7517/3, 2025

Online Website-based Revision Quizzer

Harry Kinchin

WOOD GREEN SCHOOL

Table of Contents

Analysis

Description of the problem

My aim for this NEA is to create a revision quiz accessible via a website in which a user can assess themselves on various subjects. The end users of my project are students doing A-Level mathematics and computer science; as, for the purposes of this project, they will be the two subjects available to revise in my solution.
I will be using the following coding languages: Python for the back-end functionality; SQL (via python's SQLite package) for database management; and HTML, CSS and JS for the website's HCI.

Research of similar solutions

While doing some background research of this problem, I came across three similar solutions: Educake (1), Quizizz (2), and Carousel (3). These websites have been created for students to use to revise their subjects in various manners:

- Educake uses premade questions from a database, for each exam board, subject, and topic. From which users can select a topic(s) to be questioned on, or a teacher can set a class questions.
- Quizizz relies on the creation of quizzes from its users to study with, resulting in a wide variety of subject (and fun) themed quizzes.
- Carousel also uses premade questions, but instead of comparing the answer of the user to the answer given, the user in shown both their answer and the site's answer and are asked to compare and grade themselves.

All of these are elegant examples of a revision quiz website, but all are largely different in their approach- when compared to each other and the design of my own project- that I am not worried about potential issues of copying their methods.

Research of my solution's needs

After a meeting with my client for the project (a student), I have created a set of objectives for the project, which would be tailored to fit the needs of the student. I have created both a list of project objectives, as well as a diagram showing the connections between various aspects of my project.

Project objectives

1. User is sent to a login page where they can choose to either login or create an account.
    - 1.1. If they choose to login, they enter their username, password, and email, which is then checked with a database of login information and either confirms the user and sends them on, or tells them the information is incorrect and prompts for a re-enter.
    - 1.2. If they choose to create an account, they can enter a username, password, and email address.
        - 1.2.1. The password will be checked upon entering to see if it meets the criteria for a safe password.
        - 1.2.2. The username and email address will be checked that it is unique, and prompted to change if it is not.
        - 1.2.3. The information will be added to the database, with the password hashed for security.
2. The user is taken to the account page, in which they can see their personal details (username, password, email), and subjects taken.
    - 2.1. All information on the account page can be changed, with the username and password requiring a login check to be changed.
3. On the subjects' page, the user can select a subject from their chosen subjects, and do one of the following.
    - 3.1. The user can select a number of topics, and a number of questions, to create a quiz.

3.1.1. They will then be taken to a page with the questions that they can answer and afterwards see if they were correct or not.

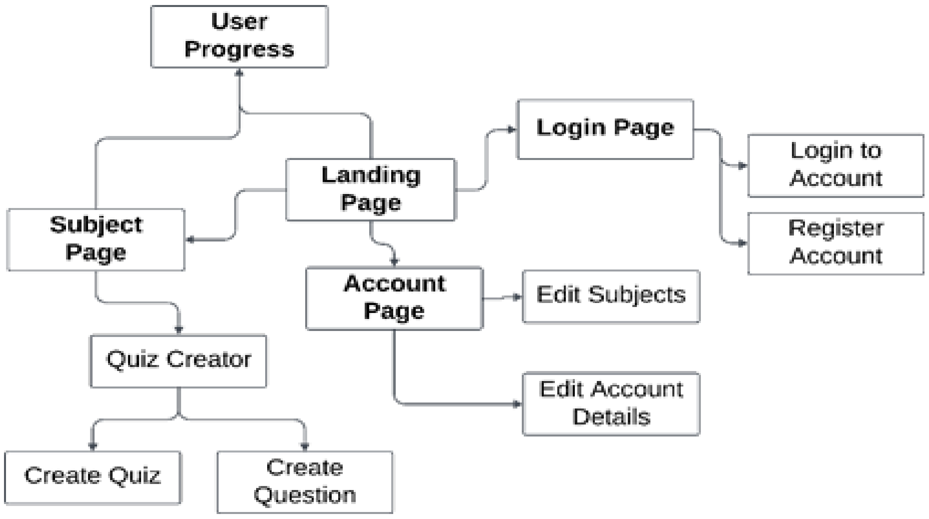
3.2. The user can create their own question, by choosing a topic, the type of question, the question itself, and the answer(s).

3.3. The user can also (from this page and main page) see their progress, broken down into a RAG format (red, amber, green) for each topic and overall, and for each subject.

4. The user progress page will be laid out by subjects and topics, it will also display a traffic light style design beside each topic, showing their confidence level on a given topic.

4.1. The user can edit their RAG value depending on how confident they are in each topic.

Project sitemap



This is a rough outline of the sitemap of my project, which aims to show how the user can navigate between areas of the website, and what actions can be done on each page.

Use of external resources

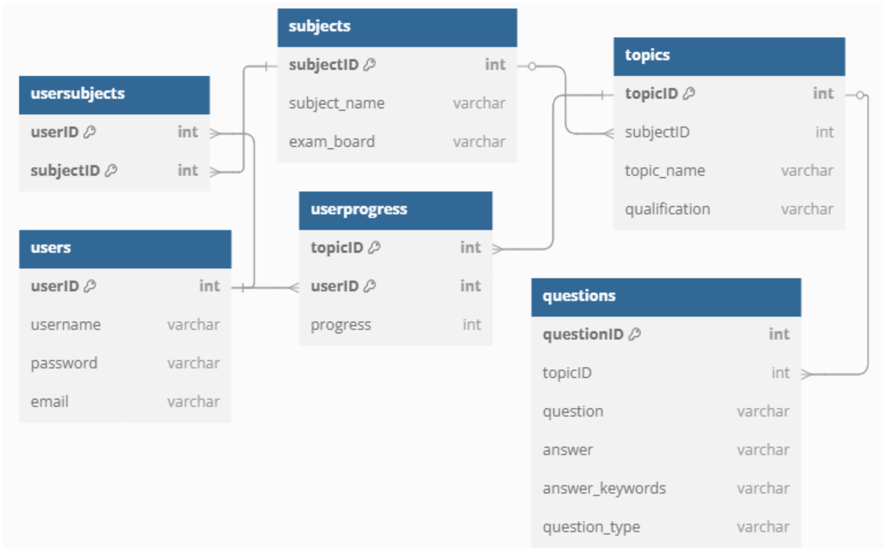
During the summer months before starting my project- as well as during the first handful of weeks into the project- I was researching and looking at various tutorials and explanations of how various packages and languages work. All websites used to aid in my learning of how to carry out my project can be located in the **Error! Reference source not found.** section of this documentation. The main purpose of these sites were for the purpose of learning JavaScript (4) (5), CSS (6), and Flask (7) (8) for their various uses throughout my project. The use of any other external resources will be mentioned and referenced throughout this project's documentation.

Documented design

This section contains my plans for my projects, including: database layout; database-website interaction; webpage designs; and pseudo forms of algorithms I plan to have in my technical solution. Unless otherwise stated, all of my tables, flow-charts, and other designs were all made using the Lucid (9) suite of design products (LucidSpark and LucidChart). This is due to its accessible design features that make creating such diagrams easy and simple, while still allowing versatility and complexity.

Database design

Following discussions with my end user, and personal design sessions, I have created a database table layout that fulfils both the needs of the project and meets all 3 database normalisation requirements. It has no repeated or redundant field; no partial key dependencies; and all fields depend on the whole key in their respective tables. Here is the diagram I have produced to articulate the design of the database (left):



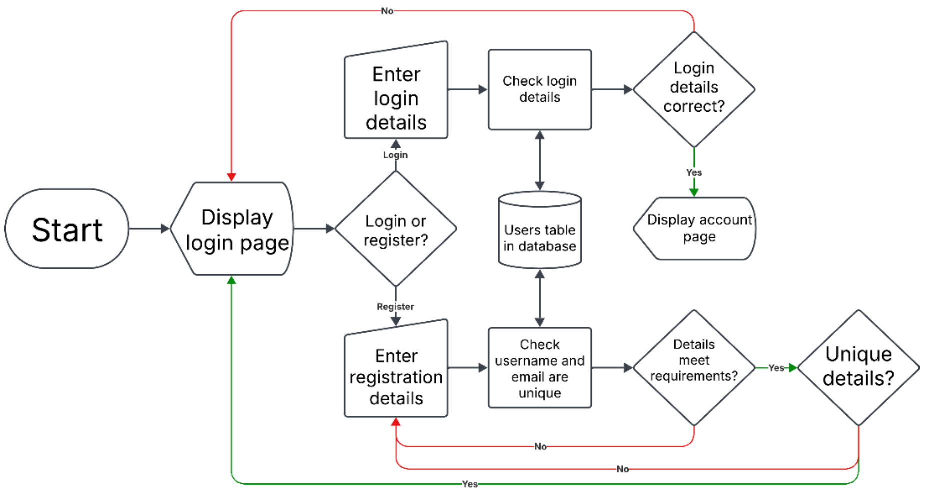
This design was made using dbdiagram.io (10); while it does use their own pseudocode to create the diagram, it did not aid in my own coding of the database in my project as the on-website coding is most similar to a Java formatting of coding, and my own database coding was written in python using the SQLite (11) package.

Database-website interaction design

Here I have created some flow-charts for various interactions between the user and the database within my website. These flow charts were made to illustrate how each page of my website functions and specifically how they will utilise the database to perform certain actions.

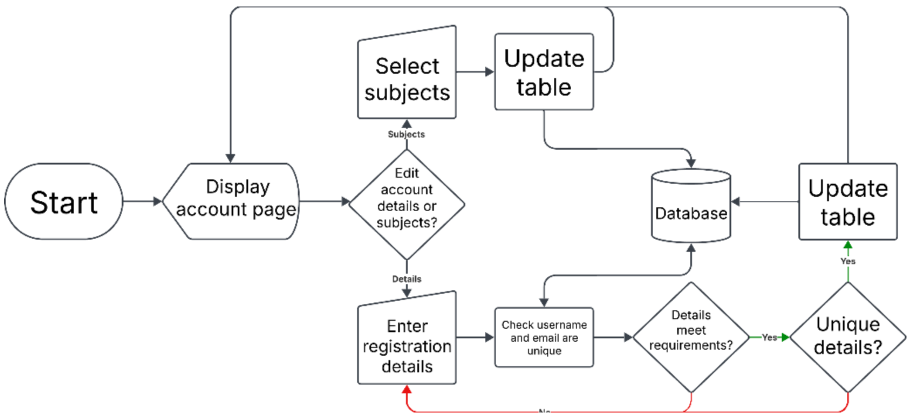
Login page interactions

This is a flow-chart designed to show how a user gets from the login page to the next area of the website. When the website is initially loaded, the user will automatically be redirected to the login page from anywhere else, and will have to either create an account or login to their existing account.



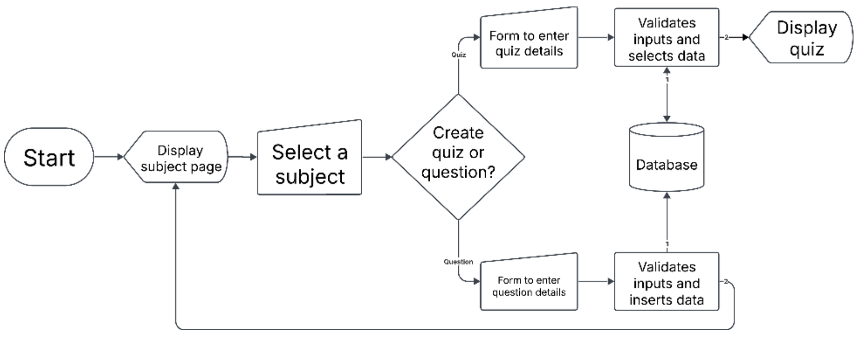
This design contains checks to ensure that duplicate data cannot be added to the database; and making sure that only one account per email is allowed.

Account page interactions


Start
Display account page
Select subjects
Update table
Subjects
Edit account details or subjects?
Details
Enter registration details
Check username and email are unique
Details meet requirements?
Yes
Unique details?
Yes
Update table
Database
No

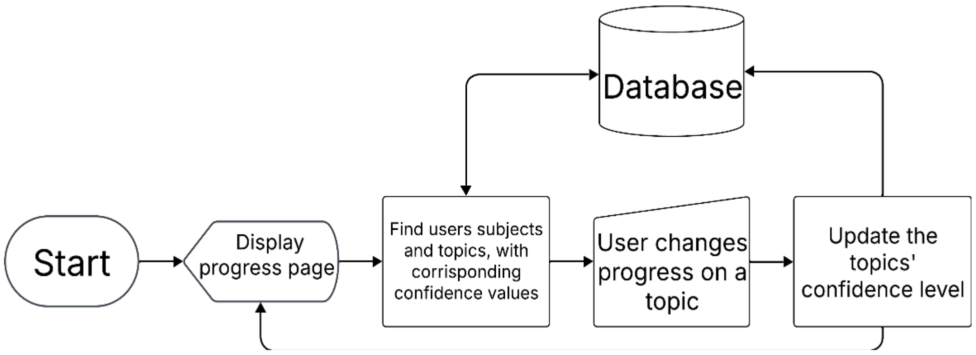
On the account page, the user can view their account details, as well as the subjects they are taking. From this page they can edit both of these sets of information. This flow-chart illustrates the way in which the user can choose to edit either their subjects or information, and the process of how that information is use in the database. The way in which the user information is purposely similar to the way in which a new user is created on the login page, this is to add reusability to the project so that there is no redundant code that could be reused rather than rewritten.

Subject page interactions


Start
Display subject page
Select a subject
Create quiz or question?
Quiz
Form to enter quiz details
Validates inputs and selects data
Display quiz
Database
Question
Form to enter question details
Validates inputs and inserts data

The subject page is where the user can create their own questions, or create their own quiz, both based on their chosen subjects. This flow-chart shows the way in which the database is used to create and retrieve rows. This is the main purpose of the website and thus has the most functionality when compared to other areas.

User-progress page interactions


Database
Start
Display progress page
Find users subjects and topics, with corrisponding confidence values
User changes progress on a topic
Update the topics' confidence level

The user progress page is used to display a student's confidence in the topics of their chosen subject; this page does not entail much detail as it simply shows the user their progress in each topic and allows them to edit their RAG level too.

Landing page interactions

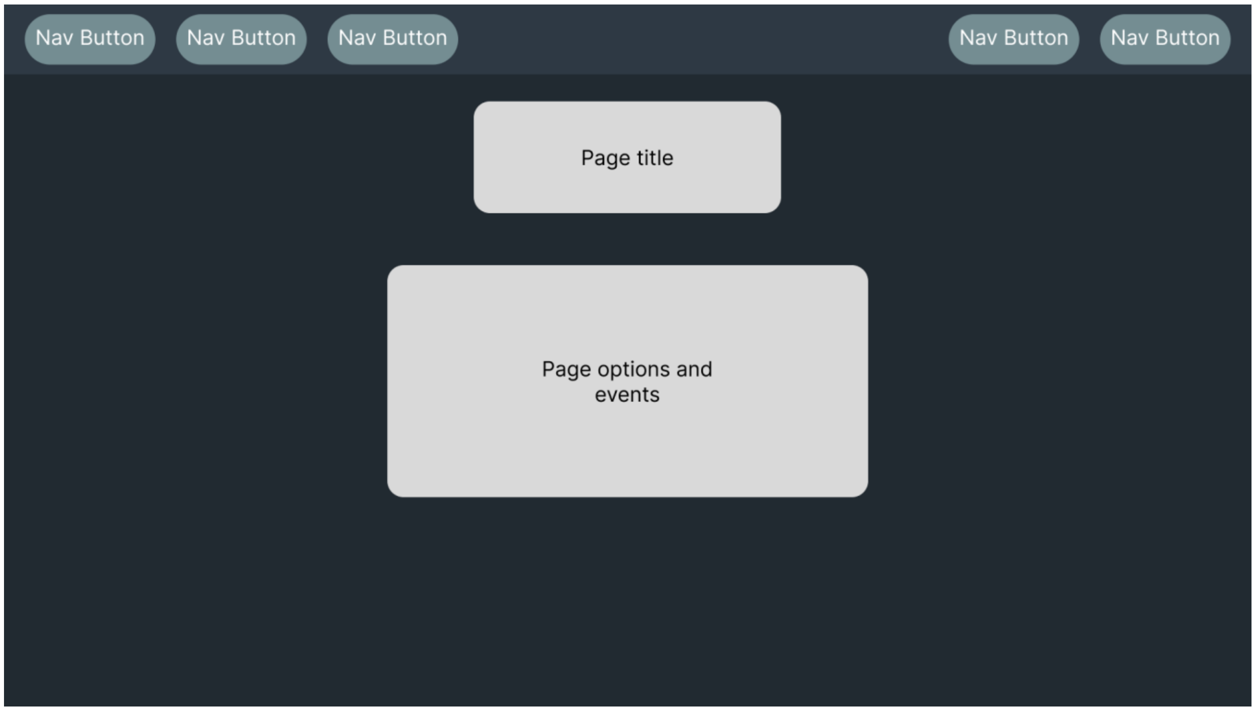
The landing page has no interactions with the database, simply because it is the hub of the website in which you can navigate to the other pages. There is no need for it to create, retrieve, update, or delete data from the database so I have aptly chosen not to create a diagram for this section of the documentation.

Human-computer interaction and page designs

This heading contains rough plans for the layouts of various webpages throughout my site, with annotations labelling why I have made certain choices and how I carried out the logic presented in the chapter before. These designs were made using the Figma (12) software, this is because it is an intuitive software to use to be able to design webpages and allow for versatility when creating mock-ups.

Wireframe for all webpages

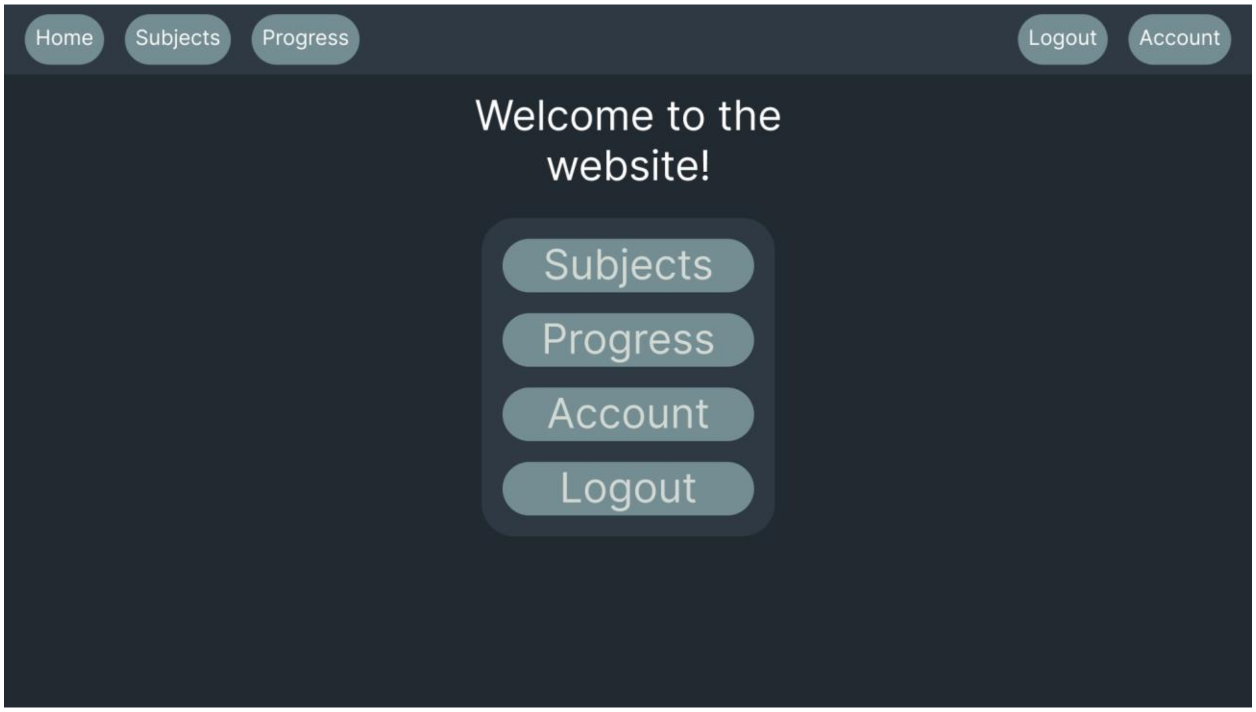
This is a basic design that I will be using for all the pages on my website, every other page will be based on this design and will use it for any styling and layout planning.



This design is themed around greys and blues, this is so it is easy on the eyes while still looking pleasant. All of the webpages on my site will be based on this, having the nav bar situated at the top, with a header/title underneath, followed by the main contents of the page.

Landing page design

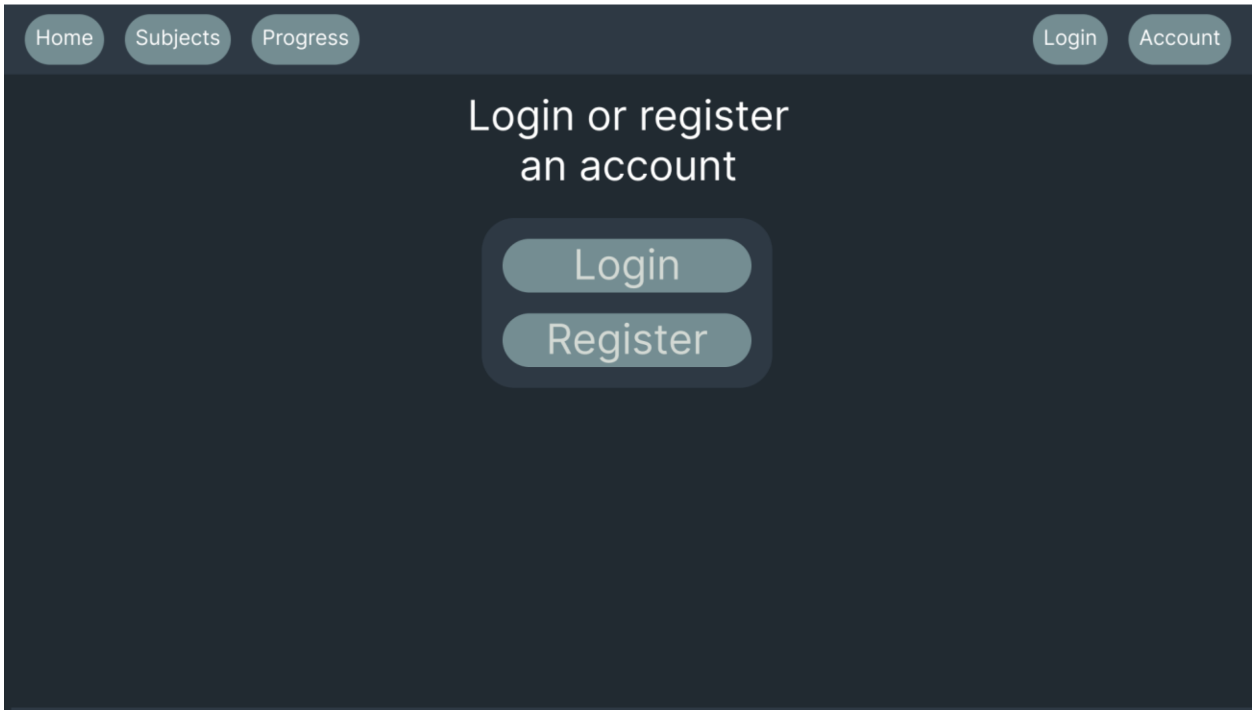
The landing page is where a user can access all areas of the website, and thus only really needs functionality to redirect to other places. Due to this, it has a rather straightforward design that allows for easy navigation.

