Controlled Assessment

Component 3

Computing

**Candidate Name:** Alexander Dowsett

**Candidate Number:** 4131

**Centre Name:** Mayflower High School

**Centre Number:** 16117

Table of Contents

[Analysis 5](#_Toc4795509)

[Discussion 5](#_Toc4795510)

[Introduction 5](#_Toc4795511)

[Background 5](#_Toc4795512)

[Feedback 5](#_Toc4795513)

[Limitations 6](#_Toc4795514)

[Investigation 7](#_Toc4795515)

[Research Outcome 7](#_Toc4795516)

[Questionnaire & Observation 7](#_Toc4795517)

[Interview 8](#_Toc4795518)

[Existing solutions 8](#_Toc4795519)

[Old system’s inputs and outputs 9](#_Toc4795520)

[Research conclusion 9](#_Toc4795521)

[Aims 10](#_Toc4795522)

[What will the program do? 10](#_Toc4795523)

[Who is the target audience? 10](#_Toc4795524)

[What hardware will it need? 10](#_Toc4795525)

[What software will it need? 10](#_Toc4795526)

[How will the user interact with the program? 10](#_Toc4795527)

[What will the program aim to achieve? 11](#_Toc4795528)

[Success Criteria 11](#_Toc4795529)

[Objectives 11](#_Toc4795530)

[Design 13](#_Toc4795531)

[Interface 13](#_Toc4795532)

[GUI Designs 13](#_Toc4795533)

[File structure 13](#_Toc4795534)

[Variable table 13](#_Toc4795535)

[Validation 13](#_Toc4795536)

[Names 13](#_Toc4795537)

[Phone numbers 13](#_Toc4795538)

[Passwords 13](#_Toc4795539)

[Boolean fields 14](#_Toc4795540)

[Rate 14](#_Toc4795541)

[Price 14](#_Toc4795542)

[Flow Chart 15](#_Toc4795543)

[Program Code 17](#_Toc4795544)

[Prototypes 70](#_Toc4795545)

[Prototype 1 70](#_Toc4795546)

[Prototype 2 70](#_Toc4795547)

[Prototype Refinement 70](#_Toc4795548)

[Prototype 1 70](#_Toc4795549)

[Prototype 2 73](#_Toc4795550)

[Prototype Testing 73](#_Toc4795551)

[Maintenance Documentation 76](#_Toc4795552)

[Table of Classes 76](#_Toc4795553)

[Table of Routines/Functions 77](#_Toc4795554)

[Table of Variables 82](#_Toc4795555)

[Testing 91](#_Toc4795556)

[Testing strategy 91](#_Toc4795557)

[Pre- Planned Tests 91](#_Toc4795558)

[Evaluation 95](#_Toc4795559)

[Test results 95](#_Toc4795560)

[Objectives and my experience while programming 95](#_Toc4795561)

[Success Criteria 95](#_Toc4795562)

[Using Python over other programming languages 97](#_Toc4795563)

[Existing solutions 97](#_Toc4795564)

[Future Improvements 97](#_Toc4795565)

[What I could have done differently 98](#_Toc4795566)

[What I could improve in the future 98](#_Toc4795567)

[User Documentation 99](#_Toc4795568)

[Python 99](#_Toc4795569)

[Installation 99](#_Toc4795570)

[How to open/edit the program 101](#_Toc4795571)

[SQL DB Browser (optional) 102](#_Toc4795572)

[Installation 102](#_Toc4795573)

[Opening SQL DB Browser 103](#_Toc4795574)

[How to open database 104](#_Toc4795575)

[Viewing data within the database 104](#_Toc4795576)

[Editing data within database 105](#_Toc4795577)

[Booking system 107](#_Toc4795578)

[Installation 107](#_Toc4795579)

[Login 107](#_Toc4795580)

[Navigation 107](#_Toc4795581)

[Appendix 111](#_Toc4795582)

[Appendix Order 111](#_Toc4795583)

[Appendix A 111](#_Toc4795584)

[Appendix B 112](#_Toc4795585)

[Appendix C 113](#_Toc4795586)

[Appendix D 114](#_Toc4795587)

[Appendix E 115](#_Toc4795588)

# Analysis

## Discussion

### Introduction

My aim for this project to create a system that simplifies the calculation and booking processes into one long-lasting, easy and reliable computerised system.

I decided to choose this particular company based on the fact they still use a pen and a book for their booking process. Hopefully the introduction of a computer system will speed up this booking process as well as simplify it. Staff will no longer need to scan through pages of appointments but rather quickly navigate to the day of the appointment, and then the appointment itself. The system will be accessed through a user-friendly GUI for staff. I will attempt to simplify the system as much as possible, while including all essentials features, to provide a good balance between functionality and accessibility. It’s also important to establish a easy and reliable system so staff feel comfortable with the system.

### Background

Based in Billericay High Street, Essex, this hairdressers has been using a pen and a book for booking appointments. Jim Shaw is the manger of this business and has agreed to work together to improve this process which should benefit the company overall. We will communicate primarily through emails, however when possible, meet and discuss outside of working hours to avoid interfering and potentially hurting the business and profits. When meeting we will discuss any adjustments I can make to ensure the system is most suitable for the company. The very high majority of haircuts are booked ahead of time, sometimes weeks so the company heavily relies on their booking system.

### Feedback

After receiving some feedback from my peers, we discussed improvements that could refine some problem of the current system.

Databases can be scary for the average user so the program should include very simple ways for users of the program to add clients and appointments. The user should only be asked only essential details about the client and be able to easily enter it into entry boxes, followed by a button add it to the database. This also allows me to add validation to ensure the fields are correctly formatted and detect duplicates to keep data redundancy low. Validation could include checking client’s names only contain alphabetical characters and phone numbers only contain digits and are the correct length. Editing the database directly has very little to no validation.

Booking appointments should also be made as easy and quick. The user should not have to enter information that the program should already know. For example, the program can estimate the duration of an appointment based on the type of service the client of the company is booking. The user is still free to edit the duration in case the appointment is going to take longer than expected. Again, adding an appointment through the program allows heavy validation to remove data redundancy and stop inputted data being in incorrect datatypes and formats.

Someone suggested some sort of loyalty system. There could be a reward system introduced where customers can earn points or a discount for haircuts and therefore get cheaper haircuts in the future. This will make loyal clients feel appreciated. The ability to set custom prices for haircuts will also allow offers to be added to attract new clients.

I will have to discuss with Jim about what approach would be preferred on rewarding loyal clients and attracting new ones.

### Limitations

Although the program will feature easy-to-use features and will be as simple as possible, without losing any functionality, some people will always prefer a pen and a book to a computerised system.

The computerised system also brings all the other problems with computers with it. Rare chance of database corruption, dependency on the computer working during all working hours. However, some of these limitations can be avoided by having a backup database stored offsite and a secondary computer ready to go. Switching to a computerised system will eliminate many old problems with the paper book system.

During the programming phase, I will need to learn how to allow SQL and Python to work together. I’m planning on using the mysqlite3 module of Python. However, this will introduce new functions to learn. Other new challenges include finding a suitable way to display the appointments. This could be achieved by stacking Tkinter list boxes vertically, or by using the Tkinter tree view table.

# Investigation

## Research Outcome

It was important to analyse the current system to understand completely how it works within the company. Through the analysing process, I learnt what problems affected the company by the system and fully how the system works. I decided to firstly complete an Observation to gain basic knowledge of the system in action. Here I could see any problems encountered by staff and see if they used the system as intended. This also ensured a truthful, unbiased first look at the system.

I decided on 3 methods of investigation completing Observation, Questionnaires and Interviews. Document collection interferes with the company too much so could not be completed. It also reveals private client information that could potentially breach the Data Protection Act.

Below are the following results that were revealed after collecting my questionnaire results.

### Questionnaire & Observation

I was able to observe someone using the booking system in an everyday working scenario and plan my questionnaire questions accordingly. I ran the census with 8 people who had previously used or currently use the booking system. After I got the questionnaire results, I was able to investigate further and expand on the results in an interview with Jim. See Appendix A for observation notes and Appendix B for the questionnaire given to users of the previous booking system.

1. I feel comfortable with adding appointments to the book.

1 said disagree, 6 said agree and 1 said strongly agree. It was obvious that the majority of the staff were relatively comfortable with using a book and pen to add appointments. But there was major room for improvement.

1. I can easily navigate through the book to find dates.

Similar to the question above, finding a date or time in a record book in chronological order was not difficult. 5 said agree and 3 said strongly agree. A similar system of chronological order should be implemented in my computer system.

1. I can easily remind clients what date they have booked their appointments.

3 said strongly disagree, 5 said. In an interview with Jim, he stated that sometimes clients couldn’t even remember what day they booked their appointment. Searching through the book was frustrating and time-consuming scanning sometimes dozens of pages to find a client’s name. This made it apparent that navigation was the biggest problem with the current system.

1. I can easily remove and edit appointments to the book.

1 said strongly disagree and 4 said disagree and 3 said agree. It became apparent that editing the book was also frustrating as staff would have to use a rubber and pencil to edit the book which is not ideal and sometimes messy. This is another main problem with the system.

1. I feel comfortable with calculating prices for haircuts.

4 said agree and 4 said strongly agree. When asking a member of staff, they stated that it was hard at first however learning the prices was easy over time, however with a computerised system it would eliminate the risk of the user getting the price wrong, as it will be right in front of them when making the appointment.

1. I favour a computerised booking system over a book and pen.

1 said disagree, 1 said agree and 6 said strongly agree. It is obvious that a computerised system is mostly preferred and its negatives outweigh the negatives of the book. I managed to ask the person who disagreed and it was simply because they disliked a change, especially when using book and pen for such a long time.

1. Overall, how happy are you with using a book to record bookings?

The answer for this question could be answered between 1 and 10. The results were as followed, 5,5,5,6,7,7,8,10 out of 10. This gives us an average of 6.6 out of 10. Therefore, it’s clear that the majority believe the current system is not perfect and has plenty of potential improvements.

### Interview

While conducting an interview I asked three main questions and followed with sub-questions to further expand on their answer. I used the interview to answer some of the ‘why’ questions I had developed after the questionnaire results. Below are some summarised points noted from the interview.

1. **Do you feel there would be any potential limitations with a computerised booking system?**

* Some people automatically dislike the idea of using a computer.
* Reliance on technology.

1. **What features that are not in the current booking system would you like to see in the new one?**

* Calculation of price.
* Much easier to navigate and find appointments with a search bar.
* Consistent formatting.
* Bigger record, a computer database is more timeless than booking books.
* Easier cancellation and rebooking in same slot.
* A secure system.

1. **What is one of the biggest problems you have with the current booking system and how could this be improved with a computerised system.**

* Cancelling multiple appointments can become messy and confusing.
* Also finding client’s lost appointments times is extremely time-consuming and can take over minutes to find their appointment.
* Can be improved by allowing searching client names with a search bar.

### Existing solutions

#### Microsoft Excel

Microsoft Excel is an alternative option as it can be used as a database. Excel is a spreadsheet program however can simulate a database with Python. Excel does however have potential problems.

