsSwOrD_234	Harry	McDonagh	1

And used my viewFromDatabase function to do so:

```
adminUsers = viewingFromDatabase.viewFromDatabase(c, False, False, 'username', 'accountType', 'Admin', 'UserDetails')
print(adminUsers)
```

I set the allFields and allRecords variable to False, since we were looking for only specific records and fields in the database. I set the field variable to username since I wanted the function to return the usernames of the admin users. Although in the database class, I named it as recordPrimaryKey, this function does in fact work for other fields of the database. Hence, I passed in 'accountType' here and 'Admin' for its corresponding value. The table was set to UserDetails.
Initially, there was the error:

```
sqlite3.OperationalError: no such column: Admin
```

However, this was easily rectifiable by doing this:

```
"'Admin'"
```

Without passing on these quotation marks to the function, the program instead looked for columns named Admin, and since there were not any, it returned an error. However, with these quotation marks, it then looked for the value of Admin within the accountType field. This returned:

```
[('hemanseego01',), ('emmahirst04',), ('harrymcdonagh05',)]
```

As desired.

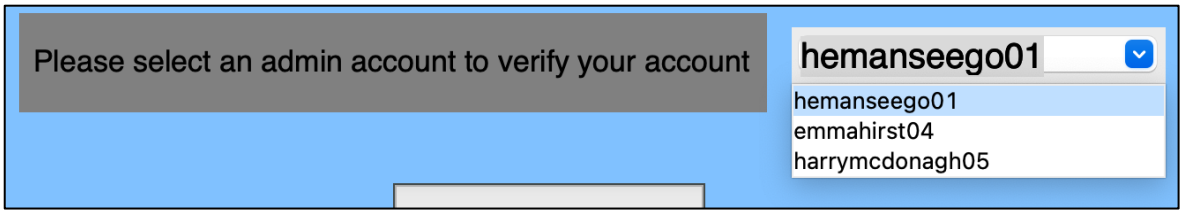
I inserted these values into a list and passed the list onto the combo-box:

```
adminUsers = viewingFromDatabase.viewFromDatabase(c, False, False, 'username', 'accountType', "'Admin'", 'UserDetails')
print(adminUsers)
adminUserList = []
for each in range(len(adminUsers)):
    adminUserList.append(adminUsers[each][0])
print(adminUserList)
adminUsersList = ttk.Combobox(mainframe, values = adminUserList, font = buttonFont)
```

```
[('hemanseego01',), ('emmahirst04',), ('harrymcdonagh05',)]
['hemanseego01', 'emmahirst04', 'harrymcdonagh05']
```

Although it came up in the combo-box as desired, some of the usernames were very long so I extended the width of the combo-box.

```
adminUsersList.place(relx = 0.67, rely = 0.76, relheight = 0.04, relwidth = 0.3)
```



I again made the state of this combo-box read-only:

```
adminUsersList = ttk.Combobox(mainframe, values = adminUserList, font = buttonFont, state = 'readonly')
```

I also changed the checkTakenUsername function to make use of the database class:

```
def checkTakenUsername(self, username):
    ## Check whether inputted username is already taken
    takenUsername = False
    ## Fetch all usernames from the database and put it in a list
    takenUsernames = viewingFromDatabase.viewFromDatabase(c, False, True, 'username', None, None, 'UserDetails')
    ## Check inputted username against list of current usernames
    for each in takenUsernames:
        if each[0] == username:
            print('Yes')
            takenUsername = True
    return takenUsername
```

The create account function therefore looked like this:

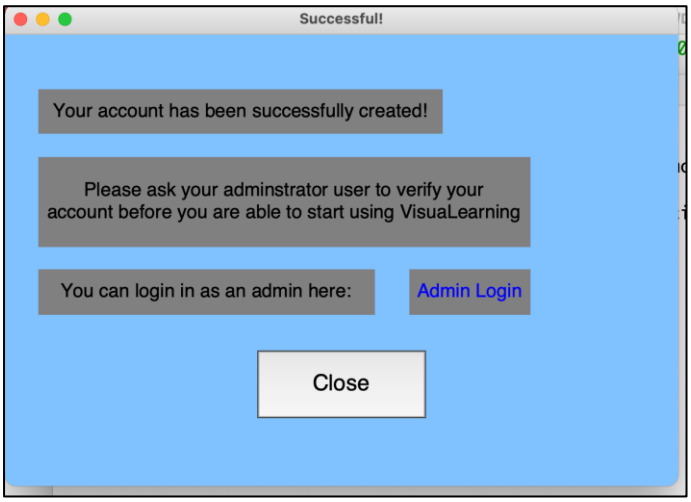
```
def createUserAccount(self, previousPage, userType, firstName, lastName, username, password, adminUser):
    ## Previous page closed
    previousPage.destroy()

    ## Validation checks
    takenUsername = self.checkTakenUsername(username)
    invalidUsername, usernameErrorList = self.checkValidUsername(username)
    weakPassword, passwordErrorList = self.checkValidPassword(password)
    invalidFirstName = self.checkValidName(firstName)
    invalidLastName = self.checkValidName(lastName)

    ## If no inconsistencies...
    if takenUsername == False and invalidUsername == False and weakPassword == False and invalidFirstName == False and invalidLastName == False:
        self.accountSuccessful(self.titleFont, self.buttonFont, self.defaultFont)
    else:
        self.accountFailed(self.titleFont, self.buttonFont, self.defaultFont)
```

Where after all validation checks, all errors were False, the success page is brought up, else, the failed page is brought up. I also included a line of code to destroy the previous page, which is taken as a parameter of the function. This change has been made consistent with the rest of the function.

The accountSuccessful function looks as follows:

```
def accountSuccessful(self, titleFont, buttonFont, defaultFont):
    accountSuccessfulPage = tk.Tk()
    accountSuccessfulPage.title('Successful!')

    mainScreenWidth = succesfulAccount.getScreenWidth()
    mainScreenHeight = succesfulAccount.getScreenHeight()

    canvas = tk.Canvas(accountSuccessfulPage, height = mainScreenHeight, width = mainScreenWidth)
    canvas.pack()

    mainframe = tk.Frame(accountSuccessfulPage, bg = '#80c1ff')
    mainframe.place(relheight = 1, relwidth = 1)

    successfulLabel = tk.Label(mainframe, text = 'Your account has been successfully created!', bg = 'gray', font = defaultFont)
    successfulLabel.place(relx = 0.05, rely = 0.12, relheight = 0.1, relwidth = 0.6)

    askAdminLabel = tk.Label(mainframe, text = 'Please ask your adminstrator user to verify your\naccount before you are able to start using VisuaLearning',
    askAdminLabel.place(relx = 0.05, rely = 0.27, relheight = 0.2, relwidth = 0.73)
                                                                                                    bg = 'gray', font = defaultFont)
    loginLabel = tk.Label(mainframe, text = 'You can login in as an admin here:', bg = 'gray', font = defaultFont)
    loginLabel.place(relx = 0.05, rely = 0.52, relheight = 0.1, relwidth = 0.5)

    adminLoginLabel = tk.Label(mainframe, text = 'Admin Login', bg = 'gray', fg = 'blue', font = defaultFont)
    adminLoginLabel.place(relx = 0.6, rely = 0.52, relheight = 0.1, relwidth = 0.18)
    adminLoginLabel.bind("<Button-1>", lambda x: self.teacherButtonClicked(accountSuccessfulPage))

    closeButton = tk.Button(mainframe, text = "Close", highlightbackground = '#424949', font = buttonFont, command = lambda:
    closeButton.place(relx = 0.375, rely = 0.7, relheight = 0.15, relwidth = 0.25)
                                                                                    self.returnPage(accountSuccessfulPage))
```

Which produces:



As desired.

The accountFailed function is as follows:

```
def accountFailed(self, invalidFirstName, firstNameEntry, invalidLastName, lastNameEntry, takenUsername, invalidUsername, usernameEntry, weakPassword,
    print('Failed')                                                                                                    passwordEntry, usernameErrorList, passwordErrorList,
                                                                                                                       titleFont, buttonFont, defaultFont):
    if invalidFirstName == True:
        firstNameEntry.config(highlightthickness = 2, highlightbackground = 'red')
    if invalidLastName == True:
        lastNameEntry.config(highlightthickness = 2, highlightbackground = 'red')
    if takenUsername == True or invalidUsername == True:
        usernameEntry.config(highlightthickness = 2, highlightbackground = 'red')
    if weakPassword == True:
        passwordEntry.config(highlightthickness = 2, highlightbackground = 'red')

    self.failedPage(usernameErrorList, passwordErrorList, self.titleFont, self.buttonFont, self.defaultFont)
```

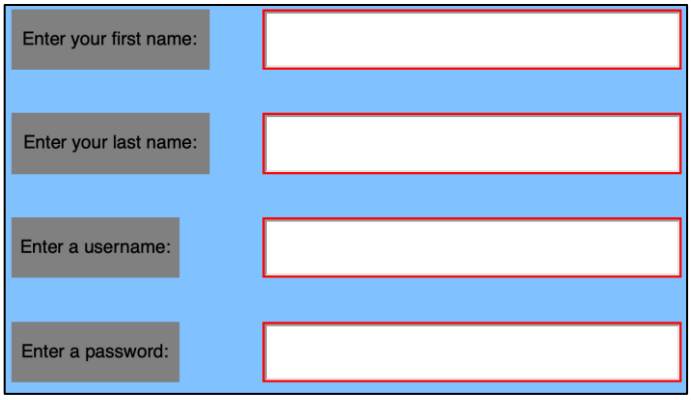
In this function, for all entries that are invalid, the program pinpoints them to the user by creating a red border around those entries.
At first, this error was shown:

```
_tkinter.TclError: invalid command name ".!frame.!entry"
```

This was because in the previous function, I destroyed the create account page before this function was carried out, therefore meaning it couldn't be carried out.
I therefore modified the previous function:

```
def createUserAccount(self, previousPage, userType, firstName, lastName, username, password, adminUser, firstNameEntry, lastNameEntry, usernameEntry, passwordEntr
    ## Validation checks
    takenUsername = self.checkTakenUsername(username)
    invalidUsername, usernameErrorList = self.checkValidUsername(username)
    weakPassword, passwordErrorList = self.checkValidPassword(password)
    invalidFirstName = self.checkValidName(firstName)
    invalidLastName = self.checkValidName(lastName)

    ## If no inconsistencies...
    if takenUsername == False and invalidUsername == False and weakPassword == False and invalidFirstName == False and invalidLastName == False:
        ## Close previous page
        previousPage.destroy()
        self.accountSuccessful(self.titleFont, self.buttonFont, self.defaultFont)
    else:
        self.accountFailed(invalidFirstName, firstNameEntry, invalidLastName, lastNameEntry, takenUsername, invalidUsername, usernameEntry, weakPassword, passwor
```

Firstly, I moved the previousPage.destroy() code to be included within the if statement for when the account creation has been successful. This keeps the page open for if it has failed. I also passed in all the relevant parameters in the function and called the accountFailed function with the correct parameters.
This then allowed the function to work as desired. When pressing the create account button when all entries are blank:

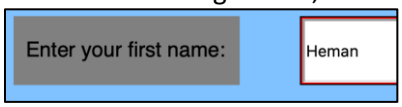


However, this meant that once an entry box had turned red, it wasn't reverted if the user then correctly typed in an entry and pressed the create account button again.

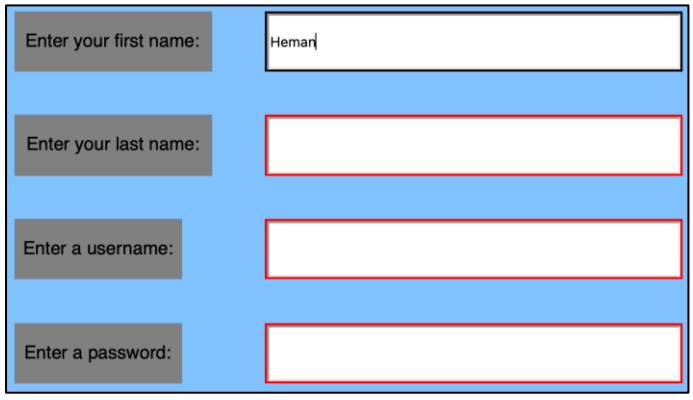
To combat this, I extended the code a bit:

```
def accountFailed(self, invalidFirstName, firstNameEntry, invalidLastName, lastNameEntry, takenUsername, inva
    print('Failed')

    if invalidFirstName == True:
        firstNameEntry.config(highlightthickness = 2, highlightbackground = 'red')
    else:
        firstNameEntry.config(highlightthickness = 2, highlightbackground = 'black')

    if invalidLastName == True:
        lastNameEntry.config(highlightthickness = 2, highlightbackground = 'red')
    else:
        lastNameEntry.config(highlightthickness = 2, highlightbackground = 'black')

    if takenUsername == True or invalidUsername == True:
        usernameEntry.config(highlightthickness = 2, highlightbackground = 'red')
    else:
        usernameEntry.config(highlightthickness = 2, highlightbackground = 'black')

    if weakPassword == True:
        passwordEntry.config(highlightthickness = 2, highlightbackground = 'red')
    else:
        passwordEntry.config(highlightthickness = 2, highlightbackground = 'black')

    self.failedPage(usernameErrorList, passwordErrorList, self.titleFont, self.buttonFont, self.defaultFont)
```

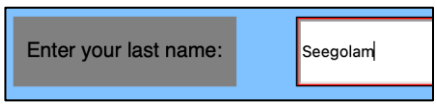
Here, if the invalid variables are not shown to be True, then the border is configured to have a black border, as it does before configuration.
So after first being invalid, after this input:

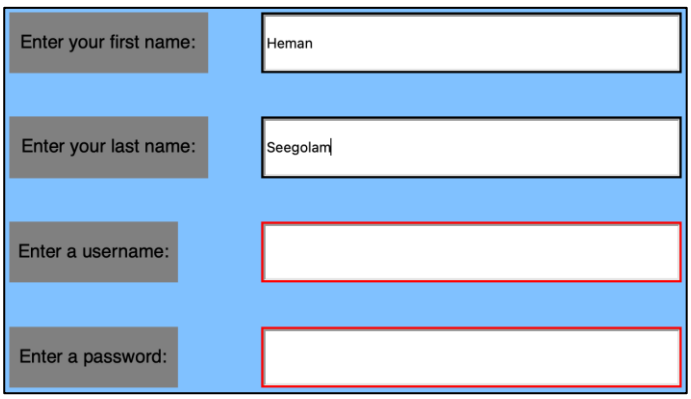


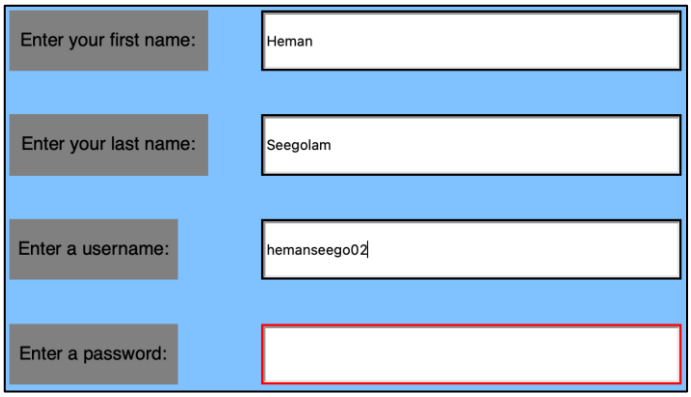
It returns this output:



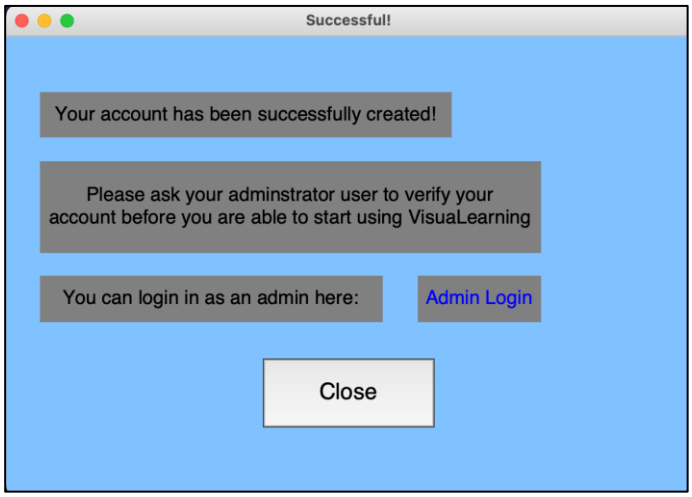
For the other fields:





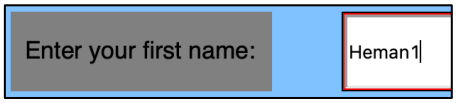


(Pass-word123)

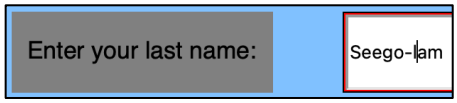


Invalid fields:

(taken username)



(number in name)



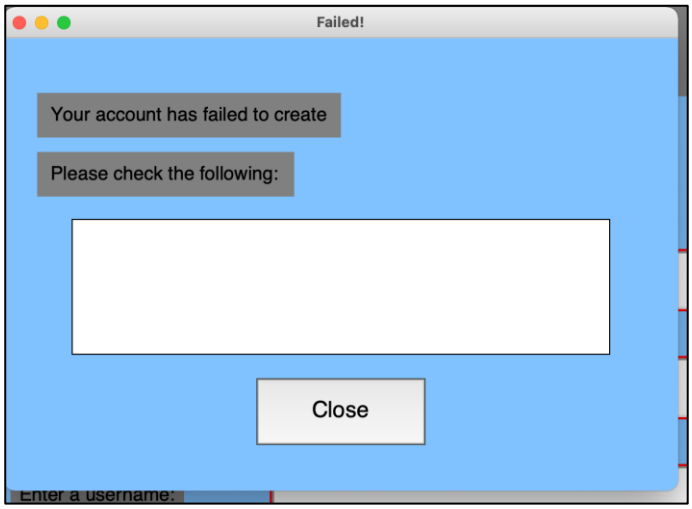
(symbol in name)

Account failed page:

```
def failedPage(self, invalidFirstName, invalidLastName, takenUsername, usernameErrorList, passwordErrorList, titleFont, buttonFont, defaultFont):
    accountFailedPage = tk.Tk()
    accountFailedPage.title('Failed!')

    mainScreenWidth = newAccount.getScreenWidth()
    mainScreenHeight = newAccount.getScreenHeight()

    canvas = tk.Canvas(accountFailedPage, height = mainScreenHeight, width = mainScreenWidth)
    canvas.pack()

    mainframe = tk.Frame(accountFailedPage, bg = '#80c1ff')
    mainframe.place(relheight = 1, relwidth = 1)

    failedLabel = tk.Label(mainframe, text = 'Your account has failed to create', bg = 'gray', font = defaultFont)
    failedLabel.place(relx = 0.05, rely = 0.12, relheight = 0.1, relwidth = 0.45)

    checkLabel = tk.Label(mainframe, text = 'Please check the following:', bg = 'gray', font = defaultFont)
    checkLabel.place(relx = 0.05, rely = 0.25, relheight = 0.1, relwidth = 0.38)
```

```
    listbox = tk.Listbox(mainframe, font = defaultFont)
    listbox.place(relx = 0.1, rely = 0.4, relheight = 0.3, relwidth = 0.8)
    errorList = ['You have entered an invalid first name',
                 'You have entered an invalid last name',
                 'This username has already been taken. Please choose another one',
                 'Your username must not contain any of the following symbols: = ;',
                 'Your username must not contain a space',
                 'Your username cannot be blank',
                 'Your username cannot be username',
                 'Your password is too short',
                 'Your password needs to contain a lower case letter',
                 'Your password needs to contain an upper case letter',
                 'Your password needs to contain a numerical character',
                 'Your password needs to contain a symbol',
                 'Your password must not contain any of the following symbols: = ;',
                 'Your password is not secure enough',
                 'Your password cannot be blank']
```

Here, I've created a list of all possible errors that a user may experience when creatin an account. According to what variables turn out to be True and the contents of the username and password error list, some of these possible errors will appear in a listbox.



Without first populating the list box, it appears as so.

```
position = 1
if invalidFirstName == True:
    listbox.insert(position, errorList[0])
    position += 1
if invalidLastName == True:
    listbox.insert(position, errorList[1])
    position += 1
if takenUsername == True:
    listbox.insert(position, errorList[2])
    position += 1
if len(usernameErrorList) != 0:
    for each in usernameErrorList:
        if each == '=':
            listbox.insert(position, errorList[3])
            position += 1
        if each == ';':
            listbox.insert(position, errorList[3])
            position += 1
        if each == 'Space':
            listbox.insert(position, errorList[4])
            position += 1
        if each == 'empty':
            listbox.insert(position, errorList[5])
            position += 1
        if each == 'username':
            listbox.insert(position, errorList[6])
            position += 1
```

To insert values into the list box, you first have to declare its position. Since my list box would have varying numbers of values within the box at one time, I decided to create a general position variable, which increments after every insertion into the listbox.
And so this part of the function identifies first any errors with the first and last names, and displays them into the list box if necessary. It then checks for a taken username and all other validation checks for it, assigning the relevant error messages in the list box.

```
if len(passwordErrorList) != 0:
    for each in passwordErrorList:
        if each == 'short':
            listbox.insert(position, errorList[7])
            position += 1
        if each == 'lowerCase':
            listbox.insert(position, errorList[8])
            position += 1
        if each == 'upperCase':
            listbox.insert(position, errorList[9])
            position += 1
        if each == 'number':
            listbox.insert(position, errorList[10])
            position += 1
        if each == 'symbol':
            listbox.insert(position, errorList[11])
            position += 1
        if each == '=;':
            listbox.insert(position, errorList[12])
            position += 1
        if each == 'password':
            listbox.insert(position, errorList[13])
            position += 1
        if each == 'empty':
            listbox.insert(position, errorList[14])
            position += 1
```

In this part of the function, all password errors are checked and the corresponding error message assigned to the list box.

To simplify the code a little, I slightly modified the username validation function and the corresponding part in this function:

```
if '=' in username or ';' in username:
    usernameErrorList.append('=;')
```

```
if each == '=;':
    listbox.insert(position, errorList[3])
    position += 1
```

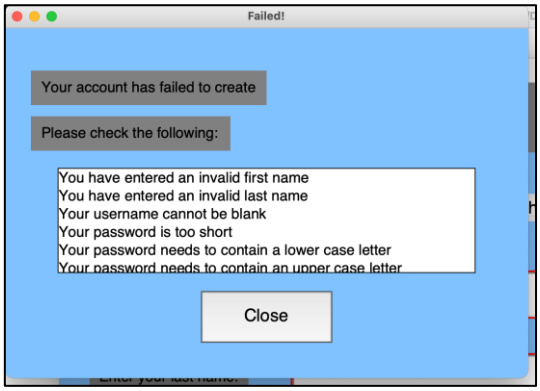
Since they use the same error message.

Although the program will take up some extra time to perform these checks, it will be extremely convenient for a user when creating an account since any errors are pinpointed and allows them to change it accordingly.

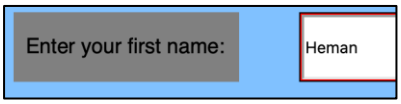
Also, since the account failed page doesn't close the previous page, it allows the user to see both the form and the errors at the same time, which saves them from having to remember them all and / or restarting.

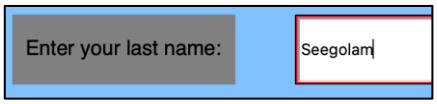
Tests:

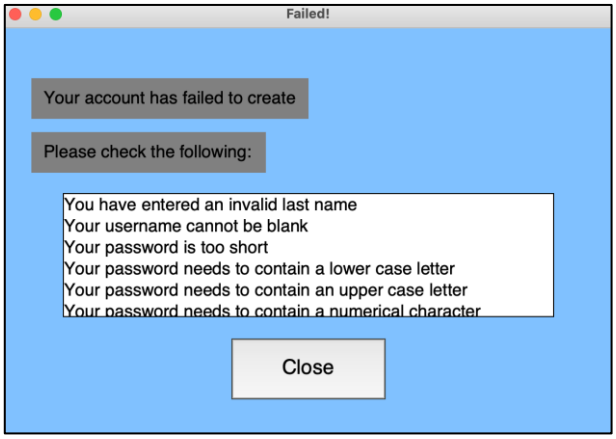
When all inputs are blank:

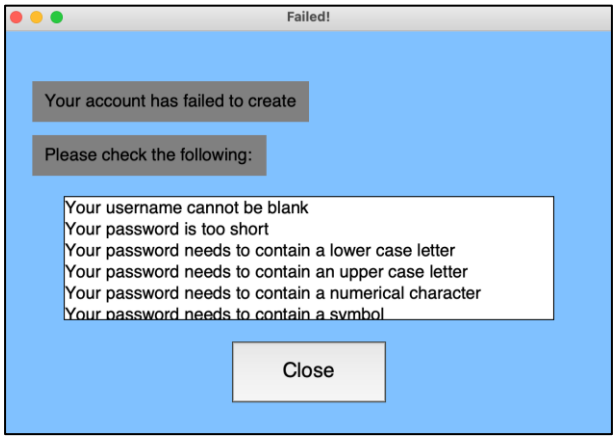


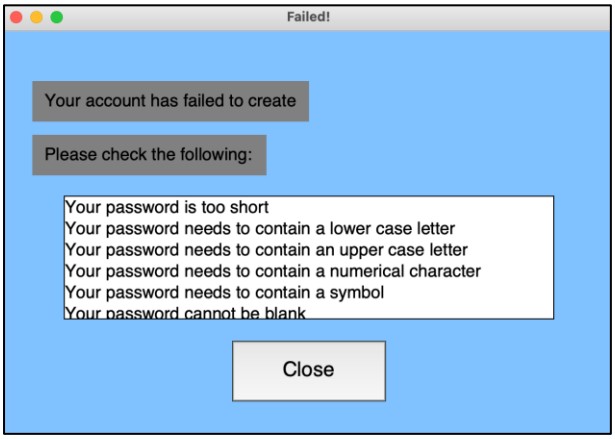
When filling them in correctly:

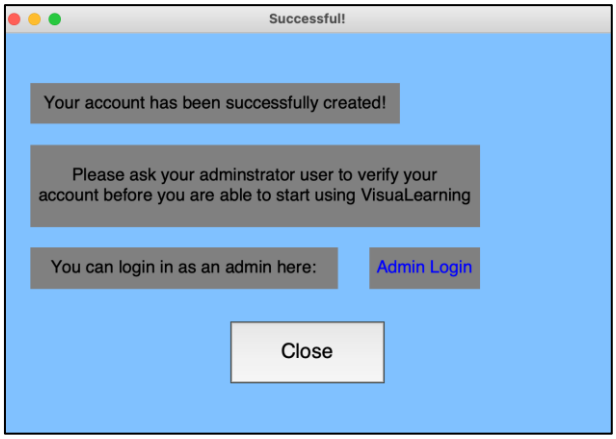




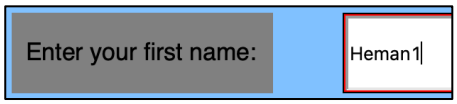


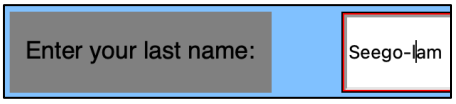


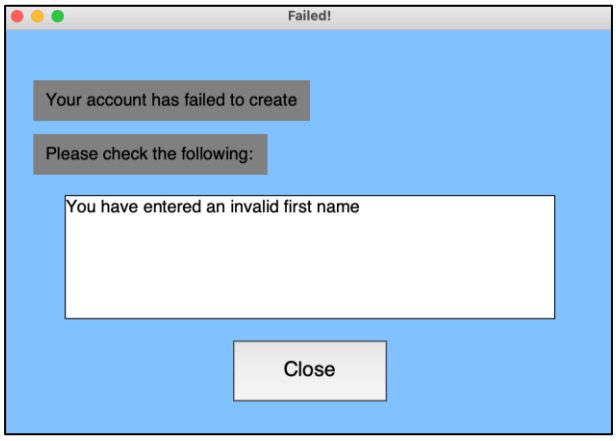


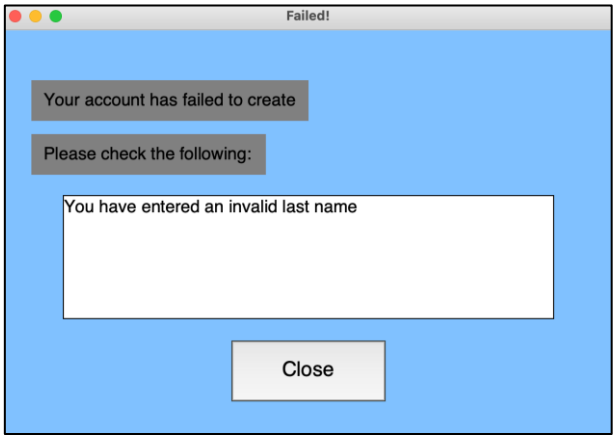


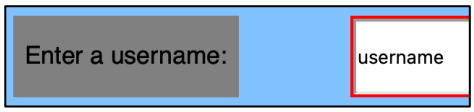
Invalid fields (where all else are correct):

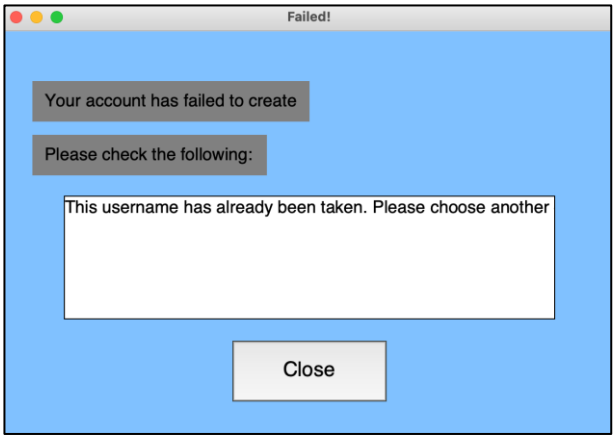


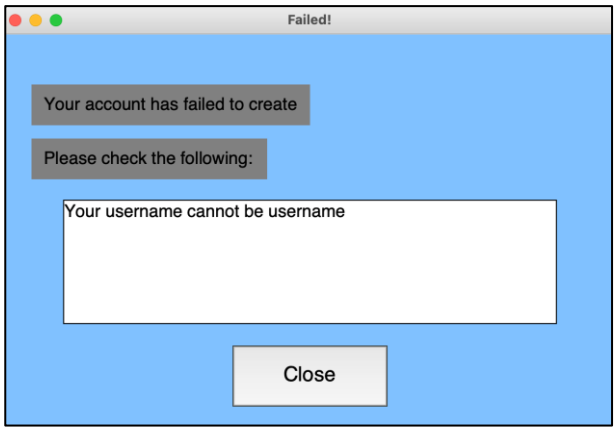


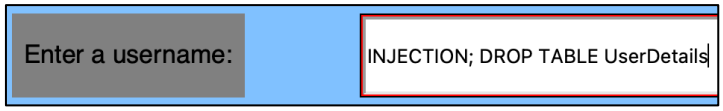


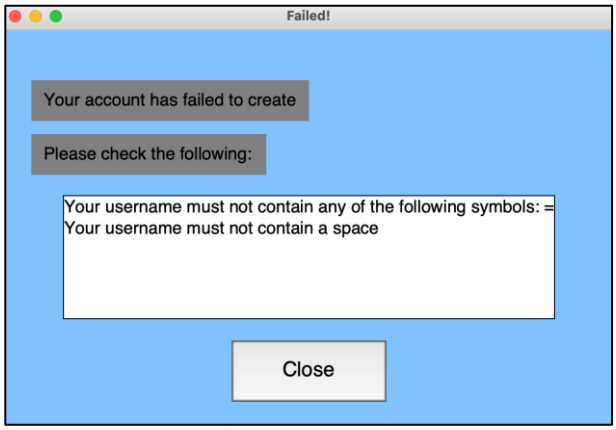


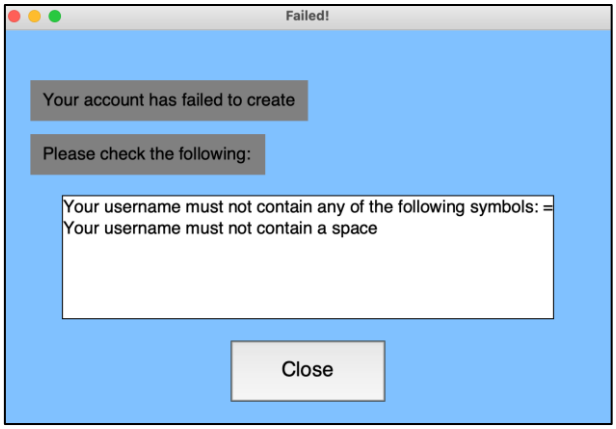




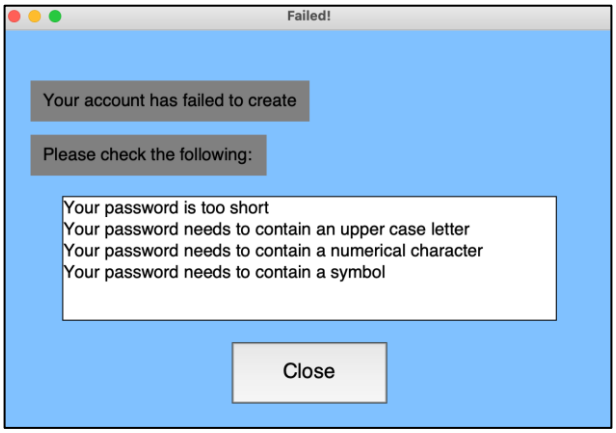




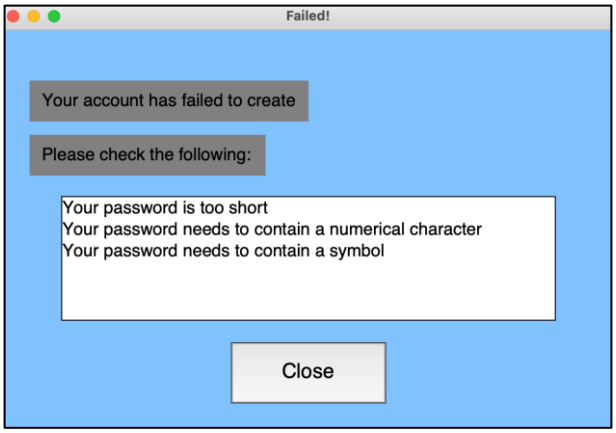




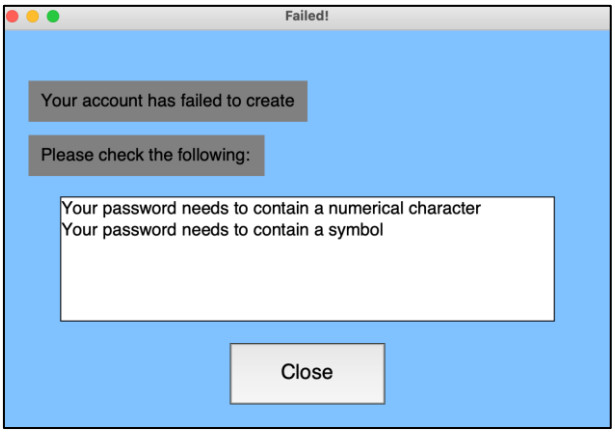
Password as ‘pass’:



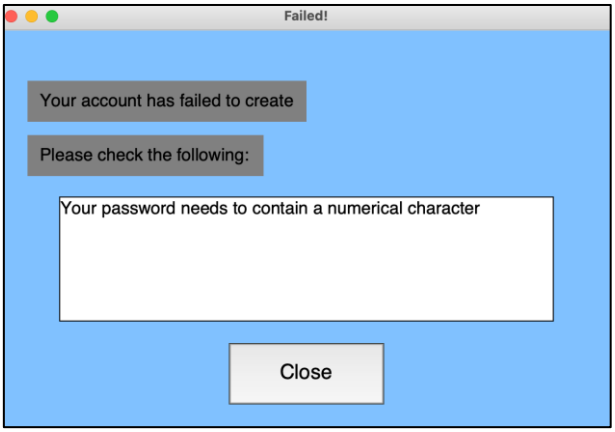
Password as ‘Pass’:



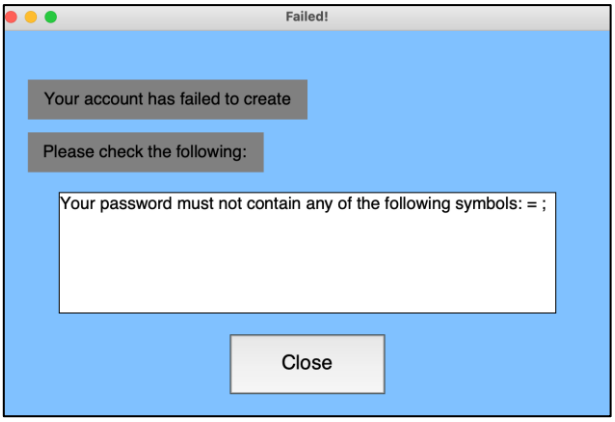
Password as ‘Password’:



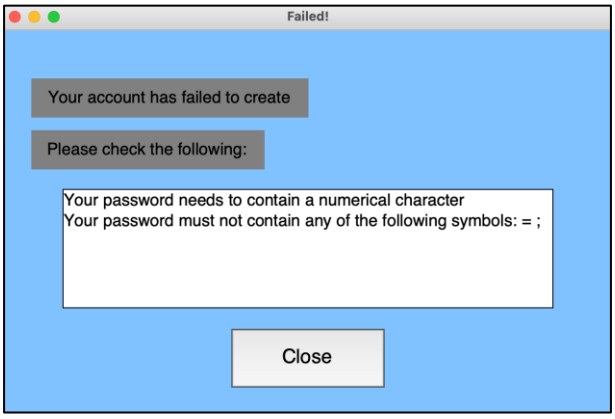
Password as ‘Pass-word’:



Password as ‘Injection OR 1=1’:



Password as ‘Injection; DROP TABLE Quiz’:



To complete the account creation process, the new account would have to first be sent off for verification before then being added as a formal user. Without verification, a user must not be able to access the program. Perhaps I could add a feature later on that allows a user awaiting verification to access some parts, but not all, of the program. The main reasons for such verification is that firstly, teachers can track each student effectively, and can prevent any students with multiple accounts. Secondly, if the program was to be sold at a later time, it would prevent random users accessing it without appropriate payment.

Since admin users have to verify new accounts before they may use them, this means that these may have to be externally stored so non-volatile. If the program is closed after a new account is created, an admin user must still have access to verify them once the program is re-run. Therefore, I shall place the new accounts with all details, in the database. As shown previously, in the UserReview table, there is a record which establishes new accounts. Hence, I can add these accounts in the same way into my database, but with a review ID of 2.

To do this, I modified the code in this way:
The createUserAccount function:

```
## If no inconsistencies...
if takenUsername == False and invalidUsername == False and weakPassword == False and invalidFirstName == False and invalidLastName == False:
    ## Close previous page
    previousPage.destroy()
    record = [None, userType, username, password, firstName, lastName, 2]
    self.accountSuccessful(record, self.titleFont, self.buttonFont, self.defaultFont)
```

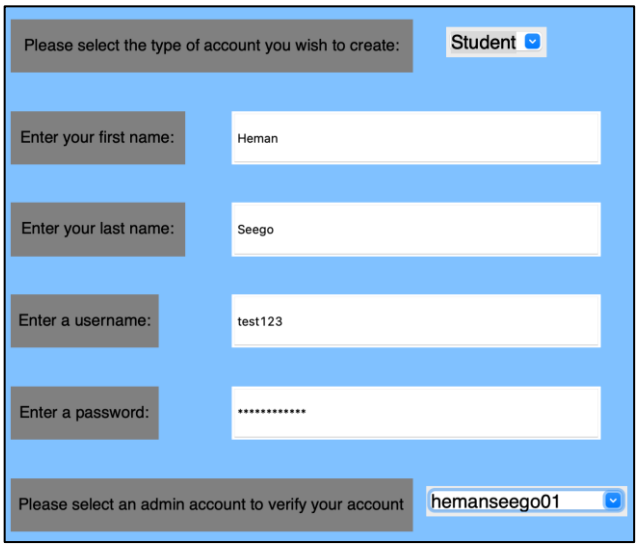
Here, I created a record in the same order as in the database of all the relevant details, retrieved from the entries from the create account page after all validation checks. For the userID, I passed in None to the record. This is because, I had previously set the userID field to auto-increment. For simplicity and to avoid any potential conflicts, I let SQL handle the auto incrementation in this way. I also set the reviewID to 2, since later on, this will signal to the relevant admin user that this account needs to be verified.

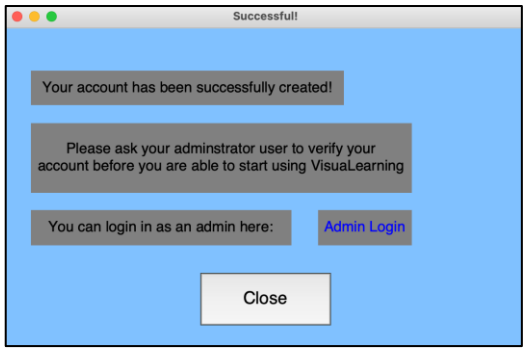
And the accountSuccessful function:

```
def accountSuccessful(self, record, titleFont, buttonFont, defaultFont):
    addingToDatabase.addToDatabase(conn, c, record, 'UserDetails')

    accountSuccessfulPage = tk.Tk()
    accountSuccessfulPage.title('Successful!')
```

To first take in record as a parameter, and then add it to the database using my database class, the table set to ‘UserDetails’.

Using these values:





This was the output:

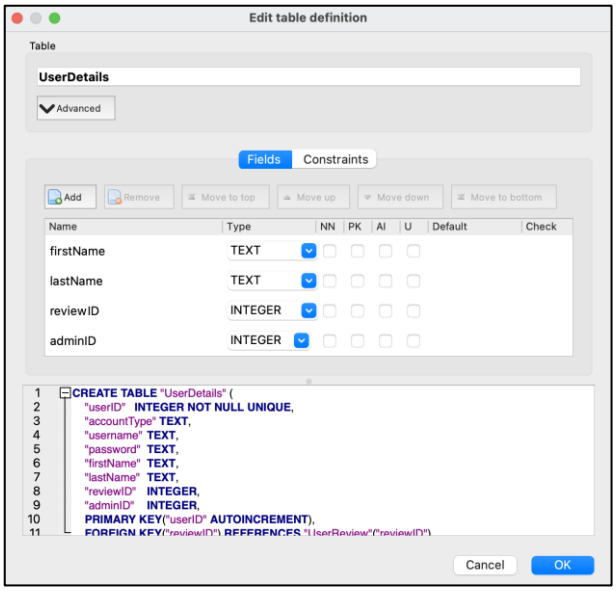
Table: UserDetails

	userID	accountType	username	password	firstName	lastName	reviewID
	Filter	Filter	Filter	Filter	Filter	Filter	Filter
1	1	Admin	hemanseego01	Pass-Word123	Heman	Seegolam	1
2	2	Student	sampinch02	passWORD456	Sam	Pinchback	1
3	3	Parent	mattbumpus03	PASSword678!	Matthew	Bumpus	2
4	4	Admin	emmahirst04	pass-WORD901	Emma	Hirst	1
5	5	Admin	harrymcdonagh05	pAsSwOrD_234	Harry	McDonagh	1
6	6	Student	test123	Pass-word123	Heman	Seego	2

As desired.

However, I noticed that this failed to store the relevant admin user.

To fix this problem, I added a field to the UserDetails table which would include the username of userID of the admin user that would need to take action on an account. This will also be helpful later on when a user has other review issues such as forgetting their password or blocking their account. By allowing them to choose which admin user to act on their account, this prevents colleagues for example to attain information about students that are not theirs.
I added this field using a database software:



And modified my database accordingly:

Table: UserDetails

	userID	accountType	username	password	firstName	lastName	reviewID	adminID
	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter
1	1	Admin	hemanseego01	Pass-Word123	Heman	Seegolam	1	NULL
2	2	Student	sampinch02	passWORD456	Sam	Pinchback	1	NULL
3	3	Parent	mattbumpus03	PASSword678!	Matthew	Bumpus	2	1
4	4	Admin	emmahirst04	pass-WORD901	Emma	Hirst	1	NULL
5	5	Admin	harrymcdonagh05	pAsSwOrD_234	Harry	McDonagh	1	NULL
6	6	Student	test123	Pass-word123	Heman	Seego	2	1

For all records which have a reviewID of 1, i.e. where no action is required, their adminID contains no value. However, for all those with a reviewID which is not 1, they will have an appropriate admin ID in place.
I decided to use the admin's userID here instead of their username since searching would be faster.

I modified my addToDatabase function to make this change consistent:

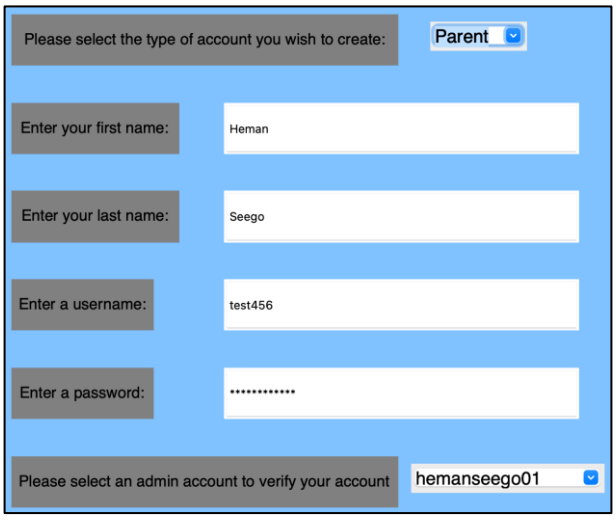
```
def addToDatabase(self, conn, c, record, table):
    ## If to be added to UserDetails table
    if len(record) == 8:
        # Establish SQL statement
        sqlAdd = ("INSERT INTO " + table + " VALUES (?,?,?,?,?,?,?,?)")
```

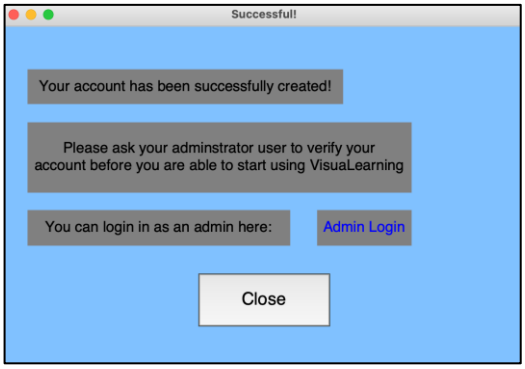
Where before it was 7, the length of the record has now been changed to 8.

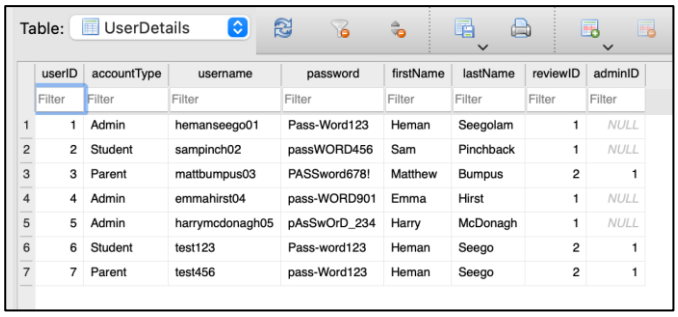
I again modified my createUserAccount function:

```
## If no inconsistencies...
if takenUsername == False and invalidUsername == False and weakPassword == False and invalidFirstName == False and invalidLastName == False:
    ## Close previous page
    previousPage.destroy()
    adminID = viewingFromDatabase.viewFromDatabase(c, False, False, 'userID', 'username', "'"+adminUser+"'", 'UserDetails')
    adminID = adminID[0][0]
    record = [None, userType, username, password, firstName, lastName, 2, adminID]
    self.accountSuccessful(record, self.titleFont, self.buttonFont, self.defaultFont)
```

Here, I had to establish the admin user ID from the chosen username. I did this using the view function from the database class, with all relevant parameters. Initially an error was experienced since I passed in adminUser for the username of the admin user which led the program to search for a column instead of a value. To combat this, I put quotation marks around it. I then faced an issue where admin ID wasn't being correctly identified. This was because when retrieving values from the database it was encapsulated within a tuple within a list. By reassigning adminID with the first value within the tuple within the first value within the list, it then proceeded to work as desired.







	userID	accountType	username	password	firstName	lastName	reviewID	adminID
1	1	Admin	hemanseego01	Pass-Word123	Heman	Seegolam	1	NULL
2	2	Student	sampinch02	passWORD456	Sam	Pinchback	1	NULL
3	3	Parent	mattbumpus03	PASSword678!	Matthew	Bumpus	2	1
4	4	Admin	emmahirst04	pass-WORD901	Emma	Hirst	1	NULL
5	5	Admin	harrymcdonagh05	pAsSwOrD_234	Harry	McDonagh	1	NULL
6	6	Student	test123	Pass-word123	Heman	Seego	2	1
7	7	Parent	test456	pass-Word123	Heman	Seego	2	1

The final stage left of account creation concerns the security of the users. When designing my program, I compared the benefits of both encryption and hashing and concluded that hashing was the optimal solution when regarding the raw data that users may see if they gain unauthorised access to the database. I therefore created a function to first hash the password before uploading it to the database.

```
def hashPassword(self, password):
    newPassword = hash(password)
    return newPassword
```

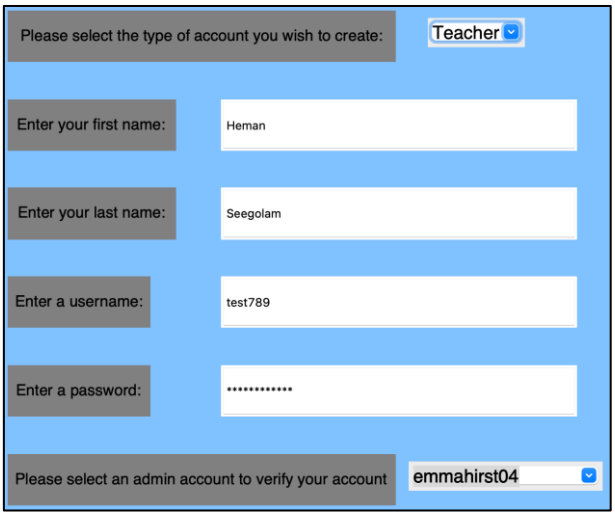
We can see the effect of the hash function here:

```
>>> createAccount.hashPassword('password')
-8333460651459619022
>>> createAccount.hashPassword('password1')
4955316502339549093
>>> createAccount.hashPassword('pass-word')
-5611060222602233588
>>> createAccount.hashPassword('password')
-8333460651459619022
>>>
```

We see the adverse effect that a small change in the string causes and also the abstractness shown in the final result. We also see that one input will always result in the same output.

I modified the create user account function accordingly:

```
## If no inconsistencies...
if takenUsername == False and invalidUsername == False and weakPassword == False and invalidFirstName == False and invalidLastName == False:
    ## Close previous page
    previousPage.destroy()
    adminID = viewingFromDatabase.viewFromDatabase(c, False, False, 'userID', 'username', "'"+adminUser+"'", 'UserDetails')
    adminID = adminID[0][0]
    newPassword = self.hashPassword(password)
    record = [None, userType, username, newPassword, firstName, lastName, 2, adminID]
    self.accountSuccessful(record, self.titleFont, self.buttonFont, self.defaultFont)
```



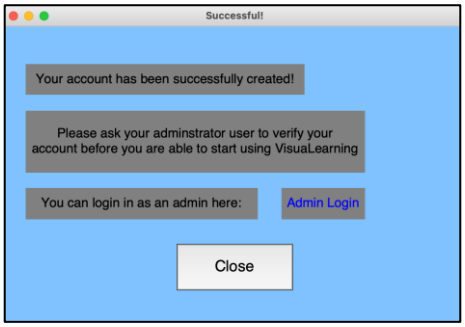


Table: UserDetails

	userID	accountType	username	password	firstName	lastName	reviewID	adminID
	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter
1	1	Admin	hemanseego01	Pass-Word123	Heman	Seegolam	1	NULL
2	2	Student	sampinch02	passWORD456	Sam	Pinchback	1	NULL
3	3	Parent	mattbumpus03	PASSword678!	Matthew	Bumpus	2	1
4	4	Admin	emmahirst04	pass-WORD901	Emma	Hirst	1	NULL
5	5	Admin	harrymcdonagh05	pAsSwOrD_234	Harry	McDonagh	1	NULL
6	6	Student	test123	Pass-word123	Heman	Seego	2	1
7	7	Parent	test456	pass-Word123	Heman	Seego	2	1
8	8	Teacher	test789	-5371957360183907341	Heman	Seegolam	2	4

```
>>> createAccount.hashPassword('Pass-Word123')
-4614026787805931621
>>>
```

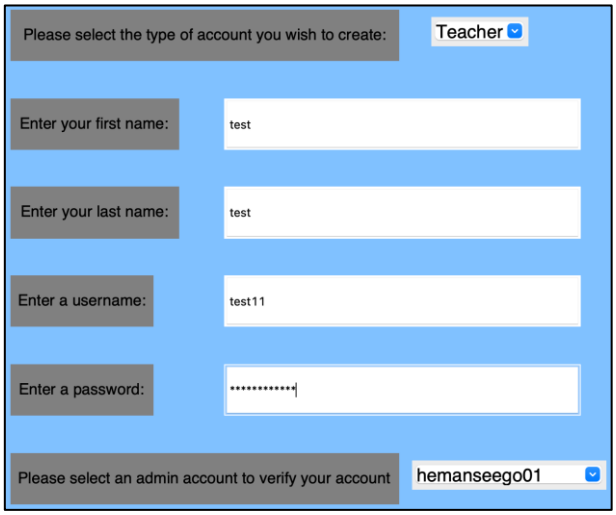
However, it appeared that on every time a new session was run, the python hash function would obtain a different value, and would only remain the same within that session. This would be hugely problematic since users would never be able to access their account outside of one session. As a result, I used a different hash function.
First I had to import the hashlib library.

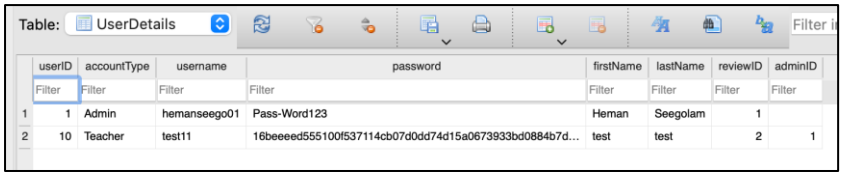
```
import hashlib
```

I therefore modified the hashPassword function:

```
def hashPassword(self, password):
    newPassword = hashlib.sha256(password.encode()).hexdigest()
    return newPassword
```

This function required the string to first be encoded. After hashing (using SHA-256), it then needed to be 'digested' to appear in a suitable format. Hashed values now remained the same between different sessions.






```
>>> createAccount.hashPassword('Pass-Word123')
'16beeeed555100f537114cb07d0dd74d15a0673933bd0884b7dac755b4b71fdd'
>>>
```

As desired.

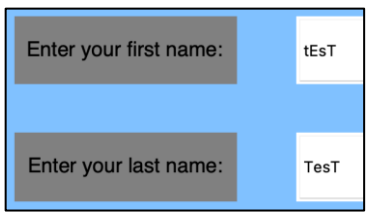
Even if anyone was to gain unauthorised access to the database, if they were to input the hash value, they still wouldn't gain access to one's account since that value would go through the hash function and would not return either the password or the hash value in the database.

Although the account creation process is hereby complete, there is an aspect that can be improved for completeness and professionalism. This is where users can enter their first and last name. To keep it all in a neat format (concerning capital letters) within my database, and also to make it easier when retrieving to welcome a user later on:

```
record = [None, userType, username, newPassword, firstName.capitalize(), lastName.capitalize(), 2, adminID]
```

I pass in firstName.capitalize() and lastName.capitalize(), which capitalises the first letter of the respective names and makes the rest of the string lower case, as desired.

Test	Test



Upon testing, I discovered that there were some inconsistencies in my database. This was where I pass in the user type to the record which from the account creation page says Teacher, but in my database translates as Admin. Therefore:

```
if userType == 'Teacher':
    userType = 'Admin'
adminID = viewingFromDatabase.viewFromDatabase(c, False, False, 'userID', 'username', "'"+adminUser+"'", 'UserDetails')
adminID = adminID[0][0]
record = [None, userType, username, newPassword, firstName.capitalize(), lastName.capitalize(), 2, adminID]
```

Which then produced 'Admin' in the database in the place of 'Teacher'.

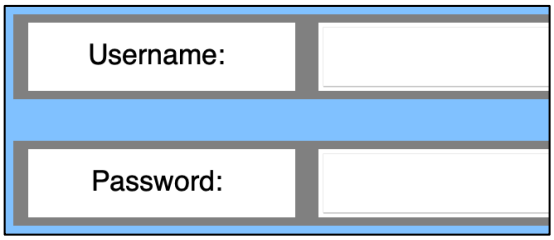
Login Function:

```
def loginClicked(self, usernameEntry, passwordEntry):
    correctUsername = False
    correctPassword = False

    ## Retrieve all usernames from database
    allUsers = viewingFromDatabase.viewFromDatabase(c, False, True, 'username', None, None, 'UserDetails')
    allUsersList = []
    for each in range(len(allUsers)):
        allUsersList.append(allUsers[each][0])
    ## Check whether username exists & retrieves its corresponding password
    if usernameEntry in allUsersList:
        correctUsername = True
        password = viewingFromDatabase.viewFromDatabase(c, False, False, 'password', 'username', "'"+usernameEntry+"'", 'UserDetails')
        password = password[0][0]
        ## Compare hashed password input with correct password hash
        if self.hashPassword(passwordEntry) == password:
            correctPassword = True

    if correctUsername == True and correctPassword == True:
        print('Access Granted')
    else:
        print('Login Failed')
```

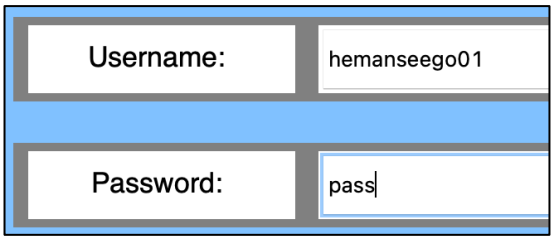
In this function, I create two boolean variables which determine whether the inputted username and password is correct. They are both initially set as False. The function then goes on to use the viewFromDatabase function from the database class to retrieve all the usernames and put them in a list. It then proceeds to check whether the inputted username from the login page is in the database, or the list that has just been created. If it is found to be in the list, the correctUsername variable is set to be True, and the corresponding password is fetched from the database. Since this password is in its hashed form, I then proceed to compare it with a hashed version of the inputted password. If they match, correctPassword is set to be True. For testing purposes I then go on to display 'Access Granted' if both boolean variables are found to be True, and 'Login Failed' for any other case.

To test I displayed the two boolean variables as follows:

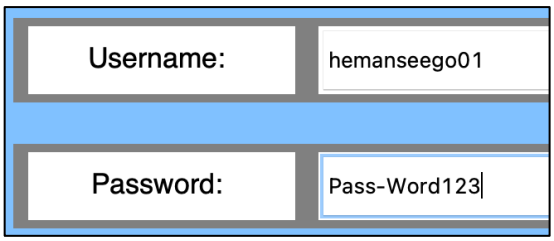
```
print('username ='+str(correctUsername),'password ='+str(correctPassword))
```




```
username =False password =False
Login Failed
```




```
username =True password =False
Login Failed
```

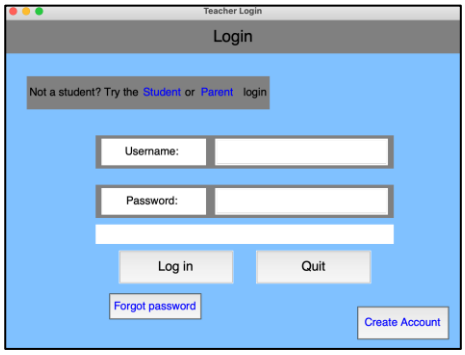



```
username =True password =True
Access Granted
```

There is a limitation with this method of logging in and that is as the number of users grows, the program will take longer and longer to search for a username and its corresponding password. Since it would follow the format of a linear search, first to retrieve all usernames from the database and then compare it with the inputted username, it would have a time of O(n) for each of these respective tasks. This could make it inefficient as the program becomes more popular. However, by splitting up the logins into an admin, student and parent login, I found I could significantly reduce this time, simply by searching for all usernames with the correct user type. This could potentially reduce the overall searching time by a third, thus much more convenient for the user.

First, I modified the login pages to include a section where any errors can be pinpointed to the user, e.g. incorrect password, non-existent username, wrong user login.

```
errorLabel = tk.Label(mainframe)
errorLabel.place(relx = 0.2, rely = 0.62, relheight = 0.06, relwidth = 0.65)
```

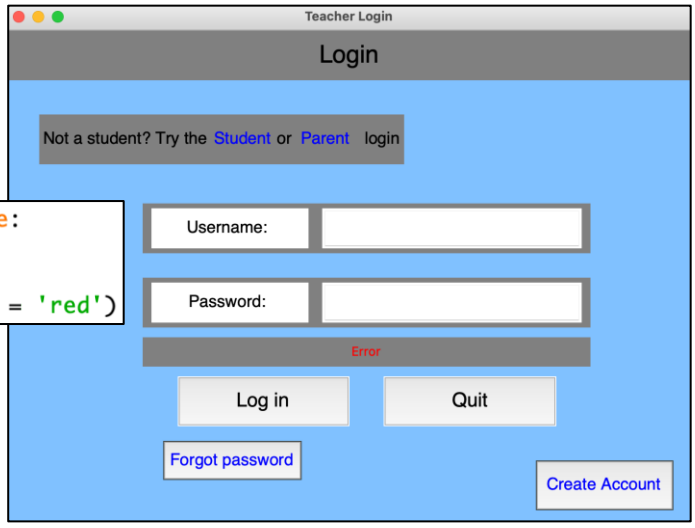


And I configured its background to match that of the page, so isn't seen by users when the first use it.

I then passed on this error label as a parameter to the loginClicked function, making the change consistent in all parts of the code.

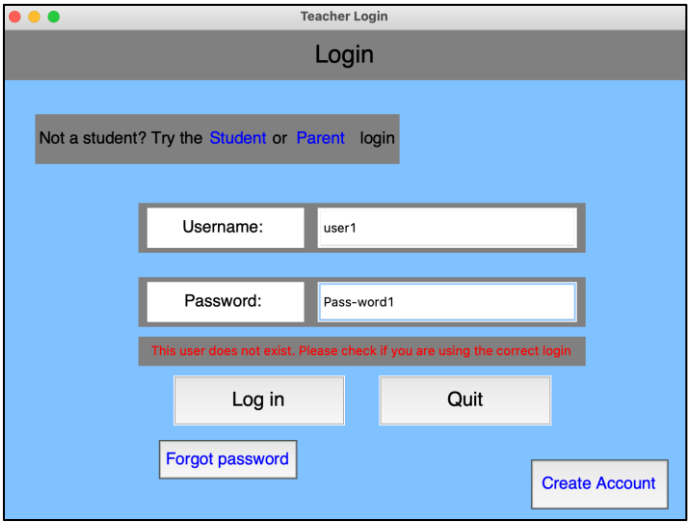
I modified the function to test it:

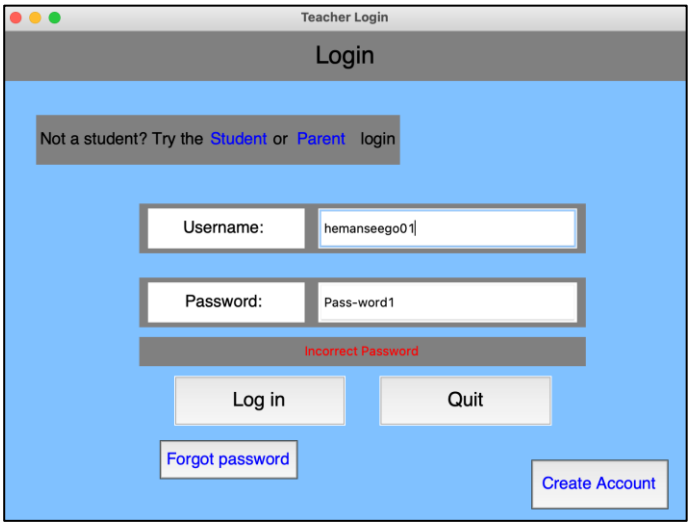
```
if correctUsername == True and correctPassword == True:
    print('Access Granted')
else:
    errorLabel.config(text = 'Error', bg = 'gray', fg = 'red')
```



I then proceeded to create the different label text phrases for the different scenarios:

```
if correctUsername == True and correctPassword == True:
    mainMenu.mainMenuPage()
else:
    if correctUsername == True and correctPassword == False:
        error = 'Incorrect Password'
    elif correctUsername == False and correctPassword == False:
        error = 'This user does not exist. Please check if you are using the correct login'
    errorLabel.config(text = error, bg = 'gray', fg = 'red')
```



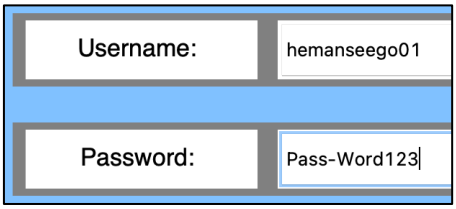


When successfully logged in, the user will be brought to the main menu page. For now, this function looks like this:

```
def mainMenuPage(self):
    print('Main Menu')
```

As part of the Interface class:

```
mainMenu = Interface(1000, 800)
```




```
Main Menu
```

To retrieve the relevant users for each login page, I passed the loginType parameter which was used to determine the different login labels into the login pages function and into the login clicked function to allow the program to know which users to retrieve.

```
## Retrieve all usernames of the correct account type from database
allUsers = viewingFromDatabase.viewFromDatabase(c, False, False, 'username', 'accountType', "'"+loginType+"'", 'UserDetails')
allUsersList = []
for each in range(len(allUsers)):
    allUsersList.append(allUsers[each][0])
print(loginType)
print(allUsersList)
```

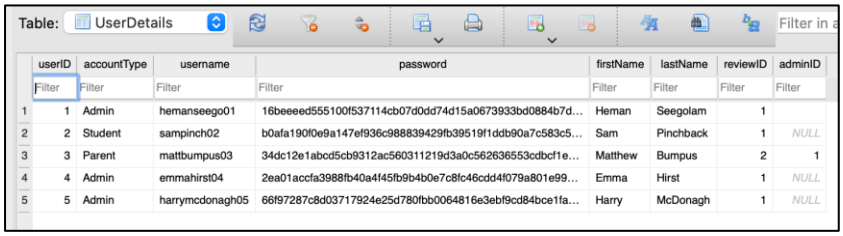


Table: UserDetails

	userID	accountType	username	password	firstName	lastName	reviewID	adminID
1	1	Admin	hemanseego01	16beeeed555100f537114cb07d0dd74d15a0673933bd0884b7d...	Heman	Seegolam	1	
2	2	Student	sampinch02	b0afa190f0e9a147ef936c988839429fb39519f1ddb90a7c583c5...	Sam	Pinchback	1	NULL
3	3	Parent	mattbumpus03	34dc12e1abcd5cb9312ac560311219d3a0c562636553cdbcf1e...	Matthew	Bumpus	2	1
4	4	Admin	emmahirst04	2ea01accfa3988fb40a4f45fb9b4b0e7c8fc46cdd4f079a801e99...	Emma	Hirst	1	NULL
5	5	Admin	harrymcdonagh05	66f97287c8d03717924e25d780fbb0064816e3ebf9cd84bce1fa...	Harry	McDonagh	1	NULL

With this database, it retrieved these values for the given login type:

```
Admin
['hemanseego01', 'emmahirst04', 'harrymcdonagh05']
Student
['sampinch02']
Parent
['mattbumpus03']
>>>
```

As per the database.

In the function, I also hashed and executed the password matching only once the username had been found, which saved the program from doing any unnecessary steps when the login inputs were already wrong. This saves time and makes it more convenient for the user.