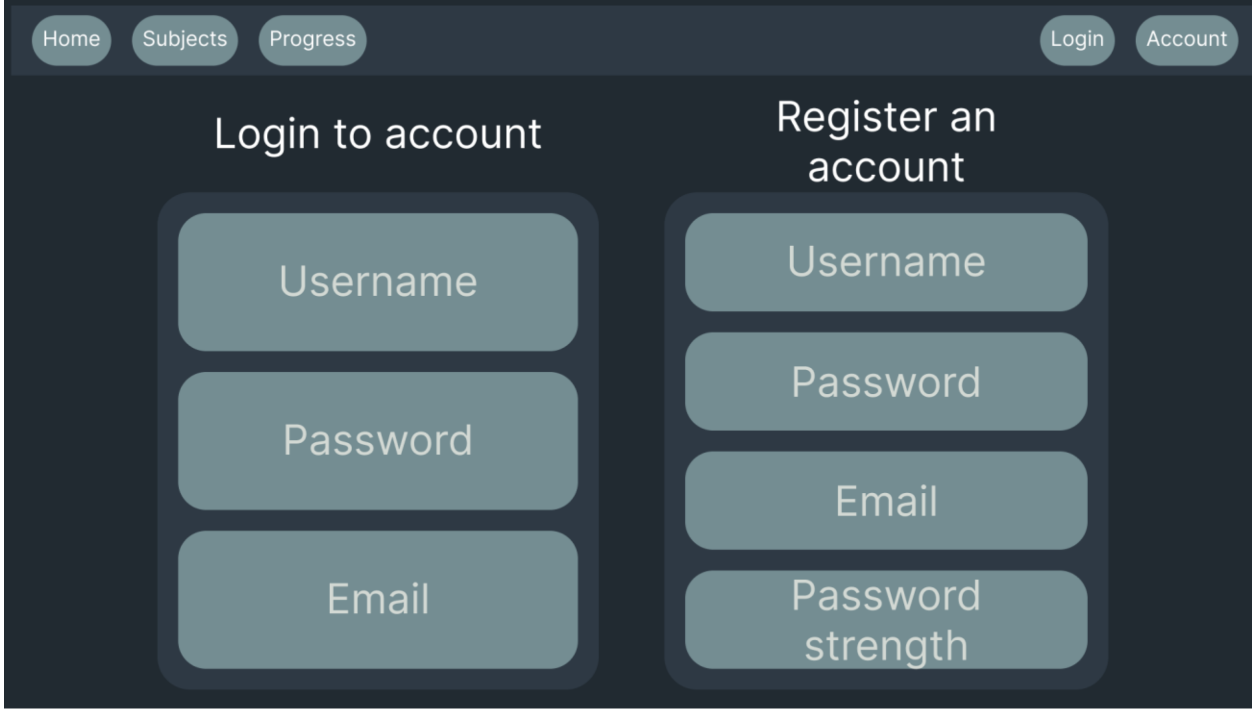


As the imagine shows, this page only contains the bare minimum to be useful, having the navigation bar (which will be universal across all pages), and below containing buttons to take the user to the other areas of the website. The dual choice of the contents' buttons and the nav bar buttons does make the other one redundant, but as this is simple the landing page of the website I believe that does not matter as there is nothing else that would be useful to have on the landing page, anything else that could be put here would make more sense on another page.

Login page design

As seen in the Login page interactions, the page is where the user can login to their existing account or create a new account, and the design of the page allows for easy human understanding of what they can do and how they can do it.

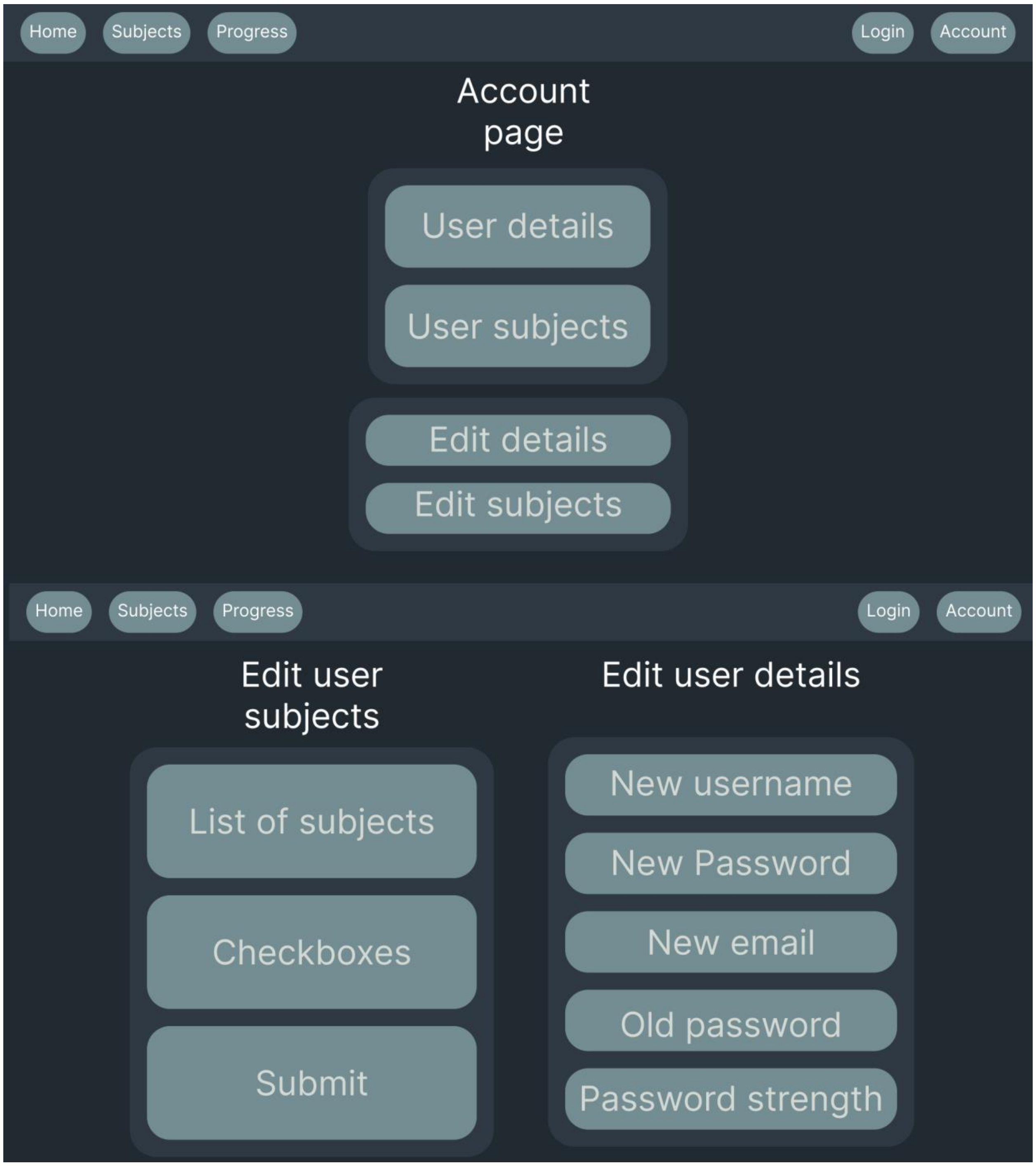




The second of these pictures represents what happens when the user selects one of the two options; if they choose to login, they will be shown a form to fill in with the relevant information, and if they choose to create an account, then they will be prompted to fill in the data, and their password will be checked to make sure it is secure enough.

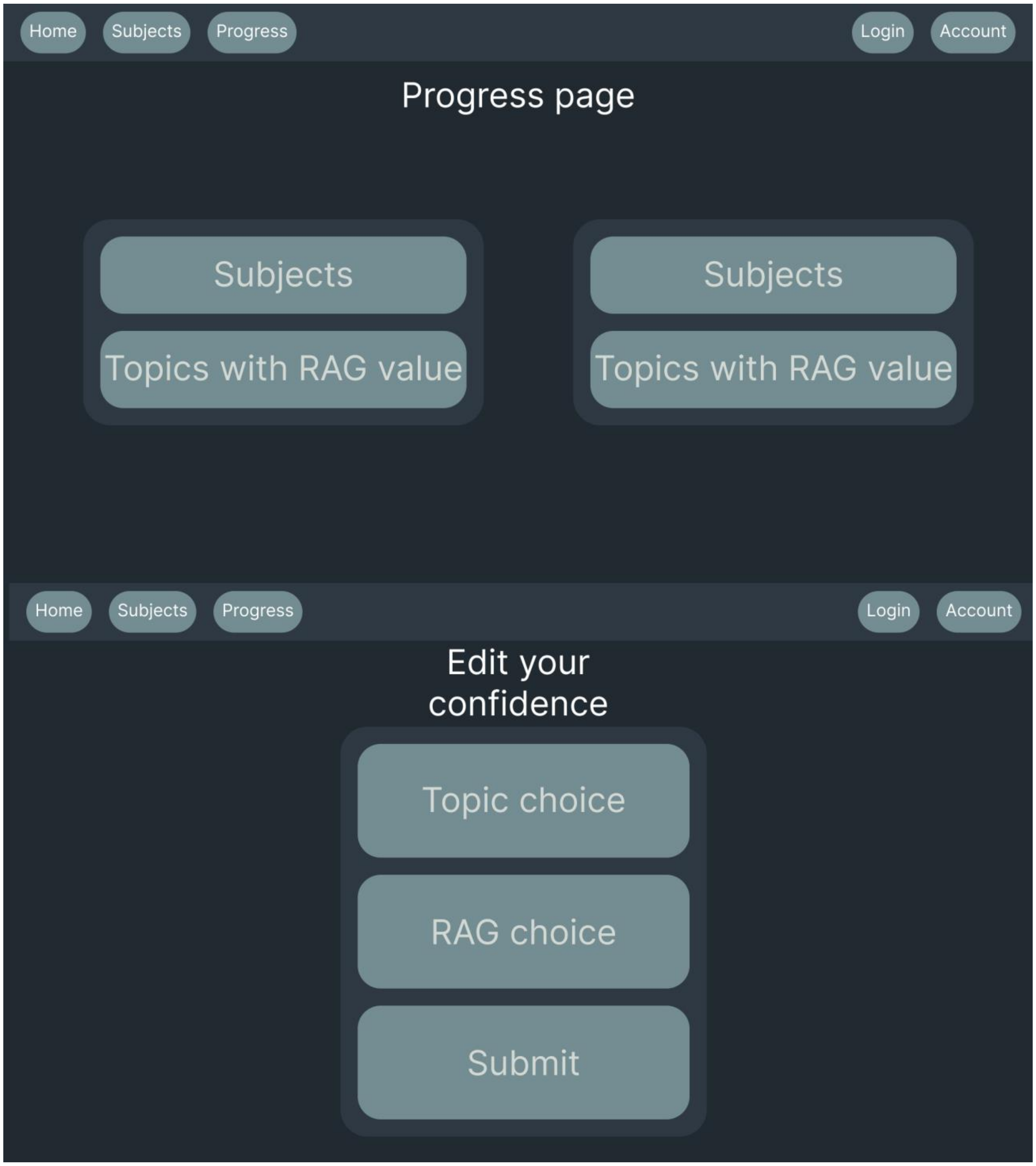
Account page design

The account page is where the user can see their details, as well as what subjects they are taking. Both of these can be edited via a form and will therefore be laid out in a way similar to that of the login page, which also has two forms to choose from on its page.



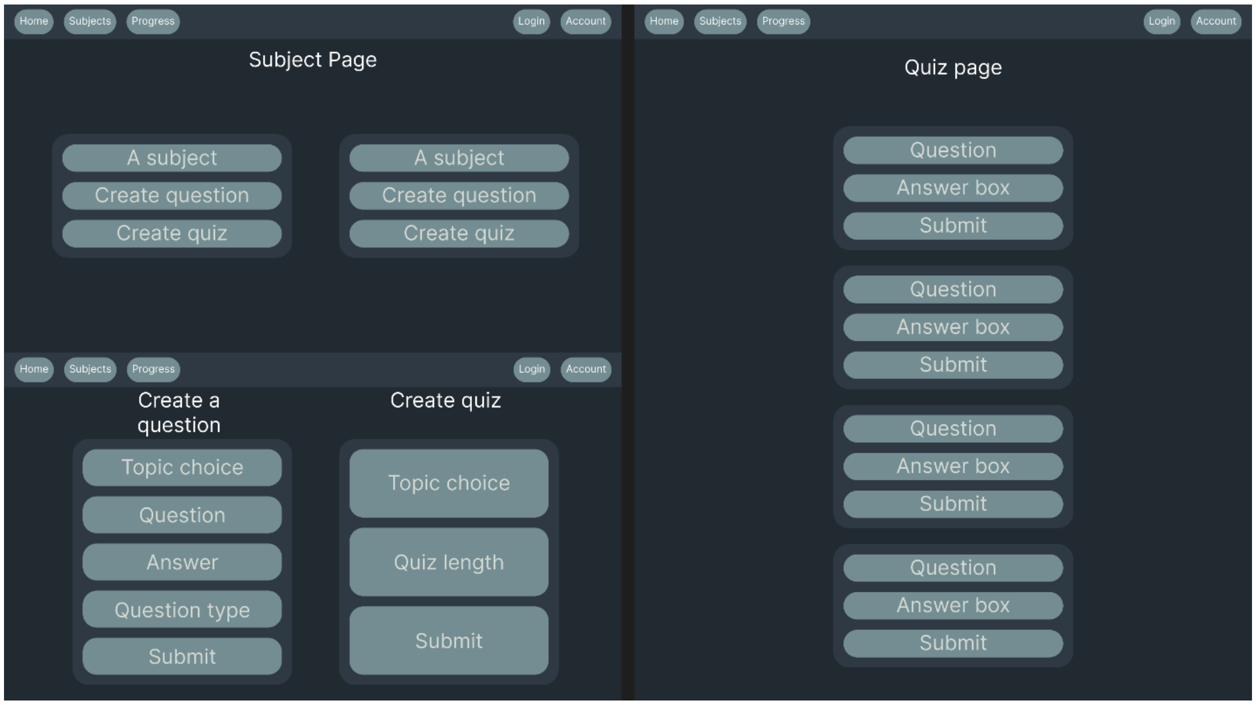
User progress page design

This page displays the user's topics in their respective topics, which can then be edited by means of a form where they select the topic and their confidence level. This design is based on the use of a table in order to layout the subjects' topics.



Subject page design

The subject page will have the biggest plan for its design because it has the most functionality of the pages on the website. From this page you can create custom questions; create custom quizzes; and do said quizzes along with checking your answers to the quiz. All of which would need to be laid out in a way that would make sense to the user and would not cause any issues with visualising how to get from A to B.



The two screens on the left show the initial page, and what the page will look like if the user chooses to do either of the options presented in the top left screen. When the user chooses to create a quiz, they will be shown the screen on the right, showing all of the questions (depending on how long they want the quiz to be). This design is simple as it is clear what the user can do, and how they can do it.

Algorithm design

This portion of the document design contains designs and prototypes for various complex algorithms to be used throughout the various aspects of my project. The formatting of the code has been edited in order to fit better on the page, for the full formatting and code, see the appendix, where there is the code of my project in its entirety, with proper formatting, indentation, and commenting to show what functions do. This section only contains four algorithms that I have planned; I have done this so as they are the key parts of my project that needs to be designed thoroughly for my code to run smoothly.

Registering system and login checks

These pairs of algorithms allow the user to create an account, and then to login to their account. The algorithms I have designed do not take into consideration the fact that any database handling will take place in a different file for ease; this is because it makes more sense to design an algorithm and then split it into what is necessary for the project.

```
def registering(username, password, email):
        if username == '' or password == '' or email == '':
                return # error handling
        elif re.fullmatch(r"[A-Za-z0-9]+", username == None or re.fullmatch(r"[a-z]+[A-Z]+[0-9]+[!-\/:-@[-`{-~]+", password
        or re.fullmatch(r"[-A-Za-z0-9!#$%&'*+/=?^_`{|}~]+(?:\.[-A-Za-z0-9!#$%&'*+/=?^_`{|}~]+)*@(?:[A-Za-z0-9](?:[-A-Za-
        z0-9]*[A-Za-z0-9])?\.)+[A-Za-z0-9](?:[-A-Za-z0-9]*[A-Za-z0-9])?", email:
                return # error handling
        exists = # search for pre-existing login details (with same username/password)
        If exists is empty:
                # hash password
                # add to database
                return # prompt to login to new account
        else:
                return # error handling
```

This is a rough version of the registering algorithm made to show how I will be checking the login details of the user. However, this check happens after the user has submitted the form (in the back-end), therefore I will also be doing a check in real time on the front-end using JavaScript so that the user can make sure the details are appropriate before they can submit it to the database. In this algorithm I am using regular expressions (using the regex package (13) in python), this is so that I can easily perform checks that the login details contain only correct characters and layouts. The username can contain alphanumerical characters; the password is the same but also includes special characters, while also requiring at least 1 uppercase, lowercase, number, and special; and the email, which requires any characters, followed by an '@' symbol, followed by and characters, followed by a '.' character, followed by any alphanumeric characters. The regular expression for the email was found using the RFC2822 Email formatting (14), this was used

```
def login_check():
        if login_oop.login_check(request.form.get('uname'), request.form.get('pword'), request.form.get('email')):
                session['username'], session['email'] = # username from form, # email from form
                session['userID'] = # get userID from database, using the login details
                return # send to landing page
        else:
                return # error handling
```

as it is a standard for email validation, and felt appropriate to use to allow for all emails to be used on the website.

This is the algorithm for when the user logs into an account, it will check that the account corresponding to the details given exists, and will then set the cookies for the relevant information. Both algorithms have error handling, where if the user enters details that are incorrect, insufficient, or already exist (for registration), they will be sent to a catch-all error page (see failures.html) and will be prompted to try again.

Creating a quiz or question

This is the most important algorithm in my project, as it is what allows the user to quiz themselves on topics. For this reason, it is the algorithm that was most thought out prior to writing the code for it.

```
def create_quiz(quiz_data):
    # get a list of questions
    questions = # 2D array, an array of each question in the topic
    chosen_questions = []
    if len(questions) >= int(quiz_data["quiz_length"]):
        for i in range(0, len(questions)):
            chosen_temp = random.choice((questions))
            questions.remove(chosen_temp)
            chosen_questions.append(chosen_temp)
        return chosen_questions[0:in(quiz_data["quiz_length"])]
    else:
        return # error handling
```

This algorithm makes use of many features, including a 2D array for storing all the questions concisely. The function takes a dictionary as a parameter, which contains a value for the length of the quiz, and the topic ID of the quiz. The first # in the code will be replaced with an SQLite statement that fetches an array of all the questions associated with the topic ID. The algorithm will then randomly sort the list, and select the first n in it, where n is the number of questions needed for the quiz. As seen below in the function below, the keywords are stored as an array converted into a string. I have chosen to not convert the keywords back into an array here because it would cause an issue when iterating through the question data later, and so the splitting of the string into an array will be handled on the front-end via JavaScript, which can be seen in the technical solution.

```
def create_question(question_data):
    keywords_list = re.findall(r"\*[A-Za-z0-9 ]+\*", question_data["answer"])
    keywords = "-".join(keywords)
    # add all data needed to database
```

The function that creates a question is much simpler, as the user will be prompted to enter all the details for it, and the function just needs to format it before inserting it into the database. For the keywords of the question, the user will be asked to surround the important words in their answer with *s, and then once passed through the function, will be converted into an array using regex to find all the keywords, and then finally converted into a string so that it can be stored in the table.

Technical Solution

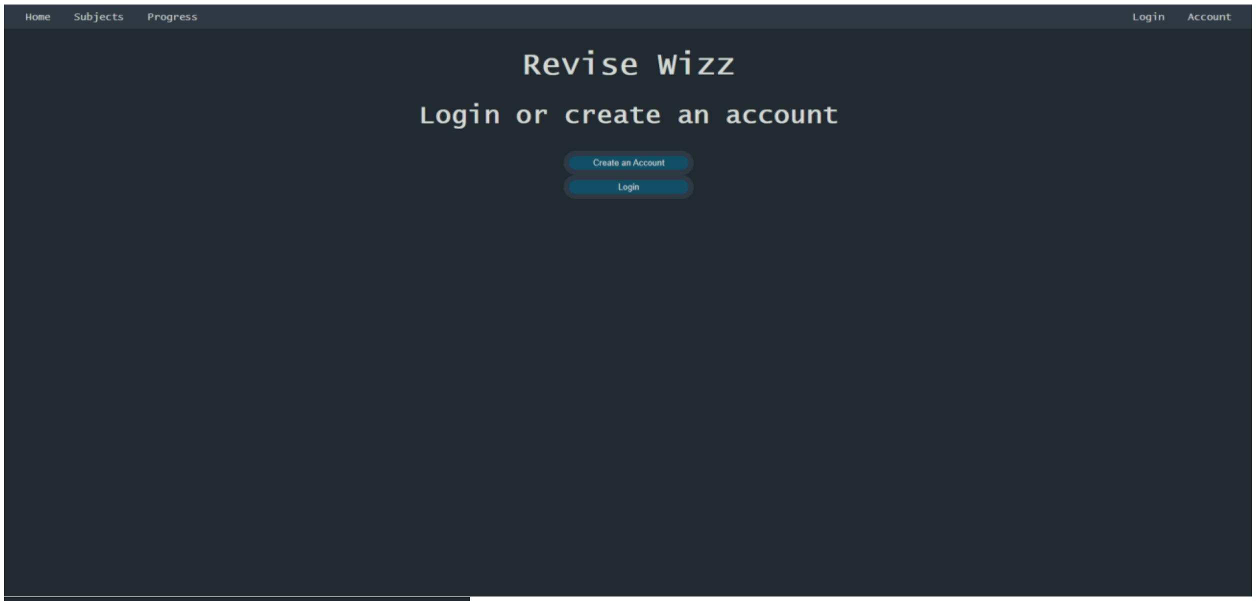
As mentioned previously, the full code of my project (including file names, comments, and proper formatting) can be found in the appendix of my project, with a contents page to assist in the navigation of my code. In this section of the report, I will be talking through various parts of my code: explaining how it works; what it does; and how it relates to other parts of the code. All sections talked about will have the relevant code and screenshots of the website underneath to help better show what the code is performing.