##### Pros

Excel is inexpensive and has an online version which is free. The program also offers benefits such as automatic recalculations using macros. Excel allows formatting of cells and visual representation such as graphs, which can be useful when used independently, without Python, to show data such as each hairdresser’s value in appointments over time. Excel also offers automatic sorting and filtering.

##### Cons

Excel isn’t very good at handling large amounts of data, slow to open and load, however can be avoided using multiple spreadsheets. No data integrity and no protection for data corruption unlike some paid database software. No automatic backups included, manual backup or development of an automatic backup would be essential to ensure the data is protected.

Some of the pros such as macros, formatting and searching require very advanced knowledge of the program already and isn’t very user-friendly. Spreadsheets will become very crowded very quickly.

Therefore, it’s apparent that excel is not the ideal solution for long-term data storage as the limited file size and slow loading speeds will soon become a problem. Also, with no data integrity, the risk of losing the data can have a massive potential economic impact on the company.

#### SystemBookings

##### Pros

Trusted History, used by the NHS, EDF Energy, Dulux, Halfords and more. Security and backup database. Support is provided including telephone and live chat. No continuous payment for support, one-time fee. Very experienced system, been making systems for over 9 years.

##### Cons

The main and biggest con is the price. So much is provided within this booking system however this comes with a cost. This type of plan is very rigid, not as easy to make structural changes once the database is set up.

### Old system’s inputs and outputs

The old system that included an appointment book. When a client of the company called or visited the hairdressers to make an appointment the following information was taken; name of client, time and date of appointment, name of hairdresser and type of service they want (just a haircut or colouring, etc.). Additionally, if it’s the first appointment the client has had, more information such as contact information will be asked on the phone or in-person on the day of the appointment. Once a client would turn up for their appointment a member of staff would then check the book for the appointment. After the appointment the staff member would mark the appointment as attended and paid. Therefore, any time-wasters that frequently do not turn up to appointments can be denied appointments in the future. The current system has minimal processes and whatever inputted is just recorded for viewing at a later date.

Since it was a very basic paper system, whatever the user inputted into the appointments book is ‘outputted’ when the user looks for appointments.

The layout of the book is pretty standard and had a page dedicated to each day, with times going downwards on the left. The book can get very cluttered with appointments on a busy day. The company used a very similar daily appointments book to the one in Appendix D, with client names and the corresponding time and the hairdresser’s initials to save on space in the book.

### Research conclusion

Through observation, questionnaire results and the interview I have come to the following conclusion. The booking system has some major limitations and is in need of improvement especially in the area of accessibility, as the navigation through a paper-based system is time-consuming and frustrating. Other standard problems with a paper-based system such as risk of double booking would also be avoided, and other problems highlighted in during research.

## Aims

### What will the program do?

The program aims to make booking appointments quick and easy. The user should be able to book an appointment in a matter of seconds and without having to think how to do it. The user should also be able to find existing appointments in similar time and therefore be able to quickly navigate throughout the system.

Admins will have control of databases and have the ablilty to add, remove and edit hairdresser, staff, haircut and client information through tables browser within the program. This will have heavy validation to ensure the any values entered are correct before being updated in the database. This ensures no issues will occur later while using the program, such as TypeErrors.

The basic security in place will require users to log in before use. The program will allow users to change their password once logged it. This also allow admins to keep track of when and who logs in and if they add, edit or remove any appointments. Any changes will be printed to the console.

### Who is the target audience?

Any staff that work on the front desk, which can be determined by the manager, Jim . This could range from hairdressers duing their less busy periods to other staff. The program will be designed to be as used by people with basic computer knowledge and a standard GUI layout. The program is aimed to be as user-friendly as possible, so no command line.

### What hardware will it need?

The program will be lightweight and therefore only require the Python IDLE 3.4.2 requirements. Below is a basic idea of the minimum requirements from Intels website for Python IDLE 2.7.X to 3.6.X.

* Processors: Intel Atom® processor or Intel® Core™ i3 processor
* Disk space: 1 GB
* Operating systems: Windows\* 7 or later, macOS, and Linux
* Python\* versions: 2.7.X, 3.6.X
* Included development tools: conda\*, conda-env, Jupyter Notebook\* (IPython)
* Compatible tools: Microsoft Visual Studio\*, PyCharm\*
* Included Python packages: NumPy, SciPy, scikit-learn\*, pandas, Matplotlib, Numba\*, Intel® Threading Building Blocks, pyDAAL, Jupyter, mpi4py, PIP\*, and others.

(Intel®, n.d.)

These requirements do not cover the Tkinter package, however the requirements shouldn’t be much different to the standard Python requirements. They are also very minimal so even very budget computers shouldn’t have a problem running the booking system.

### What software will it need?

The program will need Python IDLE 3.4.2 installed. If advanced maintenance is performed a SQL database browser would be ideal if the engineer wishes to perform any changes to the database without the program, however there is no validation with an external SQL Browser.

### How will the user interact with the program?

The users will be presented with a login window as a basic layer of security. They will be required to enter their user ID and password. For the user it will be as simply as clicking on the entry boxes typing their user ID and password then clicking the login button or by pressing enter on their keyboard.

If the login is successful then the main window will display all the appointments for the day and simplying clicking on an appointment slot will bring up a small window allowing them to add a new appointment. Clicking on a booked appointment slot will bring up a similar small window to view and edit the appointment. A settings button will be available for admins to alter the database within the program and for users to change their password.

I will try to make the program’s GUI very visually standard and similar to other popular programs so users feels comfortable using the booking system. Hopefully prototyping and testing can help achieve a GUI design that everyone is happy with.

### What will the program aim to achieve?

A method for staff to book appointments in a simple and quick way. Only relevant, neccessary information will be asked about the appointment and client.

## Success Criteria

The program will need to achieve the following criteria to be successful and meet my client’s requirements. It should:

* Provide a basic security system so unauthorised people cannot access the system while unoccupied.
* Allow users to view all existing appointments.
* Allow users to add appointments and ask only the relevant information. This includes any day or week.
* Display to the user the price of the appointment in advance, taking into account the type of service being provided and which hairdresser is providing it.
* Allow users to mark appointments as paid.
* Allow users to remove existing appointments.
* Allow users to add clients to the database in a user-friendly way.
* Allow users to change their password.
* Allow admins to access and view the tables located the database within the Python program.
* Allow admins to add, edit and remove records in the database within the Python program.

## Objectives

In order to achieve my success criteria I need to use Tkinter with Python to achieve a user friendly GUI. I will need to learn how to define and manager a Tkinter window and populating it with Tkinter widgets. Labels are important to use to output information to the user. Buttons are important to allow the user to be able to interactive with the GUI. Buttons could lead to other windows such as settings or a window for users to add clients. Entry boxes are important to allow the user to enter data into the program. The user may need to enter client information where this would be approiate. Other widgets such as Tkinter combo boxes can be used as a drop-down menu so users can select an option from a list of options. This will be useful when the user needs to make an appointment and can select the hairdresser and haircut from a list/drop-down menu. I also need to be able to display appointments to the user in a table. A Tkinter treeview could be used which is a widget with columns, rows and cells. On a busy day there may be too many appointments to fit on the screen so a scroll bar could be used to allow the user to scroll vertically through the table. All this widgets have unique attributes that can be altered to achieve what I plan out to achieve. While making this program, I will be able to experiment and test these widgets and their properties to customise them to be what I need for the system. I will also be able to use a similar table to replicate a database browser within the program. This is necessary for admins to add, edit and remove hairdressers and haircuts. The table will need to be interactivable to make editing it easier. This could be achieved by defining an entry box over the cell clicked in the table then setting the user’s cursor in the entry box. This gives the illision that the user is editing the table directly but actually just typing into an entry box. When the user clicks away, the contents of the entry box can be moved into that cell and the entry box can be destroyed.

I will need to also use the SQLite3 module of Python to allow communication between Python and a database. Therefore, I will need to learn how the functions that come with SQLite3 and how to usitilise them. Using my knowledge of SQL from A-Level computer science I can use SQL querys to select, update, remove records from the database and add condition to the queries.

I will use object-oriented programming in Python where I will define a class for each window in the program. This will help keep track of which functions and code belong to what class. This also will allow windows to be summoned by simply creating an instance of that class.

# Design

## Interface

The user will have to open the booking system by double-clicking the program.py as long as they the correct Python interpreter installed. This will launch a login window that they must login with their ID and password. The interface is simple and not overcrowded while maintaining all the essentials features to a booking system. The user will simply click on an appointment slot to get more details. Depending on whenever the slot is already booked, affects the information that is displayed. The user can simply cancel an appointment at the click of a button and book a new one in a few clicks.

### GUI Designs

The GUIs can be found in the Prototypes section fully annotated where the first GUI design is for Prototype 2 and the second GUI design is for the final version.

## File structure

All the files required for the program to run will be included in a program folder. Inside the program folder there will be a Python (program.py) file which can be run through Python IDLE. There will also be a /resources/ folder which includes images, icons and other resources used in the program. Thirdly, there will be a database (.db) file which will be prefilled inside the /db/ folder. When the application is used in a real-life scenario, the placeholder database should be deleted so the client can add the relevant information for his company. This may include the hairdressers, types of services (haircuts) and prices of his choosing. When the database file is deleted, the next time the program is run it will generate an empty database with all the relevant tables and fields as well as a default admin login for the first login.

## Variable table

In this variable table are all the variable I except to use in at least my prototypes. The actually variables may vary in the final code.

(Click here to go to the table of variables that are used in the final code).