I extended it to include the main menu pages depending on the login of the user:

```
if correctUsername == True and correctPassword == True:
    if loginType == 'Admin':
        mainMenu.adminMainMenuPage()
    if loginType == 'Student':
        mainMenu.studentMainMenuPage()
    if loginType == 'Parent':
        mainMenu.parentMainMenuPage()
```

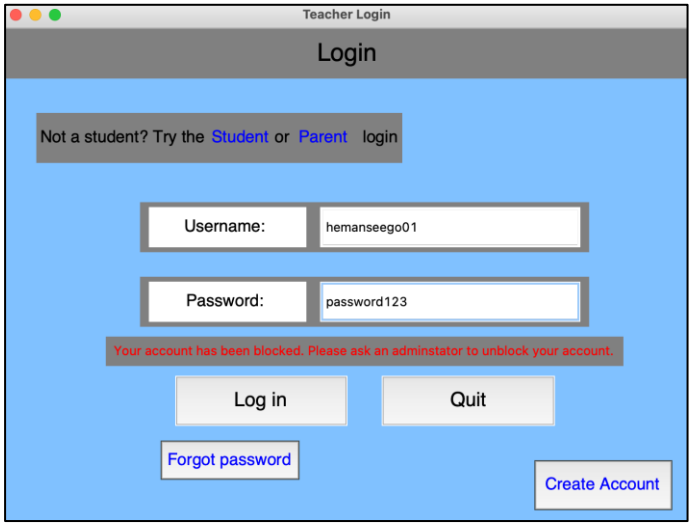
With the respective methods of the Interface class:

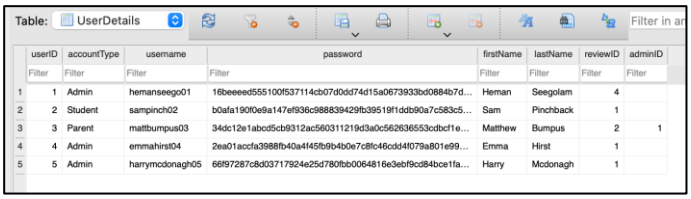
```
def adminMainMenuPage(self):
    print('Admin Main Menu')

def studentMainMenuPage(self):
    print('Student Main Menu')

def parentMainMenuPage(self):
    print('Parent Main Menu')
```

For the prevention of any brute force attacks, as per the design:

```
blockCounter = 0
```

I created a counter variable within the login class which I passed on to the login clicked function, using self.blockCounter.



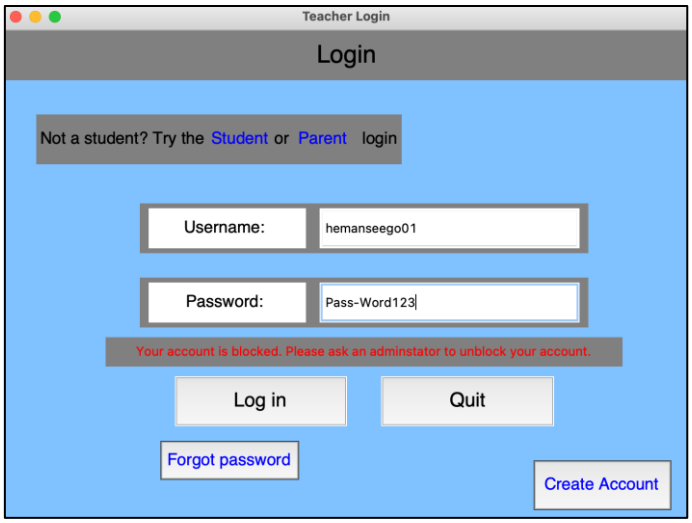


FULL LOGIN FUNCTION:

```
blockCounter = 0
def loginClicked(self, usernameEntry, passwordEntry, loginType, errorLabel):
    correctUsername = False
    correctPassword = False
    accountBlocked = False

    if self.blockCounter < 3:
        ## Retrieve all usernames of the correct account type from database
        allUsers = viewingFromDatabase.viewFromDatabase(c, False, False, 'username', 'accountType', "'"+loginType+"'", 'UserDetails')
        allUsersList = []
        for each in range(len(allUsers)):
            allUsersList.append(allUsers[each][0])
        ## Check whether username exists & retrieves its corresponding password
        if usernameEntry in allUsersList:
            self.blockCounter+=1
            correctUsername = True
            password = viewingFromDatabase.viewFromDatabase(c, False, False, 'password', 'username', "'"+usernameEntry+"'", 'UserDetails')
            password = password[0][0]
            ## Compare hashed password input with correct password hash
            if self.hashPassword(passwordEntry) == password:
                correctPassword = True
            accountReview = viewingFromDatabase.viewFromDatabase(c, False, False, 'reviewID', 'username', "'"+usernameEntry+"'", 'UserDetails')
            accountReview = accountReview[0][0]
            if accountReview == 4:
                accountBlocked = True
```

```
    if correctUsername == True and correctPassword == True and accountBlocked == False:
        if loginType == 'Admin':
            mainMenu.adminMainMenuPage()
        if loginType == 'Student':
            mainMenu.studentMainMenuPage()
        if loginType == 'Parent':
            mainMenu.parentMainMenuPage()
    else:
        if self.blockCounter == 3:
            error = 'Your account has been blocked. Please ask an adminstator to unblock your account.'
            updatingDatabase.updateInDatabase(conn, c, 'reviewID', '4', 'username', "'"+usernameEntry+"'", 'UserDetails')
        elif accountBlocked == True:
            error = 'Your account is blocked. Please ask an adminstator to unblock your account.'
        elif correctUsername == True and correctPassword == False:
            error = 'Incorrect Password'
        elif correctUsername == False and correctPassword == False:
            error = 'This user does not exist. Please check if you are using the correct login.'
        errorLabel.config(text = error, bg = 'gray', fg = 'red')
```

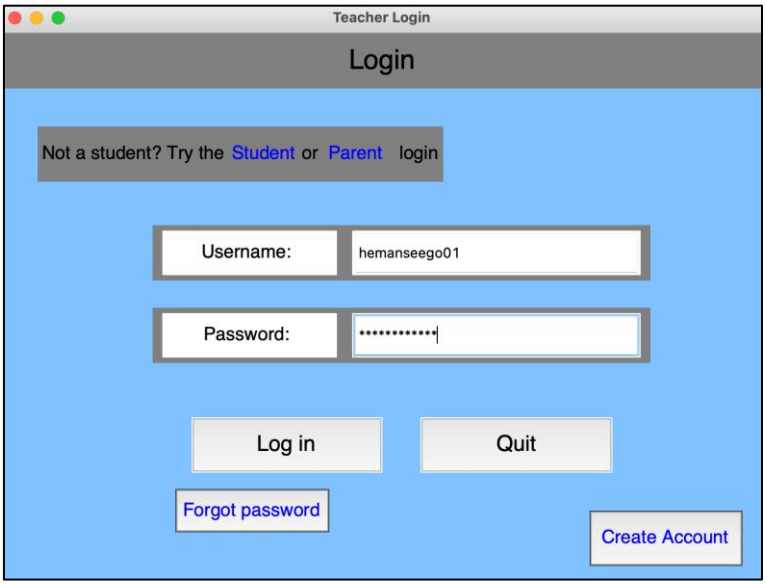
In this full function, I also included a new boolean variable which determined whether the account which is being accessed is blocked or not. If it is, permission is not granted, and they are asked to ask an admin user to unblock their account. When incrementing the block counter, a user isn't penalised for mistyping their username. For every incident where the username is correct and the corresponding password is incorrect, the counter is then incremented. There are some limitations to this method. Namely, if two users had similar usernames and indeed mistyped, this exception would not apply to them, and also, if a user has failed to log in and another user attempted to log in, the counter would not be reset for them. However, it is successful in preventing users from closing the page and reopening to reset the block counter, since it would remain the same until the program is completely restarted. This minimises the opportunity for a bruce force attack. When designing the program, I decided that the block counter would not be applicable to admin users, however upon further reflection, it would be inadequate to compromise the security by making this part prone to



bruce force attacks and therefore implemented the counter for all users. All admin users will be able to see blocked accounts.

Finally, I hid the password from the user in the login pages:

```
passwordEntry = tk.Entry(passwordFrame, show = '*')
```

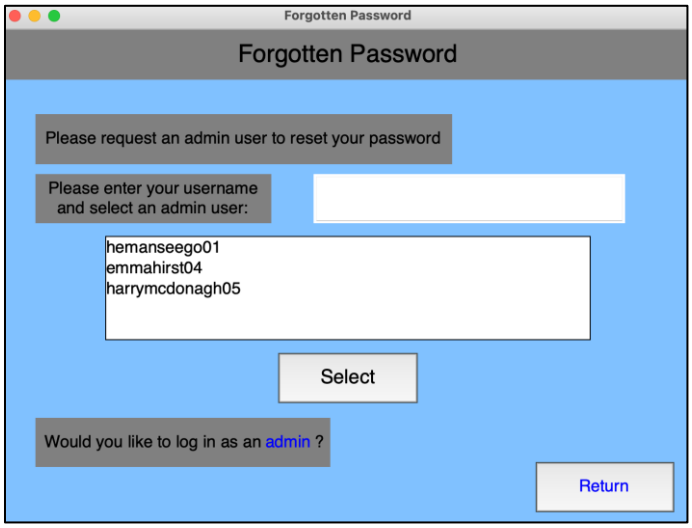


Forgot Password Page:

```
def forgotPasswordClicked(self, titleFont, buttonFont, defaultFont):
    ## Retrieves the assigned width and height for the forgot password page
    mainScreenWidth = forgotPassword.getScreenWidth()
    mainScreenHeight = forgotPassword.getScreenHeight()

    forgotPasswordPage = tk.Tk()
    forgotPasswordPage.title("Forgotten Password")

    canvas = tk.Canvas(forgotPasswordPage, height = mainScreenHeight, width = mainScreenWidth)
    canvas.pack()

    mainframe = tk.Frame(forgotPasswordPage, bg = '#80c1ff')
    mainframe.place(relheight = 1, relwidth = 1)

    welcomeLabel = tk.Label(mainframe, text = "Forgotten Password", bg = 'gray', font = titleFont)
    welcomeLabel.place(relx = 0, rely = 0, relheight = 0.1, relwidth = 1)

    requestAdminLabel = tk.Label(mainframe, text = 'Please request an admin user to reset your password', bg = 'gray', font = defaultFont)
    requestAdminLabel.place(relx = 0.05, rely = 0.17, relheight = 0.1, relwidth = 0.6)

    selectAdminLabel = tk.Label(mainframe, text = 'Please enter your username\nand select an admin user:', bg = 'gray', font = defaultFont)
    selectAdminLabel.place(relx = 0.05, rely = 0.29, relheight = 0.1, relwidth = 0.34)

    usernameEntry = tk.Entry(mainframe)
    usernameEntry.place(relx = 0.45, rely = 0.29, relheight = 0.1, relwidth = 0.45)

    listbox = tk.Listbox(mainframe, font = defaultFont)
    listbox.place(relx = 0.15, rely = 0.415, relheight = 0.21, relwidth = 0.7)
    adminUsers = viewingFromDatabase.viewFromDatabase(c, False, False, 'username', 'accountType', "'Admin'", 'UserDetails')
    position = 1
    for each in adminUsers:
        listbox.insert(position, each)
        position += 1
```

```
selectButton = tk.Button(mainframe, text = "Select", highlightbackground = '#424949', font = buttonFont, command =
selectButton.place(relx = 0.4, rely = 0.65, relheight = 0.1, relwidth = 0.2)
                                                    lambda: self.selectAdminUser(listbox.get(listbox.curselection())
                                                    , usernameEntry.get()))

loginAdminLabel = tk.Label(mainframe, text = 'Would you like to log in as an', bg = 'gray', font = defaultFont)
loginAdminLabel.place(relx = 0.05, rely = 0.78, relheight = 0.1, relwidth = 0.34)

adminLabel = tk.Label(mainframe, text = 'admin', bg = 'gray', fg = 'blue', font = defaultFont)
adminLabel.place(relx = 0.38, rely = 0.78, relheight = 0.1, relwidth = 0.07)
adminLabel.bind("<Button-1>", lambda x: teacherButtonClicked)

questionMarkLabel = tk.Label(mainframe, text = '?', bg = 'gray', font = defaultFont)
questionMarkLabel.place(relx = 0.445, rely = 0.78, relheight = 0.1, relwidth = 0.03)

returnButton = tk.Button(mainframe, text = "Return", highlightbackground = '#424949', fg = 'blue', font = defaultFont, command = lambda:
returnButton.place(relx = 0.77, rely = 0.87, relheight = 0.1, relwidth = 0.2)
                                                    self.returnPage(forgotPasswordPage))

forgotPasswordPage.mainloop()
```

Which with the above database produced:



The list-box here allows one of the values to be selected which is retrieved when passing it on to the selectAdminUser function when the Select button is pressed. The inputted username will also be passed as a parameter to this function for validation and to change the appropriate review ID in the database and accord it with the chosen admin user.

When instructing the selectAdminUser to simply display the value passed in to it, tested by pressing the second option, then the first, the third and finally the second again:

```
('emmahirst04',)
('hemanseego01',)
('harrymcdonagh05',)
('emmahirst04',)
```

However, I encountered this error:

```
_tkinter.TclError: bad listbox index "": must be active, anchor, end, @x,y, or a number
```

This occurred when no option from the list-box was selected.

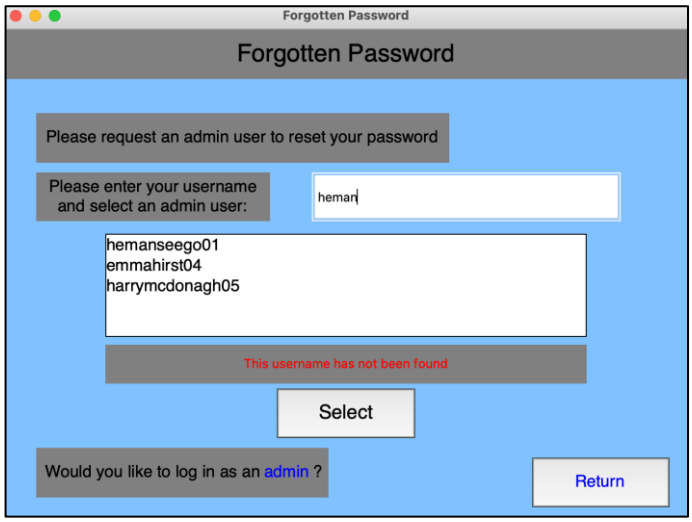
As a result, I therefore resorted to this method:

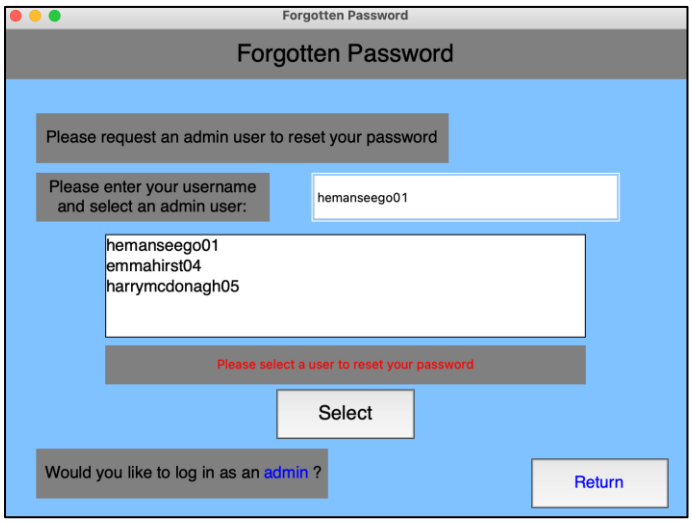
```
def selectAdminUser(self, listbox, username, errorLabel):
    ## Retrieve all usernames of the correct account type from database
    allUsers = viewingFromDatabase.viewFromDatabase(c, False, True, 'username', None, None, 'UserDetails')
    allUsersList = []
    for each in range(len(allUsers)):
        allUsersList.append(allUsers[each][0])
    ## Check whether inputted username is present in database
    if username in allUsersList:
        ## Retrieve selection from list-box (in the case of no selection, display error)
        try:
            adminUser = listbox.get(listbox.curselection())
            updatingDatabase.updateInDatabase(conn, c, 'reviewID', '3', 'username', "'"+username+"'", 'UserDetails')
            adminID = viewingFromDatabase.viewFromDatabase(c, False, False, 'userID', 'username', "'"+adminUser[0]+"'", 'UserDetails')
            updatingDatabase.updateInDatabase(conn, c, 'adminID', str(adminID[0][0]), 'username', "'"+username+"'", 'UserDetails')
            print('Done')
        except:
            error = 'Please select a user to reset your password'
            errorLabel.config(text = error, bg = 'gray', fg = 'red')
    else:
        ## If not found in database
        error = 'This username has not been found'
        errorLabel.config(text = error, bg = 'gray', fg = 'red')
```

In this modified function, I passed on the list-box instead of the predetermined value as a parameter, as it allowed me to use a try except statement when seeing whether any options had been selected. I extended the function to retrieve all users and match the given username against them to see if it exists. Only if this username is found to be present within the database, the function begins to retrieve the admin user selected, retrieve the relevant admin ID and update both the review ID and admin ID within the database for the given verified username. By including this within the if statement in this way, as before, it saves the program from unnecessarily executing code which will not anyways be needed. If the except statement is triggered, this indicates to the program that there has been an error in retrieving the admin user, which it assumes to mean that none has been selected, and therefore, as before creates a relevant error message as done in the login pages,

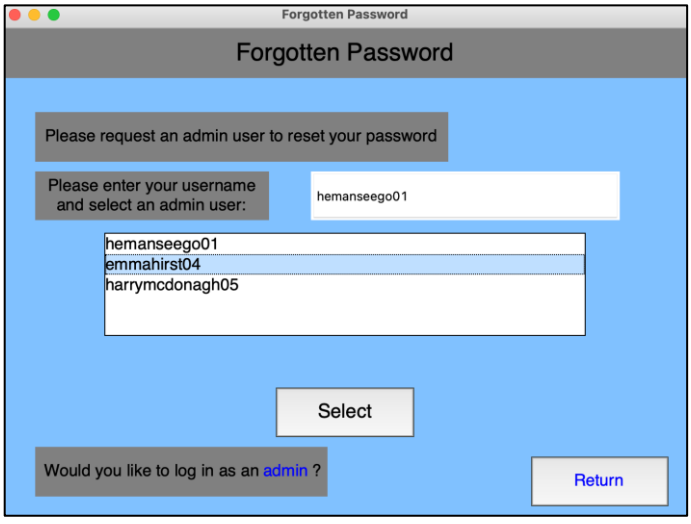
```
errorLabel = tk.Label(mainframe, bg = '#80c1ff')
errorLabel.place(relx = 0.15, rely = 0.64, relheight = 0.08, relwidth = 0.7)
```

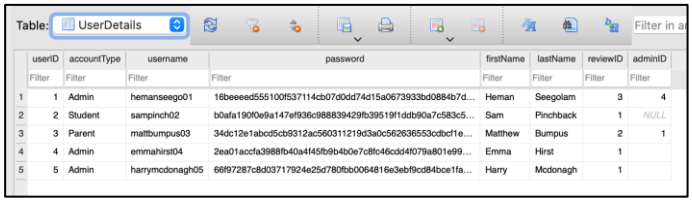
which initially has no values and is assigned the same background as the page to blend in. When the except statement or the else statement (in the case of the inputted username not being in the list), and appropriate error message is written and displayed.





When however completing said parts correctly:





As desired.

This method could mean that admin users accidentally select their own account to verify them which they obviously will be unable to do. To combat this, I shall simply leave this process to occur as many times as it may, so users are unable to request for a different admin user to reset their account.

Main Menu:

Since my main menu pages would make use of external images for a better aesthetic look, I first installed the pillow package and imported the Image and ImageTk libraries.

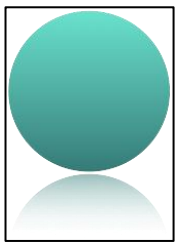
```
MacBook-Pro:~ hemans$ python3 -m pip install pillow
Collecting pillow
  Downloading https://files.pythonhosted.org/packages/ce/cd/3bd4b98bd0b9bba38951
102f465b665e794751de98ecb655001d0f2c0962/Pillow-8.1.0-cp38-cp38-macosx_10_10_x86
_64.whl (2.2MB)
     |████████████████████████████████| 2.2MB 1.2MB/s
Installing collected packages: pillow
Successfully installed pillow-8.1.0
```

```
from PIL import Image, ImageTk
```

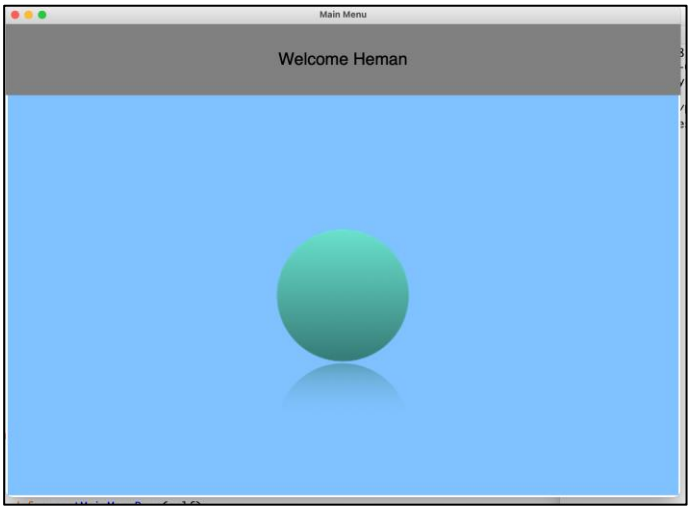
```
def adminMainMenuPage(self, firstName, titleFont, buttonFont, defaultFont):
    print('Admin Main Menu')
    mainMenuAdmin = tk.Tk()
    mainMenuAdmin.title("Main Menu")

    ## Retrieves the assigned width and height for the main menu page
    mainScreenWidth = mainMenu.getScreenWidth()
    mainScreenHeight = mainMenu.getScreenHeight()

    mainMenuAdmin.geometry(str(mainScreenWidth)+'x'+str(mainScreenHeight))

    canvas = tk.Canvas(mainMenuAdmin, height = mainScreenHeight, width = mainScreenWidth, bg = '#80c1ff')
    canvas.pack()

    welcomeLabel = tk.Label(canvas, text = "Welcome "+firstName, bg = 'gray', font = titleFont)
    welcomeLabel.place(relx = 0, rely = 0, relheight = 0.15, relwidth = 1)

    menuOptionImage = Image.open('MenuOption.png')
    menuOptionImage = menuOptionImage.resize((202,282))

    tkMenuOptionImage = ImageTk.PhotoImage(menuOptionImage)
    canvas.create_image(399,300, image = tkMenuOptionImage, anchor = 'nw')

    mainMenuAdmin.mainloop()
```

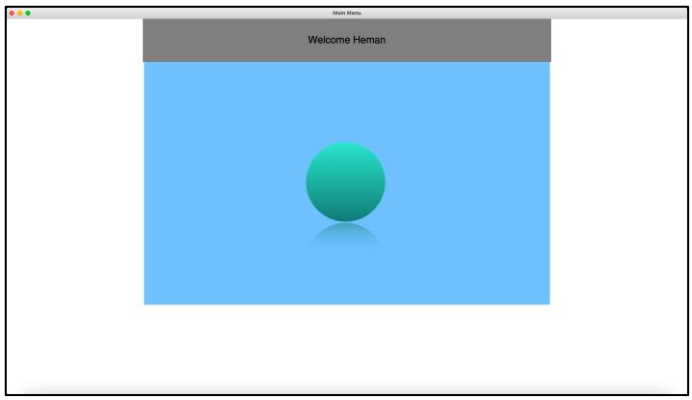
In this first stage of the function, I used the mainMenu object created earlier to retrieve the size of the window, however, I changed the dimensions to 700 x 1000, to make it look more professional. I approached this page differently to the other ones I've created thus far. This was because the images I were to include in my page had a transparent background and this transparency seemed to only be compatible with the canvas of a page, and not any frames. As a result, I had to fix the size of my pages using mainMenuAdmin.geometry() with the desired dimensions in string form to allow it to concatenate with the 'x'. I then opened the image file which I placed in the same directory as the python file and resized it to make it a bit bigger. This had to be done using the Image library from pillow. Then, using the ImageTk library, I created a photo-image in tkinter using the newly created image variable and organised where to place it in the canvas using a number of pixels for the x and the y direction. I anchored this image to the north west (or the top left) of the page so I could decide how far along from that position it had to translate.

The imaged used was:



There were initially some issues with this:





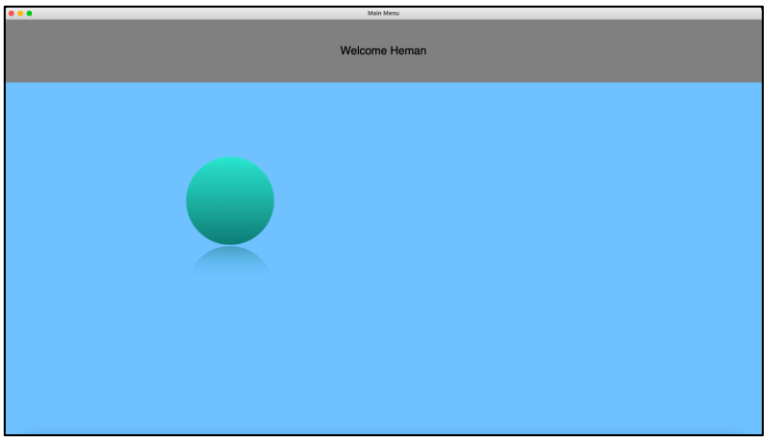
The first issue was a very thin white border across the whole of the canvas which made the program appear very unprofessional. The second issue here was that when enlarged, the canvas did not change size and it did not expand with the rest of the window, again very unprofessional. To combat this:

```
canvas = tk.Canvas(mainMenuAdmin, height = mainScreenHeight, width = mainScreenWidth, bg = '#80c1ff', highlightthickness=0, relief='ridge')
```

This successfully got rid of the white border.
Initially, to combat the issue of expanding the window, I did this:

```
canvas.pack(fill = 'both', expand = True)
```

However, whilst it solved the issue, it brought another to light:



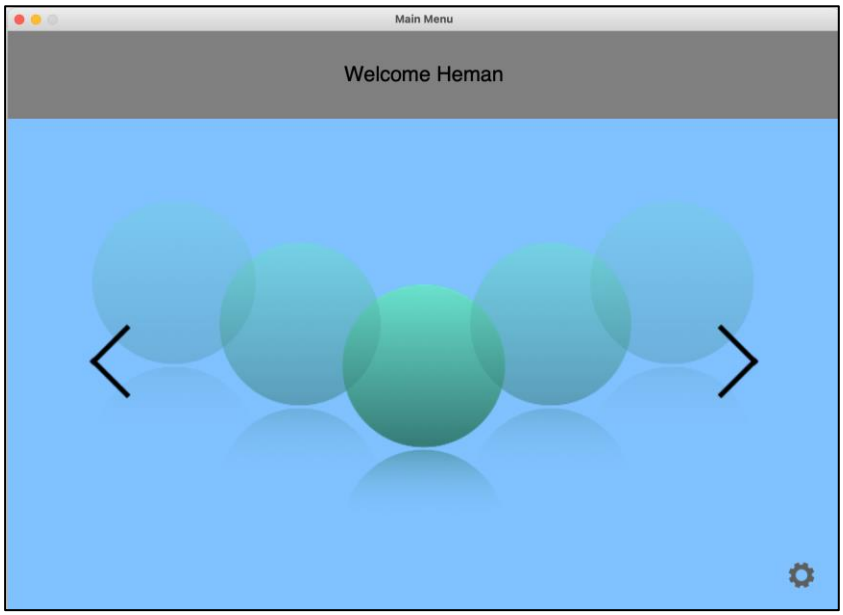
Here, when expanded, the canvas now covered the whole area of the window, but the image was now not centred as it should. Since a user could enlarge the window by any size the wished, it would be hard to move the image for every given dimension. I decided to make the window not resizable:

```
mainMenuAdmin.resizable(False, False)
```

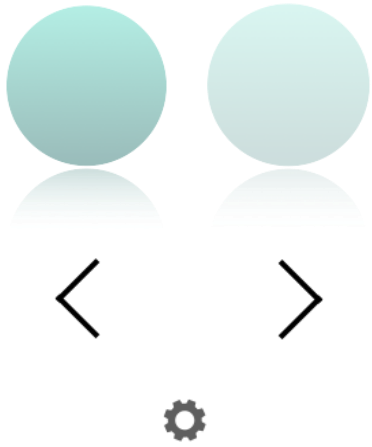
There were some limitations to this since the user could not expand the window if they wanted to. I decided that later, I could provide a setting where the user inputted acceptable dimensions and the program would redesign the page for them.

I imported the rest of the images, created them in the correct tkinter format and placed them in a professional manner:

```
menuOptionImage = Image.open('MenuOption.png')
menuOptionImage = menuOptionImage.resize((202,282))

faintMenuOptionImage = Image.open('FaintMenuOption.png')
faintMenuOptionImage = faintMenuOptionImage.resize((202,282))

faintMenuOptionImage2 = Image.open('FaintMenuOption2.png')
faintMenuOptionImage2 = faintMenuOptionImage2.resize((202,282))

rightArrowImage = Image.open('RightArrow.png')
rightArrowImage = rightArrowImage.resize((54,93))

leftArrowImage = Image.open('LeftArrow.png')
leftArrowImage = leftArrowImage.resize((54,93))

settingsImage = Image.open('Settings.png')

tkMenuOptionImage = ImageTk.PhotoImage(menuOptionImage)
tkFaintMenuOptionImage = ImageTk.PhotoImage(faintMenuOptionImage)
tkFaintMenuOptionImage2 = ImageTk.PhotoImage(faintMenuOptionImage2)
tkRightArrowImage = ImageTk.PhotoImage(rightArrowImage)
tkLeftArrowImage = ImageTk.PhotoImage(leftArrowImage)
tkSettingsImage = ImageTk.PhotoImage(settingsImage)

canvas.create_image(399,300, image = tkMenuOptionImage, anchor = 'nw')
canvas.create_image(250,250, image = tkFaintMenuOptionImage, anchor = 'nw')
canvas.create_image(551,250, image = tkFaintMenuOptionImage, anchor = 'nw')
canvas.create_image(100,200, image = tkFaintMenuOptionImage2, anchor = 'nw')
canvas.create_image(698,200, image = tkFaintMenuOptionImage2, anchor = 'nw')
canvas.create_image(850,350, image = tkRightArrowImage, anchor = 'nw')
canvas.create_image(96,350, image = tkLeftArrowImage, anchor = 'nw')
canvas.create_image(935,635, image = tkSettingsImage, anchor = 'nw')
```



With the respective images:



To add some functionality to the arrows:

```
canvas.tag_bind(rightArrow, "<Button-1>", lambda x: self.arrowPressed(menuOption))
canvas.tag_bind(leftArrow, "<Button-1>", lambda x: self.arrowPressed(menuOption))
```

For testing:

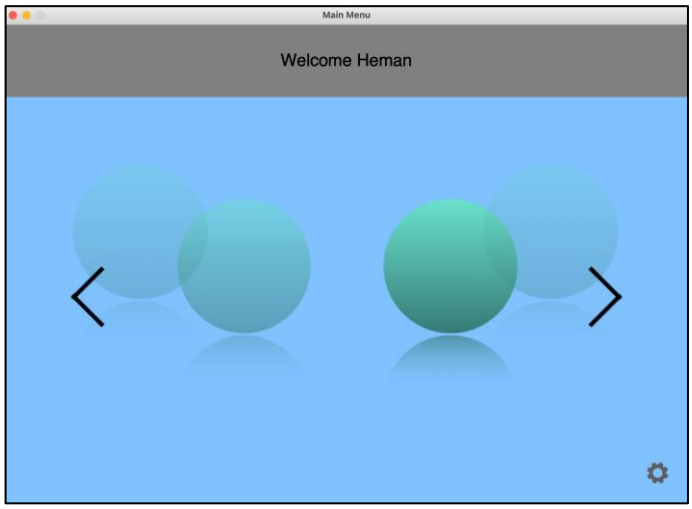
```
def arrowPressed(self, menuOption):
    print('Arrow Pressed')
```

```
Arrow Pressed
Arrow Pressed
```

I experimented with moving the images when pressing an arrow.

```
def arrowPressed(self, mainMenuAdmin, canvas, menuOption):
    print('Arrow Pressed')
    coordinates = canvas.coords(menuOption)
    print(coordinates)
    while coordinates != [551.0, 262.0]:
        canvas.move(menuOption, 4, -1)
        mainMenuAdmin.update()
        coordinates = canvas.coords(menuOption)
    while coordinates != [551.0, 250.0]:
        canvas.move(menuOption, 0, -1)
        mainMenuAdmin.update()
        coordinates = canvas.coords(menuOption)
```

In this new modified function, I passed in the window variable, the canvas and the menu option image to the arrowPressed function. This allowed me to first retrieve the coordinates of the image. This was the same as what I set it to earlier, so [399.0, 300.0]. I wanted to translate this image to the position of the next menu option to its right, coordinates [551.0, 250.0]. Using the .move() function required first the image being translated, and the 'speed' or the amount of pixels it translated in each the x and y-direction. The difference in the x-direction of the two positions was 152 pixels. This meant that the only x 'speeds' available were 1, 2, 4, 8, 19, 38, 76 and 152. To display a professional animation to the user which was neither too slow nor too fast, I chose to have a x-direction displacement of 4. The difference in the y-direction, however, between the two positions was 50 which posed problematic since not divisible by any of the above numbers. Hence, I created the animation in two separate motions. I calculated the new y-position with a displacement of -1 for every 4 pixels in the x-direction. From calculating 152 divided by 4, this showed that the first motion had a y-direction displacement of -38 bringing it to y-coordinate 262.0. Then I translated the image in a strict y-direction until it reached its final destination. After every move within the while loop, I updated the window to show the movement of the image.



However, as I needed all the images to move in the correct direction in accordance with the arrow, and so couldn't rely on such precise figures.

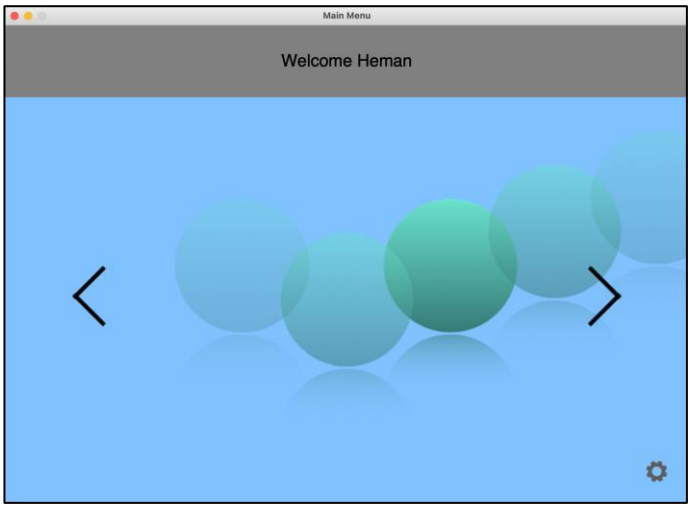
To make the program more efficient and to make all animations in time, I preferred causing all movements within the same loop rather than doing them separately. To allow for this, I modified the initial positions of the faint menu options slightly to make the calculations more efficient.

```
menuOption = canvas.create_image(399,300, image = tkMenuOptionImage, anchor = 'nw')
faintMenuOption1 = canvas.create_image(247,250, image = tkFaintMenuOptionImage, anchor = 'nw')
faintMenuOption2 = canvas.create_image(551,250, image = tkFaintMenuOptionImage, anchor = 'nw')
fainterMenuOption1 = canvas.create_image(95,200, image = tkFaintMenuOptionImage2, anchor = 'nw')
fainterMenuOption2 = canvas.create_image(703,200, image = tkFaintMenuOptionImage2, anchor = 'nw')
rightArrow = canvas.create_image(850,350, image = tkRightArrowImage, anchor = 'nw')
leftArrow = canvas.create_image(96,350, image = tkLeftArrowImage, anchor = 'nw')
settings = canvas.create_image(935,635, image = tkSettingsImage, anchor = 'nw')
```

Here, I made each menu option have a space of 152 pixels in the x-direction and 50 pixels in the y-direction for the first instance.
I then produced this function:

```
def  leftArrowPressed(self, mainMenuAdmin, canvas, menuOption, faintMenuOption1, faintMenuOption2, fainterMenuOption1, fainterMenuOption2):
    for i in range(1,39):
        canvas.move(menuOption, 4, -1)
        canvas.move(faintMenuOption1, 4, 1)
        canvas.move(faintMenuOption2, 4, -1)
        canvas.move(fainterMenuOption1, 4, 1)
        canvas.move(fainterMenuOption2, 4, -1)
        mainMenuAdmin.update()
    for j in range(1,13):
        canvas.move(menuOption, 0, -1)
        canvas.move(faintMenuOption1, 0, 1)
        canvas.move(faintMenuOption2, 0, -1)
        canvas.move(fainterMenuOption1, 0, 1)
        canvas.move(fainterMenuOption2, 0, -1)
        mainMenuAdmin.update()
```

Using the calculations done previously, to show that the image first completed 38 rounds of movement, I was able to use a for loop to do this. This didn't require me to use any coordinates to track movement. For all elements on the right side of and including the main option, the image moved 4 pixels in an x-direction and -1 pixels in a positive y-direction per iteration. However, all elements to the left of the main option moved 4 pixels in an x-direction and 1 pixel in a y-direction per iteration. In the second part of the motion, all to the left moved down, and all to the right moved up to finalise their position.

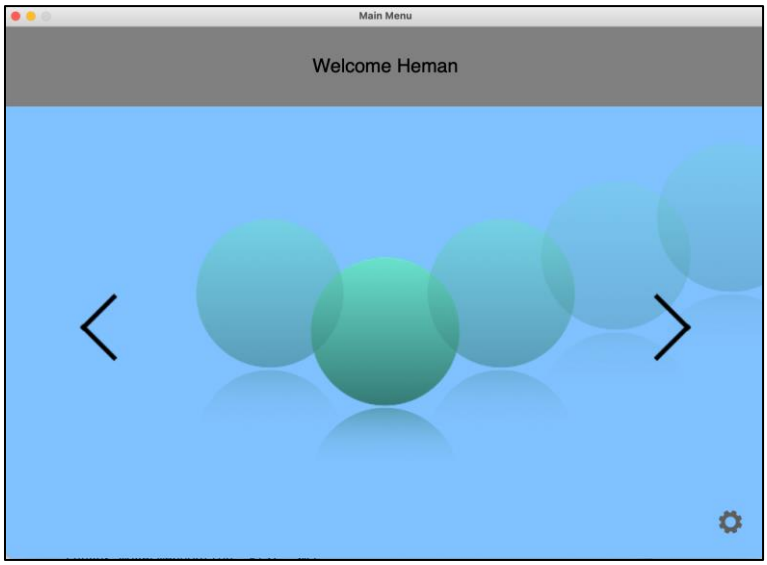


This successfully moved all the menu options to the desired position. To finish off this movement, I needed to correct the images in terms of their transparency for the position they were in.

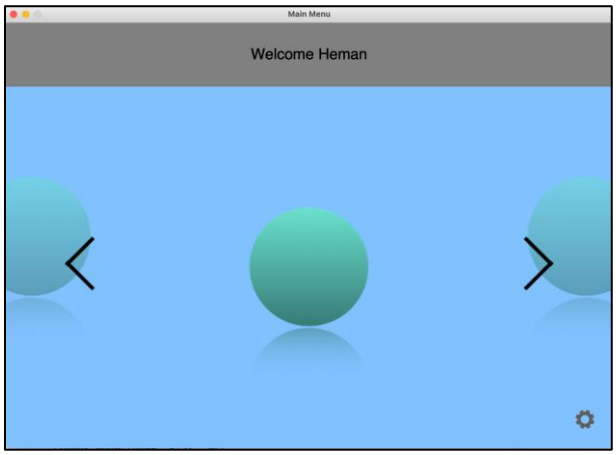
To do this:

```
canvas.move(menuOption, -152, 50)
canvas.move(faintMenuOption1, 152, -50)
canvas.move(faintMenuOption2, -456, 50)
canvas.move(fainterMenuOption1, 456, -50)
```

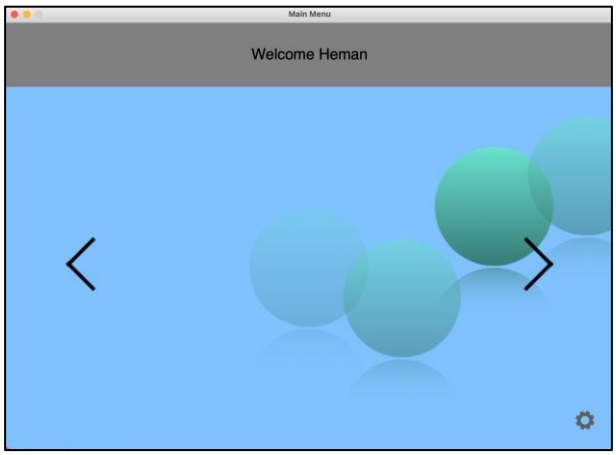
Once all motions were finished, I swapped the bolder image with the one in the centre, and the faint one on its new right with the one on the complete left. This produced:



However, this only worked for one press. After one press:



Without the movements added in at the end:



To combat this, I created two lists to show what was on left and right of the selected option.

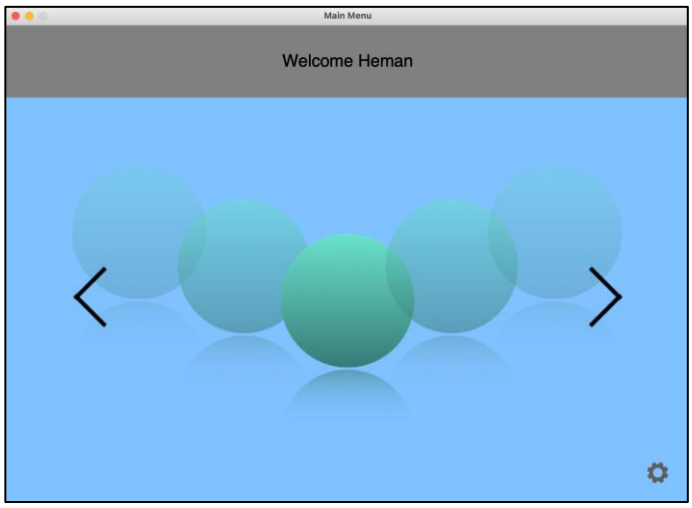
```
def leftArrowPressed(self, mainMenuAdmin, canvas, one, two, three, four, five, tkMenuOptionImage, tkFaintMenuOptionImage):
    leftList = []
    rightList = []
    menuOptionsList = [one, two, three, four, five]

    for each in menuOptionsList:
        if ((canvas.coords(each))[0]) < 399:
            leftList.append(each)
        else:
            rightList.append(each)

    if len(leftList) != 0:
        for i in range(1,39):
            if len(leftList) > 1:
                canvas.move(one, 4, 1)
                canvas.move(two, 4, 1)
                canvas.move(three, 4, -1)
                canvas.move(four, 4, -1)
                canvas.move(five, 4, -1)
            elif len(leftList) > 0:
                canvas.move(four, 4, 1)
                canvas.move(three, 4, -1)
                canvas.move(two, 4, -1)
                canvas.move(one, 4, -1)
                canvas.move(five, 4, -1)
            mainMenuAdmin.update()
        for j in range(1,13):
            if len(leftList) > 1:
                canvas.move(one, 0, 1)
                canvas.move(two, 0, 1)
                canvas.move(three, 0, -1)
                canvas.move(four, 0, -1)
                canvas.move(five, 0, -1)
            elif len(leftList) > 0:
                canvas.move(four, 0, 1)
                canvas.move(three, 0, -1)
                canvas.move(two, 0, -1)
                canvas.move(one, 0, -1)
                canvas.move(five, 0, -1)
            mainMenuAdmin.update()
```

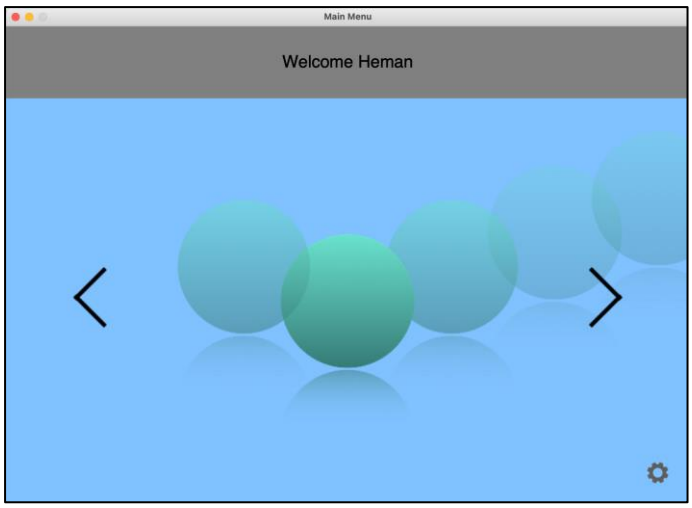
In this modified function, I created a list for all options on the left of the selected one and a list for all on the right. This was done at the start of the function on every press of the arrow by analysing the x-coordinates of each of the options against the base value of 399. All less than 399 were appended to the left list and all greater than, to the right list.
This allowed the program to decide what would occur. The function first determines whether the length of the list is above 0. If it isn't, no result is obtained; the user is unable to move the options in that direction any longer. if it is above 0, the options are moved accordingly. To make understanding easier, I numbered the options from left to right, one to five. Within the for loop, another if statement is executed to check whether the length of the left list is 1 or 2. If it is 2, the options move as before.

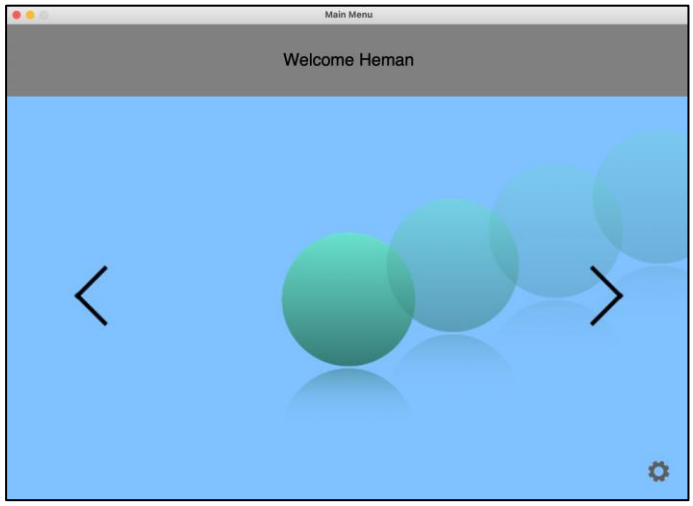
```
if len(leftList) > 1:
    canvas.move(three, -152, 50) ## Swap with two
    canvas.move(two, 152, -50) ## Swap with three
    canvas.move(four, -456, 50) ## Swap with one
    canvas.move(one, 456, -50) ## Swap with four
elif len(leftList) > 0:
    canvas.move(three, -152, 50) ## Swap with four (to position 1)
    canvas.move(four, 152, -50) ## Swap with three (to position 2)
    canvas.move(five, -304, 100) ## Swap with two (to out of frame)
    canvas.move(two, 304, -100) ## Swap with five (to position 3)
```

However, since the next part of the function changes the order of the options (after one press, become 43215), I had to organise the options according to their respective name and not their numbered position.

This allowed it to work as desired. For now it takes care of situations where the length of the left list is 0, 1 or 2, but as of yet disregards the opportunity for 3 or 4 if the user was to first move across to the right first. This will be dealt with after creating the rightArrowPressed function.







The button binds now looked like this:

```
canvas.tag_bind(rightArrow, "<Button-1>", lambda x: self.rightArrowPressed(mainMenuAdmin, canvas, fainterMenuOption1, faintMenuOption1, menuOption,
canvas.tag_bind(leftArrow, "<Button-1>", lambda x: self.leftArrowPressed(mainMenuAdmin, canvas, fainterMenuOption1, faintMenuOption1, menuOption, f
                                                  , faintMenuOption2, fainterMenuOption2, tkMenuOptionImage, tkFaintMenuOptionImage))
                                                  faintMenuOption2, fainterMenuOption2, tkMenuOptionImage, tkFaintMenuOptionImage))
```

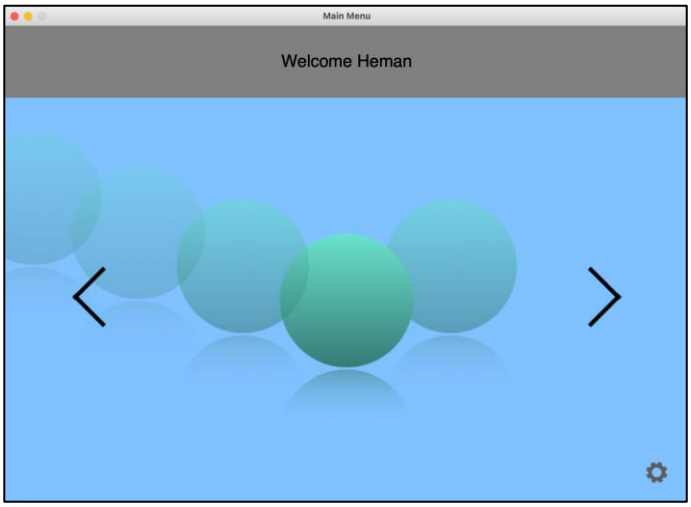
rightArrowPressed function:

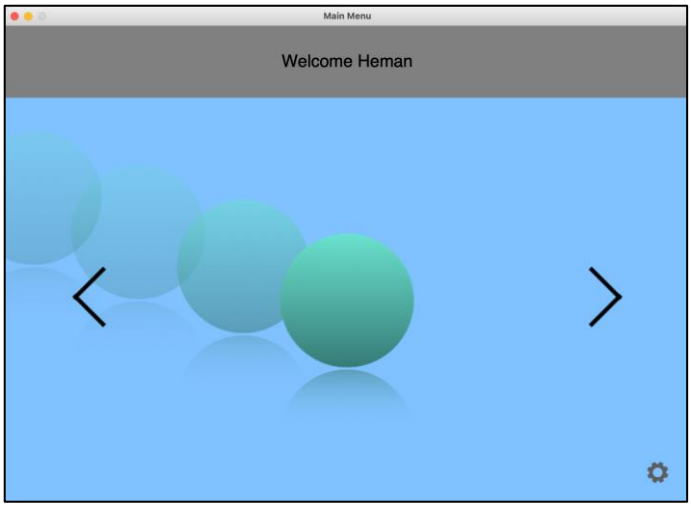
```
def rightArrowPressed(self, mainMenuAdmin, canvas, one, two, three, four, five, tkMenuOptionImage, tkFaintMenuOptionImage):
    leftList = []
    rightList = []
    menuOptionsList = [one, two, three, four, five]

    for each in menuOptionsList:
        if ((canvas.coords(each))[0]) < 399:
            leftList.append(each)
        elif ((canvas.coords(each))[0]) > 550:
            rightList.append(each)
```

In this first bit of the function, it resembles very closely to the function above. However, I made some changes to the else statement in the above function. This was because the if statement looked at the x-coordinate for all the menu options and compared them with the coordinate 399 to add them into the left list, or else the right list. This meant that it included the selected menu option in the right list. I therefore changed the else statement to an elif statement which looked for all options with an x-coordinate greater than 550 (since next option occurred at x-coordinate 551). This filtered the options as desired.

```
if len(rightList) != 0:
    for i in range(1,39):
        if len(rightList) > 1:
            canvas.move(one, -4, -1)
            canvas.move(two, -4, -1)
            canvas.move(three, -4, -1)
            canvas.move(four, -4, 1)
            canvas.move(five, -4, 1)
        elif len(rightList) > 0:
            canvas.move(one, -4, -1)
            canvas.move(five, -4, -1)
            canvas.move(four, -4, -1)
            canvas.move(three, -4, -1)
            canvas.move(two, -4, 1)
        mainMenuAdmin.update()
    for j in range(1,13):
        if len(rightList) > 1:
            canvas.move(one, 0, -1)
            canvas.move(two, 0, -1)
            canvas.move(three, 0, -1)
            canvas.move(four, 0, 1)
            canvas.move(five, 0, 1)
        elif len(rightList) > 0:
            canvas.move(one, 0, -1)
            canvas.move(five, 0, -1)
            canvas.move(four, 0, -1)
            canvas.move(three, 0, -1)
            canvas.move(two, 0, 1)
        mainMenuAdmin.update()

    if len(rightList) > 1:
        canvas.move(three, 152, 50) ## Swap with four
        canvas.move(four, -152, -50) ## Swap with three
        canvas.move(five, -456, -50) ## Swap with two
        canvas.move(two, 456, 50) ## Swap with five
    elif len(rightList) > 0:
        canvas.move(three, 152, 50) ## Swap with two (to position 5)
        canvas.move(two, -152, -50) ## Swap with three (to position 4)
        canvas.move(four, -304, -100) ## Swap with one (to out of frame)
        canvas.move(one, 304, 100) ## Swap with four (to position 3)
```

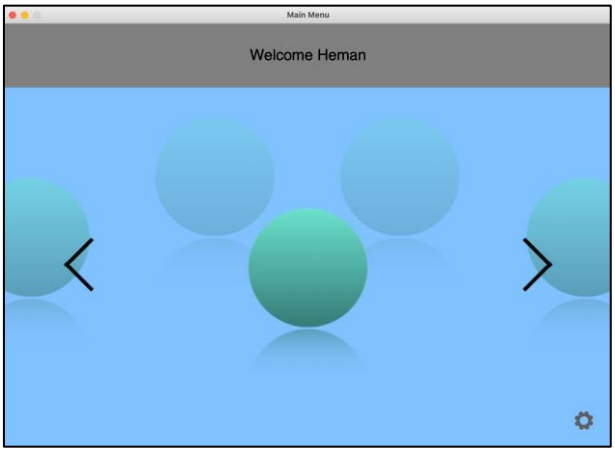
Using the same process above, I modified the numbers accordingly to provide an animation of the menu options and swap the relevant options. In this function, they all consider the right list instead of the left list.



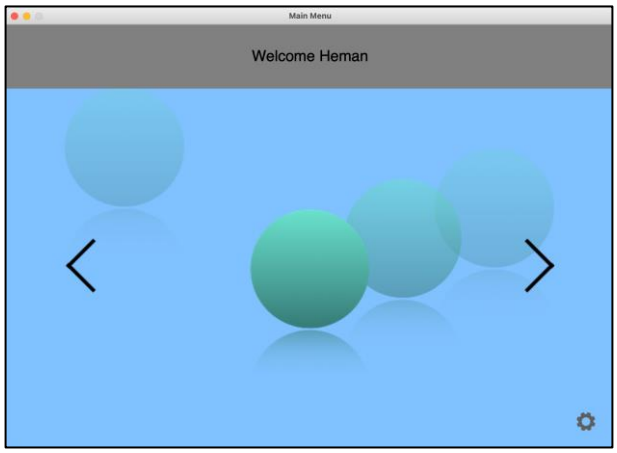


Again, as above, the function was not yet compatible with successfully carrying out animations for where after going one direction, returning to the other direction:

After one right press, then one left:



After two right presses, then one left:



I finalised the rightArrowPressed function by modifying the motion section of the code:

```
if len(rightList) != 0:
    for i in range(1,39):
        if len(rightList) == 2:
            canvas.move(one, -4, -1)
            canvas.move(two, -4, -1)
            canvas.move(three, -4, -1)
            canvas.move(four, -4, 1)
            canvas.move(five, -4, 1)
        elif len(rightList) == 1:
            ## 15432
            canvas.move(one, -4, -1)
            canvas.move(five, -4, -1)
            canvas.move(four, -4, -1)
            canvas.move(three, -4, -1)
            canvas.move(two, -4, 1)
```

```
        elif len(rightList) == 3:
            ## 43215
            canvas.move(four, -4, -1)
            canvas.move(three, -4, -1)
            canvas.move(two, -4, 1)
            canvas.move(one, -4, 1)
            canvas.move(five, -4, 1)
        elif len(rightList) == 4:
            ## 34512
            canvas.move(three, -4, -1)
            canvas.move(four, -4, 1)
            canvas.move(five, -4, 1)
            canvas.move(one, -4, 1)
            canvas.move(two, -4, 1)
        mainMenuAdmin.update()
```

Here I firstly changed the < and > to ==, since when trying to obtain a length of list greater than 2 for a size of 3, this would also comply with the condition of being greater than 1 and so would interfere. In this way, there is no chance of interference. I then extended it to include options for right list sizes of 3 and 4. The comments above each section represent the position of the options before being moved.

```
for j in range(1,13):
    if len(rightList) == 2:
        canvas.move(one, 0, -1)
        canvas.move(two, 0, -1)
        canvas.move(three, 0, -1)
        canvas.move(four, 0, 1)
        canvas.move(five, 0, 1)
    elif len(rightList) == 1:
        canvas.move(one, 0, -1)
        canvas.move(five, 0, -1)
        canvas.move(four, 0, -1)
        canvas.move(three, 0, -1)
        canvas.move(two, 0, 1)
    elif len(rightList) == 3:
        canvas.move(four, 0, -1)
        canvas.move(three, 0, -1)
        canvas.move(two, 0, 1)
        canvas.move(one, 0, 1)
        canvas.move(five, 0, 1)
    elif len(rightList) == 4:
        canvas.move(three, 0, -1)
        canvas.move(four, 0, 1)
        canvas.move(five, 0, 1)
        canvas.move(one, 0, 1)
        canvas.move(two, 0, 1)
    mainMenuAdmin.update()
```

```
if len(rightList) == 2:
    ## From 12345 to 43125
    canvas.move(three, 152, 50) ## Swap with four
    canvas.move(four, -152, -50) ## Swap with three
    canvas.move(five, -456, -50) ## Swap with two
    canvas.move(two, 456, 50) ## Swap with five
    ## From 43125 to 13452
elif len(rightList) == 1:
    canvas.move(three, 152, 50) ## Swap with two (to position 5)
    canvas.move(two, -152, -50) ## Swap with three (to position 4)
    canvas.move(four, -304, -100) ## Swap with one (to out of frame)
    canvas.move(one, 304, 100) ## Swap with four (to position 3)
elif len(rightList) == 3:
    ## Back to 12345 from 43215
    canvas.move(three, 152, 50) ## Swap with two
    canvas.move(two, -152, -50) ## Swap with three
    canvas.move(four, 456, 50) ## Swap with one
    canvas.move(one, -456, -50) ## Swap with four
elif len(rightList) == 4:
    ## Back to 43215 from 34512
    canvas.move(three, 152, 50) ## Swap with four
    canvas.move(four, -152, -50) ## Swap with three
    canvas.move(two, -304, 100) ## Swap with five
    canvas.move(five, 304, -100) ## Swap with two
```

I then moved on to the next part to finalise the positions of the options when the size of the right list was 3 or 4, and swapped the options accordingly once moved. When according these swaps, I had to be careful to move them into the same order as it should be for every way of getting to an option. The use of comments and previous stages helped me do this.

I was now able to move all the way to the left and to the complete right. I hence modified and extended the leftArrowPressed function.

```
if len(leftList) != 0:
    for i in range(1,39):
        if len(leftList) == 2:
            canvas.move(one, 4, 1)
            canvas.move(two, 4, 1)
            canvas.move(three, 4, -1)
            canvas.move(four, 4, -1)
            canvas.move(five, 4, -1)
        elif len(leftList) == 1:
            ##43215
            canvas.move(four, 4, 1)
            canvas.move(three, 4, -1)
            canvas.move(two, 4, -1)
            canvas.move(one, 4, -1)
            canvas.move(five, 4, -1)
        elif len(leftList) == 3:
            ## 15432
            canvas.move(one, 4, 1)
            canvas.move(five, 4, 1)
            canvas.move(four, 4, 1)
            canvas.move(three, 4, -1)
            canvas.move(two, 4, -1)
        elif len(leftList) == 4:
            ## 45123
            canvas.move(four, 4, 1)
            canvas.move(five, 4, 1)
            canvas.move(one, 4, 1)
            canvas.move(two, 4, 1)
            canvas.move(three, 4, -1)
        mainMenuAdmin.update()
```

```
    for j in range(1,13):
        if len(leftList) == 2:
            canvas.move(one, 0, 1)
            canvas.move(two, 0, 1)
            canvas.move(three, 0, -1)
            canvas.move(four, 0, -1)
            canvas.move(five, 0, -1)
        elif len(leftList) == 1:
            canvas.move(four, 0, 1)
            canvas.move(three, 0, -1)
            canvas.move(two, 0, -1)
            canvas.move(one, 0, -1)
            canvas.move(five, 0, -1)
        elif len(leftList) == 3:
            canvas.move(one, 0, 1)
            canvas.move(five, 0, 1)
            canvas.move(four, 0, 1)
            canvas.move(three, 0, -1)
            canvas.move(two, 0, -1)
        elif len(leftList) == 4:
            canvas.move(four, 0, 1)
            canvas.move(five, 0, 1)
            canvas.move(one, 0, 1)
            canvas.move(two, 0, 1)
            canvas.move(three, 0, -1)
        mainMenuAdmin.update()
```

```
if len(leftList) == 2:
    ## From 12345 to 43215
    canvas.move(three, -152, 50) ## Swap with two
    canvas.move(two, 152, -50) ## Swap with three
    canvas.move(four, -456, 50) ## Swap with one
    canvas.move(one, 456, -50) ## Swap with four
elif len(leftList) == 1:
    ## From 43215 to 34512
    canvas.move(three, -152, 50) ## Swap with four (to position 1)
    canvas.move(four, 152, -50) ## Swap with three (to position 2)
    canvas.move(five, -304, 100) ## Swap with two (to out of frame)
    canvas.move(two, 304, -100) ## Swap with five (to position 3)
elif len(leftList) == 3:
    ## Back to 12345 from 15432
    canvas.move(three, -152, 50) ## Swap with four
    canvas.move(four, 152, -50) ## Swap with three
    canvas.move(five, 456, -50) ## Swap with two
    canvas.move(two, -456, 50) ## Swap with five
elif len(leftList) == 4:
    print('done')
    ## Back to 15432 from 45123
    canvas.move(one, -304, -100) ## Swap with four
    canvas.move(four, 304, 100) ## Swap with one
    canvas.move(three, -152, 50) ## Swap with two
    canvas.move(two, 152, -50) ## Swap with three
```

Now that all menu options functioned as desired, I assigned a label to show the user what function each option has. Initially I started to do this by placing the label on top of the menu options. I set this initially to 'Play Quiz'.

```
optionLabel = tk.Label(canvas, text = "Play Quiz", bg = '#80c1ff', font = titleFont)
optionLabel.place(relx = 0.4, rely = 0.32, relheight = 0.06, relwidth = 0.2)
```

However, since the options were gradient images, I found no background which looked professional for the label to appear on the option itself. I hence resorted to placing the option label just above the central option. For further professionalism and aesthetic purposes, I created an image of a select button which would fit my menu.

I had to import this image, create a variable for this image using the tkinter photo-image library, and place it on the canvas in an appropriate place. I also resized it appropriately.

```
selectButtonImage = Image.open('SelectButton.png')
selectButtonImage = selectButtonImage.resize((135,55))
```

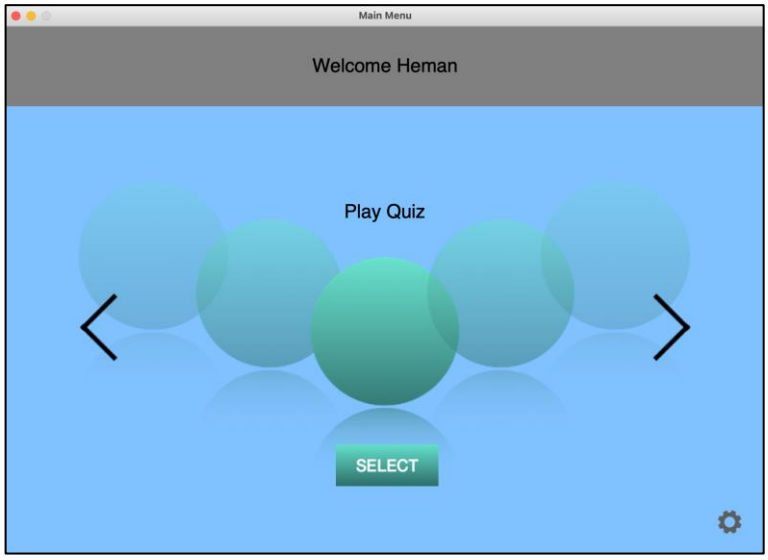
```
tkSelectButtonImage = ImageTk.PhotoImage(selectButtonImage)
```

```
selectButton = canvas.create_image(435,550, image = tkSelectButtonImage, anchor = 'nw')
```

I also bound the select button and settings images to their respective functions:

```
canvas.tag_bind(selectButton, "<Button-1>", lambda x: self.selectButtonPressed(optionLabel))
canvas.tag_bind(settings, "<Button-1>", lambda x: self.settingsMenu())
```

The main menu page now looked like this:



I went on to configure the option label to show the different menu option functions when an arrow was clicked. To do this, I used the length of the left and right lists again within the left arrow and right arrow functions, which required me to coordinate carefully the desired text for each option. They all had to be the same no matter which arrow the user pressed and in which order they were pressed.

In the leftArrowPressed function:

```
if len(leftList) == 2:
    optionLabel.config(text = 'View Quizzes')
elif len(leftList) == 1:
    optionLabel.config(text = 'Create Quiz')
elif len(leftList) == 3:
    optionLabel.config(text = 'Play Quiz')
elif len(leftList) == 4:
    optionLabel.config(text = 'View Classes')
```

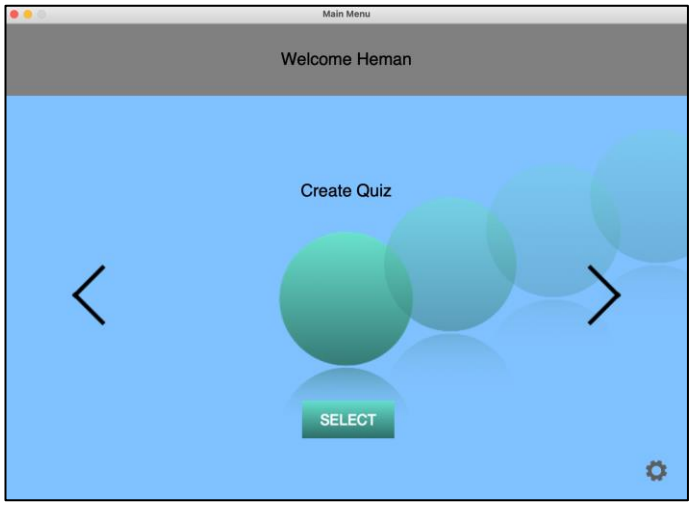
I included this section of code within the if statement which checked whether the length of the left list was 0. Initially, I placed all the configurations of the option label within the for loop, but decided against it since it would configure the label as many times as the loop occurred, and hence would increase the time for the movements to occur. I therefore created another if statement after all of the initial movements of 4 pixels right and 1 pixel up or down accordingly. This if statement, considered the lengths of the left list and replaced the text of the option label with the ones set above. I didn't have to consider where the length of the left list was 5 since there would never be a way for the user to get to that option using the left arrow, since it on the furthest right. By placing this in between the two movements (after this if statement comes the movement up or down 1 pixel times 12 accordingly), the animations and transitions appeared more smooth and professional.

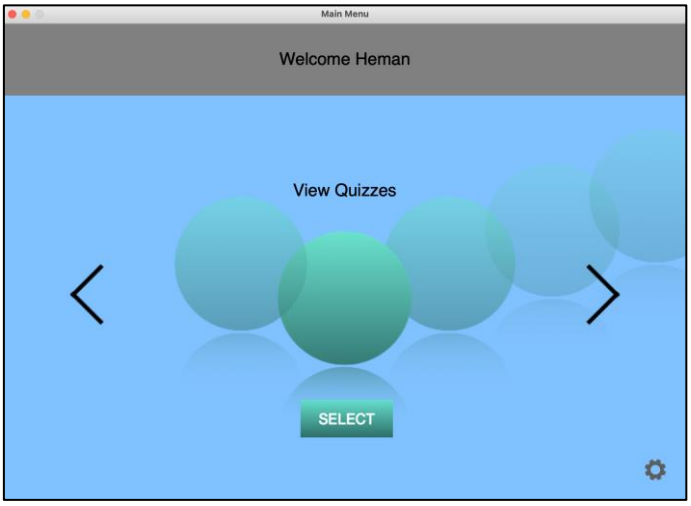
I did the same for the rightArrowPressed function:

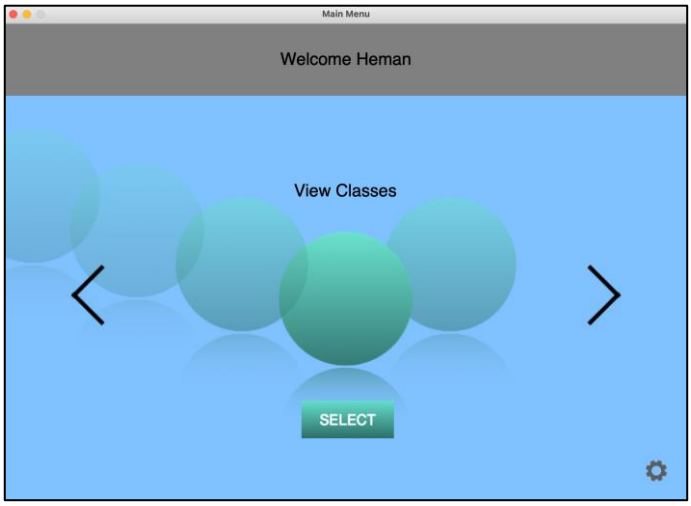
```
if len(rightList) == 2:
    optionLabel.config(text = 'View Classes')
elif len(rightList) == 1:
    optionLabel.config(text = 'Log off')
elif len(rightList) == 3:
    optionLabel.config(text = 'Play Quiz')
elif len(rightList) == 4:
    optionLabel.config(text = 'View Quizzes')
```

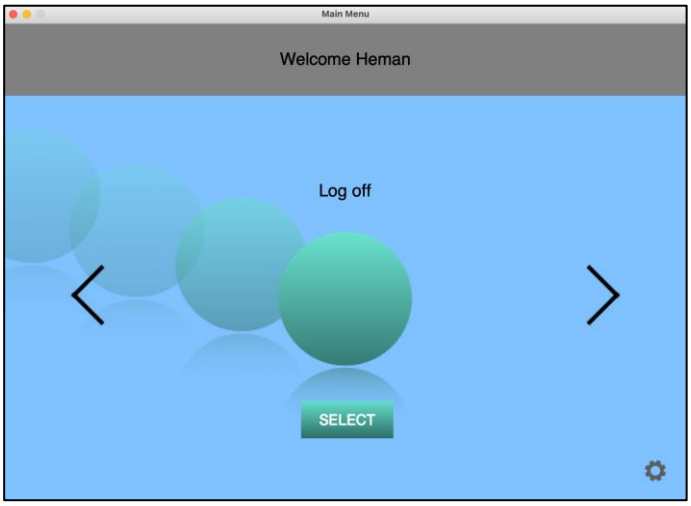
In these two sections of code, where the lengths are 3, the text matches, where they are 2 and 4 respectively, they match and finally where they are 1, the texts are unique.