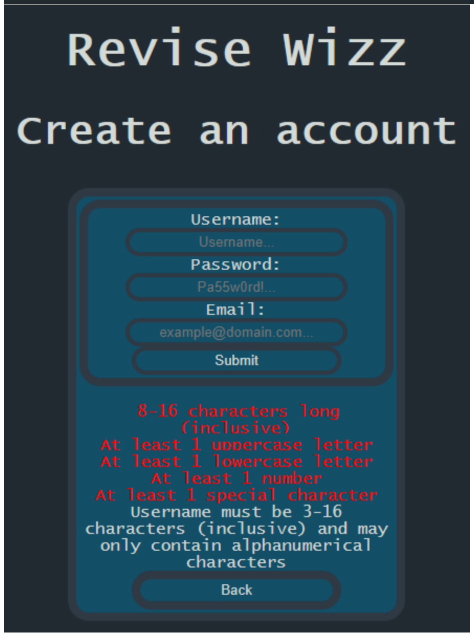
Project objectives fulfilled

I will start by going through the project objectives stated in the analysis section of the documentation and seeing how I met the requirements. For each section there will be the code and screenshots of the website; both of which will be explained as to how it works.

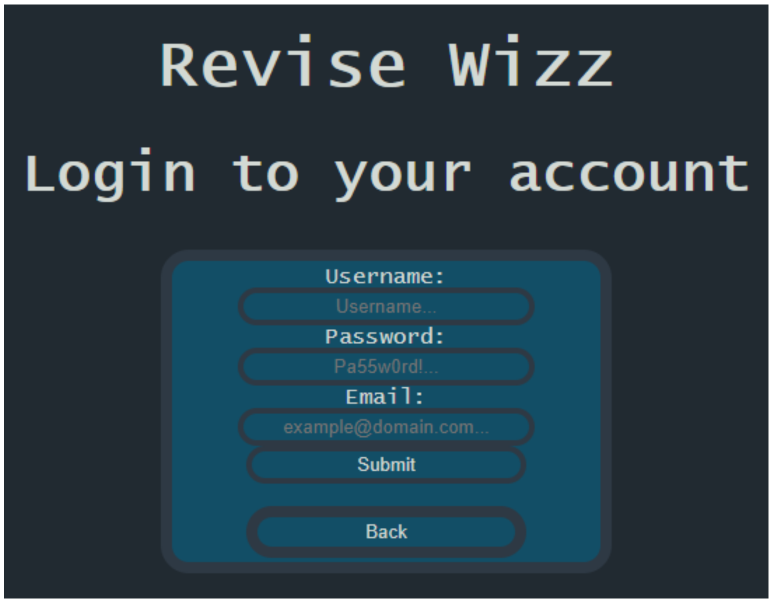
Objective 1

This objective was to create a working login system that allows the user to login and register an account. The testing for this can be found in tests 1-17 below. Here is a set of screenshots of the website, showing the initial page, as well as what happens once you click to either create an account or to login to an account that already exists:





8-16 characters long (inclusive)
At least 1 uppercase letter
At least 1 lowercase letter
At least 1 number
At least 1 special character
Username must be 3-16 characters (inclusive) and may only contain alphanumerical characters

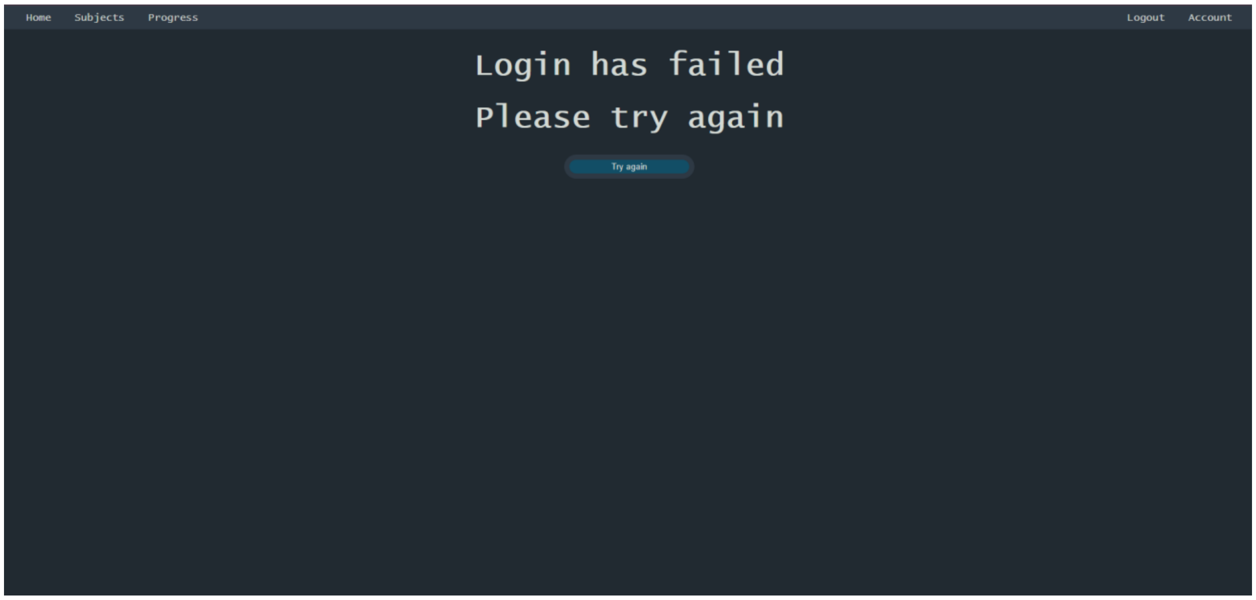


Both of these HTML forms call a function that will do either of the corresponding tasks (logging in or registering an account).

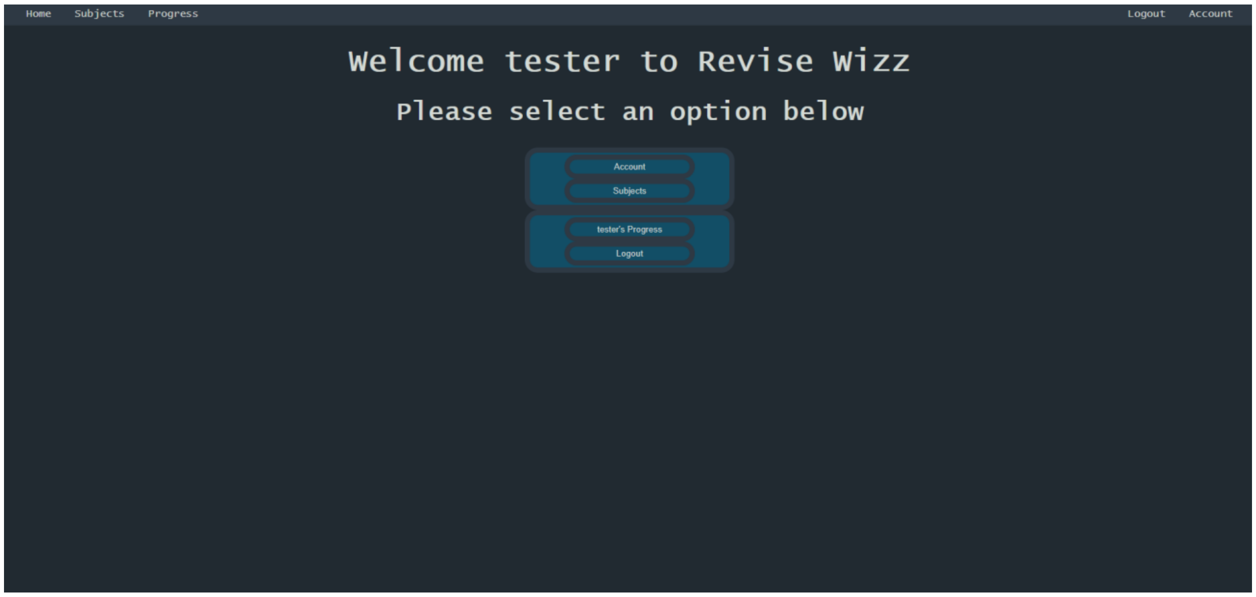
```
    # this is called when a user logs in, to make sure that the details they
entered are correct
    def login_check(self, user, pword, email):
        uname_check = self.cursor.execute("""SELECT username FROM users WHERE
username = ?""", (user,))
        uname_check = uname_check.fetchone()
        email_check = self.cursor.execute("""SELECT email FROM users WHERE
email = ?""", (email,))
        email_check = email_check.fetchone()
        if uname_check is None or email_check is None:
            return False
        else:
            pword_check = self.cursor.execute("""SELECT password FROM users
WHERE username = ?""", (user,))
            if self.ph.verify(pword_check.fetchone()[0], pword):
                return True
            else:
                return False
```

```
# this routine checks that the login details given match what is in the
datbase, more depth given in the comments of the subroutine called
@app.route('/login_check', methods=['POST','GET'])
def login_check():
    login_oop = UsersTable()
    if login_oop.login_check(request.form.get('uname'),
request.form.get('pword'), request.form.get('email')):
        session['username'], session['email'] = request.form.get('uname'),
request.form.get('email')
        session['userID'] = login_oop.userID_get(request.form.get('uname'),
request.form.get('pword'), request.form.get('email'))
        return redirect('/')
    else:
        return render_template("failures.html", type="login")
```

This is the code from the main file followed by the database handling file. The error handling in the first chunk of code calls the routine to check the login details, where each part of the login details is checked that they exist and are correct. Firstly, by checking that the user and email address are valid and are tied to the same account, then checking the password separately. The password checking uses the argon2 (15) .verify() method, which checks that a given string (in this case a password) could have been hashed into the password stored in the database. If either the first or second check fails, the function returns false, and the user is sent to the failure.html, which displays the following page:



From which the user can return to the login page. When the user successfully logs into their account, they will be redirected to the landing page of the website, which looks like this:



The user can then access the rest of the website, or logout, which is a simple function that just clears the cookie data and redirects back to the login page. The name 'tester' is one of the test accounts for the website, the site uses the cookies set when logging in to display the username as shown above.

The user can also select the registration form, which will check that username and email are unique, as well as checking the password meets certain requirements. The requirements are checked in the front-end via JavaScript, so that the user cannot enter a weak password. For ease of viewing, I have broken up the code on the front end for each part of the account details, starting with the initial setting up and username validation:

```
// creating setup for username, email, and password validation
document.getElementById("submit").disabled = true;
var email_valid = false;
var pword_valid = false;
var uname_valid = false;

// creating regex and input reading for username validation
var uname_input = document.getElementById('new_uname');
var uname_regex = /[A-Za-z0-9]+/g;

// username
uname_input.onkeyup = function() {
    var char_bool = false;
    var length_bool = false;

    if (uname_input.value.match(uname_regex)) {
        char_bool = true;
    } else {
        char_bool = false
    }
    if (uname_input.value.length > 2 && uname_input.value.length < 17) {
        length_bool = true;
    } else {
        length_bool = false;
    }
    if (char_bool == true && length_bool == true) {
        uname_valid = true;
    } else {
        uname_valid = false;
    }
    all_valid()
}
```

This section of code checks that the username only contains alphanumerical characters and is between 2 and 17 (exclusive). If both requirements are met, then uname_valid is set to true, and the code moves on. All 3 checks for the login are within a function that is called whenever the user types a character in the respective box, creating a dynamic checking system that will activate the submit button once all 3 checks are validated. The next check is the email checking function, which looks similar but uses a different pattern:

```
// Setting up variables for email validation (regex for email found using the
RFC2822 Email formatting)
var email_input = document.getElementById('new_email');
var email_regex = /[-!"£$%^&*(){}[\]#~'@;:?.>,<A-Za-z0-9]+[@][-
!"£$%^&*(){}[\]#~'@;:?.>,<A-Za-z0-9]+[.][a-z]+/g;

// Function for making sure email is in a valid format: "abc@def.ghi"
email_input.onkeyup = function() {
    if (email_input.value.match(email_regex)) {
        email_valid = true;
    } else {
        email_valid = false;
    }
    all_valid()
}
```

As stated in the comment at the begin of the code. The regular expression pattern for checking the email uses the RFC2822 (14) email formatting, which is a global standard for email formatting, hence why I chose to use it here. The email check is therefor a lot simpler than the other two, purely because it only has one requirement, which is the formatting given. Finally is the password validation, and the final if statement that checks all 3 checks have been validated: (written in 2 columns to take up less space)

```
// defining of vaiables for password
security checking
var pword_input =
document.getElementById('new_password
');
var pword_length =
document.getElementById('len_check');
var pword_lower =
document.getElementById('lower_check'
);
var pword_upper =
document.getElementById('upper_check'
);
var pword_number =
document.getElementById('num_check');
var pword_special =
document.getElementById('spec_check')
;

// defining all regex for password
validation
var lower_case_letters = /[a-z]/g;
var upper_case_letters = /[A-Z]/g;
var numbers = /[0-9]/g;
var special_characters = /[^\d\w]/g
```

```
// starts checking the user's inputs
pword_input.onkeyup = function() {
    var length_bool = false;
    var lower_bool = false;
    var upper_bool = false;
    var num_bool = false;
    var spec_bool = false;
    // length validation
    if (pword_input.value.length > 7
&& pword_input.value.length < 17) {
        pword_length.classList.remove
('invalid');
        pword_length.classList.add('v
alid');
        length_bool = true;
    } else {
        pword_length.classList.remove
('valid');
        pword_length.classList.add('i
nvalid');
        length_bool = false;
    }
    // lowercase validation
    if
(pword_input.value.match(lower_case_l
etters)) {
        pword_lower.classList.remove(
'invalid');
        pword_lower.classList.add('va
lid');
        lower_bool = true;
    } else {
        pword_lower.classList.remove(
'valid');
        pword_lower.classList.add('in
valid');
        lower_bool = false;
    }
    // uppercase validation
    if
(pword_input.value.match(upper_case_l
etters)) {
        pword_upper.classList.remove(
'invalid');
        pword_upper.classList.add('va
lid');
        upper_bool = true;
    } else {
        pword_upper.classList.remove(
'valid');
        pword_upper.classList.add('in
valid');
        upper_bool = false;
    }
    // number validation
    if
(pword_input.value.match(numbers)) {
        pword_number.classList.remove
('invalid');
        pword_number.classList.add('v
alid');
        num_bool = true;
    } else {
        pword_number.classList.remove
('valid');
        pword_number.classList.add('i
nvalid');
        num_bool = false;
    }
    // special character validation
    if
(pword_input.value.match(special_char
acters)) {
        pword_special.classList.remov
e('invalid');
        pword_special.classList.add('
valid');
        spec_bool = true;
    } else {
        pword_special.classList.remov
e('valid');
        pword_special.classList.add('
invalid');
        spec_bool = false;
    }
    if (length_bool == true &&
lower_bool == true && upper_bool ==
true && num_bool == true && spec_bool
== true) {
        pword_valid = true;
    } else {
        pword_valid = false;
    }
    all_valid()
}
```

This code is what checks the password in the following 4 ways: length between 8 and 16 (inclusive); at least 1 lowercase; at least 1 uppercase; at least 1 number; and at least 1 special character. Each of these (excluding length) has its own regular expression pattern that gets matched to the value of the input box. Each of the validation requirements have a corresponding label on the registration page that can have 1 of 2 CSS classes, 'invalid' makes the text red, and 'valid' makes the text green, this is so the user can clearly see what their password needs in order to be valid; to see this in action, see tests 6-11. Finally, there is the final function called all_valid() which checks that all 3 inputs for the registration are correct, then makes the submit box clickable for the user to add an account:

```
// function to check for a valid email and password before submission
function all_valid() {
    if (pword_valid == true && email_valid == true && uname_valid == true) {
        document.getElementById("submit").disabled = false;
    } else {
        document.getElementById("submit").disabled = true;
    }
}
```

This function simply checks that all three registration checks are true, then makes the submit box on the page true. Once the user has pressed the sumbit button, the back-end code runs as follows:

```
# this is similar to the routine above, but instead checking that the details
match all requirements
@app.route('/register', methods=['POST','GET'])
def registering():
    login_oop = UsersTable()
    if request.form.get('new_uname') == '' or request.form.get('new_pword') == ''
or request.form.get('new_email') == '':
        return render_template("failures.html", type="registering")
    elif re.fullmatch(r"[A-Za-z0-9]+", request.form.get('new_uname')) == None or
re.fullmatch(r"[a-z]+[A-Z]+[0-9]+[!-\/:-@[-`{-~]+", request.form.get('new_pword'))
or re.fullmatch(r"[-A-Za-z0-9!#$%&'*+/=?^_`{|}~]+(?:\.[-A-Za-z0-
9!#$%&'*+/=?^_`{|}~]+)*@(?:[A-Za-z0-9](?:[-A-Za-z0-9]*[A-Za-z0-9])?\.)+[A-Za-z0-
9](?:[-A-Za-z0-9]*[A-Za-z0-9])?", request.form.get('new_email')):
        return render_template("failures.html", type="registering")
    elif login_oop.create_login(request.form.get('new_uname'),
request.form.get('new_pword'), request.form.get('new_email')):
        return redirect('/login')
    else:
        return render_template("failures.html", type="registering")
```

The function started by doing another check of each input, to make sure that the details the user has inputted are definitely correct, then it calls the create_login() function, which looks like this:

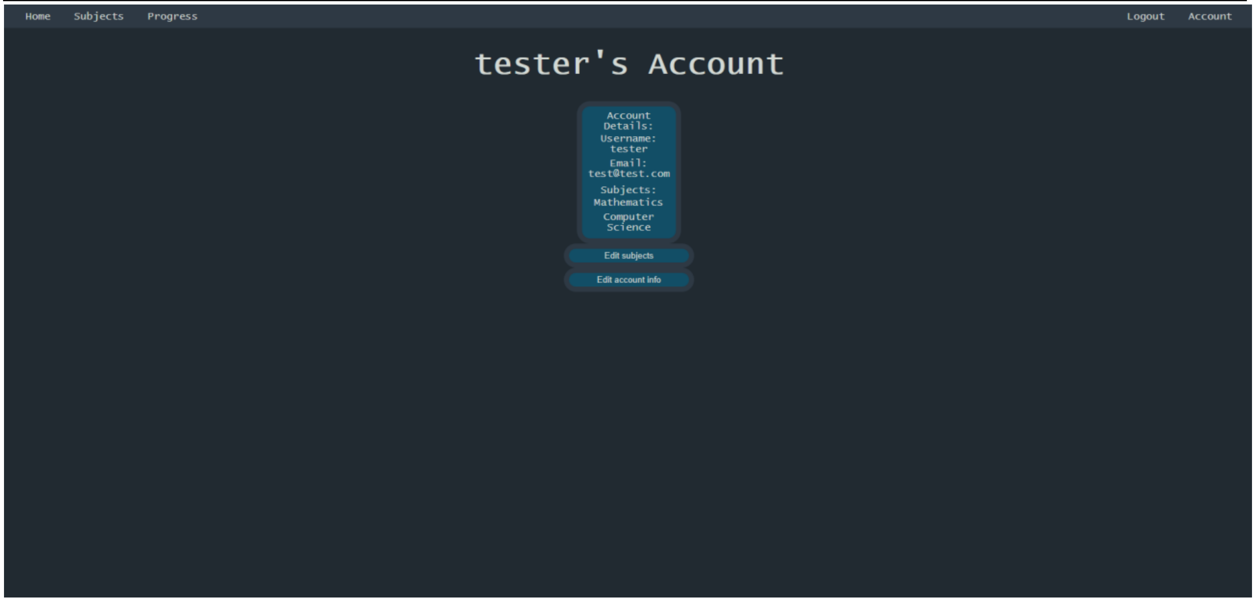
```
# this subroutine checks if a username and email already exists, and then adds
it if it doesn't
def create_login(self, user, pword, email):
   exists = self.cursor.execute("""SELECT username, email FROM users WHERE
username = ?""", (user,))
   if exists.fetchone() is None:
       hashed_pword = self.ph.hash(pword)
       self.cursor.execute("""INSERT INTO users(username, password, email)
VALUES(?, ?, ?)""", (user, hashed_pword, email,))
       self.connection.commit()
       return True
   else:
       return False
```

This function first checks that the username and email address are unique, to make sure that duplicate accounts do not exist. Then it will hash the password using argon2's hash() function and finally add all the necessary details to the database. This completes all of the sub-objectives in objective 1, done in a way that makes sense and works efficiently.

Objective 2

Objective 2 was based on the account page of the website, in which the user can view their details, and change them. This includes their personal details (username, email address, and password) and their subjects (in this project: maths and computer science). Here is the code behind the handling of the backend of the page, followed by what the page looks like:

```
# the account page displays all users personal details, with options to change
when needed
@app.route('/account', methods=['POST', 'GET'])
def account():
    if 'username' in session:
        username, email, userID = session['username'], session['email'],
session['userID']
        subjectsuser_oop, subject_oop = SubjectUserTable(), SubjectsTable() #
this is how the code and use subroutines in the 'databases.py' file, and are used
as and when needed throughout
        subjectIDs = subjectsuser_oop.subIDs_get_from_userID(userID)
        subjects = subject_oop.subIDs_to_sub(subjectIDs)
        subs = []
        for item in subjects:   # this is one of many methods used to convert the
tuple (returned from sqlite select statements) to arrays, html does not work well
with tuples and thus I made ways to change from tuples to arrays
            subs.append(item)
        return render_template('account_page.html', name=username, email=email,
subs=subs)
    else:
        return redirect('/login')
```



tester's Account

Account Details:
Username: tester
Email: test@test.com
Subjects: Mathematics
Computer Science

As shown in the picture, the app route gets the user data from the cookies set in the login function, and then gets the user's chosen subjects using 2 functions from the database file, which look like this:

```
    # this fetches the subject IDs tied to a specific user
    def subIDs_get_from_userID(self, userID):
        self.cursor.execute(f"""SELECT subjectID FROM usersubjects WHERE 
userID = ?""", (userID,))
        sub_id = self.cursor.fetchall()
        if sub_id == []:
            return ''
        else:
            return sub_id
```

```
    # this takes an array of subject IDs and 'converts' it to subject names
    def subIDs_to_sub(self, subject_ids):
        subjects = []
        for item in subject_ids:
            self.cursor.execute(f"""SELECT subjectName FROM subjects WHERE 
subjectID = ?""", (item,))
            subjects.append(self.cursor.fetchone()[0])
        return subjects
```

These functions run in turn with each other to get the subjects that a user is taking, firstly by taking a user's ID and getting a list of subject IDs, and then taking those IDs and converting them into their respective subject names. All the information is then stored on the page. On the page itself, the user can select one of two options: change their personal details; or change their subjects. Both options will open a form where the user can edit. The 'edit details' form uses the same validation as the registration form for obvious reasons but also adds the requirement of entering their current password, irrespective of the personal details they wish to change. This is to avoid an unwanted person can change the details of someone else's account. The app route called when the user changes their personal details is as follows:

```
# this is where the user can change their account information
@app.route('/account/info_change', methods=['POST','GET'])
def info_change():
    if 'username' in session:
        userID = session['userID']
        login_oop = UsersTable()    # this uses a dictionary to get each piece
of the data from the form
        new_information = {'new_uname': request.form.get('new_uname'),
                           'new_email': request.form.get('new_email'),
                           'new_pass': request.form.get('new_password')}
        updated_info = login_oop.change_details(userID,
request.form.get('old_password'), new_information)
        if updated_info == True:
            return redirect('/account')
        else:   # the failure page is a default page that changes based on the
'type' to suit where it got redirected from
            return render_template("failures.html", type="account")
    else:
        return redirect('/login')
```

This routine makes use of a dictionary to contain all the information that needs to be changed. The function change_details() is what validates and changes the data depending on what the user has chosen to change.

```
    # changing the users' details is performed here, by first verifying that the
old password entered is correct, then updating what is needed
    def change_details(self, userID, old_pass, new_details):
        try:
            self.ph.verify(self.cursor.execute("""SELECT password FROM users WHERE
userID=?""", (userID,)).fetchone()[0], old_pass)
            if new_details["new_pass"] != '':   # this changes the user's password
                new_hashed_pword = self.ph.hash(new_details["new_pass"])
                self.cursor.execute(f"""UPDATE users SET password=? WHERE
userID={userID}""", (new_hashed_pword,))
                self.connection.commit()
            else:
                pass
            if new_details["new_uname"] != '':   # this changes the username
                self.cursor.execute(f"""UPDATE users SET username=? WHERE
userID={userID}""", (new_details["new_uname"],))
                self.connection.commit()
            else:
                pass
            if new_details["new_email"] != '':   # this changes the user's email
                self.cursor.execute(f"""UPDATE users SET email=? WHERE
userID={userID}""", (new_details["new_email"],))
                self.connection.commit()
            else:
                pass
            return True
        except:
            return False
```

change_details() uses a try… except method of error handling, this is so that any error encountered by the database handling will immediately stop the execution and return false, leading to the user being redirected to the failure page. The code firstly uses the verify() function mentioned earlier to check that the current password is correct, then it goes through each item in the dictionary, and changes it if the value is not null.