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| first\_appointment\_time | String | Holds the first appointment time which allows the system to easily adapt to a different opening time if necessary. This time is in the format of hh:mm. |
| last\_appointment\_time | String | Holds the last appointment time which allows the system to easily adapt to a different closing time if necessary. This time is in the format of hh:mm. |
| appointment\_intervals | Integer | Holds the intervals in minutes of each appointment slot. This must be in a factor of 15 and is 15 by default. This allows the system to adapt if the company wants to transfer to 30 minute appointments. Each appointment is not limited to 15 as appointments can be booked over multiple slots. |
| working\_days | Array of integers | An array that contains the days the company operates where Monday is 1 and Sunday is 7. E.g. [2, 3, 5] would translate to Tuesday, Wednesday and Friday. |
| bg\_colour | String | A string that stores a colour in a hexadecimal format. This value can be changed to change the colour of all the widgets using this colour. |
| text\_colour | String | A string that stores a colour in a hexadecimal format. This value can be changed to change the colour of all the text of this colour. |
| primary\_colour | String | A string that stores a colour in a hexadecimal format. This value can be changed to change the colour of all the widgets using this colour. |
| secondary\_colour | String | A string that stores a colour in a hexadecimal format. This value can be changed to change the colour of all the widgets using this colour. |
| primary\_font | String | A string that stores the name of a font. This can be changed to change the font of all the text using this font. |
| secondary\_font | String | A string that stores the name of a font. This can be changed to change the font of all the text using this font. |
| login\_window | Login (class instance) | An instance of the class Login. |
| root | Tkinter window | A Tkinter window that will be used for the login window. This window is then used in the ‘login\_window’ class instance. |
| main\_window | Main (class instance) | An instance of the class Main. |
| main | Tkinter window | A Tkinter window that will be used for the main window. This window is then used in the ‘main\_window’ class instance. |
| settings\_window | Settings (class instance) | An instance of the class Settings. |
| sett | Tkinter window | A Tkinter window that will be used for the settings window. This window is then used in the ‘settings\_window’ class instance. |
| add\_app\_window | AddAppointment (class instance) | An instance of the class AddAppointment. |
| add\_app | Tkinter window | A Tkinter window that will be used for the add appointments window. This window is then used in the ‘add\_app\_window’ class instance. |
| x | Integer | Holds the x co-od of a Tkinter window. |
| y | Integer | Holds the y co-od of a Tkinter window. |
| topLabel | Tkinter label | A Tkinter label which is used as a header. |
| bottomLabel | Tkinter label | A Tkinter label which is used as a footer. |
| imageLabel | Tkinter label | A Tkinter label which holds an image used in the header. |
| title\_small | Tkinter photo image | A Tkinter photo image that holds the photo of the company’s title at a small size (281x50). |
| title\_large | Tkinter photo image | A Tkinter photo image that holds the photo of the company’s title at a larger size (382x68). |
| staffID | Tkinter string | A Tkinter string variable that stores the ID inputted by the user. |
| password | Tkinter string | A Tkinter string variable that stores the password inputted by the user. |
| staffIDLabel | Tkinter label | A Tkinter label that displays which next to an entry box letting the user know the information to enter there. |
| passwordLabel | Tkinter label | A Tkinter label that displays which next to an entry box letting the user know the information to enter there. |
| staffIDEntry | Tkinter entry box | A Tkinter entry box that allows the user to enter their ID. |
| passwordEntry | Tkinter entry box | A Tkinter entry box that allows the user to enter their password. |
| loginButton | Tkinter button | A Tkinter button that calls the function Login.login. |
| helpLabel | Tkinter label | A Tkinter label that is clickable. It calls the function Login.login\_info. |
| conn | SQL connection | The variable used to store the connection to the database. |
| c | SQL cursor | The variable used to store the cursor in the database. |
| num\_of\_staff | Integer | Temporary variable that holds the total number of staff within the database. |
| ID | String | Holds the retrieved value from the Tkinter string variable StaffID. |
| p | String | Holds the retrieved value from the Tkinter string variable Password. |
| users\_details | Array of strings | Holds all the details of the logged in user for later use. |
| is\_admin | String | Holds either ‘User’ or ‘Admin’ depending on if the logged in user has admin privileges. |
| settings\_light | Tkinter photo image | Holds the lighter verison of the settings icon in a Tkinter photo image. |
| settings\_dark | Tkinter photo image | Holds the darker verison of the settings icon in a Tkinter photo image. |
| settingsLabel | Tkinter label | Holds either of the settings icons in a label displayed at the top left of the window. This label is clickable and will call Main.settings. |
| tab\_font | Tkinter font | Holds the font ‘font\_1’ of font size 12. This is so it can be used in the notebook tabs by inserting it into a theme. |
| style | Tkinter style | Holds the style that is used in main window. |
| current\_theme | Tkinter theme | Holds the theme in use so it can be edited to insert the ‘tab\_font’ and padding into the notebook tabs. |
| hairdressers | 2D array of integers and strings | Holds all the hairdressers and details relating |
| i | Integer | A temporary variable that is used for couting in for loops. |
| j | Integer | A temporary variable that is used for couting in for loops. Used when i is in use. |
| k | Integer | A temporary variable that is used for couting in for loops. Used when i and k are in use. |
| l | Integer | A temporary variable that is used for couting in for loops. Used when i, j and k are in use. |
| temp | Dynamic (constantly changing between string, interger and boolean) | This variable was purely used for holding values temporary in calculations. Therefore it doesn’t have a constant datatype. This is useful as it’s something necessary to hold a number of string for a short period of time to then be printed or used in another calculation. |
| match | Integer or array of integers | Used to collect matched hairdresser names so the program can add surname letters so each hairdresser’s name is unique in the notebook. It is used in Main.update\_table as a Integer to check if there are appointments in the appointment slots. |
| appointments\_notebook | Tkinter notebook | The notebook that holds the table that contains all appointments. It has a tab for each working day and each tab contains the frame in the array pages with index equal to the day. |
| pages | Array of Tkinter frames | An array of frames that each contains a list box for each hairdresser as well as the frame ‘namesFrame’. |
| list boxes | 2D array of Tkinter list boxes | A 2D array of list boxes with the first index for the day and the second index for the hairdresser. |
| namesFrame | Array of Tkinter frames | An array of frames that contain a number of canvases for that page. This is within a frame to allow the size of the canvases to be altered easily. |
| names | 2D array of Tkinter canvases | A 2D array of list boxes with the first index for the day and the second index for the hairdresser. The canvases contain vertical text containing the hairdressers’ names. |
| scrolls | Array of Tkinter scrollbars | An array of scrollbars so there is one for each page (day). The scrollbars are linked to the list boxes so they can be scrolled through. |
| days | Array of strings | An array that is a copy of working\_days except the date are stored in a datetime module format instead of a day relative to the week. |
| clock | Tkinter photo image | A Tkinter photo image that contains a small clock icon used in the heading for time. |
| date | String | Contains the current date in datetime module format. |
| searchBar | Tkinter entry box | A Tkinter entry box that is used for searching appointments, clients, hairdressers in the database. |
| search\_icon | Tkinter photo image | A Tkinter photo image that contains a small magnifying glass icon used in the heading for time. |
| searchIconLabel | Tkinter label | A Tkinter label that contains the search\_icon image. |
| clockLabel | Tkinter label | A Tkinter label that contains the current time in a standard digital format. |
| dateLabel | Tkinter label | A Tkinter label that contains the current date in a standard date format. |
| weekLabel | Tkinter label | A Tkinter label that contains the week including the first date in working\_days and the last. |
| welcomeLabel | Tkinter label | A Tkinter label that contains a welcome message that contains the users name and if they are an admin or user. |
| current\_time | String | A variable that contains last updated time in datetime module format. |
| current\_day | String | A variable that contains last updated date in datetime module format. |
| new\_time | String | A variable that contains the current time in datetime module format. |
| new\_day | String | A variable that contains the current date in datetime module format. |
| day | Integer | A integer that contains a number between 1 and 31. |
| suffix | String | A suffix such as ‘st’ ‘nd’ ‘rd’ or ‘th’ that is related to the ‘day’ variable. |
| start\_time | String | Contains the appointment time for the appointment slot when updating the table. |
| clients | 2D Array of integers and strings | A 2D array that contains the client ID, first name and last name for each client. |
| client\_names | Array of strings | An array that contains all the client names where each value is the client’s first name combined with the last name. |
| client | Integer | An integer that holds the client ID of the selected client. |
| clientLabel | Tkinter label | A Tkinter label that contains some text indicating which combobox can be used to select the client. |
| clientBox | Tkinter combo-box | A Tkinter combobox that contains all the clients’ names in the database. The user can select a client from the list. |
| hairdressers | 2D Array of integers and strings | A 2D array that contains the hairdresser ID, first name and last name for each hairdresser. |
| hairdresser\_names | Array of strings | An array that contains all the hairdressers names where each value is the hairdresser’s first name combined with the last name. |
| hairdresser | Integer | An integer that holds the hairdresser ID of the selected hairdresser. |
| hairdresserLabel | Tkinter label | A Tkinter label that contains some text indicating which combobox can be used to select the hairdresser. |
| hairdresserBox | Tkinter combo-box | A Tkinter combobox that contains all the hairdressers’ names in the database. The user can select a hairdresser from the list. |
| haircuts | Array of integers, floats and strings | A 2D array that contains the haircut ID, haircut name, price and estimated time for each haircut. |
| haircuts\_names | Array of strings | An array that contains all the haircut names. |
| haircut | Integer | An integer that holds the haircut ID of the selected haircut. |
| haircutLabel | Tkinter label | A Tkinter label that contains some text indicating which combobox can be used to select the haircut. |
| haircutBox | Tkinter combo-box | A Tkinter combobox that contains all the haircuts’ names in the database. The user can select a haircut from the list. |
| durations | Array of integers | An array of all the durations possible for the haircut. |
| duration | Integer | An integer that holds the duration of an appointment to be added. |
| durationLabel | Tkinter label | A Tkinter label that contains some text indicating which combobox can be used to select the duration. |
| durationBox | Tkinter combo-box | A Tkinter combobox that contains all the possible durations. The user can select a duration from the list. |
| addButton | Tkinter button | A button that when pressed runs the function add\_appointment. |

## Validation

Validation is important to ensure no unexpected values are entered into the program. Each time the user is asked to input data the program will heavily validate that data before considering to put it into the database. Data of an incorrect datatype or format could result in errors in the program later on and in extreme cases therefore corrupt the database. The program will validate the following data:

### Names

Names will be checked to ensure they only contain alphabetical characters. The upper case and lower case of the name will not matter as the program will automatically lowercase the string and capitalise the first character.

### Phone numbers

Phone numbers will be checked to ensure they only contain digits. It is important however phone numbers are not stored as an integer as if the first character is a zero, it will be forgotten. Therefore, its stored as a string in the database. It will also only be valid if the phone number contains between ten and eleven characters.

### Passwords

Passwords are able to contain almost anything however the program will deny the password if it does not deem it strong enough. The password must contain at least one digit and one alphabetical character. It also must at least eight characters long.

### Boolean fields

Boolean fields such as if the staff member is an admin and if they are a hairdresser only accept certain values. If the boolean will accept ‘false’, ‘0’ and ‘no’ if false regardless of capitalisation. The boolean will accept ‘true’, ‘1’ and ‘yes’ if true regardless of capitalisation.

### Rate

Rate will be checked to see if it’s an integer. It also must be positive. This value is the percentage multiple of the price of the haircut, where 100 is 100%.

### Price

Price will be checked to be a float value where the float is in terms of pounds. The float can only go to two decimal places, so the user cannot pay fractions of a penny.

All valid data will then be updated into the database. Data such as Rate and Price will be involved in calculation first, then result in the database.

During my programming phase I will perform white boxing for every new code for validation. Once prototypes are created, I will perform some black box testing to ensure no unexpected inputs are falsely validated.