This produced:










```
def selectButtonPressed(self, menuOption):
    print('Select')
    if menuOption.cget('text') == 'Play Quiz':
        self.playQuiz()
    elif menuOption.cget('text') == 'View Quizzes':
        self.viewQuizzes()
    elif menuOption.cget('text') == 'Create Quiz':
        self.createQuiz()
    elif menuOption.cget('text') == 'View Classes':
        self.viewClasses()
    elif menuOption.cget('text') == 'Log off':
        self.logOff()

def settingsMenu(self, settings):
    print('Settings')

def playQuiz(self):
    print('Play Quiz')

def viewQuizzes(self):
    print('View Quiz')

def createQuiz(self):
    print('Create Quiz')

def viewClasses(self):
    print('View Classes')

def logOff(self):
    print('Log Off')
```

In the selectButtonPressed function, I passed in the optionLabel as a parameter. This allowed me to retrieve the text of the label at the time the button is pressed. I did this using the .cget() method, where it retrieved the value of the configuration of the specified attribute, in this case the text. Where the labels are found to be each of the relevant options, they are directed to the corresponding function. These function as of yet just display their role to test the authenticity of the select image bind, which was shown to be successful.

Settings:

```
def settingsMenu(self, titleFont, buttonFont, defaultFont):
    settingsPage = tk.Tk()
    settingsPage.title("Settings")

    ## Retrieves the assigned width and height for the main menu page
    mainScreenWidth = settings.getScreenWidth()
    mainScreenHeight = settings.getScreenHeight()

    settingsPage.resizable(False, False)

    canvas = tk.Canvas(settingsPage, height = mainScreenHeight, width = mainScreenWidth)
    canvas.pack()

    mainframe = tk.Frame(canvas, bg = '#80c1ff')
    mainframe.place(relheight = 1, relwidth = 1)

    welcomeLabel = tk.Label(mainframe, text = "Settings", bg = 'gray', font = titleFont)
    welcomeLabel.place(relx = 0, rely = 0, relheight = 0.15, relwidth = 1)

    reviewUsersFrame = tk.Frame(mainframe, bg = 'gray')
    reviewUsersFrame.place(relx = 0.3, rely = 0.25, relheight = 0.15, relwidth = 0.4)

    reviewUsersButton = tk.Button(reviewUsersFrame, text = "Review Users", bg = '#E5E7E9', font = buttonFont, command = self.reviewUsers)
    reviewUsersButton.place(relx = 0.03, rely = 0.1, relheight = 0.8, relwidth = 0.94)

    changeMenuDimensionsFrame = tk.Frame(mainframe, bg = 'gray')
    changeMenuDimensionsFrame.place(relx = 0.3, rely = 0.45, relheight = 0.15, relwidth = 0.4)

    changeMenuDimensionsButton = tk.Button(changeMenuDimensionsFrame, text = "Change Main Menu Dimensions", bg = '#E5E7E9', font = buttonFont, command = self.char
    changeMenuDimensionsButton.place(relx = 0.03, rely = 0.1, relheight = 0.8, relwidth = 0.94)

    myAccountFrame = tk.Frame(mainframe, bg = 'gray')
    myAccountFrame.place(relx = 0.3, rely = 0.65, relheight = 0.15, relwidth = 0.4)

    myAccountButton = tk.Button(myAccountFrame, text = "My Account", bg = '#E5E7E9', font = buttonFont, command = self.myAccount)
    myAccountButton.place(relx = 0.03, rely = 0.1, relheight = 0.8, relwidth = 0.94)
```

Within settings, you have three options. One to review all users (to be retrieved from the database), one to change the dimensions of the main menu, and finally one to review their own account. Since, however, the option to review users would be exclusive to admin users, I modified the main menu and the settings function slightly.

```
def adminMainMenuPage(self, username, titleFont, buttonFont, defaultFont):
```

First, instead of taking the first name of the user, I passed on their username as a parameter.

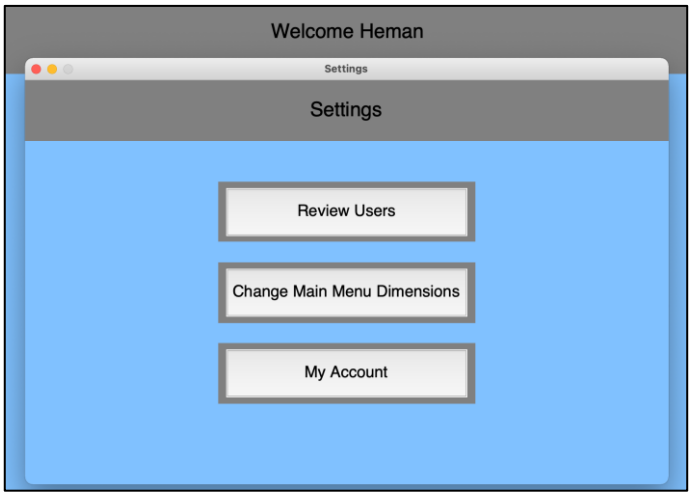
```
firstName = viewingFromDatabase.viewFromDatabase(c, False, False, 'firstName', 'username', "'"+username+"'", 'UserDetails')
welcomeLabel = tk.Label(canvas, text = "Welcome "+firstName[0][0], bg = 'gray', font = titleFont)
welcomeLabel.place(relx = 0, rely = 0, relheight = 0.15, relwidth = 1)
```

This required me to retrieve the first name of the user from the database and pass on the first value within the tuple within the list that was retrieved. This was necessary since it allowed me to pass on the username to the settings function, which could in turn allow for the function to retrieve the user type from the database. Instead of creating separate settings functions for all the user types, I decided to use the user types as the subject of an if statement to decide what appeared on the settings page and where they were placed.

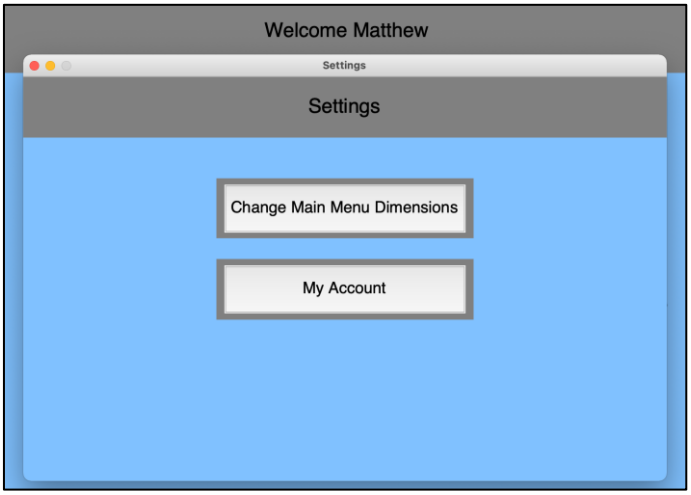
```
userType = viewingFromDatabase.viewFromDatabase(c, False, False, 'accountType', 'username', "'"+username+"'", 'UserDetails')
if userType[0][0] == 'Admin':
    reviewUsersFrame = tk.Frame(mainframe, bg = 'gray')
    reviewUsersFrame.place(relx = 0.3, rely = 0.25, relheight = 0.15, relwidth = 0.4)

    reviewUsersButton = tk.Button(reviewUsersFrame, text = "Review Users", bg = '#E5E7E9', font = buttonFont, command = self.reviewUsers)
    reviewUsersButton.place(relx = 0.03, rely = 0.1, relheight = 0.8, relwidth = 0.94)

    cmdPosition = 0.45
    maPosition = 0.65

else:
    cmdPosition = 0.25
    maPosition = 0.45

changeMenuDimensionsFrame = tk.Frame(mainframe, bg = 'gray')
changeMenuDimensionsFrame.place(relx = 0.3, rely = cmdPosition, relheight = 0.15, relwidth = 0.4)

changeMenuDimensionsButton = tk.Button(changeMenuDimensionsFrame, text = "Change Main Menu Dimensions", bg = '#E5E7E9', font = buttonFont, command = self.char
changeMenuDimensionsButton.place(relx = 0.03, rely = 0.1, relheight = 0.8, relwidth = 0.94)

myAccountFrame = tk.Frame(mainframe, bg = 'gray')
myAccountFrame.place(relx = 0.3, rely = maPosition, relheight = 0.15, relwidth = 0.4)

myAccountButton = tk.Button(myAccountFrame, text = "My Account", bg = '#E5E7E9', font = buttonFont, command = self.myAccount)
myAccountButton.place(relx = 0.03, rely = 0.1, relheight = 0.8, relwidth = 0.94)
```

Here, I retrieved the user type from the database and compared it to the value ‘Admin’. If it matched, the program then created a review users frame and button, but also set the position of the change menu dimensions frame and the my account frame accordingly. If there was no match, the else statement would then be triggered and the position of the change menu dimensions frame and the my account frame were modified so to be higher, in the place of the review users frame. By using frames in this way, I wasn’t required to mess about with the positions of any buttons, as well as providing a more elegant look.

When logged in as hemanseego01 (Admin user) as per the database:



When logged in as mattbumpus03 (Parent user):



Review Users Function:
For this function, I wanted to create a table of all the users which required reviewing and all their relevant details.
I first attempted to create a table with all the data within my database. I created a separate function for this since this could be used later on when creating the progress function.

```
def reviewUsers(self):
    print('Review Users')
    reviewUsersPage = tk.Tk()
    reviewUsersPage.title("Review Users")

    ## Retrieves the assigned width and height for the settings page
    mainScreenWidth = settings.getScreenWidth()
    mainScreenHeight = settings.getScreenHeight()

    canvas = tk.Canvas(reviewUsersPage, height = mainScreenHeight, width = mainScreenWidth)
    canvas.pack()

    mainframe = tk.Frame(reviewUsersPage, bg = '#80c1ff')
    mainframe.place(relheight = 1, relwidth = 1)

    ## Fetch all of the rows from the database
    data = viewingFromDatabase.viewFromDatabase(c, True, True, None, None, None, 'UserDetails')
    self.createTable(data, mainframe, Login.defaultFont)
```

```
def createTable(self, data, frame, defaultFont):
    # Find total number of rows and columns in data list
    totalRows = len(data)
    totalColumns = len(data[0])

    for i in range(totalRows):
        for j in range(totalColumns):
            field = tk.Label(frame, width = 15, font = defaultFont, borderwidth = 1, relief = "solid")
            field.grid(row = i, column = j)
            field.config(text = data[i][j])
```

In the reviewUsers function, I created a page, along with a canvas and mainframe. I proceeded to retrieve all the data from the database using the viewFromDatabase function in the database class by assigning both allFields and allRecords to True and the table as 'UserDetails'. The rest of the variables were irrelevant and so were filled in with None. This data was then passed along to the createTable function as well as the mainframe. I created this function to use the format provided when retrieving the database. I first determined the total number of rows (by identifying how many records there were in the data list) and the total number of columns (by identifying how many fields were within one record). I then used two for loops, the first with a range of the number of rows, and within that, a for loop with a range of the number of columns. Per column, per row, a label was created with the relevant width, font and border and was placed within the mainframe that was passed as a parameter. I created this labels in the form of a table using the grid function within tkinter, setting the row to the current row being worked on and the column the current column being worked on, attained from the value in the for loop. I then configured the text for the newly created label to read the value within data[i][j], which contained the values that were to be displayed onto the table.

Review Users

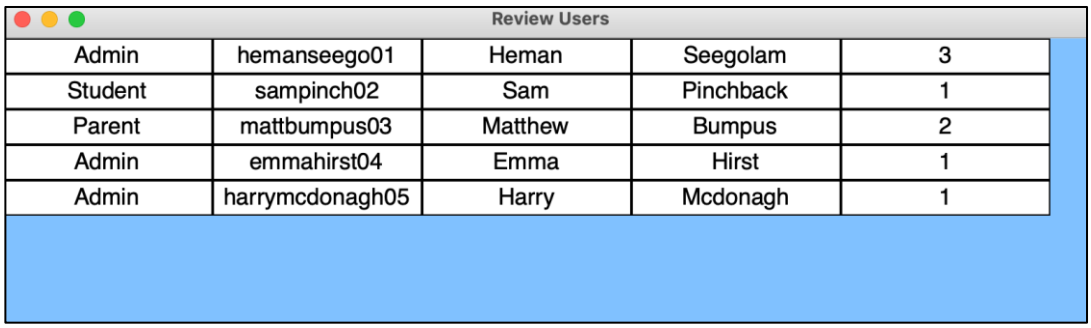
1	Admin	hemanseego01	7d0dd74d15a06739	Heman	Seegolam	3	4
2	Student	sampinch02	8839429fb39519f1d	Sam	Pinchback	1	
3	Parent	mattbumpus03	0311219d3a0c56263	Matthew	Bumpus	2	1
4	Admin	emmahirst04	b9b4b0e7c8fc46cdd	Emma	Hirst	1	
5	Admin	harrymcdonagh05	780fbb0064816e3eb	Harry	Mcdonagh	1	

I refrained from making the dimensions of this page fixed since there was a lot of hidden data.

After testing the review users and create table functions, I decided to specify which data should be collected from the database, since not all of it will be relevant to the user, nor directly understandable.
I modified the values going into my view from database function:

```
## Fetch the relevant data from the database
data = viewingFromDatabase.viewFromDatabase(c, False, True, 'accountType, username, firstName, lastName, reviewID', None, None, 'UserDetails')
self.createTable(data, mainframe, Login.defaultFont)
```

Here, I set allFields as False and instead of naming one field to retrieve data from, I listed all the relevant ones in a format that would be understandable in SQL. This produced:



This removed a lot of the unnecessary detail which could be inconvenient for the user.
However, this table still wasn't clear for its purpose; the user could not tell the cause for review. As a result, I had to first perform an inner join on the UserDetails table and the UserReview table to also show the types of review as per the review ID.
In my database class, I had not explored the option of performing an inner join on two tables. And so I did.
Within my database class:

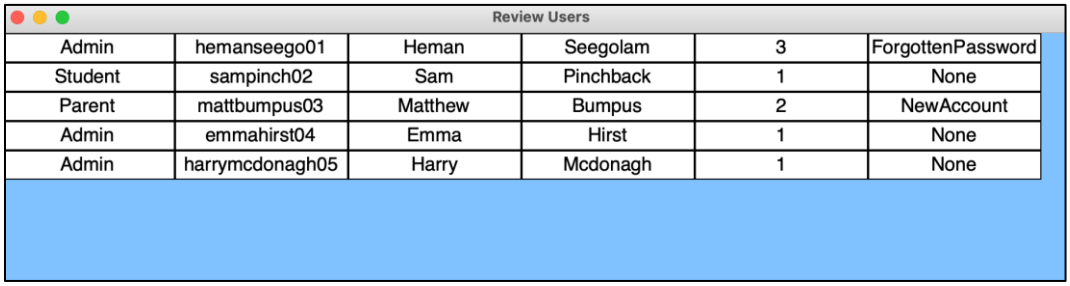
```
## Perform an inner join on two tables (combine two tables)
def innerJoin(self, c, tableFields, table1, table2, table1MatchingField, table2MatchingField):
    table = c.execute("SELECT " + tableFields + " FROM " + table1 + " INNER JOIN " + table2 + " ON " + table1MatchingField + " = " + table2MatchingField)
    newTable = table.fetchall()
    return newTable
```

```
## Fetch the relevant data from the database by performing an inner join between the UserDetails and UserReview tables
newTable = innerJoinTables.innerJoin(c, 'UserDetails.accountType, UserDetails.username, UserDetails.firstName, UserDetails.lastName, UserDetails.reviewID,
self.createTable(newTable, mainframe, Login.defaultFont)                                                    UserReview.reviewType', 'UserDetails', 'UserReview',
                                                                                                            'UserDetails.reviewID', 'UserReview.reviewID')
```

Initially, there was an error before I wrote newTable = table.fetchall() in the innerJoin function:

```
TypeError: object of type 'sqlite3.Cursor' has no len()
```

Since the table was not in a form that python could understand and could hence not obtain a length, but this was easily resolvable by implementing the code above.
This now produced:

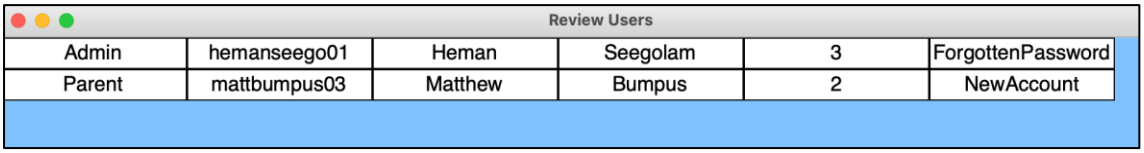


As desired.

I extended the function to seek out only the records which have a review ID of above 1.

```
newTable = innerJoinTables.innerJoin(c, 'UserDetails.ac
data = []
for each in newTable:
    if each[4] > 1:
        data.append(each)
self.createTable(data, mainframe, Login.defaultFont)
```

I compared the fifth value of each tuple within the new table list to check whether greater than one and appended all of these to a list.



A limitation with this table is that long usernames and perhaps even names may be partially hidden since the length of the cells were fixed, even when expanding out the window. However, when a user clicked on a row, It would hopefully bring them to a new page which showed all the details in full, and so would not be hidden there.

Next, I had to bind each of the labels to a function that would actually let the user change the review status of the users in the table. I did this by extending the create table function by creating a dictionary for each cell in the table:

```
cells = {}

## For each row...
for i in range(totalRows):
    ## For each column...
    for j in range(totalColumns):
        ## Create label with relevant text and place it in a grid format
        field = tk.Label(frame, width = 15, font = defaultFont, borderwi
        field.grid(row = i, column = j)
        field.config(text = data[i][j])
        ## Add cell with text to dictionary
        cells[(i,j)] = field.cget('text')

return cells
```

Within the cells dictionary, I used the i and j counters as the row and column to form the key and assigned to this key the text of the relevant cell. I then returned this cells dictionary.

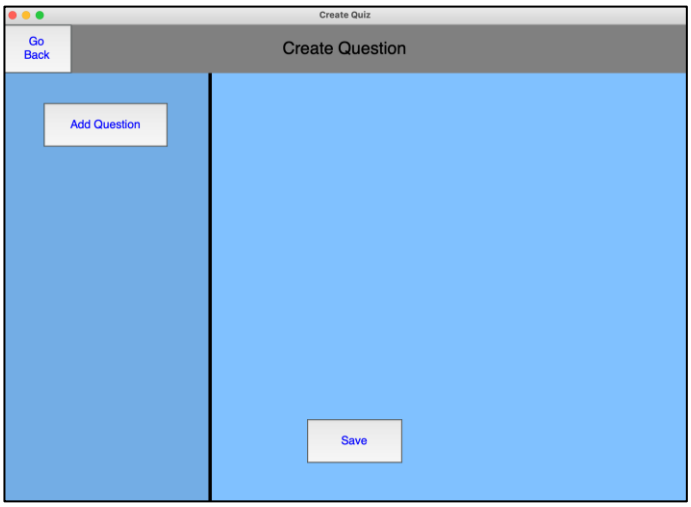
In the reviewUsers function, I therefore wrote:

```
## Create table with data from new table
cells = self.createTable(data, tableFrame, Login.defaultFont)

## Create counter for each cell
counter = 0
## Retrieve children from tableFrame and assign to list
wlist = tableFrame.winfo_children()
for each in wlist:
    ## Calculate row of the child
    rowNumber = counter // 6
    ## Bind button even to each label to print username of the selected row
    each.bind("<Button-1>", lambda x: print(cells[(rowNumber, 1)]))
    ## Add one to counter
    counter += 1
```

Here, I retrieved the cells dictionary from the createTable function. I created a for loop, which took each child of the table frame (i.e. each cell label), and bound an event. This event was meant to take the row number of the selected cell, and using that display the username associated with that cell using the value from the cells dictionary. The row number was calculated by performing an integer division of the counter by 6 (the number of columns). This failed to work however, since it appeared that each label had been bound to an event, but with the value of the final rowNumber, at the end of the loop. It did not bind an event to each label with different values of rowNumber. This meant that the username of the last row shown in the table was always displayed for any selected cell.

Creating a Quiz:

```
def createQuiz(self, username, titleFont, defaultFont, buttonFont):
    print('Create Quiz')
    ## Retrieves the assigned height, width and fonts for the main create question page
    mainScreenWidth = mainMenu.getScreenWidth()
    mainScreenHeight = mainMenu.getScreenHeight()

    mainCreateQuestionPage = tk.Tk()
    mainCreateQuestionPage.title("Create Quiz")

    canvas = tk.Canvas(mainCreateQuestionPage, height = mainScreenHeight, width = mainScreenWidth)
    canvas.pack()

    mainframe = tk.Frame(mainCreateQuestionPage, bg = '#80c1ff')
    mainframe.place(relheight = 1, relwidth = 1)

    welcomeLabel = tk.Label(mainframe, text = "Create Question", bg = 'gray', font = titleFont)
    welcomeLabel.place(relheight = 0.1, relwidth = 1)

    returnButton = tk.Button(mainframe, text = "Go\nBack", highlightbackground = '#424949', fg = 'blue', font = defaultFont, command = lambda: self.returnTo
    returnButton.place(relheight = 0.1, relwidth = 0.1)                                                                    self.returnToMainMenu(mainCreateQuestionPage, username))

    background = tk.Frame(mainCreateQuestionPage, bg = 'black')
    background.place(rely = 0.1, relheight = 0.9, relwidth = 1)

    questionsFrame = tk.Frame(mainCreateQuestionPage, bg = '#73ade5')
    questionsFrame.place(rely = 0.1, relheight = 0.9, relwidth = 0.3)

    rightFrame = tk.Frame(mainCreateQuestionPage, bg = '#80c1ff')
    rightFrame.place(relx = 0.305, rely = 0.1, relheight = 0.9, relwidth = 0.695)

    addQuestionButton = tk.Button(questionsFrame, text = "Add Question", highlightbackground = '#424949', fg = 'blue', font = defaultFont, command = lambda:
    addQuestionButton.place(relx = 0.2, rely = 0.07, relheight = 0.1, relwidth = 0.6)  self.mainCreateQuestionMenu(Login.titleFont, Login.defaultFont, Login.buttonFont))

    saveButton = tk.Button(rightFrame, text = "Save", highlightbackground = '#424949', fg = 'blue', font = defaultFont, command = lambda: self.saveQuiz(Logi
    saveButton.place(relx = 0.2, rely = 0.8, relheight = 0.1, relwidth = 0.2)                                  self.saveQuiz(Login.titleFont, Login.defaultFont, Login.buttonFont))
```



When a user presses the create quiz option from the main menu, they are brought to this page. Here, I've created two distinct frames, one to appear as a sidebar to show all the questions the user has created so far, and another on the right to preview a selected question among those created by the user. The user has the option to return to the main menu, to create a new question or to save the quiz.

Go back button:

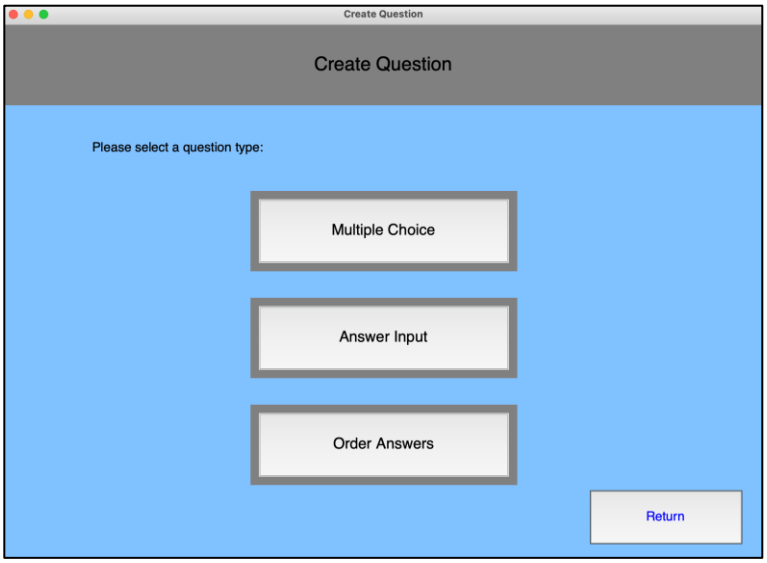
```
def returnToMainMenu(self, previousPage, username):
    previousPage.destroy()
    userType = viewingFromDatabase.viewFromDatabase(c, False, False, 'accountType', 'username', "'"+username+"'", 'UserDetails')
    if userType[0][0] == 'Admin':
        mainMenu.adminMainMenuPage(username, Login.titleFont, Login.buttonFont, Login.defaultFont)
    if userType[0][0] == 'Student':
        mainMenu.studentMainMenuPage()
    if userType[0][0] == 'Parent':
        mainMenu.parentMainMenuPage()
```

For this function, I have passed the username as a parameter from the previous page so the function can retrieve the user type of that user and open the respective main menu. Initially, with this function I experienced an error: `_tkinter.TclError: image "pyimage3" doesn't exist`
This was because all the image variables from before were conflicting with the new ones that were being made. This was rectifiable by first destroying the previous page, and then creating the new main menu page.

Main Create Question Menu Page –

```
## Creates the main create question page interface
def mainCreateQuestionMenu(self, username, mainCreateQuestionPage, titleFont, defaultFont, buttonFont):
    ## Retrieves the assigned height, width and fonts for the main create question page
    mainScreenWidth = mainMenu.getScreenWidth()
    mainScreenHeight = mainMenu.getScreenHeight()

    mainCreateQuestionMenuPage = tk.Tk()
    mainCreateQuestionMenuPage.title("Create Question")

    canvas = tk.Canvas(mainCreateQuestionMenuPage, height = mainScreenHeight, width = mainScreenWidth)
    canvas.pack()

    mainframe = tk.Frame(mainCreateQuestionMenuPage, bg = '#80c1ff')
    mainframe.place(relheight = 1, relwidth = 1)

    welcomeLabel = tk.Label(mainframe, text = "Create Question", bg = 'gray', font = titleFont)
    welcomeLabel.place(relx = 0, rely = 0, relheight = 0.15, relwidth = 1)

    selectQuestionTypeLabel = tk.Label(mainframe, text = "Please select a question type:", bg = '#80c1ff', font = defaultFont)
    selectQuestionTypeLabel.place(relx = 0.03, rely = 0.18, relheight = 0.1, relwidth = 0.4)

    multipleChoiceFrame = tk.Frame(mainframe, bg = 'gray')
    multipleChoiceFrame.place(relx = 0.325, rely = 0.31, relheight = 0.15, relwidth = 0.35)

    multipleChoiceButton = tk.Button(multipleChoiceFrame, text = "Multiple Choice", highlightbackground = '#E5E7E9', font = buttonFont,
    multipleChoiceButton.place(relx = 0.03, rely = 0.1, relheight = 0.8, relwidth = 0.94)       command = self.multipleChoiceButtonClicked)

    answerInputFrame = tk.Frame(mainframe, bg = 'gray')
    answerInputFrame.place(relx = 0.325, rely = 0.51, relheight = 0.15, relwidth = 0.35)

    answerInputButton = tk.Button(answerInputFrame, text = "Answer Input", highlightbackground = '#E5E7E9', font = buttonFont, command
    answerInputButton.place(relx = 0.03, rely = 0.1, relheight = 0.8, relwidth = 0.94)          command = self.answerInputButtonClicked)

    orderAnswersFrame = tk.Frame(mainframe, bg = 'gray')
    orderAnswersFrame.place(relx = 0.325, rely = 0.71, relheight = 0.15, relwidth = 0.35)

    orderAnswersButton = tk.Button(orderAnswersFrame, text = "Order Answers", highlightbackground = '#E5E7E9', font = buttonFont, comma
    orderAnswersButton.place(relx = 0.03, rely = 0.1, relheight = 0.8, relwidth = 0.94)       command = self.orderAnswersButtonClicked)

    returnButton = tk.Button(mainframe, text = "Return", highlightbackground = '#424949', fg = 'blue', font = defaultFont, command = ma
    returnButton.place(relx = 0.77, rely = 0.87, relheight = 0.1, relwidth = 0.2)                  mainCreateQuestionMenuPage.destroy)

    mainCreateQuestionMenuPage.mainloop()
```

```
def multipleChoiceButtonClicked(self):
    print('Multiple Choice Clicked')

def answerInputButtonClicked(self):
    print('Answer Input Clicked')

def orderAnswersButtonClicked(self):
    print('Order Answers Clicked')
```



In this function, I have created a general create question menu that follows a similar format to the main login page, and allows a user to pick between three question types: multiple choice, answer input or order answers. The return button here simply destroys the page, since the previous page was to stay open.

Creating a multiple choice question:

```
## Creates the create multiple choice question page interface
def createMultipleChoiceQuestion(self, titleFont, defaultFont, buttonFont):
    ## Retrieves the assigned height, width and fonts for the main create question page
    mainScreenWidth = mainMenu.getScreenWidth()
    mainScreenHeight = mainMenu.getScreenHeight()

    mainCreateMultipleChoiceQuestionPage = tk.Tk()
    mainCreateMultipleChoiceQuestionPage.title("Create Multiple Choice Question")

    canvas = tk.Canvas(mainCreateMultipleChoiceQuestionPage, height = mainScreenHeight, width = mainScreenWidth)
    canvas.pack()

    background = tk.Frame(mainCreateMultipleChoiceQuestionPage, bg = 'black')
    background.place(relheight = 1, relwidth = 1)

    mainframe = tk.Frame(mainCreateMultipleChoiceQuestionPage, bg = '#80c1ff')
    mainframe.place(relheight = 1, relwidth = 0.695)

    questionEntry = tk.Entry(mainframe, font = titleFont, justify = 'center')
    questionEntry.place(relx = 0.05, rely = 0.05, relheight = 0.25, relwidth = 0.9)
    questionEntry.insert('end', 'Enter a Question...')

    answerOneEntry = tk.Entry(mainframe, justify = 'center')
    answerOneEntry.place(relx = 0.15, rely = 0.4, relheight = 0.15, relwidth = 0.3)
    answerOneEntry.insert('end', 'Answer 1...')

    answerTwoEntry = tk.Entry(mainframe, justify = 'center')
    answerTwoEntry.place(relx = 0.55, rely = 0.4, relheight = 0.15, relwidth = 0.3)
    answerTwoEntry.insert('end', 'Answer 2...')

    answerOptions = 2

    createButton = tk.Button(mainframe, text = "Create", highlightbackground = '#424949', fg = 'blue', font = defaultFont, command = lambda:
    createButton.place(relx = 0.4, rely = 0.85, relheight = 0.1, relwidth = 0.2)          self.createQuestion('Multiple Choice', questionEntry.get(),
                                                                                          answerOptions, answerOneEntry.get(), answerTwoEntry.get(), timeLimitEntry.get(),
                                                                                          trackCheckbutton, trackRadiobutton))
```

From the above code, I created a create multiple choice question page, in which I created a canvas, black background, and a mainframe, similar to the format of the create quiz page, this time to the left. Within that mainframe, I created an entry for the question and the first two options, as per my design. I pre-filled these entry boxes with relevant messages: 'Enter a Question…', 'Answer 1…' and 'Answer 2…'. I placed these in the middle of the entry box using justify = 'center'. Since I will have an option to add more options, I set a variable answerOptions to 2, a variable to represent how many options the user has. This would help me later on when creating a function to add more options. I then created a create button, to finalise the creation of the question.

```
sideframe = tk.Frame(mainCreateMultipleChoiceQuestionPage, bg = '#80c1ff')
sideframe.place(relheight = 1, relwidth = 0.3, relx = 0.7)

addAnswerOptionButton = tk.Button(sideframe, text = 'Add Answer Option', highlightbackground = '#424949', fg = 'blue', font = defaultFont, command = lambda:
addAnswerOptionButton.place(relx = 0.225, rely = 0.1, relheight = 0.06, relwidth = 0.55)          self.addOption(answerOptions, mainframe))

correctAnswerLabel = tk.Label(sideframe, text = 'Correct Answer:', bg = 'gray', font = defaultFont)
correctAnswerLabel.place(relx = 0.225, rely = 0.25, relheight = 0.06, relwidth = 0.55)

trackCheckbutton = tk.IntVar() ## Creating a variable which will track the selected checkbutton
checkButtonList = [] ## Empty list which is going to hold all the checkbuttons
## Creating four checkbuttons (where only one can be selected at a time)
position = 0.22
for i in range(answerOptions):
    checkButtonList.append(tk.Checkbutton(sideframe, onvalue = i, variable = trackCheckbutton, text = i+1))
    checkButtonList[i].place(relx = position, rely = 0.35)
    position += 0.15

timeLimitLabel = tk.Label(sideframe, text = 'Time Limit:', bg = 'gray', font = defaultFont)
timeLimitLabel.place(relx = 0.225, rely = 0.5, relheight = 0.06, relwidth = 0.55)

timeLimitEntry = tk.Entry(sideframe, justify = 'center', validate = 'key', validatecommand = (sideframe.register(self.validate),'%d', '%i', '%P', '%s', '%S',
timeLimitEntry.place(relx = 0.4, rely = 0.59, relheight = 0.05, relwidth = 0.2)                                                                    '%S', '%v', '%V', '%W'))
```

Here I created a side frame to take up most of the remainder of the page. This frame is to preview all the questions that a user has created for that quiz. In it, there is a button which allows a user to add an answer option, various check buttons in which the user has to indicate the correct answer for the question they are creating, and a label and entry for the time limit for the question. To create the check buttons, I first had to create a variable that tracked which check button was selected. I did this by assigning tk.IntVar() to it. I then created a list for all check buttons, created a starting x-position for them and started a for loop. This for loop ran for the number of answer options there were, to create the correct number of check buttons. Within the for loop, I appended a new instance of a check button to the check button list and gave it an on-value of i, which indicated which loop it was on, I assigned the track variable to it and assigned the text to be i+1 (since the index started at 0). I proceeded to place that check button on the side frame at a relative y-position of 0.35, constant throughout and a relative x-position that started at 0.22 and incremented by 0.15 for every check button. For the time limit, I had to make sure that no characters or symbols were allowed to be typed into the entry. An optional parameter which I passed into the entry is validate, which specifies when validation should occur. I have selected 'key', which will cause the entry to be validated whenever the user types something inside it. I also created a separate validate function, which was prompted under validatecommand.

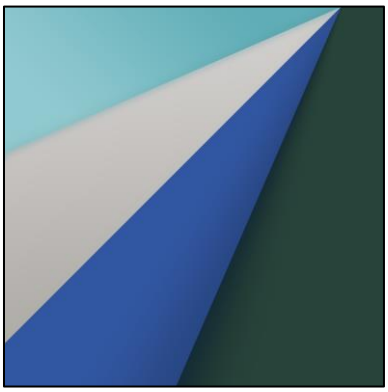
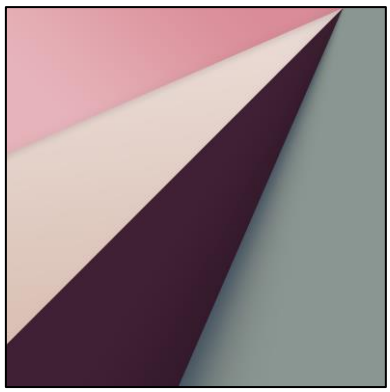
```
def validate(self, action, index, valueIfAllowed, priorValue, text, validationType, triggerType, widgetName):
    if valueIfAllowed:
        try:
            int(valueIfAllowed)
            return True
        except:
            return False
    else:
        return False
```

The function used for this command must return True if the entry's value is allowed to change and False otherwise, and it must be wrapped using a widget's register method. The validate method checks that the contents of the entry field are a valid integer: whenever the user types something inside the field, the contents will only change if the new value is a valid number.
For this to work, I had to use substitution codes exclusive to validatecommand.
'%d' - action code: function must receive a 0 for an attempted deletion, and a 1 for an attempted insertion.
'%i' - when the user attempts to insert or delete text, this argument will be the index of the beginning of the insertion or deletion.
'%P' - The value that the text will have if the change is allowed.
'%s' - The text in the entry before the change.
'%S' - If the call was due to an insertion or deletion, this argument will be the text being inserted or deleted.
'%v' - The current value of the widget's validate option.
'%V' - The reason for this callback, i.e. 'key' in this instance.
'%W' - The name of the widget.
These were all required to provide an effective validation.

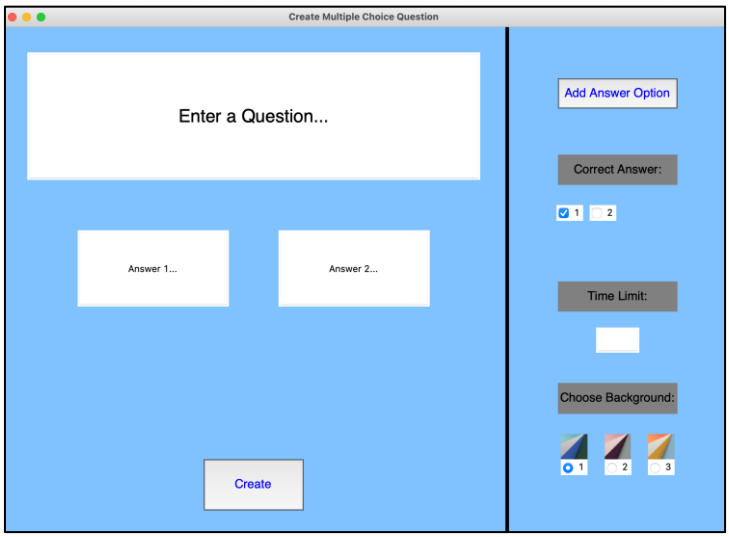
```
selectBackgroundLabel = tk.Label(sideframe, text = "Choose Background:", bg = 'gray', font = defaultFont)
selectBackgroundLabel.place(relx = 0.225, rely = 0.7, relheight = 0.06, relwidth = 0.55)

backgroundOneImage = Image.open('BackgroundOne.jpg')
backgroundOneImage = backgroundOneImage.resize((40,40))
tkBackgroundOneImage = ImageTk.PhotoImage(backgroundOneImage)
backgroundOneImageLabel = tk.Label(sideframe, image = tkBackgroundOneImage)
backgroundOneImageLabel.place(relx = 0.24, rely = 0.8, relheight = 0.05, relwidth = 0.12)
```

```
backgroundTwoImage = Image.open('BackgroundTwo.jpg')
backgroundTwoImage = backgroundTwoImage.resize((40,40))
tkBackgroundTwoImage = ImageTk.PhotoImage(backgroundTwoImage)
backgroundTwoImageLabel = tk.Label(sideframe, image = tkBackgroundTwoImage)
backgroundTwoImageLabel.place(relx = 0.44, rely = 0.8, relheight = 0.05, relwidth = 0.12)

backgroundThreeImage = Image.open('BackgroundThree.jpg')
backgroundThreeImage = backgroundThreeImage.resize((40,40))
tkBackgroundThreeImage = ImageTk.PhotoImage(backgroundThreeImage)
backgroundThreeImageLabel = tk.Label(sideframe, image = tkBackgroundThreeImage)
backgroundThreeImageLabel.place(relx = 0.64, rely = 0.8, relheight = 0.05, relwidth = 0.12)

trackRadiobutton = tk.IntVar() ## Creating a variable which will track the selected radiobutton
radioButtonList = [] ## Empty list which is going to hold all the radiobuttons
## Creating three radiobuttons (where only one can be selected at a time)
position = 0.24
for j in range(3):
    radioButtonList.append(tk.Radiobutton(sideframe, value = j, variable = trackRadiobutton, text = j+1))
    radioButtonList[j].place(relx = position, rely = 0.85)
    position += 0.2

mainCreateMultipleChoiceQuestionPage.mainloop()
```

I then went on to create a label to select a background. I imported the three relevant images, resized them, converted them to a tkinter format and finally placed them in the correct places as labels. The images used were:

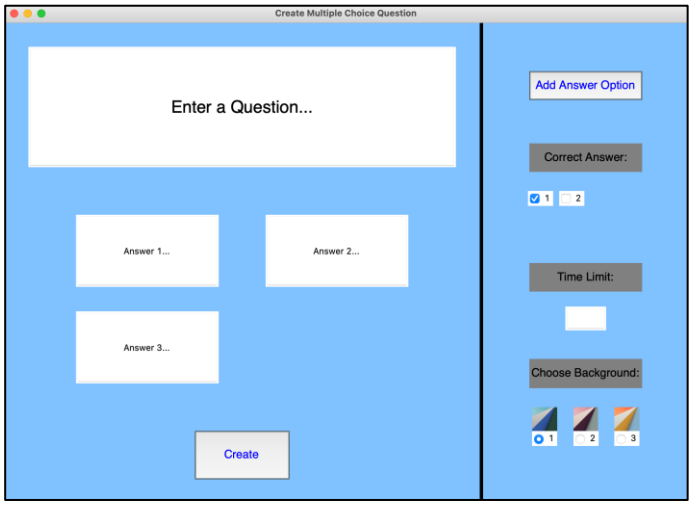

These images will also be used as the background of the questions when users are playing a quiz. Finally, I created radio buttons in a similar way to the check buttons and placed them appropriately. This produced:



To add an option:

```
def addOption(self, answerOptions, mainframe):
    if answerOptions == 2:
        answerThreeEntry = tk.Entry(mainframe, justify = 'center')
        answerThreeEntry.place(relx = 0.15, rely = 0.6, relheight = 0.15, relwidth = 0.3)
        answerThreeEntry.insert('end', 'Answer 3...')
        answerOptions += 1
    elif answerOptions == 3:
        answerFourEntry = tk.Entry(mainframe, justify = 'center')
        answerFourEntry.place(relx = 0.55, rely = 0.6, relheight = 0.15, relwidth = 0.3)
        answerFourEntry.insert('end', 'Answer 4...')
        answerOptions += 1
```

On one click, it produced:



However, upon a further click, no result was obtained. This was because the changes made to the answerOptions variable made in this function was not passed on to the original function.
First I tried to return answerOptions and start the function using

```
command = lambda: answerOptions = self.addOption(
```

However, python could not recognise this format and produced an invalid syntax error.

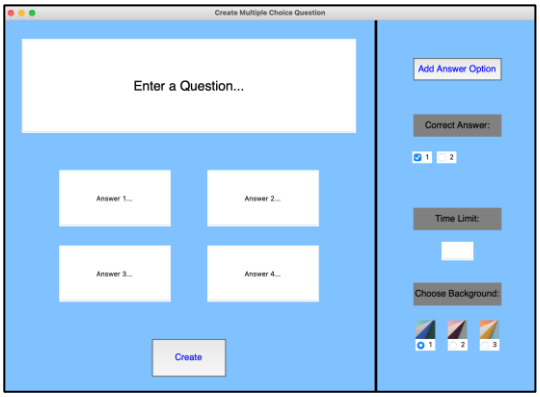
I hence decided to take a similar approach as in my login function, where I created the variable outside the function, within the class, so as to allow it to undergo change from another function.
This firstly meant I had to change the for loop for my check buttons:

```
for i in range(self.answerOptions):
```

I then modified the adoption function:

```
def addOption(self, mainframe):
    if self.answerOptions == 2:
        answerThreeEntry = tk.Entry(mainframe, justify = 'center')
        answerThreeEntry.place(relx = 0.15, rely = 0.6, relheight = 0.15, relwidth = 0.3)
        answerThreeEntry.insert('end', 'Answer 3...')
        self.answerOptions += 1
    elif self.answerOptions == 3:
        answerFourEntry = tk.Entry(mainframe, justify = 'center')
        answerFourEntry.place(relx = 0.55, rely = 0.6, relheight = 0.15, relwidth = 0.3)
        answerFourEntry.insert('end', 'Answer 4...')
        self.answerOptions += 1
```

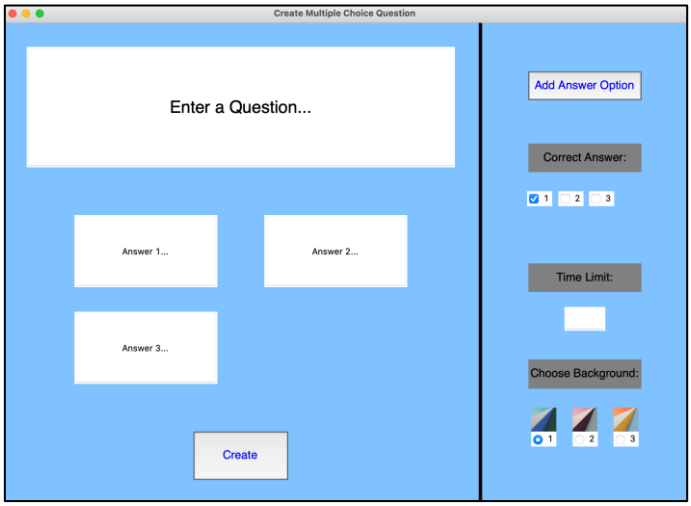
This produced the desired result:

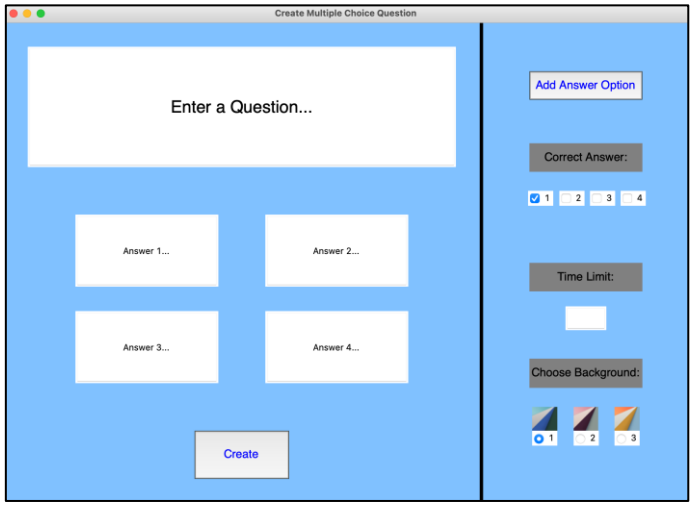


However, the check buttons didn't change with the options. I therefore modified it further:

```
def addOption(self, sideframe, mainframe, checkButtonList, trackCheckbutton):
    if self.answerOptions == 2:
        answerThreeEntry = tk.Entry(mainframe, justify = 'center')
        answerThreeEntry.place(relx = 0.15, rely = 0.6, relheight = 0.15, relwidth = 0.3)
        answerThreeEntry.insert('end', 'Answer 3...')
        checkButtonList.append(tk.Checkbutton(sideframe, onvalue = 2, variable = trackCheckbutton, text = 3))
        checkButtonList[2].place(relx = 0.52, rely = 0.35)
        self.answerOptions += 1
    elif self.answerOptions == 3:
        answerFourEntry = tk.Entry(mainframe, justify = 'center')
        answerFourEntry.place(relx = 0.55, rely = 0.6, relheight = 0.15, relwidth = 0.3)
        answerFourEntry.insert('end', 'Answer 4...')
        checkButtonList.append(tk.Checkbutton(sideframe, onvalue = 3, variable = trackCheckbutton, text = 4))
        checkButtonList[3].place(relx = 0.67, rely = 0.35)
        self.answerOptions += 1
```

Which now produced the desired result:





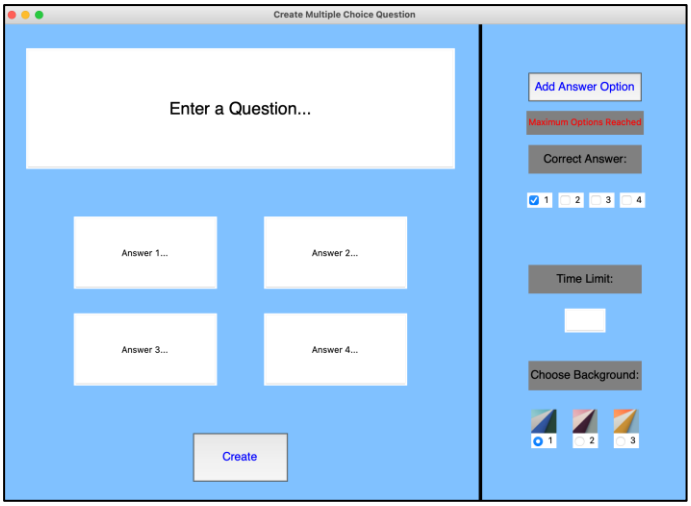
After four options have been created, the add answer option button doesn't result in anything. To let the user know that the maximum has been reached, I created an error label. Within my original function I wrote:

```
fourOptionsLabel = tk.Label(sideframe)
```

And extended the addOptions function:

```
elif self.answerOptions == 4:
    fourOptionsLabel.config(bg = 'gray', fg = 'red', text = 'Maximum Options Reached')
    fourOptionsLabel.place(relx = 0.215, rely = 0.18, relheight = 0.05, relwidth = 0.57)
```

By passing the fourOptionsLabel as a parameter to the function.
This now produced:



After three clicks.

By creating the answerOptions label outside of the function, this meant that it would not be reset and would have to be done manually after a question has finished being created. This will be shown later when I code the create question function.

Create question function:

```
def createQuestion(self, questionType, question, answerOptions, answerOne, answerTwo, timeLimit, trackCheckbutton, trackRadiobutton):
    print('Create Question')
    answers = [answerOne, answerTwo]
    correctAnswer = trackCheckbutton.get() + 1
    background = trackRadiobutton.get() + 1
    print(questionType, question, answers, timeLimit, correctAnswer, background)
```

When creating this function, I first printed 'Create Question' to test the link between the button and the function. I then retrieved all the relevant details from the page and displayed them for testing purposes. I had great difficulty producing the desired result, however, when the user added more options. This was because those variables were created from a different function and couldn't be retrieved from the original function. This meant that the third and fourth option could not be passed on to this function. I hence settled with this method:

```
answerThreeEntry = tk.Entry(mainframe, justify = 'center')
answerFourEntry = tk.Entry(mainframe, justify = 'center')
```

I created the variables for the third and fourth option within the createMultipleChoiceQuestion function, and went on to modify and place them in the addOption function, as with the check buttons:

```
for i in range(4):
    checkButtonList.append(tk.Checkbutton(sideframe, onvalue = i, variable = trackCheckbutton, text = i+1))
checkButtonList[0].place(relx = 0.22, rely = 0.35)
checkButtonList[1].place(relx = 0.37, rely = 0.35)
```

Here I created all four check buttons but only placed the first two.

```
def addOption(self, answerThreeEntry, answerFourEntry, fourOptionsLabel, checkButtonList, sideframe, mainframe):
    if self.answerOptions == 2:
        answerThreeEntry.place(relx = 0.15, rely = 0.6, relheight = 0.15, relwidth = 0.3)
        answerThreeEntry.insert('end', 'Answer 3...')
        checkButtonList[2].place(relx = 0.52, rely = 0.35)
        self.answerOptions += 1
    elif self.answerOptions == 3:
        answerFourEntry.place(relx = 0.55, rely = 0.6, relheight = 0.15, relwidth = 0.3)
        answerFourEntry.insert('end', 'Answer 4...')
        checkButtonList[3].place(relx = 0.67, rely = 0.35)
        self.answerOptions += 1
    elif self.answerOptions == 4:
        fourOptionsLabel.config(bg = 'gray', fg = 'red', text = 'Maximum Options Reached')
        fourOptionsLabel.place(relx = 0.215, rely = 0.18, relheight = 0.05, relwidth = 0.57)
```

I then modified the adoption function with all the relevant parameters to only place and configure the relevant entries and check buttons. This now allowed me to obtain all the user inputs from the page in my create question function. First I created an empty answers list which would contain all the answers of the question, outside of my function:

```
answers =[]
```

```
def createQuestion(self, questionType, question, answerOne, answerTwo, answerThree, answerFour, timeLimit, trackCheckbutton, trackRadiobutton):
    print('Create Question')
    self.answers = [answerOne.get(), answerTwo.get()]
    if self.answerOptions > 2:
        self.answers.append(answerThree.get())
    if self.answerOptions == 4:
        self.answers.append(answerFour.get())
    correctAnswer = trackCheckbutton.get() + 1
    background = trackRadiobutton.get() + 1
    print(questionType, question, self.answers, timeLimit, correctAnswer, background)
```

In my create question function, now all of the user's inputs were now accounted for and displayed. To retrieve the correct answer and the selected background, I used the .get() function, and also added one due to the index. I also created the answer list to contain the values of the first two answer options (since they would always be included), and then depending on the value of the answerOptions variable, I appended the value of the third option and the fourth option.

For the time limit, I wanted it to be between 1 and 60 seconds. To do this, I created a validateTimeLimit function:

```
def validateTimeLimit(self, questionType, question, answerOne, answerTwo, answerThree, answerFour, timeLimit, trackCheckbutton, trackRadiobutton, maxTimeLimitLabel):
    if 0 < int(timeLimit) <= 60:
        print('Valid')
        self.createQuestion(questionType, question, answerOne, answerTwo, answerThree, answerFour, timeLimit, trackCheckbutton, trackRadiobutton)
    else:
        print('Invalid')
        maxTimeLimitLabel.config(bg = 'gray', fg = 'red', text = 'Maximum Time Limit is 60s')
        maxTimeLimitLabel.place(relx = 0.2, rely = 0.66, relheight = 0.05, relwidth = 0.6)
```

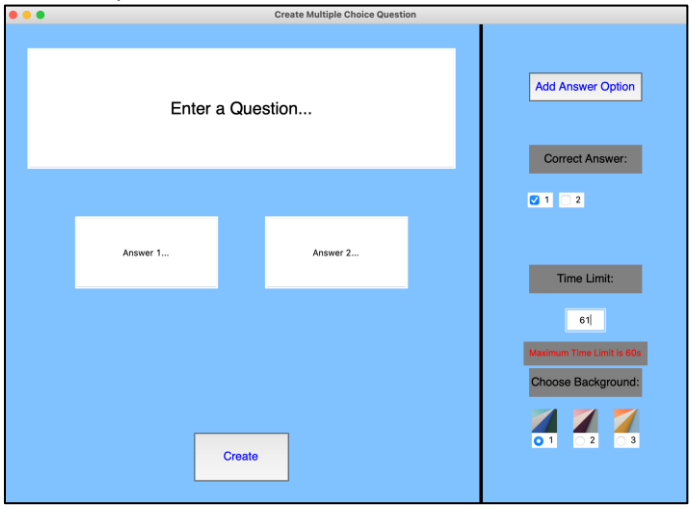
If the time limit is between 1 and 60, the function then implements the create question function. If it isn't, an error label appears.
I created the error label in the original function:

```
maxTimeLimitLabel = tk.Label(sideframe)
```

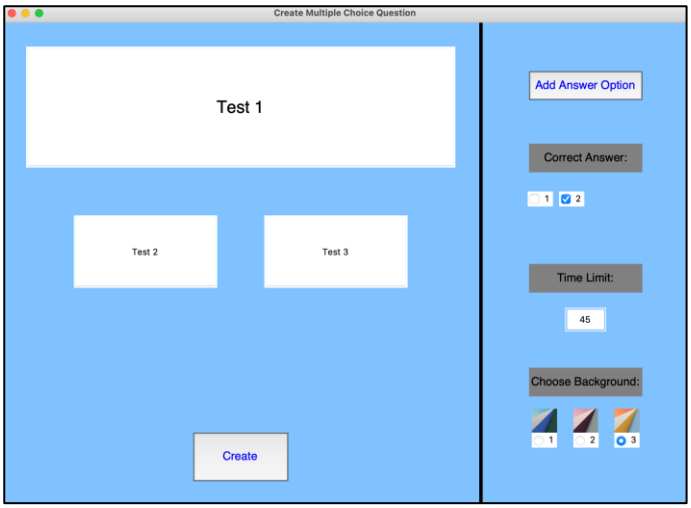
I had to move down the select background label to allow enough space for this one.
I also set the input of the time limit entry to be 30 so to avoid a blank response and any consequent errors:

```
timeLimitEntry.insert('end', 30)
```

This now produced:




```
Invalid
```




```
Valid
Create Question
Multiple Choice Test 1 ['Test 2', 'Test 3'] 45 2 3
```

When trying this function with the other functions, i.e. the admin main menu function and the create quiz function, I experienced an error which I didn't find when individually testing this function:

```
_tkinter.TclError: image "pyimage8" doesn't exist
```

When I experienced this error earlier when returning to the main menu, I was able to combat it by destroying the previous pages. However, I didn't wish to do that here since I wanted the user to be able to see both the create quiz and create question pages at the same time. I therefore made a modification on how I created the window:

```
mainCreateMultipleChoiceQuestionPage = tk.Toplevel()
```

Here, I replaced tk.Tk() with tk.Toplevel(), which indicated to tkinter that two windows would open at once, and hence prevented any conflicts with new images being imported and created.

Although I wished to keep the create quiz page and create multiple choice question page up at the same time, I had no need to keep the create question menu page up, and so I destroyed it upon opening the create multiple choice question function:

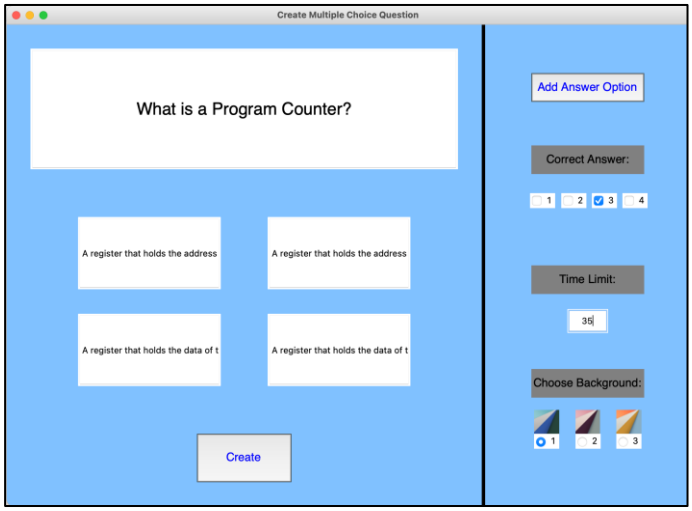
```
def createMultipleChoiceQuestion(self, previousPage, titleFont, defaultFont, buttonFont):
    previousPage.destroy()
```

To finalise the create question function, I created a list for all questions and associated details within the class:

```
questions = []
```

To test it:

```
def createQuestion(self, questionType, question, answerOne, answerTwo, answerThree, answerFour, timeLimit, trackCheckbutton, trackRadiobutton):
    self.answers = [answerOne.get(), answerTwo.get()]
    if self.answerOptions > 2:
        self.answers.append(answerThree.get())
    if self.answerOptions == 4:
        self.answers.append(answerFour.get())
    print(self.answers)
    correctAnswer = trackCheckbutton.get() + 1
    background = trackRadiobutton.get() + 1
    self.questions.append([question, self.answers, correctAnswer, timeLimit, background])
    print(self.questions)
```

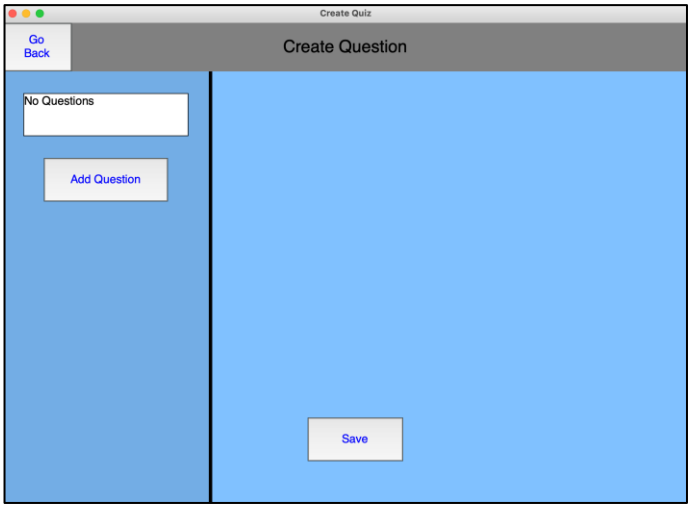


The output produced first displayed whether the time limit was valid, the answers list and finally the questions list. When taking this test, I noticed a limitation in the design: when a user types into the entry boxes, if the text exceeds the width of the box, it will not wrap around but instead lose sight of the user. The user is still able, however, to access all the text they input. I shall take into account the length of questions and options when creating the play quiz function.

```
Valid
['A register that holds the address of the current instruction', 'A register that
holds the address of the next instruction', 'A register that holds the data of the
current instruction', 'A register that holds the data of the next instruction']
[['What is a Program Counter?', ['A register that holds the address of the current
instruction', 'A register that holds the address of the next instruction', 'A regi
ster that holds the data of the current instruction', 'A register that holds the d
ata of the next instruction'], 3, '35', 1]]
```

I next wanted the create question function to configure the create quiz window, so as to add the questions in the questions frame and show a preview of the question in the right frame. To do this, I first created a list box within the create quiz page, initially showing 'No Questions'.

```
questionListbox = tk.Listbox(questionsFrame, font = defaultFont)
questionListbox.place(relx = 0.1, rely = 0.05, relheight = 0.1, relwidth = 0.8)
questionListbox.insert('end', 'No Questions')
```



In order to configure and update the list box and the add question button, I first considered passing these as parameters through my functions to the create question function. However, when considering how to always update the list box and position of the button in an efficient way, I instead passed only the page name and the username to the create question function. This allowed me to destroy the old create question page, and recreate a new one with an updated list box.

To make this work, I first changed the questions list:

```
questions = [['No Questions']]
```

I then changed the way I created the list box (and the add question button):

```
questionListbox = tk.Listbox(questionsFrame, font = defaultFont)
listboxHeight = 0
buttonPos = 0.1
for each in self.questions:
    questionListbox.insert('end', each[0])
    listboxHeight += 0.1
    buttonPos += 0.1
questionListbox.place(relx = 0.1, rely = 0.05, relheight = listboxHeight, relwidth = 0.8)

addQuestionButton = tk.Button(questionsFrame, text = "Add Question", highlightbackground =
addQuestionButton.place(relx = 0.2, rely = buttonPos, relheight = 0.1, relwidth = 0.6)
```

Here, I make the list box take its inputs from the questions list. I also created a height variable and a position variable for the list box and add question button respectively so that the box increases in size with every question added and the button moves down as the box enlarges.
I soon realised that as a quiz may exceed a certain number of questions, the add question button will continue to move down, and eventually off screen. To combat this:

```
if buttonPos < 0.8:
    listboxHeight += 0.1
    buttonPos += 0.1
```

This required the position of the button to be less than 0.8 for the height of the list box and the position of the button to increase.

I then extended the create question function to read:

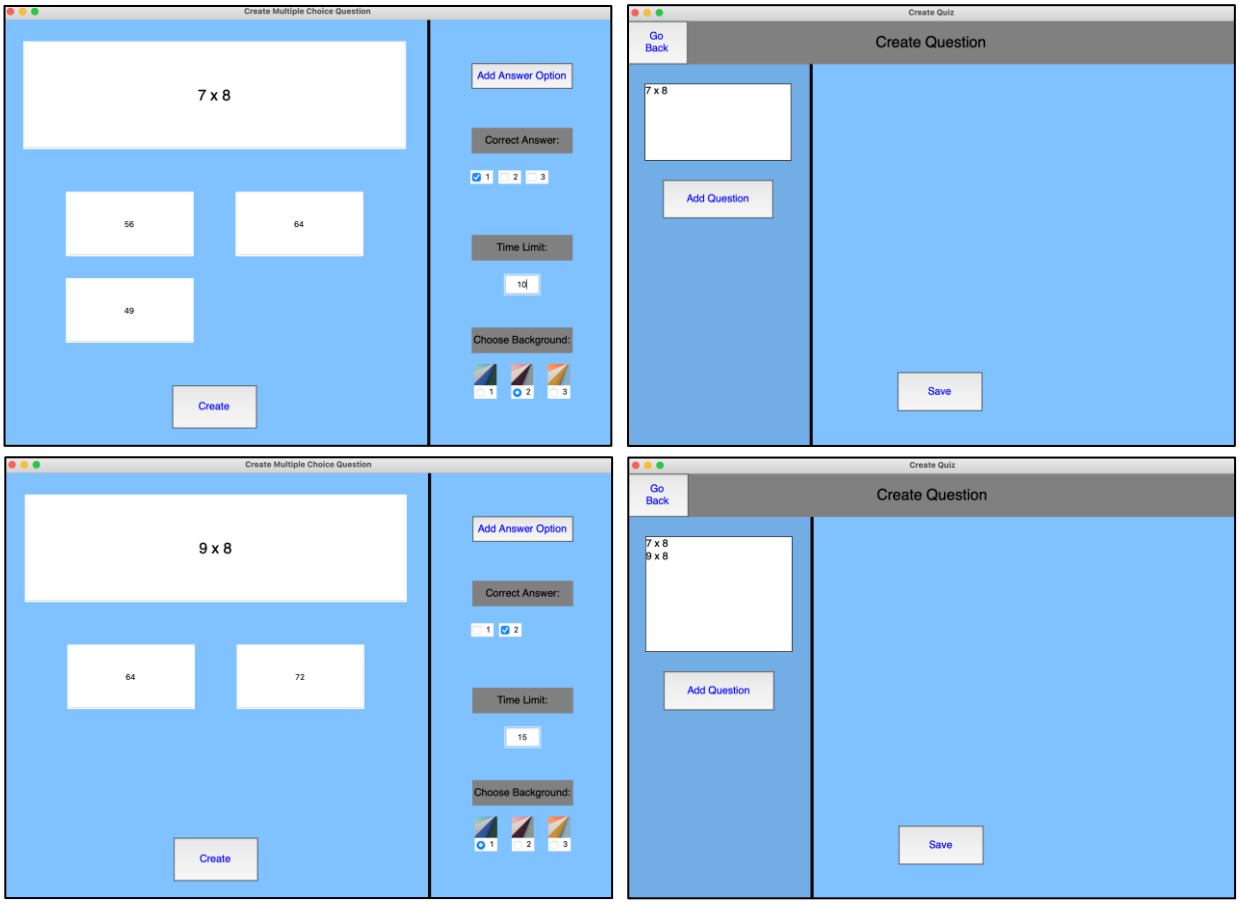
```
previousPage.destroy()
mainCreateQuestionPage.destroy()
self.createQuiz(username, Login.titleFont, Login.defaultFont, Login.buttonFont)
```

This destroyed the create multiple choice question page, the main create question page and then recreated it using the new values of the questions list for the list box.
However, this meant that 'No Questions' would be shown within the list box.

So I modified the list box code further:

```
questionListbox = tk.Listbox(questionsFrame, font = defaultFont)
listboxHeight = 0
buttonPos = 0.1
for each in self.questions:
    if len(self.questions) == 1:
        questionListbox.insert('end', each[0])
    else:
        if each != ['No Questions']:
            questionListbox.insert('end', each[0])
    if buttonPos < 0.8:
        listboxHeight += 0.1
        buttonPos += 0.1
questionListbox.place(relx = 0.1, rely = 0.05, relheight = listboxHeight, relwidth = 0.8)
```

Which produced the desired result.



The program seemed to be working thus far, however, upon creating one further question, and adding an option, it only created the fourth option, and upon another question, it declared that the maximum options had been reached when only two options were present. This was because I hadn't reset the answerOptions counter when creating a question. And so I did:

```
self.answerOptions = 2
self.answers = []
```

I also reset the answers list, since it may interfere later when creating other question types.
It now worked as desired.

Within my create quiz page, I wanted a preview of the selected question within the list box to be displayed in the right frame. To do this:

```
questionLabel = tk.Label(rightFrame)
answerOneLabel = tk.Label(rightFrame)
answerTwoLabel = tk.Label(rightFrame)
answerThreeLabel = tk.Label(rightFrame)
answerFourLabel = tk.Label(rightFrame)
```

I created these variables within the create quiz function.