The way in which the user can change their subjects is by selecting the subjects they would like to take via a series of checkboxes, and then the database will add said subjects to its information, as well as add the 'default progress' for the user, which is set to red as the user would likely not be confident on a subject by that point.

```
# this is the way in which a user changes their subject choices
@app.route('/account/subject_change', methods=['POST','GET'])
def subject_change():
    if 'username' in session:
        userID = session['userID']
        subchange_oop = SubjectUserTable()
        progress_oop = UserProgressTable()
        subjects_selected = []  # the following if statements separate the
subjects so that they can be used to add/remove the default progress of the
user
        if request.form.get('maths_check'):
            subjects_selected.append('1,true')
        else:
            subjects_selected.append('1,false')
        if request.form.get('comp_check'):
            subjects_selected.append('2,true')
        else:
            subjects_selected.append('2,false')
        subchange_oop.subject_change(userID, subjects_selected)
        for item in subjects_selected:
            split_item = item.split(',')
            if split_item[1] == 'false':
                progress_oop.del_progress(userID, split_item[0])
            else:
                progress_oop.add_progress(userID, split_item[0])    # this is
knowingly a closed-minded way of coding, as it does not allow for expansion of
other subjects
        return redirect('/account')
    else:
        return redirect('/login')
```

This app route takes each value of the check boxes and appends the subject ID and a true/false value depending on if it was ticked or not. The true/false value is then used to see if the database needs to add or remove the progress data for a given subject, those two functions look like this:

```
    # this adds all the default progress to a user, using both foreign keys
    def add_progress(self, userID, subjectID):
        topics = TopicsTable.subID_to_topicID(self, [subjectID])
        for i in range(0, len(topics)):
            self.cursor.execute("""INSERT INTO user_progress(userID, topicID, 
progress) VALUES(?, ?, 1)""", (userID, topics[i], ))
            self.connection.commit()
```

```
    # this removes all the progress for a given user's subject
    def del_progress(self, userID, subjectID):
        topics = TopicsTable.subID_to_topicID(self, [subjectID])
        for i in range(0, len(topics)):
            self.cursor.execute("""DELETE FROM user_progress WHERE userID=? 
AND topicID=?""", (userID, topics[i], ))
            self.connection.commit()
```

These pairs of functions simple go through the topics within a subject, and then either removes all the topics that are tied to a user ID, or adds the topics to the table with the corresponding user ID. The app route also will add or remove the subjects from the user subjects table, as seen in the function called:

```
    # this takes the user ID and array of subjects to add/remove
    def subject_change(self, userID, changing_subjectIDs):
        for item in changing_subjectIDs:
            sub_change = item.split(',')
            self.cursor.execute(f"""SELECT subjectID FROM usersubjects WHERE 
subjectID={sub_change[0]}""")
            sub_found = self.cursor.fetchall()
            if sub_change[1] == 'true' and sub_found == 
[]:                         #add subject to table
                self.cursor.execute("""INSERT INTO usersubjects 
VALUES(?,?)""", (userID, sub_change[0],))
            elif sub_change[1] == 'false' and sub_found == 
[int(sub_change[0])]:    #remove from table
                self.cursor.execute("""DELETE FROM usersubjects WHERE 
subjectID=? AND userId=?""", (sub_found[0], userID,))
            self.connection.commit()
```

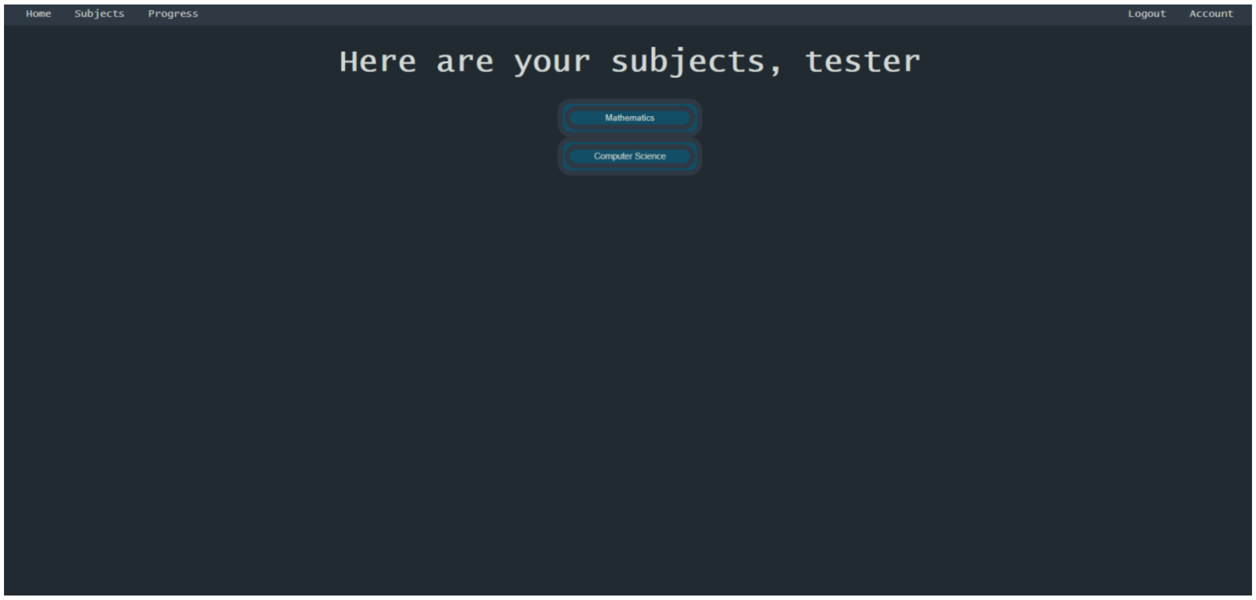
This subroutine will iterate through the array given, and then will take the subject ID, and the true/false value to either add or remove the row from the table. The user will then be taken back to the account page, now showing their newly selected subjects.

Objective 3

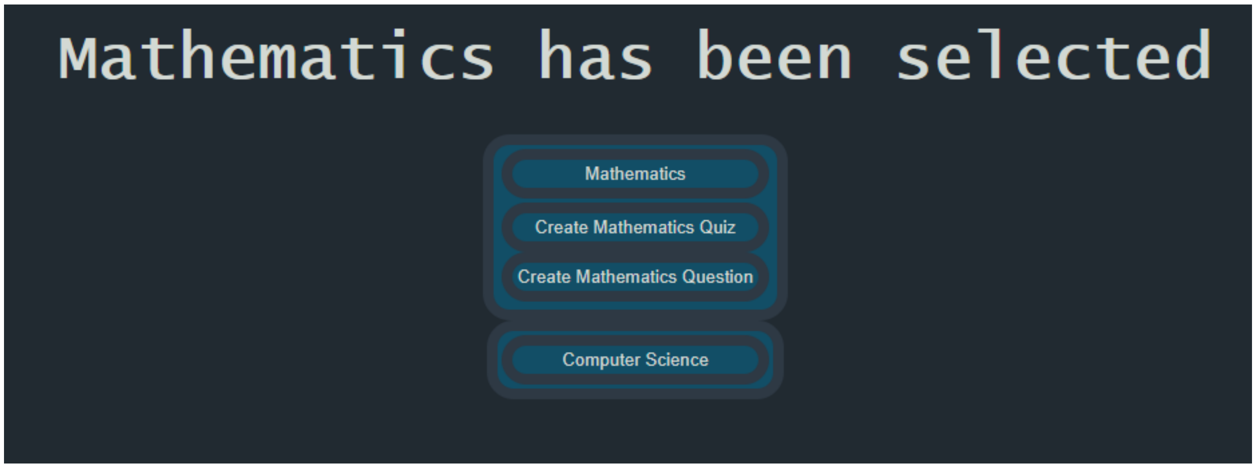
Objective three was the subject page and its ability to create questions and quizzes. This goal was the largest for the project and therefore has the most code behind it. The initial code for the webpage displays all subjects the user is taking and adds options to create a quiz and questions:

```
# this page displays the users' subjects, from which they can create questions
of quizzes
@app.route('/subjects', methods=['POST', 'GET'])
def subjects():
    if 'username' in session:
        username, userID = session['username'], session['userID']
        subjectsuser_oop, subject_oop, topics_oop = SubjectUserTable(),
SubjectsTable(), TopicsTable()
        subjectIDs = subjectsuser_oop.subIDs_get_from_userID(userID)
        subjects = subject_oop.subIDs_to_sub(subjectIDs)
        topics = topics_oop.get_topics(subjectIDs)
        subs = []
        for item in subjects:
            subs.append(item)
        topic_list = []
        for item in topics:
            topic_list.append(item)
        maths_topics = []
        comp_topics = []
        for item in topic_list:
            if item[0] == 1:
                maths_topics.append(item)
            else:
                comp_topics.append(item)
        return render_template('subjects.html', name=username, subs=subs,
maths=maths_topics, comps=comp_topics)
    else:
        return redirect('/login')
```

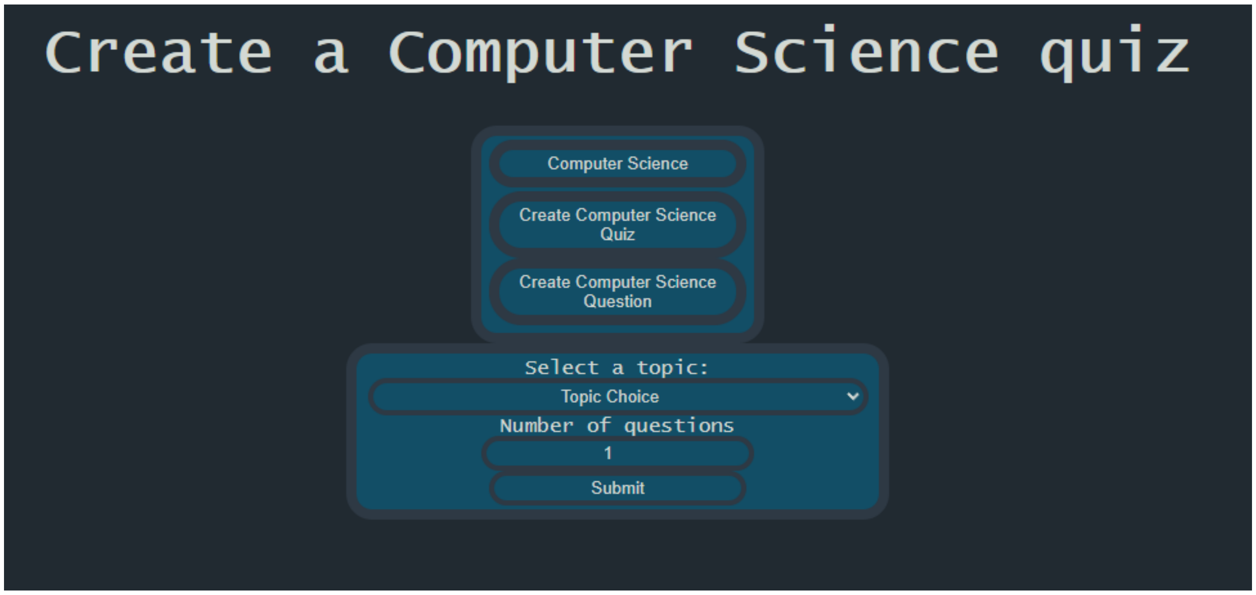
This code starts by getting the user ID, which will then call a function to get the associated subject IDs, which will then be converted into their respective subject names. The subject IDs will also be used to get a list of topics for each subject before loading the webpage, which looks like this:



And then when one of the subjects are clicked on, opens this drop down menu:



From here the user can choose either of the options to do what is says on the button. If the user chooses to create a quiz, the following form appears:


Create a Computer Science quiz
Computer Science
Create Computer Science Quiz
Create Computer Science Question
Select a topic:
Topic Choice
Number of questions
1
Submit

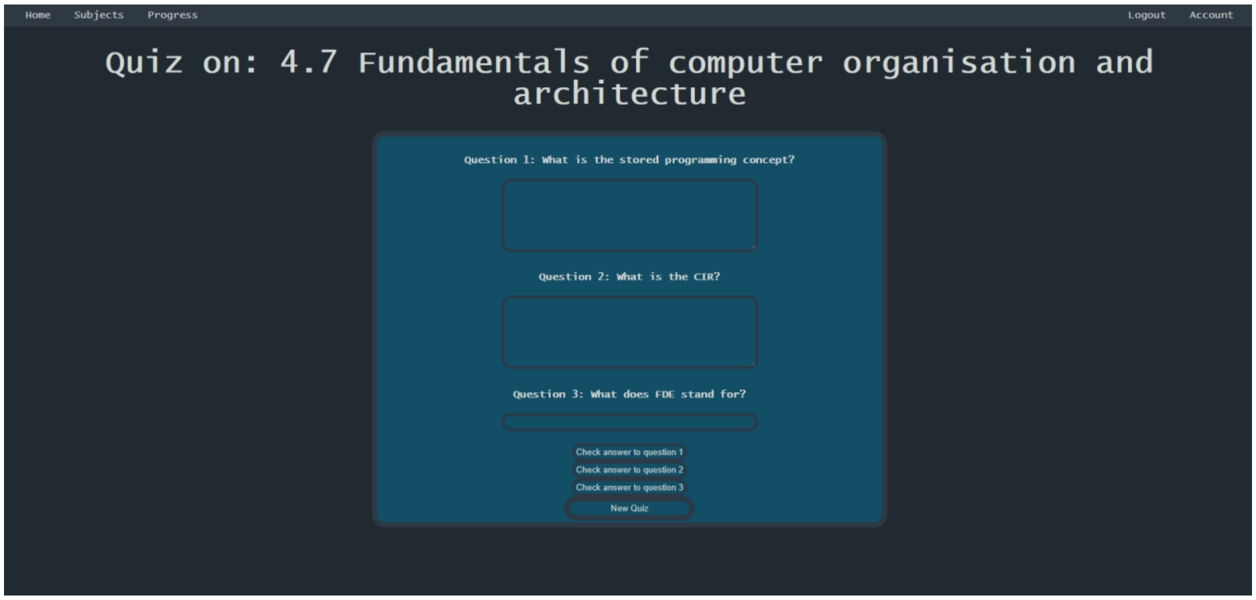
The form contains the list of topics for the given subject, and a number input that can take any integer from 1 to 20 (inclusive). When the user has selected a topic and length, the create_quiz app route is called, which does the following:

```
# creating a quiz requires all the data from a question to be used very
specifically in the html code
@app.route('/subjects/create_quiz', methods=['POST', 'GET'])
def create_quiz():
    if 'username' in session:
        questions_oop = QuestionsTable()
        quiz_data = {"topic": request.form.get('topic_choice'),
                     "quiz_length": request.form.get('quiz_length')}
        question_data = questions_oop.create_quiz(quiz_data)
        length = []
        for i in range(1, int(quiz_data["quiz_length"])+1):
            length.append(i)
        if question_data[0] == True:
            list_of_tuples = question_data[1]
            list_of_lists = []
            for item in list_of_tuples:
                list_of_lists.append(list(item))
            return render_template('quiz.html', topic=quiz_data['topic'],
length=length , questions=list_of_lists)
        else:
            return render_template("failures.html", type="quiz",
topic=quiz_data['topic'])
    else:
        return redirect('/login')
```

The function first takes the two pieces of information from the form and puts them into a dictionary, then the return value of create_quiz() (the one within the app route) is called which will fetch a series of questions, the details of which I will go into after I have explained the main app route function. The question_data is then iterated through to create a new 2D array, where each pair of data is in the format [[question number, question], [question number, question]] etc. the 2D array is then passed into the webpage, as well as the length of the quiz and the topic name. The database function called looks like this:

```
    # this returns fetches the questions for the quizzes, if it cannot find
enough or any questions, it returns false which leads to the failure page
    def create_quiz(self, quiz_data):
        topicId = TopicsTable.topic_to_topicID(self, [quiz_data["topic"]])
        self.cursor.execute(f"""SELECT question, answer, answer_keywords,
question_type FROM questions WHERE topicID='{topicId[0][0]}'""")
        questions = self.cursor.fetchall()
        questions = [item for item in questions]
        chosen_questions = []
        if len(questions) >= int(quiz_data["quiz_length"]):
            for i in range(0, len(questions)):  # this loop takes the
questions fetched for a given topic, and creates a 2D array of random
questions, depending on the quiz length provided
                chosen_temp = random.choice((questions))
                questions.remove(chosen_temp)
                chosen_questions.append(chosen_temp)
            return [True, chosen_questions[0:int(quiz_data["quiz_length"])]]
        else:
            return [False]
```

The function first fetches all the questions associated with the chosen topic, which is then converted from a tuple into an array. Then the code will check that there are at least as many questions as the length chosen, if not then it returns false and the user is sent to the failures page. If there are enough questions in the database, the list of questions is then randomised so that the user will not get the same question(s) each time the quiz for that topic is created. The function then returns true as well as the list of randomised questions (only the first n questions in the list, where n is the length of the quiz). The quiz page is then displayed, which for this example I will be creating a computer science quiz in topic 4.7 with a length of 3 questions:


Home
Subjects
Progress
Logout
Account
Check answer to question 1
Check answer to question 2
Check answer to question 3
New Quiz

Quiz on: 4.7 Fundamentals of computer organisation and architecture

Question 1: What is the stored programming concept?

Question 2: What is the CIR?

Question 3: What does FDE stand for?

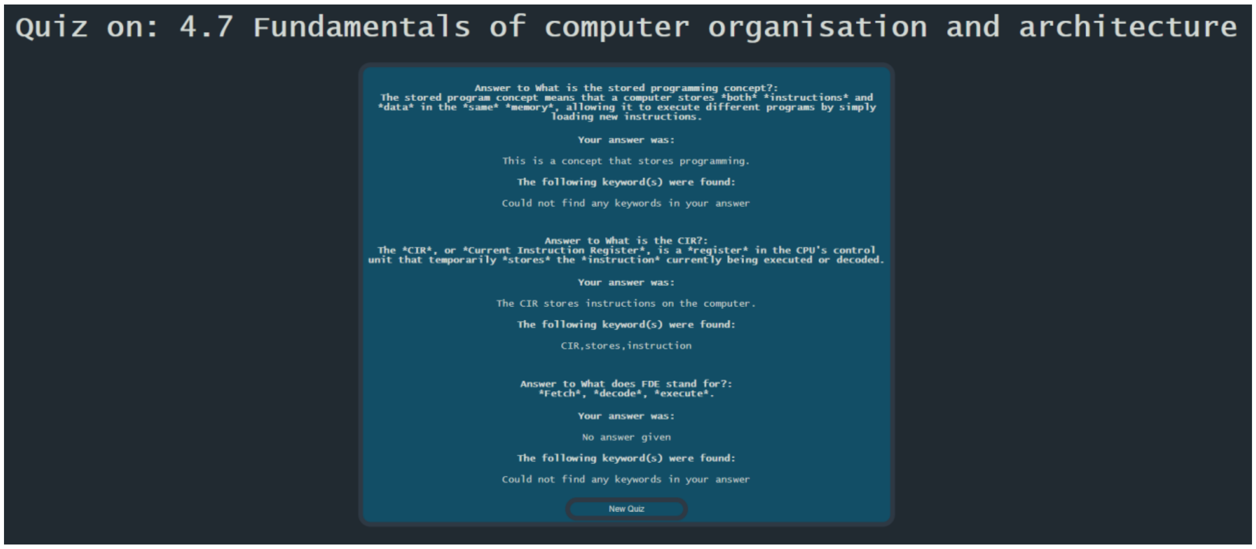
This page contains each question, along with an appropriate answer box (as there are long and short answers), followed by a button to check each answer to the quiz. I will show the html behind the quiz page as it uses iteration to display all of the elements:

```
    <body>
        <h1 style="text-align: center;">Quiz on: {{topic}}</h1>
        <!--
        this div contains the questions and answer boxes for the quiz
        a for loop is used to display each question
        -->
        <div class="parent" style="width: 40%;">
            {% for i in length %}
            <div class="child" name="questions" id="question_{{i}}">
                <h4 style="text-align: center; color: #d3d9d4;">Question
{{i}}: {{questions[i-1][0]}}</h4>
                {% if questions[i-1][3] == 'long_answer' %}
                    <textarea id="question_{{i}}_answer" style="width: 50%;"
required></textarea>
                {% else %}
                    <input type="text" id="question_{{i}}_answer"
style="width: 50%;" required>
                {% endif %}
            </div>
            {% endfor %}
            {% for i in length %}
            <!-- this div contains the answers, user's answers, and the button
to check the answer to a question -->
            <div class="child" name="answers" id="answer_{{i}}"
style="display: none;">
                <h4 style="text-align: center; color: #d3d9d4;">Answer to
{{questions[i-1][0]}}:<br> {{questions[i-1][1]}}</h4>
                <p><b>Your answer was:</b></p>
                <div class="child" id="question_{{i}}_answer_given"></div>
                <p><b>The following keyword(s) were found:</b></p>
                <div class="child" id="question_{{i}}_keywords"></div>
            </div>
            <br><input id="check_answers_{{i}}" type="submit"
onclick="check_questions(`{{i}}`, `{{questions[i-1][2]}}`)" style="cursor:
pointer; width: fit-content;" value="Check answer to question {{i}}">
            {% endfor %}
            <button id="end_quiz" style="display: none;"
onclick="window.location.href='/subjects'">Finish Quiz</button>
            <button id="new_quiz"
onclick="window.location.href='/subjects'">New Quiz</button>
        </div>
    </body>
```