## Flow Chart



# Program Code

1. version = '1.0'
2. #                               Created by Alex Dowsett
3. #--------------------------------------------------------------------------------------
4. #                        CONFIGURATION (READ USER GUIDE FIRST!)
5. first\_appointment\_time = '07:00' # The time of the first appointment of the day in a 'hh:mm' format.
6. last\_appointment\_time = '20:00' # The time of the last appointment of the day in a 'hh:mm' format.
7. appointment\_intervals = 15 # Time in minutes for each appointment slot. If this value is changed, keep it as a factor of 15.
8. working\_days = [1, 2, 3, 4, 5, 6] # Where Monday = 1, Tuesday = 2... Sunday = 7. For example: If the working days are Monday, Wednesday and Sunday then workings\_days = [1, 3, 7].
9. bg\_colour = '#E7E8EA' # Blackground colour **in** hexadecimal.
10. text\_colour = '#343038' # Text colour **in** hexadecimal.
11. primary\_colour = '#A09D9C' # 1st colour **in** hexadecimal.
12. secondary\_colour = '#343038' # 2nd colour **in** hexadecimal.
13. primary\_font = 'Helvetica' # A font to be used primarily.
14. secondary\_font = 'Courier' # A font to be used secondarily.
15. tertiary\_font = 'Trebuchet MS' # A font to be used tertiarily.
16. database\_file = 'bookings.db' # Name of database file.
17. main\_window\_zoomed = True # If 'True' the main window will be zoomed on startup.
18. main\_window\_width\_relative\_size = 0.95 # The percentage of the width that the main window will fill on startup. Only applies if main\_window\_zoomed = False.
19. main\_window\_height\_relative\_size = 0.90 # The percentage of the height that the main window will fill on startup. Only applies if main\_window\_zoomed = False.
20. #
21. #--------------------------------------------------------------------------------------
23. # IMPORTS
24. **import** tkinter as tk # Imports the tkinter module. Any functions from this module will be start with a 'tk.' to show what module it came from.
25. **import** tkinter.ttk as ttk # Imports the tkk part of the tkinter module. Any functions from this module will be start with a 'ttk.' to show what module it came from.
26. **from** tkinter **import** messagebox # Imports tkinter messageboxes seperately otherwise they will encounter errors when being used outside of Python IDLE.
27. **import** sqlite3 # Imports the sqlite3 module. This module is used to communicate with the datebase file (datebase.db).
28. **import** time # Imports the time module.
29. **import** datetime # Imports the datetime module.
31. # VARIABLES
32. root = tk.Tk() # Defines the variable 'root' as a tkinter window.
33. screen\_width = root.winfo\_screenwidth() # Defines a new integer that holds the width of the screen in pixels.
34. screen\_height = root.winfo\_screenheight() # Defines a new integer that holds the height of the screen in pixels.
35. main\_window\_width = round(screen\_width \* main\_window\_width\_relative\_size) # Calculates the inital width of the main window in pixels.
36. main\_window\_height = round(screen\_height \* main\_window\_height\_relative\_size) # Calculates the inital height of the main window in pixels.
37. first\_appointment\_time = int(first\_appointment\_time.split(':')[0]) \* 60 + int(first\_appointment\_time.split(':')[1]) # This calculation converts the time hh:mm to minutes from midnight.
38. last\_appointment\_time = int(last\_appointment\_time.split(':')[0]) \* 60 + int(last\_appointment\_time.split(':')[1]) # This calculation converts the time hh:mm to minutes from midnight.
40. # GLOBAL FUNCTIONS
41. **def** main():
42. '''''This function is run when the program starts.'''
43. login\_window = Login(root) # Creates an instance of the class Login with the tkinter window 'root' as a parameter.
44. root.mainloop() # Wait until the tkinter window 'root' is closed.
46. **def** add\_date\_suffix(day):
47. '''''This function adds the corresponding date suffix to the end of a number.'''
48. day = int(day) # Converts variable to integer.
49. suffix = '' # Defines a string called 'suffix'.
50. **if** 4 <= day <= 20 **or** 24 <= day <= 30: # If variable 'day' in the stated range.
51. suffix = 'th' # Change 'suffix' value.
52. **elif** day **in** {1, 21, 31}: # If variable 'day' is one of the following values.
53. suffix = 'st' # Change 'suffix' value.
54. **elif** day **in** {2, 22}: # If variable 'day' is one of the following values.
55. suffix = 'nd' # Change 'suffix' value.
56. **elif** day **in** {3, 23}: # If variable 'day' is one of the following values.
57. suffix = 'rd' # Change 'suffix' value.
58. **return** str(day) + suffix # Return 'day' followed by the suffix as a string value.
60. **def** log(statement):
61. '''''This function simply prints a string with the date and time as a prefix.'''
62. **print**('[{}] {}'.format(time.strftime('%d/%m/%y|%H:%M:%S'), statement)) # Prints message to the console.
64. **def** db\_open():
65. '''''This function opens the connection and cursor to the database.'''
66. **global** c, conn # Makes these variable global and therefore accessible in all functions.
67. conn = sqlite3.connect('db/' + database\_file) # Starts a connection to the database named 'bookings.db' within the /db/ folder.
68. c = conn.cursor() # Defines a cursor to interact with the database.
70. **def** db\_close():
71. '''''This function closes the connection and cursor to the database.'''
72. c.close() # Stops interaction.
73. conn.close() # Closes connection with database.

76. #======================================================================================
77. #                                        LOGIN WINDOW
78. #======================================================================================
79. **class** Login: # Defines a class that will represent the login window.
80. '''''This class is a login window for security. A username and password is required to access the system.''' # Class Information.
81. **def** \_\_init\_\_(self, master): # Initializes this function when an instance of the class is created.
82. '''''This function is run when the class is initialised.'''
83. ### VARIABLES
84. self.master = master # Defines the varaible master as an attribute to self (main\_window).
85. self.id\_var = tk.StringVar() # Defines a tkinter string variable that stores the staff ID as an attribute of self (main\_window).
86. self.p\_var = tk.StringVar() # Defines a tkinter string variable that stores the password as an attribute of self (main\_window).
88. ### TKINTER WINDOW CONFIG
89. self.master.title('Essensuals Booking System Login (Version {})'.format(version)) # Sets title of the window.
90. self.master.geometry('540x360+{}+{}'.format(((screen\_width-540)//2), ((screen\_height-360)//2)))  # Sets window geometry and centers the window.
91. self.master.resizable(False, False) # Disables resizing for the window.
92. self.master.wm\_iconbitmap('resources/icon.ico') # Sets window icon.
93. self.master.config(bg=bg\_colour) # Sets background colour of window.
95. ### TKINTER LABELS
96. self.header\_label = tk.Label(self.master, bg=primary\_colour) # Defines a label.
97. self.header\_label.place(x=0, y=0, w=540, h=60) # Sets the geometry of the label.
99. self.footer\_label = tk.Label(self.master, bg=secondary\_colour, fg='#FFFFFF', text='Programmed by Alex Dowsett', font=(primary\_font,  7)) # Defines a label.
100. self.footer\_label.place(x=0, y=325, w=540, h=35) # Sets the geometry of the label.
102. self.title\_image = tk.PhotoImage(file='resources/title\_small.png') # Defines a tkinter variable that contains an image.
103. self.title\_label = tk.Label(self.master, bg=primary\_colour, image=self.title\_image) # Defines a label that will contain the image.
104. self.title\_label.place(x=240, y=5, w=281, h=50) # Sets the geometry of the label.
105. self.title\_label.image = self.title\_image # Sets the label's image to the tkinter image 'title\_image'.
107. self.id\_label = tk.Label(self.master, text='Staff ID:', bg=bg\_colour, fg=text\_colour, font=primary\_font) # Defines a label.
108. self.id\_label.place(x=100, y=120, w=80, h=25) # Sets the geometry of the label.
110. self.p\_label = tk.Label(self.master, text='Password:', bg=bg\_colour, fg=text\_colour, font=primary\_font) # Defines a label.
111. self.p\_label.place(x=100, y=160, w=80, h=25) # Sets the geometry of the label.
113. ### TKINTER ENTRY BOXES
114. self.id\_entry = tk.Entry(self.master, textvariable=self.id\_var, fg=text\_colour, font=secondary\_font) # Defines an entry box that allows the user to enter their staff ID. The value entered is stored in the variable created earlier.
115. self.id\_entry.place(x=200, y=120, w=200, h=25) # Sets the entry box's geometry.
116. self.id\_entry.bind('<Return>', **lambda** event: self.p\_entry.focus()) # When pressing the return key while this widget is focused, will shift focus to the password entry.
117. self.id\_entry.bind('<Escape>', **lambda** event: self.master.focus()) # When pressing the escape key while this widget is focued, will force focus off this widget.
118. self.id\_entry.focus() # Focuses this widget so the user can type their ID without having to click the entry box first.
120. self.p\_entry = tk.Entry(self.master, textvariable=self.p\_var, fg=text\_colour, font=secondary\_font, show='\*')# Defines an entry box that allows the user to enter their password. The value entered is stored in the variable created earlier. The password is also hidden from view by showing ever character as '\*'s.
121. self.p\_entry.place(x=200, y=160, w=200, h=25) # Sets the entry box's geometry.
122. self.p\_entry.bind('<Return>', **lambda** event: self.log\_in()) # When pressing the return key while this widget is focused, will run the 'log\_in' function.
123. self.p\_entry.bind('<Escape>', **lambda** event: self.master.focus()) # When pressing the escape key while this widget is focued, will force focus off this widget.
125. ### TKINTER BUTTONS
126. self.login\_button = tk.Button(self.master, text='Log In', command=self.log\_in, bg=bg\_colour, fg=text\_colour, font=primary\_font) # Defines a button which when pressed, runs the function 'log\_in'.
127. self.login\_button.place(x=320, y=200, w=80, h=30) # Sets the geometry of the button.
128. self.login\_button.bind('<Return>', **lambda** event: self.log\_in()) # When pressing the return key while this widget is focused, will run the 'log\_in' function.
129. self.login\_button.bind('<Escape>', **lambda** event: self.master.focus()) # When pressing the escape key while this widget is focued, will force focus off this widget.
131. self.help\_button = tk.Label(self.master, text='Do not know your details?', bg=bg\_colour, fg=text\_colour, font=(primary\_font, 12)) # Defines a label that contains text coloured 'text\_colour' in the font 'font\_1' sized 12.
132. self.help\_button.place(x=100, y=202, w=180, h=25) # Sets the geometry of the label.
133. self.help\_button.bind('<Button-1>', **lambda** event: messagebox.showinfo('Information: Login Help', 'Ask your manager for your login details\nor for first time use, read the User Guide\nor view the "README.txt" text file.')) # Makes the label clickable, that when clicked runs the function 'info'.
134. self.help\_button.bind('<Enter>', **lambda** event: self.help\_button.configure(font=(primary\_font, 12, 'underline'))) # When the mouse cursor is hovering above this label, the text becomes underlined to indicate it's clickable.
135. self.help\_button.bind('<Leave>', **lambda** event: self.help\_button.configure(font=(primary\_font, 12))) # When the mouse cursor is no longer hovering above this label, the text is no longer underlined.
137. ### GENERATES A DATABASE IF NONE EXISTS
138. db\_open() # Creates connection and interaction to the database.
139. c.execute('CREATE TABLE IF NOT EXISTS Staff(StaffID INTEGER PRIMARY KEY, FirstName VARCHAR(255) NOT NULL, LastName VARCHAR(255) NOT NULL, Password VARCHAR(255) NOT NULL, IsAdmin BIT DEFAULT 0)') # Creates Staff table in the database.
140. conn.commit() # Processes the SQL command.
141. c.execute('SELECT COUNT(\*) FROM Staff') # Retrieves the amount of records in the Staff table.
143. **if** c.fetchone()[0] == 0: # If the Staff table is empty. (In other words if the table has just been created).
144. log('Database not found. Creating a new one.') # Logs a message to the console to record that a new database was created.
145. c.execute('INSERT INTO Staff(FirstName, LastName, Password, IsAdmin) VALUES("Jim", "Shaw", "ess19", 1)') # Inserts a temporary admin login into the staff table for the admins in order to access the program.
146. c.execute('CREATE TABLE Clients(ClientID INTEGER PRIMARY KEY, FirstName VARCHAR(255) NOT NULL, LastName VARCHAR(255) NOT NULL, Mobile VARCHAR(255), Home VARCHAR(255))') # Creates Client table in the database.
147. c.execute('CREATE TABLE Haircuts(HaircutID INTEGER PRIMARY KEY, Haircut VARCHAR(255) NOT NULL, Price REAL NOT NULL, EstimatedTime INTEGER)') # Creates Haircut table in the database.
148. c.execute('CREATE TABLE Hairdressers(StaffID INTEGER NOT NULL, Rate INTEGER DEFAULT 100)') # Creates Hairdressers table in the database.
149. c.execute('CREATE TABLE Appointments(AppointmentID INTEGER PRIMARY KEY, StaffID INTEGER NOT NULL, ClientID INTEGER NOT NULL, HaircutID INTEGER NOT NULL, StartTime INTEGER NOT NULL, Duration INTEGER NOT NULL, Date VARCHAR(255) NOT NULL, AmountPaid REAL)') # Creates Appointment table in the database.
150. conn.commit() # Processes the SQL commands.
151. db\_close() # Closes connection and interaction to the database.
153. ### 'LOGIN' CLASS FUNCTIONS
154. **def** log\_in(self): # Defines a function called 'login' (This function performs the login process).
155. '''''The function ran when the user has inputted a Staff ID and password and then pressed enter or clicked the log in button. The function compares the inputted values to the ones in the database and then performs accordingly.'''
156. self.id = self.id\_var.get() # Retrieves the value in the tkinter variable linked to the 'staffID' entry box, and stores it as 'users\_ID'.
157. self.p = self.p\_var.get() # Retrieves the value in the tkinter variable linked to the 'password' entry box, and stores it as 'p'.
158. **if** self.id == '' **or** self.p == '': # Checks that neither entry boxes were empty.
159. messagebox.showerror('Error: Empty field(s)', 'Please ensure you have entered both your Staff ID and password.') # If one of them was, an tkinter error window is displayed informing the user on their mistake.
161. **else**:
162. db\_open() # Opens the connection and interaction to the database
163. c.execute('SELECT \* FROM Staff WHERE StaffID=?', (self.id,)) # Retrieves the staff's details of the staff with ID 'users\_ID'.
164. **global** user
165. user = list(c.fetchone()) # Fetches the result.
166. db\_close() # Closes connection and interaction to the database.
168. **if** **not** user == None **and** self.p == user[3]: # Condition: If the staff ID is within the database and the password matches.
169. log('{} {} logged in.'.format(user[1], user[2])) # Log the following string.
170. **global** main\_window # Defines the main window as global so it can be used outside this function.
171. main\_window = Main(root) # Creates a new instance of the class 'Main' called 'main\_window'.
172. root.mainloop() # Once the instance (main window) is created, stop until the window is closed. Then the program will run the next line of code.
173. **return** # End function early.
175. **else**:
176. messagebox.showerror('Error: Incorrect login details', 'Your ID or password is incorrect. Please check your details and try again.') # Displays a Tkinter error window informing the user the login details are incorrect.
178. self.id\_entry.delete(0, tk.END) # Clears the 'StaffID' entry box.
179. self.p\_entry.delete(0, tk.END) # Clears the 'password' entry box.
180. self.id\_entry.focus() # Refocuses the 'StaffID' entry box.