And I created a bind on the list box:

```
questionListbox.bind('<<ListboxSelect>>', lambda x: self.displayQuestion(questionLabel, questionListbox.get(questionListbox.curselection()),
answerOneLabel, answerTwoLabel, answerThreeLabel, answerFourLabel, Login.titleFont, Login.defaultFont, Login.buttonFont))
```

This bind allowed the program to carry out the displayQuestion function on every selection in the list box. The displayQuestion function looked as follows:

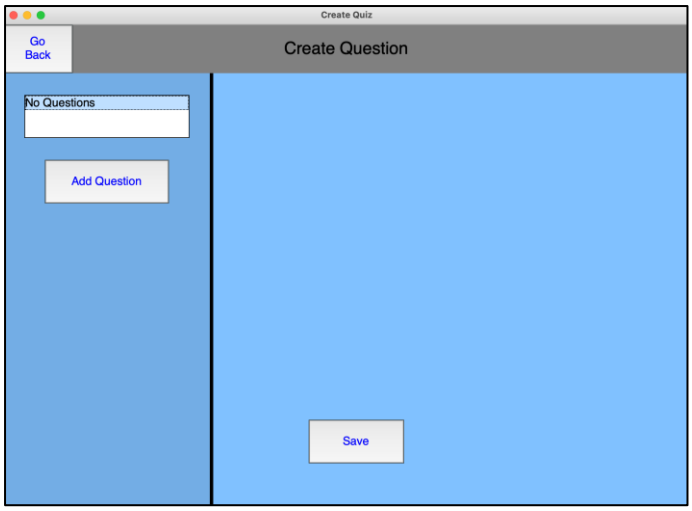
```
def displayQuestion(self, questionLabel, selectedQuestion, answerOneLabel, answerTwoLabel, answerThreeLabel, answerFourLabel, titleFont, defaultFont, buttonFont):
    for each in self.questions:
        print(each)
        if each[0] == selectedQuestion:
            try:
                answers = each[1]
            except:
                answers = []
    if len(answers) > 0:
        questionLabel.config(text = selectedQuestion, bg = 'gray', font = titleFont)
        questionLabel.place(relx = 0.05, rely = 0.05, relheight = 0.25, relwidth = 0.9)
        answerOneLabel.config(text = answers[0], bg = 'gray')
        answerOneLabel.place(relx = 0.15, rely = 0.35, relheight = 0.15, relwidth = 0.3)
    if len(answers) > 1:
        answerTwoLabel.config(text = answers[1], bg = 'gray')
        answerTwoLabel.place(relx = 0.55, rely = 0.35, relheight = 0.15, relwidth = 0.3)
    if len(answers) > 2:
        answerThreeLabel.config(text = answers[2], bg = 'gray')
        answerThreeLabel.place(relx = 0.15, rely = 0.55, relheight = 0.15, relwidth = 0.3)
    if len(answers) > 3:
        answerFourLabel.config(text = answers[3], bg = 'gray')
        answerFourLabel.place(relx = 0.55, rely = 0.55, relheight = 0.15, relwidth = 0.3)
```

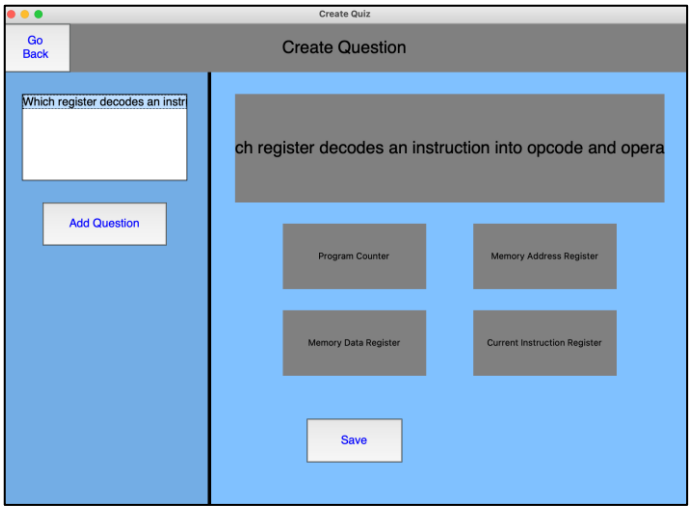
In this function, the selection of the list box is run against all of the first items of every list in the questions list. Where there is a match, it takes the answers from that list and assigns it to the variable answers. Here, I have used a try except statement which first tries to assign each[1] (i.e. the answers) to answers, but resorts to assigning an empty list to answers if it fails. This is because, on first testing, this error was produced:

```
    answers = each[1]
IndexError: list index out of range
```

This was because the first list within the questions list was the 'No Questions' list and so did not have any corresponding answers.

The function then goes on to check the length of the answers list and accordingly displays the question and first option if greater than 0, and then the rest of the options accordingly. I placed the question label configuration and placement within this if statement so nothing would be shown when 'No Questions' was selected.





I proceeded to extend the function to show the correct answer, the time limit and the chosen background.

For the background:

```
backgroundImageLabel = tk.Label(rightFrame)
```

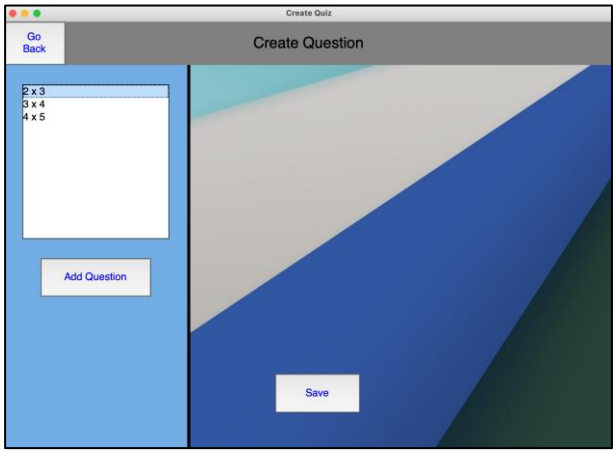
I first created a background image label in the create quiz function and passed it on as a parameter to the display question function.
I then extended the display question function:

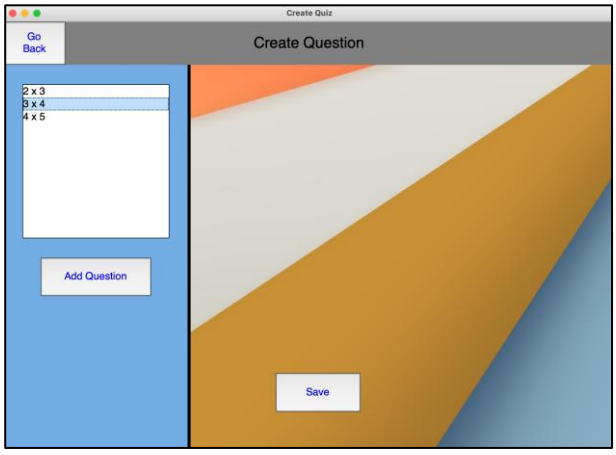
```
for each in self.questions:
    if each[0] == selectedQuestion:
        try:
            answers = each[1]
            correctAnswer = each[2]
            timeLimit = each[3]
            background = each[4]
        except:
            answers = []
            correctAnswer = ''
            timeLimit = ''
            background = ''

if background == 1:
    backgroundOneImage = Image.open('BackgroundOne.jpg')
    backgroundOneImage = backgroundOneImage.resize((1500,1000))
    tkBackgroundOneImage = ImageTk.PhotoImage(backgroundOneImage)
    backgroundImageLabel.configure(image = tkBackgroundOneImage)
    backgroundImageLabel.image = tkBackgroundOneImage
    backgroundImageLabel.place(relx = 0, rely = 0, relheight = 1, relwidth = 1)
elif background == 2:
    backgroundTwoImage = Image.open('BackgroundTwo.jpg')
    backgroundTwoImage = backgroundTwoImage.resize((1500,1000))
    tkBackgroundTwoImage = ImageTk.PhotoImage(backgroundTwoImage)
    backgroundImageLabel.configure(image = tkBackgroundTwoImage)
    backgroundImageLabel.image = tkBackgroundTwoImage
    backgroundImageLabel.place(relx = 0, rely = 0, relheight = 1, relwidth = 1)
elif background == 3:
    backgroundThreeImage = Image.open('BackgroundThree.jpg')
    backgroundThreeImage = backgroundThreeImage.resize((1500,1000))
    tkBackgroundThreeImage = ImageTk.PhotoImage(backgroundThreeImage)
    backgroundImageLabel.configure(image = tkBackgroundThreeImage)
    backgroundImageLabel.image = tkBackgroundThreeImage
    backgroundImageLabel.place(relx = 0, rely = 0, relheight = 1, relwidth = 1)
```

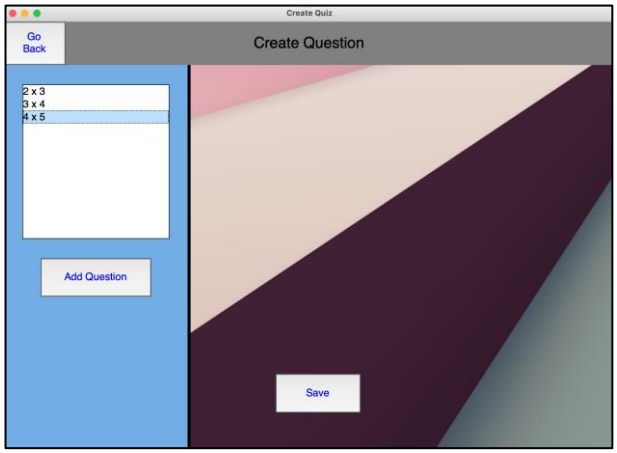
Here, I attempted to also retrieve a question's correct answer, time limit and chosen background. I then created an if statement to check whether the background was one of 1, 2 or 3 and imported the relevant image and configured the label accordingly. I resized it to a size of 1500 x 1000 so that even when the window was enlarged, the image covered the entirety of the right frame.

When testing this, I came across a limitation to this method, where for the sake of quick testing, I left the question as 'Enter a Question…' and also the answers. This now meant that multiple questions had the same question, but perhaps with different backgrounds. This would then only show the background of the first instance of that question within the questions list. However, since it is very unlikely that a quiz would have the same two questions, it is safe to leave it as so.

This now produced:







Although the backgrounds now worked as desired, there was an issue where the question and answer labels were now hidden under the background image label.

To combat this, I created frames for each of the labels that needed to appear on the forefront.

```
backgroundImageLabel = tk.Label(rightFrame)
questionFrame = tk.Frame(rightFrame)
questionLabel = tk.Label(questionFrame)
answerOneFrame = tk.Frame(rightFrame)
answerOneLabel = tk.Label(answerOneFrame)
answerTwoFrame = tk.Frame(rightFrame)
answerTwoLabel = tk.Label(answerTwoFrame)
answerThreeFrame = tk.Frame(rightFrame)
answerThreeLabel = tk.Label(answerThreeFrame)
answerFourFrame = tk.Frame(rightFrame)
answerFourLabel = tk.Label(answerFourFrame)
```

Passing these as parameters to the display question function, I hence modified it to:

```
if len(answers) > 0:
    questionLabel.config(text = selectedQuestion, bg = 'gray', font = titleFont)
    questionFrame.place(relx = 0.05, rely = 0.05, relheight = 0.25, relwidth = 0.9)
    questionLabel.place(relx = 0, rely = 0, relheight = 1, relwidth = 1)
    answerOneLabel.config(text = answers[0], bg = 'gray')
    answerOneFrame.place(relx = 0.15, rely = 0.35, relheight = 0.15, relwidth = 0.3)
    answerOneLabel.place(relx = 0, rely = 0, relheight = 1, relwidth = 1)
if len(answers) > 1:
    answerTwoLabel.config(text = answers[1], bg = 'gray')
    answerTwoFrame.place(relx = 0.55, rely = 0.35, relheight = 0.15, relwidth = 0.3)
    answerTwoLabel.place(relx = 0, rely = 0, relheight = 1, relwidth = 1)
if len(answers) > 2:
    answerThreeLabel.config(text = answers[2], bg = 'gray')
    answerThreeFrame.place(relx = 0.15, rely = 0.55, relheight = 0.15, relwidth = 0.3)
    answerThreeLabel.place(relx = 0, rely = 0, relheight = 1, relwidth = 1)
if len(answers) > 3:
    answerFourLabel.config(text = answers[3], bg = 'gray')
    answerFourFrame.place(relx = 0.55, rely = 0.55, relheight = 0.15, relwidth = 0.3)
    answerFourLabel.place(relx = 0, rely = 0, relheight = 1, relwidth = 1)
```

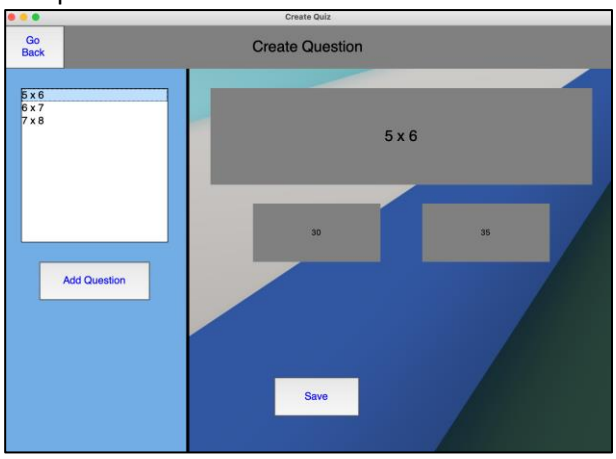
This modified code took the old placements of the labels for the frames and filled the frames with the labels.

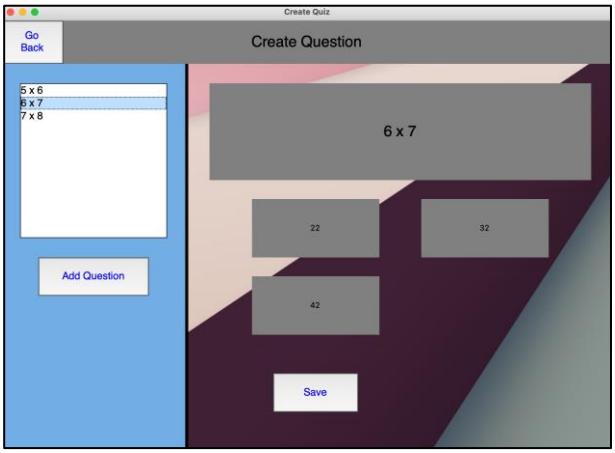
Whilst this appeared to work, another setback started to prevail. When first going on a question that has more than two options, and then going back to one that has less, the extra options seemed not to disappear. This made sense since I coded in an incremental fashion to build up to a final solution. This meant that before any (new) configuration was to occur, all labels and frames had to be reset. Although there wasn't away for me to reset them necessarily, I was able to simply hide all answer labels.

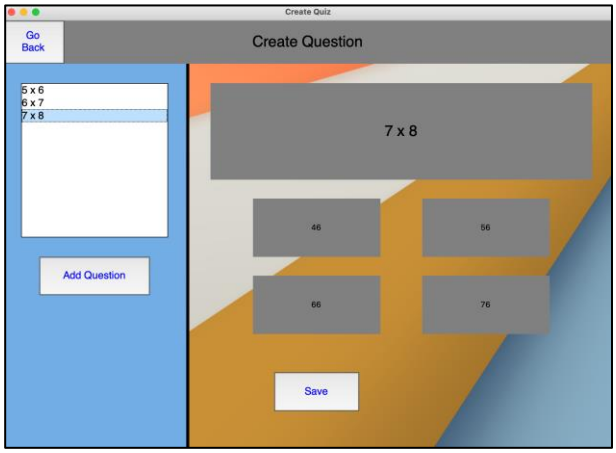
```
answerOneFrame.place_forget()
answerTwoFrame.place_forget()
answerThreeFrame.place_forget()
answerFourFrame.place_forget()
```

The function then called the relevant labels to show later on with the above code, dependant on how many answer options there were.

This produced:







This now allowed me to move between questions, in any order, to show the correct background, question and answer options.

I now finalised these functions by displaying the correct answer and the time limit.

For the time limit:

```
timeLimitFrame = tk.Frame(rightFrame)
timeLimitLabel = tk.Label(timeLimitFrame)
```

I created a frame and label in the create quiz function.

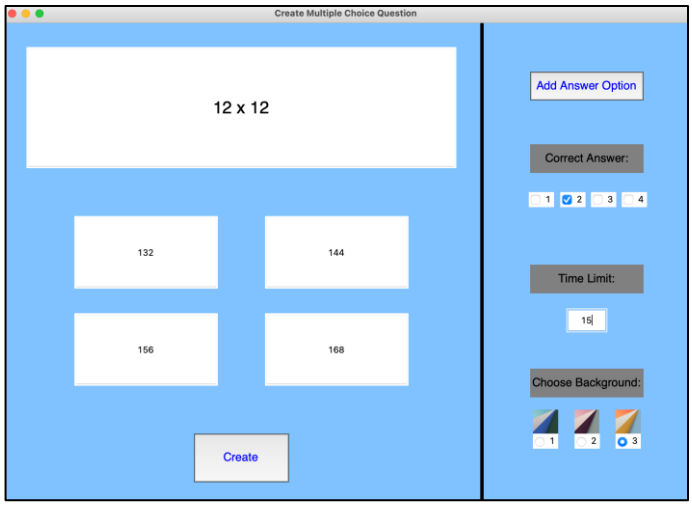
```
if timeLimit != '':
    timeLimitLabel.config(text = 'Time Limit: '+timeLimit, bg = 'gray')
    timeLimitFrame.place(relx = 0.75, rely = 0.75, relheight = 0.1, relwidth = 0.2)
    timeLimitLabel.place(relx = 0, rely = 0, relheight = 1, relwidth = 1)
```

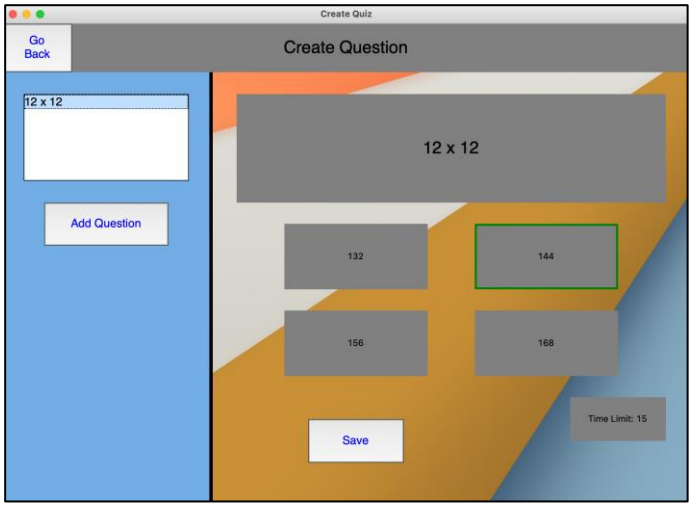
And configured and placed it accordingly. Since all questions will have a time limit, this label does not need to be hidden.

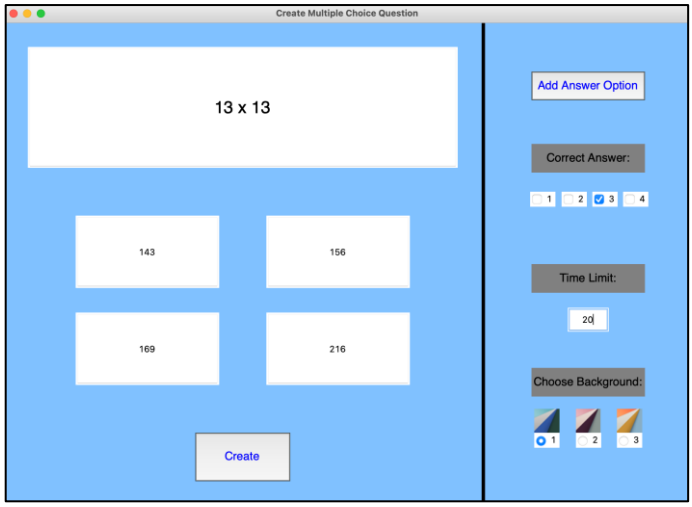
To show the correct answer:

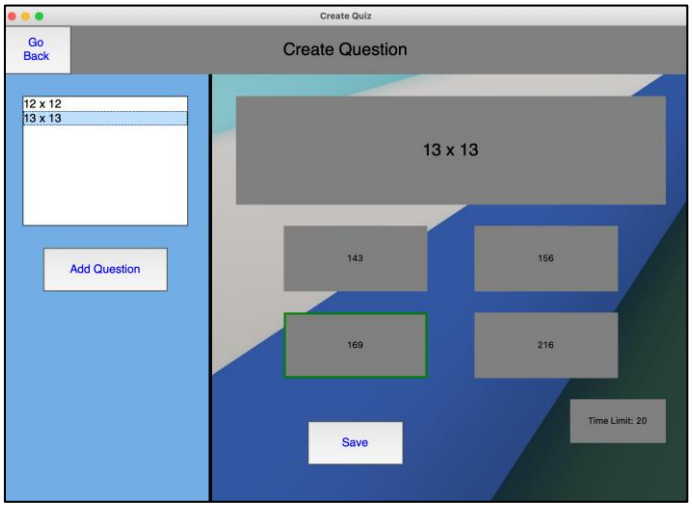
```
if correctAnswer == 1:
    answerOneFrame.config(highlightthickness = 3, highlightbackground = 'green')
elif correctAnswer == 2:
    answerTwoFrame.config(highlightthickness = 3, highlightbackground = 'green')
elif correctAnswer == 3:
    answerThreeFrame.config(highlightthickness = 3, highlightbackground = 'green')
elif correctAnswer == 4:
    answerFourFrame.config(highlightthickness = 3, highlightbackground = 'green')
```

I configured the frame of the label with the correct answer to have a green border.





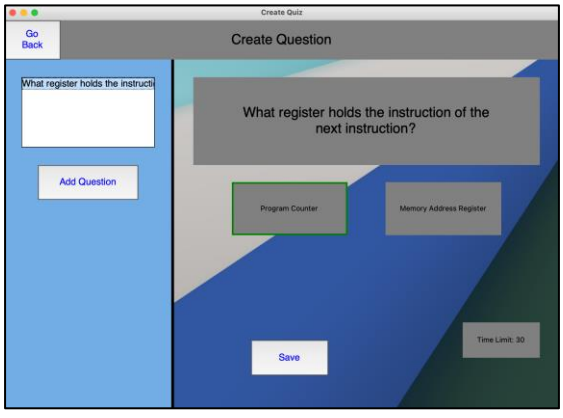




To deal with the issue of the labels not wrapping around, so text may be hidden:

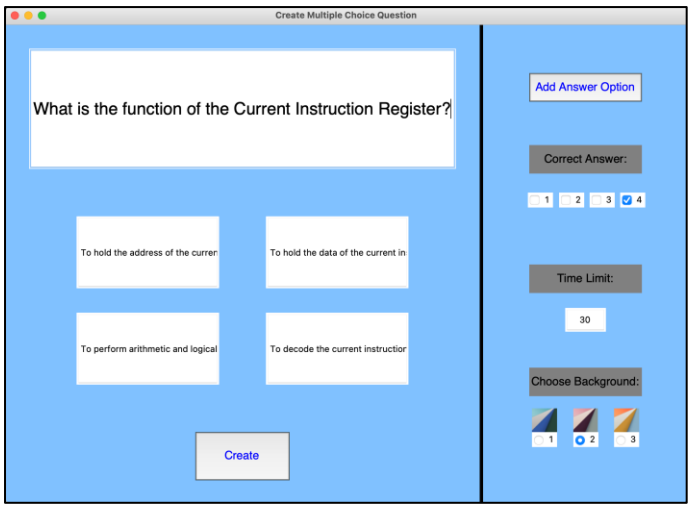
```
questionLabel.config(text = selectedQuestion, bg = 'gray', font = titleFont, wraplength = 500)
```

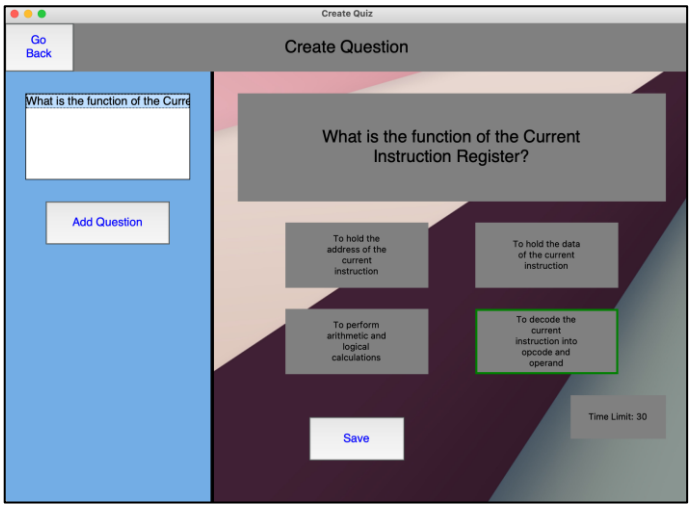
Where I configured the question label, I added wraplength = 500, so that the label automatically wrapped itself around.



I did the same for the answer labels:

```
answerOneLabel.config(text = answers[0], bg = 'gray', wraplength = 100)
answerTwoLabel.config(text = answers[1], bg = 'gray', wraplength = 100)
answerThreeLabel.config(text = answers[2], bg = 'gray', wraplength = 100)
answerFourLabel.config(text = answers[3], bg = 'gray', wraplength = 100)
```



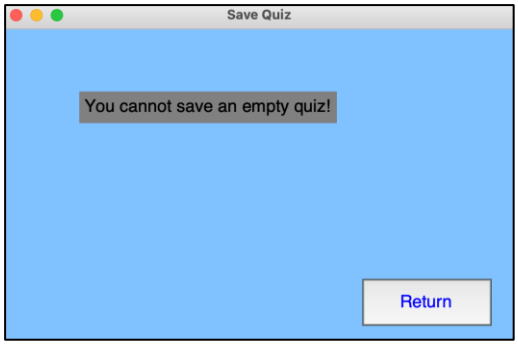


Finally, I had to finalise the saving of a quiz:

```
def saveQuiz(self, titleFont, defaultFont, buttonFont):
    print('Save Quiz')
    ## Retrieves the assigned height, width and fonts for the main create question page
    mainScreenWidth = saveNewQuiz.getScreenWidth()
    mainScreenHeight = saveNewQuiz.getScreenHeight()

    saveQuizPage = tk.Tk()
    saveQuizPage.title("Save Quiz")

    canvas = tk.Canvas(saveQuizPage, height = mainScreenHeight, width = mainScreenWidth)
    canvas.pack()

    mainframe = tk.Frame(saveQuizPage, bg = '#80c1ff')
    mainframe.place(relheight = 1, relwidth = 1)
```

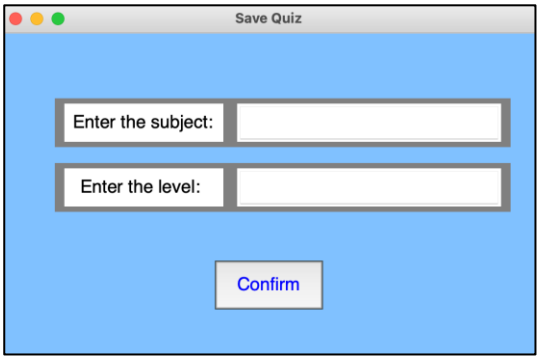
First, I created a tkinter page with its title, canvas and mainframe.
I then checked to see if the questions list had length one (so only containing [‘No Questions’]), and informed the user that the quiz could not be saved.

```
## If no questions have been created
if len(self.questions) == 1:
    blankLabel = tk.Label(mainframe, text = "You cannot save an empty quiz!", bg = 'gray', font = defaultFont)
    blankLabel.place(relx = 0.15, rely = 0.2, relheight = 0.1, relwidth = 0.5)

    returnButton = tk.Button(mainframe, text = "Return", highlightbackground = '#424949', fg = 'blue', font = defaultFont, command = saveQuizPage.destroy)
    returnButton.place(relx = 0.7, rely = 0.8, relheight = 0.15, relwidth = 0.25)
```



The return button simply closes this page so the user is back to the create quiz page.

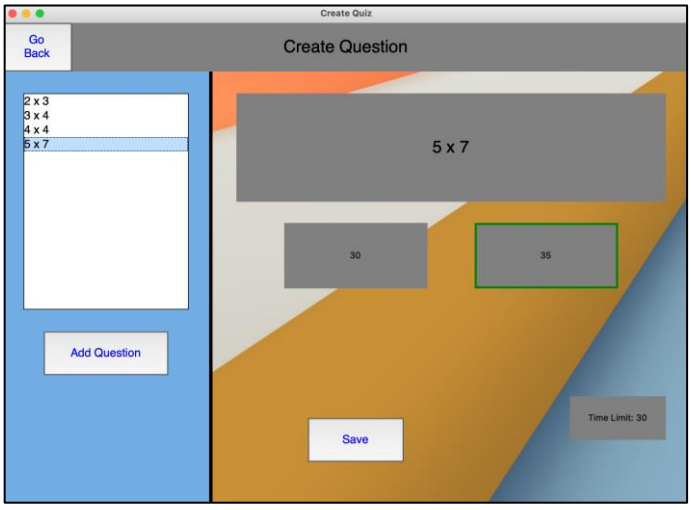
```
## If questions have been created
else:
    subjectFrame = tk.Frame(mainframe, bg = 'gray')
    subjectFrame.place(relx = 0.1, rely = 0.2, relheight = 0.15, relwidth = 0.85)

    subjectLabel = tk.Label(subjectFrame, text = "Enter the subject:", font = defaultFont)
    subjectLabel.place(relx = 0.02, rely = 0.1, relheight = 0.8, relwidth = 0.35)

    subjectEntry = tk.Entry(subjectFrame)
    subjectEntry.place(relx = 0.4, rely = 0.1, relheight = 0.8, relwidth = 0.58)
    subjectEntry.bind("<Return>", lambda x: self.confirmSave(subjectEntry.get(), levelEntry.get()))

    levelFrame = tk.Frame(mainframe, bg = 'gray')
    levelFrame.place(relx = 0.1, rely = 0.4, relheight = 0.15, relwidth = 0.85)

    levelLabel = tk.Label(levelFrame, text = "Enter the level: ", font = defaultFont)
    levelLabel.place(relx = 0.02, rely = 0.1, relheight = 0.8, relwidth = 0.35)

    levelEntry = tk.Entry(levelFrame)
    levelEntry.place(relx = 0.4, rely = 0.1, relheight = 0.8, relwidth = 0.58)
    levelEntry.bind("<Return>", lambda x: self.confirmSave(subjectEntry.get(), levelEntry.get()))

    confirmButton = tk.Button(mainframe, text = "Confirm", highlightbackground = '#424949', fg = 'blue', font = defaultFont, command = lambda:
    confirmButton.place(relx = 0.4, rely = 0.7, relheight = 0.15, relwidth = 0.2)
                                                    self.confirmSave(subjectEntry.get(), levelEntry.get()))

    saveQuizPage.mainloop()
```

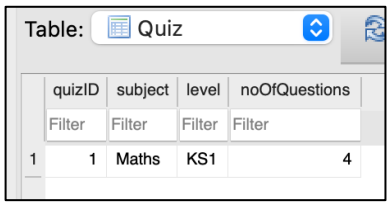


The next part occurs if questions have been created. It asks the user for the subject and level of the quiz they wish to create and to confirm by pressing a button. I have also created binds on the entries to initiate the confirmSave function when the user presses the enter key.

The confirmSave function:
This function would do three things. It would first create a new record in the Quiz table with an automatically allocated quiz ID, the subject and level given by the user, and the number of questions. It would then add all the relevant questions into the Questions table within the database with all the details provided by the user and the question and quiz ID automatically filled in. Finally it would reset the questions list so no interference was run if a new quiz was to be created.

```
def confirmSave(self, subject, level):
    record = [None, subject, level, len(self.questions)-1]
    addingToDatabase.addToDatabase(conn, c, record, 'Quiz')
```





I created the record with None for the quiz ID since that would be automatically done by SQL, the subject and level from the previous page and finally the length of the questions list minus one (since ['No Questions'] was part of the list).

```
def confirmSave(self, subject, level):
    ## Create record for quiz table and add it to database
    quizRecord = [None, subject, level, len(self.questions)-1]
    addingToDatabase.addToDatabase(conn, c, quizRecord, 'Quiz')

    ## Retrieve the quiz ID of the last entry in the quiz table
    quizID = viewingFromDatabase.viewFromDatabase(c, False, True, 'quizID', None, None, 'Quiz ORDER BY quizID DESC LIMIT 1')
    quizID = quizID[0][0]

    ## Remove ['No Questions'] from questions list
    self.questions.remove(['No Questions'])
    for each in self.questions:
        ## If correct answer is greater than 1 (i.e. the correct answer is not at the front of the list)
        if each[2] > 1:
            ## Place correct answer at front of answers list
            each[1].insert(0, each[1].pop(each[2]-1))
        ## Create record for question table and add it to database
        questionRecord = [None, quizID, 'Multiple Choice', each[0], len(each[1]), each[1], each[3], None, each[4]]
        addingToDatabase.addToDatabase(conn, c, questionRecord, 'Question')

    ## Reset the questions list
    self.questions = [['No Questions']]

    print('done')
```

I extended the function to perform the remaining required tasks: adding the questions to the database and resetting the questions list. To add questions to the database, I had to assign the relevant quiz ID, and so to do that I retrieved the first entry of the database in descending ID order. However, upon creating and testing this function, I found some limitations to my design of the database. Firstly, one field was unable to take in a list from python as how proposed above. Secondly, this data would not be atomic, and would therefore violate the rules of a first normal form database. I hence made some changes to the database before finalising my function.
I modified the Question table to:

Question		CREATE TABLE "Question" ("questionID" IN
questionID	INTEGER	"questionID" INTEGER NOT NULL UNIQUE
quizID	INTEGER	"quizID" INTEGER
questionType	TEXT	"questionType" TEXT
question	TEXT	"question" TEXT
noOfAnswers	INTEGER	"noOfAnswers" INTEGER
answerOne	TEXT	"answerOne" TEXT
answerTwo	TEXT	"answerTwo" TEXT
answerThree	TEXT	"answerThree" TEXT
answerFour	TEXT	"answerFour" TEXT
timeLimit	INTEGER	"timeLimit" INTEGER
characterLimit	INTEGER	"characterLimit" INTEGER
background	INTEGER	"background" INTEGER

Here, I separated out the answer field to four separate fields, since there will be a maximum of four answers per question for multiple choice questions. This would also mean that there would have to be a four answer option maximum for order answers questions as well. In this table, answerOne would contain the correct answer. Any question with less than four options would have the fields as blank.

I modified my database class accordingly:

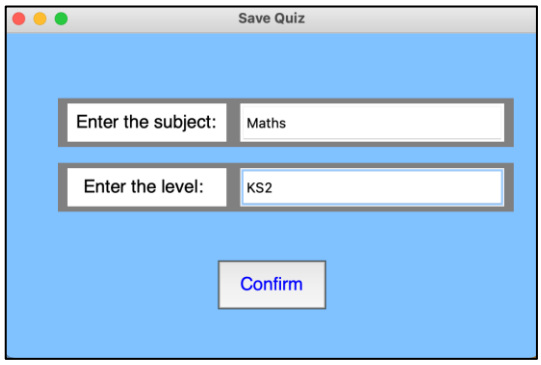
```
## If to be added to Question table
if len(record) == 12:
    # Establish SQL statement
    sqlAdd = ("INSERT INTO " + table + " VALUES (?,?,?,?,?,?,?,?,?,?,?,?)")
```

I therefore changed my confirmSave function to:

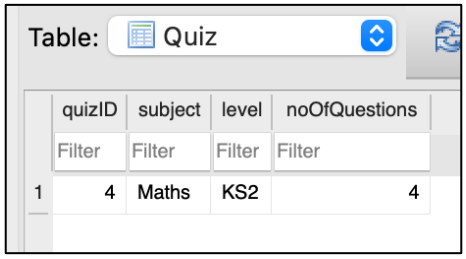
```
## Remove ['No Questions'] from questions list
self.questions.remove(['No Questions'])
for each in self.questions:
    ## If correct answer is greater than 1 (i.e. the correct answer is not at the front of the list)
    if each[2] > 1:
        ## Place correct answer at front of answers list
        each[1].insert(0, each[1].pop(each[2]-1))
    ## Retrieve answers from questions list
    if len(each[1]) > 1:
        answerOne = each[1][0]
        answerTwo = each[1][1]
    else:
        answerTwo = None
    if len(each[1]) > 2:
        answerThree = each[1][2]
    else:
        answerThree = None
    if len(each[1]) > 3:
        answerFour = each[1][3]
    else:
        answerFour = None
    ## Create record for question table
    questionRecord = [None, quizID, 'Multiple Choice', each[0], len(each[1]), answerOne, answerTwo, answerThree, answerFour, each[3], None, each[4]]
    addingToDatabase.addToDatabase(conn, c, questionRecord, 'Question')
```

Here, I extended it to define four variables for each corresponding answer. The program checks the size of the answer list and depending on how many answers there are, the four variables are filled with the respective answers or with None.

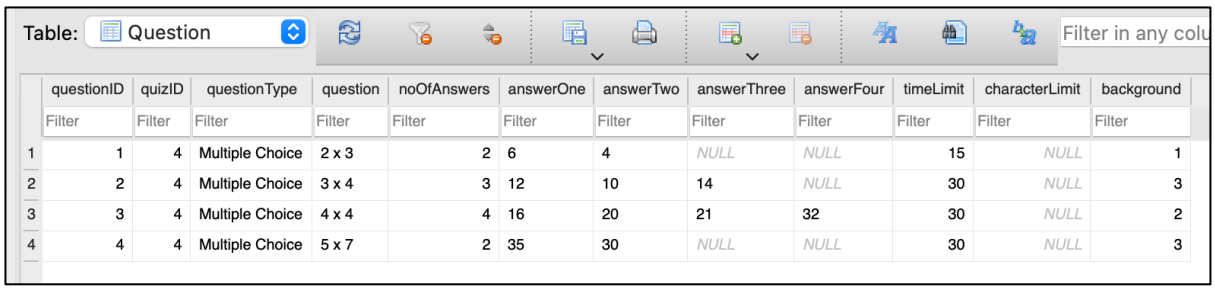
Using the questions shown above and:



The program produced:



	quizID	subject	level	noOfQuestions
	Filter	Filter	Filter	Filter
1	4	Maths	KS2	4



	questionID	quizID	questionType	question	noOfAnswers	answerOne	answerTwo	answerThree	answerFour	timeLimit	characterLimit	background
	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter
1	1	4	Multiple Choice	2 x 3	2	6	4	NULL	NULL	15	NULL	1
2	2	4	Multiple Choice	3 x 4	3	12	10	14	NULL	30	NULL	3
3	3	4	Multiple Choice	4 x 4	4	16	20	21	32	30	NULL	2
4	4	4	Multiple Choice	5 x 7	2	35	30	NULL	NULL	30	NULL	3

I created a function that warns the user that any unsaved work will be discarded if they press the go back button. This function then asks them to confirm if they want to return to the main menu.

```
def returnConfirmation(self, previousPage, username, titleFont, defaultFont, buttonFont):
    ## Retrieves the assigned height, width and fonts for the main create question page
    mainScreenWidth = saveNewQuiz.getScreenWidth()
    mainScreenHeight = saveNewQuiz.getScreenHeight()

    returnConfirmationPage = tk.Tk()
    returnConfirmationPage.title("Confirm Return")

    canvas = tk.Canvas(returnConfirmationPage, height = mainScreenHeight, width = mainScreenWidth)
    canvas.pack()

    mainframe = tk.Frame(returnConfirmationPage, bg = '#80c1ff')
    mainframe.place(relheight = 1, relwidth = 1)

    confirmationLabel = tk.Label(mainframe, text = "Are you sure you want to return?\nAnything not saved will be discarded", bg = 'gray', font = defaultFont)
    confirmationLabel.place(relx = 0.2, rely = 0.25, relheight = 0.2, relwidth = 0.6)

    confirmButton = tk.Button(mainframe, text = "Confirm", highlightbackground = '#424949', fg = 'blue', font = defaultFont, command = lambda: (returnConfirm
    confirmButton.place(relx = 0.375, rely = 0.65, relheight = 0.15, relwidth = 0.25)
                                                                                    (returnConfirmationPage.destroy(),
                                                                                    self.returnToMainMenu(previousPage, username)))
```

Finally, I extended the confirmSave function to bring the user back to the main menu once finished creating a quiz, passing all relevant parameters along:

```
## Return to main menu
mainCreateQuestionPage.destroy()
self.returnToMainMenu(previousPage, username)
```

It destroys both the create quiz page and save page and then opens the admin user using the returnToMainMenu function and the username.

Creating an answer input question:
Since the layout of my create answer input question function would resemble largely that of the create multiple choice question, in favour of effectiveness, I chose to do it as follows. I changed the create multiple choice question to:

```
## Creates the create multiple choice question page interface
answerOptions = 2
answers =[]
def createNewQuestion(self, typeOfQuestion, previousPage, username, mainCreateQuizPage, titleFont, defaultFont, buttonFont):
    previousPage.destroy()
    ## Retrieves the assigned height, width and fonts for the main create question page
    mainScreenWidth = mainMenu.getScreenWidth()
    mainScreenHeight = mainMenu.getScreenHeight()

    mainCreateQuestionPage = tk.Toplevel()
    mainCreateQuestionPage.title("Create Question")

    canvas = tk.Canvas(mainCreateQuestionPage, height = mainScreenHeight, width = mainScreenWidth)
    canvas.pack()

    background = tk.Frame(mainCreateQuestionPage, bg = 'black')
    background.place(relheight = 1, relwidth = 1)

    mainframe = tk.Frame(mainCreateQuestionPage, bg = '#80c1ff')
    mainframe.place(relheight = 1, relwidth = 0.695)

    questionEntry = tk.Entry(mainframe, font = titleFont, justify = 'center')
    questionEntry.place(relx = 0.05, rely = 0.05, relheight = 0.25, relwidth = 0.9)
    questionEntry.insert('end', 'Enter a Question...')

    createButton = tk.Button(mainframe, text = "Create", highlightbackground = '#424949', fg = 'blue', font = defaultFont, command = lambda: self.validateTimeLim
    createButton.place(relx = 0.4, rely = 0.85, relheight = 0.1, relwidth = 0.2)

    sideframe = tk.Frame(mainCreateQuestionPage, bg = '#80c1ff')
    sideframe.place(relheight = 1, relwidth = 0.3, relx = 0.7)

    timeLimitLabel = tk.Label(sideframe, text = 'Time Limit:', bg = 'gray', font = defaultFont)
    timeLimitLabel.place(relx = 0.225, rely = 0.5, relheight = 0.06, relwidth = 0.55)

    timeLimitEntry = tk.Entry(sideframe, justify = 'center', validate = 'key', validatecommand = (sideframe.register(self.validate),'%d', '%i', '%P', '%s', '%S',
    timeLimitEntry.place(relx = 0.4, rely = 0.59, relheight = 0.05, relwidth = 0.2)
    timeLimitEntry.insert('end', 30)
```

```
    maxTimeLimitLabel = tk.Label(sideframe)

    selectBackgroundLabel = tk.Label(sideframe, text = "Choose Background:", bg = 'gray', font = defaultFont)
    selectBackgroundLabel.place(relx = 0.225, rely = 0.715, relheight = 0.06, relwidth = 0.55)

    backgroundOneImage = Image.open('BackgroundOne.jpg')
    backgroundOneImage = backgroundOneImage.resize((40,40))
    tkBackgroundOneImage = ImageTk.PhotoImage(backgroundOneImage)
    backgroundOneImageLabel = tk.Label(sideframe, image = tkBackgroundOneImage)
    backgroundOneImageLabel.place(relx = 0.24, rely = 0.8, relheight = 0.05, relwidth = 0.12)

    backgroundTwoImage = Image.open('BackgroundTwo.jpg')
    backgroundTwoImage = backgroundTwoImage.resize((40,40))
    tkBackgroundTwoImage = ImageTk.PhotoImage(backgroundTwoImage)
    backgroundTwoImageLabel = tk.Label(sideframe, image = tkBackgroundTwoImage)
    backgroundTwoImageLabel.place(relx = 0.44, rely = 0.8, relheight = 0.05, relwidth = 0.12)

    backgroundThreeImage = Image.open('BackgroundThree.jpg')
    backgroundThreeImage = backgroundThreeImage.resize((40,40))
    tkBackgroundThreeImage = ImageTk.PhotoImage(backgroundThreeImage)
    backgroundThreeImageLabel = tk.Label(sideframe, image = tkBackgroundThreeImage)
    backgroundThreeImageLabel.place(relx = 0.64, rely = 0.8, relheight = 0.05, relwidth = 0.12)

    trackRadiobutton = tk.IntVar() ## Creating a variable which will track the selected radiobutton
    radioButtonList = [] ## Empty list which is going to hold all the radiobuttons
    ## Creating three radiobuttons (where only one can be selected at a time)
    position = 0.24
    for j in range(3):
        radioButtonList.append(tk.Radiobutton(sideframe, value = j, variable = trackRadiobutton, text = j+1))
        radioButtonList[j].place(relx = position, rely = 0.85)
        position += 0.2
```

Here I changed the function to a general createNewQuestion function and initially created all the aspects of the page that would be universal to all three create question pages. I also passed in an extra parameter here: typeOfQuestion, which would determine what button had been pressed, with this variable, I could now create an if statement to fill in the remainder of the aspects of the page that would be unique to that question type.

```
tonFont, command = lambda: self.createNewQuestion('Multiple Choice',

command = lambda: self.createNewQuestion('Answer Input', mainCreateQ

t, command = lambda: self.createNewQuestion('Order Answers', mainCre
```

In the previous function, depending on the button they pressed, I passed in 'Multiple Choice', 'Answer Input' or 'Order Answers' accordingly.

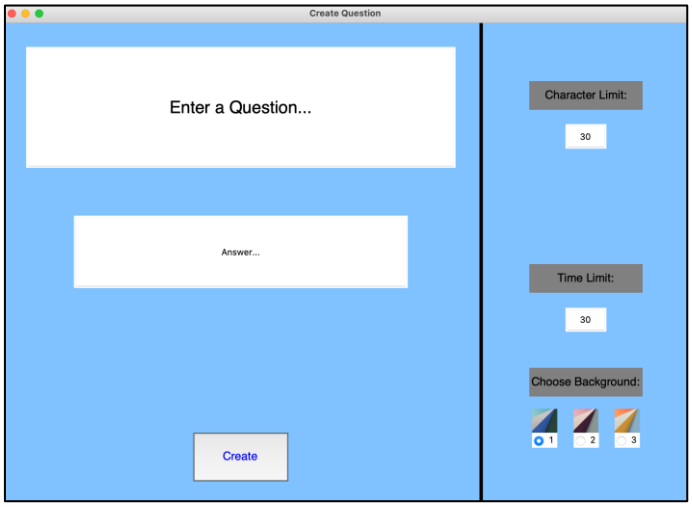
```
if typeOfQuestion == 'Multiple Choice':
    answerOneEntry = tk.Entry(mainframe, justify = 'center')
    answerOneEntry.place(relx = 0.15, rely = 0.4, relheight = 0.15, relwidth = 0.3)
    answerOneEntry.insert('end', 'Answer 1...')

    answerTwoEntry = tk.Entry(mainframe, justify = 'center')
    answerTwoEntry.place(relx = 0.55, rely = 0.4, relheight = 0.15, relwidth = 0.3)
    answerTwoEntry.insert('end', 'Answer 2...')

    answerThreeEntry = tk.Entry(mainframe, justify = 'center')
    answerFourEntry = tk.Entry(mainframe, justify = 'center')

    addAnswerOptionButton = tk.Button(sideframe, text = 'Add Answer Option', highlightbackground = '#424949', fg = 'blue', font = defaultFont, command =
    addAnswerOptionButton.place(relx = 0.225, rely = 0.1, relheight = 0.06, relwidth = 0.55)

    fourOptionsLabel = tk.Label(sideframe)

    correctAnswerLabel = tk.Label(sideframe, text = 'Correct Answer:', bg = 'gray', font = defaultFont)
    correctAnswerLabel.place(relx = 0.225, rely = 0.25, relheight = 0.06, relwidth = 0.55)

    trackCheckbutton = tk.IntVar() ## Creating a variable which will track the selected checkbutton
    checkButtonList = [] ## Empty list which is going to hold all the checkbuttons
    ## Creating four checkbuttons (where only one can be selected at a time)
    for i in range(4):
        checkButtonList.append(tk.Checkbutton(sideframe, onvalue = i, variable = trackCheckbutton, text = i+1))
    checkButtonList[0].place(relx = 0.22, rely = 0.35)
    checkButtonList[1].place(relx = 0.37, rely = 0.35)
```

Here is the remainder of the create new multiple choice question section of the code.

For answer input questions:

```
elif typeOfQuestion == 'Answer Input':
    answerOneEntry = tk.Entry(mainframe, justify = 'center')
    answerOneEntry.place(relx = 0.15, rely = 0.4, relheight = 0.15, relwidth = 0.7)
    answerOneEntry.insert('end', 'Answer...')

    characterLimitLabel = tk.Label(sideframe, text = 'Character Limit:', bg = 'gray', font = defaultFont)
    characterLimitLabel.place(relx = 0.225, rely = 0.12, relheight = 0.06, relwidth = 0.55)

    characterLimitEntry = tk.Entry(sideframe, justify = 'center', validate = 'key', validatecommand = (sideframe.register(self.validate),'%d', '%i', '%P', '%
    characterLimitEntry.place(relx = 0.4, rely = 0.21, relheight = 0.05, relwidth = 0.2)
    characterLimitEntry.insert('end', 30)
```

Here, there would be only one answer option, and a character limit. The character limit undergoes the same validation as the time limit in terms of typing only numbers. It is also pre-filled with a base value of 30.
This produced:



However, this was now incompatible with my validate time limit and create question function since there were variables that were parameters of those functions which now didn't exist (the answer two, three, four entry and the check button tracker).
I therefore created all answer options label outside the if statement and used the if statements just to place the correct ones. This was also more efficient since it prevented me from repeating creating the answer options

```
trackCheckbutton = None
```

I also initially set the trackCheckbutton variable as None outside the if statement, so answer input questions and order answers questions pass on a value of None to the next function.
I had to pass the character limit entry to the next function too, so I also initialised that outside of the if statement.
I passed in the character limit entry to the validate time limit and the create question function and modified the latter:

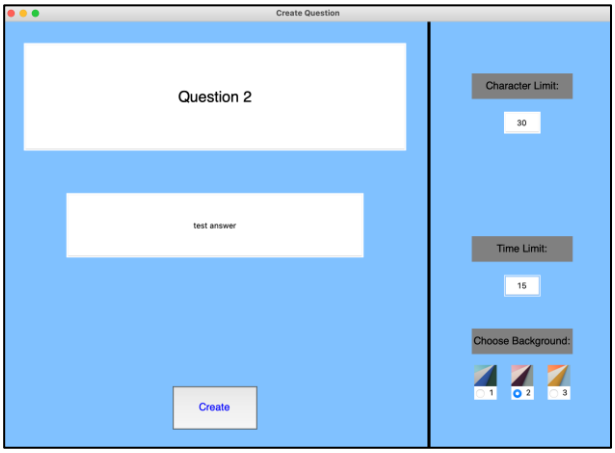
```
def createQuestion(self, previousPage, username, mainCreateQuizPage, questionType, question, a
    background = trackRadiobutton.get() + 1

    if questionType == 'Multiple Choice':
        self.answers = [answerOne.get(), answerTwo.get()]
        if self.answerOptions > 2:
            self.answers.append(answerThree.get())
        if self.answerOptions == 4:
            self.answers.append(answerFour.get())
        correctAnswer = trackCheckbutton.get() + 1
        self.questions.append([question, self.answers, correctAnswer, timeLimit, background])
        self.answerOptions = 2
    elif questionType == 'Answer Input':
        self.answers = [answerOne.get()]
        characterLimit = characterLimitEntry.get()
        self.questions.append([question, self.answers, characterLimit, timeLimit, background])

    self.answers = []
    previousPage.destroy()
    mainCreateQuizPage.destroy()
    self.createQuiz(username, Login.titleFont, Login.defaultFont, Login.buttonFont)
```

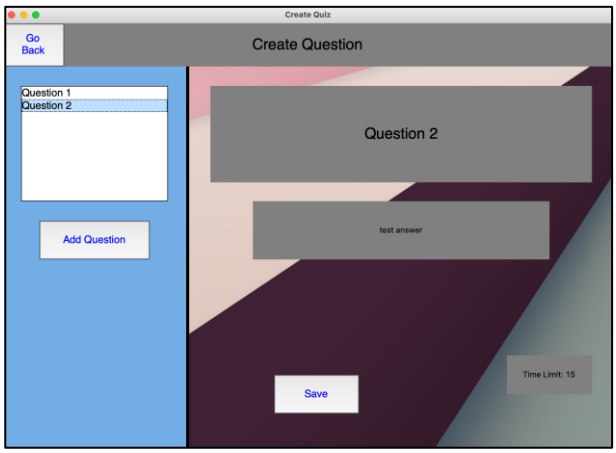
This function now retrieved the background (since all questions would have one) and assigned its answers according to its questionType. In the place of the correctAnswer variable for multiple choice question, I put the character limit for an answer input question. I changed the following functions to agree with this new format.

I first modified the displayQuestion function:

```
if len(answers) > 0:
    questionLabel.config(text = selectedQuestion, bg = 'gray', font = titleFont, wraplength = 500)
    questionFrame.place(relx = 0.05, rely = 0.05, relheight = 0.25, relwidth = 0.9)
    questionLabel.place(relx = 0, rely = 0, relheight = 1, relwidth = 1)
    answerOneLabel.config(text = answers[0], bg = 'gray', wraplength = 100)
if len(answers) == 1:
    answerOneFrame.place(relx = 0.15, rely = 0.35, relheight = 0.15, relwidth = 0.7)
    answerOneLabel.place(relx = 0, rely = 0, relheight = 1, relwidth = 1)
if len(answers) > 1:
    answerOneFrame.place(relx = 0.15, rely = 0.35, relheight = 0.15, relwidth = 0.3)
    answerOneLabel.place(relx = 0, rely = 0, relheight = 1, relwidth = 1)
    answerTwoLabel.config(text = answers[1], bg = 'gray', wraplength = 100)
    answerTwoFrame.place(relx = 0.55, rely = 0.35, relheight = 0.15, relwidth = 0.3)
    answerTwoLabel.place(relx = 0, rely = 0, relheight = 1, relwidth = 1)
```

I modified this section of the code to include an option for where the length of the answers list is equal to one. First I configured the answer one label when the length of answers was greater than 0 (to prevent repetition). I then placed it accordingly for when there was one answer, i.e. an answer input question, and where there were multiple.



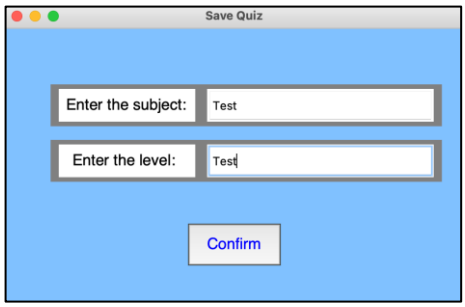


I consequently had to modify a section of my confirmSave function:

```
characterLimit = None
## For each question
for each in self.questions:
    ## Retrieve answers from questions list
    if len(each[1]) == 1:
        answerOne = each[1][0]
        questionType = 'Answer Input'
        characterLimit = each[2]
    if len(each[1]) > 1:
        ## If correct answer is greater than 1 (i.e. the correct answer is not at the front of the list)
        if each[2] > 1:
            ## Place correct answer at front of answers list
            each[1].insert(0, each[1].pop(each[2]-1))
        answerOne = each[1][0]
        answerTwo = each[1][1]
        questionType = 'Multiple Choice'
    else:
        answerTwo = None
    if len(each[1]) > 2:
        answerThree = each[1][2]
    else:
        answerThree = None
    if len(each[1]) > 3:
        answerFour = each[1][3]
    else:
        answerFour = None
    ## Create record for question table
    questionRecord = [None, quizID, questionType, each[0], len(each[1]), answerOne, answerTwo, answerThree, answerFour, each[3], characterLimit, each[4]]
    addingToDatabase.addToDatabase(conn, c, questionRecord, 'Question')
```

In this new function, I created the option of the length of the answers list being just one, and only changed places of the correct answer if the length was above one. I also created a questionType variable where when the length of the answers list was just one, it became 'Answer Input' and when it was greater than one, it became 'Multiple Choice'. I may have to modify this further later on for the order answers questions. I also declared a characterLimit variable which was initially set as None and changed to the third value of the question list if the length of the answers list for that question was one.

Using the questions created above, and:



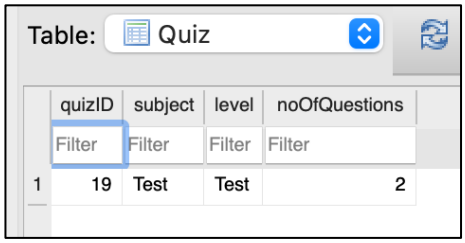
It initially produced:

```
    if each[2] > 1:
TypeError: '>' not supported between instances of 'str' and 'int'
```

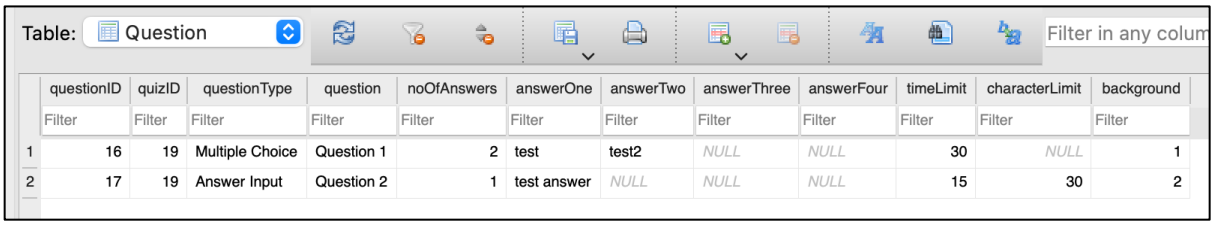
So I modified where I obtained the character limit and made it an integer.

```
characterLimit = int(characterLimitEntry.get())
```

After all these changes, it then brought me to the main menu and produced:



	quizID	subject	level	noOfQuestions
	Filter	Filter	Filter	Filter
1	19	Test	Test	2



	questionID	quizID	questionType	question	noOfAnswers	answerOne	answerTwo	answerThree	answerFour	timeLimit	characterLimit	background
	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter
1	16	19	Multiple Choice	Question 1	2	test	test2	NULL	NULL	30	NULL	1
2	17	19	Answer Input	Question 2	1	test answer	NULL	NULL	NULL	15	30	2

View Quizzes:

```
def viewQuizzes(self, username, titleFont, defaultFont, buttonFont):
    ## Retrieves the assigned height, width and fonts for the main create question page
    mainScreenWidth = viewQuiz.getScreenWidth()
    mainScreenHeight = viewQuiz.getScreenHeight()

    viewQuizzesPage = tk.Tk()
    viewQuizzesPage.title("View Quizzes")

    canvas = tk.Canvas(viewQuizzesPage, height = mainScreenHeight, width = mainScreenWidth)
    canvas.pack()

    mainframe = tk.Frame(viewQuizzesPage, bg = '#80c1ff')
    mainframe.place(relheight = 1, relwidth = 1)

    welcomeLabel = tk.Label(mainframe, text = "View Quizzes", bg = 'gray', font = titleFont)
    welcomeLabel.place(relheight = 0.1, relwidth = 1)

    returnButton = tk.Button(mainframe, text = "Go\nBack", highlightbackground = '#424949', fg = 'blue', font = defaultFont, command = lambda: self.
    returnButton.place(relheight = 0.1, relwidth = 0.1)

    background = tk.Frame(viewQuizzesPage, bg = 'black')
    background.place(rely = 0.1, relheight = 0.9, relwidth = 1)

    quizFrame = tk.Frame(viewQuizzesPage, bg = '#73ade5')
    quizFrame.place(rely = 0.1, relheight = 0.9, relwidth = 0.2)

    quizListbox = tk.Listbox(quizFrame, font = defaultFont)
    listboxHeight = 0
    buttonPos = 0.1
    quizzes = viewingFromDatabase.viewFromDatabase(c, True, True, None, None, None, 'Quiz')
    print(quizzes)
    for i in range(len(quizzes)):
        quizListbox.insert('end', 'Quiz ' + str(i+1))
        if buttonPos < 0.8:
            listboxHeight += 0.1
            buttonPos += 0.1
    quizListbox.place(relx = 0.1, rely = 0.05, relheight = listboxHeight, relwidth = 0.8)
    quizListbox.bind('<<ListboxSelect>>', lambda x: self.displayQuestions(questionLabel, quizListbox.get(quizListbox.curselection()), questionFrame,
```

Here, I created a page similar to the create quiz page, where there is a frame on the left for all the quizzes. This frame contains a list box that shows all the questions of that quiz.

```
questionsFrame = tk.Frame(viewQuizzesPage, bg = '#73ade5')
questionsFrame.place(relx = 0.205, rely = 0.1, relheight = 0.9, relwidth = 0.2)

questionListbox = tk.Listbox(questionsFrame, font = defaultFont)

rightFrame = tk.Frame(viewQuizzesPage, bg = '#80c1ff')
rightFrame.place(relx = 0.410, rely = 0.1, relheight = 0.9, relwidth = 0.695)

backgroundImageLabel = tk.Label(rightFrame)
questionFrame = tk.Frame(rightFrame)
questionLabel = tk.Label(questionFrame)
answerOneFrame = tk.Frame(rightFrame)
answerOneLabel = tk.Label(answerOneFrame)
answerTwoFrame = tk.Frame(rightFrame)
answerTwoLabel = tk.Label(answerTwoFrame)
answerThreeFrame = tk.Frame(rightFrame)
answerThreeLabel = tk.Label(answerThreeFrame)
answerFourFrame = tk.Frame(rightFrame)
answerFourLabel = tk.Label(answerFourFrame)
timeLimitFrame = tk.Frame(rightFrame)
timeLimitLabel = tk.Label(timeLimitFrame)
```

I also created another ‘side’ frame for the questions of a quiz, right beside the quiz frame. I extended the size of the page so the selected question can be viewed on the right of the question frame, as in the create quiz function.

For the quiz list box, I created a new function that displayed all the questions for that quiz in the questions list box:

```
def displayQuestions(self, questionListbox, questionLabel, selectedQuiz, questionFrame, answerOneFrame, answerOneLabel, answerTwoFrame, answe
    ## Retrieve quiz ID from database for selected quiz
    quizID = viewingFromDatabase.viewFromDatabase(c, False, True, "quizID", None, None, "Quiz LIMIT " + str(int(selectedQuiz[5])-1) + ", 1")
    quizID = quizID[0][0] ## Isolate it from list and tuple
    ## Retrieve all questions from database for selected quiz
    questions = viewingFromDatabase.viewFromDatabase(c, True, False, None, 'quizID', str(quizID), 'Question')

    ## Clear the question list box (to allow for changing questions)
    questionListbox.delete(0,'end')
    listboxHeight = 0
    ## Add each question from questions list to questions list box
    for each in questions:
        questionListbox.insert('end', each[3])
        if listboxHeight < 0.8:
            listboxHeight += 0.1
    questionListbox.place(relx = 0.1, rely = 0.05, relheight = listboxHeight, relwidth = 0.8)
    self.questions = []
    ## Bind any selection within listbox to event
    questionListbox.bind('<<ListboxSelect>>', lambda x: self.displayQuestion(questionListbox, questions, questionLabel, questionListbox.get(
```

This function takes in as parameters all those that the displayQuestion function would take and the question list box. It first retrieves the quiz ID of the quiz chosen using the row of the selection in the list box. Since the quizzes are ordered numerically, it uses the 6th character, which will always be the number of the quiz. Using the quiz ID, it then retrieves all the questions which has a matching quiz ID. It then goes on to clear the question list box in the case of changing questions (since questions from a previous question may be displayed if it contains more options than the next). With the now empty list box, it inserts all the questions of the quiz, increasing the size of the list box as it goes until it reaches a height of 0.8 or 80% of the height of the screen. It then goes on to empty the questions list within the class. This is important in the modified version of the displayQuestion function when recognising which function it has come from. It does require that I reset it when creating a quiz though.

```
self.questions = [['No Questions']]
```

Although the above code initially appears to work, since the database is matched against the 6th character of the quiz name, this would mean that once more than 9 quizzes have been created, this would no longer work as desired. To fix this:

```
## Retrieve quiz ID from database for selected quiz
index = quizListbox.get(0, "end").index(selectedQuiz)
quizID = viewingFromDatabase.viewFromDatabase(c, False, True, "quizID", None, None, "Quiz LIMIT " + str(index) + ", 1")
```

I instead retrieved the row that was selected and matched that against the database.
In the modified displayQuestion function, I had to pass in two extra parameters: the questions list and the question list box.

First, I checked the length of the questions list within the class to see whether it was above 0. If it was, all the necessary values from the create quiz functions would apply.

```
def displayQuestion(self, questionListbox, questions, questionLabe
    ## Hide all answer frames (in the case of changing questions)
    answerOneFrame.place_forget()
    answerTwoFrame.place_forget()
    answerThreeFrame.place_forget()
    answerFourFrame.place_forget()

    ## Check if class questions list is above 0
    if len(self.questions) > 0:
        ## Find position of selected question
        for each in self.questions:
            if each[0] == selectedQuestion:
                try:
                    answers = each[1]
                    correctAnswer = each[2]
                    timeLimit = each[3]
                    background = each[4]
                except:
                    answers = []
                    correctAnswer = ''
                    timeLimit = ''
                    background = ''
```

If not:

```
else:
    ## If retrieving from database
    index = questionListbox.get(0, "end").index(selectedQuestion)
    answers = [questions[index][5], questions[index][6], questions[index][7], questions[index][8]]
    correctAnswer = questions[index][5]
    timeLimit = questions[index][9]
    background = questions[index][11]
```

The background configuration remained as it was:

```
if background == 1:
    ## Import and convert background one and place it as background
    backgroundOneImage = Image.open('BackgroundOne.jpg')
    backgroundOneImage = backgroundOneImage.resize((1500,1000))
    tkBackgroundOneImage = ImageTk.PhotoImage(backgroundOneImage)
    backgroundImageLabel.configure(image = tkBackgroundOneImage)
    backgroundImageLabel.image = tkBackgroundOneImage
    backgroundImageLabel.place(relx = 0, rely = 0, relheight = 1, relwidth = 1)
elif background == 2:
    ## Import and convert background two and place it as background
    backgroundTwoImage = Image.open('BackgroundTwo.jpg')
    backgroundTwoImage = backgroundTwoImage.resize((1500,1000))
    tkBackgroundTwoImage = ImageTk.PhotoImage(backgroundTwoImage)
    backgroundImageLabel.configure(image = tkBackgroundTwoImage)
    backgroundImageLabel.image = tkBackgroundTwoImage
    backgroundImageLabel.place(relx = 0, rely = 0, relheight = 1, relwidth = 1)
elif background == 3:
    ## Import and convert background three and place it as background
    backgroundThreeImage = Image.open('BackgroundThree.jpg')
    backgroundThreeImage = backgroundThreeImage.resize((1500,1000))
    tkBackgroundThreeImage = ImageTk.PhotoImage(backgroundThreeImage)
    backgroundImageLabel.configure(image = tkBackgroundThreeImage)
    backgroundImageLabel.image = tkBackgroundThreeImage
    backgroundImageLabel.place(relx = 0, rely = 0, relheight = 1, relwidth = 1)
```

```
answers = [x for x in answers if x != None]
if len(answers) > 0:
    ## Configure and place question frame and label and configure answer one label
    questionLabel.config(text = selectedQuestion, bg = 'gray', font = titleFont, wraplength = 500)
    questionFrame.place(relx = 0.05, rely = 0.05, relheight = 0.25, relwidth = 0.9)
    questionLabel.place(relx = 0, rely = 0, relheight = 1, relwidth = 1)
    answerOneLabel.config(text = answers[0], bg = 'gray', wraplength = 100)
if len(answers) == 1:
    ## Place answer one frame and label for answer input questions (wider frame)
    answerOneFrame.place(relx = 0.15, rely = 0.35, relheight = 0.15, relwidth = 0.7)
    answerOneLabel.place(relx = 0, rely = 0, relheight = 1, relwidth = 1)
if len(answers) > 1:
    ## Configure and place answer one and two frame and label for multiple choice questions
    answerOneFrame.place(relx = 0.15, rely = 0.35, relheight = 0.15, relwidth = 0.3)
    answerOneLabel.place(relx = 0, rely = 0, relheight = 1, relwidth = 1)
    answerTwoLabel.config(text = answers[1], bg = 'gray', wraplength = 100)
    answerTwoFrame.place(relx = 0.55, rely = 0.35, relheight = 0.15, relwidth = 0.3)
    answerTwoLabel.place(relx = 0, rely = 0, relheight = 1, relwidth = 1)
if len(answers) > 2:
    ## Configure and place answer three frame and label
    answerThreeLabel.config(text = answers[2], bg = 'gray', wraplength = 100)
    answerThreeFrame.place(relx = 0.15, rely = 0.55, relheight = 0.15, relwidth = 0.3)
    answerThreeLabel.place(relx = 0, rely = 0, relheight = 1, relwidth = 1)
if len(answers) > 3:
    ## Configure and place answer four frame and label
    answerFourLabel.config(text = answers[3], bg = 'gray', wraplength = 100)
    answerFourFrame.place(relx = 0.55, rely = 0.55, relheight = 0.15, relwidth = 0.3)
    answerFourLabel.place(relx = 0, rely = 0, relheight = 1, relwidth = 1)
```

Here, I had to search for and remove all the None values from the answers list. These values may be present if the questions were retrieved from the database which had less than four options. I then configured and placed the different question and answer frames and labels as before. I left the correctAnswer and timeLimit sections of the code as they were.