This page contains for loops to iterate between each question and displays what is needed, it also makes use of an if statement to differentiate between a big and small answer box, so that the question can have the adequate room to respond to. When the user chooses to check an answer to a question, the will click on the corresponding button, and the following function is called is JavaScript:

```
// this large function handles the checking of a user's answer, from
displaying results to finding keywords
function check_questions(q_num, q_keywords) {
    document.getElementById('check_answers_'+q_num).style.display='none';
    document.getElementById('question_'+q_num).style.display='none';
    document.getElementById('answer_'+q_num).style.display='block';
    var answer = document.getElementById('question_'+q_num+'_answer').value;
    if (answer != '') { // if the user has written an answer, it will be
written in the div mentioned
        document.getElementById('question_'+q_num+'_answer_given').innerHTML=a
nswer;
    } else {    // if there is no answer, it will display this
        document.getElementById('question_'+q_num+'_answer_given').innerHTML='
No answer given';
    }
    keyword_list = string_to_array(q_keywords);
    found_keywords = [];
    for (i = 0; i < keyword_list.length; i++) {     // this finds all the
keywords in a user's answer
        let re = new RegExp(keyword_list[i], 'i');
        found = answer.match(re);
        if (found != null) {
            found_keywords.push(keyword_list[i]);
        } else {
            // pass
        }
    }
    found_keywords_string = array_to_string(found_keywords);
    if (found_keywords_string.length > 0) {     // if the user's answer
contains keywords, it will display them
        document.getElementById('question_'+q_num+'_keywords').innerHTML=found
_keywords_string;
    } else {    // if there are no keywords, it will display this
        document.getElementById('question_'+q_num+'_keywords').innerHTML='Coul
d not find any keywords in your answer';
    }
}
```

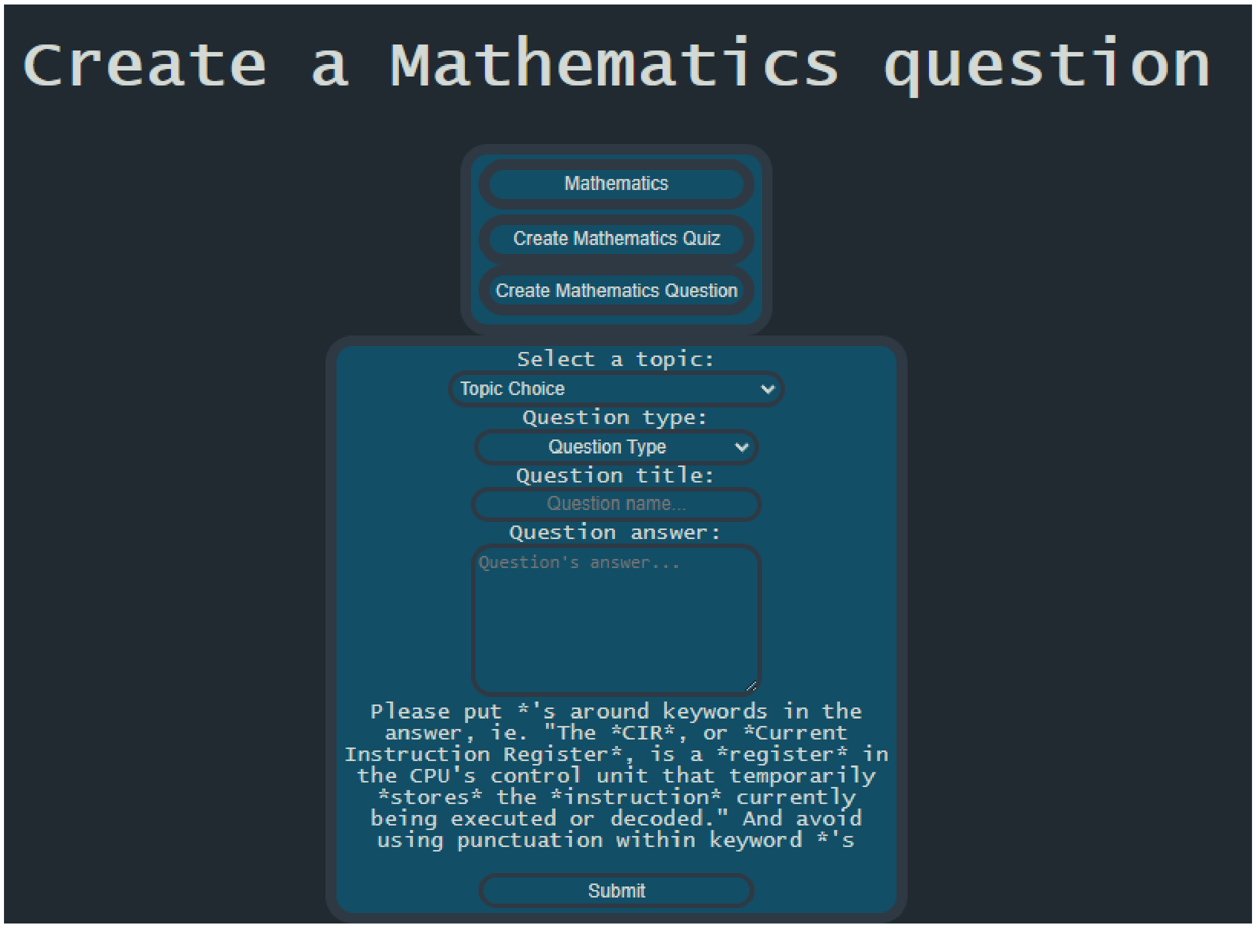
This is a very large function. It will start by removing the question and answer box from the user's view, as well as getting the value of what the user wrote in the answer box. It will then display the users answer (or show an appropriate message if they left the answer blanks). The function will then look at the answer given and use the keywords for the question to iterate through and use the .match() regular expression function to see if the user has written any of the keywords in their answer. string_to_array() and array_to_string() are not shown here as they simply use the .join() and .split() functions to change the keywords and found_keywords. Both of these functions can be found in the appendix in the quiz.js file. The function will finally display all the keywords found (or show a message if there were none found), and will show this on the webpage. For this example, I have given an answer to question 1 and 2, but left question 3 blank:



```
Quiz on: 4.7 Fundamentals of computer organisation and architecture

Answer to What is the stored programming concept?:
The stored program concept means that a computer stores *both* *instructions* and
*data* in the *same* *memory*, allowing it to execute different programs by simply
loading new instructions.

Your answer was:

This is a concept that stores programming.

The following keyword(s) were found:

Could not find any keywords in your answer


Answer to What is the CIR?:
The *CIR*, or *Current Instruction Register*, is a *register* in the CPU's control
unit that temporarily *stores* the *instruction* currently being executed or decoded.

Your answer was:

The CIR stores instructions on the computer.

The following keyword(s) were found:

CIR,stores,instruction


Answer to What does FDE stand for?:
*Fetch*, *decode*, *execute*.

Your answer was:

No answer given

The following keyword(s) were found:

Could not find any keywords in your answer
```

As you can see, the first question contained no keywords and as such said that there were not keywords in my answer, the second question contained some of the keywords as can be seen, and the final question has no answer so tells us that we left it blank. When the user clicks the 'new quiz' button, they are taken back to the subject page where they can do any of the options again.

When the user decides to create a question, they are presented with the following form:



Create a Mathematics question

Select a topic: Topic Choice

Question type: Question Type

Question title: Question name...

Question answer: Question's answer...

Please put *'s around keywords in the answer, ie. "The *CIR*, or *Current Instruction Register*, is a *register* in the CPU's control unit that temporarily *stores* the *instruction* currently being executed or decoded." And avoid using punctuation within keyword *'s

From this they can select the topic they would like to do, the question type (long or short answer), as well as the question itself and it's answer. The text beneath the answer prompt contains instructions to the user on how to define the keywords of the question. The explanation of why *'s are used will be told when I look at the code behind it, all of the information gets sent to the server to be handled. The question can contain no keywords if the user wishes, and this does not cause an issue as when a question is called later for a quiz, there will simply be no keywords to check and will always display as such when the answer is checked. The code that is called when the form is submitted is as follows:

```
# creating a question utilises a dictionary to parse in the data to be added
to the database
@app.route('/subjects/create_question', methods=['POST', 'GET'])
def create_question():
    if 'username' in session:
        questions_oop = QuestionsTable()
        question_data = {"topic": request.form.get('topic_choice'),
                         "type": request.form.get('question_type'),
                         "name": request.form.get('question_name'),
                         "answer": request.form.get('question_answer')}
        questions_oop.create_question(question_data)
        return redirect('/subjects')
    else:
        return redirect('/login')
```

This makes use of a dictionary to store all the question data, which allows for easy use in the database function. All inputs in the form are required and as such no error handling is necessary as the user simply cannot submit the form without giving all the information. The topic choice and question type boxes do start on an invalid option, but both of these options are disabled so they must be changed before the user can submit too. The database function called here is as follows:

```
    # this creates the list of keywords, then adds all the necessary data to
the questions table in the database
    def create_question(self, question_data):
        keywords_list = re.findall(r"\*[A-Za-z0-9 ]+\*",
question_data["answer"])
        keywords = self.construct_keywords(keywords_list)
        topicId = TopicsTable.topic_to_topicID(self,
[question_data["topic"]])
        self.cursor.execute("""INSERT INTO questions(topicID, question,
answer, answer_keywords, question_type) VALUES(?, ?, ?, ?, ?)""",
(topicId[0][0], question_data["name"], question_data["answer"], keywords,
question_data["type"], ))
        self.connection.commit()
```

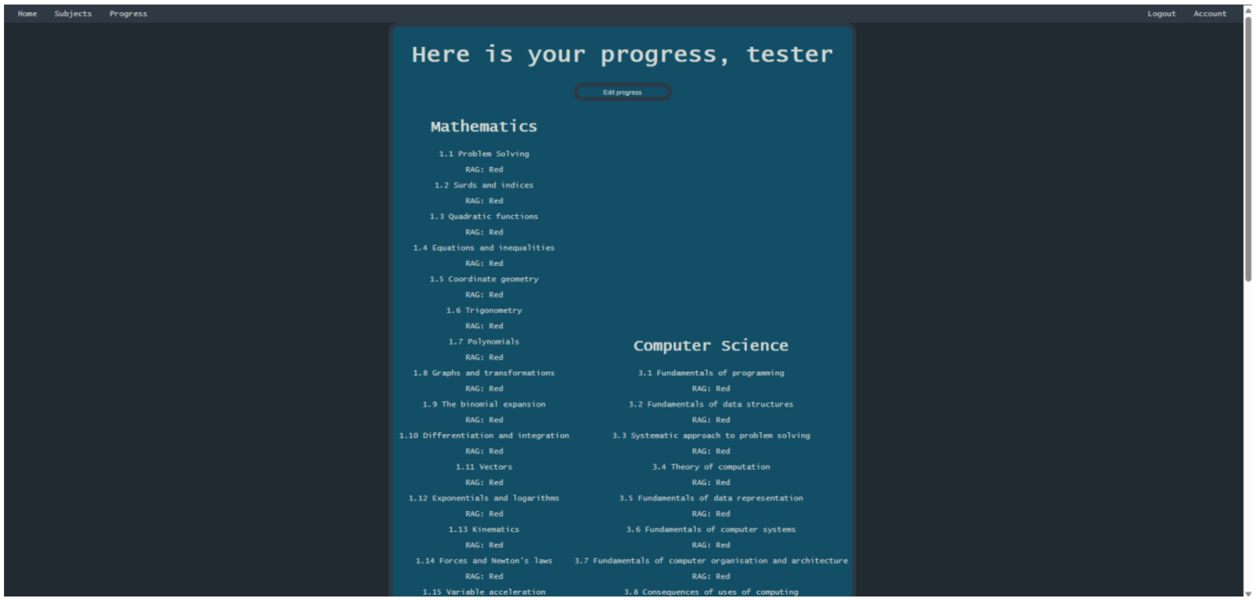
Like with the JavaScript function earlier, I have not included the construct_keywords() function as it is very simple and only joins up the keywords array into a string, this can be found in the database.py file in the appendix within the QuestionsTable class. The database method will find all of the keywords in the answer by using the regular expression function .findall() with the regex set to any alphanumeric characters or spaces surrounded by *'s. It will then get the topic ID for the topic and insert all the fields into the database. The user is then sent back to the default subject page, where they can create another question, create a quiz, or navigate to another area of the website.

Objective 4

The final aim was the user progress page, where the user can see and edit their confidence (RAG) levels for all of their topics in their chosen subjects. This is arguably the smallest of the objectives to add functionality to as it only required a display of progress and a form to change the progress. Firstly, here is the app route for the progress page:

```
# the progress page displays a users progress within a given topic, which they
can choose and change as they deem fit
@app.route('/progress', methods=['POST', 'GET'])
def progress():
    if 'username' in session:
        username, userID = session['username'], session['userID']
        subjectsuser_oop, subject_oop, topics_oop, progress_oop =
SubjectUserTable(), SubjectsTable(), TopicsTable(), UserProgressTable()
        subjectIDs = subjectsuser_oop.subIDs_get_from_userID(userID)
        subjects = subject_oop.subIDs_to_sub(subjectIDs)
        topics, rag_values = topics_oop.get_topics(subjectIDs),
progress_oop.get_progress(userID)
        subs = []
        for item in subjects:
            subs.append(item)
        topic_list = []
        for item in topics:
            topic_list.append(item)
        maths_topics = []
        comp_topics = []
        for i in range(len(topic_list)):
            if topic_list[i][0] == 1:
                maths_topics.append([topic_list[i][1], rag_values[i]])
            else:
                comp_topics.append([topic_list[i][1], rag_values[i]])
        return render_template('user_progress.html', name=username, subs=subs,
topics=topic_list, maths=maths_topics, comps=comp_topics)
    else:
        return redirect('/login')
```

This code appears to be doing a lot but it is simply getting all of the data required to display the page. It will first get the user ID, which is used to get the subject ID(s), which is used to get the subject name(s), which is used to get the topics, it will also get call the get_progress() function that will be shown shortly. It will then do three for loops to get the subject names, topic names, and then a final loop to get a list of the topics and its corresponding RAG value in a 2D array. It will then load the page showing all of the information. Here is a screenshot of the page, zoomed out slightly to show as much of it as possible, while still being readable:



Here is your progress, tester

Mathematics

1.1 Problem Solving
RAG: Red
1.2 Surds and indices
RAG: Red
1.3 Quadratic functions
RAG: Red
1.4 Equations and inequalities
RAG: Red
1.5 Coordinate geometry
RAG: Red
1.6 Trigonometry
RAG: Red
1.7 Polynomials
RAG: Red
1.8 Graphs and transformations
RAG: Red
1.9 The binomial expansion
RAG: Red
1.10 Differentiation and integration
RAG: Red
1.11 Vectors
RAG: Red
1.12 Exponentials and logarithms
RAG: Red
1.13 Kinematics
RAG: Red
1.14 Forces and Newton's laws
RAG: Red
1.15 Variable acceleration

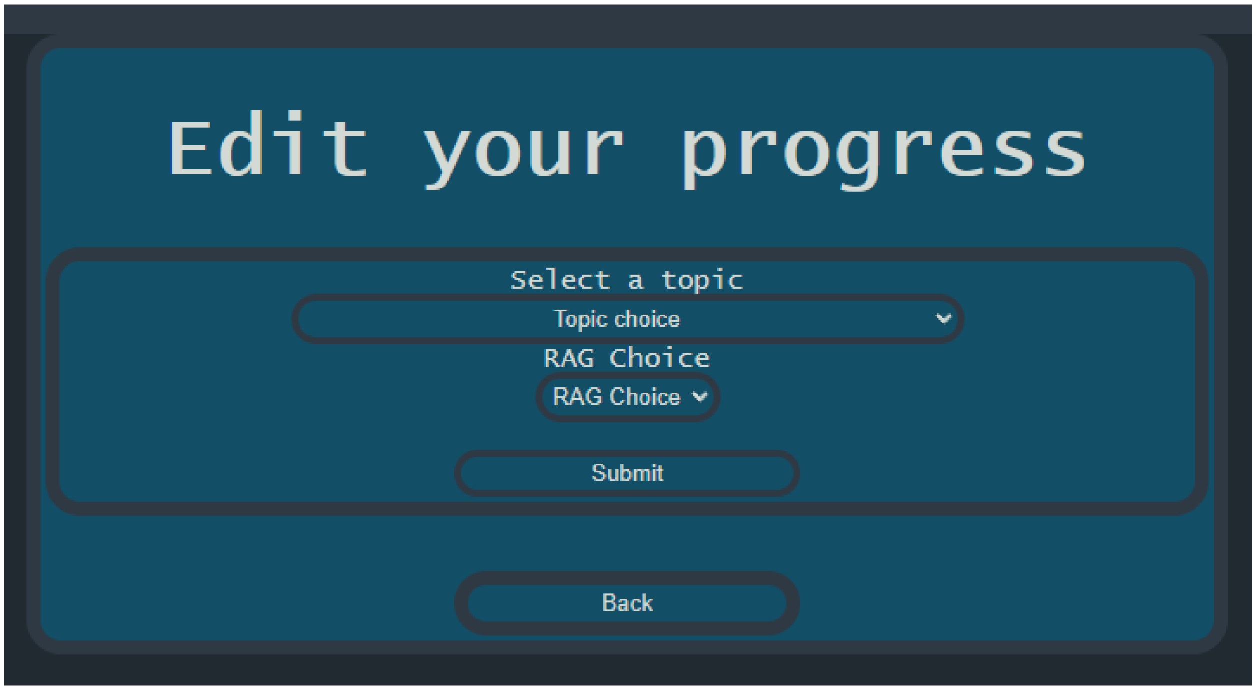
Computer Science

3.1 Fundamentals of programming
RAG: Red
3.2 Fundamentals of data structures
RAG: Red
3.3 Systematic approach to problem solving
RAG: Red
3.4 Theory of computation
RAG: Red
3.5 Fundamentals of data representation
RAG: Red
3.6 Fundamentals of computer systems
RAG: Red
3.7 Fundamentals of computer organisation and architecture
RAG: Red
3.8 Consequences of uses of computing

Here you can see that in this case the user is doing computer science and maths, both are shown here in each column with each topic and its RAG value underneath, the computer science column starts at the midpoint of the DIV it is in, I did spend a long time trying to get it to display at the top but as you can see I could not find a way to do it. As is well known, making elements display where you want them to be in HTML and CSS is a nightmare. The database function to fetch the progress looks like this:

```
    # this retrieves all the progress for a given user
    def get_progress(self, userID):
        self.cursor.execute("""SELECT progress FROM user_progress WHERE 
userID=?""", (userID, ))
        fetched_progress = self.cursor.fetchall()
        progress_values = self.progNum_to_prog(fetched_progress)
        return progress_values

    # this converts the stored RAG value (1, 2, or 3) to the associated name
    def progNum_to_prog(self, progress_number):
        progress_values = []
        for item in progress_number:
            if item == 1:
                progress_values.append("Red")
            elif item == 2:
                progress_values.append("Amber")
            else:
                progress_values.append("Green")
        return progress_values
```

As you can see, I have included two functions here, as the first calls the second. The first function will fetch all of the progress values associated with the user, and the second function will convert the value fetched (an integer value of 1, 2, or 3) and convert it into red, amber, or green. The page then displays all the information in a clear way to the user. If the user would like to change their progress level on a given topic, they can press the button at the top of the page which will display the following form:



From here they can select any of their topics, and the RAG value they would like to change it to, at which point the following app route and database function are called:

```
# this allows users to change their progress of a given topic, on a scale of
Red, Amber, Green
@app.route('/progress/change_progress', methods=['POST', 'GET'])
def change_progress():
    if 'username' in session:
        progress_oop = UserProgressTable()
        userID = session['userID']
        change_data = {"userID": userID,
                       "topic": request.form.get("topic_choice"),
                       "rag": request.form.get("rag_choice")}
        progress_oop.change_progress(change_data)
        return redirect('/progress')
    else:
        return redirect('/login')
```

This app route will add all the details for the database into a dictionary and call the database function to change the progress value.

```
    # this changes the RAG value of a topic for a user
    def change_progress(self, change_data):
        topicID = TopicsTable.topic_to_topicID(self, [change_data["topic"]])
        self.cursor.execute(f"""UPDATE user_progress SET progress=? WHERE
userID=? AND topicID=?""", (change_data['rag'], change_data['userID'],
topicID[0], ))
        self.connection.commit()
```

This function will convert the given topic into its topic ID and then update the table with the new RAG value, where the value in the form is already an integer or 1, 2, or 3. The user will then be redirected to the progress page, with the new RAG value being displayed.

This ends the technical solution section with all four project objectives shown to be complete and fully operational. Anything on the website that is not mentioned here will be tested in the following section.

Testing

The testing of my code can be found in this unlisted YouTube playlist, which has a video for each of the main tests listed below. For parts of the program that have multiple inputs, I will be looking at each input and using valid inputs for the rest of the data needed for the part of the program. For example, when assessing the user login system, I will test:

1. using an incorrect username with a correct password and email.
2. test an incorrect password with a correct username and email.
3. test an incorrect email with a correct password and correct username.

By doing this I can be certain that all individual parts of the code are working correctly as intended. Below I have a table of all the tests performed in a table, listing: test number; the test type; input data; expected outcome; and whether the test passed or failed. The table also contains references to project objective numbers, which were made in the analysis section of the documentation.

Proj. Obj.	**No.**	**Testing What?**	**Test Type**	**Test Input**	**Test Outcome**	**Pass /Fail**
1.2.3	1	Registering a username	Erroneous (length)	U aA2@aA2@ test@example.com	No submission	Pass
" "	2	" "	Erroneous (special characters)	Username! aA2@aA2@ test@example.com	Failure page	Pass
" "	3	" "	Normal	Username1 aA2@aA2@ test@example.com	Home page, added to database	Pass
" "	4	Registering an email	Erroneous (formatting)	Username2 aA2@aA2@ Notanemail.com	No submission	Pass
" "	5	" "	Normal	Username2 aA2@aA2@ test@gmail.com	Home page, added to database	Pass

1.2.1	6	Registering a password	Erroneous (length)	Username3 Pass testing@email.com	No submission	Pass
" "	7	" "	Erroneous (lowercase)	Username3 PASS123! testing@email.com	No submission	Pass
" "	8	" "	Erroneous (uppercase)	Username3 pass123! testing@email.com	No submission	Pass
" "	9	" "	Erroneous (numbers)	Username3 Password! testing@email.com	No submission	Pass
" "	10	" "	Erroneous (special characters)	Username3 Password123 testing@email.com	No submission	Pass
" "	11	" "	Normal	Password123!	Home page	pass
1.1	12	Login username	Erroneous	Test1	Failure page	Pass
" "	13	" "	Normal	tester	Home page	Pass
" "	14	Login password	Erroneous	NotCorrect	Failure page	Pass
" "	15	" "	Normal	1!qQ1!qQ	Home page	pass
" "	16	Login email	Erroneous	not@correct.co	Failure page	Pass
" "	17	" "	Normal	test@test.com	Home page	pass
2.1	18	Change subject details	Normal (none selected)	(untick all subject boxes)	Database cleared for user	pass
" "	19	" "	Normal (tick one box)	(tick one of the two boxes- test both)	Only one subject and progress added	pass

" "	20	" "	Normal (tick all boxes)	(tick both boxes)	Both subjects and progress added	pass
4.1	21	Editing RAG value	Normal	(select any topic and RAG value)	RAG changed in database	Pass
3.2	22	Creating a question	Absent	(leave both title and answer blank)	Does not allow submission	Pass
" "	23	" "	Normal	(see video, lots of data)	Submits and adds to database	pass
3.1	24	Creating a quiz	Erroneous	Select more questions than available	Failure page	Pass
" "	25	" "	Boundary	Select max questions available	Creates quiz with all questions	Pass
" "	26	" "	Normal	Select less than total questions available	Creates quiz with set no. of questions	pass
3.1.1	27	Answering a quiz	Absent	Write no answer	Both boxes say it is empty	pass
" "	28	" "	Erroneous	Write answer with no keywords	Answer shown, no keywords shown	Pass
" "	29	" "	Boundary	Write an answer with all keywords	Answer shown, all keywords shown	Pass
" "	30	" "	Normal	Write an answer with keywords	Answer shown,	pass

					keywords said shown	

As you can see from the table, there are some of the project objectives not being tested, here I will explain why they are omitted:

1. 1.2.2, the testing of a unique username and email address are checked at the same time as the other tests of 1.2.3 and thus could not be isolated to perform their own tests.
2. 2.1, the testing of changing account data does not have its own testing either. This is due to the code of the changing is using the same functions of both the checking login details and registering account (in terms of checking data) and therefore does not need to be tested separately.