183. #======================================================================================
184. #                                        MAIN WINDOW
185. #======================================================================================
186. **class** Main(): # Defines a class that will be used for the main window.
187. '''''The main window of the program that contains the booking system.'''
188. **def** \_\_init\_\_(self, master): # Initializes the following code when an instance of a class is created.
189. '''''This function is run when the class is initialised.'''
190. ### VARIABLES
191. self.master = master # Make the variable 'master' an attribute to the Main class.
192. self.width = main\_window\_width # Make the variable an attribute to the Main class.
193. self.height = main\_window\_height # Make the variable an attribute to the Main class.
194. self.update = False # Define a new attribute.
196. ### TKINTER WINDOW CONFIG
197. **for** widget **in** self.master.winfo\_children(): # For all widgets in the tkinter window.
198. widget.destroy() # Destroy it.
200. self.master.title('Essensuals Booking System (Version {})'.format(version)) # Sets title.
201. self.master.geometry('{}x{}+0+10'.format(self.width, self.height)) # Sets windows geometry.
202. self.master.resizable(True, True) # Sets the window so it cannot be resized.
203. self.master.bind('<Configure>', self.on\_resize) # Bind the function 'on\_resize' to trigger whenever the windows geometry is altered.
204. self.master.bind('<Left>', **lambda** event: self.move\_day(False)) # Bind the function 'move\_day' with parameter False when the left arrow key is pressed.
205. self.master.bind('<Right>', **lambda** event: self.move\_day(True)) # Bind the function 'move\_day' with parameter True when the right arrow key is pressed.
207. ### TKINTER LABELS
208. self.header\_label = tk.Label(self.master, bg=primary\_colour) # Defines the header label which is coloured 'primary\_colour'.
209. self.header\_label.place(x=0, y=0, w=screen\_width, h=80) # Sets the geometry of the header.
211. self.footer\_label = tk.Label(self.master, bg=secondary\_colour, fg='#FFFFFF', text='Programmed by Alex Dowsett', font=(primary\_colour+' 7')) # Defines the footer which **is** coloured 'colour\_2'. It also adds some centered, white text **in** the font 'font\_1'.
212. self.footer\_label.place(x=0, h=40) # Sets the geometry of the footer label.
214. self.time\_label = tk.Label(self.master, bg=primary\_colour, fg=bg\_colour, font=(tertiary\_font, 20, 'bold'), anchor='w') # Defines the following Tkinter label.
215. self.time\_label.place(x=235,y=5,w=120, h=40) # Sets the geometry of the label.
217. self.date\_label = tk.Label(self.master, bg=primary\_colour, fg=bg\_colour, font=(tertiary\_font, 20, 'bold'), anchor='e') # Defines the following Tkinter label.
218. self.date\_label.place(x=100,y=5,w=135, h=40) # Sets the geometry of the label.
220. self.welcome\_label = tk.Label(self.master, bg=primary\_colour, fg=bg\_colour, text='Welcome {}.'.format(user[1]), font=(tertiary\_font, 20, 'bold'), anchor='w') # Defines the following tkinter label.
221. self.welcome\_label.place(x=100, y=35, w=400, h=40) # Sets the geometry of the label.
223. self.week\_label = tk.Label(self.master, bg=bg\_colour, fg=text\_colour, font=(tertiary\_font, 26, 'bold')) # Defines the following Tkinter label.
224. self.week\_label.place(x=0, y=80) # Sets the geometry of the label.
226. self.title\_image = tk.PhotoImage(file='resources/title\_large.png') # Imports a photo file as an attribute named 'title\_large'.
227. self.title\_label = tk.Label(self.master, bg=primary\_colour, image=self.title\_image) # Defines a label that will contain the image stored in the attribute 'title\_large'.
228. self.title\_label.place(y=6, w=382, h=68) # Sets the geometry of the label.
229. self.title\_label.image = self.title\_image # Sets the image in 'title\_large' to an attribute of the new label.
231. ### TKINTER BUTTONS
232. self.logout\_button = tk.Label(self.master, bg=bg\_colour, fg=primary\_colour, text='Log Out', font=(primary\_font, 12)) # Defines the following Tkinter button.
233. self.logout\_button.place(y=80, w=80, h=30)
234. self.logout\_button.bind('<Button-1>', **lambda** event: self.log\_out()) # Makes the label clickable, that when clicked runs the function 'info'.
235. self.logout\_button.bind('<Enter>', **lambda** event: self.logout\_button.configure(font=(primary\_font, 12, 'underline'))) # When the mouse cursor is hovering above this label, the text becomes underlined to indicate it's clickable.
236. self.logout\_button.bind('<Leave>', **lambda** event: self.logout\_button.configure(font=(primary\_font, 12))) # When the mouse cursor is no longer hovering above this label, the text is reverted to its normal state.
238. self.settings\_light\_image = tk.PhotoImage(file='resources/settings\_light.png') # Imports a photo file as an attribute named 'settings\_light'.
239. self.settings\_dark\_image = tk.PhotoImage(file='resources/settings\_dark.png') # Imports a photo file as an attribute named 'settings\_dark'.
240. self.settings\_button = tk.Label(self.master, bg=primary\_colour, image=self.settings\_light\_image) # Defines a label that contains the image stored in the variable 'settings\_light'.
241. self.settings\_button.place(x=15, y=15, w=50, h=50) # Sets the geometry of the label.
242. self.settings\_button.image = self.settings\_light\_image # Sets the image in 'settings\_light' to an attribute of the label.
243. self.settings\_button.bind('<Button-1>', **lambda** event: self.open\_settings\_window()) # Makes the label clickable, that when clicked runs the function 'settings'.
244. self.settings\_button.bind('<Enter>', **lambda** event: self.on\_hover(0)) # When the mouse cursor is hovering above this label, the function 'change\_settings\_dark' is ran.
245. self.settings\_button.bind('<Leave>', **lambda** event: self.on\_hover(1)) # When the mouse cursor is no longer hovering above this label, the function 'change\_settings\_dark' is ran.
247. self.arrow\_left\_light\_image = tk.PhotoImage(file='resources/arrow\_left\_light.png') # Imports a photo file as an attribute named 'title\_large'.
248. self.arrow\_left\_dark\_image = tk.PhotoImage(file='resources/arrow\_left\_dark.png') # Imports a photo file as an attribute named 'title\_large'.
249. self.arrow\_left\_button = tk.Label(self.master, bg=bg\_colour, image=self.arrow\_left\_dark\_image) # Defines a label that will contain the image stored in the attribute 'title\_large'.
250. self.arrow\_left\_button.place(y=80, w=70) # Sets the geometry of the label.
251. self.arrow\_left\_button.image = self.arrow\_left\_dark\_image # Sets the image in 'title\_large' to an attribute of the new label.
252. self.arrow\_left\_button.bind('<Button-1>', **lambda** event: self.move\_week(False)) # Makes the label clickable, that when clicked runs the function 'settings'.
253. self.arrow\_left\_button.bind('<Enter>', **lambda** event: self.on\_hover(2)) # When the mouse cursor is hovering above this label, the function 'change\_settings\_dark' is ran.
254. self.arrow\_left\_button.bind('<Leave>', **lambda** event: self.on\_hover(3)) # When the mouse cursor is no longer hovering above this label, the function 'change\_settings\_dark' is ran.
256. self.arrow\_right\_light\_image = tk.PhotoImage(file='resources/arrow\_right\_light.png') # Imports a photo file as an attribute named 'title\_large'.
257. self.arrow\_right\_dark\_image = tk.PhotoImage(file='resources/arrow\_right\_dark.png') # Imports a photo file as an attribute named 'title\_large'.
258. self.arrow\_right\_button = tk.Label(self.master, bg=bg\_colour, image=self.arrow\_right\_dark\_image) # Defines a label that will contain the image stored in the attribute 'title\_large'.
259. self.arrow\_right\_button.place(y=80, w=70) # Sets the geometry of the label.
260. self.arrow\_right\_button.image = self.arrow\_right\_dark\_image # Sets the image in 'title\_large' to an attribute of the new label.
261. self.arrow\_right\_button.bind('<Button-1>', **lambda** event: self.move\_week(True)) # Makes the label clickable, that when clicked runs the function 'settings'.
262. self.arrow\_right\_button.bind('<Enter>', **lambda** event: self.on\_hover(4)) # When the mouse cursor is hovering above this label, the function 'change\_settings\_dark' is ran.
263. self.arrow\_right\_button.bind('<Leave>', **lambda** event: self.on\_hover(5)) # When the mouse cursor is no longer hovering above this label, the function 'change\_settings\_dark' is ran.
265. ### TKINTER STYLE
266. # This section of code sets a default style used for tkinter widgets. This style however will only apply for the tabs within the tkinter notebook we will define later on.
267. self.style = ttk.Style() # Define the style
268. self.style.configure('.', bg=bg\_colour, fg=text\_colour, font=primary\_font) # Adds the font just defined to the style. Also sets the background colour to 'bg\_colour'.
269. self.current\_theme = self.style.theme\_use() # Retrieves the current theme being used.
270. self.style.theme\_settings (self.current\_theme, { # Configures the current theme by adding padding to the tabs in the widget 'TNotebook'.
271. 'TNotebook.Tab': {
272. 'configure': {'padding': [5, 10] } } } ) # The padding around the text within the notebook tab (Increases the size of the tabs slightly).
273. self.style.theme\_use(self.current\_theme) # Enforces the new theme that's been configured.
275. # These 3 lines of code define a variable that contains padding for the tabs in the notebook. Adding this padding to the tabs ensures they take up all the space they are allowed.
276. self.padding = [] # Defines an array.
277. **for** i **in** range(100): # Loops the following code 100 times.
278. self.padding.append(' ') # Adds a space character to the array.
280. ### CLOCK
281. # Starts a recursive function that calls itself every 0.2 seconds. This function checks the displayed time and date against the real time and date and updates the displayed values if neccessary.
282. self.time = '' # Defines the time as an empty string so it has to be updated on the first call of the function.
283. self.date = '' # Defines the date as an empty string so it has to be updated on the first call of the function.
284. self.tick() # Calls the tick function.
286. ### CALCULATING THE WORKING DATES
287. # This section of code converts the working days of the week to actual dates. These dates are used for the notebook/table tabs and the text above the notebook/table stating what working week it is.
288. day\_of\_the\_week = int(time.strftime('%w')) # Retrieves the day of the week it is today. Between 0-6 where 0 is Sunday.
289. **if** day\_of\_the\_week == 0: # If 0.
290. day\_of\_the\_week = 7 # Change to 7.
291. **global** working\_dates # Defines a variable and makes it global.
292. working\_dates = [] # Defines the variable as an array.
293. **for** i **in** range(len(working\_days)): # Loop i amount of times, where i is the number of days the company is open in a week.
294. working\_dates.append(datetime.date.today() + datetime.timedelta(days=(working\_days[i] - day\_of\_the\_week))) # Calculates the dates for this working week.
296. ### FETCHES HAIRDRESSERS' DETAILS FROM THE DATABASE
297. # This section of code fetches all the hairdressers' details from the database.
298. db\_open()
299. c.execute('SELECT StaffID, FirstName, LastName FROM Staff WHERE StaffID IN (SELECT StaffID FROM Hairdressers)') # Retrieve the staff information for all hairdressers.
300. self.hairdressers = c.fetchall() # Fetch all records that match the above SQL statement.
301. db\_close()
303. # These 3 lines of code convert the list of tuples to a list of lists (2D list).
304. **for** i **in** range(len(self.hairdressers)): # Loop for each hairdresser.
305. self.hairdressers[i] = list(self.hairdressers[i]) # Converts tuple to array/list.
306. self.hairdressers[i].append('') # Adds a string value onto the end of the array.