To test it I created another quiz using my create quiz function.

Table: Quiz

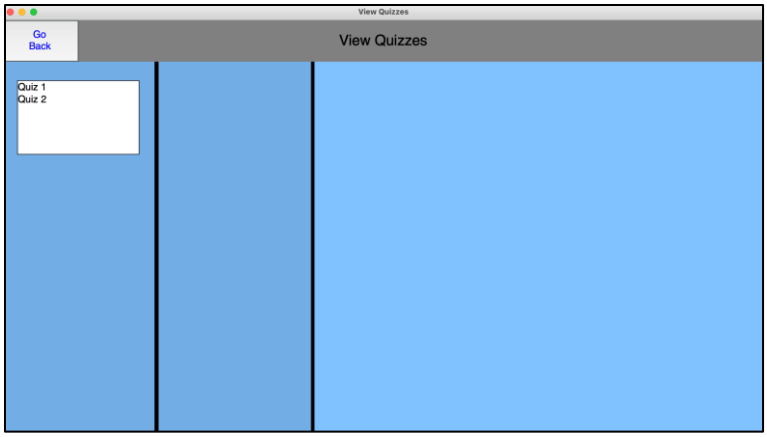
	quizID	subject	level	noOfQuestions
	Filter	Filter	Filter	Filter
1	19	Test	Test	2
2	20	Maths	KS1	3

Table: Question

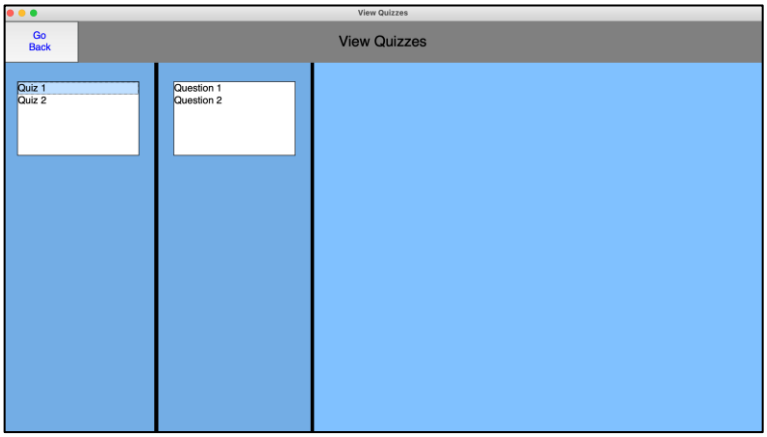
Filter in any colum

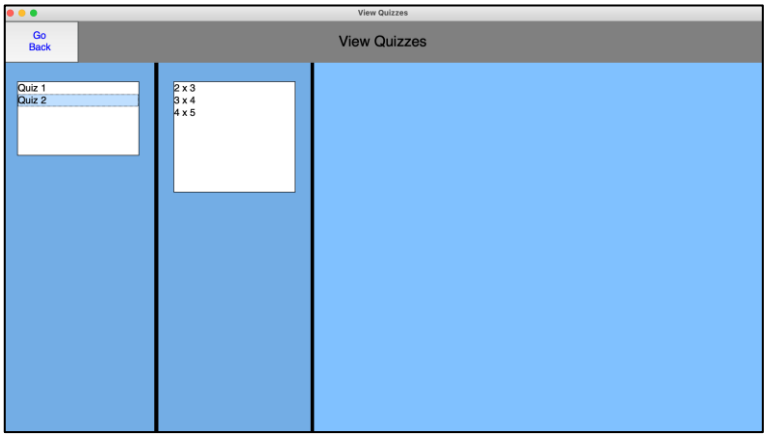
	questionID	quizID	questionType	question	noOfAnswers	answerOne	answerTwo	answerThree	answerFour	timeLimit	characterLimit	background
	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter
1	16	19	Multiple Choice	Question 1	2	test	test2	NULL	NULL	30	NULL	1
2	17	19	Answer Input	Question 2	1	test answer	NULL	NULL	NULL	15	30	2
3	18	20	Multiple Choice	2 x 3	2	4	6	NULL	NULL	30	NULL	1
4	19	20	Multiple Choice	3 x 4	3	12	10	14	NULL	30	NULL	2
5	20	20	Answer Input	4 x 5	1	20	NULL	NULL	NULL	30	30	3

When you first initiate the function:

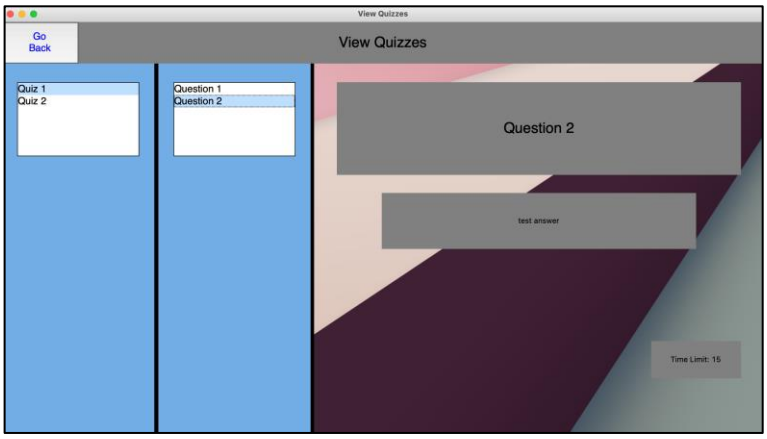


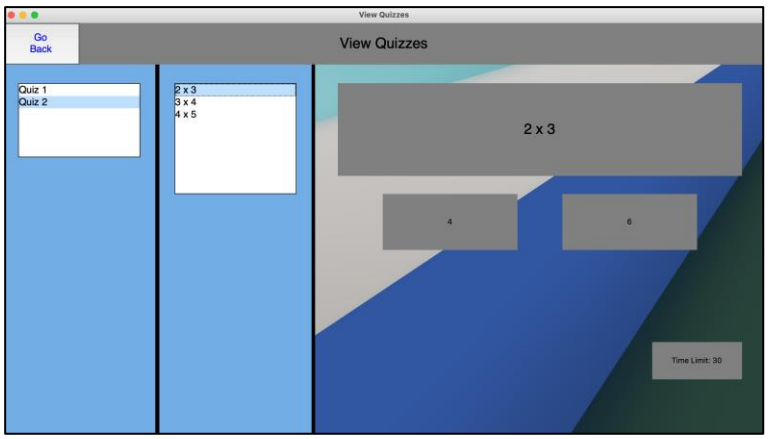
When selecting a quiz:

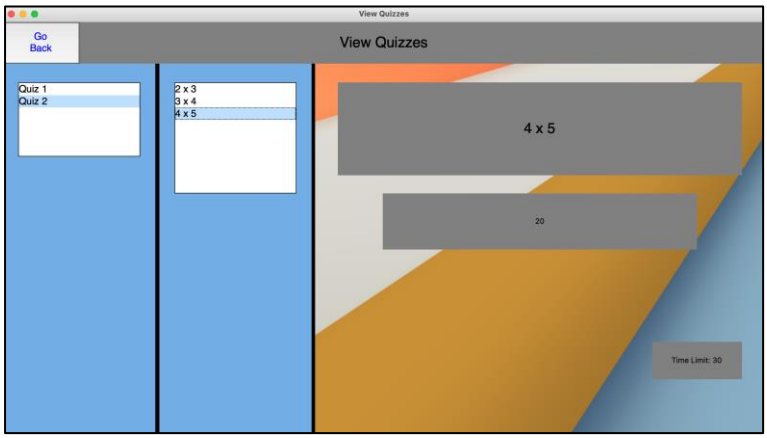




When selecting a question:



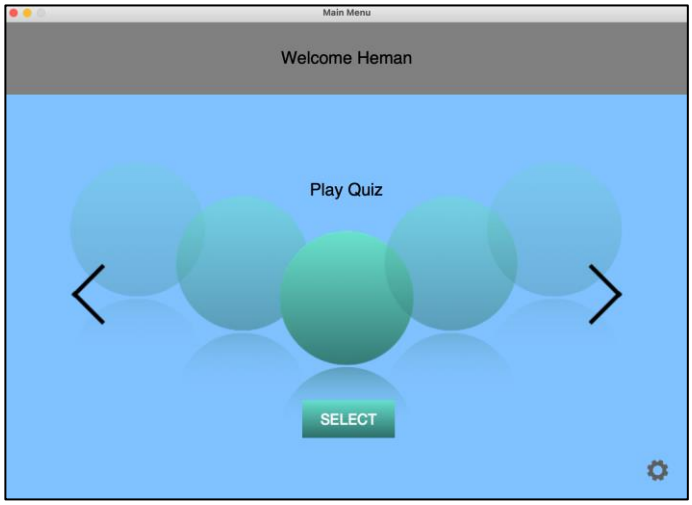




Initially when testing the selection of a question, an error occurred. This was because upon selecting a question from the question list box, the selection from the quiz list box was deselected. This caused issues since the program no longer recognised what information to work with. However this was fixed using:

```
quizListbox = tk.Listbox(quizFrame, font = defaultFont, exportselection = False)
```

By setting exportselection to False here, it would prevent deselection in this way. I therefore did this for all my list boxes within this function and the create quiz function.
When pressing the 'Go Back' button:



As desired.

When testing the functions together, from the main menu, I noticed an issue. This was with the create quiz function. Because I had set self.questions to contain [‘No Questions’] at the start of the function, when a question was created and it closes the page to open a new one, the questions list loses the question, and thus produces nothing. To combat this, I created a new parameter for the displayQuestion function named previousFunction, which in the create quiz function I passed in ‘Create Quiz’. I then modified the displayQuestion function to this:

```
## Check if class questions list is above 0
if previousFunction == 'Create Quiz':
    ## Find position of selected question
    for each in self.questions:
        if each[0] == selectedQuestion:
            try:
                answers = each[1]
                correctAnswer = each[2]
                timeLimit = each[3]
                background = each[4]
            except:
                answers = []
                correctAnswer = ''
                timeLimit = ''
                background = ''
elif previousFunction == 'View Quizzes':
    ## If retrieving from database
    index = questionListbox.get(0, "end").index(selectedQuestion)
    answers = [questions[index][5], questions[index][6], questions[index][7], questions[index][8]]
    correctAnswer = questions[index][5]
    timeLimit = questions[index][9]
    background = questions[index][11]
```

Which worked a lot more efficiently.

Play Quiz function:

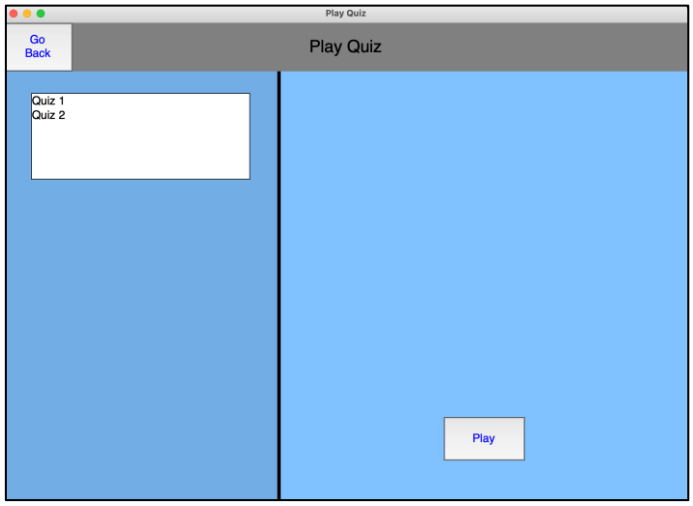
```
def playQuiz(self, username, titleFont, defaultFont, buttonFont):
    ## Retrieves the assigned height, width and fonts for the main create question page
    mainScreenWidth = mainMenu.getScreenWidth()
    mainScreenHeight = mainMenu.getScreenHeight()

    playQuizPage = tk.Tk()
    playQuizPage.title("Play Quiz")

    canvas = tk.Canvas(playQuizPage, height = mainScreenHeight, width = mainScreenWidth)
    canvas.pack()

    background = tk.Frame(playQuizPage, bg = 'black')
    background.place(rely = 0.1, relheight = 0.9, relwidth = 1)

    ## Create a frame for listbox of all quizzes on the left
    quizFrame = tk.Frame(playQuizPage, bg = '#73ade5')
    quizFrame.place(rely = 0.1, relheight = 0.9, relwidth = 0.4)

    ## Create a frame for details of all quizzes on the right and option to play
    mainframe = tk.Frame(playQuizPage, bg = '#80c1ff')
    mainframe.place(relx = 0.405, rely = 0.1, relheight = 0.9, relwidth = 0.595)

    welcomeLabel = tk.Label(canvas, text = "Play Quiz", bg = 'gray', font = titleFont)
    welcomeLabel.place(relheight = 0.1, relwidth = 1)

    returnButton = tk.Button(canvas, text = "Go\nBack", highlightbackground = '#424949', fg = 'blue', font = defaultFont, command = lambda: self
    returnButton.place(relheight = 0.1, relwidth = 0.1)                                                                    self.returnToMainMenu(playQuizPage, username))

    ## Create listbox for all quizzes
    quizListbox = tk.Listbox(quizFrame, font = defaultFont, exportselection = False)
    listboxHeight = 0
    buttonPos = 0.1
    quizzes = viewingFromDatabase.viewFromDatabase(c, True, True, None, None, None, 'Quiz')
    for i in range(len(quizzes)):
        quizListbox.insert('end', 'Quiz ' + str(i+1))
        if buttonPos < 0.8:
            listboxHeight += 0.1
            buttonPos += 0.1
    quizListbox.place(relx = 0.1, rely = 0.05, relheight = listboxHeight, relwidth = 0.8)
    ## Bind list box to displayQuizDetails function when a quiz is selected
    quizListbox.bind('<<ListboxSelect>>', lambda x: self.displayQuizDetails(quizListbox, quizListbox.get(quizListbox.curselection()), mainframe,
                                                                            quizSubjectFrame, quizSubjectLabel, quizLevelFrame, quizLevelLabel,
                                                                            noOfQuestionsFrame, noOfQuestionsLabel, Login.titleFont,
                                                                            Login.defaultFont, Login.buttonFont))

    ## Create frame and label for a quiz's subject
    quizSubjectFrame = tk.Frame(mainframe)
    quizSubjectLabel = tk.Label(quizSubjectFrame)

    ## Create frame and label for a quiz's level
    quizLevelFrame = tk.Frame(mainframe)
    quizLevelLabel = tk.Label(quizLevelFrame)

    ## Create frame and label for the number of questions in a quiz
    noOfQuestionsFrame = tk.Frame(mainframe)
    noOfQuestionsLabel = tk.Label(noOfQuestionsFrame)

    playButton = tk.Button(mainframe, text = "Play", highlightbackground = '#424949', fg = 'blue', font = defaultFont)
    playButton.place(relx = 0.4, rely = 0.8, relheight = 0.1, relwidth = 0.2)

    playQuizPage.mainloop()
```

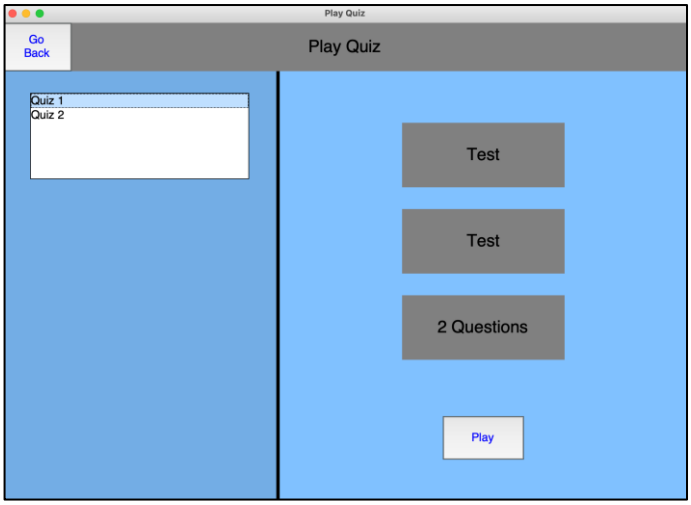
In this function, I create a page that has a frame on the left for the quizzes, as in the view quizzes function. However, I don't show the questions of the quiz. Instead I show the details of the quiz using the pre-created frames and labels shown above and this function, which binds to the list box for any selection made:

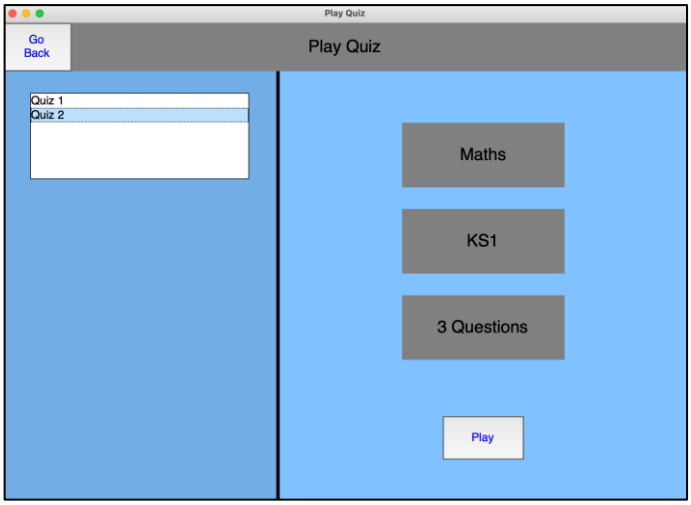
```
def displayQuizDetails(self, quizListbox, selectedQuiz, mainframe, quizSubjectFrame, quizSubjectLabel, quizLevelFrame, quizLevelLabel, noOfQuestionsFrame,
    ## Retrieve all details from database for selected quiz
    index = quizListbox.get(0, "end").index(selectedQuiz)
    quizDetails = viewingFromDatabase.viewFromDatabase(c, True, True, None, None, None, "Quiz LIMIT " + str(index) + ", 1")

    quizSubjectFrame.place(relx = 0.3, rely = 0.12, relheight = 0.15, relwidth = 0.4)
    quizSubjectLabel.config(text = quizDetails[0][1], bg = 'gray', font = titleFont)
    quizSubjectLabel.place(relheight = 1, relwidth = 1)
```

```
quizLevelFrame.place(relx = 0.3, rely = 0.32, relheight = 0.15, relwidth = 0.4)
quizLevelLabel.config(text = quizDetails[0][2], bg = 'gray', font = titleFont)
quizLevelLabel.place(relheight = 1, relwidth = 1)

noOfQuestionsFrame.place(relx = 0.3, rely = 0.52, relheight = 0.15, relwidth = 0.4)
noOfQuestionsLabel.config(text = str(quizDetails[0][3]) + ' Questions', bg = 'gray', font = titleFont)
noOfQuestionsLabel.place(relheight = 1, relwidth = 1)
```

In this function, much like before it finds out the row that has been selected, and using that, retrieves details from the database. This time, however, all details are retrieved from the Quiz table. Using these details, I can now configure the text, font and background of the labels, and place both the frames and the labels accordingly. From the same quizzes stored in the database as before, this produced:







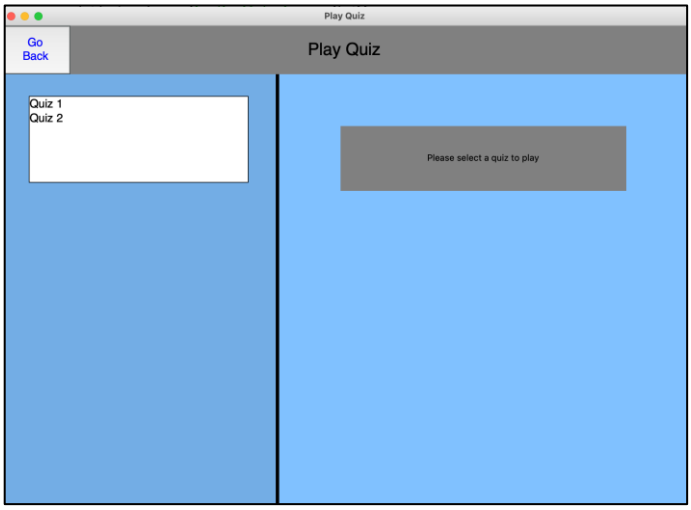
Before creating the function for the play button, I noticed an opportunity for error here: if the user presses the play quiz before having selected a quiz, the program would not have had the data for any quiz. Hence, I moved the placement of the play button code within the displayQuizDetails function.

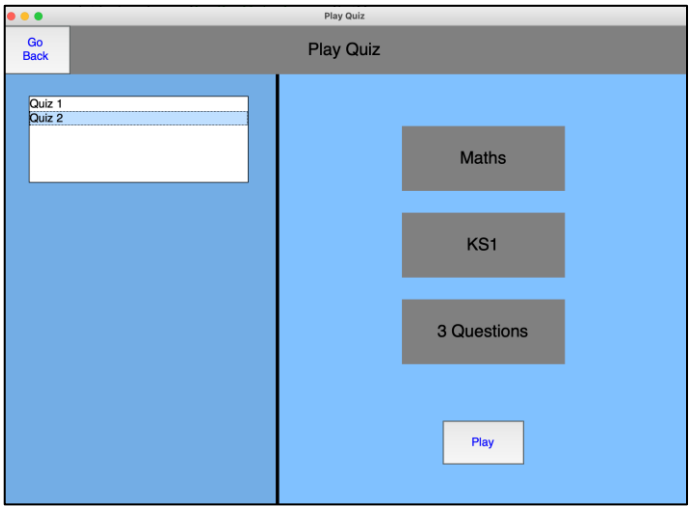
```
playButton.config(command = lambda: self.playSelectedQuiz(playQuizPage, quizDetails))
playButton.place(relx = 0.4, rely = 0.8, relheight = 0.1, relwidth = 0.2)
```

I also configured it to command it to initiate the playSelectedQuiz function. By placing it in the displayQuizDetails function in this way, it also prevented any repetition in retrieiving the details of the quiz from the database, making the code more efficient.

I also extended the playQuiz function to ask the user to select a quiz, using the quiz subject frame and label.

```
## Create frame and label for a quiz's subject
quizSubjectFrame = tk.Frame(mainframe)
quizSubjectLabel = tk.Label(quizSubjectFrame, text = 'Please select a quiz to play', bg = 'gray')
quizSubjectFrame.place(relx = 0.15, rely = 0.12, relheight = 0.15, relwidth = 0.7)
quizSubjectLabel.place(relheight = 1, relwidth = 1)
```





I then created a playSelectedQuiz function:

```
def playSelectedQuiz(self, previousPage, quizDetails):
    ## Destroy previous page
    previousPage.destroy()
    ## Retrieve all questions from database for selected quiz
    questions = viewingFromDatabase.viewFromDatabase(c, True, False, None, 'quizID', str(quizDetails[0][0]), 'Question')
    print(questions)

    ## Retrieves the assigned height, width and fonts for the main create question page
    mainScreenWidth = mainMenu.getScreenWidth()
    mainScreenHeight = mainMenu.getScreenHeight()

    playSelectedQuizPage = tk.Tk()
    playSelectedQuizPage.title("Play Quiz")

    canvas = tk.Canvas(playSelectedQuizPage, height = mainScreenHeight, width = mainScreenWidth)
    canvas.pack()

    mainframe = tk.Frame(playSelectedQuizPage, bg = '#80c1ff')
    mainframe.place(relheight = 1, relwidth = 1)

    backgroundImageLabel = tk.Label(mainframe)
    questionFrame = tk.Frame(mainframe)
    questionLabel = tk.Label(questionFrame)
    answerOneFrame = tk.Frame(mainframe)
    answerOneLabel = tk.Label(answerOneFrame)
    answerTwoFrame = tk.Frame(mainframe)
    answerTwoLabel = tk.Label(answerTwoFrame)
    answerThreeFrame = tk.Frame(mainframe)
    answerThreeLabel = tk.Label(answerThreeFrame)
    answerFourFrame = tk.Frame(mainframe)
    answerFourLabel = tk.Label(answerFourFrame)
    timeLimitFrame = tk.Frame(mainframe)
    timeLimitLabel = tk.Label(timeLimitFrame)

    for i in range(len(questions)):
        timeLimit = questions[i][9]
        self.displayQuestion('Play Quiz', playSelectedQuizPage, None, questions, questionLabel, i, questionFrame, answerOneFrame, answerOneLabel, answerTwoFrame,
                             answerTwoLabel, answerThreeFrame, answerThreeLabel, answerFourFrame, answerFourLabel, backgroundImageLabel,
                             timeLimitFrame, timeLimitLabel, mainframe, Login.titleFont, Login.defaultFont, Login.buttonFont)

    playSelectedQuizPage.mainloop()
```

This function first destroys the previous page, and retrieves all the questions for the chosen quiz from the database. I then created a page, frames and labels that allowed me to use the displayQuestion function (with some modification) to display the questions, in turn.

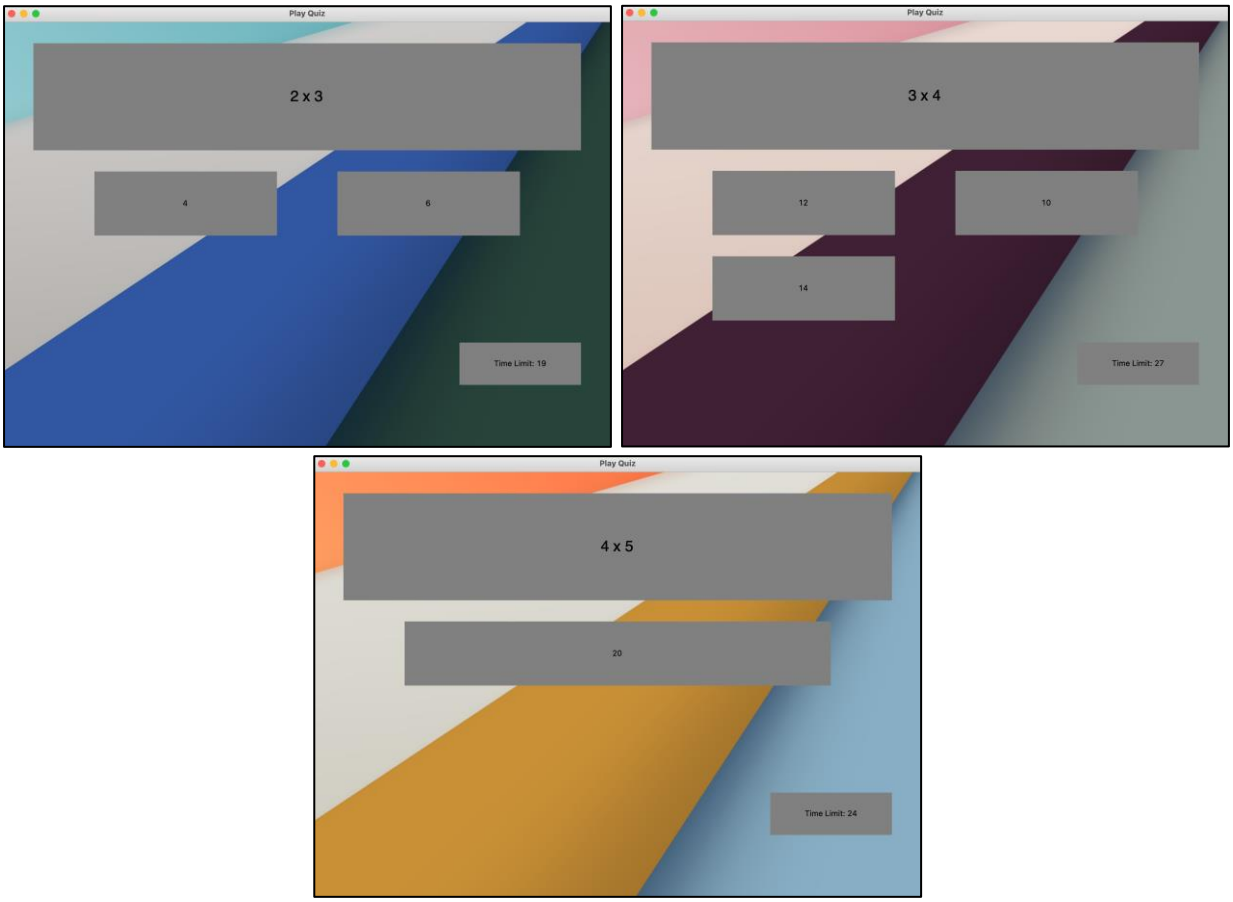
Within the displayQuestion function, I extended it to be compatible for when the previousFunction parameter was ‘Play Quiz’:

```
elif previousFunction == 'Play Quiz':
    ## Retrieve question details from list (where selectedQuestion is the round of the for loop)
    answers = [questions[selectedQuestion][5], questions[selectedQuestion][6], questions[selectedQuestion][7], questions[selectedQuestion][8]]
    correctAnswer = questions[selectedQuestion][5]
    timeLimit = questions[selectedQuestion][9]
    background = questions[selectedQuestion][11]
    selectedQuestion = questions[selectedQuestion][3]
```

I first extended it to retrieve all the relevant question details from the questions list that had been passed into the function from the play quiz function. This allowed me to keep the configuration of the frames and labels the same as they were previously. I also extended the time limit section of the code:

```
if timeLimit != '':
    timeLimitLabel.config(text = 'Time Limit: ' + str(timeLimit), bg = 'gray')
    timeLimitFrame.place(relx = 0.75, rely = 0.75, relheight = 0.1, relwidth = 0.2)
    timeLimitLabel.place(relx = 0, rely = 0, relheight = 1, relwidth = 1)
    if previousFunction == 'Play Quiz':
        ## while time limit is greater than 0 (i.e. there is time remaining)
        while timeLimit > 0:
            timeLimit -= 1
            time.sleep(1)
            timeLimitLabel.config(text = 'Time Limit: ' + str(timeLimit))
            playQuizPage.update()
```

This allowed firstly for a page to be displayed for a length of time determined by the time limit, and then once finished, go on to the next. It also allowed the users to see a live countdown of the time limit so they can see how long they have left to answer the question. From Quiz 2:

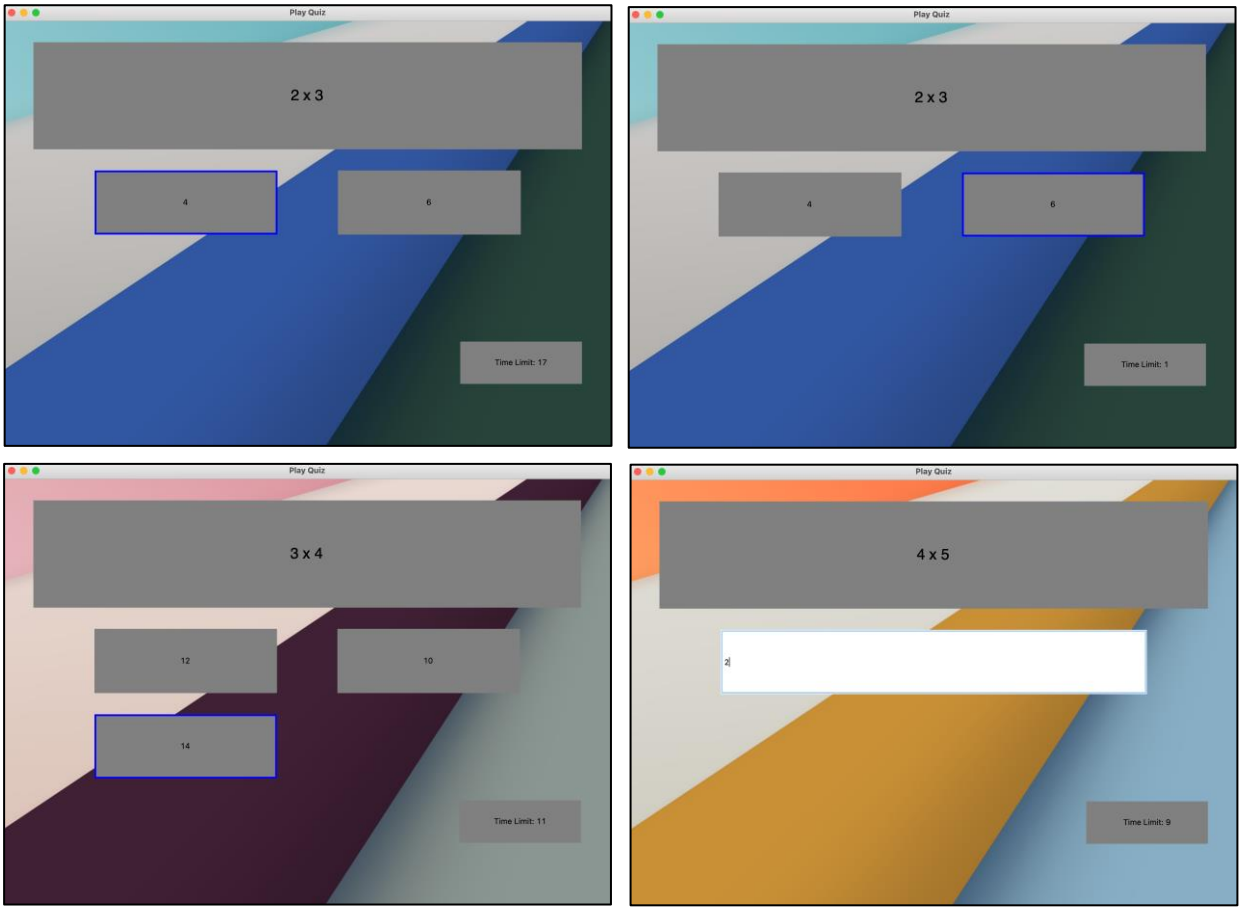


So far, this only provided the options for labels, and didn't yet allow the user to input any answer of their own. I hence modified the displayQuestion function:

```
if previousFunction == 'Play Quiz':
        ## If answer input question...
        if len(answers) == 1:
            ## Create entry
            answerOneEntry.place(relx = 0, rely = 0, relheight = 1, relwidth = 1)
        ## If multiple choice question...
        else:
            ## Bind answer one label to highlightOption function
            answerOneLabel.bind("<Button-1>", lambda x: self.highlightOption(answerOneFrame, answerOneFrame, answerTwoFrame, answerThreeFrame, answerFourFram
        ## Bind all other answer labels to highlightOption function
        answerTwoLabel.bind("<Button-1>", lambda x: self.highlightOption(answerTwoFrame, answerOneFrame, answerTwoFrame, answerThreeFrame, answerFourFrame))
        answerThreeLabel.bind("<Button-1>", lambda x: self.highlightOption(answerThreeFrame, answerOneFrame, answerTwoFrame, answerThreeFrame, answerFourFram
        answerFourLabel.bind("<Button-1>", lambda x: self.highlightOption(answerFourFrame, answerOneFrame, answerTwoFrame, answerThreeFrame, answerFourFrame)
```

And created this function:

```
def highlightOption(self, selectedFrame, answerOneFrame, answerTwoFrame, answerThreeFrame, answerFourFrame):
    ## Remove border from all options (in case of previous selection)
    answerOneFrame.config(highlightthickness = 0)
    answerTwoFrame.config(highlightthickness = 0)
    answerThreeFrame.config(highlightthickness = 0)
    answerFourFrame.config(highlightthickness = 0)
    ## Add blue border on chosen frame
    selectedFrame.config(highlightthickness = 3, highlightbackground = 'blue')
```



So far, the program hasn't a way of registering the input of the user although they are able to select or write one.
I therefore extended the highlightOption function:

```
## Find text of label within selected frame
for child in selectedFrame.winfo_children():
    answerInput = child.cget('text')
```

Here, the program finds the children of the selected frame, i.e. the answer label. It then proceeds to assign the text of the label to an answer input variable. This caused issues when deciding how to store this input, since it would change for each question and would be lost if not immediately dealt with. I considered adding this input to the database at this stage but realised the constraints this provided for the user in terms of changing answers. It could lead to the possibility of multiple answer inputs in the database for one question. As it is, this method of retrieving the answer would only be acceptable for multiple choice questions since any answer input or order answers questions would not be directed to the highlightOption function. Another option I could've explored was adding a record to the Progress table of the database with missing values, and for every chosen answer, the program updates the record rather than adding a new one. This could potentially make the program slower though since for every change of answer, the database would need to be updated. The final option I considered was adding a new record to the database for a question once the time limit had reached zero. This posed some difficulties however since the answerInput variable in this function couldn't be passed back to the previous function due to the formatting issues with the event bind, i.e. this was unacceptable:

```
answerinput = answerOneLabel.bind("<Button-1>", la
print(answerinput)
```

Since it would output:

```
140485300809664<lambda>
```

I therefore chose to take the second option mentioned above where I added a record to the Progress table of the database within the playSelectedQuiz function.
To do this I had to create a record of all the relevant values that were to be entered into the database. One of these values was the time field which would store the time of answering the question. To do this, I had to import the datetime library:

```
from datetime import datetime
```

I then extended the play quiz functions. I firstly passed the username from the playQuiz function to the displayQuizDetails function and the playSelectedQuiz function. This was so I could retrieve the corresponding user ID from the database. I then modified the playSelectedQuiz function:

```
## Establish userID for given username
userID = viewingFromDatabase.viewFromDatabase(c, False, False, 'userID', 'username', "'"+username+"'", 'UserDetails')
userID = userID[0][0]

## For each question in quiz
for i in range(len(questions)):
    ## Retrieve time limit from questions list
    timeLimit = questions[i][9]
    ## Establish current time
    currentTime = datetime.now()
    formattedDate = currentTime.strftime('%Y-%m-%d %H:%M:%S')
    ## Create record to be added to database
    record = [None, userID, quizDetails[0][0], questions[i][0], None, formattedDate]
    ## Add incomplete record to Progress table in database for the question
    addingToDatabase.addToDatabase(conn, c, record, 'Progress')
    ## Go to display question function with relevant parameters
    self.displayQuestion('Play Quiz', answerOneEntry, playSelectedQuizPage, None, questions, questionLabel, i, questi
```

Here I first retrieved the user ID with the given username. I then included, within the for loop, a variable that held the current date and time, using the datetime library now imported, and used the strftime function to write it in an appropriate format for SQL. I then created a record for the Progress table, which had None as its id since it would automatically be incremented by SQL, the user ID, the quiz ID, retrieved from the quizDetails list previously passed in as a parameter, the question ID retrieved from the questions list, None for the 'correct' field since it had not yet been

determines and finally the formatted date and time variable just created. I then used my database class to add this record to the 'Progress' table. This was done for each question. Upon playing Quiz 2 shown above, this was the result in the database:

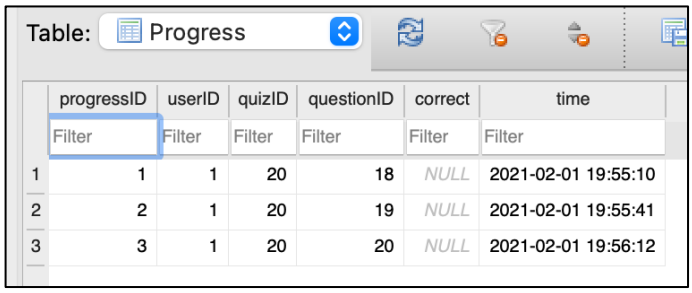


Table: Progress

	progressID	userID	quizID	questionID	correct	time
	Filter	Filter	Filter	Filter	Filter	Filter
1	1	1	20	18	NULL	2021-02-01 19:55:10
2	2	1	20	19	NULL	2021-02-01 19:55:41
3	3	1	20	20	NULL	2021-02-01 19:56:12

When testing the playQuiz, playSelectedQuiz, displayQuestion and highlightOption functions together, I also noticed that once a user has selected an option, and the next question appears, the highlight from the previous question remained. Hence, I added this bit of code:

```
answerOneFrame.place_forget()
answerOneFrame.config(highlightthickness = 0)
answerTwoFrame.place_forget()
answerTwoFrame.config(highlightthickness = 0)
answerThreeFrame.place_forget()
answerThreeFrame.config(highlightthickness = 0)
answerFourFrame.place_forget()
answerFourFrame.config(highlightthickness = 0)
```

At the beginning of the displayQuestion function, so for every question, the highlight was reset by setting the thickness to 0.

I then passed on the correct answer variable from the displayQuestion function to the highlightOption function. When testing this function again, I noticed an inconsistency. Where I retrieved the text from answer labels two, three or four, the answerInput variable was shown as desired. However, upon choosing the first option, a blank variable was displayed. To investigate I printed the children of each frame to see where the issue was:

```
.!frame.!frame3.!label
6
.!frame.!frame2.!label
.!frame.!frame2.!entry
```

On doing so, I realised that the answer one frame held not only the answer label, but the answer entry for answer input questions. Since the entry was blank and occurred after the label, the answer input variable was blank.

To combat this I swapped the position of where I declared the entry and label in the playSelectedQuiz function:

```
answerOneFrame = tk.Frame(mainframe)
answerOneEntry = tk.Entry(answerOneFrame)
answerOneLabel = tk.Label(answerOneFrame)
```

By declaring the label second, within the highlightOption function, the label would be the last thing checked for text.
This, however, caused an issue when playing the quiz, since now the label would be displayed on top of the entry in answer input questions. To combat this, I therefore added to the displayQuestion function:

```
if previousFunction == 'Play Quiz':
    ## If answer input question...
    if len(answers) == 1:
        ## Hide answer label
        answerOneLabel.place(relheight = 0, relwidth = 0)
        ## Create entry
        answerOneEntry.place(relx = 0, rely = 0, relheight = 1, relwidth = 1)
```

To reduce the size of the label to 0.

To finalise the answer checking process, I changed the previous decision of passing the correct answer to the highlightOption function and retrieving the text of the selected frame. This was because this would still require me to extend other parts of the program to separately check for multiple choice and order answers question. Therefore, extending the code I previously had in the highlightOption function, I extended the displayQuestion function:

```
## When time has finished
if timeLimit == 0:
    ## Create variable for user's input for the answer
    answerInput = ''
    ## For each widget in the mainframe of the play quiz page
    for child in rightFrame.winfo_children():
        ## Check thickness of the widget to see if 3
        if child.cget('highlightthickness') == 3:
            ## If 3, find children of the given frame
            for child1 in child.winfo_children():
                ## Assign answer from retrieving text from that frame
                answerInput = child1.cget('text')
            print('input: '+answerInput)
```

Here, the program waits for the time limit to reach 0, when it then creates a variable for the input for the answer from the user. It is initially set to '' so that if a user doesn't choose an option at all, a blank answer is given. It then proceeds to check the children of the mainframe (named rightframe as a parameter). This is so it can identify any frames with a border highlight. This would be the answer that the user has selected. By doing this when the time reaches 0, it prevents the program having to constantly update the database when a new option has been selected, saving time. The function then goes on to check the thickness of the highlight on each frame to see if it is 3 (as set when an option is selected) and proceeds to find the children of that widget (which will be the frame of the option). Finally the children of that frame are checked and the text within the included label is assigned to an answerInput variable as before. I displayed this variable to test out this function.

```
input: 4
input: 10
```

This was the result when playing Quiz 2 shown previously when selecting the first option for question 1 and the second option for question 2. I then extended the function further to gain inputs for answer input questions:

```
if len(answers) == 1:
    answerInput = answerOneEntry.get()
else:
    ## For each widget in the mainframe of the play quiz page
    for child in rightFrame.winfo_children():
        ## Check thickness of the widget to see if 3
        if child.cget('highlightthickness') == 3:
            ## If 3, find children of the given frame
            for child1 in child.winfo_children():
                ## Assign answer from retrieving text from that frame
                answerInput = child1.cget('text')
print('input: '+answerInput)
```

Here, if the length of the answers list is 1, the program recognises the question as an answer input question, and then assigns the input from the answer one entry to the answer input variable.

When selecting the second option for the first question, the third option for the second question and entering '20' for the third question, it produced:

```
input: 6
input: 14
input: 20
```

As desired.

Check answer function:

```
def checkAnswer(self, answerInput, correctAnswer):
    if answerInput == correctAnswer:
        return True
    else:
        return False
```

This function simply takes in the input of the user and the correct answer, and compares the two to return True if they match and False if they don't.

I then had retrieve the progress ID of the record I wished to add to:

```
progressID = viewingFromDatabase.viewFromDatabase(c, False, True, 'progressID', None, None, 'Progress ORDER BY progressID DESC LIMIT 1')
print(progressID)
```

The extension to the table instructed SQL to order the table by its progress ID field and then retrieve the first value of the table in descending order, or in other words, the last record to be inputted to the table. I specified the field to retrieve only the progressID of this field. When testing this on Quiz 2:

```
[(72,)]
[(73,)]
[(74,)]
```

Finally, I added a bit of code, using my database class to update the relevant record for each question after checking the answer.

```
updatingDatabase.updateInDatabase(conn, c, 'correct', str(self.checkAnswer(answerInput, correctAnswer)), 'progressID', "'"+str(progressID[0][0])+"'", 'Progress')
```

Here I passed in the values 'correct' for the field, 'Progress' for the table and 'progressID' for the record primary key. I then passed in the result of the check answer function when passing in the user input for the question and the correct answer and converted it to a string value to allow for concatenation. SQL is able to deal with the string value (the corresponding data type for the 'correct' field is boolean). I also passed in the progress ID just retrieved.

When answering Quiz 2, where the answer for the first question was 4, the second question 10 and the third question 20, it produced:

Table: Progress

	progressID	userID	quizID	questionID	correct	time
	Filter	Filter	Filter	Filter	Filter	Filter
1	77	1	20	18	True	2021-02-03 17:41:50
2	78	1	20	19	False	2021-02-03 17:42:01
3	79	1	20	20	True	2021-02-03 17:42:12

As desired.

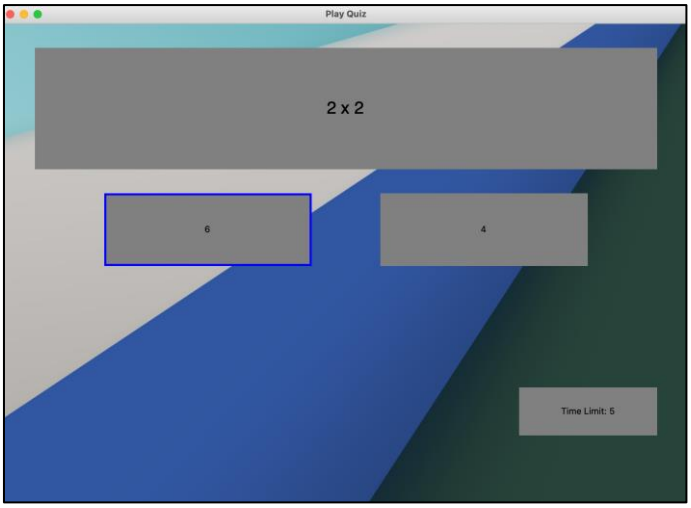
Although the play quiz function along with all its corresponding function now works, there is a limitation. As of yet, I haven't shuffled the answers in the answer list, meaning that for all multiple choice questions, the correct answer will always be the first option. I therefore extended the displayQuestion function. To do this, I had to import the random library:

```
import random
```

And add to the displayQuestion function:

```
elif previousFunction == 'Play Quiz':
    ## Retrieve question details from list (where selectedQuestion is the round of the for loop)
    answers = [questions[selectedQuestion][5], questions[selectedQuestion][6], questions[selectedQuestion][7], questions[selectedQuestion][8]]
    ## Shuffle answers list
    random.shuffle(answers)
    correctAnswer = questions[selectedQuestion][5]
```

This produced:



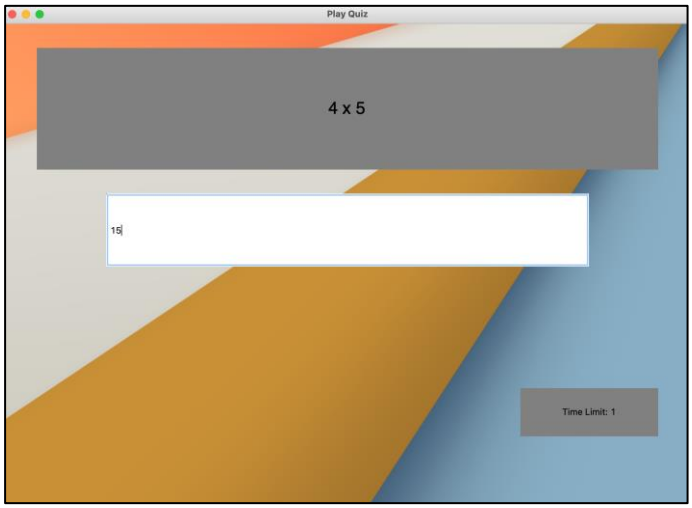


Table: Progress

	progressID	userID	quizID	questionID	correct	time
	Filter	Filter	Filter	Filter	Filter	Filter
1	85	1	20	18	False	2021-02-03 18:04:24
2	86	1	20	19	True	2021-02-03 18:04:35
3	87	1	20	20	False	2021-02-03 18:04:46

I now created a page which would appear at the end of every question to show the user's input, the correct answer and if they were correct or incorrect. First, I assigned the result of the checkAnswer function to a 'correct' variable and returned this value, along with the user input for the answer and the correct answer.

```
        correct = self.checkAnswer(answerInput, correctAnswer)
        updatingDatabase.updateInDatabase(conn, c, 'correct', str(correct), 'progressID', "'"+str(progressID[0][0])+"'", 'Progress')

return answerInput, correctAnswer, correct
```

I then modified the playSelectedQuiz function to retrieve these values:

```
answerInput, correctAnswer, correct = self.displayQuestion('Play Quiz', answ
```

I was now able to create a new finishQuestion function.

First, I had to introduce the points system.

```
points = 0
```

I declared a points variable within the playSelectedQuiz function and assigned the value 0 to it. This allowed me to pass it on as a parameter to the finishQuestion function and add points accordingly if needed.

The finishQuestion function:

```
def finishQuestion(self, answerInput, correctAnswer, correct, points, titleFont, buttonFont, defaultFont):
    ## Retrieves the assigned height, width and fonts for the finish question page
    mainScreenWidth = mainMenu.getScreenWidth()
    mainScreenHeight = mainMenu.getScreenHeight()

    ## Create page and title
    finishQuestionPage = tk.Tk()
    finishQuestionPage.title("Play Quiz")

    ## Create canvas for page dimensions
    canvas = tk.Canvas(finishQuestionPage, height = mainScreenHeight, width = mainScreenWidth)
    canvas.pack()

    ## If answered correctly, assign green background and text to 'Correct!'
    if correct == True:
        bgColour = '#9FE2BF'
        correctText = 'Correct!'
        points+=1000
    ## else, assign red background and text to 'Incorrect!'
    else:
        bgColour = '#F08080'
        correctText = 'Incorrect!'
    ## Create main frame with correct background colour
    mainframe = tk.Frame(finishQuestionPage, bg = bgColour)
    mainframe.place(relheight = 1, relwidth = 1)

    ## Create label to show if correct or incorrect
    correctLabel = tk.Label(mainframe, text = correctText, bg = 'gray', font = titleFont)
    correctLabel.place(relheight = 0.15, relwidth = 1)
```

```
    ## If answered incorrectly...
    if correct == False:
        ## Create label to show user's input
        answerInputLabel = tk.Label(mainframe, text = 'You entered ' + answerInput, bg = 'gray', font = titleFont)
        answerInputLabel.place(relx = 0.3, rely = 0.25, relheight = 0.12, relwidth = 0.4)

        ## Create label to show correct answer
        correctAnswerLabel = tk.Label(mainframe, text = 'The correct answer is ' + correctAnswer, bg = 'gray', font = titleFont)
        correctAnswerLabel.place(relx = 0.3, rely = 0.42, relheight = 0.12, relwidth = 0.4)

    ## Create label to show how many points the user is on
    pointsLabel = tk.Label(mainframe, text = 'Points: ' + str(points), bg = 'gray', font = titleFont)
    pointsLabel.place(relx = 0.3, rely = 0.59, relheight = 0.12, relwidth = 0.4)

    ## Set time limit to 10 seconds
    timeLimit = 10
    ## Create label to show how long until the next question begins
    nextQuestionLabel = tk.Button(mainframe, text = 'Next Question in: ' + str(timeLimit), highlightbackground = '#424949', font = defaultFont)
    nextQuestionLabel.place(relx = 0.4, rely = 0.77, relheight = 0.12, relwidth = 0.2)

    ## While time limit is greater than 0 (i.e. there is time remaining)
    while timeLimit > 0:
        timeLimit -= 1
        time.sleep(1)
        nextQuestionLabel.config(text = 'Next Question in: ' + str(timeLimit))
        finishQuestionPage.update()

    ## Destroy page at the end of countdown
    finishQuestionPage.destroy()

    return points
    finishQuestionPage.mainloop()
```

This function creates a new page. The background of this page is dependent on whether the ‘correct’ variable is True or False, creating a green background for when True and a red background for when False (inspired by Kahoot). This if statement also allocates the text for a label at the top of the page that says either ‘Correct!’ or ‘Incorrect!’. If the correct variable is found to be True, 1000 points are

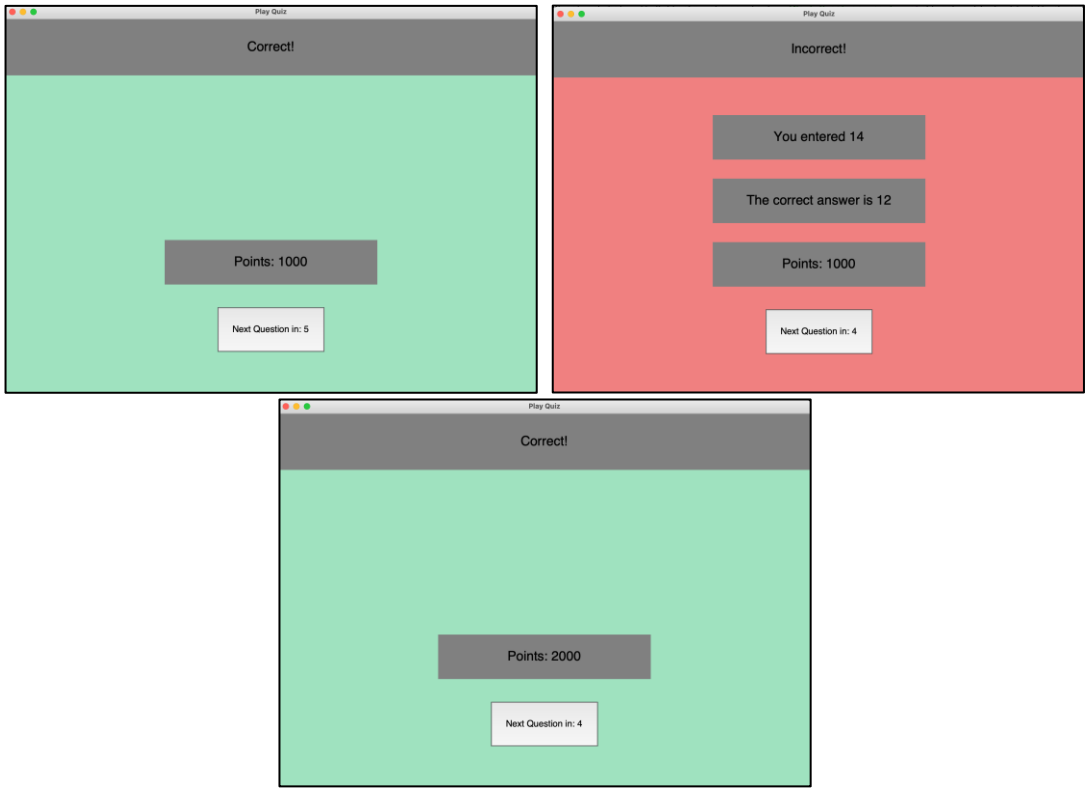
also added to the points. After this, I then created labels if the correct variable is False for the user answer input and the correct answer. I created another label outside this if statement for the number of points the user is on. I then set a time limit of 10, and used a while loop for the program to count down from this time limit and display it to the user by updating the page after every second. After 10 seconds, the finishQuestionPage is destroyed and the next question is presented.

To make this work, I had to also modify my playSelectedQuiz function:

```
## For each question in quiz
for i in range(len(questions)):
    ## Show playSelectedQuizPage
    playSelectedQuizPage.deiconify()
    ## Retrieve time limit from questions list
    timeLimit = questions[i][9]
    ## Establish current time
    currentTime = datetime.now()
    formattedDate = currentTime.strftime('%Y-%m-%d %H:%M:%S')
    ## Create record to be added to database
    record = [None, userID, quizDetails[0][0], questions[i][0], None, formattedDate]
    ## Add incomplete record to Progress table in database for the question
    addingToDatabase.addToDatabase(conn, c, record, 'Progress')
    ## Go to display question function with relevant parameters
    answerInput, correctAnswer, correct = self.displayQuestion('Play Quiz', answerOneEntry, playSelectedQuizPage, None, questions, c
    ## Hide playSelectedQuizPage
    playSelectedQuizPage.withdraw()
    ## Initiate finishQuestion and assign returned value to points variable
    points = self.finishQuestion(answerInput, correctAnswer, correct, points, Login.titleFont, Login.buttonFont, Login.defaultFont)
```

The first change I made was retrieving the answerInput, correctAnswer and correct variables from the displayQuestion function. This was so I could pass it into my finishedQuestion function later on as shown above. I also modified it to retrieve the points from the finishedQuestion function, so it wouldn't reset after every question. Finally I used the withdraw() function on the playSelectedQuizPage after the time limit for a question was over to hide it so I could then display the finishQuestionPage. Because of this, I had to re-show the page after every question so I added that at the beginning of the loop.

Playing Quiz 2, and inputting 4, then 14 then 20:



When designing my program, I created a point system which included a streak of correct answers and points associated to this streak. To implement this, I created a multiplier variable in my playSelectedQuiz function:

```
multiplier = 1
```

Within my for loop in the same function, but before initiating the displayQuestion function and changing the value of the 'correct' variable, I wrote:

```
## Determine streak and multiplier
if correct == True:
    multiplier += 0.1
else:
    multiplier = 1
```

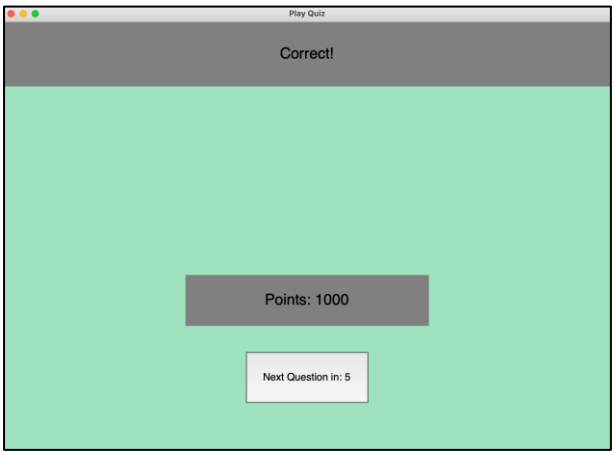
I noticed, however, this would cause an error for the first question since the 'correct' variable is declared after this point. Therefore, outside the for loop, I wrote:

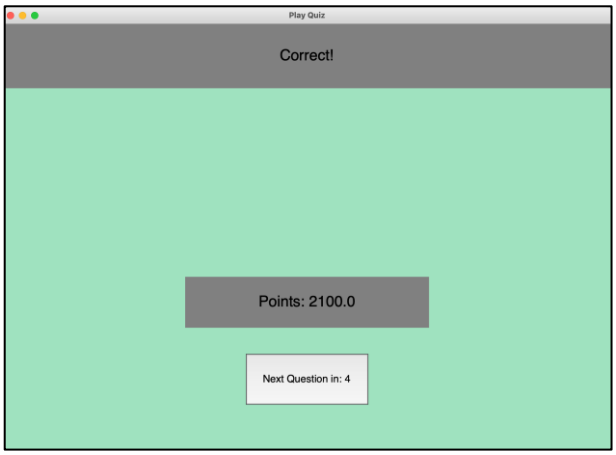
```
correct = False
```

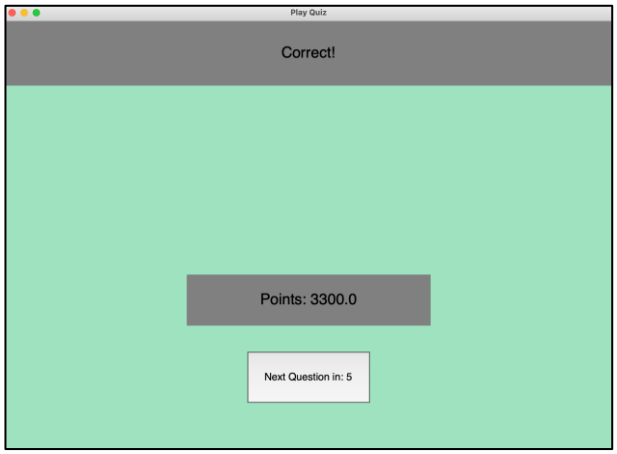
I then passed the multiplier as a parameter to the finishQuestion function and modified the code concerned with allotting points:

```
points += (1000*multiplier)
```

This now meant for every consecutive question the user answered correctly, they would receive an extra 100 points than the amount they received for the last question. For example, if a user answered every question correctly in Quiz 2, they would see:







Before creating a page for the end of the quiz, I modified the code slightly so that for the last question, instead of saying 'Next Question in:' it would say 'Finish in:'. To do this, I passed in the value of i and the length of the questions list to the finishQuestion function as noOfQuestions:

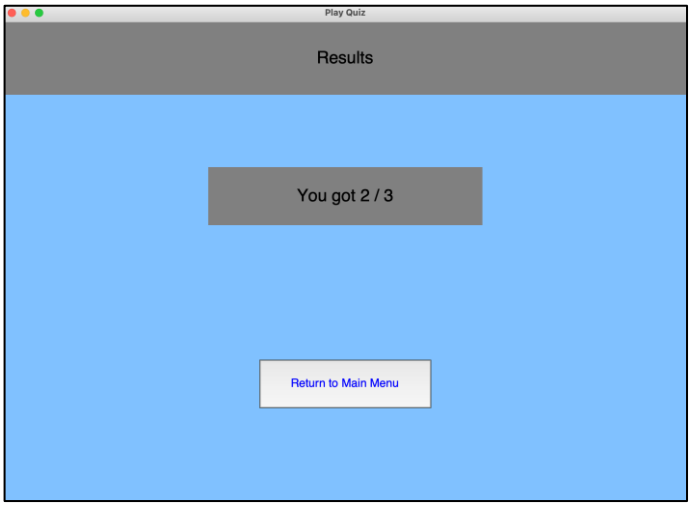
```
## Assign label text for when not the last question
labelText = 'Next Question in: '
## If last iteration...
if i == noOfQuestions-1:
    ## Assign label text for when the last question
    labelText = 'Finish in: '
## While time limit is greater than 0 (i.e. there is time remaining)
while timeLimit > 0:
    timeLimit -= 1
    time.sleep(1)
    nextQuestionLabel.config(text = labelText + str(timeLimit))
    finishQuestionPage.update()
```

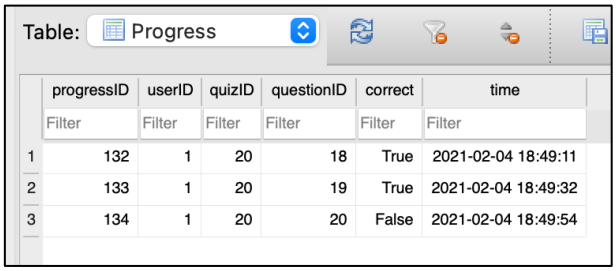
Here, I assigned the string 'Next Question in:' to a labelText variable. I then checked using an if statement the value of i was one less than the value of the number of questions in the quiz.

Next, I created a page to show the user the quiz has ended, their results and an option to return to the main menu.

```
def finishQuiz(self, username, noOfQuestions, titleFont, buttonFont, defaultFont):
    ## Retrieve 'correct' field from progress records from database for the quiz played
    progress = viewingFromDatabase.viewFromDatabase(c, False, True, 'correct', None, None, 'Progress ORDER BY progressID DESC LIMIT ' + str(noOfQuestions))
    correctAnswers = 0
    ## For each correct answer...add one to counter
    for each in progress:
        if each[0] == 'True':
            correctAnswers += 1

    ## Retrieves the assigned height, width and fonts for the finish quiz page
    mainScreenWidth = mainMenu.getScreenWidth()
    mainScreenHeight = mainMenu.getScreenHeight()

    ## Create page and title
    finishQuizPage = tk.Tk()
    finishQuizPage.title("Play Quiz")

    ## Create canvas for page dimensions
    canvas = tk.Canvas(finishQuizPage, height = mainScreenHeight, width = mainScreenWidth)
    canvas.pack()

    ## Create main frame
    mainframe = tk.Frame(finishQuizPage, bg = '#80c1ff')
    mainframe.place(relheight = 1, relwidth = 1)

    ## Create label displaying 'Results'
    resultsLabel = tk.Label(mainframe, text = 'Results', bg = 'gray', font = titleFont)
    resultsLabel.place(relheight = 0.15, relwidth = 1)

    ## Create label to show user's result
    userResultsLabel = tk.Label(mainframe, text = 'You got ' + str(correctAnswers) + ' / ' + str(noOfQuestions), bg = 'gray', font = titleFont)
    userResultsLabel.place(relx = 0.3, rely = 0.3, relheight = 0.12, relwidth = 0.4)

    ## Create button to return to main menu
    returnButton = tk.Button(mainframe, text = 'Return to Main Menu', highlightbackground = '#424949', fg = 'blue', font = defaultFont, command = lambda: self.r
    returnButton.place(relx = 0.375, rely = 0.7, relheight = 0.1, relwidth = 0.25)

    finishQuizPage.mainloop()
```

This function retrieves the 'correct' field from the Progress table for the records of the quiz just played. It does this by taking in as a parameter the number of questions (derived from the length of the questions list in the previous function) and retrieving that number of records from the bottom of the table (or literally from the top of the table in descending order). It then goes through all the values retrieved and adds to a counter correctAnswers for every 'True'. The function then creates a page and labels displaying 'Results', the number of question the user answered correctly and a button allowing the user to return to the main menu (using the returnToMainMenu function created previously). It looked like this:





I wrote this line in the playSelectedQuiz function at the end, outside my for loop, to initiate this function:

```
self.finishQuiz(username, len(questions), Login.titleFont, Login.buttonFont, Login.defaultFont)
```

When testing the main menu and all the related play quiz functions, I noticed an inconsistency. This was that when choosing the play quiz function from the main menu, the main menu was not closed and therefore prevented the displayQuestion function to work since a conflict of tkinter images. I

```
def selectButtonPressed(self, previousPage, username, menuOption):
    if menuOption.cget('text') == 'Play Quiz':
        previousPage.destroy()
        self.playQuiz(username, Login.titleFont, Login.defaultFont, Login.buttonFont)
    elif menuOption.cget('text') == 'View Quizzes':
        previousPage.destroy()
        self.viewQuizzes(username, Login.titleFont, Login.defaultFont, Login.buttonFont)
    elif menuOption.cget('text') == 'Create Quiz':
        previousPage.destroy()
```

therefore destroyed the main menu upon opening the play quiz function. I did so in my selectButtonPressed function, as I did for my createQuiz and viewQuizzes functions:
When testing further, although most of the function worked together as desired, there was an issue right at the end with the finishQuiz function when returning to the main menu. This error was produced:

```
    return self.tk.getint(self.tk.call(
_tkinter.TclError: image "pyimage13" doesn't exist
```

This had meant that there was another page interfering with the main menu since there was an interference with the images. I noticed that this was the playSelectedQuizPage, which was previously only hidden, but not destroyed. Hence, I passed this page to the finishQuiz function and wrote:

```
command = lambda: (playSelectedQuizPage.destroy(), self.returnToMainMenu(finishQuizPage, username)))
```

As the command for the return button to first destroy that page.