Evaluation

Evaluating how well the requirements were met

This project had four overall goals set out in the analysis, and I think that what I have coded has adequately fulfilled these requirements. The login system works without any room for errors, allowing users to create an account that has a safe strong password; a unique username and email address; and allows them to login into the account with good error or incorrect data handling.

The second objective, which was an account page that allows users to change their details, also works very well. The user can change any of their personal details, both individually and at the same time; as well as edit their subjects, which lets them customise their experience on the website by selecting different subjects.

The third objective is where the bulk of the technical solution lies. This is where the user can create their own questions, and then use both their questions, and the questions of others to create their own quizzes, which can be self-evaluated using the keyword list that they defined when making a question. This solution is effective as it is highly customisable for the user and lets them choose how they want to test themselves; they can choose to do short rapid-fire quizzes, or do a quiz with many questions to assess how they can do in a long form test scenario.

The final objective was for the user to be able to set their progress within a given subject's topics, the RAG scale used is simple and allows the user to clearly show how well they think they are doing in a topic, and the method in which the user can change their confidence level is elegant and clear, making sure the user knows what they are doing and showing in real time the progress they are making in their subjects.

Given this, I would say that my solution meets the requirements well and shows that a unique solution is possible and can be done in a way that is helpful to students for their revision. The website acts in a 'cue-card-esque' way which lets the user control how they revise their topics and gives a customisable experience that has the qualities needed to make the website a great tool for subject practice.

Evaluation from a third party

In this section, I got a few third parties (some students) to take a look at the working project and give an evaluation of it. Here are the summaries of what some of them had said, the red text is what they had said, black is my response to it. I have also included references to the project objectives, to make it easier to understand which aspect of my project they are referring to:

I initially had a little problem registering (1.2) and logging into (1.1) an account, perhaps a password reset feature would be a useful addition to the project. Once I was logged in, I found it easy to create quizzes (3.1) and to go back and create additional revision questions (3.2).

I agree that the login and registration systems can seem vague as it does not tell you what the requirements are for the username and email address, if I were to amend the project, that would definitely be an area I would improve on.

I like that all students can create questions for a topic (3.2) which are then shared so my quizzes would then also contain questions which I hadn't written myself (3.1.1). I can see that the revision website would become more and more useful as more students make use of it.

I think that the ability for a user to be asked a question created by another user was a good idea, as it allows for people to be given unfamiliar questions from areas of the specifications that they may have forgotten or not asked a certain question of. It is definitely a feature that would only really be effective once many users have made their imprint on the website, so it may have been useful to incorporate some 'default' questions pre-packaged into the website.

The website is able to tell if my answers contain the right keywords for the question (3.1.1), it would be good if this automatically gave me a score which was then saved against my progress rather than relying on myself to rank my progress.

Given the time constraints and my own ability to code in the website languages (HTML, CSS, JS) being relatively new, the ability for an updating score as well as saving test scores in the database would have been a great idea to include, but could not have been done due to the limitations. If I had more time to acquaint myself with the languages, it very well could have been in the final product and have been a good feature for the user to have.

The onscreen instructions for creating a new question (3.2) were straightforward to follow and it was also useful to have error messages when I tried to create a quiz with more questions that existed (3.1) – perhaps in future it could tell you in advance what the pool of questions for each topic is.

The error messages were a rather late edition the website, which came as a result of it being unhelpful for the user being redirected to the main page whenever something went wrong (i.e. being sent to the login page if the registration failed or succeeded), so I think their addition made sense for the user experience. The ability to know if something will succeed in advance is a good idea, and it happens in the registration process when typing in a password, so the idea of seeing the number of questions for a topic does not seem very farfetched as a worthwhile inclusion. While I did not have it in my project, the inclusion of this could have made the HCI better for the user, by giving them a warning of how large a quiz they can make for a given topic.

Appendices

/static

/static/css

stylings.css

```
header {
    position: sticky;
    top: 0;
    padding: 10px 16px;
    background-color: #2e3944;
    color: #5a635b;
}
body {
    background-color: #212a31;
    margin: 0px;
    font-family: "Lucida Console", monospace !important;
}
h1 {
    color: #d3d9d4;
    font-size: 300%;
}
h2 {
    color: #d3d9d4;
    font-size: 250%;
}
h3 {
    color: #d3d9d4;
    font-size: 200%;
}
form {
    color: #d3d9d4;
    background-color: #124e66;
    padding: 3px;
    text-align: center;
    border: 8px solid #2e3944;
    width: 300px;
    margin: auto;
    border-radius: 20px;
}
form.register {
    color: #d3d9d4;
    text-align: center;
    width: 300px;
    box-sizing: border-box;
}
form.account_form {
    color: #d3d9d4;
    text-align: center;
    width: 300px;
}
button {
    display: block;
    margin: auto;
    cursor: pointer;
    color: #d3d9d4;
    background-color: #124e66;
    padding: 3px;
    text-align: center;
    border: 8px solid #2e3944;
    width: 200px;
    transition: 0.5s;
    border-radius: 25px;
}
button:hover {
    transform: scale(1.1)
}
input {
    border: 2px;
    padding: 3px;
    cursor: text;
    background-color:ghostwhite;
}
ul {
    list-style-type: none;
    margin: 0;
    padding: 0;
    overflow: hidden;
    background-color: #2e3944;
}
li {
    float: left;
}
li a {
    color: #d3d9d4;
    text-align: center;
    padding: 12px 18px;
    text-decoration: none;
  }
li a:hover {
    background-color: #748d92;
}
label.invalid {
    color: red;
    background-color: #124e66;
    padding: 3px;
    align-items: center;
    border-radius: 20px;
    margin: auto;
}
label.valid {
    color: yellowgreen;
    background-color: #124e66;
    padding: 3px;
    align-items: center;
    border-radius: 20px;
    margin: auto;
}
div {
    color: #d3d9d4;
    background-color: #124e66;
    padding: 3px;
    text-align: center;
    border: 8px solid #2e3944;
    width: 300px;
    margin: auto;
    border-radius: 20px;
}
div div {
    border: 8px !important;
}
.parent {
    flex-direction: row;
    width: min-content;
    height: min-content;
    gap: 4px;

}
.child {
    flex-direction: column;
    width: auto;
}
select {
    margin: auto;
    cursor: pointer;
    color: #d3d9d4;
    background-color: #124e66;
    padding: 2px;
    text-align: center;
    align-items: center;
    border: 4px solid #2e3944;
    transition: 0.5s;
    border-radius: 25px;
}
input {
    margin: auto;
    cursor: text;
    color: #d3d9d4;
    background-color: #124e66;
    padding: 2px;
    text-align: center;
    border: 4px solid #2e3944;
    width: 200px;
    border-radius: 25px;
}
textarea {
    margin: auto;
    cursor: text;
    color: #d3d9d4;
    background-color: #124e66;
    padding: 2px;
    border: 4px solid #2e3944;
    width: 200px;
    height: 100px;
    border-radius: 15px;
    resize: vertical;
```

```
}
```

/static/images

favicon.ico

/static/js

account_funcs.js

```
    // functions for the account page

    // this function displays the form to edit a user's subjects
    function edit_subjects() {
        document.getElementById('sub_edit').style.display='none'
        document.getElementById('info_edit').style.display='none'
        document.getElementById('account_info').style.display='none'
        document.getElementById('edit_sub_form').style.display='block'
    }

    // this function displays the form to edit a user's details
    function edit_info() {
       document.getElementById('info_edit').style.display='none'
       document.getElementById('sub_edit').style.display='none'
       document.getElementById('account_info').style.display='none'
       document.getElementById('edit_info_div').style.display='block'
    }
```

login_funcs.js

```
// functions for the login page

// function to display account login form on button press
function login_account() {
    document.getElementById('create_acc').style.display='none'
    document.getElementById('login_acc').style.display='none'
    document.getElementById('login_form').style.display='block'
    document.getElementById('choice_text').innerHTML='Login to your account'
}
// function to display create account form on button press
function create_account() {
    document.getElementById('create_acc').style.display='none'
    document.getElementById('login_acc').style.display='none'
    document.getElementById('create_form').style.display='block'
    document.getElementById('choice_text').innerHTML='Create an account'
}
```

```
// creating setup for username, email, and password validation
document.getElementById("submit").disabled = true;
var email_valid = false;
var pword_valid = false;
var uname_valid = false;

// creating regex and input reading for username validation
var uname_input = document.getElementById('new_uname');
var uname_regex = /[A-Za-z0-9]+/g;

// username
uname_input.onkeyup = function() {
    var char_bool = false;
    var length_bool = false;

    if (uname_input.value.match(uname_regex)) {
        char_bool = true;
    } else {
        char_bool = false
    }
    if (uname_input.value.length > 2 && uname_input.value.length < 17) {
        length_bool = true;
    } else {
        length_bool = false;
    }
    if (char_bool == true && length_bool == true) {
        uname_valid = true;
    } else {
        uname_valid = false;
    }
    all_valid()
}
// defining of vaiables for password security checking
var pword_input = document.getElementById('new_password');
var pword_length = document.getElementById('len_check');
var pword_lower = document.getElementById('lower_check');
var pword_upper = document.getElementById('upper_check');
var pword_number = document.getElementById('num_check');
var pword_special = document.getElementById('spec_check');

// defining all regex for password validation
var lower_case_letters = /[a-z]/g;
var upper_case_letters = /[A-Z]/g;
var numbers = /[0-9]/g;
var special_characters = /[^\d\w]/g

// starts checking the user's inputs
pword_input.onkeyup = function() {
    var length_bool = false;
    var lower_bool = false;
    var upper_bool = false;
    var num_bool = false;
    var spec_bool = false;
    // length validation
    if (pword_input.value.length > 7 && pword_input.value.length < 17) {
        pword_length.classList.remove('invalid');
        pword_length.classList.add('valid');
        length_bool = true;
    } else {
        pword_length.classList.remove('valid');
        pword_length.classList.add('invalid');
        length_bool = false;
    }
    // lowercase validation
    if (pword_input.value.match(lower_case_letters)) {
        pword_lower.classList.remove('invalid');
        pword_lower.classList.add('valid');
        lower_bool = true;
    } else {
        pword_lower.classList.remove('valid');
        pword_lower.classList.add('invalid');
        lower_bool = false;
    }
    // uppercase validation
    if (pword_input.value.match(upper_case_letters)) {
        pword_upper.classList.remove('invalid');
        pword_upper.classList.add('valid');
        upper_bool = true;
    } else {
        pword_upper.classList.remove('valid');
        pword_upper.classList.add('invalid');
        upper_bool = false;
    }
    // number validation
    if (pword_input.value.match(numbers)) {
        pword_number.classList.remove('invalid');
        pword_number.classList.add('valid');
        num_bool = true;
    } else {
        pword_number.classList.remove('valid');
        pword_number.classList.add('invalid');
        num_bool = false;
    }
    // special character validation
    if (pword_input.value.match(special_characters)) {
        pword_special.classList.remove('invalid');
        pword_special.classList.add('valid');
        spec_bool = true;
    } else {
        pword_special.classList.remove('valid');
        pword_special.classList.add('invalid');
        spec_bool = false;
    }
    if (length_bool == true && lower_bool == true && upper_bool == true &&
num_bool == true && spec_bool == true) {
        pword_valid = true;
    } else {
        pword_valid = false;
    }
    all_valid()
}
```

```
// Setting up variables for email validation (regex for email found using the
RFC2822 Email formatting)
var email_input = document.getElementById('new_email');
var email_regex = /[-!"£$%^&*(){}[\]#~'@;:?.>,<A-Za-z0-9]+[@][-
!"£$%^&*(){}[\]#~'@;:?.>,<A-Za-z0-9]+[.][a-z]+/g;

// Function for making sure email is in a valid format: "abc@def.ghi"
email_input.onkeyup = function() {
    if (email_input.value.match(email_regex)) {
        email_valid = true;
    } else {
        email_valid = false;
    }
    all_valid()
}

// function to check for a valid email and password before submition
function all_valid() {
    if (pword_valid == true && email_valid == true && uname_valid == true) {
        document.getElementById("submit").disabled = false;
    } else {
        document.getElementById("submit").disabled = true;
    }
}
```

progress_funcs.js

```
// functions for the account page

// this function displays the form to edit the progress of a topic
function edit_progress() {
    document.getElementById('page_text').innerHTML="Edit your progress"
    document.getElementById('main_body').style.display='none'
    document.getElementById('edit_body').style.display='block'
}
```

quiz_funcs.js

```
// functions for the quiz page

// this large function handles the checking of a user's answer, from
displaying results to finding keywords
function check_questions(q_num, q_keywords) {
    document.getElementById('check_answers_'+q_num).style.display='none';
    document.getElementById('question_'+q_num).style.display='none';
    document.getElementById('answer_'+q_num).style.display='block';
    var answer = document.getElementById('question_'+q_num+'_answer').value;
    if (answer != '') { // if the user has written an answer, it will be
written in the div mentioned

document.getElementById('question_'+q_num+'_answer_given').innerHTML=answer;
    } else {    // if there is no answer, it will display this

document.getElementById('question_'+q_num+'_answer_given').innerHTML='No
answer given';
    }
    keyword_list = string_to_array(q_keywords);
    found_keywords = [];
    for (i = 0; i < keyword_list.length; i++) {     // this finds all the
keywords in a user's answer
        let re = new RegExp(keyword_list[i], 'i');
        found = answer.match(re);
        if (found != null) {
            found_keywords.push(keyword_list[i]);
        } else {
            // pass
        }
    }
    found_keywords_string = array_to_string(found_keywords);
    if (found_keywords_string.length > 0) {     // if the user's answer
contains keywords, it will display them

document.getElementById('question_'+q_num+'_keywords').innerHTML=found_keywor
ds_string;
    } else {    // if there are no keywords, it will display this

document.getElementById('question_'+q_num+'_keywords').innerHTML='Could not
find any keywords in your answer';
    }
}

// function to split a string into an array
function string_to_array(string) {
    string = string.replace(/\*/g, '');
    const array = string.split('-');
    return array
}

// function to convert an array of data into a concatenated string
function array_to_string(array) {
    string = array.toString();
    return string
}
```

subject_funcs.js

```
// functions for the subject page

// this function displays the options for a subject (create quiz or question)
and hides the other subjects
function display_subject_options(subject, user) {
    if (document.getElementById(subject+'_options').style.display=='none') {
        document.getElementById(subject+'_options').style.display='block'
        document.getElementById('welcome_text').innerHTML=(subject + ' has
been selected')
        var forms = document.querySelectorAll('form')
        for (let i = 0; i < forms.length; i++) {
            forms[i].style.display='none'
        }
    }
    else {
        document.getElementById(subject+'_options').style.display='none'
        document.getElementById('welcome_text').innerHTML=('Here are your
subjects, '+user)
```

```
        var forms = document.querySelectorAll('form')
        for (let i = 0; i < forms.length; i++) {
            forms[i].style.display='none'
        }
    }
}

// this function displays the form to create a quiz for a given subject
function create_quiz(subject) {
    document.getElementById('available_subjects').style.display='none'
    document.getElementById('welcome_text').innerHTML='Create a '+subject+'
quiz'

document.getElementById('create_'+subject+'_quiz_form').style.display='block'
}

// this function displays the form to create a question for a given topic
function create_question(subject) {
    document.getElementById('welcome_text').innerHTML='Create a '+subject+'
question'
    document.getElementById('create_'+subject+'_form').style.display='block'
}
```

/static/templates

account_page.html

```
<!DOCTYPE html>
<html>

    <head>
        <title>Account</title>
        <meta name="viewport" content="width=device-width, initial-
scale=1.0">
        <link rel="icon" type="image/x-icon"
href="/static/images/favicon.ico">
        <link rel="stylesheet" type="text/css"
href="/static/css/stylings.css">
     </head>

     <header>
        <ul>
           <li><a href="/">Home</a></li>
           <li><a href="/subjects">Subjects</a></li>
           <li><a href="/progress">Progress</a></li>
           <li style="float:right"><a href="/account">Account</a></li>
           <li style="float:right"><a href="/logout">Logout</a></li>
        </ul>
     </header>

     <body>
      <h1 style="text-align: center;">{{name}}'s Account</h1>
      <!-- this is the info panel for the user's information -->
      <div id="account_info" class="parent">
         <div class="child">Account Details:
            <div class="child">Username: {{name}}</div>
            <div class="child">Email: {{email}}</div>
         </div>
         <div class="child">Subjects:
            {% for sub in subs %}
               <div class="child">{{sub}}</div>
            {% endfor %}
         </div>
      </div>

      <button id="sub_edit" onclick="edit_subjects()">Edit subjects</button>
      <button id="info_edit" onclick="edit_info()">Edit account info</button>
      <!-- the form from which the user can edit which subjects they are
doing -->
      <form id="edit_sub_form" method="post" action="/account/subject_change"
style="display: none; width: fit-content;">
         <input type="checkbox" name="maths_check" id="maths_check"
style="justify-content: center;">Mathematics<br>
         <input type="checkbox" name="comp_check" id="comp_check"
style="justify-content: center;">Computer Science<br>
         <input type="submit"><br><br>
         <button onclick="window.location.href='/account'">Back</button>
      </form>
      <!-- the form where a user can edit their personal details -->
      <div id="edit_info_div" style="display: none;" class="parent">
         <form id="edit_info_form" method="post"
action="/account/info_change" class="child">
            <label for="new_uname">New Username:</label>
            <input type="text" id="new_uname" name="new_uname"
placeholder="Username..."><br>
            <label for="new_email">New Email:</label><br>
            <input type="email" id="new_email" name="new_email"
placeholder="example@domain.com..."><br>
            <label for="old_password">Old Password:</label>
            <input type="password" id="old_password" name="old_password"
placeholder="Old Pa55w0rd!" required><br>
            <label for="new_password">New Passowrd:</label>
            <input type="password" id="new_password" name="new_password"
placeholder="N3w Pa55w0rd!"><br>
            <input type="submit" id="submit">
            <div class="child">
               <!-- this, as well as the checking, is the same as the login
page -->
               <label class="invalid" id="len_check">8-16 characters long
(inclusive)</label><br>
               <label class="invalid" id="upper_check">At least 1 uppercase
letter</label><br>
               <label class="invalid" id="lower_check">At least 1 lowercase
letter</label><br>
               <label class="invalid" id="num_check">At least 1
number</label><br>
               <label class="invalid" id="spec_check">At least 1 special
character</label><br>
               <label class="child">Username must be 3-16 characters
(inclusive) and may only contain alphanumerical characters</label>
            </div>
         </form><br>
```

```
          <button onclick="window.location.href='/account'">Back</button>
       </div>
      </body>

      <footer>
       <script src="/static/js/account_funcs.js"></script>
       <script src="/static/js/login_funcs.js"></script>
      </footer>

</html>
```

failures.html

```
<!DOCTYPE html>
<html>

    <head>
        <title>Subjects</title>
        <meta name="viewport" content="width=device-width, initial-
scale=1.0">
        <link rel="icon" type="image/x-icon"
href="/static/images/favicon.ico">
        <link rel="stylesheet" type="text/css"
href="/static/css/stylings.css">
    </head>

    <header>
        <ul>
           <li><a href="/">Home</a></li>
           <li><a href="/subjects">Subjects</a></li>
           <li><a href="/progress">Progress</a></li>
           <li style="float:right"><a href="/account">Account</a></li>
           <li style="float:right"><a href="/logout">Logout</a></li>
        </ul>
     </header>

     <body>
        <!-- these if statements take the value of 'type' from the redirect
and displays one of 5 error messages depending on what has caused the user to
be redirected here -->
        {% if type == 'quiz' %}
            <h1 style="margin: auto; padding: 10px; text-align:
center;">Unfortunately, there are either not enough or no questions for the
topic "{{topic}}" in the database</h1>
            <h1 style="text-align: center;">Please try again</h1>
            <button onclick="window.location.href='/subjects'">Try
again</button>
        {% elif type == 'login' %}
            <h1 style="text-align: center;">Login has failed</h1>
            <h1 style="text-align: center;">Please try again</h1>
            <button onclick="window.location.href='/login'">Try
again</button>
        {% elif type == 'registering' %}
            <h1 style="text-align: center;">Registering has failed</h1>
            <h1 style="text-align: center;">Please try again</h1>
            <button onclick="window.location.href='/login'">Try
again</button>
```

```
        {% elif type == 'account' %}
            <h1 style="text-align: center;">Account info updater has 
failed</h1>
            <h1 style="text-align: center;">Please try again</h1>
            <button onclick="window.location.href='/account'">Try 
again</button>
        {% else %}
            <h1 style="text-align: center;">Unknown error occured</h1>
            <h1 style="text-align: center;">Please try again</h1>
            <button onclick="window.location.href='/'">Try again</button>
        {% endif %}
     </body>

     <footer>
        <script src="/static/js/quiz_funcs.js"> </script>
    </footer>
</html>
```