309. ### ENSURES HAIRDRESSER NAMES ARE UNIQUE
310. # Ensures all hairdressers have unique names displayed. If two hairdressers share the same first name then the first letter of their surnames will also be displayed.
311. # If their names still clash the next letter will be displayed. In the case their surnames also match it will displayed both their full name. An admin may add a
312. # number or other unique identifier to their name if both of the same names being displayed is not suitable.
313. **for** i **in** range(len(self.hairdressers)): # Loop for each hairdresser.
314. surname\_clash = [] # Defines an array.
315. **while** **not** surname\_clash == [i]: # Loop until no surname clash.
316. surname\_clash = [i] # Sets value of variable.
317. **for** j **in** range(len(self.hairdressers)): # Loop for all hairdressers.
318. **if** i != j **and** str(self.hairdressers[i][1]) + str(self.hairdressers[i][3]) == str(self.hairdressers[j][1]) + str(self.hairdressers[j][3]) **and** self.hairdressers[j][2] != '': # If surname clash and not a clash with self.
319. surname\_clash.append(j) # Add index which clashed with.
320. **if** len(surname\_clash) != 1: # If clash is not just self.
321. **for** j **in** range(len(surname\_clash)): # Loop for each clash.
322. self.hairdressers[surname\_clash[j]][3] += str(self.hairdressers[surname\_clash[j]][2][0]) # Move first character of string to last index (index 3) of array.
323. self.hairdressers[surname\_clash[j]][2] = self.hairdressers[surname\_clash[j]][2][1:] # Remove first character of string from index 2.
325. ### NOTEBOOK/TABLE
326. self.nb = ttk.Notebook(self.master) # Defines a Tkinter notebook widget.
327. self.nb.bind('<Escape>', **lambda** event: self.master.focus()) # If user presses Escape key the program will unfocus all widgets.
329. ### MORE VARIABLES
330. # Defines variables to be used in the appoinments table.
331. self.tabs = [] # Defines an array that will contain the 6 pages (a page for each working day).
332. self.lbs = [] # This 2d array will contain the listboxes for each page. We need a listbox for each hairdresser and we need a set of listboxes for each page in the notebook.
333. self.names\_frame = [] # Defines an array that will contain a frame that will contain the names of the hairdressers. There will be a frame per day.
334. self.names = [] # This 2d array will contain the canvases to display the hairdressers' names. We use a canvas instead to display the text vertically to save space. We need a set of canvases for each page.
335. self.scrolls = [] # Defines an array that will contain the 6 scroll bars (a scrollbar for each page).
336. self.clock\_image = tk.PhotoImage(file='resources/clock.png') # Defines a new variable to hold a Tkinter image of 'clock.png'.
338. **for** i **in** range(len(working\_days)): # Loop for all working days in a week.
339. self.names.append([]) # Add array to names array.
340. self.lbs.append([]) # Add array to list boxes array.
342. ### TABS
343. self.tabs.append(tk.Frame(self.nb)) # Add frame for each working day of the week.
344. self.tabs[i].rowconfigure(1, weight=1) # Add a weight to the frame (Takes piority of space over weightless widgets).
345. self.tabs[i].bind('<Visibility>', **lambda** event: [self.on\_resize(), self.update\_table(), self.master.focus()]) # When the tab is changed in the notebook run the following functions.
346. self.nb.add(self.tabs[i]) # Add the tab to the notebook.
348. ### FRAMES
349. # A frame to contain all the canvasses which display the hairdressers' names.
350. self.names\_frame.append(tk.Frame(self.tabs[i], height=64)) # Adds a frame to contain hairdressers' names inside the tab frame for each working day of the week.
351. self.names\_frame[i].grid(row=0, column=0, sticky='new', columnspan=len(self.hairdressers)+1) # Set geometry for frame.
353. ### SCROLLBARS
354. # The scrollbar is used to scroll thought the list boxes containing all the appointments.
355. self.scrolls.append(ttk.Scrollbar(self.tabs[i], orient='vertical', command=self.on\_scroll\_by\_bar)) # Add a Tkinter scrollbar for each tab.
356. self.scrolls[i].grid(row=0, column = len(self.hairdressers)+1, sticky='ns', rowspan = 2) # Sets geometry and sets it to last column of tab frame.
358. **for** j **in** range(len(self.hairdressers)+1): # Loop for each hairdresser.
359. ### CANVASSES
360. # The canvesses hold the names of the haidressers as column headings. Theses are displayed on canvasses instead of labels to allow them to be rotated 90 degrees.
361. self.tabs[i].columnconfigure(j, weight=1) # Sets a weight for all columns except last (scrollbar column).
362. self.names[i].append(tk.Canvas(self.names\_frame[i])) # Add canvas for hairdresser's name.
363. self.names[i][j].place(y=0, h=64) # Sets the canvas' geometry.
365. **if** j == 0: # If first column (If the column that contains the appointment times).
366. self.names[i][0].clock = self.names[i][0].create\_image(0, 0, anchor='nw', image=self.clock\_image) # Add image to canvas.
367. self.names[i][0].image = self.clock\_image # Set canvas attribute 'image' to 'clock\_image'.
369. **else**:
370. self.names[i][j].create\_text(24, 56, anchor='sw', angle=90, text=self.hairdressers[j-1][1], fill=text\_colour, font=(primary\_font, 10)) # Add hairdresser's name at 90 degree angle.
371. **if** **not** self.hairdressers[j-1][3] == '': # If another hairdresser shares this name.
372. self.names[i][j].create\_text(40, 56, anchor='sw', angle=90, text=self.hairdressers[j-1][3], fill=text\_colour, font=(primary\_font, 10)) # Add the first part of the hairdresser's surname aswell.
374. ### LISTBOXES
375. # Each listbox is a column a table. As there are multiple tables (one for each day) there is a 2D array of listboxes.
376. self.lbs[i].append(tk.Listbox(self.tabs[i], yscrollcommand=self.on\_scroll\_by\_wheel, fg=text\_colour, font=(secondary\_font, 9))) # Add a listbox below the hairdresser's name.
377. self.lbs[i][j].tab = i # Add an attribute to the list box so we know what tab is belongs in.
378. self.lbs[i][j].lb = j # Add an attribute to the list box so we know which list box it is from left to right.
379. self.lbs[i][j].grid(row=1, column=j, sticky='nesw') # Set geometry of list box and make it expand in all directions in the frame.
380. self.lbs[i][j].bind('<<ListboxSelect>>', self.open\_appointments\_window) # If the list box is clicked run the following function.
381. self.lbs[i][j].bind('<FocusIn>', **lambda** event: self.master.focus()) # Makes the listbox unfocusable.
383. ### RESIZING WIDGETS
384. # Resizes the widgets to appropriate sizes depending on the screen size. Also sets the window in a zoomed state if enabled.
385. # Zooming the window naturally triggers self.on\_resize() because the tkinter window is set to run that function whenever the windows width of height is altered (set on line 186).
386. # Therefore the function is only required to be run if the window is not set to be in a zoomed state on start up.
387. **if** main\_window\_zoomed == True:
388. self.master.state('zoomed') # Maximises window to fullscreen.
389. **else**:
390. self.on\_resize() # Run the function 'on\_resize'.
392. ### 'MAIN' CLASS FUNCTIONS
393. **def** on\_resize(self, event=None):
394. '''''The function called when the geometry of the main window is altered. This function adjusts the geometry any widgets that depend on the height and width of the main window.'''
395. **if** self.width != self.master.winfo\_width() **or** event == None:  # This code is run if the width has changed, or an 'event' value was not provided.
396. self.width = self.master.winfo\_width() # Get the new width of the main window in pixels.
398. self.week\_label.place(w=self.width) # Alter the width of the label to equal the new width.
399. self.logout\_button.place(x=self.width-80) # Alter the position of the log out button based on the new width.
400. self.arrow\_left\_button.place(x=self.width\*0.5-320) # Alter the position of the left arrow button based on the new width.
401. self.arrow\_right\_button.place(x=self.width\*0.5+250) # Alter the position of the right arrow button based on the new width.
402. self.title\_label.place(x=self.width-405) # Alter the position of the label based on the new width.
403. self.footer\_label.place(w=self.width) # Alter the position of the label based on the new width.
404. self.nb.place(x=(self.width-10)\*0.03+5, w=(self.width-10)\*0.94) # Alter the position and width of the notebook based on the new width.
405. self.names\_frame[self.nb.index(self.nb.select())].update\_idletasks() # Update geometry of the name frame of the current tab in the notebook.
406. **for** i **in** range(len(self.hairdressers)+1): # For each hairdresser.
407. self.names[self.nb.index(self.nb.select())][i].place(x=self.names\_frame[self.nb.index(self.nb.select())].winfo\_width()\*i/(len(self.hairdressers)+1), w=self.names\_frame[self.nb.index(self.nb.select())].winfo\_width()/(len(self.hairdressers)+1)) # Alter the position and width of the canvas in the name frame.
408. self.names[self.nb.index(self.nb.select())][0].coords(self.names[self.nb.index(self.nb.select())][0].clock, (self.names\_frame[self.nb.index(self.nb.select())].winfo\_width()/(len(self.hairdressers)+1)/2)-16, 24) # Alter the position of the clock image in the first canvas.
410. **if** self.height != self.master.winfo\_height() **or** event == None: # This code is run if the height has changed, or an 'event' value was not provided.
411. self.height = self.master.winfo\_height() # Get the new height of the main window in pixels.
413. self.week\_label.place(h=(self.height-165)\*0.04+40) # Alter the height of the label based on the new height.
414. self.arrow\_left\_button.place(h=(self.height-165)\*0.04+40) # Alter the height of the left arrow button based on the new height.
415. self.arrow\_right\_button.place(h=(self.height-165)\*0.04+40) # Alter the height of the right arrow button based on the new height.
416. self.footer\_label.place(y=self.height-40) # Alter the postition of the label based on the new height.
417. self.nb.place(y=(self.height-165)\*0.04+120, h=(self.height-165)\*0.93-5) # Alter the position and width of the notebook based on the new height.
419. **def** on\_hover(self, hover):
420. '''''This function is triggered when the windows cursor hovers or unhovers over any interactable labels.'''
421. **if** hover == 0: # If the cursor hovers over the settings button.
422. self.settings\_button.configure(image=self.settings\_dark\_image) # Change image to darker version.
423. self.settings\_button.image = self.settings\_dark\_image # Change 'image' attribute to darker version.
424. **elif** hover == 1: # If the cursor unhovers over the settings button.
425. self.settings\_button.configure(image=self.settings\_light\_image) # Change image to lighter version.
426. self.settings\_button.image = self.settings\_light\_image # Change 'image' attribute to lighter version.
428. **elif** hover == 2: # If the cursor hovers over the left arrow (previous week) button.
429. self.arrow\_left\_button.configure(image=self.arrow\_left\_light\_image) # Change image to lighter version.
430. self.arrow\_left\_button.image = self.arrow\_left\_light\_image # Change 'image' attribute to lighter version.
431. **elif** hover == 3: # If the cursor unhovers over the left arrow (previous week) button.
432. self.arrow\_left\_button.configure(image=self.arrow\_left\_dark\_image) # Change image to darker version.
433. self.arrow\_left\_button.image = self.arrow\_left\_dark\_image # Change 'image' attribute to darker version.
435. **elif** hover == 4: # If the cursor hovers over the right arrow (next week) button.
436. self.arrow\_right\_button.configure(image=self.arrow\_right\_light\_image) # Change image to lighter version.
437. self.arrow\_right\_button.image = self.arrow\_right\_light\_image # Change 'image' attribute to lighter version.
438. **elif** hover == 5: # If the cursor unhovers over the right arrow (next week) button.
439. self.arrow\_right\_button.configure(image=self.arrow\_right\_dark\_image) # Change image to darker version.
440. self.arrow\_right\_button.image = self.arrow\_right\_dark\_image # Change 'image' attribute to darker version.
442. **def** log\_out(self):
443. '''''This function is called when the user clicks the log out button. The function prepares the window to be converted back into a log in window and then converts it.'''
444. **for** widget **in** self.master.winfo\_children(): # For all widgets in the tkinter window.
445. widget.destroy() # Destroy it.
446. self.master.unbind('<Configure>') # Unbind the 'on\_resize' function.
447. self.master.state('normal') # Unzooms the window.
449. **global** login\_window # Defines the main window as global so it can be used outside this function.
450. log('{} {} logged out.'.format(user[1], user[2])) # Logs the following string.
451. login\_window = Login(root) # Creates a new instance of the class 'Main' called 'main\_window'.
452. root.mainloop() # Once the instance (main window) is created, stop until the window is closed. Then the program will run the next line of code.
454. **def** move\_day(self, direction):
455. '''''This function is called when the tab is changed or needs to be changed.'''
456. **if** direction == True: # If direction is left (next day).
457. **if** self.nb.index(self.nb.select()) == len(working\_days)-1: # If last day of week.
458. self.move\_week(True) # Move to next week.
459. self.nb.select(0) # Select first tab (day) of new week.
460. **else**:
461. self.nb.select(self.nb.index(self.nb.select())+1) # Otherwise move to next day.
462. **elif** direction == False: # If direction is right (previous day).
463. **if** self.nb.index(self.nb.select()) == 0: # If first day of week.
464. self.move\_week(False) # Move to previous week.
465. self.nb.select(len(working\_days)-1) # Select last tab (day) of new week.
466. **else**:
467. self.nb.select(self.nb.index(self.nb.select())-1) # Otherwise move to previous day.
469. self.on\_resize() # Set new widgets' initial geometry.
470. self.update\_table() # Update / repopulate table as it's a new day.
471. self.master.focus() # Unfocus all widgets.
473. **def** move\_week(self, direction):
474. '''''This function allows the user to navigate through the weeks in the table.'''
475. **global** working\_dates # Makes the variable global.
477. **if** direction == False: # Move back a week.
478. **for** i **in** range(len(working\_dates)): # For all working dates.
479. working\_dates[i] -= datetime.timedelta(days=7) # Minus 7 days from the date.
481. **elif** direction == True: # Move forward a week.
482. **for** i **in** range(len(working\_dates)): # For all working dates.
483. working\_dates[i] += datetime.timedelta(days=7) # Add 7 days to the date.
485. self.update\_table() # Updates the appointments table. Repopulates the list boxes with appointments.
487. **def** tick(self):
488. '''''This function compares real time and date to the displayed time and date and updates them where necessary. This function calls itself after every 0.2 seconds.'''
489. **if** self.time != time.strftime('%H:%M:%S'): # If displayed time does not equal real time.
490. self.time = time.strftime('%H:%M:%S') # Then update the displayed time variable.
491. self.time\_label.config(text=self.time) # Redisplay displayed time.
493. **if** self.date != time.strftime('%d/%m/%y'): # If displayed date does not equal real date.
494. self.date = time.strftime('%d/%m/%y') # Then update the displayed time variable.
495. self.date\_label.config(text=self.date + '|') # Redisplay displayed date.
497. self.time\_label.after(200, self.tick) # Call this function again in 0.2 seconds.
499. **def** on\_scroll\_by\_bar(self, \*args):
500. '''''This function is triggered when the user scrolls using the scroll bar. This sets all the listboxes 'yview' to equal the 'yview' of the scrollbar. In simplier words, the function syncs the yview of the listboxes to the scrollbar.'''
501. **for** i **in** range(len(self.hairdressers)+1): # For all listboxes.
502. self.lbs[self.nb.index(self.nb.select())][i].yview(\*args) # Update yview of list box to match the yview of scrollbar.
504. **def** on\_scroll\_by\_wheel(self, \*args):
505. '''''This function is triggered when the user scrolls using the scroll wheel on a listbox. This sets all the other listboxes 'yview' to the 'yview' of the scrolled listbox. It also synces the scrollbar 'yview' to the 'yview' of the scrolled listbox.'''
506. **for** i **in** range(len(self.hairdressers)+1): # For all listboxes.
507. self.lbs[self.nb.index(self.nb.select())][i].yview\_moveto(args[0]) # Update yview of list box to match the yview of the list box that was scrolled.
508. self.scrolls[self.nb.index(self.nb.select())].set(\*args) # Updates the yview of the scrollbar to match the list box that was scrolled.
510. **def** update\_table(self):
511. '''''This function populates the table thats currently being displayed. This function is called whenever any changes are made to appointments or when the user changes the displayed day or week.'''
512. ### CLEARING CONTENTS OF LIST BOXES
513. **for** i **in** range(len(self.hairdressers)+1): # For all listboxes.
514. self.lbs[self.nb.index(self.nb.select())][i].delete(0, tk.END) # Clears the listbox's content (so it can be repopulated).
515. **if** i != 0: # If not the first listbox (that holds time)
516. self.lbs[self.nb.index(self.nb.select())][i].appointment\_id = [] # Adds an attribute to the listbox to hold the appointment ids corresponding to the appointments stored in this list box.
518. ### UPDATING DISPLATED DATES
519. **for** i **in** range(len(working\_dates)): # For all working days.
520. self.nb.tab(i, text=(working\_dates[i].strftime('%a') + ' ' + add\_date\_suffix(working\_dates[i].strftime('%d')) + ' ' + working\_dates[i].strftime('%b') + ' ' + working\_dates[i].strftime('%Y') + ''.join(self.padding))) # Display working date in the tab.
522. self.lbs[self.nb.index(self.nb.select())][0].bindtags((self.lbs[i][0], self.tabs[i], 'all')) # Makes first listbox (that holds the times) unclickable.
523. self.week\_label.config(text='{} {} - {} {}'.format(working\_dates[0].strftime('%A'), add\_date\_suffix(working\_dates[0].strftime('%d')), working\_dates[len(working\_dates)-1].strftime('%A'), add\_date\_suffix(working\_dates[len(working\_dates)-1].strftime('%d')))) # Updates the label that displayed week above the table.
525. ### REPOPULATING LIST BOXES
526. db\_open()
527. **for** i **in** range((last\_appointment\_time - first\_appointment\_time)//appointment\_intervals+1): # For all appointment slots per hairdresser.
528. start\_time = time.strftime('%M:%S', time.gmtime(first\_appointment\_time + i \* appointment\_intervals)) # Calculate time of the appointment.
529. self.lbs[self.nb.index(self.nb.select())][0].insert(tk.END, start\_time) # Add that time to the first list box.
531. c.execute('SELECT AppointmentID, StaffID, ClientID, Duration FROM Appointments WHERE Date=? AND StartTime=?', ((working\_dates[self.nb.index(self.nb.select())].strftime('%d/%m/%Y')), start\_time)) # Retrieve any appointments at that time.
532. self.appointments = c.fetchall() # Fetch all those appointments at that time.
534. **for** j **in** range(len(self.hairdressers)): # For all hairdressers.
535. **for** k **in** range(len(self.appointments)): # For all appointments.
536. self.appointments[k] = list(self.appointments[k]) # Convert tuple to array/list.
538. **if** self.hairdressers[j][0] == self.appointments[k][1]: # If hairdresser id for hairdresser equals hairdresser id in appointment record.
539. c.execute('SELECT FirstName FROM Clients WHERE ClientID=?', (self.appointments[k][2],)) # Retrieve first name of the client that this appointment is for.
540. client = c.fetchone() # Fetch the first name.
542. **for** l **in** range(self.appointments[k][3]//15): # For the amount of slots the appointment takes.
543. self.lbs[self.nb.index(self.nb.select())][j+1].insert(tk.END, client) # Insert client's first name into listbox for that time.
544. self.lbs[self.nb.index(self.nb.select())][j+1].appointment\_id.append(self.appointments[k][0]) # Insert appointment id into the list boxes' attribute (array).
546. **if** self.lbs[self.nb.index(self.nb.select())][j+1].size() <= i: # If the size of the listbox does not match the appointment time we are on.
547. self.lbs[self.nb.index(self.nb.select())][j+1].insert(tk.END, ' -') # Add '-' string (make it an empty slot).
548. self.lbs[self.nb.index(self.nb.select())][j+1].appointment\_id.append(-1) # Append a null appointment id to represent it is empty.
549. db\_close()
551. **def** open\_settings\_window(self):
552. '''''This function opens a popup (Tkinter toplevel) window so the user can config the settings.'''
553. popup = tk.Toplevel(self.master) # Defines a Tkinter toplevel window.
554. **global** settings\_window # Make the tkinter window global.
555. settings\_window = Settings(popup) # Creates a new instance of the class 'Settings'.
556. popup.mainloop() # Wait until the settings window is closed.
558. **def** open\_appointments\_window(self, event):
559. '''''This function is triggered when the user clicks on a list box. It works out the column and row that was clicked then creates an instance of Appointments with that information.'''
560. popup = tk.Toplevel(self.master) # Defines a Tkinter toplevel window.
561. **global** appointments\_window # Makes the tkinter window global.
562. appointments\_window = Appointments(popup, (event.widget.lb-1), int(event.widget.curselection()[0])) # Creates a new instance of the class 'Appointments' with the coords of the appointment that was clicked.
563. popup.mainloop() # Wait until the settings window is closed.