View Classes function:
For this function, the students had to be separated into classes. As of yet, this hasn't explicitly happened. However, upon creating an account, the user was required to select an admin user to verify them, supposedly being their teacher if they were a student. Using this, I could now separate out the students for their corresponding admin user. I followed a similar format to the view quizzes function:

```
def viewClasses(self, username, titleFont, defaultFont, buttonFont):
    ## Retrieves the assigned height, width and fonts for the main create question page
    mainScreenWidth = viewQuiz.getScreenWidth()
    mainScreenHeight = viewQuiz.getScreenHeight()

    ## Create page and title
    viewClassesPage = tk.Tk()
    viewClassesPage.title("View Classes")

    ## Create canvas with dimensions of the view quizzes page
    canvas = tk.Canvas(viewClassesPage, height = mainScreenHeight, width = mainScreenWidth)
    canvas.pack()

    ## Create main frame with default background
    mainframe = tk.Frame(viewClassesPage, bg = '#80c1ff')
    mainframe.place(relheight = 1, relwidth = 1)

    ## Create label displaying 'View Classes'
    welcomeLabel = tk.Label(mainframe, text = "View Classes", bg = 'gray', font = titleFont)
    welcomeLabel.place(relheight = 0.1, relwidth = 1)

    ## Create return button to allow users to return to main menu
    returnButton = tk.Button(mainframe, text = "Go\nBack", highlightbackground = '#424949', fg = 'blue', font = defaultFont, command = lambda: self.returnToMainM
    returnButton.place(relheight = 0.1, relwidth = 0.1)

    ## Create black background frame to show partition between studentsFrame, studentQuizzesFrame and studentProgressFrame
    background = tk.Frame(viewClassesPage, bg = 'black')
    background.place(rely = 0.1, relheight = 0.9, relwidth = 1)

    ## Create frame for all relevant students
    studentsFrame = tk.Frame(viewClassesPage, bg = '#73ade5')
    studentsFrame.place(rely = 0.1, relheight = 0.9, relwidth = 0.2)
```

```
    ## Create listbox for all relevant students
    studentsListbox = tk.Listbox(studentsFrame, font = defaultFont, exportselection = False)
    ## Set height of listbox and position of select button
    listboxHeight = 0
    buttonPos = 0.1
    ## Retrieve userID from database with given username
    userID = viewingFromDatabase.viewFromDatabase(c, False, False, 'userID', 'username', "'"+username+"'", 'UserDetails')
    ## Retrieve students' usernames from database with given adminID and 'Students' as account type
    students = viewingFromDatabase.viewFromDatabase(c, False, False, 'username', 'adminID', str(userID[0][0])+" AND accountType = 'Student'", 'UserDetails')
    for i in range(len(students)):
        ## Insert each student into the listbox
        studentsListbox.insert('end', students[i][0])
        ## If not reached bottom of page
        if buttonPos < 0.8:
            ## Increase height of listbox and button position
            listboxHeight += 0.1
            buttonPos += 0.1
    ## Place the listbox
    studentsListbox.place(relx = 0.1, rely = 0.05, relheight = listboxHeight, relwidth = 0.8)

    ## Create frame for all quizzes undertook by selected student
    studentQuizzesFrame = tk.Frame(viewClassesPage, bg = '#73ade5')
    studentQuizzesFrame.place(relx = 0.205, rely = 0.1, relheight = 0.9, relwidth = 0.2)

    ## Create listbox for all quizzes undertook by selected student
    studentQuizzesListbox = tk.Listbox(studentQuizzesFrame, font = defaultFont, exportselection = False)

    ## Create frame for progress of selected quiz of a selected student
    studentProgressFrame = tk.Frame(viewClassesPage, bg = '#80c1ff')
    studentProgressFrame.place(relx = 0.410, rely = 0.1, relheight = 0.9, relwidth = 0.590)

    viewClassesPage.mainloop()
```

As with the view quizzes function, I created a page with three distinct frames: one for all the students which had the corresponding adminID to the userID of the username passed into the function as a parameter, one for all the quizzes a selected user has taken, and one for the progress of that student in that quiz. I modified the database so the testing of this function would become clearer:

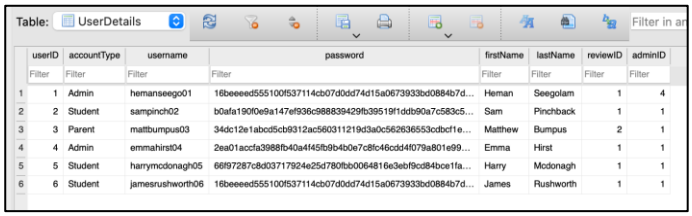
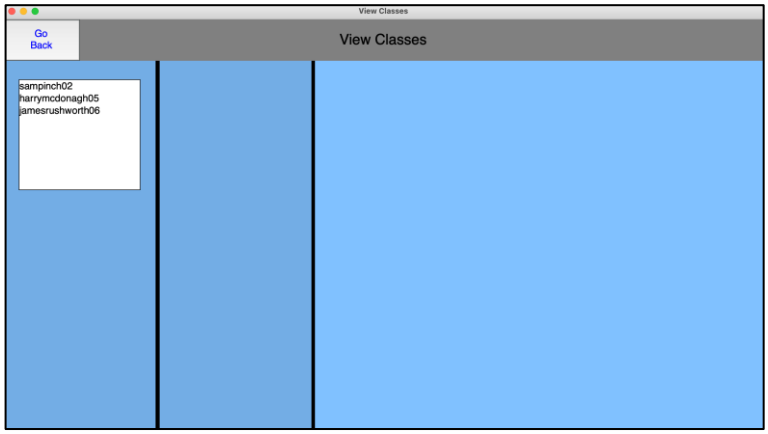


Table: UserDetails

	userID	accountType	username	password	firstName	lastName	reviewID	adminID
1	1	Admin	hemanseego01	16beeeed555100f537114cb07d0dd74d15a0673933bd0884b7d...	Heman	Seegolam	1	4
2	2	Student	sampinch02	b0afa190f0e9a147ef936c988839429fb39519f1ddb90a7c583c5...	Sam	Pinchback	1	1
3	3	Parent	mattbumpus03	34dc12e1abcd5cb9312ac560311219d3a0c562636553cdbcf1e...	Matthew	Bumpus	2	1
4	4	Admin	emmahirst04	2ea01accfa3988fb40a4f45fb9b4b0e7c8fc46cdd4f079a801e99...	Emma	Hirst	1	1
5	5	Student	harrymcdonagh05	66f97287c8d03717924e25d780fbb0064816e3ebf9cd84bce1fa...	Harry	Mcdonagh	1	1
6	6	Student	jamesrushworth06	16beeeed555100f537114cb07d0dd74d15a0673933bd0884b7d...	James	Rushworth	1	1

This produced (logged in as hemanseego01):



To retrieve the users from the database that contained the relevant adminID and also 'Student' as their account type, it required me to extend the SQL statement. This was still executable using my database class by adding the second condition after the value of 'recordPrimaryKeyValue'.

I now created another function to create another list box in the middle frame upon selection in the first:

```
def displayStudentQuizzes(self, student, studentQuizzesListbox):
    ## Clear the question list box (to allow for changing questions)
    studentQuizzesListbox.delete(0,'end')
    ## Set height of listbox and position of select button
    listboxHeight = 0
    buttonPos = 0.1
    print(student)
    ## Retrieve userID from database with given student
    userID = viewingFromDatabase.viewFromDatabase(c, False, False, 'userID', 'username', "'"+student+"'", 'UserDetails')
    ## Retrieve quizIDs from database for given student
    quizIDList = viewingFromDatabase.viewFromDatabase(c, False, False, 'quizID', 'userID', str(userID[0][0]), 'Progress')
    ## Remove duplicates
    quizIDList = list(dict.fromkeys(quizIDList))

    quizzes = []
    for each in quizIDList:
        ## Retrieve quizzes from database for given student and add to quizzes list
        quiz = viewingFromDatabase.viewFromDatabase(c, True, False, None, 'quizID', str(each[0]), 'Quiz')
        quizzes.append(quiz)
    ## Add each quiz to the listbox
    for i in range(len(quizzes)):
        studentQuizzesListbox.insert('end', quizzes[i][0][1] + ' ' + quizzes[i][0][2])
        if buttonPos < 0.8:
            listboxHeight += 0.1
            buttonPos += 0.1
    studentQuizzesListbox.place(relx = 0.1, rely = 0.05, relheight = listboxHeight, relwidth = 0.8)
```

I created a bind to the students list box to this function:

```
studentsListbox.bind('<<ListboxSelect>>', lambda x: self.displayStudentQuizzes(studentsListbox.get(studentsListbox.curselection()), studentQuizzesListbox))
```

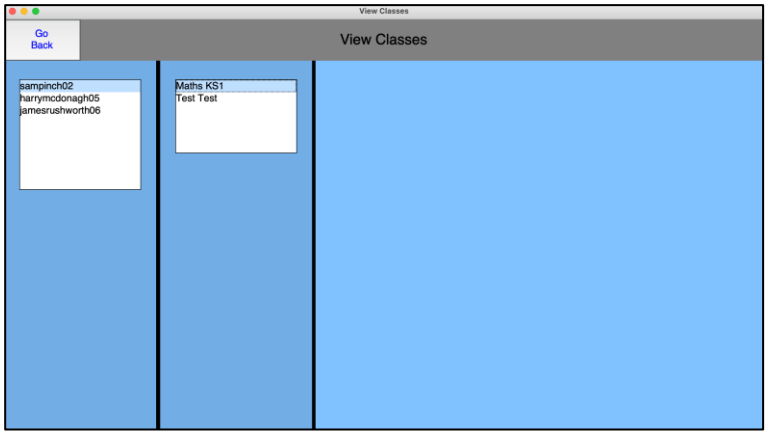
This function first clears the student quizzes list box in case its preoccupied from an earlier selection. Then, using the student parameter (as the current selection within the students list box), it retrieves the user ID of the student and the quiz ID of all questions they have completed. Since multiple questions will refer to the same quiz, I made the list into a dictionary to remove any duplicate values and then back into a list. It then creates a list for the details of all quizzes played by the user, and appends the values retrieved from the database for the quiz ID of each value in the quiz ID list. Each quiz is then inserted into the list box and the list box is placed onto the page. Within the list box, I show the subject and the level of each quiz.

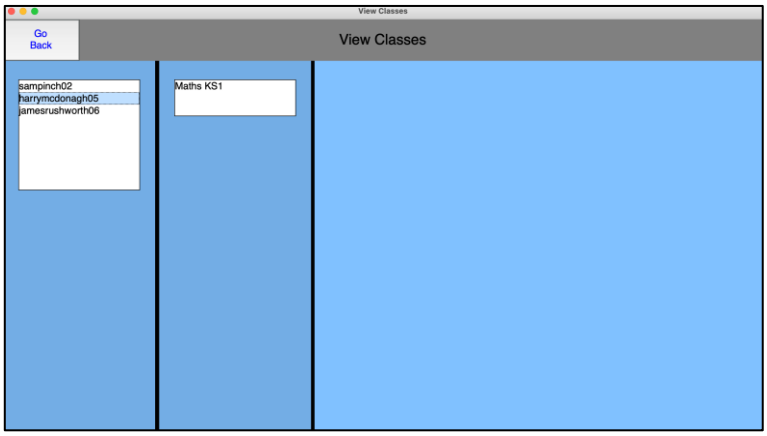
To test this effectively, I had to modify the database:

Table: Progress

	progressID	userID	quizID	questionID	correct	time
	Filter	Filter	Filter	Filter	Filter	Filter
1	132	1	20	18	True	2021-02-04 18:49:11
2	133	1	20	19	True	2021-02-04 18:49:32
3	134	1	20	20	False	2021-02-04 18:49:54
4	135	2	20	18	False	2021-02-06 18:50:00
5	136	2	20	19	True	2021-02-06 18:50:21
6	137	2	20	20	False	2021-02-06 18:50:43
7	138	5	20	18	False	2021-02-06 18:51:00
8	139	5	20	19	False	2021-02-06 18:51:21
9	140	5	20	20	True	2021-02-06 18:51:43
10	141	6	20	18	True	2021-02-06 18:52:00
11	142	6	20	19	True	2021-02-06 18:52:21
12	143	6	20	20	True	2021-02-06 18:52:43
13	144	2	19	16	False	2021-02-06 18:53:00
14	145	2	19	17	True	2021-02-06 18:53:21

This produced:





However, there was a limitation here. Since the program deleted any duplicate quiz IDs, multiple attempts for a quiz were not shown in the list. For example, if I added the following to the database:

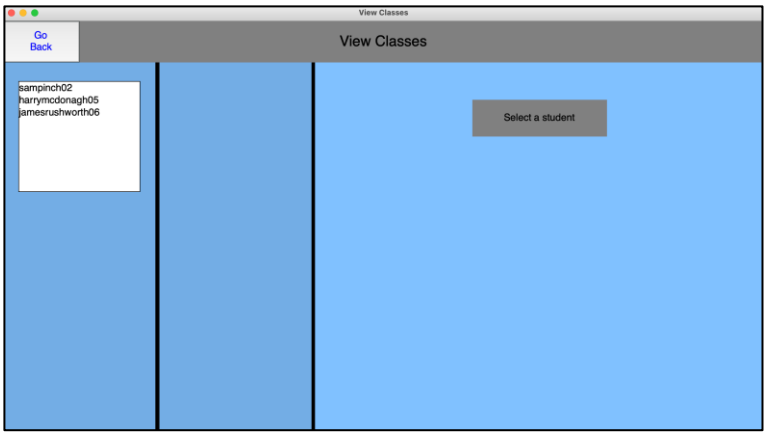
15	146	5	20	18	True	2021-02-06 18:55:00
16	147	5	20	19	False	2021-02-06 18:55:21
17	148	5	20	20	True	2021-02-06 18:55:43

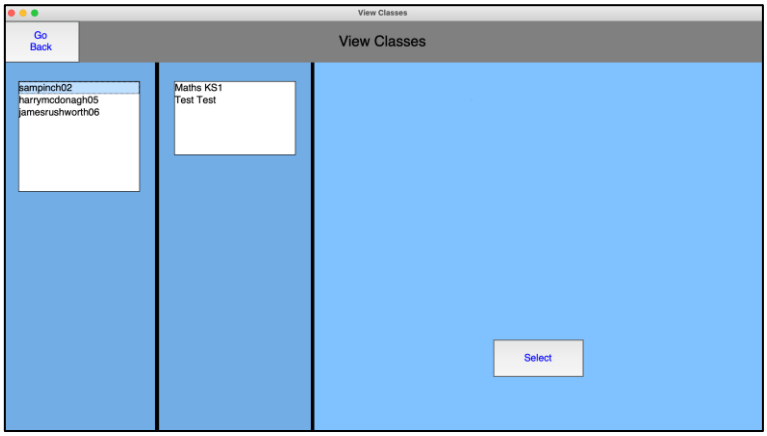
No changes were made to the list shown above. I decided to deal with this when displaying the progress, so although a quiz would not be explicitly mentioned twice in the student quizzes list box, their progress for multiple attempts would be shown when it is selected.
I created a select student label and button within the view classes function:

```
## Create and place label to select student
selectLabel = tk.Label(studentProgressFrame, text = "Select a student", bg = 'gray', font = defaultFont)
selectLabel.place(relx = 0.35, rely = 0.1, relheight = 0.1, relwidth = 0.3)

## Create button to select student
selectButton = tk.Button(studentProgressFrame, text = "Select", highlightbackground = '#424949', fg = 'blue', font = defaultFont)
```

I placed the select button and hid the label in the displayStudentQuizzes function:

```
## Place select button
selectButton.place(relx = 0.4, rely = 0.75, relheight = 0.1, relwidth = 0.2)
## Hide select label
selectLabel.place(relheight = 0, relwidth = 0)
```





I now created a function to create another list box in the right frame upon selection in the second:

```
def displayStudentProgress(self, student, userID, selectButton, selectLabel, quizzes, studentQuizzesListbox, selectedQuiz, quizTitleLabel, usernameLabel, attemptsLabel, progressListbox):
    ## Find which row of the listbox was selected
    index = studentQuizzesListbox.get(0, "end").index(selectedQuiz)
    ## Find the quizID of selected quiz using above index
    quizID = quizzes[index][0][0]
    noOfQuestions = quizzes[index][0][3]
    ## Find progress of student for selected quiz
    progress = viewingFromDatabase.viewFromDatabase(c, True, False, None, 'userID', str(userID[0][0]) + ' AND quizID = ' + str(quizID), 'Progress')
    ## Find how many attempts the user had at the quiz (by dividing the number of progress records by the number of questions)
    attempts = len(progress) // noOfQuestions

    ## Clear the progress list box (to allow for changing quizzes)
    progressListbox.delete(0,'end')

    ## Configure and place the labels, the select button and listbox created before
    selectLabel.place(relheight = 0, relwidth = 0)
    selectButton.place(relx = 0.4, rely = 0.83, relheight = 0.1, relwidth = 0.2)
    quizTitleLabel.config(text = quizzes[index][0][1] + ' ' + quizzes[index][0][2])
    quizTitleLabel.place(relx = 0.1, rely = 0.05, relheight = 0.08, relwidth = 0.2)
    usernameLabel.config(text = student)
    usernameLabel.place(relx = 0.75, rely = 0.05, relheight = 0.08, relwidth = 0.2)
    attemptsLabel.config(text = 'Number of attemps: ' + str(attempts))
    attemptsLabel.place(relx = 0.1, rely = 0.18, relheight = 0.08, relwidth = 0.3)
    progressListbox.place(relx = 0.1, rely = 0.29, relheight = 0.5, relwidth = 0.8)

    ## For each question in the selected quiz
    for each in progress:
        ## Find questionID and corresponding question from database
        questionID = each[3]
        question = viewingFromDatabase.viewFromDatabase(c, False, False, 'question', 'questionID', str(questionID), 'Question')
        if each[4] == 'True':
            answer = 'Correct'
        elif each[4] == 'False':
            answer = 'Incorrect'
        ## Insert the question, whether correct and the date and time to the progress listbox
        progressListbox.insert('end', question[0][0] + '   ' + answer + '   ' + each[5])
```

This function binds to the student quizzes list box:

```
studentQuizzesListbox.bind('<<ListboxSelect>>', lambda x: self.displayStudentProgress(student, userID, selectButton, selectLabel, quizzes, studentQuizzesList
```

The purpose of this function is to find out the selection of the student quizzes list box by checking which row was selected. This produced an index which I then used to retrieve the quiz ID from the quizzes list declared in the previous function. This works regardless of the order of the student quizzes list box for two different students since the quizzes list would be declared differently accordingly. The index also helped to find the number of questions of the selected quiz from the quizzes list which is helpful since the number of attempts can easily be calculated by dividing the number of progress records (retrieved from the database with the conditions that the user ID matches that from the previous function and the quiz ID matches the one just found), by the number

of questions. Although normal division works all the same, I made use of integer division for this calculation since I would be displaying this later on. The function then goes on to clear the list box in case there were already values present from a previous selection and then configure and place some labels and a list box that I created in the original view classes function:

```
## Create labels and listbox for progress frame to use later
quizTitleLabel = tk.Label(studentProgressFrame, bg = 'gray', font = defaultFont)
usernameLabel = tk.Label(studentProgressFrame, bg = 'gray', font = defaultFont)
attemptsLabel = tk.Label(studentProgressFrame, bg = 'gray', font = defaultFont)
progressListbox = tk.Listbox(studentProgressFrame, font = defaultFont, exportselection = False)
```

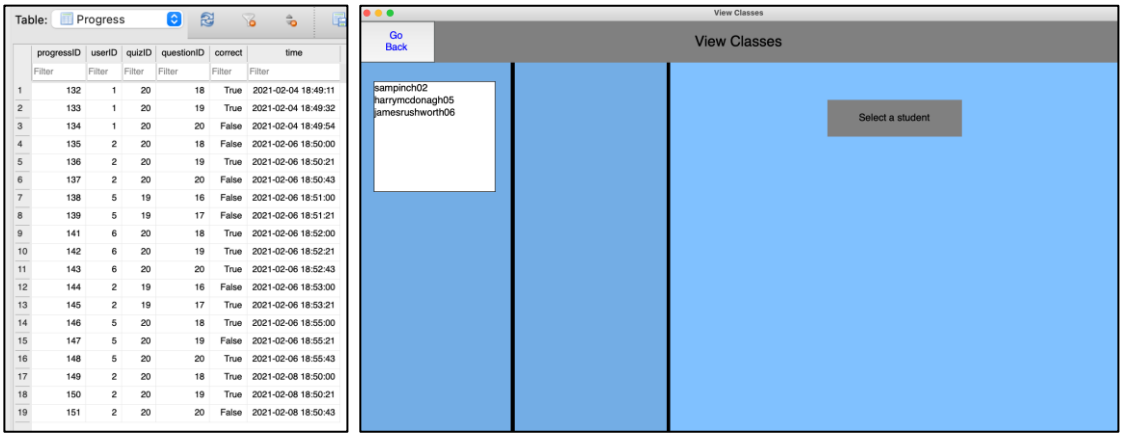
I also changed the configuration and placement of the select label and button:

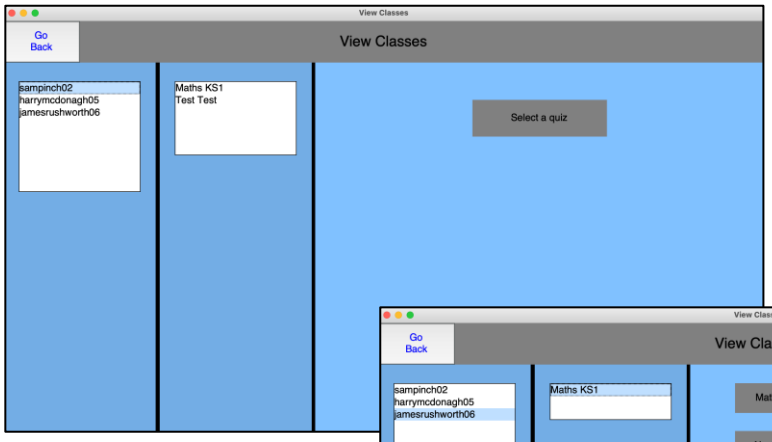
```
## Configure select label
selectLabel.config(text = 'Select a quiz')
```

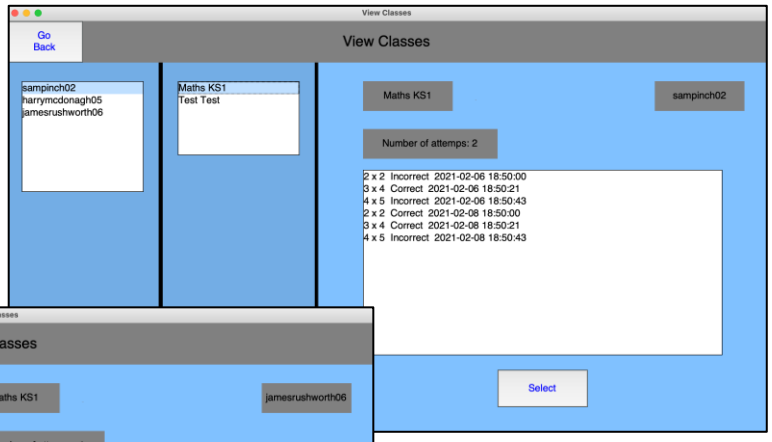
Instead of placing the label and the button in the displayStudentQuizzes function as before, I simply configured it to read ‘Select a quiz’.

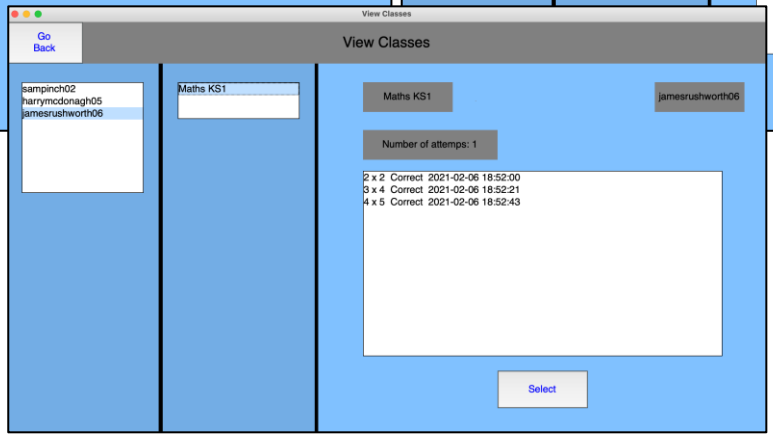
In the displayStudentProgress, I then hid the select label and placed the select button as before, as shown above. Finally, this function creates a for loop for each list in the progress list and finds the question ID of the progress record and retrieves the question from the database. This question, along with whether the user answered it correctly or incorrectly and the date and time are then shown in the list box.

To test it efficiently, I again modified the database slightly. This produced:





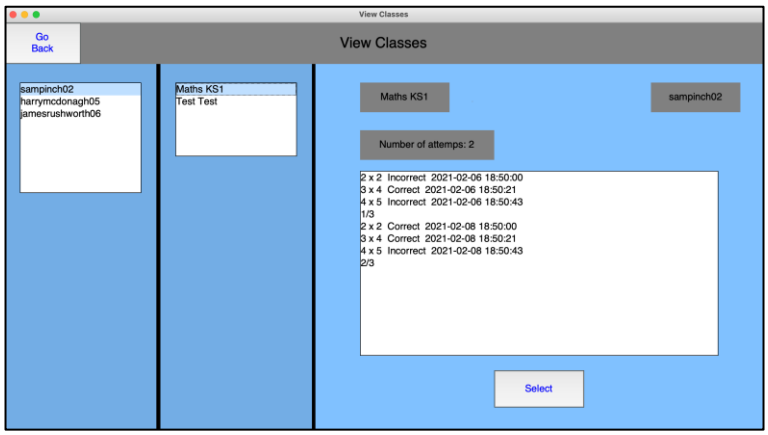




I extended the function to show how many questions the user answered correctly out of the total number of questions after each attempt. I modified the for loop to:

```
correctAnswers = 0
counter = 0
## For each question in the selected quiz
for each in progress:
    counter += 1
    ## Find questionID and corresponding question from database
    questionID = each[3]
    question = viewingFromDatabase.viewFromDatabase(c, False, False, 'question', 'questionID', str(questionID), 'Question')
    if each[4] == 'True':
        answer = 'Correct'
        correctAnswers += 1
    elif each[4] == 'False':
        answer = 'Incorrect'
    ## Insert the question, whether correct and the date and time to the progress listbox
    progressListbox.insert('end', question[0][0] + '   ' + answer + '   ' + each[5])
    if counter == noOfQuestions:
        ## Show number of correctly answered questions out of total for each attempt
        progressListbox.insert('end', str(correctAnswers) + '/' + str(noOfQuestions))
        correctAnswers = 0
        counter = 0
```

This produced:



As desired.

I now created a function for the select button:

```
def displayProgress(self, previousPage, username, student, progress, noOfQuestions, titleFont, defaultFont, buttonFont):
    ## Destroy previous page
    previousPage.destroy()

    ## Retrieves the assigned height, width and fonts for the main create question page
    mainScreenWidth = mainMenu.getScreenWidth()
    mainScreenHeight = mainMenu.getScreenHeight()

    ## Create page and title
    displayProgressPage = tk.Tk()
    displayProgressPage.title("View Progress")

    ## Create canvas with dimensions of the view quizzes page
    canvas = tk.Canvas(displayProgressPage, height = mainScreenHeight, width = mainScreenWidth)
    canvas.pack()

    ## Create main frame with default background
    mainframe = tk.Frame(displayProgressPage, bg = '#80c1ff')
    mainframe.place(relheight = 1, relwidth = 1)

    ## Create label displaying 'View Progress'
    welcomeLabel = tk.Label(mainframe, text = "View Progress", bg = 'gray', font = titleFont)
    welcomeLabel.place(relheight = 0.1, relwidth = 1)

    ## Create return button to allow users to return to main menu
    returnButton = tk.Button(mainframe, text = "Go\nBack", highlightbackground = '#424949', fg = 'blue', font = defaultFont, command = lambda:
    returnButton.place(relheight = 0.1, relwidth = 0.1)
                                                    (displayProgressPage.destroy(), self.viewClasses(username,
                                                    Login.titleFont, Login.defaultFont, Login.buttonFont)))
```

```
    ## Create counter to track questions
    counter = 0
    ## Create counter to track correct answers
    correctAnswers = 0
    ## Create list to store number of correct answers for each attempt
    correctAnswersList = []
    ## For each record in progress list
    for each in progress:
        counter += 1
        ## Find all correct answers and add to counter
        if each[4] == 'True':
            correctAnswers += 1
        if counter == noOfQuestions:
            ## If correctAnswersList is empty...add correctAnswers counter
            if len(correctAnswersList) == 0:
                correctAnswersList.append(correctAnswers)
            ## If not...add correctAnswers counter - previous value (or last term of list)
            else:
                ## Subtract correctAnswers answer by all previous values
                correctAnswersCounter = correctAnswers
                for each in correctAnswersList:
                    correctAnswersCounter = correctAnswersCounter - each
                ## Append subtracted value to correct answers list
                correctAnswersList.append(correctAnswersCounter)
            ## Reset the question counter
            counter = 0
```

```
    ## Create counter to track attempts
    attempt = 0
    progressText = ''
    ## For each attempt...
    for each in correctAnswersList:
        attempt += 1
        ## Add attempt number and progress to progressText string
        progressText = progressText + '\n' + 'Attempt ' + str(attempt) + ': ' + '\n' + str(each) + '/' + str(noOfQuestions) + '\n'

    ## Create label for progress of all attempts
    progressLabel = tk.Label(mainframe, text = progressText, bg = 'gray', font = defaultFont)
    progressLabel.place(relx = 0.6, rely = 0.24, relheight = 0.54, relwidth = 0.35)

    ## Create button to view full progress
    viewFullProgressButton = tk.Button(mainframe, text = "View Full Progress", highlightbackground = '#424949', fg = 'blue', font = defaultFont)
    viewFullProgressButton.place(relx = 0.4, rely = 0.82, relheight = 0.1, relwidth = 0.2)

    displayProgressPage.mainloop()
```

I configured the command of the select button for it to bind to this function within the displayStudentProgress function:

```
selectButton.config(command = lambda: self.displayProgress(viewClassesPage, username, student, progress, noOfQuestions, Login.titleFont, Login.defaultFont, Lo
```

This function starts off by destroying the previous page and setting up a page of its own. It has a canvas, main frame, welcome label stating 'View Progress' and a return button that destroys the page and initiates the viewClasses function. The next part of this function counts the amount of correct answers in total, similar to in the previous function, but doesn't reset the correct answers counter for every attempt of the quiz. Instead, it appends the value to a correct answers list for every attempt. It does this by checking whether the list is empty – if it is, it appends the value of the correct answers counter, if not, it appends the value of the correct answers counter minus the previous value in the list. This is done using the -1 index which retrieves the last value of the list. Next, I use a separate function to create a pie chart on to the page. I first set up the data needed, here being the ratio of correct answers to incorrect answers. The incorrect answers are found by subtracting the number of correct answers from the total number of questions. A new function is now initiated:

```
def createPieChart(self, page, data):
    ## Create list for pie chart labels, i.e. Correct and Incorrect
    pieLabels = ['Correct', 'Incorrect']
    ## Create list for pie chart colours, i.e. green and red
    colours = ['#9FE2BF', '#F08080']

    ## Create figure for pie chart
    figure = plt.Figure(figsize=(5,5))
    ## Create background for pie chart
    figure.patch.set_facecolor('#73ade5')
    ## Create pie chart
    ax = figure.add_subplot(111)
    ax.pie(data, labels = pieLabels, colors = colours, explode = (0.1, 0), autopct = '%1.1f%%')

    ## Create canvas for pie chart
    pieChart = FigureCanvasTkAgg(figure, page)
    return pieChart
```

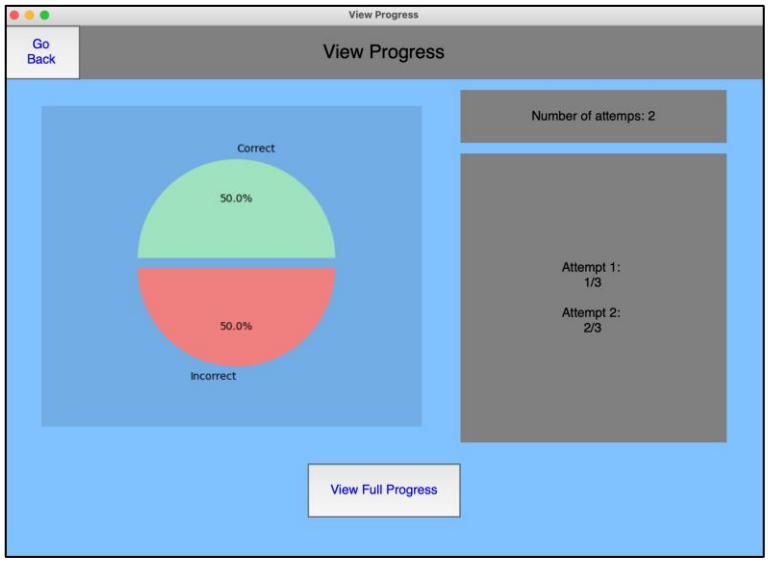
In this createPieChart function, it takes in as parameters the page in which the pie chart needs to displayed on, and the data for the pie chart. In this function, labels for the pie chart are set up as 'Correct' and 'Incorrect' and also corresponding colours as in the play quiz function. To create the chart, I made use of the matplotlib library:

```
MacBook-Pro:~ hemans$ pip3 install matplotlib
Collecting matplotlib
  Downloading https://files.pythonhosted.org/packages/a7/cf/30e026703d8748d0109c
201329b36369fa1ecb2ac404cfd588a37d858f6c/matplotlib-3.3.4-cp38-cp38-macosx_10_9_
x86_64.whl (8.5MB)
     |████████████████████████████████| 8.5MB 2.0MB/s
```

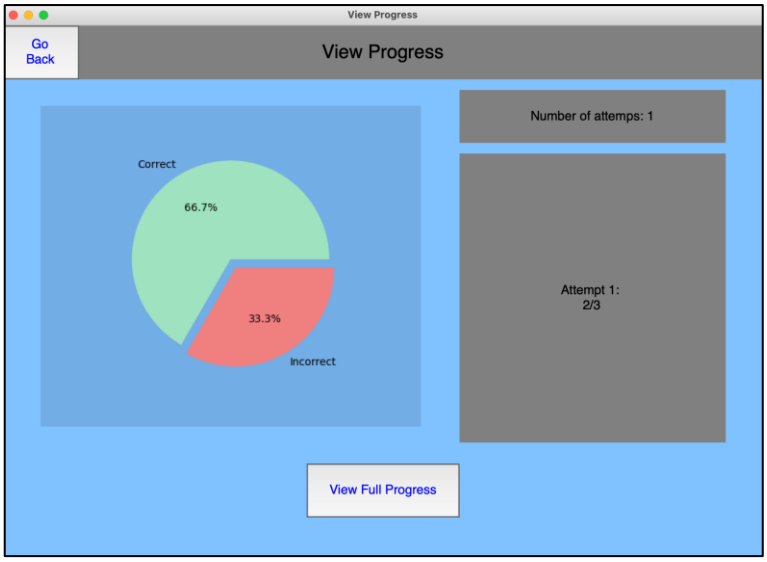
```
import matplotlib.pyplot as plt
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
```

Then the function creates a figure for the pie chart with set dimensions and a background colour. Next, the pie chart is 'plotted'. I also extended it to 'explode' (bring out) the correct section of the pie chart and show the percentage of each section. Finally, the function creates a canvas which is compatible with the matplotlib library and the tkinter library and places the figure and places it on the page. I then return this canvas.

The displayProgress function then places it accordingly. To finish off this function, I created labels to show the user the number of attempts that the selected user has had at the selected quiz and the results from each attempt. Finally, I created a button to allow the user to see more progress. This function produced (when selecting sampinch02 and Maths KS1):



When selecting harrymcdonagh05 and Maths KS1:



View full progress function:

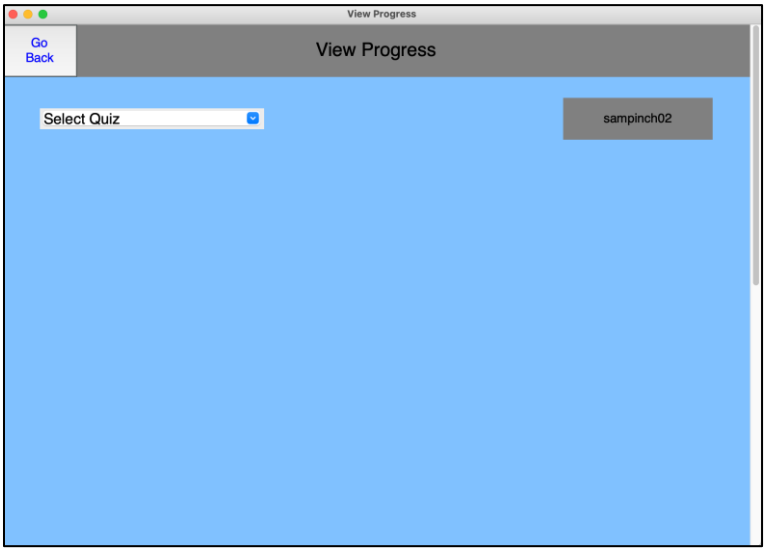
```
def viewFullProgress(self, username, student, titleFont, defaultFont, buttonFont):
    ## Retrieves the assigned height, width and fonts for the view full progress page
    mainScreenWidth = mainMenu.getScreenWidth()
    mainScreenHeight = mainMenu.getScreenHeight()

    ## Create page and title
    viewFullProgressPage = tk.Tk()
    viewFullProgressPage.title("View Progress")

    ## Create container and canvas with dimensions of the main menu page
    container = tk.Frame(viewFullProgressPage)
    container.pack()
    canvas = tk.Canvas(container, height = mainScreenHeight, width = mainScreenWidth, highlightthickness = 0, relief = 'ridge')
    canvas.pack(side = "left", fill = "both", expand = True)

    ## Create scroll bar
    scrollbar = ttk.Scrollbar(container, orient = "vertical", command = canvas.yview)
    scrollbar.pack(side = "right", fill = "y")

    ## Create frame which will be scrolled
    scrollableFrame = tk.Frame(canvas, bg = '#80c1ff')
    canvasFrame = canvas.create_window((0, 0), window = scrollableFrame, anchor = "nw")
    canvas.itemconfig(canvasFrame, height = 1400, width = mainScreenWidth)
    canvas.configure(yscrollcommand = scrollbar.set)
    scrollableFrame.bind("<Configure>", lambda x: canvas.configure(scrollregion = canvas.bbox("all")))

    ## Create label displaying 'View Progress'
    welcomeLabel = tk.Label(scrollableFrame, text = "View Progress", bg = 'gray', font = titleFont)
    welcomeLabel.place(relheight = 0.05, relwidth = 1)

    ## Create return button to allow users to return to view classes page
    returnButton = tk.Button(scrollableFrame, text = "Go\nBack", highlightbackground = '#424949', fg = 'blue', font = defaultFont, command = lambda: (viewFullProg
    returnButton.place(relheight = 0.05, relwidth = 0.1)

    studentLabel = tk.Label(scrollableFrame, text = student, bg = 'gray', font = defaultFont)
    studentLabel.place(relx = 0.75, rely = 0.07, relheight = 0.04, relwidth = 0.2)
```

For this page, I created a scroll bar to allow the user to see more data on the same page by scrolling up and down. I did this by creating a container frame which packed the canvas on the left, filling both sides, and the scroll bar on the right. I created a frame which would be bound to the scroll bar which allowed it to move up and down accordingly with the scroll bar. I also created a welcome label displaying 'View Progress', a return button destroying the current page and going back to the view classes function and a student label to show which student was selected.

```
    ## Retrieve userID from database with given student
    userID = viewingFromDatabase.viewFromDatabase(c, False, False, 'userID', 'username', "'"+student+"'", 'UserDetails')
    ## Retrieve quizIDs from database for given student
    quizIDList = viewingFromDatabase.viewFromDatabase(c, False, False, 'quizID', 'userID', str(userID[0][0]), 'Progress')
    ## Remove duplicates
    quizIDList = list(dict.fromkeys(quizIDList))

    quizzes = []
    quizLabels = ['Select Quiz']
    for each in quizIDList:
        ## Retrieve quizzes from database for given student and add to quizzes list
        quiz = viewingFromDatabase.viewFromDatabase(c, True, False, None, 'quizID', str(each[0]), 'Quiz')
        quizzes.append(quiz)
    ## Add names for each quiz to quizLabels list (i.e. what will appear in combobox)
    for each in quizzes:
        quizLabels.append([each[0][1],each[0][2]])
    ## Create combobox for all quizzes completed by selected student
    quizCombobox = ttk.Combobox(scrollableFrame, values = quizLabels, font = buttonFont, state = 'readonly')
    quizCombobox.place(relx = 0.05, rely = 0.08, relheight = 0.02, relwidth = 0.3)
    quizCombobox.current(0)
    quizCombobox.bind('<<ComboboxSelected>>', lambda x: self.showProgressGraph(student, userID, quizCombobox.get(), quizzes, scrollableFrame, otherUsersLabel, tab

    ## Create label for progress of other users
    otherUsersLabel = tk.Label(scrollableFrame, text = 'Other users for this quiz:', bg = 'gray')

    ## Create frame for all users progress table
    tableFrame = tk.Frame(scrollableFrame, bg = '#80c1ff')

    viewFullProgressPage.mainloop()
```

Then, using the student username passed into the function as a parameter, I retrieved the user ID of that student from the database so I could go on to retrieve all the IDs of the quizzes completed by the student. I remove any duplicates in the list by converting to a dictionary and back to a list. For each ID in the list, I then retrieved the information of all the quizzes and inserted each corresponding subject and level to a combo-box to allow the users to display progress for the different quizzes done by the user. The default value for the combo-box is 'Select Quiz', pre-written into the quizLabels list. The selections of this combo-box are bound to an event which initiates the showProgressGraph function (shown later). Finally, the function creates a label displaying 'Other uses for this quiz:' indicating the table for the progress of all other users of the selected quiz, and creates a frame for this table.

This produced:



The showProgressGraph function:

```
def showProgressGraph(self, student, studentUserID, quiz, quizzes, page, otherUsersLabel, tableFrame):
    if quiz != 'Select Quiz':
        for each in range(len(quizzes)):
            if quiz == quizzes[each][0][1] + ' ' + quizzes[each][0][2]:
                quizID = quizzes[each][0][0]
                noOfQuestions = quizzes[each][0][3]

        progress = viewingFromDatabase.viewFromDatabase(c, True, False, None, 'userID', str(studentUserID[0][0]) + ' AND quizID = ' + str(quizID), 'Progress')
        ## Create counter to track questions
        counter = 0
        ## Create counter to track correct answers
        correctAnswers = 0
        ## Create list to store number of correct answers for each attempt
        correctAnswersList = []
        ## For each record in progress list
        for each in progress:
            counter += 1
            ## Find all correct answers and add to counter
            if each[4] == 'True':
                correctAnswers += 1
            if counter == noOfQuestions:
                ## If correctAnswersList is empty...add correctAnswers counter
                if len(correctAnswersList) == 0:
                    correctAnswersList.append(correctAnswers)
                ## If not...add correctAnswers counter - previous value (or last term of list)
                else:
                    ## Subtract correctAnswers answer by all previous values
                    correctAnswersCounter = correctAnswers
                    for each in correctAnswersList:
                        correctAnswersCounter = correctAnswersCounter - each
                    ## Append subtracted value to correct answers list
                    correctAnswersList.append(correctAnswersCounter)
                ## Reset the question counter
                counter = 0
```

Upon selection in the quiz combo-box, if not 'Select Quiz', the selected quiz is matched against the quizzes list, passed as a parameter so all other relevant information can be picked out, i.e. the quiz ID and the number of questions in the quiz. Using the student's user ID and the newly obtained quiz ID, all progress records matching these values are then stored within a progress list variable. Next, the function creates some counters, one for the iteration in the coming for loop which represents the question and one for the number of correct answers. It also creates a list which is to contain the number of correct answers for each attempt the selected student has taken at the selected quiz. The function goes through each value in the progress list and checks to see whether the user correctly answered the question. If yes, the correct answers counter is incremented by one. The function also checks at this stage whether the iteration counter is equal to the number of questions. This would indicate that one attempt of the quiz has been checked. As a result, it checks to see if the correct answers list is empty in which case it appends the value of the correct answers counter, and if not if appends the value of the correct answers counter minus all the previous values within the list. This is to ensure the value is only that of the attempt of the quiz. The counter is then reset at this stage. The values of the correct answers list is used later on to create a line graph if appropriate.

```
        percentages = []
        ## If only one attempt...create pie chart
        if len(progress) == noOfQuestions:
            ## Determine size of pie chart slices
            data = [correctAnswers, len(progress)-correctAnswers]
            ## Initiate createPieChart function to create pie chart frame
            pieChart = self.createPieChart(page, data)
            ## Place pie chart frame
            pieChart.get_tk_widget().place(relx = 0.01, rely = 0.12, relheight = 0.3, relwidth = 0.5)
        ## Else create line graph
        else:
            ## For each result, find the percentage and append to percentages list
            for each in correctAnswersList:
                percentage = each / noOfQuestions * 100
                percentages.append(int(percentage))
            lineGraph = self.createLineGraph(page, len(progress)//noOfQuestions, percentages)
            lineGraph.get_tk_widget().place(relx = 0.01, rely = 0.12, relheight = 0.3, relwidth = 0.5)
```

Next, the function checks to see the number of attempts taken by the selected student for the selected quiz. If only one, the function creates a pie chart with the correct data as before. If not, however, the function creates a line chart using the data from the correct answers list. It does this

by creating an empty percentages list, and for every value in the correct answers list, it calculates its percentage by dividing the value by the number of questions and multiplying it by 100. I then go on to append an integer version of this value to the percentages list which python does by simply truncating the value. Initially when coding this part, to obtain an integer value, I used integer division instead of normal division, but this didn't work since if not 100%, the value would not be divisible by an amount which was one or more and therefore produced 0 every time. I fixed this using the method above. I then use this function to create the line graph:

```
def createLineGraph(self, page, noOfAttemps, percentages):
    ## Create list for each number for number of attempts
    attempts = [x+1 for x in range(noOfAttemps)]

    ## Create figure for line graph
    figure = plt.Figure(figsize=(5,5))
    ## Create background for line graph
    figure.patch.set_facecolor('#73ade5')
    ## Create line graph
    ax = figure.add_subplot(111)
    ax.plot(attempts, percentages)
    ax.set_xlabel('Attempt')
    ax.set_ylabel('Percentage')

    ## Create canvas for line graph
    lineGraph = FigureCanvasTkAgg(figure, page)
    return lineGraph
```

This function creates a list from 1 to the number of attempts and plots it against the values in the percentages list. It also creates a default size for the graph, sets the background colour of the graph and labels the axes accordingly. It then returns the line graph, ready for placement.

To finish off the showProgressGraph function:

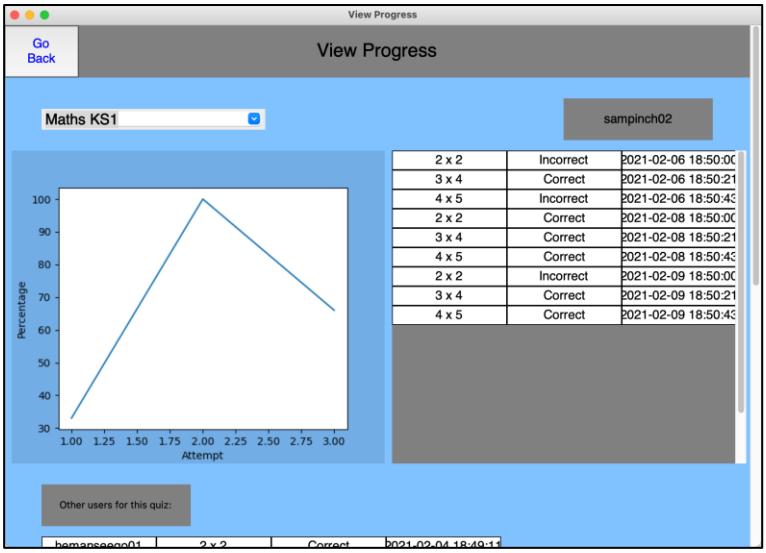
```
    ## Create frame and scroll bar for selected user progress table
    container2 = tk.Frame(page)
    container2.place(relx = 0.52, rely = 0.12, relheight = 0.3, relwidth = 0.475)
    canvas2 = tk.Canvas(container2, highlightthickness = 0, relief = 'ridge')
    canvas2.pack(side = "left", fill = "both", expand = True)
    scrollbar2 = ttk.Scrollbar(container2, orient = "vertical", command = canvas2.yview)
    scrollbar2.pack(side = "right", fill = "y")

    ## Create frame for selected user progress table which will be scrolled
    scrollableFrame2 = tk.Frame(canvas2, bg = 'gray')
    canvasFrame2 = canvas2.create_window((0, 0), window = scrollableFrame2, anchor = "nw")
    canvas2.itemconfig(canvasFrame2, height = 500)
    canvas2.configure(yscrollcommand = scrollbar2.set)
    scrollableFrame2.bind("<Configure>", lambda x: canvas2.configure(scrollregion = canvas2.bbox("all")))

    ## Clear frame in case of changing quizzes
    for widget in tableFrame.winfo_children():
        widget.destroy()
    ## Create and place table for progress of all students for selected quiz
    ## Fetch the relevant data from the database by performing an inner join between the Progress, UserDetails and Quiz tables
    newTable = c.execute(""" SELECT UserDetails.username, Quiz.subject, Quiz.level, Progress.questionID, Progress.correct, Progress.time
                         FROM ((Progress
                         INNER JOIN UserDetails ON Progress.userID = UserDetails.userID)
                         INNER JOIN Quiz ON Progress.quizID = Quiz.quizID); """)
    newTableData = newTable.fetchall()
```

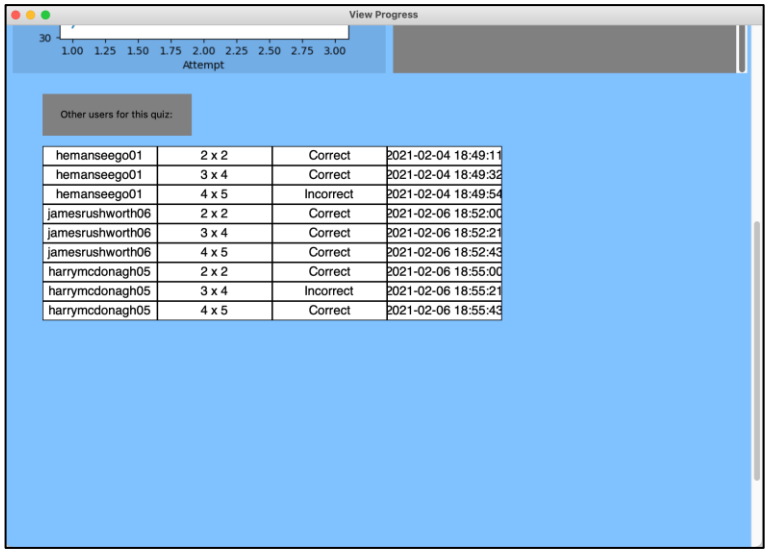
Here, I created another container, frame and scrollbar. This was for the table of the users progress of the selected quiz. In the case of changing between quizzes, the function also clears the table frame for the progress of all other users for the selected quiz by destroying all the included widgets using a for loop and the winfo.children() function to retrieve them. Next, the function created a new SQL table. It did this by performing an inner join on three tables: the Progress table, the UserDetails table and the Quiz table. From the UserDetails table, it retrieved the students' username. From the Quiz table, it retrieved the quizzes' subject and level. From the Progress table, it retrieved each records' question ID, correct field and time.

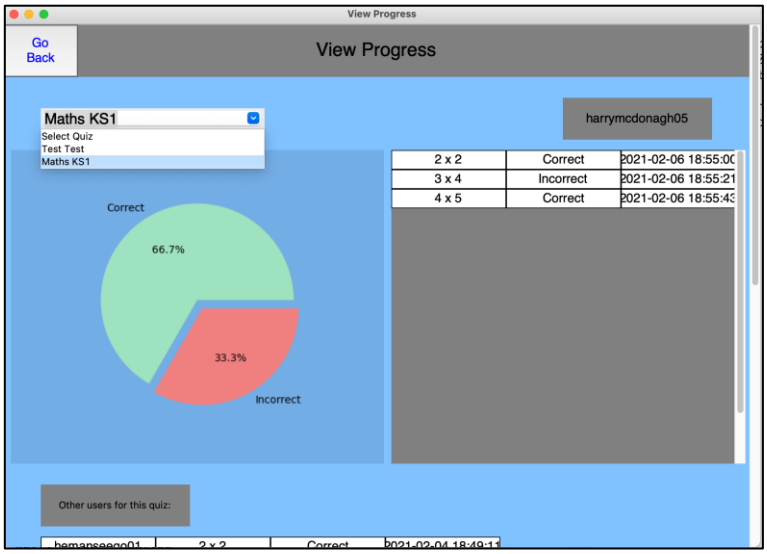
```
        table1Data = []
        table2Data = []
        ## For each record that matches the selected quiz...
        for each in newTableData:
            if quiz == each[1] + ' ' + each[2]:
                ## Retrieve question from database
                question = viewingFromDatabase.viewFromDatabase(c, False, False, 'question', 'questionID', str(each[3]), 'Question')
                if each[4] == 'True':
                    correctField = 'Correct'
                else:
                    correctField = 'Incorrect'
                ## If progress of selected student...
                if each[0] == student:
                    ## Append the question, whether correct and the time for each such record to table 1 list
                    table1Data.append([question[0][0], correctField, each[5]])
                ## If not selected student...
                else:
                    ## Append the username, the question, whether correct and the time for each such record to table 2 list
                    table2Data.append([each[0], question[0][0], correctField, each[5]])

        ## Create table for selected user's progress
        self.createTable(table1Data, scrollableFrame2, Login.defaultFont)
        ## Create table for progress of all other users
        self.createTable(table2Data, tableFrame, Login.defaultFont)
        ## Place other users label
        otherUsersLabel.place(relx = 0.05, rely = 0.44, relheight = 0.04, relwidth = 0.2)
        ## Place the frame for the table for progress of all other users
        tableFrame.place(relx = 0.05, rely = 0.49, relheight = 0.5, relwidth = 0.96)
```

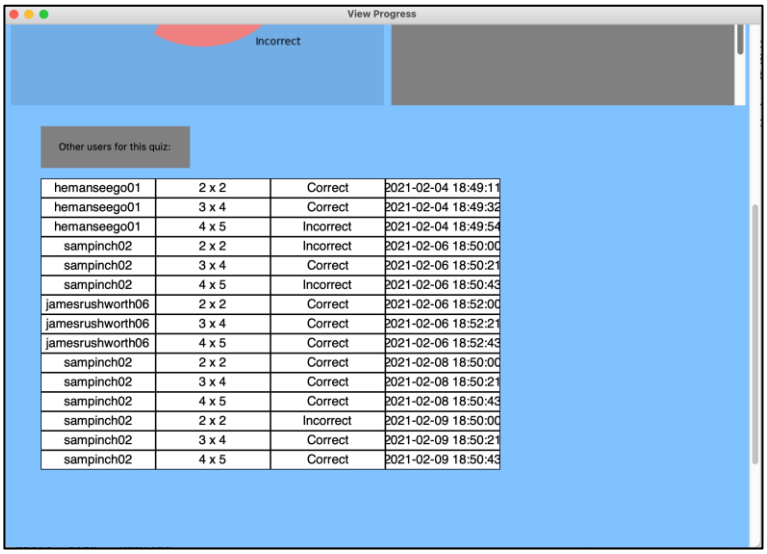
The function then goes through the newly created table by fetching all the data and searches for all records where the quiz subject and level corresponds to that of the selected quiz. With these records, it retrieves the question from the database and translates all ‘True’ values to ‘Correct’ and all ‘False’ values to ‘Incorrect’. It then checks to see whether the username associated with the record is that of the selected student. If it is, the record is appended to the table one list and if not, it’s appended to the table two list. This allows the program to efficiently separate out the data for the two tables. The function finally creates a table for the progress of the selected student for the selected quiz using the create table function created earlier and places it into the scrollableFrame2 with values from table1Data and creates a table for the progress of all other users for the selected quiz places it into the scrollableFrame with values from table2Data. The other users label and table frame created previously are now placed.

This produced:









To test it effectively, I had extended the database with more data:

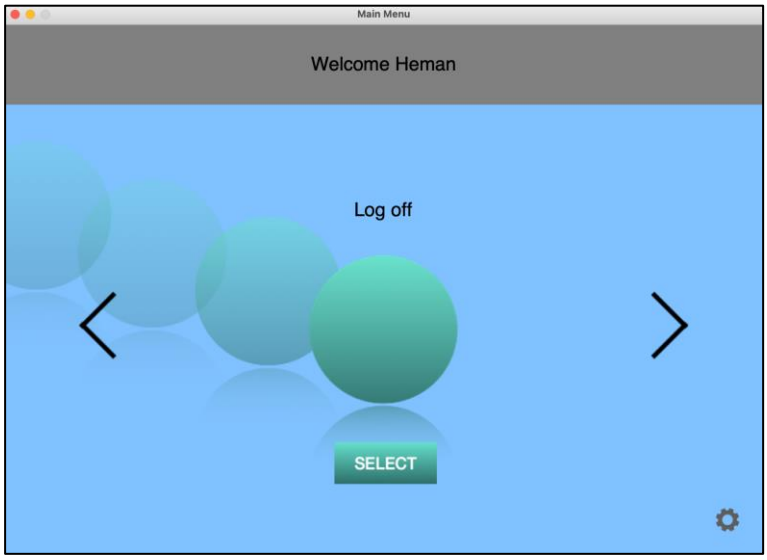
Table: Progress

	progressID	userID	quizID	questionID	correct	time
	Filter	Filter	Filter	Filter	Filter	Filter
1	132	1	20	18	True	2021-02-04 18:49:11
2	133	1	20	19	True	2021-02-04 18:49:32
3	134	1	20	20	False	2021-02-04 18:49:54
4	135	2	20	18	False	2021-02-06 18:50:00
5	136	2	20	19	True	2021-02-06 18:50:21
6	137	2	20	20	False	2021-02-06 18:50:43
7	138	5	19	16	False	2021-02-06 18:51:00
8	139	5	19	17	False	2021-02-06 18:51:21
9	141	6	20	18	True	2021-02-06 18:52:00
10	142	6	20	19	True	2021-02-06 18:52:21
11	143	6	20	20	True	2021-02-06 18:52:43
12	144	2	19	16	False	2021-02-06 18:53:00
13	145	2	19	17	True	2021-02-06 18:53:21
14	146	5	20	18	True	2021-02-06 18:55:00
15	147	5	20	19	False	2021-02-06 18:55:21
16	148	5	20	20	True	2021-02-06 18:55:43
17	149	2	20	18	True	2021-02-08 18:50:00
18	150	2	20	19	True	2021-02-08 18:50:21
19	151	2	20	20	True	2021-02-08 18:50:43
20	152	2	20	18	False	2021-02-09 18:50:00
21	153	2	20	19	True	2021-02-09 18:50:21
22	154	2	20	20	True	2021-02-09 18:50:43

I concluded my program with a function to log off:

```
def logOff(self):
    mainLogin.mainLoginPage(mainLogin.titleFont, mainLogin.buttonFont, mainLogin.defaultFont)
```

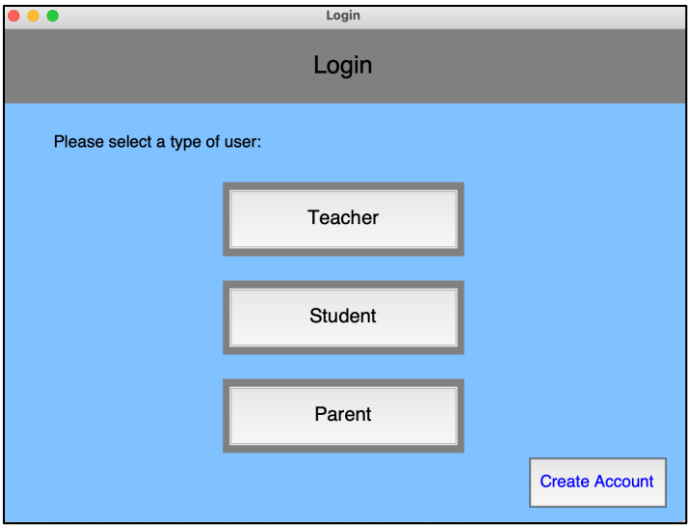
This was done simply by redirecting the user back to the main log in page.



I finalised the selectButtonPressed function to destroy this page before redirecting the user back to the main login page:

```
def selectButtonPressed(self, previousPage, username, menuOption):
    if menuOption.cget('text') == 'Play Quiz':
        ## Destroy previous page
        previousPage.destroy()
        ## Initiate playQuiz function
        self.playQuiz(username, Login.titleFont, Login.defaultFont, Login.buttonFont)
    elif menuOption.cget('text') == 'View Quizzes':
        ## Destroy previous page
        previousPage.destroy()
        ## Initiate viewQuizzes function
        self.viewQuizzes(username, Login.titleFont, Login.defaultFont, Login.buttonFont)
    elif menuOption.cget('text') == 'Create Quiz':
        ## Destroy previous page
        previousPage.destroy()
        ## Initiate createQuiz function
        self.createQuiz(username, Login.titleFont, Login.defaultFont, Login.buttonFont)
    elif menuOption.cget('text') == 'View Classes':
        ## Destroy previous page
        previousPage.destroy()
        ## Initiate viewClasses function
        self.viewClasses(username, Login.titleFont, Login.defaultFont, Login.buttonFont)
    elif menuOption.cget('text') == 'Log off':
        ## Destroy previous page
        previousPage.destroy()
        ## Initiate logOff function
        self.logOff()
```

Which closed the main menu page and brought the user back here:



3.3 Evaluation

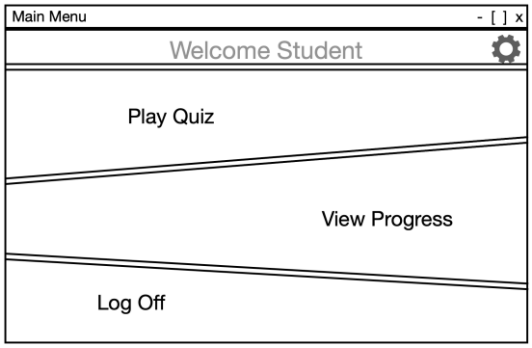
To evaluate my program, I first allowed my client, Mrs Parker, to perform a black-box test on the system. I asked for feedback for each part of my program:

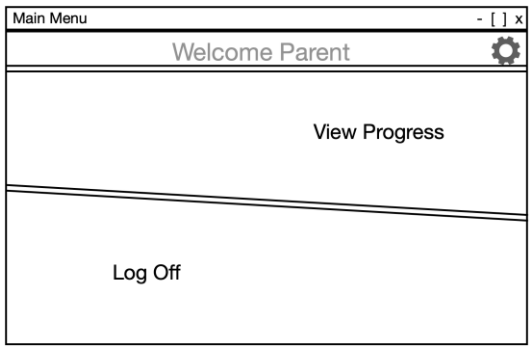
Part of Program:	**Feedback:**
General Interface	*In general, the interfaces are wonderful! The login pages are all user-friendly and easy to use. The main menu is so elegant and modern – probably my favourite page, along with the amazing animations between choosing the different options. I also love the use of colour, especially the use of a mild green and red after answering a question. Also, the separations in the view class, view quizzes and create quiz options are amazing; I can easily see everything all at once and change between different quizzes or students effectively. Finally, the pie charts and the line graphs are probably the most useful part of this program for me. They are so easy to read and all the information is just where you want it!*
Login	*As mentioned before, the login pages were very easy to use – everything is perfectly labelled and easy to follow. It's great how it's separated into three different logins for the teachers, students and parents, however, as of yet, there is no response for the student and parent login.*[1] *Creating an account seems fairly easy, with everything you need right in front of you. All 'errors' are clearly shown so you know just what's missing if you've forgotten any bits upon creation. The red outlines of the boxes also help to pinpoint exactly where the error is. As for the forgotten password page, although it is easy to use and whatnot, there is no actual place where I can reset the password. Same applies for having a blocked account and creating an account.*[2]
Quiz Creation	*These pages are lovely! It's so easy to use and you have so many functionalities. Granted the background options are limited, but that's not too important for me. I find the backgrounds amazing anyways! I love the previews that are shown for each question when you select them – it really helps when creating the questions. I was hoping however for an option to edit the questions and the quizzes.*[3] *There is also an option to view the quizzes but not to edit and sometimes I find that I want to start off a quiz and add to it later, or I see a question that I want to change, and as of now I can't.*
Viewing Quizzes	*Yeah as I said before, even when viewing the quizzes, it's really easy to use and see and change between quizzes but what's the point of viewing a quiz if you can't make changes?*
Playing a Quiz	*When choosing to play a quiz, I'm not a fan of the initial page where the quizzes are just numbered.*[4] *It makes it really hard to just find what you're looking for. Granted the details are shown when you select one but it's still time-consuming to find the quiz you want. It's much better in the view classes option where the subject and level of the quiz are shown. Perhaps there should be an option to name the quizzes*[5] *so it becomes much easier to find the quiz you're looking for. The subject and level can perhaps be used to filter out the quiz you want*[6] *if many quizzes are created. Apart from that, the actual experience of playing a quiz isn't too bad. Whilst user-friendly and easy to use, a multiplayer aspect*[7] *as well as a gaming aspect*[8] *would've been appreciated, especially for the lower years.*
Progress Tracking	*Again, I like the partitions between the page when you first click view classes since I can see everything all in one place, I can easily switch between my*

	students, their quizzes and see all their progress for them. It's really helpful that their overall result is displayed after all their individual progress since I can quickly skim over their marks when time is short for me. The next page is amazing as well where I can see their results for individual attempts and a pie chart of their collective work. The pie chart uses beautiful colours, as when playing a quiz, and is really easy to understand. I absolutely love the next page! It encompasses all that I need really and the use of a line graph is perfect for displaying progress in a nice, quick, easy format. The tables of progress for the selected student and all other students are also really helpful when comparing progress. The only suggestion here would be to display overall results as in the previous page within the tables for the students[9] so it becomes quicker to see an overall sort of progression. It would help as well if I could see what the student chose as their answer for each question[10] since I might be able to use it to focus my teaching and rectify and misunderstandings. Oh and it would be great if I was able to separate out my students into separate classes[11] rather than seeing them all in one place.
Overall Feedback	*Overall, the program is great! It does more or less what is required and the included functionalities are amazing. I love the interfaces and the simplicity and the elegance of the main menu. I do admit, however, that it could be perfected. Whilst it does the most important aspects required of it, there are small adjustments or extensions that could be made. In addition to what I've mentioned previously, it would be nice to have some quizzes already ready to use[12], since they can be time-consuming to do. This may also allow for students to get randomised questions[13], so they don't simply remember answers and I'm not required to constantly change them. Lastly, but perhaps more importantly, it would be incredibly useful if the program could automatically detect areas or certain questions within quizzes that students have difficulty with and perform poorly in.[14] this would help me significantly in targeting certain topics in my lessons.*

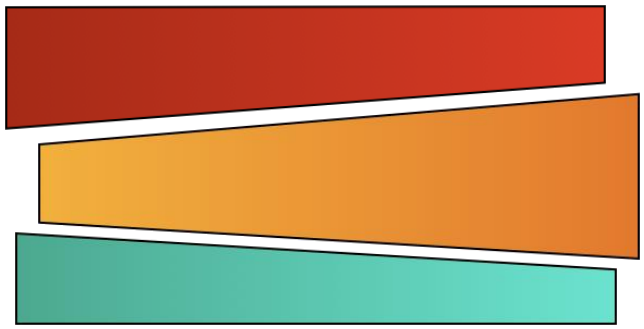
There is generally a positive outlook on the program by my client. Areas which she especially liked included the interfaces, the simplicity, the main menu and the display of progress. There were certain suggestions that she suggested however.