login_page.html

```
<!DOCTYPE html>
<html>

   <head>
      <title>Login</title>
      <meta name="viewport" content="width=device-width, initial-scale=1.0">
      <link rel="icon" type="image/x-icon" href="/static/images/favicon.ico">
      <link rel="stylesheet" type="text/css" href="/static/css/stylings.css">
   </head>

   <header>
      <ul>
         <li><a href="/">Home</a></li>
         <li><a href="/subjects">Subjects</a></li>
         <li><a href="/progress">Progress</a></li>
         <li style="float:right"><a href="/account">Account</a></li>
         <li style="float:right"><a href="/login">Login</a></li>
      </ul>
   </header>

   <body>
      <h1 style="text-align: center;">Revise Wizz</h1>
      <h2 id="choice_text" style="text-align: center;">Login or create an 
account</h2>

      <button type="button" id="create_acc" onclick="create_account()">Create 
an Account</button>
      <button type="button" id="login_acc" 
onclick="login_account()">Login</button>
      <!-- the create form allows the user to create a new account for the 
website -->
      <div id="create_form"  style="display: none;" >
         <form method="post" action="/register" class="register">
            <label for="new_uname">Username:</label><br>
            <input type="text" id="new_uname" name="new_uname" 
placeholder="Username..." required><br>
            <label for="new_password">Password:</label><br>
            <input type="password" id="new_password" name="new_password"
placeholder="Pa55w0rd!..." required><br>
            <label for="email">Email:</label><br>
            <input type="text" id="new_email" name="new_email"
placeholder="example@domain.com..." required><br>
            <input type="submit" id="submit"><br>
         </form><br>
         <!-- the password contains validation in the associated js file for
checking it meets the requirements for a strong password -->
         <label class="invalid" id="len_check">8-16 characters long
(inclusive)</label><br>
         <label class="invalid" id="upper_check">At least 1 uppercase
letter</label><br>
         <label class="invalid" id="lower_check">At least 1 lowercase
letter</label><br>
         <label class="invalid" id="num_check">At least 1 number</label><br>
         <label class="invalid" id="spec_check">At least 1 special
character</label><br>
         <label>Username must be 3-16 characters (inclusive) and may only
contain alphanumerical characters</label><br>
         <button onclick="window.location.href='/login'">Back</button>
      </div>
      <!-- this form allows the user to login to a pre-existing account
within the database -->
      <form id="login_form" method="post" style="display: none;"
action="/login_check">
         <label for="uname">Username:</label><br>
         <input type="text" id="uname" name="uname" placeholder="Username..."
required><br>
         <label for="pword">Password:</label><br>
         <input type="password" id="pword" name="pword"
placeholder="Pa55w0rd!..." required><br>
         <label for="email">Email:</label><br>
         <input type="email" id="email" name="email"
placeholder="example@domain.com..." required><br>
         <input type="submit"><br><br>
         <button onclick="window.location.href='/login'">Back</button>
      </form>
   </body>
   <footer>
      <script src="/static/js/login_funcs.js"> </script>
   </footer>

</html>
```

main_page.html

```
<!DOCTYPE html>
<html>

    <head>
        <title>Home</title>
        <meta name="viewport" content="width=device-width, initial-
scale=1.0">
        <link rel="icon" type="image/x-icon"
href="/static/images/favicon.ico">
```

```
        <link rel="stylesheet" type="text/css"
href="/static/css/stylings.css">
    </head>

    <!-- this is the default header for all pages on the website, allowing
navigation to all necessary pages -->
    <header>
        <ul>
           <li><a href="/">Home</a></li>
           <li><a href="/subjects">Subjects</a></li>
           <li><a href="/progress">Progress</a></li>
           <li style="float:right"><a href="/account">Account</a></li>
           <li style="float:right"><a href="/logout">Logout</a></li>
        </ul>
     </header>

    <body>
        <!-- this page simply displays a welcome message, as well as buttons
to each area of the website -->
        <h1 style="text-align: center;">Welcome {{name}} to Revise Wizz</h1>
        <h2 style="text-align: center;">Please select an option below</h2>

        <div>
            <button onclick="document.location='account'">Account</button>
            <button onclick="document.location='subjects'">Subjects</button>
        </div>
        <div>
            <button onclick="document.location='progress'">{{name}}'s
Progress</button>
            <button onclick="document.location='logout'">Logout</button>
        </div>
    </body>

    <footer>
    </footer>

</html>
```

quiz.html

```
<!DOCTYPE html>
<html>

    <head>
        <title>{{topic}} Quiz</title>
        <meta name="viewport" content="width=device-width, initial-
scale=1.0">
        <link rel="icon" type="image/x-icon"
href="/static/images/favicon.ico">
        <link rel="stylesheet" type="text/css"
href="/static/css/stylings.css">
    </head>

    <header>
        <ul>
           <li><a href="/">Home</a></li>
           <li><a href="/subjects">Subjects</a></li>
```

```
          <li><a href="/progress">Progress</a></li>
          <li style="float:right"><a href="/account">Account</a></li>
          <li style="float:right"><a href="/logout">Logout</a></li>
       </ul>
    </header>

    <body>
       <h1 style="text-align: center;">Quiz on: {{topic}}</h1>
       <!--
       this div contains the questions and answer boxes for the quiz
       a for loop is used to display each question
       -->
       <div class="parent" style="width: 40%;">
           {% for i in length %}
           <div class="child" name="questions" id="question_{{i}}">
               <h4 style="text-align: center; color: #d3d9d4;">Question
{{i}}: {{questions[i-1][0]}}</h4>
               {% if questions[i-1][3] == 'long_answer' %}
                   <textarea id="question_{{i}}_answer" style="width: 50%;"
required></textarea>
               {% elif questions[i-1][3] == 'short_answer' %}
                   <input type="text" id="question_{{i}}_answer"
style="width: 50%;" required>
               {% else %}
                   <p><b>WARNING: Multiple choice questions are currently
unsupported</b></p>
                   <input type="radio" id="question_{{i}}_answer" required>
               {% endif %}
           </div>
           {% endfor %}
           {% for i in length %}
           <!-- this div contains the answers, user's answers, and the
button to check the answer to a question -->
           <div class="child" name="answers" id="answer_{{i}}"
style="display: none;">
               <h4 style="text-align: center; color: #d3d9d4;">Answer to
{{questions[i-1][0]}}:<br> {{questions[i-1][1]}}</h4>
               <p><b>Your answer was:</b></p>
               <div class="child" id="question_{{i}}_answer_given"></div>
               <p><b>The following keyword(s) were found:</b></p>
               <div class="child" id="question_{{i}}_keywords"></div>
           </div>
           <br><input id="check_answers_{{i}}" type="submit"
onclick="check_questions(`{{i}}`, `{{questions[i-1][2]}}`)" style="cursor:
pointer; width: fit-content;" value="Check answer to question {{i}}">
           {% endfor %}
           <button id="end_quiz" style="display: none;"
onclick="window.location.href='/subjects'">Finish Quiz</button>
           <button id="new_quiz"
onclick="window.location.href='/subjects'">New Quiz</button>
       </div>
    </body>

    <footer>
       <script src="/static/js/quiz_funcs.js"> </script>
   </footer>
```

```
</html>
```

subjects.html

```
<!DOCTYPE html>
<html>

    <head>
        <title>Subjects</title>
        <meta name="viewport" content="width=device-width, initial-
scale=1.0">
        <link rel="icon" type="image/x-icon"
href="/static/images/favicon.ico">
        <link rel="stylesheet" type="text/css"
href="/static/css/stylings.css">
    </head>

    <header>
        <ul>
           <li><a href="/">Home</a></li>
           <li><a href="/subjects">Subjects</a></li>
           <li><a href="/progress">Progress</a></li>
           <li style="float:right"><a href="/account">Account</a></li>
           <li style="float:right"><a href="/logout">Logout</a></li>
        </ul>
     </header>

    <body>
        <h1 id="welcome_text" style="text-align: center;">Here are your
subjects, {{name}}</h1>
        <!-- this initial for loop creates a div for each subject, containing
the button to create a quiz and a question -->
        {% for sub in subs %}
        <div id="available_subjects" class="parent">
            <button id="{{sub}}" onclick="display_subject_options('{{sub}}',
'{{name}}')">{{sub}}</button>
            <div id="{{sub}}_options" class="child" style="display: none;">
                <button id="{{sub}}_quiz"
onclick="create_quiz('{{sub}}')">Create {{sub}} Quiz</button>
                <button id="{{sub}}_question"
onclick="create_question('{{sub}}')">Create {{sub}} Question</button>
            </div>
        </div>
        {% if sub == 'Mathematics' %}
        <!-- the create question form allows the user to create a question
for a topic of their choosing, allowing for each part of the question to be
set -->
        <form id="create_{{sub}}_form" class="parent" style="display: none;
width: 400px;" action="/subjects/create_question" method="post">
            <label for="topic_choice">Select a topic:</label><br>
            <select name="topic_choice" id="topic_choice" class="child">
                <option disabled selected>Topic Choice</option>
                {% for math in maths %}
                <option value="{{math[1]}}">{{math[1]}}</option>
                {% endfor %}
            </select><br>
            <label for="question_type">Question type:</label><br>
            <select name="question_type" id="question_type">
                <option disabled selected>Question Type</option>
                <option value="short_answer">Short answer (Few
words)</option>
                <option value="long_answer">Long answer (>=1
sentences)</option>
                <option value="multiple_choice_one">Multiple choice
(unsupported)</option>
            </select><br>
            <label for="question_name">Question title:</label><br>
            <input type="text" id="question_name" name="question_name"
required placeholder="Question name..."><br>
            <label for="question_answer">Question answer:</label><br>
            <textarea id="question_answer" name="question_answer" required
placeholder="Question's answer..."></textarea><br>
            <!-- this was my way of the user setting required keywords for a
question to be correct,
             it was the solution that made the most sense to be used as it
allows for the user to say what is important for the answer to be correct
             and makes it so that your answer isn't compared to the whole of
the answer in the database, just the keywords -->
            <label>Please put *'s around keywords in the answer, ie. "The
*CIR*, or *Current Instruction Register*, is a *register* in the CPU's
control unit that temporarily *stores* the *instruction* currently being
executed or decoded." And avoid using punctuation within keyword
*'s</label><br>
            <br><input type="submit">
        </form>
        <!-- to create a quiz, the user needs to set the topic and the amount
of questions, making it highly customisable, the questions are randomised so
no 2 quizzes are the same-->
        <form id="create_{{sub}}_quiz_form" class="parent" style="display:
none; width: 400px;" action="/subjects/create_quiz" method="post">
            <label for="topic_choice">Select a topic:</label><br>
            <select name="topic_choice" id="topic_choice" class="child">
                <option disabled selected>Topic Choice</option>
                {% for math in maths %}
                <option value="{{math[1]}}">{{math[1]}}</option>
                {% endfor %}
            </select><br>
            <label for="quiz_length">Number of questions</label><br>
            <input type="number" id="quiz_length" name="quiz_length" min="1"
max="20" step="1" value="1" placeholder="Between 1 and 20" required><br>
            <input type="submit">
        </form>
        {% else %}
        <!-- this section has the same things as above, but for computer
science -->
        <form id="create_{{sub}}_form" class="parent" style="display: none;
width: 400px;" action="/subjects/create_question" method="post">
            <label for="topic_choice">Select a topic:</label><br>
            <select name="topic_choice" id="topic_choice" required>
                <option disabled selected>Topic Choice</option>
                {% for comp in comps %}
                <option value="{{comp[1]}}">{{comp[1]}}</option>
                {% endfor %}
            </select><br>
            <label for="question_type">Question type:</label><br>
            <select name="question_type" id="question_type" required>
                <option disabled selected>Question Type</option>
                <option value="short_answer">Short answer (Few
words)</option>
                <option value="long_answer">Long answer (>=1
sentences)</option>
                <option value="multiple_choice_one">Multiple choice
(unsupported)</option>
            </select><br>
            <label for="question_name">Question title:</label><br>
            <input type="text" id="question_name" name="question_name"
required placeholder="Question name..."><br>
            <label for="question_answer">Question answer:</label><br>
            <textarea id="question_answer" name="question_answer" required
placeholder="Question's answer..."></textarea><br>
            <label>Please put *'s around keywords in the answer, ie. "The
*CIR*, or *Current Instruction Register*, is a *register* in the CPU's
control unit that temporarily *stores* the *instruction* currently being
executed or decoded." And avoid using punctuation within keyword
*'s</label><br>
            <br><input type="submit">
        </form>
        <form id="create_{{sub}}_quiz_form" class="parent" style="display:
none; width: 400px;" action="/subjects/create_quiz" method="post">
            <label for="topic_choice">Select a topic:</label><br>
            <select name="topic_choice" id="topic_choice" required>
                <option disabled selected>Topic Choice</option>
                {% for comp in comps %}
                <option value="{{comp[1]}}">{{comp[1]}}</option>
                {% endfor %}
            </select><br>
            <label for="quiz_length">Number of questions</label><br>
            <input type="number" id="quiz_length" name="quiz_length" min="1"
max="20" step="1" value="1" placeholder="Between 1 and 20" required><br>
            <input type="submit">
        </form>
        {% endif %}
        {% endfor %}
    </body>

    <footer>
        <script src="/static/js/subject_funcs.js"> </script>
    </footer>
</html>
```

user_progress.html

```
<!DOCTYPE html>
<html>

    <head>
        <title>Subjects</title>
        <meta name="viewport" content="width=device-width, initial-
scale=1.0">
```

```
        <link rel="icon" type="image/x-icon"
href="/static/images/favicon.ico">
        <link rel="stylesheet" type="text/css"
href="/static/css/stylings.css">
    </head>

    <header>
        <ul>
           <li><a href="/">Home</a></li>
           <li><a href="/subjects">Subjects</a></li>
           <li><a href="/progress">Progress</a></li>
           <li style="float:right"><a href="/account">Account</a></li>
           <li style="float:right"><a href="/logout">Logout</a></li>
        </ul>
     </header>

    <body>
        <h1 id="welcome_text" style="text-align: center;">Here are your
subjects, {{name}}</h1>
        <!-- this initial for loop creates a div for each subject, containing
the button to create a quiz and a question -->
        {% for sub in subs %}
        <div id="available_subjects" class="parent">
            <button id="{{sub}}" onclick="display_subject_options('{{sub}}',
'{{name}}')">{{sub}}</button>
            <div id="{{sub}}_options" class="child" style="display: none;">
                <button id="{{sub}}_quiz"
onclick="create_quiz('{{sub}}')">Create {{sub}} Quiz</button>
                <button id="{{sub}}_question"
onclick="create_question('{{sub}}')">Create {{sub}} Question</button>
            </div>
        </div>
        {% if sub == 'Mathematics' %}
        <!-- the create question form allows the user to create a question
for a topic of their choosing, allowing for each part of the question to be
set -->
        <form id="create_{{sub}}_form" class="parent" style="display: none;
width: 400px;" action="/subjects/create_question" method="post">
            <label for="topic_choice">Select a topic:</label><br>
            <select name="topic_choice" id="topic_choice" class="child">
                <option disabled selected>Topic Choice</option>
                {% for math in maths %}
                <option value="{{math[1]}}">{{math[1]}}</option>
                {% endfor %}
            </select><br>
            <label for="question_type">Question type:</label><br>
            <select name="question_type" id="question_type">
                <option disabled selected>Question Type</option>
                <option value="short_answer">Short answer (Few
words)</option>
                <option value="long_answer">Long answer (>=1
sentences)</option>
                <option value="multiple_choice_one">Multiple choice
(unsupported)</option>
            </select><br>
            <label for="question_name">Question title:</label><br>
            <input type="text" id="question_name" name="question_name"
required placeholder="Question name..."><br>
            <label for="question_answer">Question answer:</label><br>
            <textarea id="question_answer" name="question_answer" required
placeholder="Question's answer..."></textarea><br>
            <!-- this was my way of the user setting required keywords for a
question to be correct,
             it was the solution that made the most sense to be used as it
allows for the user to say what is important for the answer to be correct
             and makes it so that your answer isn't compared to the whole of
the answer in the database, just the keywords -->
            <label>Please put *'s around keywords in the answer, ie. "The
*CIR*, or *Current Instruction Register*, is a *register* in the CPU's
control unit that temporarily *stores* the *instruction* currently being
executed or decoded." And avoid using punctuation within keyword
*'s</label><br>
            <br><input type="submit">
        </form>
        <!-- to create a quiz, the user needs to set the topic and the amount
of questions, making it highly customisable, the questions are randomised so
no 2 quizzes are the same-->
        <form id="create_{{sub}}_quiz_form" class="parent" style="display:
none; width: 400px;" action="/subjects/create_quiz" method="post">
            <label for="topic_choice">Select a topic:</label><br>
            <select name="topic_choice" id="topic_choice" class="child">
                <option disabled selected>Topic Choice</option>
                {% for math in maths %}
                <option value="{{math[1]}}">{{math[1]}}</option>
                {% endfor %}
            </select><br>
            <label for="quiz_length">Number of questions</label><br>
            <input type="number" id="quiz_length" name="quiz_length" min="1"
max="20" step="1" value="1" placeholder="Between 1 and 20" required><br>
            <input type="submit">
        </form>
        {% else %}
        <!-- this section has the same things as above, but for computer
science -->
        <form id="create_{{sub}}_form" class="parent" style="display: none;
width: 400px;" action="/subjects/create_question" method="post">
            <label for="topic_choice">Select a topic:</label><br>
            <select name="topic_choice" id="topic_choice" required>
                <option disabled selected>Topic Choice</option>
                {% for comp in comps %}
                <option value="{{comp[1]}}">{{comp[1]}}</option>
                {% endfor %}
            </select><br>
            <label for="question_type">Question type:</label><br>
            <select name="question_type" id="question_type" required>
                <option disabled selected>Question Type</option>
                <option value="short_answer">Short answer (Few
words)</option>
                <option value="long_answer">Long answer (>=1
sentences)</option>
                <option value="multiple_choice_one">Multiple choice
(unsupported)</option>
```

```
            </select><br>
            <label for="question_name">Question title:</label><br>
            <input type="text" id="question_name" name="question_name"
required placeholder="Question name..."><br>
            <label for="question_answer">Question answer:</label><br>
            <textarea id="question_answer" name="question_answer" required
placeholder="Question's answer..."></textarea><br>
            <label>Please put *'s around keywords in the answer, ie. "The
*CIR*, or *Current Instruction Register*, is a *register* in the CPU's
control unit that temporarily *stores* the *instruction* currently being
executed or decoded." And avoid using punctuation within keyword
*'s</label><br>
            <br><input type="submit">
        </form>
        <form id="create_{{sub}}_quiz_form" class="parent" style="display:
none; width: 400px;" action="/subjects/create_quiz" method="post">
            <label for="topic_choice">Select a topic:</label><br>
            <select name="topic_choice" id="topic_choice" required>
                <option disabled selected>Topic Choice</option>
                {% for comp in comps %}
                <option value="{{comp[1]}}">{{comp[1]}}</option>
                {% endfor %}
            </select><br>
            <label for="quiz_length">Number of questions</label><br>
            <input type="number" id="quiz_length" name="quiz_length" min="1"
max="20" step="1" value="1" placeholder="Between 1 and 20" required><br>
            <input type="submit">
        </form>
        {% endif %}
        {% endfor %}
    </body>

    <footer>
        <script src="/static/js/subject_funcs.js"> </script>
    </footer>
</html>
```