566. #======================================================================================
567. #                                     SETTINGS WINDOW
568. #======================================================================================
569. **class** Settings():
570. **def** \_\_init\_\_(self, master):
571. '''''This function is run when the class is initialised.'''
572. ### VARIABLES
573. self.master = master # Make 'master' an attribute of the 'Settings' class.
574. self.tables\_var = tk.StringVar() # Defines a Tkinter string variable.
576. ### TKINTER WINDOW CONFIG
577. self.master.title('Settings (Version {})'.format(version)) # Sets title
578. self.master.geometry('270x180+{}+{}'.format(((screen\_width-270)//2), ((screen\_height-180)//2)))  # Sets window geometry & centers the window.
579. self.master.resizable(False, False) # Sets the window so it cannot be resized
580. self.master.wm\_iconbitmap('resources/icon.ico') # Sets icon
581. self.master.config(bg=bg\_colour) # Sets background colour of window
582. self.master.protocol('WM\_DELETE\_WINDOW', **lambda**: [self.master.destroy(), self.update\_main\_window()])
584. self.master.focus\_set() # Sets focus on this tkinter window.
585. self.master.grab\_set() # Sets grab on this tkinter window (so no other windows can be focused).
587. ### TKINTER LABELS
588. self.tables\_label = tk.Label(self.master, text="To add, manage or remove any clients, staff,\nhairdressers or haircuts select a table in the\ndropdown menu (Admin access required).", bg=bg\_colour, fg=text\_colour, font=(primary\_font, 10)) # Defines a tkinter label.
589. self.tables\_label.place(x=0, y=40, w=270, h=50) # Sets the geometry of the label.
591. ### TKINTER COMBOBOX (DROP-DOWN MENU)
592. self.tables\_box = ttk.Combobox(self.master, textvariable=self.tables\_var, font=(secondary\_font, 9), state='readonly') # Defines a tkinter combobox.
593. self.tables\_box['values'] = ['Hairdressers/Staff', 'Clients', 'Haircuts'] # Defines the array that contains the data that the drop-down menu will contain.
594. self.tables\_box.place(x=50, y=10, w=170, h=30) # Sets drop-down menu's geometry.
595. self.tables\_box.bind('<Return>', **lambda** event: self.tables\_box.event\_generate('<Down>')) # Makes the drop-down menu drop down when the enter key is pressed when this widget is focused.
596. self.tables\_box.bind('<Escape>', **lambda** event: self.master.focus()) # Unfocuses the widget when the escape key is pressed.
597. self.tables\_box.bind('<<ComboboxSelected>>', **lambda** event: self.open\_tables\_window()) # When a option is selected from the drop-down menu, call the following function.
599. ### TKINTER BUTTONS
600. self.change\_password\_button = tk.Button(self.master, text='Change password', command=**lambda**: self.open\_change\_password\_window(), bg=bg\_colour, fg=text\_colour, font=primary\_font) # Defines a tkinter button.
601. self.change\_password\_button.place(x=50, y=100, w=170, h=30) # Sets buttons's geometry.
602. self.change\_password\_button.bind('<Return>', **lambda** event: self.open\_change\_password\_window()) # Call the following function when the enter key is pressed when this widget is focused.
603. self.change\_password\_button.bind('<Escape>', **lambda** event: self.master.focus()) # Unfocuses the widget when the escape key is pressed.
605. self.dismiss\_button = tk.Button(self.master, text='Dismiss', command=**lambda**: [self.master.destroy(), self.update\_main\_window()], bg=bg\_colour, fg=text\_colour, font=primary\_font) # Defines a tkinter button.
606. self.dismiss\_button.place(x=185, y=140, w=75, h=30) # Sets button's geometry.
607. self.dismiss\_button.bind('<Return>', **lambda** event: [self.master.destroy(), self.update\_main\_window()]) # Call the following functions when the enter key is pressed when this widget is focused.
608. self.dismiss\_button.bind('<Escape>', **lambda** event: self.master.focus()) # Unfocuses the widget when the escape key is pressed.
610. ### 'SETTINGS' CLASS FUNCTIONS
611. **def** open\_change\_password\_window(self):
612. '''''This function opens the tkinter window that allows the user to change their password. This function is called when the user clicks the change passsword button.'''
613. popup = tk.Toplevel(self.master) # Defines a tkinter toplevel window.
614. **global** change\_password\_window # Makes this variable global.
615. change\_password\_window = ChangePassword(popup) # Creates a instance of the ChangePassword class.
616. popup.mainloop() # Wait until this window is closed down.
618. **def** open\_tables\_window(self):
619. '''''This function opens the tkinter window that allows the user to edit the database. This function is called when the user selects an option from the drop-down menu.'''
620. **if** user[4] == True: # If the user is an admin.
621. popup = tk.Toplevel(self.master) # Defines a tkinter toplevel window.
622. **global** tables\_window # Make this variable global.
623. tables\_window = Tables(popup, self.tables\_box.current()) # Creates a instance of the Tables class.
624. self.tables\_box.set('') # Unselects the option in the drop-down menu.
625. popup.mainloop() # Wait until this window is closed down.
627. **else**:
628. messagebox.showerror('Error: Access Denied', 'You must be an admin to edit the tables.') # Display a tkinter errror popup with the following message.
629. self.tables\_box.set('') # Unselects the option in the drop-down menu.
631. **def** update\_main\_window(self):
632. '''''This function is called when the settings window is closed and the main window is refocused/re-grabbed. If any hairdressers have been added or removed from the database, the window will need to be rebuild to account for the change of list boxes, canvases, etc.'''
633. **if** main\_window.update == True: # If the main window needs updating.
634. main\_window.master.unbind('<Configure>') # Unbinds the 'on\_resize' function from the window.
635. main\_window.master.state('normal') # Unzooms the window.
636. **for** tab **in** main\_window.tabs: # For all tabs in the notebook within the main window.
637. tab.unbind('<Visibility>') # Unbind the 'on\_resize' function from the tab.
639. main\_window.\_\_init\_\_(main\_window.master) # Reinitialise the instance of the class.
640. main\_window.update = False # Set the boolean to show the main window no longer needs to be updated.