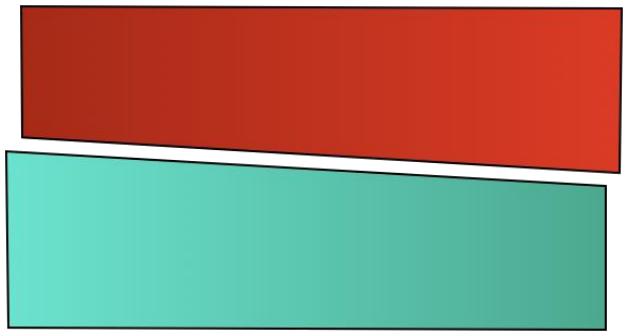
1. '*there is no response for the student and parent login*' - when logging in, there was no response from logging in as a student or parent. This was because I hadn't yet created a separate menu page for the students or parents. I also couldn't redirect them to the admin menu page since there are additional accessibilities that they should not have access to. As a student, their accessibility should be restricted to playing a quiz, viewing only their own progress and logging off. As a parent, accessibility should be restricted to viewing the progress of their child(ren) and logging off. These functions have already been created. This means that changes only have to be made with the actual layout of the main menu page. This would require me to either modify the current menu page to allow only three or two options respectively, modifying the menu options and their animations accordingly, or I can redesign the menu page completely for a student and a parent. I chose to redesign the menus in this format:





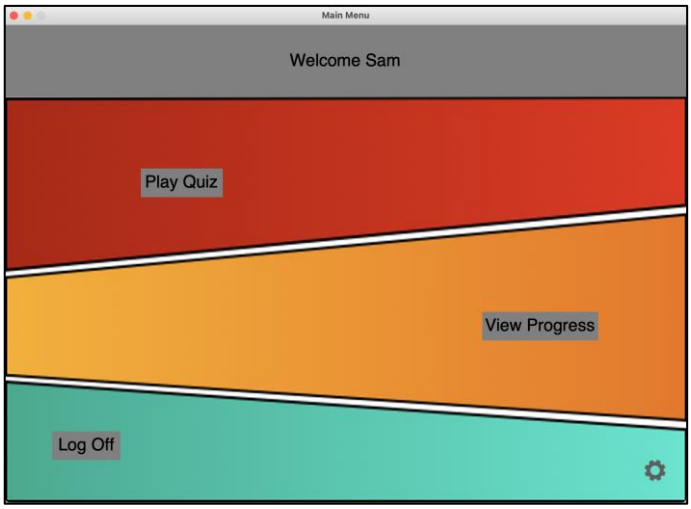
To do this, I created these images:





I imported these images and bound the relevant function to them. I also created labels as per the design above. The student menu page:

```
def studentMainMenuPage(self, username, titleFont, buttonFont, defaultFont):
    mainMenuStudent = tk.Tk()
    mainMenuStudent.title("Main Menu")

    ## Retrieves the assigned width and height for the main menu page
    mainScreenWidth = mainMenu.getScreenWidth()
    mainScreenHeight = mainMenu.getScreenHeight()

    ## Sets the dimensions of the page and sets it to non-resizable
    mainMenuStudent.geometry(str(mainScreenWidth)+'x'+str(mainScreenHeight))
    mainMenuStudent.resizable(False, False)

    ## Creates a canvas for all the menu options
    canvas = tk.Canvas(mainMenuStudent, height = mainScreenHeight, width = mainScreenWidth, highlightthickness=0, relief='ridge')
    canvas.pack(fill = 'both', expand = True)

    ## Retrieve first name from database using username
    firstName = viewingFromDatabase.viewFromDatabase(c, False, False, 'firstName', 'username', "'"+username+"'", 'UserDetails')
    ## Create welcome label using first name
    welcomeLabel = tk.Label(canvas, text = "Welcome "+firstName[0][0], bg = 'gray', font = titleFont)
    welcomeLabel.place(relx = 0, rely = 0, relheight = 0.15, relwidth = 1)

    ## Import menu option and settings images and resize them
    menuOptionOneImage = Image.open('StudentMenuOptionOne.png')
    menuOptionOneImage = menuOptionOneImage.resize((1007,265))

    menuOptionTwoImage = Image.open('StudentMenuOptionTwo.png')
    menuOptionTwoImage = menuOptionTwoImage.resize((1007,315))

    menuOptionThreeImage = Image.open('StudentMenuOptionThree.png')
    menuOptionThreeImage = menuOptionThreeImage.resize((1007,190))

    settingsImage = Image.open('Settings.png')

    ## Convert them to tkinter images
    tkMenuOptionOneImage = ImageTk.PhotoImage(menuOptionOneImage)
    tkMenuOptionTwoImage = ImageTk.PhotoImage(menuOptionTwoImage)
    tkMenuOptionThreeImage = ImageTk.PhotoImage(menuOptionThreeImage)
    tkSettingsImage = ImageTk.PhotoImage(settingsImage)
    ## Add the images to the canvas
    menuOptionOne = canvas.create_image(-3,99, image = tkMenuOptionOneImage, anchor = 'nw')
    menuOptionTwo = canvas.create_image(-3,270, image = tkMenuOptionTwoImage, anchor = 'nw')
    menuOptionThree = canvas.create_image(-3,514, image = tkMenuOptionThreeImage, anchor = 'nw')
    settings = canvas.create_image(935,635, image = tkSettingsImage, anchor = 'nw')

    ## Create labels for the menu options
    optionOneLabel = tk.Label(canvas, text = "Play Quiz", bg = 'gray', font = titleFont)
    optionOneLabel.place(relx = 0.2, rely = 0.3, relheight = 0.06, relwidth = 0.12)

    optionTwoLabel = tk.Label(canvas, text = "View Progress", bg = 'gray', font = titleFont)
    optionTwoLabel.place(relx = 0.7, rely = 0.6, relheight = 0.06, relwidth = 0.17)

    optionThreeLabel = tk.Label(canvas, text = "Log Off", bg = 'gray', font = titleFont)
    optionThreeLabel.place(relx = 0.07, rely = 0.85, relheight = 0.06, relwidth = 0.1)

    ## Bind the images and labels to destroy the main meny page and go to the relevant function
    canvas.tag_bind(menuOptionOne, "<Button-1>", lambda x: (mainMenuStudent.destroy(), self.playQuiz(username, Login.titleFont, Login.defaultFont, Login.buttonFor
    optionOneLabel.bind("<Button-1>", lambda x: (mainMenuStudent.destroy(), self.playQuiz(username, Login.titleFont, Login.defaultFont, Login.buttonFont)))

    canvas.tag_bind(menuOptionTwo, "<Button-1>", lambda x: (mainMenuStudent.destroy(), self.viewClasses(username, Login.titleFont, Login.defaultFont, Login.buttor
    optionTwoLabel.bind("<Button-1>", lambda x: (mainMenuStudent.destroy(), self.viewClasses(username, Login.titleFont, Login.defaultFont, Login.buttonFont)))

    canvas.tag_bind(menuOptionThree, "<Button-1>", lambda x: (mainMenuStudent.destroy(), self.logOff()))
    optionThreeLabel.bind("<Button-1>", lambda x: (mainMenuStudent.destroy(), self.logOff()))

    canvas.tag_bind(settings, "<Button-1>", lambda x: self.settingsMenu(username, Login.titleFont, Login.buttonFont, Login.defaultFont))

    mainMenuStudent.mainloop()
```

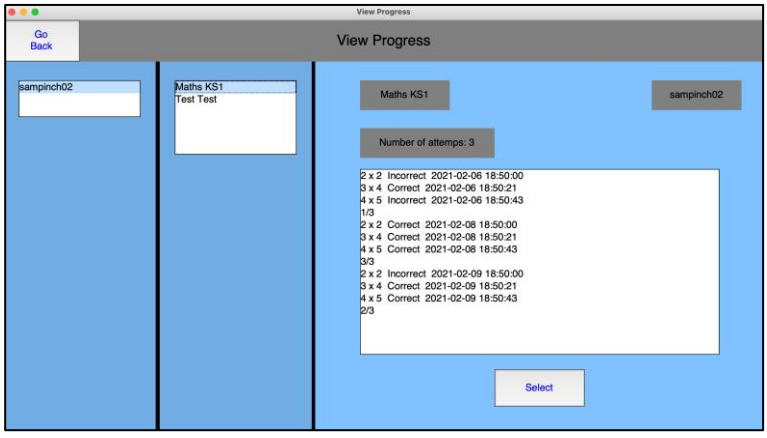


I logged in here as sampinch02. Whilst I was able to bind the image and label of the play quiz and log off option to their respective functions, the view progress option required some modification since I would be using the view classes function and all its related functions for this. I had to modify it to exclude all other students and show the progress of only the student user. When going on this function from this page without prior modification, no users were listed in left hand frame since no users would have the user ID of a student user as their admin ID. To fix this I did this:

```
## Retrieve account type for given username from database
accountType = viewingFromDatabase.viewFromDatabase(c, False, False, 'accountType', 'username', "'"+username+"'", 'UserDetails')
accountType = accountType[0][0]
## If Admin set title of page and welcome label text to 'View Classes'
if accountType == 'Admin':
    title = 'View Classes'
## If Student set title of page and welcome label text to 'View Progress'
elif accountType == 'Student':
    title = 'View Progress'
```

```
## If Admin retrieve all students for that user
if accountType == 'Admin':
    ## Retrieve userID from database with given username
    userID = viewingFromDatabase.viewFromDatabase(c, False, False, 'userID', 'username', "'"+username+"'", 'UserDetails')
    ## Retrieve students' usernames from database with given adminID and 'Students' as account type
    students = viewingFromDatabase.viewFromDatabase(c, False, False, 'username', 'adminID', str(userID[0][0])+" AND accountType = 'Student'", 'UserDetails')
## If Student add username to students list in correct format
elif accountType == 'Student':
    students = [[username]]
```

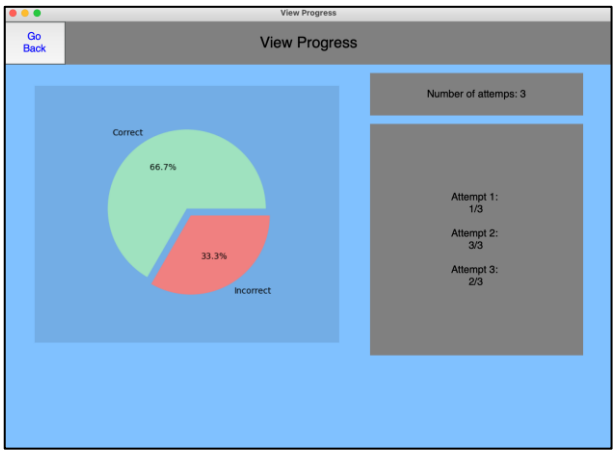
At the beginning of the view classes function, I retrieved the account type of the username provided as a parameter to the function and using this, created if statements to see whether an admin user or a student user. Using this, I changed the title of the page and the welcome label text accordingly to either 'View Classes' for an admin user and 'View Progress' for a student. I also used it to choose whether to retrieve all students using the user ID of the username (for an admin user) or instead simply display the username of a student user.
Logged in as sampinch02, this produced:



Next, I had to prevent the user from having the option to view full progress since that would show them the results of other users. Therefore, following the same solution I wrote this in the displayProgress function:

```
## Retrieve account type for given username from database
accountType = viewingFromDatabase.viewFromDatabase(c, False, False, 'accountType', 'username', "'"+username+"'", 'UserDetails')
accountType = accountType[0][0]
## If Admin create button to view full progress
if accountType == 'Admin':
    ## Create button to view full progress
    viewFullProgressButton = tk.Button(mainframe, text = "View Full Progress", highlightbackground = '#424949', fg = 'blue', font = defaultFont, command = la
    viewFullProgressButton.place(relx = 0.4, rely = 0.82, relheight = 0.1, relwidth = 0.2)
```

Which produced:

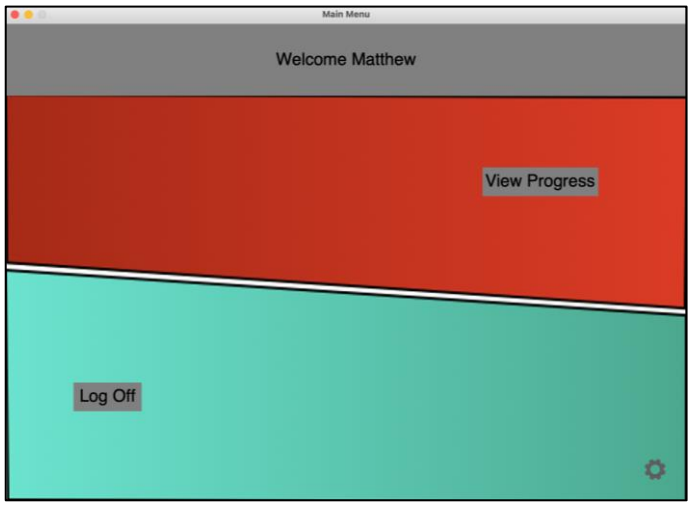


The parent menu page:

```
def parentMainMenuPage(self, username, titleFont, buttonFont, defaultFont):
    mainMenuParent = tk.Tk()
    mainMenuParent.title("Main Menu")

    ## Retrieves the assigned width and height for the main menu page
    mainScreenWidth = mainMenu.getScreenWidth()
    mainScreenHeight = mainMenu.getScreenHeight()

    ## Sets the dimensions of the page and sets it to non-resizable
    mainMenuParent.geometry(str(mainScreenWidth)+'x'+str(mainScreenHeight))
    mainMenuParent.resizable(False, False)

    ## Creates a canvas for all the menu options
    canvas = tk.Canvas(mainMenuParent, height = mainScreenHeight, width = mainScreenWidth, highlightthickness=0, relief='ridge')
    canvas.pack(fill = 'both', expand = True)

    ## Retrieve first name from database using username
    firstName = viewingFromDatabase.viewFromDatabase(c, False, False, 'firstName', 'username', "'"+username+"'", 'UserDetails')
    ## Create welcome label using first name
    welcomeLabel = tk.Label(canvas, text = "Welcome "+firstName[0][0], bg = 'gray', font = titleFont)
    welcomeLabel.place(relx = 0, rely = 0, relheight = 0.15, relwidth = 1)

    ## Import menu option and settings images and resize them
    menuOptionOneImage = Image.open('ParentMenuOptionOne.png')
    menuOptionOneImage = menuOptionOneImage.resize((1009,325))

    menuOptionTwoImage = Image.open('ParentMenuOptionTwo.png')
    menuOptionTwoImage = menuOptionTwoImage.resize((1009,355))

    settingsImage = Image.open('Settings.png')

    ## Convert them to tkinter images
    tkMenuOptionOneImage = ImageTk.PhotoImage(menuOptionOneImage)
    tkMenuOptionTwoImage = ImageTk.PhotoImage(menuOptionTwoImage)
    tkSettingsImage = ImageTk.PhotoImage(settingsImage)

    ## Add the images to the canvas
    menuOptionOne = canvas.create_image(-3,97, image = tkMenuOptionOneImage, anchor = 'nw')
    menuOptionTwo = canvas.create_image(-5,352, image = tkMenuOptionTwoImage, anchor = 'nw')
    settings = canvas.create_image(935,635, image = tkSettingsImage, anchor = 'nw')
```

```
    ## Create labels for the menu options
    optionOneLabel = tk.Label(canvas, text = "View Progress", bg = 'gray', font = titleFont)
    optionOneLabel.place(relx = 0.7, rely = 0.3, relheight = 0.06, relwidth = 0.17)

    optionTwoLabel = tk.Label(canvas, text = "Log Off", bg = 'gray', font = titleFont)
    optionTwoLabel.place(relx = 0.1, rely = 0.75, relheight = 0.06, relwidth = 0.1)

    ## Bind the images and labels to destroy the main meny page and go to the relevant function
    canvas.tag_bind(menuOptionOne, "<Button-1>", lambda x: (mainMenuParent.destroy(), self.viewClasses(username, Login.titleFont, Login.defaultFont, Login.buttor
    optionOneLabel.bind("<Button-1>", lambda x: (mainMenuParent.destroy(), self.viewClasses(username, Login.titleFont, Login.defaultFont, Login.buttonFont)))

    canvas.tag_bind(menuOptionTwo, "<Button-1>", lambda x: (mainMenuParent.destroy(), self.logOff()))
    optionTwoLabel.bind("<Button-1>", lambda x: (mainMenuParent.destroy(), self.logOff()))

    canvas.tag_bind(settings, "<Button-1>", lambda x: self.settingsMenu(username, Login.titleFont, Login.buttonFont, Login.defaultFont))

    mainMenuParent.mainloop()
```

Logging in as mattbumpus03:



Whilst the log off function and settings function worked as desired here. There were some issues with the view progress function. Initially, there was an error involving the title variable which hasn't been created when accessing the function from this page since the account type is neither Admin nor Student. I added 'or Parent' at the end of the elif statement for the Student so that for both students and parents the title and the welcome label read 'View Progress'. This didn't work however for the second if statement, however, since the program would simply pass in the username to the list box. This wouldn't show them the progress of their children but instead of themselves, which of course, there would not be any record of since they don't have access to that part of the program. This would require me to pass in the username of the child(ren) of the parent user. This presented a flaw in the system. A parent account had not been linked to any student user(s) and hence, the program hadn't a way of knowing which student's progress to display. To make this work, in future development I would have to create an extra field within the UserDetails table or perhaps create a new table altogether, which stored the user ID of the student(s) that were linked to a parent user. This would require change within the database class to align the number of value placeholders with the extra field in the UserDetails table or the new table. The create account function would have to be modified to allow parent users to choose their child(ren). Alternatively, the create account process could instead be a more in-house process as done in other programs, in which the organisation who buys the program creates accounts for all students and parents themselves. Finally, it would require some change in the view classes function where the student username would be retrieved from the database using the parent username and inserted into the list box. I had previously created an if statement within the settingsMenu function which prevented student and parent users from having access to the review users function, and so no changes were needed here.

2. *'there is no actual place where I can reset the password. Same applies for having a blocked account and creating an account'* – whilst I had created a function to review users within the settings of the main menu page, the function hadn't been finalised to change any review status of the users; the function merely displayed a table of all users in need of reviewing. The issue encountered within my development of this section was the for loop failed to bind button events to the cells in the table with different values; all cells took the final value for certain variables and therefore didn't work as desired. To fix this I did the following:

```
## Retrieve children from tableFrame and assign to list
wlist = tableFrame.winfo_children()
## For each child (i.e. each cell in table)
for each in range(len(wlist)):
    ## Find row number by integer division on current iteration by 6 (number of columns)
    ## Find username associated to selected row in cells dictionary
    username = cells[(each // 6, 1)]
    ## Bind button event which initiates reviewUser function with username as parameter
    wlist[each].bind("<Button-1>", lambda x, username = username: self.reviewUser(username))
```

I modified this section of the reviewUsers function. In this version, I changed the for loop to work as integers incrementing to the length of wlist, rather than for each value in it. This allowed me to avoid creating an unnecessary counter variable. Using the same calculation as before, I did an integer division on the current iteration by the value of 6 (the number of columns in the table) and this got me the row number that was selected. Using this, I assigned the value of (row number, 1) of the cells dictionary to a username variable. With this username, I passed it on to the reviewUser function where all the details were displayed, and the admin user had the opportunity to change the review status of the user in question. Although this is generally what was done when developing, I had to extend the lambda function statement to lambda x, username = username:, so that the values were different for each bind. This now triggered the reviewUser function.
To finish off this process in future development, the reviewUser function would have to show the admin user the full username of the selected user under review, the type of review and an option to revert it. This would differ for each review type. For example, if the review ID was 2, indicating a new account, there would have to be an option to verify the account. This would include showing the admin user all the relevant information, such as the username, first name, last name, account type, admin user and as per the changes described above, if a parent user, the user ID of the child(ren). Once the admin user is happy with all the information (which they could perhaps indicate with a check box), they should have access to a button that verifies the user and changes the review ID from 2 to 1, indicating no review, in the database. This can be done using the update function in the database class. If the review ID was 3, indicating that the user has forgotten their password, there would have to be an option to reset it. The admin user shouldn't be entrusted to see the password of the user and so it won't be shown. In this case, the program should display the username of the selected student, and simply have two entry boxes which asks for a password, and another asking to confirm the password. The program should make sure that these passwords are the same before proceeding. It should also validate the password, which can be done using the checkValidPassword function created for the create account process. This can also be linked to the accountFailed function which pinpoints to the user what parts of the password is not valid. The program should then provide a button which allows the user to confirm this process and firstly change the password in the database using the hash function and the update database function, and then update the review ID to 1. Perhaps it would be beneficial for the program to allow users to change their own passwords within the my account option in settings, so the admin user could reset the password which is shared with the user, so they are then able to make changes of their own. This may improve security since the admin user is not able to see the password when being reset (even if hidden by asterisks). If the review ID was 4, indicating that a user's account has been blocked, there would have to be an option to revert this block. This would simply involve a button at the disposal of the admin user which updates the review ID to 1. Perhaps if the program detects multiple consecutive blocks in a short period of time, a warning can be shown to both the user and the admin user, maybe even a risk of account deletion to minimise the risk of a brute force attack.

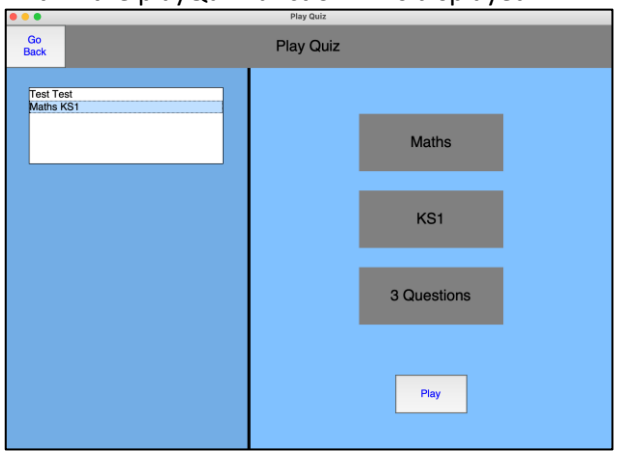
3. *an option to edit the questions and the quizzes* – in both the create quiz function and the view quizzes function, the program could be extended in future development to allow the quizzes to be edited. It would significantly improve convenience for my client and other users using the program. Perhaps the best way to do this, would be to modify the displayQuestion function. Instead of the

labels that appear when following through from the create quiz and view quizzes function, there could be entries, as per the create quiz page, which held the values within the database. A save button can be added to allow the user to confirm any changes to the entries in which case the program can update the values in the database using the update database function from the database class if needed. The option for the background and character limit can also be shown in such a function. This would require, of course, adjusted compatibility with the other functions that make use of the displayQuestion function such as the play quiz function. Alternatively, the function could be separated to fit each of the three preceding functions in question, which would improve modularity and reusability in the program. However, this would lead to a lot of repeated code and perhaps, in this way, would reduce efficiency.

4. *I'm not a fan of the initial page where the quizzes are just numbered* – when a user chooses the option to play a quiz, the quizzes are displayed within a list box named Quiz 1, Quiz 2 and so on. This could easily be modified to show instead the subject and level as in the view classes function. This would improve convenience for the user by allowing them to more easily search a quiz of their choice. It would require that I retrieve the relevant details from the database for the selected quiz. This, however, is already done within my playQuiz function, since the details are needed later on to show the subject, level and number of questions on the right hand side. This required simply this change:

```
for i in range(len(quizzes)):
    quizListbox.insert('end', quizzes[i][1] + ' ' + quizzes[i][2])
```

Within the playQuiz function. This displayed:



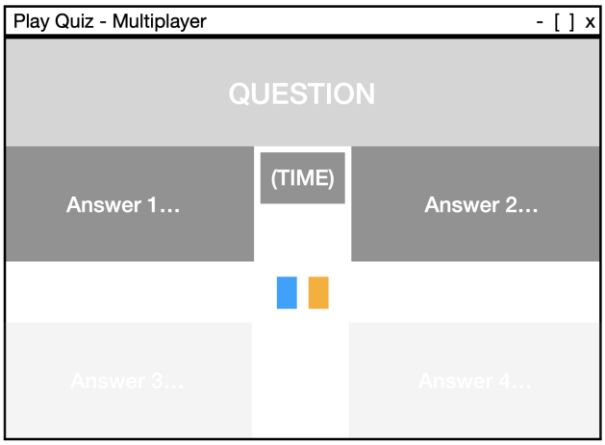
As desired.

5. *an option to name the quizzes* – this would require me to extend my database to allow a field that stored the name for each quiz. It would further require corresponding changes to the add to database function in the database class to align the new amount of placeholder values to the new amount of fields for this table, i.e. 5, and the create quiz function to allow the user to name a quiz upon creation. Perhaps even, with the modifications mentioned above, it would require changes to the view quizzes function to allow the name of a quiz to be edited. Finally, the quiz list boxes within the play quiz and view quizzes functions can be modified to read the name of the quiz instead of the subject and level.

6. *The subject and level can perhaps be used to filter out the quiz you want* – the play quiz and view quizzes functions can be extended to make quizzes easier to find, which will increase convenience for a user especially when many users are involved and many quizzes have been created. This could involve the use of combo boxes at the top of the page, one of which shows all the subject of quizzes that have been created which can be retrieved from the database, and another showing the level of

all the quizzes for the selected subject. To do this, all retrieved subjects would be inserted into the combo box, and an event would be bound to the user selection such that all the levels associated with the selected subject is inserted into the second combo box. This second combo box would be emptied before inserting the new values which change with the selection to remove any previous contents within it. With these modifications, when a user first selects the option to play a quiz or view quizzes, all quizzes will be displayed with their name. The user can then select a specific subject in the first combo box which will empty the quiz list box and insert all quizzes with the selected subject. The user can then change the selection in the second combo box which will have the levels of all quizzes of the subject chosen. Upon this selection, the quiz list box will then be emptied and all quizzes which match both criteria will be inserted. The combo boxes should also have the option of 'All' – the default value – so the user can go back to seeing all the quizzes.

7. *a multiplayer aspect* – when playing a quiz, a user cannot play against other users in real time. One way of resolving this issue would be to allow internet or Bluetooth connectivity. However, this would be a really complicated process. An alternative may include the next suggestion made by my client: 8. *a gaming aspect*. When playing a quiz, the user can be prompted whether they would like to play single player or multiplayer. If single player, it can continue as before. If they select the multiplayer option, a new play page can be introduced. This page can look like this:



Here, the question and time is displayed at the top, and the answer options are displayed in the four corners of the remaining space (or less if less than four answers). The two rectangles in the middle represent avatars for each player which they could control using the keyboard, with different keys, e.g. player one can use the arrow keys to direct their avatar towards the correct answer and player two can use WASD to do so. This clearly limits the extent of the multiplayer functionality and restricts the number of players to a maximum of maybe four, to prevent conflicts between keys and enough space for each player. It also makes it more complicated to record data since more than user is involved and only one is logged in. To combat this, before starting a multiplayer quiz, the user could be prompted asking for the number of players and could ask between a choice for these players to log in to their own accounts, or perhaps play as a guest user. The program could then be extended to store the progress for each account that has been logged in. To retrieve the answers for each player, the x and y position of their avatar can be recorded and matched against the options on the screen to see which they choose. This multiplayer feature and the use of avatars and perhaps a relevant background could encourage kids, especially those in younger years to play, as suggested by my client. This does pose limitations however. Namely, when playing answer input questions. These types of question will not be able to follow the format shown above. This might mean that this functionality might only have to be limited to pre-created quizzes which contain only multiple choice questions. Or instead, it might be the case that each player has to take it in turns when answering the questions. This is not perfect, however, because the last player to answer the question might have an unfair advantage from seeing the question earlier than the others and potentially seeing their answers. It would also cause issues with the time limit provided for the question, since it may have to be extended for multiple players to avoid being partial to certain players. However, although this multiplayer feature is limited to 4 players maximum, this could still significantly reduce the time taken for my client to attain results for all students by a minimum of a half and a maximum of a quarter, which is a major improvement from the original program. It would maximise engagement from the students from both the competitive nature and the gaming aspect, and in turn, results.

9. *display overall results as in the previous page within the tables for the students* – when viewing the full progress of a student, although all questions are shown for each student, and whether correct or not, it does not show the overall result as it does in the displayProgress function. This can be done in the same way as in the displayProgress function, and at the end of every 6th iteration (since 6 columns), a new row can be added to show the overall result. This can be done by simply modifying the data list and place the results for each quiz as a new line with blank cells where necessary. As done previously, it would involve counting up the number of 'True' values retrieved from the database for each attempt of the selected quiz. This can be done for both tables: that of the selected student, and that of all the other students.

10. *see what the student chose as their answer for each question* – whilst the user can view the quizzes and questions answered by all students, they are unable to see the answer they chose or wrote when answering a question. With this information, a teacher may be able to better guide lessons or revision. In order to do this, the Progress table would have to be extended to have a field for the inputted answer, adjusting the database class accordingly. This value can then be retrieved from the database and added to the progress tables as required. A scrollbar frame may need to be introduced for the table to go both in a vertical and horizontal direction to display all the information.

11. *separate out my students into separate classes* – this would require an extension to the database of a 'class' field within the UserDetails table. There could also be a separate 'Classes' table that encompassed all information about the class, e.g. subject, year group, etc. All students can then be put into a class accordingly. This can either be done when creating an account. i.e. if the combo box for the account type read 'Student', an option to select a class can appear, or this can be done by the teacher when they verify a new account. This may help to prevent any errors in class selection. Using this field in the database, there can be an additional list box in the view classes menu option which displays the classes. Upon selection in this list box, the students list box can then be emptied and modified accordingly, inserting the usernames of all users where the class is equal to the selected class. The rest of the functions can remain the same from here.

12. *some quizzes already ready to use* – as discussed when designing the program, there could've been some quizzes included within the program that had been already created for use for users. This could show teachers what sort of layout the questions can take when they need to create a quiz, and if core subjects are already covered by the program, it saves the teachers from making quizzes. This can be combined with the next suggestion by my client: 13. *randomised questions* – questions in the pre-created quizzes can follow a similar format on every time it is run, however, the variables in some of them can be changed, e.g. in maths problems. This allows some flexibility and diversity in the quizzes. The answers to these questions can then be calculated using a relevant equation. The extent of randomised questions, however, is limited, mainly to mathematical and other problems that can be solved immediately with a formula. Other core subjects such as English and Religious Studies may be limited in their randomisation capabilities. This doesn't mean randomised questions can't be provided necessarily. Randomised questions for these subjects can be provided not by randomising certain variables within the questions, but by creating a great surplus of questions which can be randomly put together to form a quiz of a set number of questions. The randomisation capability is obviously much more limited however.
To do this, I created a quiz in the database:

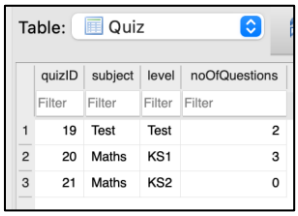


Table: Quiz

	quizID	subject	level	noOfQuestions
	Filter	Filter	Filter	Filter
1	19	Test	Test	2
2	20	Maths	KS1	3
3	21	Maths	KS2	0

I set the number of questions as 0 here to allow the user to decide how many questions they want to set.

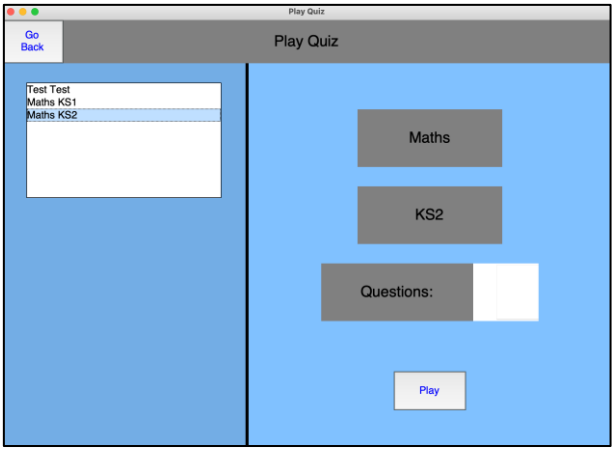
I modified the playQuiz function to create an entry allowing the user to enter how many questions they would like to play:

```
noOfQuestionsEntry = tk.Entry(noOfQuestionsFrame)
```

I then modified the displayQuizDetails function to show the entry for quizzes with 0 as the noOfQuestions in the database:

```
## If random quiz with no set questions
if quizDetails[0][3] == 0:
    noOfQuestionsFrame.place(relx = 0.2, rely = 0.52, relheight = 0.15, relwidth = 0.6)
    noOfQuestionsLabel.config(text = 'Questions:', bg = 'gray', font = titleFont)
    noOfQuestionsLabel.place(relheight = 1, relwidth = 0.7)
    noOfQuestionsEntry.place(relx = 0.8, relheight = 1, relwidth = 0.3)
## If quiz with no questions
else:
    noOfQuestionsFrame.place(relx = 0.3, rely = 0.52, relheight = 0.15, relwidth = 0.4)
    noOfQuestionsLabel.config(text = str(quizDetails[0][3]) + ' Questions', bg = 'gray', font = titleFont)
    noOfQuestionsLabel.place(relheight = 1, relwidth = 1)
```

This produced:



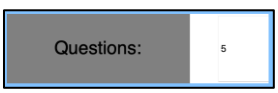
This now allowed the user to choose how many questions were going to be set. I had to add this bit of code to hide the entry when the user changed between quizzes:

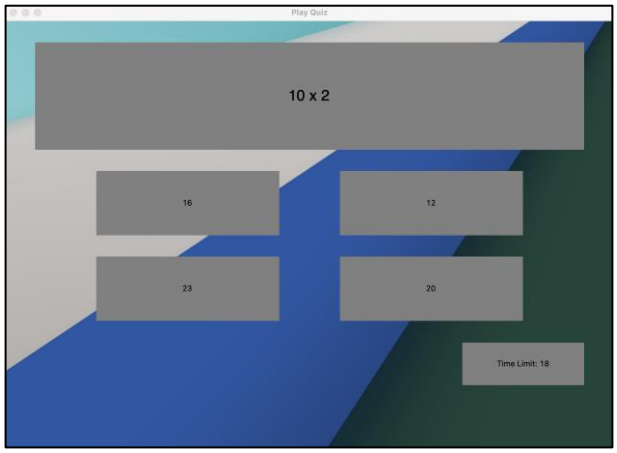
```
## Hide noOfQuestionsEntry in case of changing quizzes
noOfQuestionsEntry.place(relheight = 0, relwidth = 0)
```

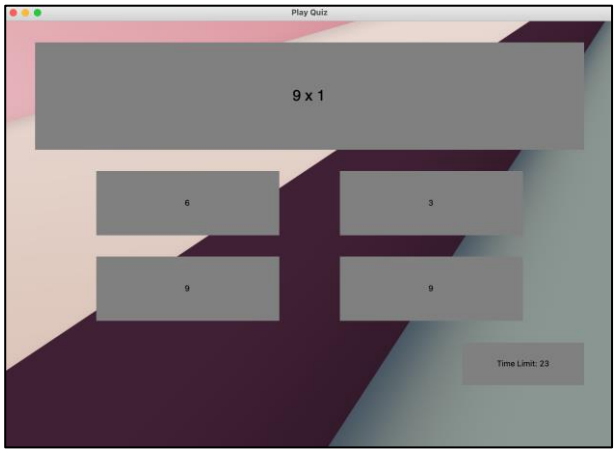
I modified the playSelectedQuiz function:

```
## If random quiz with no set questions
if quizDetails[0][3] == 0:
    questions = []
    ## If quiz ID of selected quiz is 21
    if quizDetails[0][0] == 21:
        ## For each in range chosen number of questions
        for each in range(int(noOfQuestionsEntry)):
            ## Create two random variables between 1 and 12, and a variable with a random question type
            numberOne = random.randint(1,12)
            numberTwo = random.randint(1,12)
            questionType = random.choice(['Multiple Choice','Answer Input'])
            if questionType == 'Multiple Choice':
                ## Add as a question to questions list
                questions.append((None, 21, questionType, str(numberOne) + ' x ' + str(numberTwo), 4, str(numberOne * numberTwo), str(random.randint(1,
            elif questionType == 'Answer Input':
                questions.append((None, 21, questionType, str(numberOne) + ' x ' + str(numberTwo), 1, str(numberOne * numberTwo), None, None, None, 5,
## If quiz with set questions
else:
    ## Retrieve all questions from database for selected quiz
    questions = viewingFromDatabase.viewFromDatabase(c, True, False, None, 'quizID', str(quizDetails[0][0]), 'Question')
```

Due to the need for randomisation, I had to create the quiz within the program itself. If the quiz ID is found to be 21, the program generates a random question, decides whether multiple choice or answer input and creates answers for the user to select from (if necessary). It also chooses a random background . If the noOfQuestions field from the database is not 0, the questions are just retrieved as before. This produced:







And allowed the user to play as normal. There are some obvious limitations to this, as can be seen to the right preview. Although unlikely, questions and answers may come up more than once. In future development, this could prevented by using an if statement to check firstly the generated question is not the same as any previously in the questions list, and secondly, that no two answers are the same.

I now created a more complicated quiz format (using data from my design of the program). I first modified my database:

Table: Quiz

	quizID	subject	level	noOfQuestions
	Filter	Filter	Filter	Filter
1	19	Test	Test	2
2	20	Maths	KS1	3
3	21	Maths	KS2	0
4	22	Maths	A-Level	6

Table: Question — Filter in any column

	iestion	uizII	questionType	question	oOfAnswer	answerOne	answerTwo	answerThree	answerFour
	Filter	Fil…	Filter	Filter	Filter	Filter	Filter	Filter	Filter
1	16	19	Multiple Choice	Question 1	2	test	test2	NULL	NULL
2	17	19	Answer Input	Question 2	1	test answer	NULL	NULL	NULL
3	18	20	Multiple Choice	2 x 2	2	4	6	NULL	NULL
4	19	20	Multiple Choice	3 x 4	3	12	10	14	NULL
5	20	20	Answer Input	4 x 5	1	20	NULL	NULL	NULL
6	21	22	Multiple Choice	The radius of a sphere is k_1. Calculate the volume of the sphere. (Nearest number)	4	4/3 * π * r^3	1 , 15	NULL	NULL
7	22	22	Multiple Choice	The radius of a circle is k_1. A sector is created from this circle of k_2 radians. Calculate the arc length of the sector. (Nearest number)	4	r * θ	1 , 15	1 , 6	NULL
8	23	22	Answer Input	Determine how many roots, if any the quadratic equation k_1 x^2 + k_2 x + k_3 has. (Nearest number)	1	b^2 - 4 * a * c	1 , 5	1 , 10	1 , 15
9	24	22	Multiple Choice	Two sides of a right-angled triangle are k_1and k_2cm respectively. Calculate the length of the hypotenuse. (Nearest number)	4	a^2 + b^2 = c^2	1 , 15	1 , 15	NULL
10	25	22	Multiple Choice	Two sides of a triangle are k_1and k_2cm respectively. The angle between these two sides is k_3°. Calculate the area of the triangle. (Neare…	4	1/2 * a * b * sinC	1 , 15	1 , 15	1 , 80
11	26	22	Multiple Choice	The perpendicular height of a cone is k_1. The radius of the cone is k_2. Calculate the curved surface area of the cone. (Nearest number)	4	π*^2 * slant height	1 , 15	1 , 15	NULL

Here, I filled in the question type and the questions created earlier from my design. However, instead of leaving the answers blank, I instead used the answerOne field to hold the equation that would be used by the program to calculate the answer. I used the remaining answer fields to, if necessary, store the range of randomisation. Since the most ‘k’ values, or values that had to be randomised was 3, it was sufficient in holding it. I then again modified the playSelectedQuiz function. I modified the else part of the if statement shown above. After retrieving the question details from the database, the program checks to see if the quiz ID of the selected quiz is 22. If not it continues as before. If it is, it creates a for loop for every question in the list, changing all tuples to lists. This is so that they can be edited. I then use another for loop with 3 iterations (the maximum number of k values) and determine whether first the answerTwo field is empty. With each iteration, 6 is added to the value of each2 so that consequently answerThree is checked, then answerFour.

```
## If quiz with set questions
else:
    ## Retrieve all questions from database for selected quiz
    questions = viewingFromDatabase.viewFromDatabase(c, True, False, None, 'quizID', str(quizDetails[0][0]), 'Question')
    ## If quiz ID of selected quiz is 22
    if quizDetails[0][0] == 22:
        ## For each question
        for each in questions:
            ## Convert tuple to list so can be edited
            each = list(each)
            ## Replace all 'k's with random integers within given range from database
            counter = 1
            for each2 in range(0,3):
                if each[each2 + 6] != None:
                    randomRange = (each[each2 + 6]).split()
                    each[3] = each[3].replace('k_'+str(counter), str(random.randint(int(randomRange[0]), int(randomRange[2]))))
                    counter += 1
            print(each[3])
```

The function retrieves the value from these answer fields and splits it, since it is all one string. This divides all values which are separated by a space and this is assigned as a list to a variable randomRange. So if the answer field is not empty, this range is picked out using the first and third item in the list, and the k values are replaced accordingly. This produced:

```
The radius of a sphere is 4. Calculate the volume of the sphere. (Nearest number)
The radius of a circle is 13. A sector is created from this circle of 5 radians.
Calculate the arc length of the sector. (Nearest number)
Determine how many roots, if any the quadratic equation 3 x^2 + 6 x + 1 has. (Nea
rest number)
Two sides of a right-angled triangle are 10and 7cm respectively. Calculate the le
ngth of the hypotenuse. (Nearest number)
Two sides of a triangle are 6and 1cm respectively. The angle between these two si
des is 68°. Calculate the area of the triangle. (Nearest number)
The perpendicular height of a cone is 4. The radius of the cone is 8. Calculate t
he curved surface area of the cone. (Nearest number)
```

As desired.

Next, I had to use the randomised values to calculate the correct answer using the equation provided in the answerOne field in the database. However, with the solution proposed above, the randomised values were not stored, and so could not be used later on. I thus stored them, in the form of a list. To do this:

```
## If quiz ID of selected quiz is 22
if quizDetails[0][0] == 22:
    ## Create list for all 'k' values
    kValuesList = []
    ## Create counter for each question
    question = 0
    ## For each question
    for each in questions:
        ## Append empty list to kValuesList
        kValuesList.append([])
```

I first created a list to store all the random 'k' values and appended an empty list for every question in this list. I created a counter to track which list within the kValuesList was being added to. In this way, there would be lists within this kValuesList for every question and their corresponding k values in order.

```
if each[each2 + 6] != None:
    randomRange = (each[each2 + 6]).split()
    randomValue = str(random.randint(int(randomRange[0]), int(randomRange[2])))
    each[3] = each[3].replace('k_'+str(counter), randomValue)
    kValuesList[question].append(randomValue)
    counter += 1
```

I did this by first assigning the randomised value to a variable which was then appended to the relevant list.

```
print(kValuesList)
question += 1
```

I then printed the list and incremented the question counter by one within the 'for each in questions' loop.

The final iteration produced this:

```
[['5'], ['13', '6'], ['3', '9', '14'], ['5', '1'], ['9', '4', '78'], ['2', '14']]
```

I modified the equations in the database so the program could more easily identify which values would have to be used in the correct places:

4/3*π*a*a*a
a*b
b*b-4*a*c
√(a*a+b*b)
1/2*a*b*sin(c)
π*b*b*√(a*a+b*b)

I refined the equations and wrote any squared or cubed numbers in their full basic form.

Next, I replaced all the a, b and c values within the equation with their corresponding value from the kValuesList:

```
## For number of random k values
for each3 in range(len(kValuesList[question])):
    ## If one value
    if len(kValuesList[question]) > 0:
        ## Replace a with random value
        each[5] = each[5].replace('a', kValuesList[question][0])
    ## If two values
    if len(kValuesList[question]) > 1:
        ## Replace b with random value
        each[5] = each[5].replace('b', kValuesList[question][1])
    ## If three values
    if len(kValuesList[question]) > 2:
        ## Replace c with random value
        each[5] = each[5].replace('c', kValuesList[question][2])
```

Then, I split the equation string and for each value in the new split list, I changed the data type for all numbers to integers and appended it to a new list.

```
## Split equation string and change integer data type for each number
each[5] = each[5].split()
equationList = []
for each4 in each[5]:
    if each4.isdigit() == True:
        each4 = int(each4)
    equationList.append(each4)
```

After this, I created another list. In this list, I placed the contents of any values within two brackets. This was because I had to calculate these parts of the equations first.

```
## Place all contents of any brackets in new list
bracketList = []
for each5 in range(len(equationList)):
    if equationList[each5] == '(':
        counter = 1
        while equationList[each5+counter] != ')':
            bracketList.append(equationList[each5+counter])
            counter+=1
```

To continue, I had to find a way to change my string values which contained operators to operators which the program would recognise and use. To do this, I imported the operator library:

```
import operator
```

I then created the following dictionary:

```
## Create dictionary for string operator values
operators = {
            '+' : operator.add,
            '-' : operator.sub,
            '*' : operator.mul,
            '/' : operator.truediv,
            'π' : math.pi,
            '√' : math.sqrt,
            'sin\u2061' : math.sin}
```

To make use of pi, square root and sin, I had to import the math library:

```
import math
```

I went on to solve the equation, if any, within the brackets list:

```
## If bracketList not empty, calculate contents
if len(bracketList) > 0:
    counter = 0
    while len(bracketList) != 1:
        ## If value is an operator
        if bracketList[counter] in operators:
            ## Use operator on following value and preceding value and replace following value
            bracketList[counter+1] = (operators[bracketList[counter]](bracketList[counter-1],bracketList[counter+1]))
            ## Remove preceding values
            bracketList.pop(counter-1)
            bracketList.pop(counter-1)
            ## Reset counter since index changed
            counter = 0
        counter += 1
```

If the length of the brackets list was above one, I created a while loop until the length of the list was one, by using the operator, whenever one was found in the list, on the following and preceding values to replace the following value to the operator. I then removed the preceding value and the operator from the list. Since this changed the index every time, I avoided a for loop and reset the counter for the while loop if an operator had been used. If not, the counter incremented by one. Although this worked, it disobeyed a fundamental law of mathematics and calculated parts of the equation in order from left to right. Since my equations involved multiplication, division, addition, etc. it was important that they occur in the correct order. Thus I modified the function. First, I took out the main calculation process as its own function:

```
def calculateEquation(self, operators, equationlist, counter):
    ## Use operator on following value and preceding value and replace following value
    equationlist[counter+1] = (operators[equationlist[counter]](equationlist[counter-1],equationlist[counter+1]))
    ## Remove preceding values
    equationlist.pop(counter-1)
    equationlist.pop(counter-1)
```

I then changed the while loop so all multiplications and divisions would occur first:

```
## If bracketList not empty, calculate contents
if len(bracketList) > 0:
    counter = 0
    ## Work out any multiplication or division first
    while '*' in bracketList or '/' in bracketList:
        if bracketList[counter] == '*' or bracketList[counter] == '/':
            ## Calculate operator
            self.calculateEquation(operators, bracketList, counter)
            ## Reset counter since index changed
            counter = 0
        counter += 1
```

I modified it so whilst any * or / values were detected in the list, their index were sought out and calculated.

```
    counter = 0
    ## Work out any other operator
    while len(bracketList) != 1:
        if bracketList[counter] in operators:
            ## Calculate operator
            self.calculateEquation(operators, bracketList, counter)
            ## Reset counter since index changed
            counter = 0
        counter += 1
```

I then created another while loop that dealt with the rest of the operators within the list.

Next, I had to extend the function to solve the contents of the equation list. However, this list still had the contents of the brackets within it so I first had to remove this. To do this:

```
print(equationList)
## Place all contents of any brackets in new list
bracketList = []
bracketIndexList = []
for each5 in range(len(equationList)):
    if equationList[each5] == '(':
        counter = 1
        while equationList[each5+counter] != ')':
            bracketList.append(equationList[each5+counter])
            bracketIndexList.append(each5+counter)
            counter+=1

## Reverse list to avoid index issues
bracketIndexList.reverse()
## If not empty
if len(bracketIndexList) > 0:
    ## For every index including and within brackets
    for each6 in bracketIndexList:
        ## Remove value in list in reverse index order
        equationList.pop(each6)
print(equationList)
```

I created a list to store all the index values of the values within the equation list that fell between two brackets. I did this using the loop created earlier. I avoided removing the values here since by doing so, the index of the list would not remain constant for each iteration in the for loop and would cause errors. Instead, with the indices collected in the bracketIndexList, I popped each index from the equation list in reverse order to avoid the index problem.

The last iteration of the question loop produced this:

```
The perpendicular height of a cone is 15. The radius of the cone is 4. Calculate
 the curved surface area of the cone. (Nearest number)
['π', '*', 4, '*', 4, '*', '√', '(', 15, '*', 15, '+', 4, '*', 4, ')']
['π', '*', 4, '*', 4, '*', '√', '(', ')']
[241]
```

As desired.

I then removed the brackets from the equationList and replaced them with the solution of the equation within the brackets calculated previously.

```
## Remove brackets from equationList
for each7 in range(len(equationList)-1):
    if equationList[each7] == '(':
        ## Remove ')' value from list
        equationList.pop(each7+1)
        ## Replace '(' value with result in bracket list
        equationList[each7] = bracketList[0]
```

Here, I was required to use a for loop with a number of iterations which was one less than the length of the equations list since two values (both brackets) were to be removed and only one of them replaced. The result for the relevant questions is as follows:

```
Two sides of a right-angled triangle are 1and 10cm respectively. Calculate the l
ength of the hypotenuse. (Nearest number)
['√', '(', 1, '*', 1, '+', 10, '*', 10, ')']
['√', '(', ')']
[101]
['√', 101]
```

```
Two sides of a triangle are 4and 6cm respectively. The angle between these two s
ides is 37°. Calculate the area of the triangle. (Nearest number)
[1, '/', 2, '*', 4, '*', 6, '*', 'sin\u2061', '(', 37, ')']
[1, '/', 2, '*', 4, '*', 6, '*', 'sin\u2061', '(', ')']
[37]
[1, '/', 2, '*', 4, '*', 6, '*', 'sin\u2061', 37]
The perpendicular height of a cone is 9. The radius of the cone is 10. Calculate
 the curved surface area of the cone. (Nearest number)
['π', '*', 10, '*', 10, '*', '√', '(', 9, '*', 9, '+', 10, '*', 10, ')']
['π', '*', 10, '*', 10, '*', '√', '(', ')']
[181]
['π', '*', 10, '*', 10, '*', '√', 181]
```

I completed the calculation of the equations. To do this, I first replaced the pi, root and sin values to their operators and worked out and replaced in the list any root and sin values so to not get confused when working out the operators later on.

```
for each8 in range(len(equationList)-1):
    ## Change sin string value to operator
    if equationList[each8] == 'sin\u2061':
        ## Calculate sin values and replace in list
        equationList[each8] = (operators['sin\u2061'])(equationList[each8+1]*math.pi/180)
        equationList.pop(each8+1)

    ## Change pi string value to operator
    elif equationList[each8] == 'π':
        equationList[each8] = operators['π']

    ## Change root string value to operator
    elif equationList[each8] == '√':
        ## Calculate root values and replace in list
        equationList[each8] = (operators['√'])(equationList[each8+1])
        equationList.pop(each8+1)
```

When working out the values for the sin functions, I was required to perform a calculation to convert the values in the question, represented in degrees, to a value in radians, which was used by python. I was now able to perform the final calculations for the equation:

```
## Calculate and change answers using equation in answerOne field:
counter = 0
## Work out any multiplication or division first within equationList
while '*' in equationList or '/' in equationList:
    if equationList[counter] == '*' or equationList[counter] == '/':
        ## Calculate operator
        self.calculateEquation(operators, equationList, counter)
        ## Reset counter since index changed
        counter = 0
    counter += 1
counter = 0
print(equationList)
## Work out any other operator within equationList
while len(equationList) != 1:
    if equationList[counter] in operators:
        ## Calculate operator
        self.calculateEquation(operators, equationList, counter)
        ## Reset counter since index changed
        counter = 0
    counter += 1
```

This followed the same format as the calculation for the bracket list.

This correctly answered the equations. Since the questions asked for the nearest number however, I rounded them using the python round function:

```
print(equationList)
print(round(equationList[0]))
```

```
The radius of a sphere is 9. Calculate the volume of the sphere. (Nearest number)
[4, '/', 3, '*', 'π', '*', 9, '*', 9, '*', 9]
[4, '/', 3, '*', 'π', '*', 9, '*', 9, '*', 9]
[3053.628059289279]
3054
The radius of a circle is 13. A sector is created from this circle of 2 radians.
Calculate the arc length of the sector. (Nearest number)
[13, '*', 2]
[13, '*', 2]
[26]
26
Determine how many roots, if any the quadratic equation 2 x^2 + 2 x + 4 has. (Nea
rest number)
[2, '*', 2, '-', 4, '*', 2, '*', 4]
[2, '*', 2, '-', 4, '*', 2, '*', 4]
[-28]
-28
Two sides of a right-angled triangle are 2and 8cm respectively. Calculate the len
gth of the hypotenuse. (Nearest number)
['√', '(', 2, '*', 2, '+', 8, '*', 8, ')']
['√', 68]
[8.246211251235321]
8
Two sides of a triangle are 15and 11cm respectively. The angle between these two
sides is 32°. Calculate the area of the triangle. (Nearest number)
[1, '/', 2, '*', 15, '*', 11, '*', 'sin\u2061', '(', 32, ')']
[1, '/', 2, '*', 15, '*', 11, '*', 'sin\u2061', 32]
[43.7183392992394]
44
The perpendicular height of a cone is 5. The radius of the cone is 6. Calculate t
he curved surface area of the cone. (Nearest number)
['π', '*', 6, '*', 6, '*', '√', '(', 5, '*', 5, '+', 6, '*', 6, ')']
['π', '*', 6, '*', 6, '*', '√', 61]
[883.3184281630946]
883
```

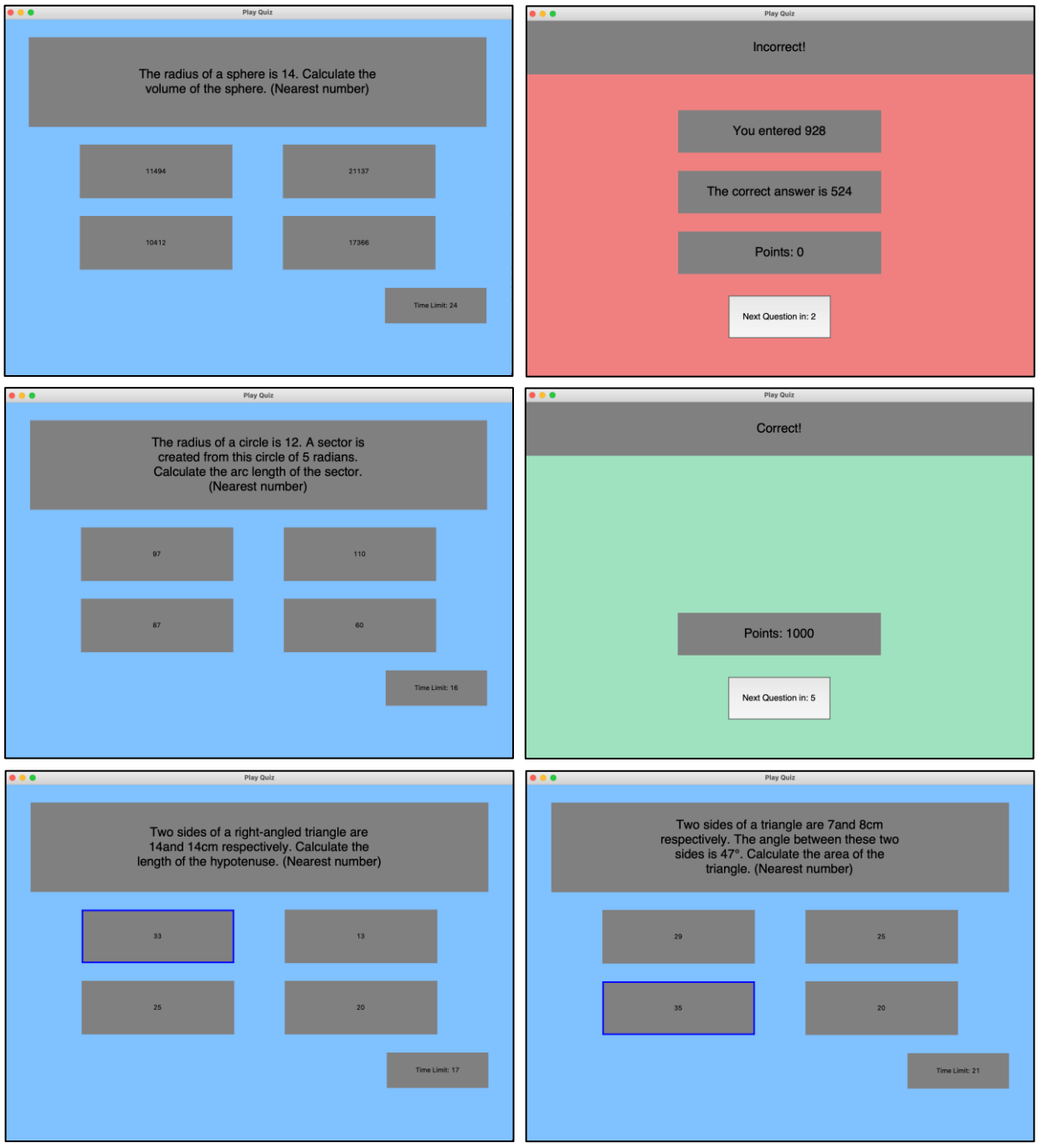
Finally, I assigned the answers to the questions list:

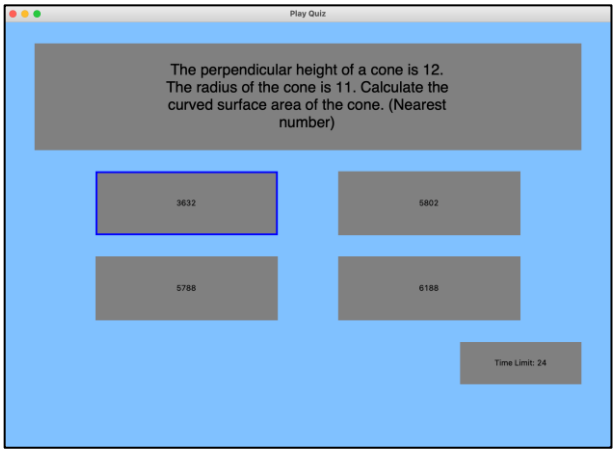
```
## If question 3
if question == 2:
    ## Determine number of roots from discriminant and assign answer
    if equationList[0] < 0:
        each[5] = 0
    elif equationList[0] == 0:
        each[5] = 1
    elif equationList[0] > 0:
        each[5] = 2
else:
    ## Assign answers for the question
    each[5] = (round(equationList[0]))
    each[6] = random.randint(each[5]//2, each[5]*2)
    each[7] = random.randint(each[5]//2, each[5]*2)
    each[8] = random.randint(each[5]//2, each[5]*2)
```

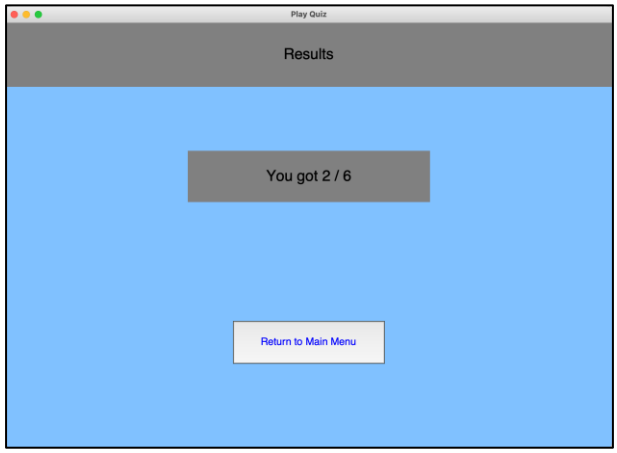
Since, the answer retrieved for question number 3 wasn't the final answer, a little more calculation was required to find the correct answer. So, I checked to see if the question was question 3 and if so did these calculations and assigned one answer since an answer input question. For the rest of the questions, I assigned the correct answer in the reflection of the answerOne field in the questions list and created randomised values for the other three answer options. I created answers from a range of the correct answer divided by 2 (integer division) and the correct answer times 2 for a more realistic range.

When first testing this out with the rest of the play function, none of the changes had seemed to appear. This was because, in the for loop, only a copy of the values of the questions list had been used. To fix this I replaced the for loop with for each in range(len(questions)) and all references to ‘each’ with questions[each] to cause direct changes to the questions list. I also had to make all answers string values to prevent a concatenation of integer and string error when displaying what the user chose as an answer.

This produced:



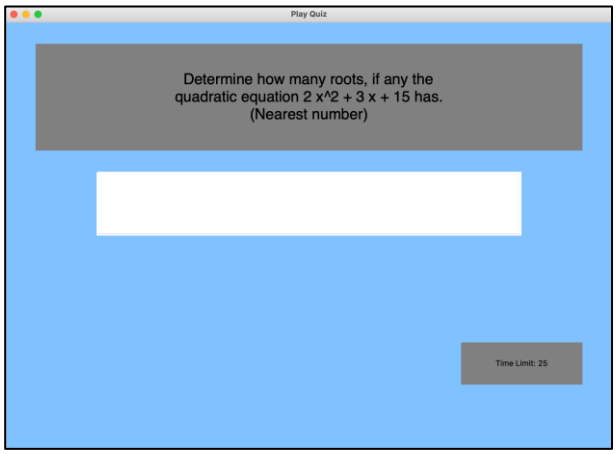




Initially, there was an issue where question 3 had shown options instead of an entry box (since it is an answer input question). This was because in the displayQuestion function, instead of looking at the type of question, it looked at the length of the answers list, and since there was more than one answer (since the ranges), it was recognised as a multiple choice question. To fix this, I just had to empty these values from the list:

```
## If question 3
if question == 2:
    ## Determine number of roots from discriminant and assign answer
    if equationList[0] < 0:
        questions[each][5] = '0'
    elif equationList[0] == 0:
        questions[each][5] = '1'
    elif equationList[0] > 0:
        questions[each][5] = '2'
    ## Empty all other answer options
    questions[each][6] = None
    questions[each][7] = None
    questions[each][8] = None
```

This produced:



As desired.

Following this process, more quizzes could be created in this format in future development, for various subjects. Also, time limits should be set more reasonably with the question at hand.
There are, however, a number of limitations to this method. Firstly, the format the question is shown in is not in the most user-friendly format, with square number and cube number not shown as it should (as a subscript), which may confuse or be an inconvenience for the user. In this method,

the program only checks for one set of brackets. Although this doesn't affect the quiz compiled above, it may affect those created at a later point. The relevant for loop could be extended, however, to create a list within the bracketList for each instance of '(', and the function could then be modified to solve each list within the bracketList and replace the '(', ')' values with the correct value from the list after calculation. Finally, the function could be modified to increase modularity within the program, since there is a lot of repeated code. Important parts of the function can be brought out as their own function and reused in this way.

14. *automatically detect areas or certain questions within quizzes that students have difficulty with and perform poorly in* – one crucial aspect that my client wanted included in the program was automatic difficulty identification so she could tailor her lessons. This would require extensive work from the program. This could be done, perhaps within the view classes menu option, giving an admin user the access to a separate window. When initiated, this function would retrieve the users, the quizzes, the questions and the progress. This could be done by performing an inner join on the essential parts of the table, such as the quizzes, questions and progress for each question. With the updates described above, where students are allocated into separate classes, an admin user must be able to choose between different classes and perhaps be blocked from accessing any other classes, as a courtesy to the privacy of the students. With this new table, the program can run through, in an iterative manner, the different records and create a list with the number of incorrect answers for each question for the selected class. The selection of classes can be done using a list box as done in other functions. If the number of incorrect answers exceeds a certain amount, perhaps 50% of the total number of answers, then these questions can be flagged (added to a new list along with the corresponding questions). After this process, the quizzes, questions and success rates can be displayed, perhaps again in the form of list boxes in which the user can select which quiz. This may look something like this:

```
// With classes database updates detailed earlier
classes <-- viewingFromDatabase.viewFromDatabase(c, False, True, 'Class', None, None, 'Classes')
// Create list box for all classes
classesListbox <-- tk.Listbox
// Insert each class into list box
FOR i in classes
        classesListbox.insert(i)
ENDFOR
// Bind to displayQuizDetails function to show all quizzes for that class in another list box
// The displayQuizDetails function would have to be modified to suit this function
classesListbox.bind('<<ListboxSelect>>', lambda x: self.displayQuizDetails(username,…))
```

With the list box created within the displayQuizDetails function, I can create a bind to a new event which identifies the difficulties:

```
function identifyDifficulties(self, class)
        // Fetch the relevant data from the database by performing an inner join between the Quiz,
        Question and Progress tables
        newTable = c.execute(""" SELECT Quiz.name, Question.question, Progress.correct
                FROM ((Question
                INNER JOIN Quiz ON Quiz.quizID = Question. quizID)
                INNER JOIN Progress ON Progress.quizID = Quiz.quizID) """)
        newTableData = newTable.fetchall()
        incorrectAnswersList = []
        flaggedQuestionsList = []
```

```
// For each record
FOR each in newTableData
        FOR each2 in range(len(incorrectAnswersList))
                IF incorrectAnswersList [each2][0] = each[1] THEN
                        // Add one to total attempts
                        incorrectAnswersList [each2][1] += 1
                        // If incorrect
                        IF each[2] = False THEN
                                // Add one to total incorrect attempts
                                incorrectAnswersList [each2][2] += 1
                        ENDIF
                ELSE
                        // If incorrect
                        IF each[2] = False THEN
                                incorrectAttempts = 1
                        // If correct
                        ELSE
                                incorrectAttempts = 0
                        ENDIF
                        // Add to list
                        incorrectAnswersList.append([each[1], 1, incorrectAttempts])
                ENDIF
        ENDFOR
ENDFOR
```

This section of the function would accumulate all the question, attempts and incorrect attempts and add it to the incorrectAnswersList.

```
// For each question
FOR each3 in incorrectAnswersList
        // If success rate lower than 50%
        IF each3[1] / each3[2] > 0.5 THEN
                // Add question list to flaggedQuestionsList
                flaggedQuestionsList.append(each3)
```

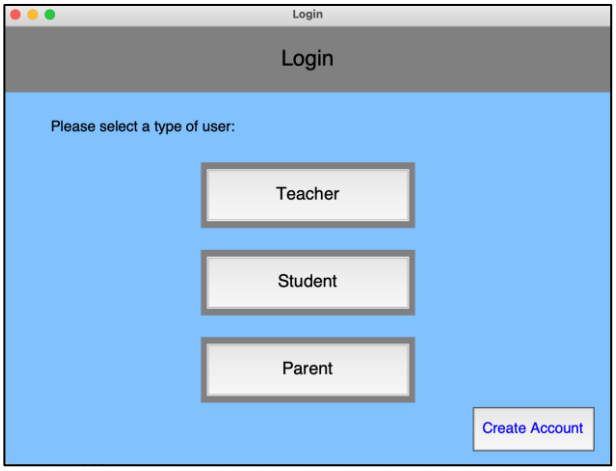
This section of the function would calculate for each question in the list whether more than 50% of attempts are successful. If not it is flagged. This value of 50% can easily be changed but with feedback from my client, she agrees it is a reasonable success rate for her students. The contents of the flaggedQuestionsList can then be displayed in various formats, whether that be in writing, in a table, or through charts, to show the admin user the questions that have caused difficulty, and the overall success rate.

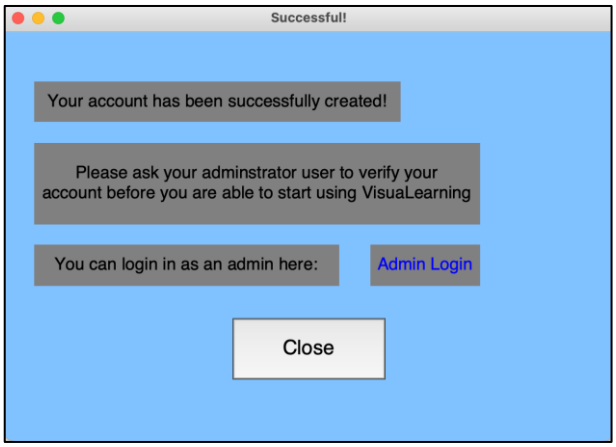
With all these suggestions improved, my client said:
'All these improvements really make this program perfect for me, really easy to use with everything you need right in front of you. Also great for the students and helps me massively with all the time taken previously for marking, etc. Now I can not only just save time marking and taking tests, but also quickly find out weaknesses among my students in all my classes, making it an amazing tool for me to target my learning and revision!'

Now, I moved on to alpha testing, by allowing a group of users to test out the program to pinpoint any bugs or errors that need to be fixed. I assigned users for different parts of my success criteria.

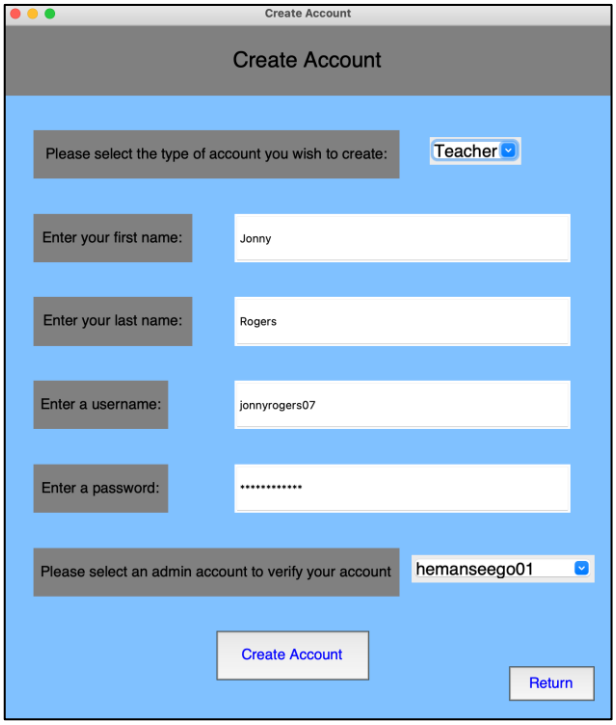
- User access levels with secure authentication –
  - Logging in has three choices of either admin login (for the teacher), student login (for the students) or parent login, all of which require a username and password and give different accessibilities to the user
  - Username and passwords shall be checked against a database to be authenticated
  - Users are locked if an incorrect password is entered three times; this is reversible through admin accounts

  - Signing up; users should be able to sign up by entering a username not already in the database and a password which is secure, i.e. meets a character limit/has special characters, etc.
  - New usernames and passwords should be added to the existing database.