databases.py

```
from argon2 import PasswordHasher
import sqlite3
import random
import re
# argon2 is a hashing package for python, it made more sense for security to
use a pre-built hashing function than to create my own
# this file uses object orientated programming, with each table having its
own class, full of methods that allow for easy use for the backend
class UsersTable:
    def __init__(self):
        self.connection = sqlite3.connect("storage.db")
        self.connection.execute("PRAGMA foreign_keys = 1")
        self.cursor = self.connection.cursor()
        self.cursor.execute("""CREATE TABLE IF NOT EXISTS users
                            (userID INTEGER PRIMARY KEY AUTOINCREMENT,
                            username TEXT NOT NULL,
                            password TEXT NOT NULL,
                            email TEXT NOT NULL)""")
        self.connection.commit()
        self.ph = PasswordHasher()

    # this subroutine checks if a username and email already exists, and then 
adds it if it doesn't
    def create_login(self, user, pword, email):
        exists = self.cursor.execute("""SELECT username, email FROM users 
WHERE username = ?""", (user,))
        if exists.fetchone() is None:
            hashed_pword = self.ph.hash(pword)
            self.cursor.execute("""INSERT INTO users(username, password, 
email) VALUES(?, ?, ?)""", (user, hashed_pword, email,))
            self.connection.commit()
            return True
        else:
            return False

    # this is called when a user logs in, to make sure that the details they 
entered are correct
    def login_check(self, user, pword, email):
        uname_check = self.cursor.execute("""SELECT username FROM users WHERE 
username = ?""", (user,))
        uname_check = uname_check.fetchone()
        email_check = self.cursor.execute("""SELECT email FROM users WHERE 
email = ?""", (email,))
        email_check = email_check.fetchone()
        if uname_check is None or email_check is None:
            return False
        else:
            pword_check = self.cursor.execute("""SELECT password FROM users 
WHERE username = ?""", (user,))
            if self.ph.verify(pword_check.fetchone()[0], pword):
                return True
            else:
                return False

    # changing the users' details is performed here, by first verifying that 
the old password entered is correct, then updating what is needed
    def change_details(self, userID, old_pass, new_details):
        try:
            self.ph.verify(self.cursor.execute("""SELECT password FROM users 
WHERE userID=?""", (userID,)).fetchone()[0], old_pass)
            if new_details["new_pass"] != '':   # this changes the user's 
password
                new_hashed_pword = self.ph.hash(new_details["new_pass"])
                self.cursor.execute(f"""UPDATE users SET password=? WHERE 
userID={userID}""", (new_hashed_pword,))
                self.connection.commit()
            else:
                pass
            if new_details["new_uname"] != '':   # this changes the username
                self.cursor.execute(f"""UPDATE users SET username=? WHERE 
userID={userID}""", (new_details["new_uname"],))
                self.connection.commit()
            else:
                pass
            if new_details["new_email"] != '':   # this changes the user's
email
                self.cursor.execute(f"""UPDATE users SET email=? WHERE
userID={userID}""", (new_details["new_email"],))
                self.connection.commit()
            else:
                pass
            return True
        except:
            return False

    # this is one of many subroutines that just retrieve a field, this one
fetches a userID using a user's details
    def userID_get(self, user, pword, email):
        hashed_pword = self.ph.verify(self.cursor.execute("""SELECT password
FROM users WHERE username = ?""", (user,)).fetchone()[0], pword)
        if hashed_pword:
            id_found = self.cursor.execute(f"""SELECT userID FROM users WHERE
username = ? AND email = ?""", (user, email,))
            id = id_found.fetchone()[0]
        return id

class SubjectsTable:
    def __init__(self):
        self.connection = sqlite3.connect("storage.db")
        self.connection.execute("PRAGMA foreign_keys = 1")
        self.cursor = self.connection.cursor()
        self.cursor.execute("""CREATE TABLE IF NOT EXISTS subjects
                            (subjectID INTEGER PRIMARY KEY AUTOINCREMENT,
                            subjectName TEXT NOT NULL)""")
        self.connection.commit()

    # this takes an array of subject IDs and 'converts' it to subject names
    def subIDs_to_sub(self, subject_ids):
        subjects = []
        for item in subject_ids:
            self.cursor.execute(f"""SELECT subjectName FROM subjects WHERE
subjectID = ?""", (item,))
            subjects.append(self.cursor.fetchone()[0])
        return subjects

    # this takes an array of subject names and 'converts' it to subject IDs
    def subs_to_subID(self, subjects):
        subjectIDs = []
        for item in subjects:
            self.cursor.execute(f"""SELECT subjectID FROM subjects WHERE
subjectName = ?""", (item,))
            subjectIDs.append(self.cursor.fetchall()[0])
        return subjectIDs

class SubjectUserTable:
    def __init__(self):
        self.connection = sqlite3.connect("storage.db")
        self.connection.execute("PRAGMA foreign_keys = 1")
        self.connection.row_factory = lambda cursor, row: row[0]
        self.cursor = self.connection.cursor()
        self.cursor.execute("""CREATE TABLE IF NOT EXISTS usersubjects
                            (userID INTEGER,
                            subjectID INTEGER,
                            FOREIGN KEY (userID) REFERENCES users (userID),
                            FOREIGN KEY (subjectID) REFERENCES subjects
(subjectID))""")
        self.connection.commit()

    # this fetches the subject IDs tied to a specific user
    def subIDs_get_from_userID(self, userID):
        self.cursor.execute(f"""SELECT subjectID FROM usersubjects WHERE
userID = ?""", (userID,))
        sub_id = self.cursor.fetchall()
        if sub_id == []:
            return ''
        else:
            return sub_id

    # this takes the user ID and array of subjects to add/remove
    def subject_change(self, userID, changing_subjectIDs):
        for item in changing_subjectIDs:
            sub_change = item.split(',')
            self.cursor.execute(f"""SELECT subjectID FROM usersubjects WHERE
subjectID={sub_change[0]}""")
            sub_found = self.cursor.fetchall()
            if sub_change[1] == 'true' and sub_found == []:
#add subject to table
                self.cursor.execute("""INSERT INTO usersubjects
VALUES(?,?)""", (userID, sub_change[0],))
            elif sub_change[1] == 'false' and sub_found ==
[int(sub_change[0])]:    #remove from table
                self.cursor.execute("""DELETE FROM usersubjects WHERE
subjectID=? AND userId=?""", (sub_found[0], userID,))
            self.connection.commit()

class TopicsTable:
    def __init__(self):
        self.connection = sqlite3.connect("storage.db")
        self.connection.execute("PRAGMA foreign_keys = 1")
        self.cursor = self.connection.cursor()
        self.cursor.execute("""CREATE TABLE IF NOT EXISTS topics
                            (topicID INTEGER PRIMARY KEY AUTOINCREMENT,
                            subjectID INTEGER,
                            topicName TEXT NOT NULL,
                            qualification TEXT NOT NULL,
                            FOREIGN KEY (subjectID) REFERENCES subjects
(subjectID))""")
        self.connection.commit()

    # this retrieves the topics relating to a given subject ID
    def get_topics(self, subID):
        topics = []
        for item1 in subID:
            self.cursor.execute("""SELECT topicName FROM topics WHERE
subjectID=?""", (item1, ))
```

```
            fetched_topics = self.cursor.fetchall()
            for item2 in fetched_topics:
                topics.append([item1, item2[0]])
        if topics == []:
            return ''
        else:
            return topics

    # this converts an array of topics into their related topic IDs
    def topic_to_topicID(self, topics):
        topicIDs = []
        for item1 in topics:
            self.cursor.execute(f"""SELECT topicID FROM topics WHERE
topicName = ?""", (item1,))
            fetched_topics = self.cursor.fetchall()
            for item2 in fetched_topics:
                topicIDs.append(item2)
        return topicIDs

    # this gets all topic IDs associated with a given subject
    def subID_to_topicID(self, subIDs):
        topicIDs = []
        for item in subIDs:
            self.cursor.execute("""SELECT topicID FROM topics WHERE
subjectID=?""", (item, ))
            fetched_ids = self.cursor.fetchall()
            for item2 in fetched_ids:
                topicIDs.append(item2)
        return topicIDs

class QuestionsTable:
    def __init__(self):
        self.connection = sqlite3.connect("storage.db")
        self.connection.execute("PRAGMA foreign_keys = 1")
        self.cursor = self.connection.cursor()
        self.cursor.execute("""CREATE TABLE IF NOT EXISTS questions
                            (questionID INTEGER PRIMARY KEY AUTOINCREMENT,
                            topicID INTEGER,
                            question TEXT NOT NULL,
                            answer TEXT NOT NULL,
                            answer_keywords TEXT,
                            question_type TEXT NOT NULL,
                            FOREIGN KEY (topicID) REFERENCES topics
(topicID))""")
        self.connection.commit()

    # this creates the list of keywords, then adds all the necessary data to
the questions table in the database
    def create_question(self, question_data):
        keywords_list = re.findall(r"\*[A-Za-z0-9 ]+\*",
question_data["answer"])
        keywords = self.construct_keywords(keywords_list)
        topicId = TopicsTable.topic_to_topicID(self,
[question_data["topic"]])
        self.cursor.execute("""INSERT INTO questions(topicID, question,
answer, answer_keywords, question_type) VALUES(?, ?, ?, ?, ?)""",
(topicId[0][0], question_data["name"], question_data["answer"], keywords,
question_data["type"], ))
        self.connection.commit()

    # this takes an array of keywords, and combines them into a list to be
inserted into the database
    def construct_keywords(self, keywords):
        keywords = "-".join(keywords)
        return keywords

    # this takes the string on keywords and converts it back into an array
    def deconstruct_keywords(self, keywords):
        split_keywords = keywords.split("-")
        return split_keywords

    # this returns fetches the questions for the quizzes, if it cannot find
enough or any questions, it returns false which leads to the failure page
    def create_quiz(self, quiz_data):
        topicId = TopicsTable.topic_to_topicID(self, [quiz_data["topic"]])
        self.cursor.execute(f"""SELECT question, answer, answer_keywords,
question_type FROM questions WHERE topicID='{topicId[0][0]}'""")
        questions = self.cursor.fetchall()
        questions = [item for item in questions]
        chosen_questions = []
        if len(questions) >= int(quiz_data["quiz_length"]):
            for i in range(0, len(questions)):  # this loop takes the
questions fetched for a given topic, and creates a 2D array of random
questions, depending on the quiz length provided
                chosen_temp = random.choice((questions))
                questions.remove(chosen_temp)
                chosen_questions.append(chosen_temp)
            return [True, chosen_questions[0:int(quiz_data["quiz_length"])]]
        else:
            return [False]

class UserProgressTable:
    def __init__(self):
        self.connection = sqlite3.connect("storage.db")
        self.connection.execute("PRAGMA foreign_keys = 1")
        self.connection.row_factory = lambda cursor, row: row[0]    # this is
another way of converting an array to a tuple, by making the fetch() function
return an array
        self.cursor = self.connection.cursor()
        self.cursor.execute("""CREATE TABLE IF NOT EXISTS user_progress
                            (topicID INTEGER,
                            userID INTEGER,
                            progress INTEGER,
                            FOREIGN KEY (topicID) REFERENCES topics (topicID)
                            FOREIGN KEY (userID) REFERENCES users
(userID))""")
        self.connection.commit()

    # this adds all the default progress to a user, using both foreign keys
    def add_progress(self, userID, subjectID):
        topics = TopicsTable.subID_to_topicID(self, [subjectID])
        for i in range(0, len(topics)):
            self.cursor.execute("""INSERT INTO user_progress(userID, topicID,
progress) VALUES(?, ?, 1)""", (userID, topics[i], ))
            self.connection.commit()

    # this removes all the progress for a given user's subject
    def del_progress(self, userID, subjectID):
        topics = TopicsTable.subID_to_topicID(self, [subjectID])
        for i in range(0, len(topics)):
            self.cursor.execute("""DELETE FROM user_progress WHERE userID=?
AND topicID=?""", (userID, topics[i], ))
            self.connection.commit()

    # this changes the RAG value of a topic for a user
    def change_progress(self, change_data):
        topicID = TopicsTable.topic_to_topicID(self, [change_data["topic"]])
        self.cursor.execute(f"""UPDATE user_progress SET progress=? WHERE
userID=? AND topicID=?""", (change_data['rag'], change_data['userID'],
topicID[0], ))
        self.connection.commit()

    # this retrieves all the progress for a given user
    def get_progress(self, userID):
        self.cursor.execute("""SELECT progress FROM user_progress WHERE
userID=?""", (userID, ))
        fetched_progress = self.cursor.fetchall()
        progress_values = self.progNum_to_prog(fetched_progress)
        return progress_values

    # this converts the stored RAG value (1, 2, or 3) to the associated name
    def progNum_to_prog(self, progress_number):
        progress_values = []
        for item in progress_number:
            if item == 1:
                progress_values.append("Red")
            elif item == 2:
                progress_values.append("Amber")
            else:
                progress_values.append("Green")
        return progress_values
```

main_webloader.py

```
from flask import Flask, render_template, redirect, request, session
from datetime import timedelta
from databases import *
import secrets
import re

# creating the inital setup for the website (encryption and cookie expiry
length)
app = Flask(__name__)
app.secret_key = secrets.token_bytes(nbytes=32)
app.config['PERMANENT_SESSION_LIFETIME'] = timedelta(hours=3)

# landing page of website
@app.route('/', methods=['POST', 'GET'])
```

```
def main():
    if 'username' in session:   # this inital if statement (present in most 
app.routes) makes sure the user is logged in, if not then they will be 
prompted to login
        username = session['username']
        return render_template('main_page.html', name=username) # the use of 
the name value here will pass in the username from the cookie to be used on 
the html page, similar variables are used through each page to be used in the 
website
    else:
        return redirect('/login')

# login page
@app.route('/login', methods=['POST', 'GET'])
def login():
    return render_template('login_page.html')

# this page will clear the cookie and send the user back to the login page
@app.route('/logout')
def logout():
    session.clear()
    return redirect('/login')

# the account page displays all users personal details, with options to 
change when needed
@app.route('/account', methods=['POST', 'GET'])
def account():
    if 'username' in session:
        username, email, userID = session['username'], session['email'], 
session['userID']
        subjectsuser_oop, subject_oop = SubjectUserTable(), SubjectsTable() # 
this is how the code and use subroutines in the 'databases.py' file, and are 
used as and when needed throughout
        subjectIDs = subjectsuser_oop.subIDs_get_from_userID(userID)
        subjects = subject_oop.subIDs_to_sub(subjectIDs)
        subs = []
        for item in subjects:   # this is one of many methods used to convert 
the tuple (returned from sqlite select statements) to arrays, html does not 
work well with tuples and thus I made ways to change from tuples to arrays
            subs.append(item)
        return render_template('account_page.html', name=username, 
email=email, subs=subs)
    else:
        return redirect('/login')

# this is the way in which a user changes their subject choices
@app.route('/account/subject_change', methods=['POST','GET'])
def subject_change():
    if 'username' in session:
        userID = session['userID']
        subchange_oop = SubjectUserTable()
        progress_oop = UserProgressTable()
        subjects_selected = []  # the following if statements separate the 
subjects so that they can be used to add/remove the default progress of the 
user
        if request.form.get('maths_check'):
            subjects_selected.append('1,true')
        else:
            subjects_selected.append('1,false')
        if request.form.get('comp_check'):
            subjects_selected.append('2,true')
        else:
            subjects_selected.append('2,false')
        subchange_oop.subject_change(userID, subjects_selected)
        for item in subjects_selected:
            split_item = item.split(',')
            if split_item[1] == 'false':
                progress_oop.del_progress(userID, split_item[0])
            else:
                progress_oop.add_progress(userID, split_item[0])    # this is
knowingly a closed-minded way of coding, as it does not allow for expansion
of other subjects
        return redirect('/account')
    else:
        return redirect('/login')

# this is where the user can change their account information
@app.route('/account/info_change', methods=['POST','GET'])
def info_change():
    if 'username' in session:
        userID = session['userID']
        login_oop = UsersTable()    # this uses a dictionary to get each
piece of the data from the form
        new_information = {'new_uname': request.form.get('new_uname'),
                           'new_email': request.form.get('new_email'),
                           'new_pass': request.form.get('new_password')}
        updated_info = login_oop.change_details(userID,
request.form.get('old_password'), new_information)
        if updated_info == True:
            return redirect('/account')
        else:   # the failure page is a default page that changes based on
the 'type' to suit where it got redirected from
            return render_template("failures.html", type="account")
    else:
        return redirect('/login')

# this routine checks that the login details given match what is in the
datbase, more depth given in the comments of the subroutine called
@app.route('/login_check', methods=['POST','GET'])
def login_check():
    login_oop = UsersTable()
    if login_oop.login_check(request.form.get('uname'),
request.form.get('pword'), request.form.get('email')):
        session['username'], session['email'] = request.form.get('uname'),
request.form.get('email')
        session['userID'] = login_oop.userID_get(request.form.get('uname'),
request.form.get('pword'), request.form.get('email'))
        return redirect('/')
    else:
        return render_template("failures.html", type="login")
```

```
# this is similar to the routine above, but instead checking that the details
match all requirements
@app.route('/register', methods=['POST','GET'])
def registering():
    login_oop = UsersTable()
    if request.form.get('new_uname') == '' or request.form.get('new_pword')
== '' or request.form.get('new_email') == '':
        return render_template("failures.html", type="registering")
    elif re.fullmatch(r"[A-Za-z0-9]+", request.form.get('new_uname')) == None
or re.fullmatch(r"[a-z]+[A-Z]+[0-9]+[!-\/:-@[-`{-~]+",
request.form.get('new_pword')) or re.fullmatch(r"[-A-Za-z0-
9!#$%&'*+/=?^_`{|}~]+(?:\.[-A-Za-z0-9!#$%&'*+/=?^_`{|}~]+)*@(?:[A-Za-z0-
9](?:[-A-Za-z0-9]*[A-Za-z0-9])?\.)+[A-Za-z0-9](?:[-A-Za-z0-9]*[A-Za-z0-9])?",
request.form.get('new_email')):
        return render_template("failures.html", type="registering")
    elif login_oop.create_login(request.form.get('new_uname'),
request.form.get('new_pword'), request.form.get('new_email')):
        return redirect('/login')
    else:
        return render_template("failures.html", type="registering")

# this page displays the users' subjects, from which they can create
questions of quizzes
@app.route('/subjects', methods=['POST', 'GET'])
def subjects():
    if 'username' in session:
        username, userID = session['username'], session['userID']
        subjectsuser_oop, subject_oop, topics_oop = SubjectUserTable(),
SubjectsTable(), TopicsTable()
        subjectIDs = subjectsuser_oop.subIDs_get_from_userID(userID)
        subjects = subject_oop.subIDs_to_sub(subjectIDs)
        topics = topics_oop.get_topics(subjectIDs)
        subs = []
        for item in subjects:
            subs.append(item)
        topic_list = []
        for item in topics:
            topic_list.append(item)
        maths_topics = []
        comp_topics = []
        for item in topic_list:
            if item[0] == 1:
                maths_topics.append(item)
            else:
                comp_topics.append(item)
        return render_template('subjects.html', name=username, subs=subs,
maths=maths_topics, comps=comp_topics)
    else:
        return redirect('/login')

# creating a question utilises a dictionary to parse in the data to be added
to the database
@app.route('/subjects/create_question', methods=['POST', 'GET'])
def create_question():
    if 'username' in session:
        questions_oop = QuestionsTable()
        question_data = {"topic": request.form.get('topic_choice'),
                         "type": request.form.get('question_type'),
                         "name": request.form.get('question_name'),
                         "answer": request.form.get('question_answer')}
        questions_oop.create_question(question_data)
        return redirect('/subjects')
    else:
        return redirect('/login')

# creating a quiz requires all the data from a question to be used very
specifically in the html code
@app.route('/subjects/create_quiz', methods=['POST', 'GET'])
def create_quiz():
    if 'username' in session:
        questions_oop = QuestionsTable()
        quiz_data = {"topic": request.form.get('topic_choice'),
                     "quiz_length": request.form.get('quiz_length')}
        question_data = questions_oop.create_quiz(quiz_data)
        length = []
        for i in range(1, int(quiz_data["quiz_length"])+1):
            length.append(i)
        if question_data[0] == True:
            list_of_tuples = question_data[1]
            list_of_lists = []
            for item in list_of_tuples:
                list_of_lists.append(list(item))
            return render_template('quiz.html', topic=quiz_data['topic'],
length=length , questions=list_of_lists)
        else:
            return render_template("failures.html", type="quiz",
topic=quiz_data['topic'])
    else:
        return redirect('/login')

# the progress page displays a users progress within a given topic, which
they can choose and change as they deem fit
@app.route('/progress', methods=['POST', 'GET'])
def progress():
    if 'username' in session:
        username, userID = session['username'], session['userID']
        subjectsuser_oop, subject_oop, topics_oop, progress_oop =
SubjectUserTable(), SubjectsTable(), TopicsTable(), UserProgressTable()
        subjectIDs = subjectsuser_oop.subIDs_get_from_userID(userID)
        subjects = subject_oop.subIDs_to_sub(subjectIDs)
        topics, rag_values = topics_oop.get_topics(subjectIDs),
progress_oop.get_progress(userID)
        subs = []
        for item in subjects:
            subs.append(item)
        topic_list = []
        for item in topics:
            topic_list.append(item)
        maths_topics = []
        comp_topics = []
        for i in range(len(topic_list)):
            if topic_list[i][0] == 1:
                maths_topics.append([topic_list[i][1], rag_values[i]])
            else:
                comp_topics.append([topic_list[i][1], rag_values[i]])
        return render_template('user_progress.html', name=username,
subs=subs, topics=topic_list, maths=maths_topics, comps=comp_topics)
    else:
        return redirect('/login')

# this allows users to change their progress of a given topic, on a scale of
Red, Amber, Green
@app.route('/progress/change_progress', methods=['POST', 'GET'])
def change_progress():
    if 'username' in session:
        progress_oop = UserProgressTable()
        userID = session['userID']
        change_data = {"userID": userID,
                       "topic": request.form.get("topic_choice"),
                       "rag": request.form.get("rag_choice")}
        progress_oop.change_progress(change_data)
        return redirect('/progress')
    else:
        return redirect('/login')

if __name__=='__main__':
   app.run(debug=True)
```

database_data.txt

```
List of data that is in each database by default

subjects table
cursor.execute("""INSERT INTO subjects(subjectName, examBoard)
VALUES("Maths", "OCR MEI B"),
                ("Computer Science","AQA")""")

topics table
cursor.execute("""INSERT INTO topics(subjectID, topicName, qualification)
VALUES(1,"1.1 Problem Solving","A1"),
                (1,"1.2 Surds and indices","A1"),
                (1,"1.3 Quadratic functions","A1"),
                (1,"1.4 Equations and inequalities","A1"),
                (1,"1.5 Coordinate geometry","A1"),
                (1,"1.6 Trigonometry","A1"),
                (1,"1.7 Polynomials","A1"),
                (1,"1.8 Graphs and transformations","A1"),
                (1,"1.9 The binomial expansion","A1"),
                (1,"1.10 Differentiation and integration","A1"),
                (1,"1.11 Vectors","A1"),
                (1,"1.12 Exponentials and logarithms","A1"),
                (1,"1.13 Kinematics","A1"),
                (1,"1.14 Forces and Newton's laws","A1"),
                (1,"1.15 Variable acceleration","A1"),
                (1,"1.16 Collecting and interpreting data","A1"),
                (1,"1.17 Probability","A1"),
                (1,"1.18 The binomial distribution","A1"),
                (1,"1.19 Statistical hypothesis testing","A1"),
                (1,"2.1 Proof","A2"),
                (1,"2.2 Trigonometry","A2"),
                (1,"2.3 Sequences and series","A2"),
                (1,"2.4 Functions","A2"),
                (1,"2.5 Differentiation and integration","A2"),
                (1,"2.6 Algebra","A2"),
                (1,"2.7 Parametric equations","A2"),
                (1,"2.8 Differential equations","A2"),
                (1,"2.9 Numerical methods","A2"),
                (1,"2.10 Kinematics","A2"),
                (1,"2.11 Forces and motion","A2"),
                (1,"2.12 Moments of forces","A2"),
                (1,"2.13 Projectiles","A2"),
                (1,"2.14 A model for friction","A2"),
                (1,"2.15 Probability","A2"),
                (1,"2.16 Statistical distributions","A2"),
                (1,"2.17 Statistical hypothesis testing","A2"),
                (2,"3.1 Fundamentals of programming","A1"),
                (2,"3.2 Fundamentals of data structures","A1"),
                (2,"3.3 Systematic approach to problem solving","A1"),
                (2,"3.4 Theory of computation","A1"),
                (2,"3.5 Fundamentals of data representation","A1"),
                (2,"3.6 Fundamentals of computer systems","A1"),
                (2,"3.7 Fundamentals of computer organisation and 
architecture","A1"),
                (2,"3.8 Consequences of uses of computing","A1"),
                (2,"3.9 Fundamentals of communication and networking","A1"),
                (2,"4.1 Fundamentals of programming","A2"),
                (2,"4.2 Fundamentals of data structures","A2"),
                (2,"4.3 Systematic approach to problem solving","A2"),
                (2,"4.4 Theory of computation","A2"),
                (2,"4.5 Fundamentals of data representation","A2"),
                (2,"4.6 Fundamentals of computer systems","A2"),
                (2,"4.7 Fundamentals of computer organisation and 
architecture","A2"),
                (2,"4.8 Consequences of uses of computing","A2"),
                (2,"4.9 Fundamentals of communication and networking","A2"),
                (2,"4.10 Fundamentals of databases","A2"),
                (2,"4.11 Big data","A2"),
                (2,"4.12 Fundamentals of functional programming","A2"),
                (2,"4.13 Systematic approach to problem solving","A2")""")
```

References

1. **Educake.** Landing Page. *Educake.* [Online] https://www.educake.co.uk/.

2. **Quizziz.** Landing Page. *Quizziz.* [Online] https://quizizz.com/.

3. **Carousel Learning.** Landing Page. *Carousel Learning.* [Online] https://www.carousel-learning.com/.

4. **W3Schools.** JavaScript Tutorial. *W3Schools.* [Online] https://www.w3schools.com/js/default.asp.

5. **Tutorialspoint.** JavaScript Tutorial. *TutorialsPoint.* [Online] https://www.tutorialspoint.com/javascript/index.htm.

6. **W3Schools.** CSS Tutorial. *W3Schools.* [Online] https://www.w3schools.com/css/default.asp.

7. **Flask.** Flask Documentation. *Flask.* [Online] https://flask.palletsprojects.com/en/3.0.x/.

8. **GeeksForGeeks.** Flask Tutoial. *GeeksForGeeks.* [Online] https://www.geeksforgeeks.org/flask-tutorial/?ref=gcse_outind.

9. **Lucid.** *Lucid.com.* [Online] https://lucid.app/.

10. **dbdiagram.** [Online] https://dbdiagram.io/.

11. **Navaigation Data Standard, Bentley, Expensify, Bloomberg Engineering.** Documentation. *SQLite.* [Online]

12. **Figma.** *figma.com.* [Online] https://www.figma.com/.

13. **Python Software Foundation.** re — Regular expression operations. *docs.python.org.* [Online] Python Software Foundation. https://docs.python.org/3/library/re.html.

14. **gskinner.** RFC2822 Email Validation. *RegExr.* [Online] https://regexr.com/2rhq7.

15. **Schlawack, Hynek.** argon2-cfii: Argon2 for Python. *pypi.org.* [Online] Python Software Foundation, 15 August 2023. https://pypi.org/project/argon2-cffi/.

16. **PG Online.** *Tackling A Level Project in Computer Science AQA 7517.* Dorset : PG Online Limited, 2020. 978-1-910523-20-9.