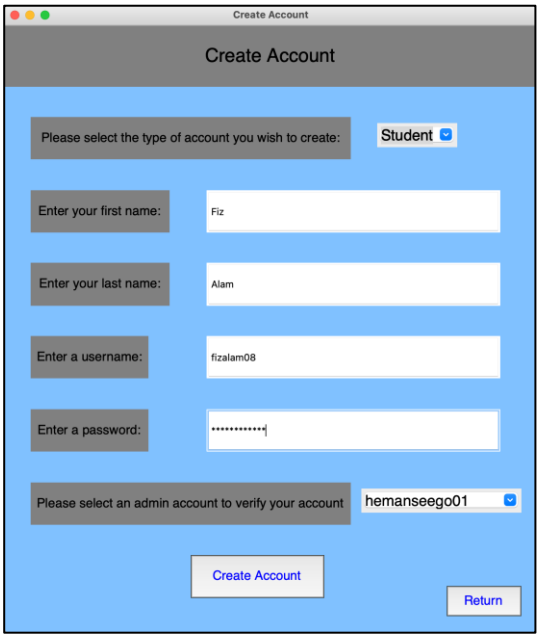


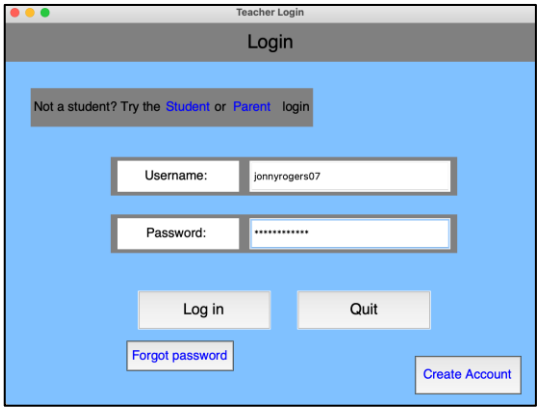


The users created new accounts using the create account option, one for each account type:



The user suggested here that the create account page could require the user to confirm their password to prevent saving any mistyped passwords. Although the user would be able to reset their password by selecting the forgot password option, this process would be really long and inconvenient, so in future development, I agree that a new frame, label and entry can be created to ask the user to retype their password. The program would then have to make sure the two inputted passwords match before continuing to store their account to the database. This one time process could ultimately prove more convenient for the user and prevent them getting blocked or losing access to their account.





When the first user logged in immediately after creating their account, they gained access to the main menu page. This shouldn't be the case. A user should first be verified by the chosen admin user before they can access the program. I therefore modified the loginClicked function. So far in this function, I had only created a check for whether the account was blocked, but not if it was unverified. Hence, I introduced an accountUnverified variable:

```
accountUnverified = False
```

```
if accountReview == 4:
    accountBlocked = True
elif accountReview == 2:
    accountUnverified = True
```

Here, I added an elif part to the if statement to check for if the account review ID, retrieved from the database, was 2, indicating a new account which had to be verified. If it was, I changed the value of the accountUnverified variable to True.

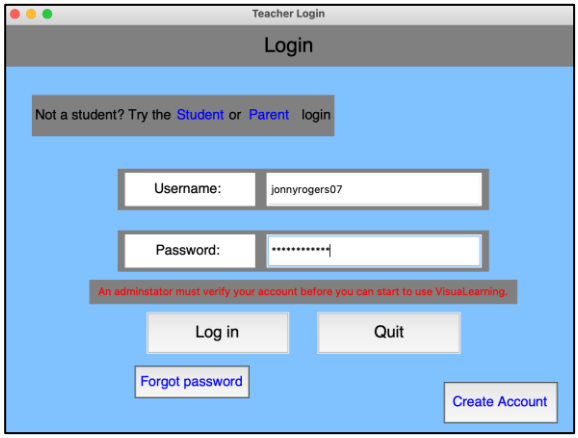
I also extended the next if statement:

```
if correctUsername == True and correctPassword == True and accountBlocked == False and accountUnverified == False:
```

To include that the accountUnverified must be False for the login process to continue. I also created an error message for when an account has not been verified:

```
elif accountUnverified == True:
    error = 'An adminstator must verify your account before you can start to use VisuaLearning.'
```

When the user tried logging in again, this produced:



With the updates described previously in future development, the admin user chosen (here hemanseego01) would be able to log in, go on settings, click the review users option, click on this user and see the details of the review and either verify or delete the account.

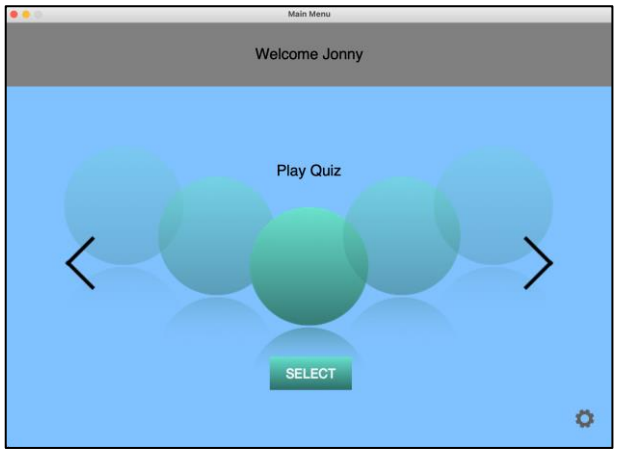
After verification, an error was experienced when logging in:

```
_tkinter.TclError: image "pyimage3" doesn't exist
```

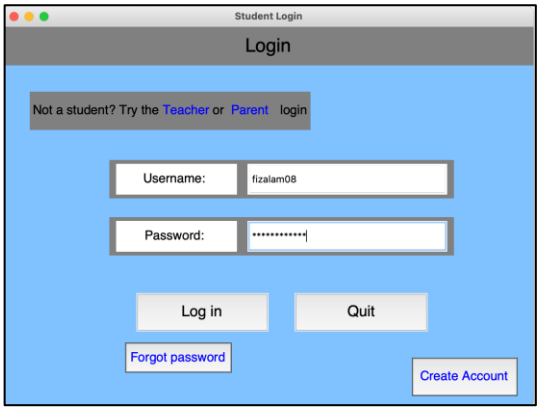
Where the login page and the main menu page were conflicting.

```
if correctUsername == True and correctPassword == True and accountBlocked == False and accountUnverified == False:
    if loginType == 'Admin':
        ## Destroy log in page
        loginPage.destroy()
        ## Open admin main menu page
        mainMenu.adminMainMenuPage(usernameEntry, Login.titleFont, Login.buttonFont, Login.defaultFont)
```

This was easily fixed by destroying the login page before opening the main menu page.
The user was now able to access the main menu.



When the other users tried to log in:

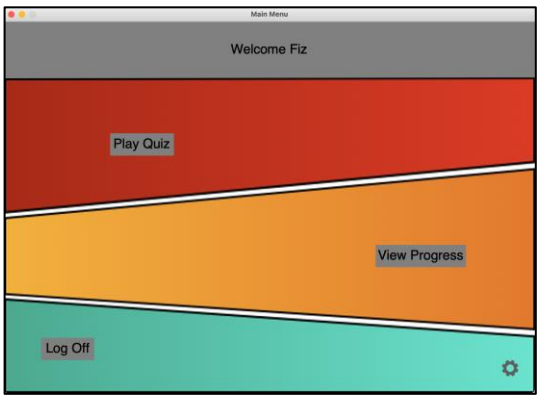


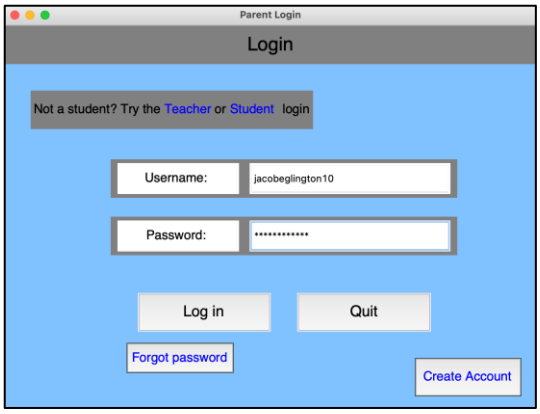
Another error was found.

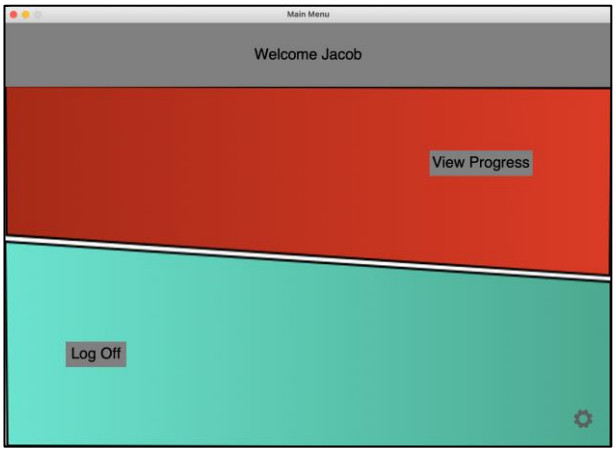
```
if loginType == 'Student':
    ## Destroy log in page
    loginPage.destroy()
    ## Open student main menu page
    mainMenu.studentMainMenuPage(usernameEntry, Login.titleFont, Login.buttonFont, Login.defaultFont)
if loginType == 'Parent':
    ## Destroy log in page
    loginPage.destroy()
    ## Open parent main menu page
    mainMenu.parentMainMenuPage(usernameEntry, Login.titleFont, Login.buttonFont, Login.defaultFont)
```

Firstly, when first creating this function, the student and parent main menu pages had not been finalised, and therefore had no parameters. In the modification shown above, I firstly filled in all the relevant parameters, and also destroyed the login page as before to prevent the pyimage error.

This then produced:



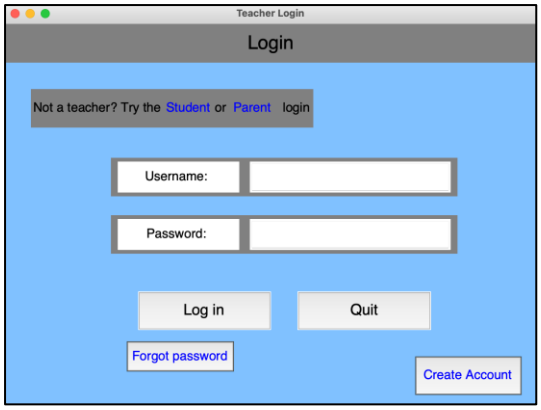


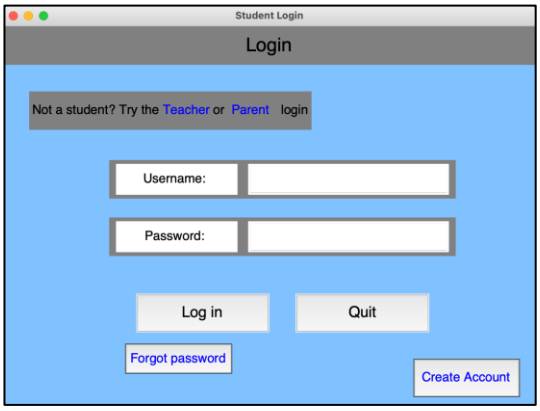


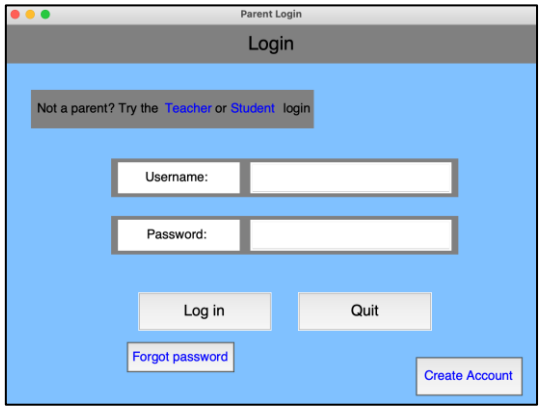
Although the login worked as desired, the users noticed a mistake within the interfaces. Whilst the links were correct, the overall change login label contained a fault. This was only in the teacher and parent login pages. Instead of reading 'Not a teacher' or 'Not a parent' respectively, all pages read 'Not a student'.

```
## Retrieves the assigned width, height and type f
if loginType == 'Admin':
    screenWidth = adminLogin.getScreenWidth()
    screenHeight = adminLogin.getScreenHeight()
    ## Assign the title of the page
    title = 'Teacher Login'
    ## Creates variables for parts of different lo
    differentLogin1 = 'Not a teacher? Try the'
    differentLogin2 = 'Student'
    differentLogin4 = 'Parent'
elif loginType == 'Student':
    screenWidth = studentLogin.getScreenWidth()
    screenHeight = studentLogin.getScreenHeight()
    title = 'Student Login'
    differentLogin1 = 'Not a student? Try the'
    differentLogin2 = 'Teacher'
    differentLogin4 = 'Parent'
elif loginType == 'Parent':
    screenWidth = parentLogin.getScreenWidth()
    screenHeight = parentLogin.getScreenHeight()
    title = 'Parent Login'
    differentLogin1 = 'Not a parent? Try the'
    differentLogin2 = 'Teacher'
    differentLogin4 = 'Student'
```

To fix this, I modified the loginPageDetails function and assigned the correct text for the first part of the differentLogin label.







When the users tried logging in a second time, this time instead of pressing the log in button, by pressing enter on the keyboard, another error was found. This was because I had modified the loginClicked function to include another parameter: the login page, and this change wasn't adjusted with the event bound to the password entry. I therefore adjusted it. I did the same for the username entry.

```
passwordEntry.bind("<Return>", lambda x: self.loginClicked(loginPage, usernameEntry.get(), passwordEntry.get(), loginType, errorLabel))
usernameEntry.bind("<Return>", lambda x: self.loginClicked(loginPage, usernameEntry.get(), passwordEntry.get(), loginType, errorLabel))
```

Post-Development Testing:
Create Account Function:
Test and result screenshots are shown pages 101-116 within the Development of the Solution.

Test Number	Test	Input Data	Expected Outcome	Result
SELECTING TYPE OF ACCOUNT				
1	User can only select one from Teacher, Student or Parent	Select 'Teacher', 'Student' or 'Parent'	Drop-down box displays selected option and passes it on to next function	Successful
FIRST NAME / LAST NAME ENTRY				
2	Field empty	''	Error message	Successful
3	Contains only white space	' '	Error message	Successful
4	Contains invalid character: number	'Heman1' 'Seegolam2'	Error message	Successful
5	Contains invalid character: symbol	'Heman!' 'See-golam'	Error message	Successful
6	Contains invalid space character	'He man' 'See golam'	Error message	Successful
USERNAME ENTRY				
7	Contains invalid space character	'heman 123'	Error message	Successful
8	Invalid username	'username'	Error message	Successful
9	Correct username	'hemanseego01'	Creates account if password is valid	Successful
10	Taken username	'hemanseego01'	Error message	Successful
11	SQL injection attempt	'INJECTION OR 1=1' 'INJECTION; DROP TABLE UserDetails'	Error message	Successful
PASSWORD ENTRY				
12	Field empty	''	Error message	Successful
13	Contains only white space	' '	Error message	Successful
14	Short password	'Pass1!'	Error message	Successful
15	No upper/lower case	'password1!'	Error message	Successful
16	No number	'Pass-word'	Error message	Successful
17	No symbol	'Password1'	Error message	Successful
18	Correct password	'Pass-Word123'	Creates account if username is valid	Successful
19	SQL injection attempt	'INJECTION OR 1=1' 'INJECTION; DROP TABLE Quiz'	Error message	Successful
SELECTING ADMIN USER				

20	User can only select one from all admin users	Select 'AdminUsername1', 'AdminUsername2',…	Drop-down box displays selected option and passes it on to next function	Successful

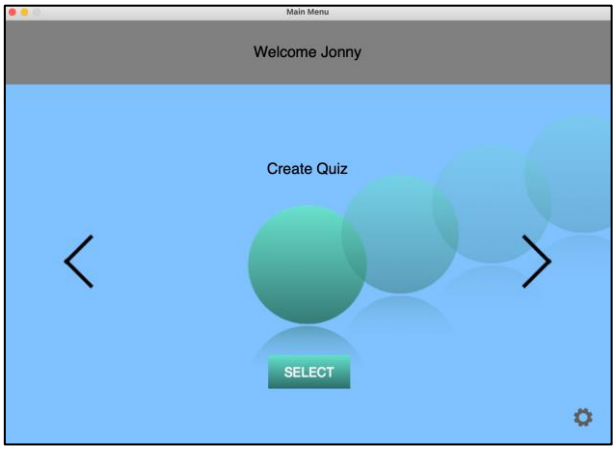
I added some extra tests here to test for the opportunity of SQL injection and whether the program successfully prevents it. This was done only for the username and password since symbols in these entries were permitted, unlike for the first name, last name and the combo-box options. Although an essential process for the username, since the password is hashed when creating an account and when checking whether matches the correct one, SQL injection prevention is unnecessary since a different, unique value is retrieved from the database. However, I shall keep it as part of my program to make sure users use it as they should and not even attempt to break it in this way.

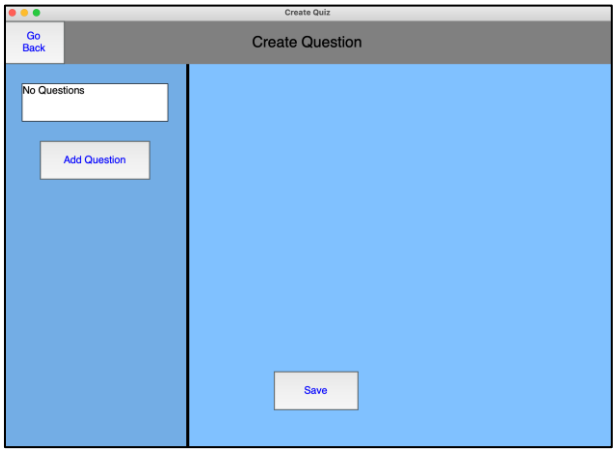
Logging in:
Test and result screenshots are shown pages 122-126 within the Development of the Solution and pages 234-235 of the Evaluation

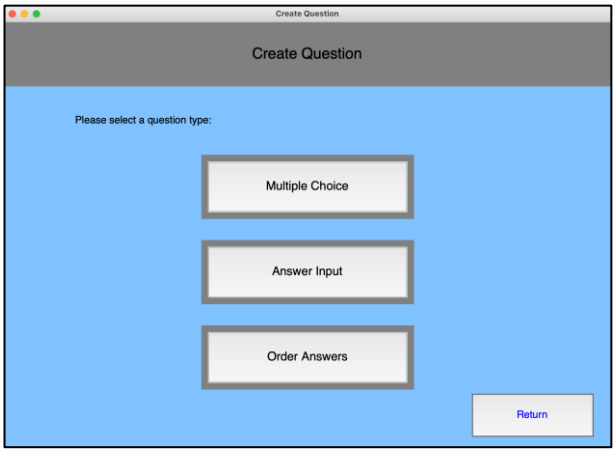
Test Number	Test	Input Data	Expected Outcome	Result
1	Whether username exists	'hemanseego01', 'hemanseego02'	Checks matching password, Error message	Successful
2	Incorrect password	'incorrectPassword'	Error message	Successful
3	Correct password	'Pass-Word123'	Access granted to main menu	Successful
4	Whether access denied when account is blocked	'hemanseego1' as username; 'Pass-Word123' as password when account is blocked	Access denied; Error message	Unsuccessful; However modified to work as desired as shown previously
5	Whether account has been blocked	Three consecutive incorrect log ins	Message stating account has been blocked	Partial Success; However modified to work as desired as shown previously
6	Whether account has been verified	Log in from newly unverified created user	Message stating account hasn't been verified	Unsuccessful; However modified to work as desired as shown previously

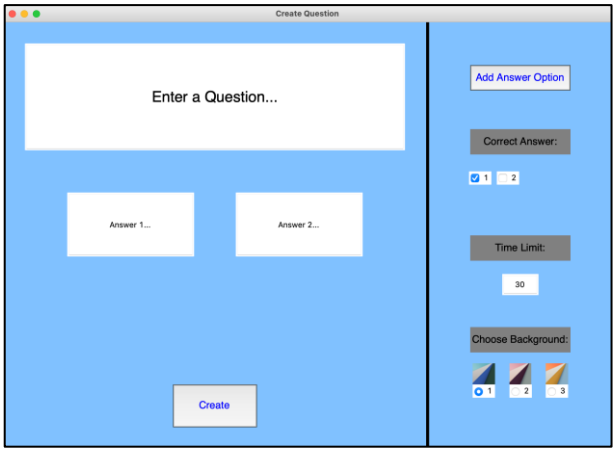
All parts of the 'User access levels with secure authentication' section of the success criteria have therefore been met.

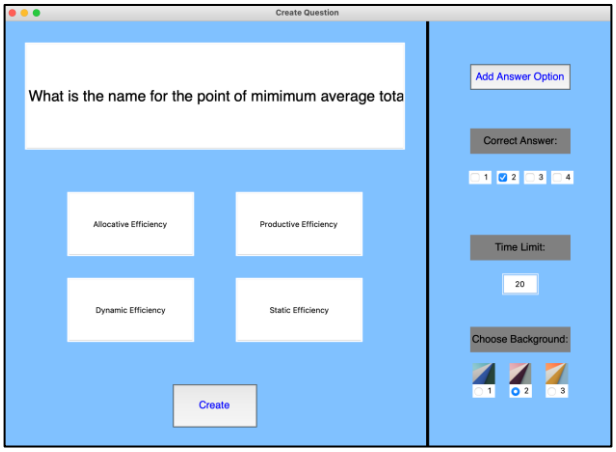
- Quiz Creation –
  - Administrator accounts should have the ability to create a quiz by adding, editing or removing different questions.
  - Teachers should be able to set questions and time limits.





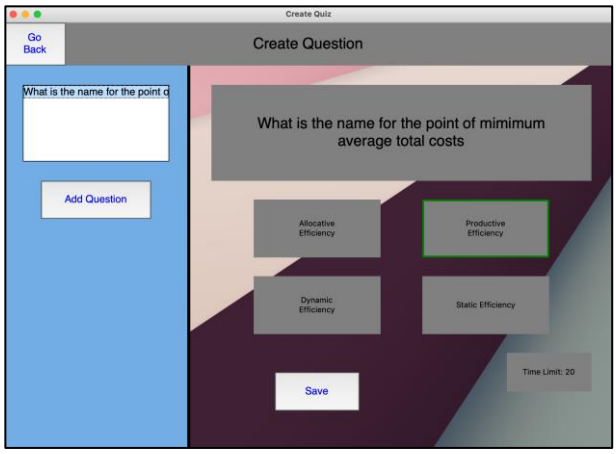






However, the question was displayed as full when the preview was displayed, and consequently when being played or viewed. Perhaps in future development it could be useful to add a horizontal scrollbar to the list box so the questions can be viewed in full before being previewed.

When the user selected the multiple choice option for when creating a question and created an example question, he noticed that the full question is hidden if it becomes too long. This also happens in the question list box when the question has been created:



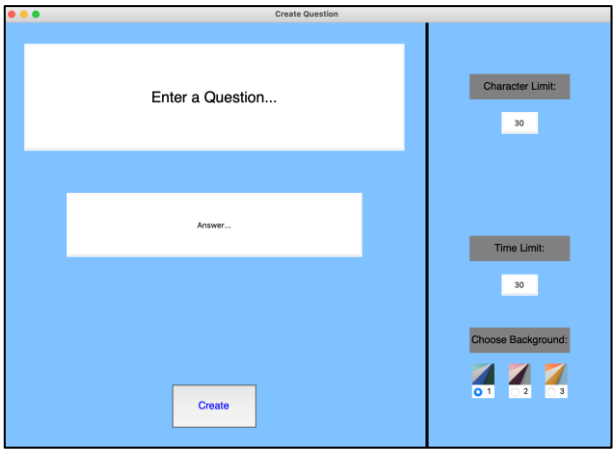
Initially, the user discovered an error with the preview of the questions. This was because the displayQuestion function had been modified to fit the play quiz functions and the create quiz functions hadn't been adjusted for the modifications. Namely, three variables were returned at the end of the function, but if the function was accessed from the create quiz functions, two of the variables didn't exist since they were declared in an if statement that were never initiated.

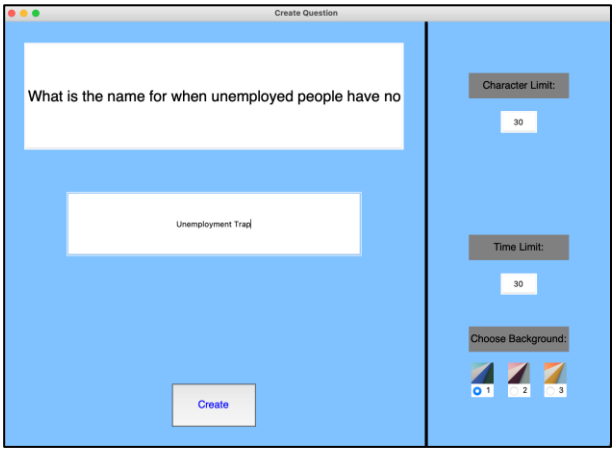
```
return answerInput, correctAnswer, correct
```

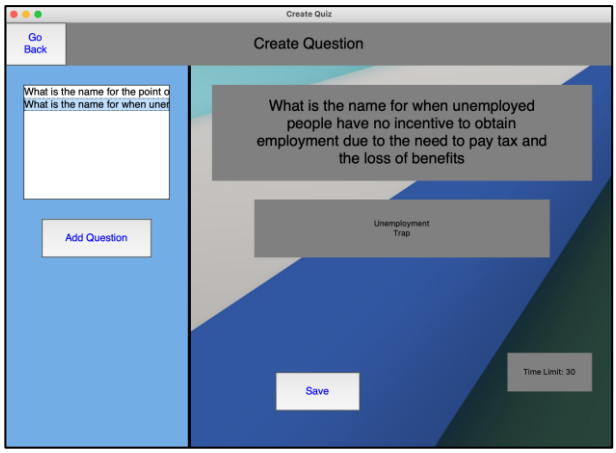
Therefore, I had to declare them before the if statements:

```
answerInput = ''
correct = False
```

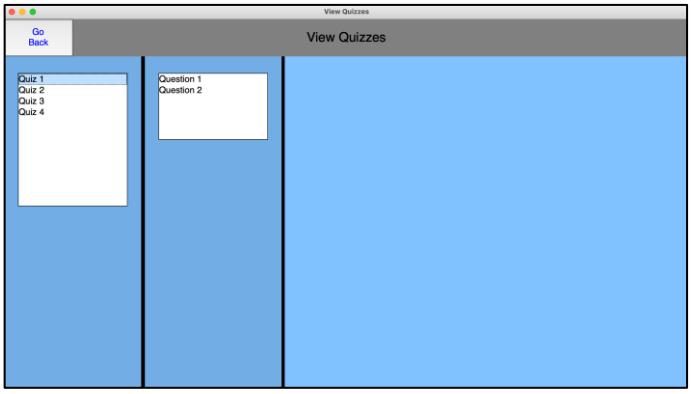
This allowed the function to work as shown above.



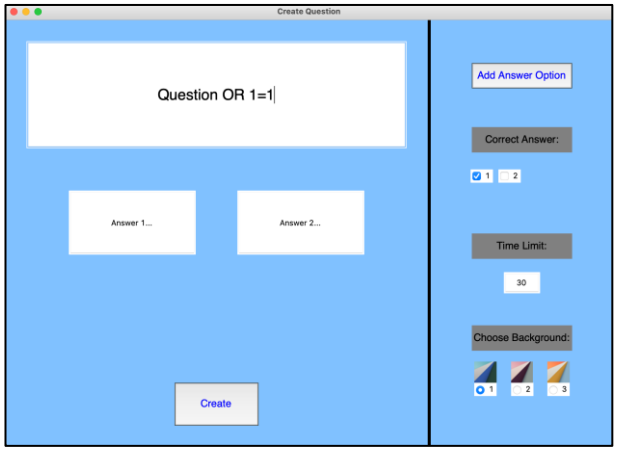


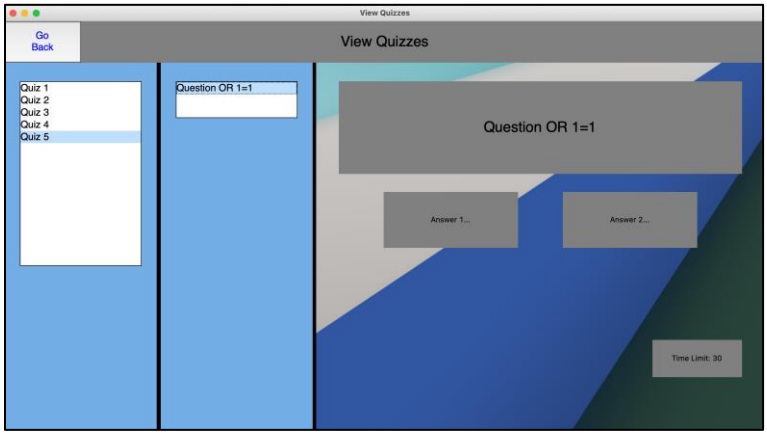


Again there was the inconvenience of not being able to see the full question in the question entry and the question list box but the program effectively wraps around the text when displaying the question label. Since text within an entry box cannot be wrapped around, there are not many options for showing all the text. Although the 'create a quiz by adding…different questions' part of the success criteria has been met, the program fails to either remove a question or quiz, or edit them. Whilst editing has already been brought up by my client and suggestions have been proposed previously, the removal of questions or quizzes had not been brought up. However, the option to delete a question or quiz may be available once a user chooses to edit a quiz. As proposed before, the displayQuestion function can be modified to allow adjustment of the questions. It can also be extended, perhaps, to provide a button that allows the user to delete a given questions. Another button can be provided within the view quizzes function within the quiz list box frame to delete the selected quiz. This can trigger a confirmation pop-up page which asked the user whether they are sure whether they want to delete the selected quiz. The name of the quiz can be displayed to ensure the user has selected the correct quiz along with all its details.



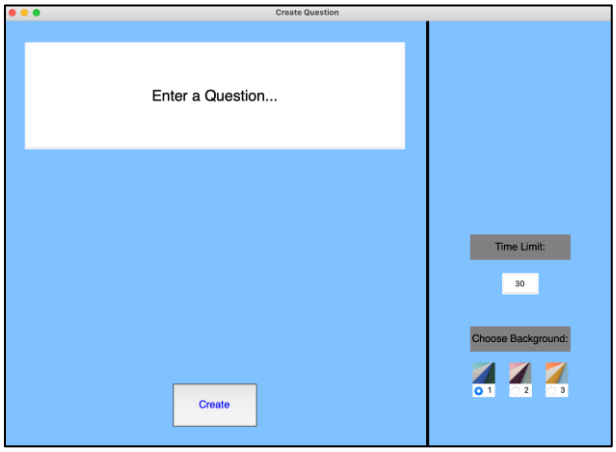
The users then tried to break the program using SQL injection. This was an aspect I considered for my login process, but not for when creating questions, where the users have no restrictions on input. However, the users were not able to break the program in this way due to the way the program retrieved the questions. The questions were retrieved using their associated quiz ID and therefore didn't affect the SQL statement in any way. The same applied for the answer options.



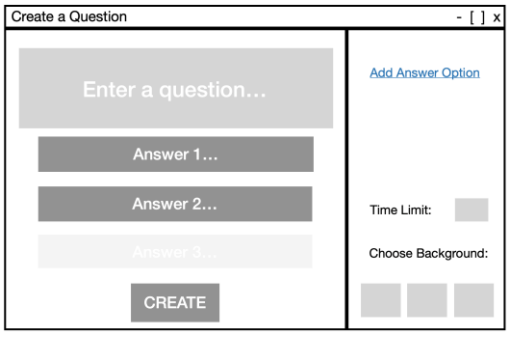


No modifications were needed for this.

Although not explicitly mentioned in my success criteria, when designing my program, there was to be an option for a third question type, one where the user would have a number of answer options and would have to order them in a correct order. Whilst the option appears on my program to do so, the function hasn't been finalised to be able to do so. When the 'Order Answers' option is selected on the main create question menu page, the following is shown:



This was because all three options directed the user to the same function which used an if statement to alter its look and options dependant on the type of question that was passed as a parameter. On the left we see all the aspects of question creation which are constant for all types of question. The createNewQuestion function could hence be extended to include a button to add an answer option as with multiple choice questions to a maximum of four. However, it would need to be modified so as to match the design shown before:



Here, the user would have to enter the question and then the answer options but in the correct order. The program would then shuffle the answers automatically in the displayQuestion function. Since the answer options are limited to a number of four, there may be the odd occasion where the randomisation of the options brings it back to the correct order, although quite unlikely. Consequently, when a user goes on to play such a question, there would be entry boxes beside each answer option in which the user would be limited to inputs between 1 and the number of options, in which they would order the options. Alternatively, I could find a way to allow the user to drag the options into the correct position which would trigger the program to replace two options. This could be done by binding events to the mouse position and click to identify which options to move. The answer can then be checked by

checking the contents of the answers list which would mirror the answers in the database. The database would have to store the answers in correct order from the answerOne field to the answerFour field so it can used when checking answers. If the order from the user matches that of the answer list, it would be shown as correct. If not, it would be incorrect.

Teachers are able to set different questions and accord each one its own time limit. The time limit box has been validated to allow only number options. With the updates suggested above, this part of the success criteria is thus concluded.

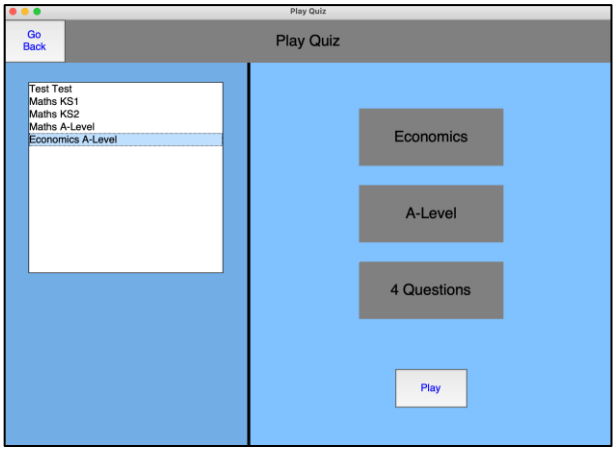
Post-Development Testing for Question Creation:

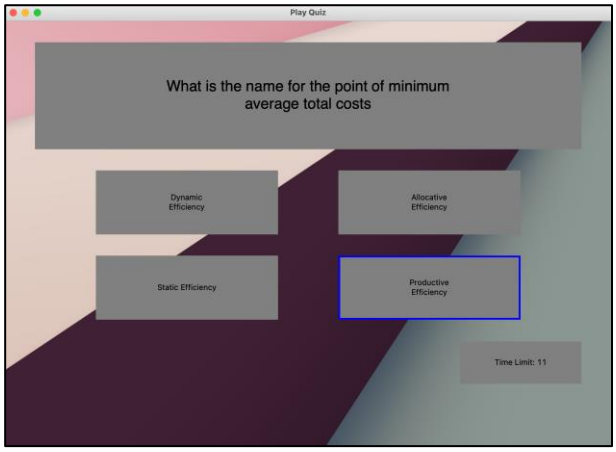
Test Number	Test	Input Data	Expected Outcome	Result
TIME LIMIT ENTRY				
1	Invalid value: symbol, space, letter	'@', '1 2' 'a' Input of this type is disallowed from validation function	Error message Invalid characters do not appear when typed	Successful
2	Valid integer value	'30'	Successfully passes it onto next function	Successful
CHARACTER LIMIT ENTRY				
3	Invalid value: symbol, space, letter	'£', '3 0' 'C' Input of this type is disallowed from validation function	Error message Invalid characters do not appear when typed	Successful
4	Valid integer value	'50'	Successfully passes it onto next function	Successful
BACKGROUND ENTRY				
5	Invalid value: symbol, space, letter	'%', ' ' 'g' Input is limited to only 1, 2 or 3	Error message	Successful
6	Valid integer value between 1 and 3	'2'	Successfully passes it onto next function	Successful

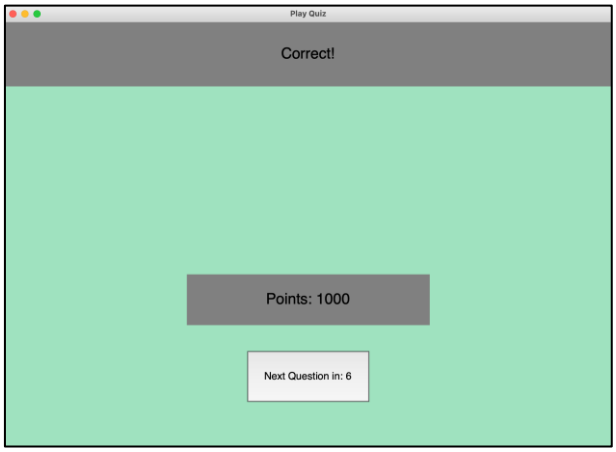
- Pre-created quizzes –
  - Random questions on an exclusive topic or subject shall be created in the form of a quiz of which teachers and students have access to.
  - Questions with a mathematical aspect (of a pre-determined format) shall randomise numbers and consequently calculate answers using equations which shall be incorporated into the program.

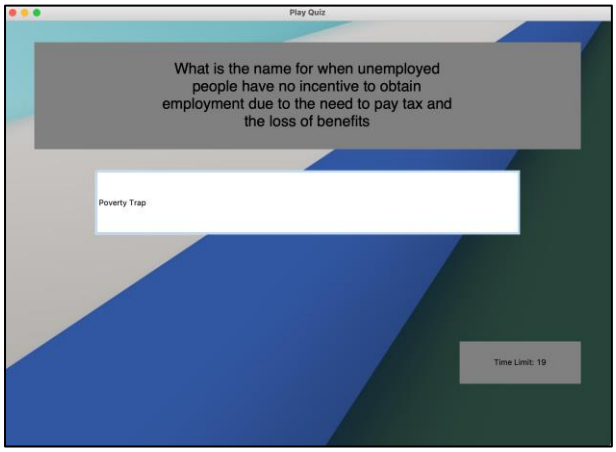
As shown above, this part of the success criteria has been met. Nonetheless, in future development, the number and the range of quizzes can be expanded to accommodate for more subject, topics and question types to fully embrace the content of a subject, and maximise purpose, convenience and effectiveness for the user. Since the program is capable of calculating equations with random variables to calculate an answer to the nearest number, many more quizzes can be created in this way, although it may require modification and extension to be able to handle more complex question types and equations.

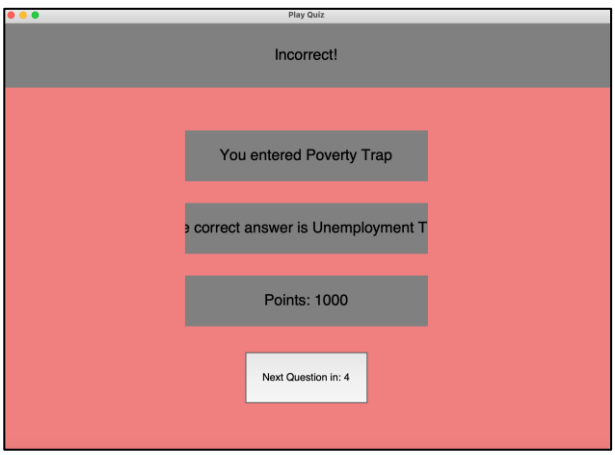
- Points system –
  - Students will earn points for answering a question and shall accumulate them over a quiz.
  - Students will earn an incremental number of points based on an answer streak that develops over a quiz.



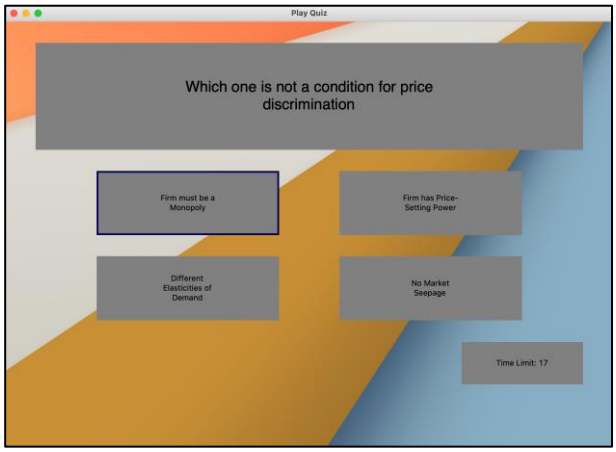


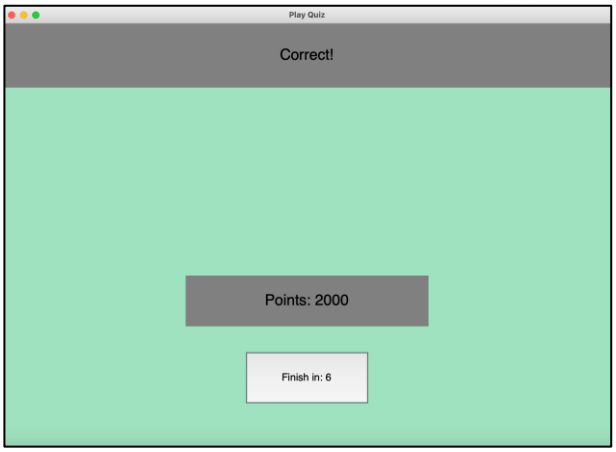


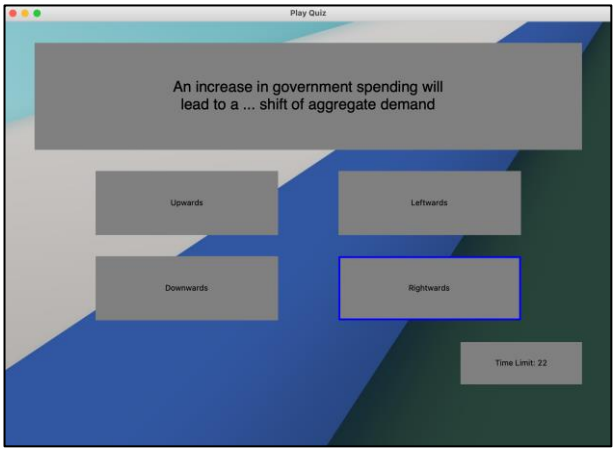


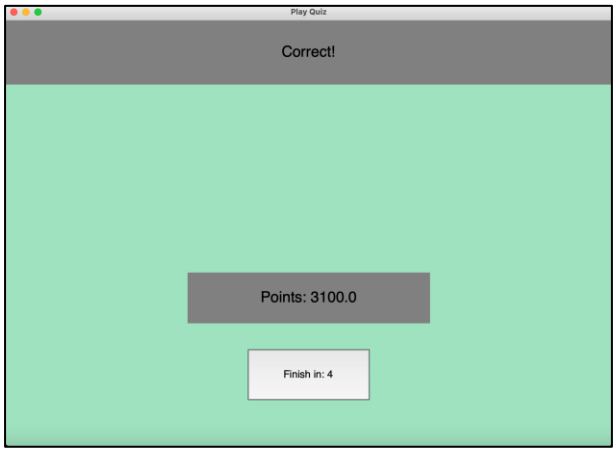


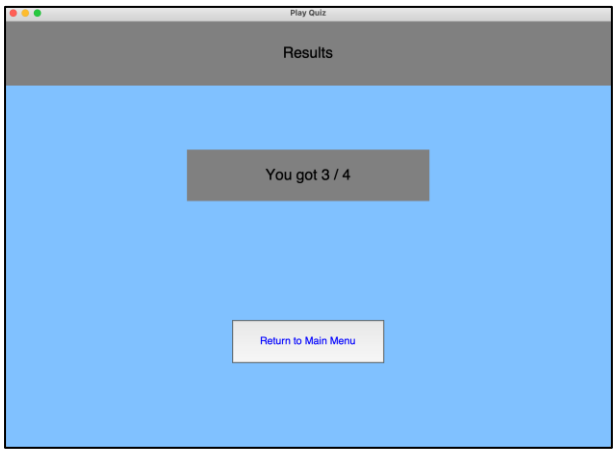
When the user was playing the quiz they had created, they noticed that for answers which were relatively long, they were hidden when shown on the incorrect page. Even if the label was extended, there may be some answers which are still too long to fit in. The program can be modified to check the length of the text and accordingly change the size of the label to make sure the entire answer fits on the screen. This can be done by changing the width of the label when the text reaches a number of characters such that some are hidden and also, the length of the label can also be extended to allow for the text to wrap around.





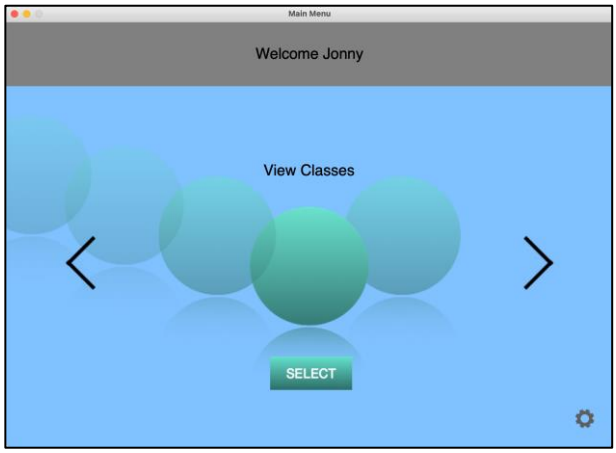


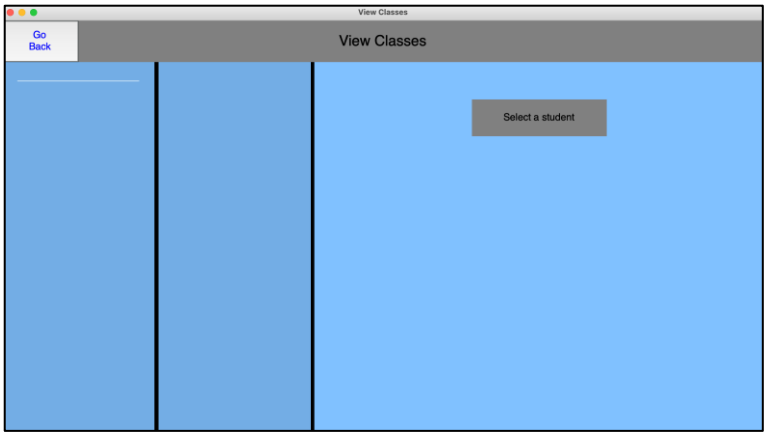




Here, we can see that the user initially obtains 1000 points for correctly answering the first question. Then, the user receives none for incorrectly answering the second question. On to the third question, the user attains another 1000 points for a correct answer, and another 1100 for correctly answering two questions in a row. From this we have evidence to say that both sections of this part of the success criteria has been met.

- Progress tracking –
  - The progress of students from quizzes taken shall be recorded for teachers to see.
  - The progress of students shall be stored in a database.
  - Teachers will have the ability to see progress in the form of a table, chart or graph.
  - Progress will be able to be represented in an external file.
  - Difficult questions / topics shall be identified when students partake in quizzes and available to see to the teachers.

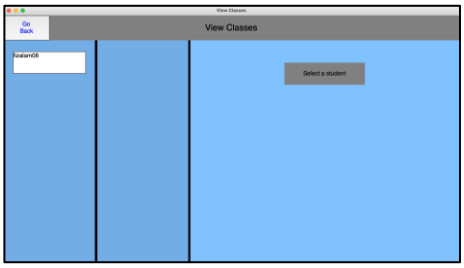


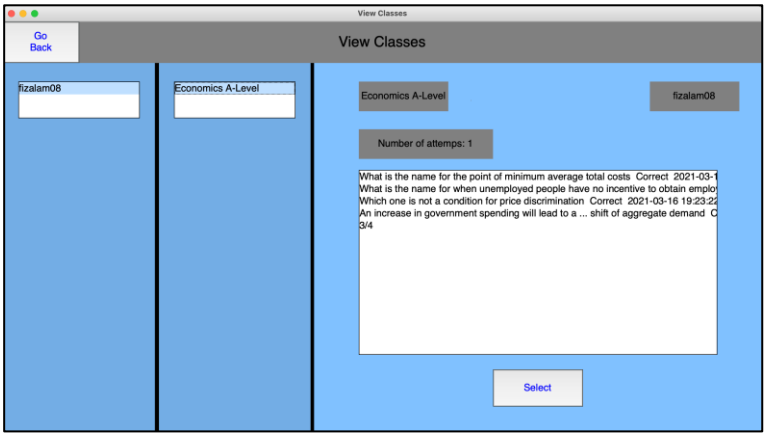


Admin users are able to track progress through the view classes option in the main menu. When the user attempted to do so, the above was produced, showing no students. This was because no students had actually been registered for this admin user. I therefore changed the admin user of the student and parent account created above to that of the admin user created above within the database:

7	Admin	jonnyrogers07	16beeeed555100f537114cb07d0dd74d15a0…	Jonny	Rogers	1	1
8	Student	fizalam08	16beeeed555100f537114cb07d0dd74d15a0…	Fiz	Alam	1	7
9	Parent	jacobeglington10	16beeeed555100f537114cb07d0dd74d15a0…	Jacob	Eglington	1	7

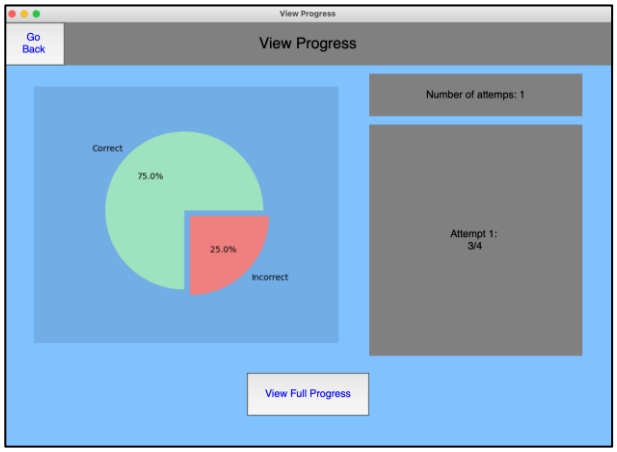
This now produced:

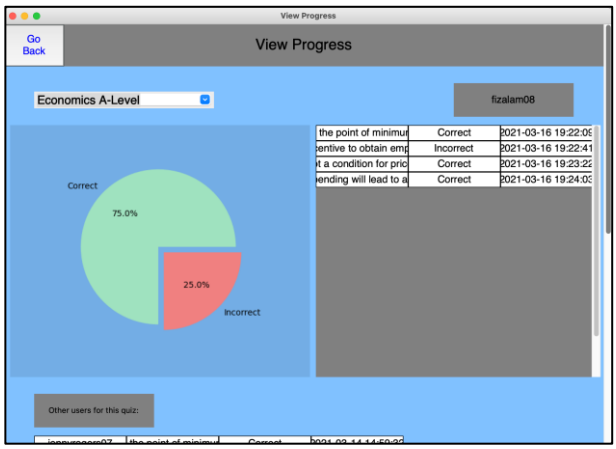




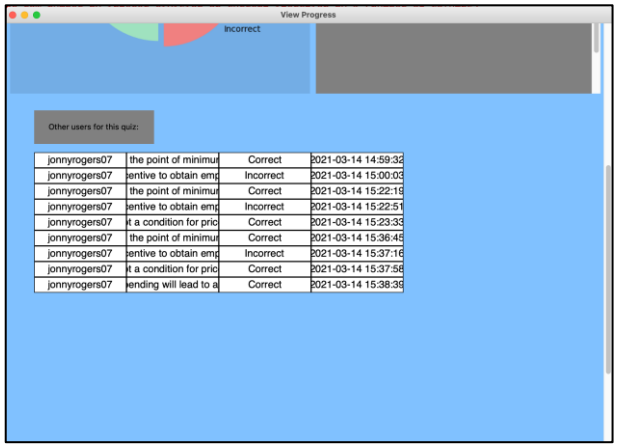
Here, we now see fizalam08 within the students list box.

By selecting the quiz played above by this user, we can see the questions, whether correct and the time for each attempt of each question. Although the list box hides parts of the questions, the user is still able to scroll horizontally to reveal the rest of the questions.





When the user clicked the select button, they were brought to a page that showed a brief outline of the progress for the selected quiz. When they then went on to press the view full progress button and selected the Economics A-Level quiz from the combo-box, all the data for that quiz was shown. Immediately, the user noticed that the questions were, for the most part, hidden. This is difficult to change since the dimensions of the cells are fixed from the createTable function. However, in future development, the best way of dealing with this may be to alter said function to determine a height which is suitable for each row. The height can be worked out by working out how many characters can fit into the cell as it is, and then calculating how many times the text would wrap around. This can be mulitplied by the default height of the cell to allow all the text to fit in. However, this may cause some inconvenience for users if text is wrapped mid-word. This can be avoided by only allowing a wrap from the space before the final character that takes up the cell. This may be difficult, however, since the wrap value is fixed for tkinter labels, but instead of using this method to wrap the text, it can be done manually by inputting the value of the enter button, i.e. '\n' to put the text onto a new line. The calculation of the height would have to be done for every row and would have to be done based on the cell with the largest value set which can be determined by using the len function to find the length of all values within a row. In this case, it would be the cell containing the question. To allow for this iteration, it would have to be done under the for i in range(totalRows): loop, which is outside of the for j in range(totalColumns): loop yet still iterative in the desired way. If all cases are to be considered, this may not be a fool-proof method. If a word exceeds the width of the cell, then it may cause an infinite loop, in which the program always created a new line for the text which is empty since the word cannot fit. In such a case, the program can include an if statement with the condition that if the a word is longer than the width of the cell, it may be wrapped mid-word.



As for the table which shows the progress of all other attempts for the quiz, there is the potential issue that with many users and many attempts, the contents of the table may fall outside the frame and therefore be hidden. To fix this, whilst the scrollable frame can have a default height value, it can be modified accordingly depending on the number of rows in the table. However, if this table follows the same suggestions as above, this will mean that there will not be a set amount of rows that can be used to determine whether the height must be extended. Perhaps, the table could be extended in width here to make use of the full space by making the table use the relative height to the frame to create more space. Then for every new row in the table, the program could retrieve these heights, and simply add it on to that of the frame when a certain, safe number of rows has been reached.

As it is, there is another limitation to this page. This is with the graphs and their associated quiz. In the showProgressGraph function, the program takes in the selection of the quiz from the combo-box as a parameter. It then proceeds to determine the quiz ID of that quiz. However, if two quizzes had the same subject and level, it would take the ID of the first instance of the quiz, and hence may show wrong results. This can be tackled with the suggestion proposed earlier whereby the quizzes are all individually named, and so, the user picks out the name of the quiz and the program determines the quiz ID of the quiz with that name. Despite this, the same conflict may appear if two quizzes are named alike. Nonetheless, this could be avoided if instead of the name, subject or level, the ID of the selected quiz is chosen. Where the quizzes are retrieved, the program can simply retrieve the index

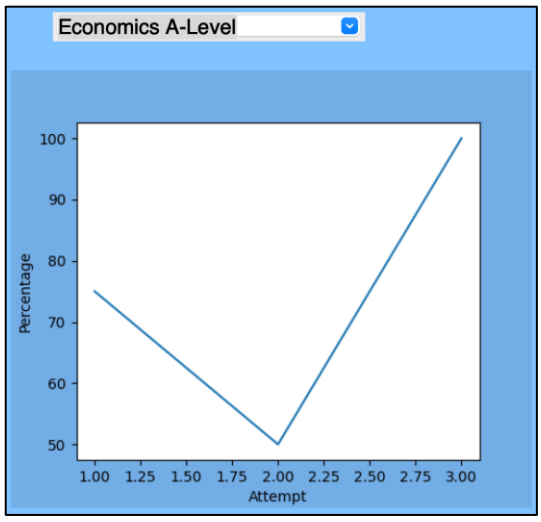
of the row they chose within the combo-box and compare it to the quiz ID of the quiz at that index to then pass on to the showProgressGraph function, thereby creating the correct graph no matter the quiz name, subject or level. This ticks off the 'ability to see progress in the form of a table, chart or graph' part of the success criteria.

As of yet, the program is unable to store and represent any progress data within an external file. In future development, this can be done by importing the csv library and using the table1Data and table2Data lists within the showProgressGraph function to write multiple rows into a csv file by doing the following:

```
// Create and open new csv file with file name 'progress'
with open('progress.csv',  'w+',  newline = '') as file:
    writer = csv.writer(file)
    // Use sublists from table1Data list to create rows in csv file
    writer.writerows(table1Data)
```

This will allow a user to gain access to an external file in a csv format presented as a table, similar to that shown within the program. It could be beneficial to also add a default sublist within the table1Data and table2Data lists containing the relevant headers, i.e. 'Question', 'Answer' and 'Time'. This would also be beneficial when viewing the progress within the program. However, the method shown above causes issues. This is that there will only ever be one file created that is overwritten with new data. To fix this, the name of the file can be manually given an appropriate name within the program to prevent any overwriting. This can be done by creating a new filename variable and assigning perhaps the username of the student, the particular quiz name if only one quiz or 'All Progress' if all quizzes and the date and time of writing to the file to make sure no files can ever be mistakingly overwritten. In the algorithm above, this can then be shown by with open(filename + '.csv'),….

The process for the identification of difficulties within the program has been detailed above and thus concludes this part of the success criteria. However, another part of the program that was considered when designing, albeit not on the success criteria, was the the prediction of grades for students based on their progress from the quizzes. To predict grades of students, I created a line of best fit on the line graph shown when viewing progress to see the general progress of the student. To do this, I first created more progress records within the database so the graph looked like this:

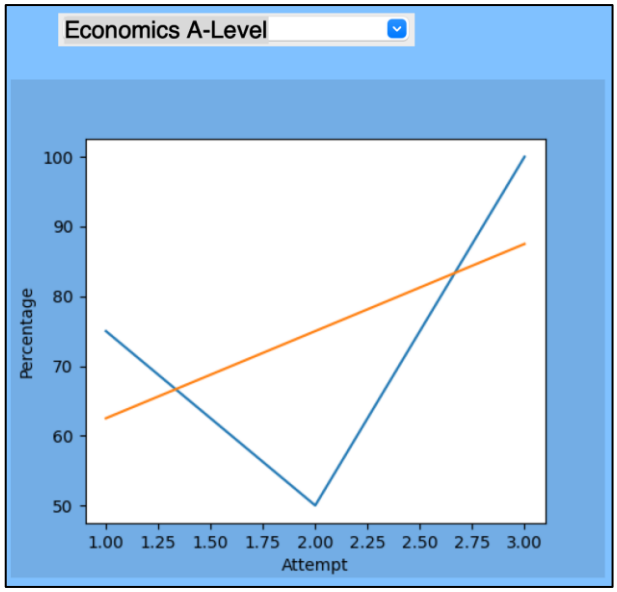


I then implemented this bit of code within the createLineGraph function:

```
## Create and plot line of best fit
x = np.array(attempts)
y = np.array(percentages)
m, c = np.polyfit(x, y, 1)
ax.plot(x, m*x + c)
```

Here, I convert the attempts and percentages list into an array compatible with the numpy library and assigned it to its correct coordinate axis. I then used the numpy library to calculate the gradient and the y-intercept of the line of best fit, which I then plotted with an x-coordinate from the attempts list and the y-coordinate from the gradient mulitplied by the x-coordinate added to the y-intercept (following the format of y = mx + c to form a straight line).

This now produced:



In future development, this can be extended so that the program can use this line of best fit to confidently speculate realistic future grades. This can be done perhaps by taking the percentage at the y-coordinate of the last attempt on the line of best fit. In this diagram, this would be taken from the third attempt and would show a percentage of around 88%. The program can then have grade boundaries entered by the teacher to calculate the grades. If the student is on the border of two grades, the gradient of the line can be used to factor in whether or not it is likely, at their current escalation of progress, that they could achieve the next boundary. This could be done by taking the value of m in the createLineGraph and checking whether it is above or below a default value of perhaps 15-20 to show a general increase of progress of 15-20%.

- User-friendly user interface –
  - Separate menus for admin users, students, parents; admin users should have extra accessibilities such as the creation of quizzes, tracking students' progress; students will be able to access teacher-created quizzes or pre-created quizzes as well as their own progress; parent users will be able to view the progress of their child
  - Clear headings and sub-headings to indicate the content of each page and relevant images if needed.
  - Use of modern, user-friendly colours for pages and buttons
  - Clear labels or logos for buttons

All three main menu pages have clear options with clear labels which collectively provide modern, user-friendly interfaces, as agreed upon by my client. They provide functionalities which are complete to an admin user and restricted to a student or parent user. With the suggestions made previously, where a parent has their children assigned to them within the database, they will be able to view the progress of their children. Each page has a clear welcome label that describes the page and all buttons, labels, list-boxes, combo-boxes, etc. are written clearly for the user to see. However, although proposed earlier, a function to change the dimensions of the pages hasn't been finalised. This is because for the main menu pages, it requires changes to the animation in the admin menu page and to the image sizes in the student and parent menu pages. Perhaps, for simplification, when changing dimensions of the admin menu page, the menu options can simply be shifted to the middle of the page with new dimensions, and the text for the options can be enlarged, leaving the answer options the same size. In future development, perhaps the size of the images can be modified too, using a general formula to modify its movement. This formula would include shifting the images by a factor of the height and width of them. The use of a formula can also be used for the student and parent menu page to scale the image by a factor of the new scale of dimensions. Perhaps it may be better to consider using less complicated shapes for this. All pages use modern, user-friendly colours as do the buttons, which are all clearly labelled. Whilst the interfaces all appear to function as desired, the users did find a glitch when using the admin main menu page. This was when a user did not wait for the animation (or movement) of the options to finish before pressing the arrow another time. This caused the menu options to fall out of the desired movement pattern. This way because, the program used the position of the menu options from when the arrow button was pressed. To

combat this, it may be beneficial to use an if or while statement to disable the effects of the arrow buttons when one of the options are out of place. If one option is out of place, the others will be so too. This can be done by finding the coordinates of the central menu option and seeing whether they match (399, 300) as so placed in the adminMainMenuPage function. In addition, it may be useful to speed up the movement of the menu options.

In exploration of other parts of the program, the users identified some other issues that were in need of improvement. Firstly, on many of the interface pages, there are buttons to allow the user to return to the main menu however this only worked for the admin main menu page since I hadn't created the student and parent menu page at the time of creating this function. I therefore modified this function to redirect the user to the correct main menu page, for all admin, student and parent users:

```
def returnToMainMenu(self, previousPage, username):
    previousPage.destroy()
    userType = viewingFromDatabase.viewFromDatabase(c, False, False, 'accountType', 'username', "'"+username+"'", 'UserDetails')
    if userType[0][0] == 'Admin':
        mainMenu.adminMainMenuPage(username, Login.titleFont, Login.buttonFont, Login.defaultFont)
    if userType[0][0] == 'Student':
        mainMenu.studentMainMenuPage(username, Login.titleFont, Login.buttonFont, Login.defaultFont)
    if userType[0][0] == 'Parent':
        mainMenu.parentMainMenuPage(username, Login.titleFont, Login.buttonFont, Login.defaultFont)
```

This now allowed the return buttons to work as desired.

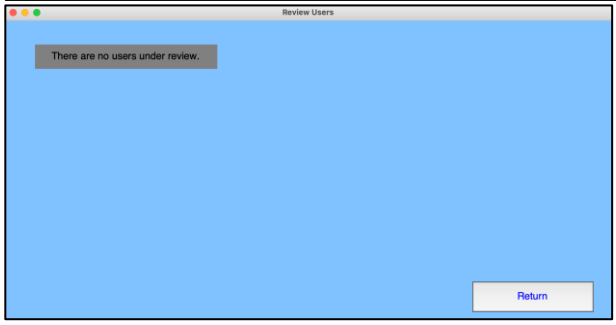
Another limitation that the users found with the program is within the settings, for when reviewing users. Since all users have a corresponding admin user that is determined when they create an account, any reviews, i.e. to unblock the account, to reset the password or to verify the user, have to be shown to the associated admin user and only them. The program fails to do so. In the reviewUsers function, a new table is created by performing an inner join on the UserDetails and UserReview tables. To resolve this issue, I can either modify the database class to allow the inner join function to take in a condition using WHERE field = fieldValue, where field is userID and the fieldValue is the admin ID of the user from the database. Alternatively, I can modify the if statement in the reviewUsers function to append to the data list those records where the review ID is greater than one, but also where the admin ID is equal to that in the database. This would require me to also join the adminID field to the new inner-joined table.
The users also noticed another issue here with this function. If an admin user has no users under review, an empty list goes to the createTable function and causes an error:

```
    totalColumns = len(data[0])
IndexError: list index out of range
```

This can be avoided however using an if statement which first checks the length of the data list:

```
## If data list is not empty...
if len(data) > 0:
    ## Create table with data from new table
    cells = self.createTable(data, tableFrame, Login.defaultFont)
## If empty...
else:
    ## Change label text
    selectUserLabel.config(text = 'There are no users under review.')
```

This fixed said error and produced this result:



Also within the settings, there were issues with the change dimensions option (briefed above) and the my account option. In the my account option, there will simply be a page with labels displaying a user's username, and an option to change their password. This could use the same function as in the create account function to validate the password and ask the user to confirm it by retyping it. This will then be hashed and updated in the database using the database class.

The users also experienced some inconveniences when playing a quiz. First of these were the need to wait for the full time allotted for a question and the waiting time between questions. The users suggested an option to skip this. This could be done by creating a 'Skip' button that changes the value of the time limit to 0, in which the next page would appear.
The users also noticed an inconsistency with the play quiz functions. When playing a quiz which contains multiple answer input questions, the value inputted from a previous question carries over to a following question. This should be emptied and can be done by setting the text of the entry to '' before it is shown. They also noticed minor character nuisances such as the points written as a decimal, e.g. 3100.0 and when showing the details of a quiz with one questions displaying '1 Questions'. To fix the points issue, before it is shown on the page, it can be converted to an integer. As for the quiz details issue, that can be resolved either by changing the text to 'Question(s)' or using an if statement to first check whether the number of questions is above 1. If it is, the text can be set to 'Questions'. If it is 1, it can be set to 'Question'.

Next, I proceeded to white box testing, where I explored the limitations of the source code. Firstly, I investigated my database class. In this class, I noticed that when viewing from the database, the program stored this data where each record of the retrieved data was stored as a tuple within a two-dimensional list. Because of this, there was an excessive requirement to remove said tuple and lists to isolate the relevant data. I propose that in future development and maintenance of the program, this be dealt with within the viewFromDatabase function itself (where instead of returning c.fetchall(), the function could return the value of this data but with two indices of 0, i.e. data[0][0]), thus isolating the relevant data and preventing the need to do so constantly throughout the program. This would however require modifying the current program to fit this format and may also cause problems with the createTable function that requires its data in this form (although sufficiently easy to manually do).

Another issue that was identified with my database class was the opportunity for corruption of data, albeit in rare cases. Where I have created a code to protect the atomicity of database transactions, whilst the program makes use of a try except statement in case of a failure on the part of the program, it fails to provide a fool-proof tactic against hardware crashes and the like. If the program is midway through the try statement and the hardware were to crash, the program would never have the chance to execute the except part of the statement, thus failing to revert the database to its previous state, thereby potentially corrupting the data. The program does attempt partially to resolve this problem by saving any transactions first into an external text file before execution (consequently removed after), however, the same problem applies for if the hardware were to crash before this happens. To deal with this, the program may have to first read the contents of the text file upon starting and determine whether a full statement has been stored. If not, the file can be erased, and a notice can be given to the user. This may require fine-tuning as to make sure the notice is given to the correct user.

Correspondingly, in regards to the external text file, there may be some issues concerning security, since it may be easy to access for some users and can therefore lead to information being accessed without correct authorisation. In this case, the file could be encrypted, although this would require any encryption key to be well hidden or difficult to decode if to be stored within the program. this

may not be necessary, however, since sensitive data such as the password is formerly hashed before storing the data externally. It may be useful to hash more sensitive data though, such as the username and perhaps first and last names to maximise security for the users. This would make gaining access to both the database and the external file insignificant. This may be particularly useful if a hacker tried to revere engineer the hash values by creating an account (although supposedly irreversible, they may be able to calculate parts of the password). In this way, they would be able to match a hash value with a known password and thus attempt reversing the hash to obtain the real passwords of all other users. By hashing the usernames, a hacker may not be able to make this match, since they cannot match their account to one within the table. This isn't perfect though since if newly created, it would appear at the bottom of the table. To make this process more difficult for a hacker, the program can make use of a salt value, which is added on to a user's password before being hashed as an extra layer of security. This may be a constant value such as the string 'salt' or may change for each user. This can be done by using the user ID (since unique to the user) along with a formula to create a unique salt value for each user. The formula could follow the format of 'salt' + str((userID * 10 – 3)!) where ! is the factorial value. In the rare case that multiple passwords produce the same hash value, using this salt value would also prevent that from occurring.