643. #======================================================================================
644. #                                CHANGE PASSWORD WINDOW
645. #======================================================================================
646. **class** ChangePassword(): # Defines a class that represents the window that lets users change their password.
647. '''''This window lets the user change their password. The window is opened when the user clicks the change password button in the settings window.'''
648. **def** \_\_init\_\_(self, master): # This function is run when a instance of the class is initialised.
649. '''''This function is run when the class is initialised.'''
650. self.master = master # Makes the variable 'master' an attribute of this class.
651. self.current\_p\_var = tk.StringVar() # Defines a new Tkinter string variable.
652. self.new\_p\_var = tk.StringVar() # Defines a new Tkinter string variable.
653. self.new\_p\_confirm\_var = tk.StringVar() # Defines a new Tkinter string variable.
655. ### TKINTER WINDOW CONFIG
656. self.master.title('Change Password (Version {})'.format(version)) # Sets title of the window.
657. self.master.geometry('270x180+{}+{}'.format(((screen\_width-270)//2), ((screen\_height-180)//2)))  # Sets window geometry and centers the window.
658. self.master.resizable(False, False) # Sets the window so it cannot be resized.
659. self.master.wm\_iconbitmap('resources/icon.ico') # Sets icon.
660. self.master.config(bg=bg\_colour) # Sets background colour of window.
661. self.master.protocol('WM\_DELETE\_WINDOW', **lambda**: [self.master.destroy(), settings\_window.master.focus\_set(), settings\_window.master.grab\_set()]) # Runs the following functions when the window is closed down.
663. settings\_window.master.grab\_release() # Release the grab of the settings windows so it can be transferred to the change password window.
664. self.master.focus\_set() # Sets the focus on this window.
665. self.master.grab\_set() # Sets the grab on this window (so other windows cannot take focus).
667. ### TKINTER LABELS
668. self.current\_p\_label = tk.Label(self.master, text='Current\nPassword:', bg=bg\_colour, fg=text\_colour, font=(primary\_font, 10), anchor='e') # Defines a Tkinter label.
669. self.current\_p\_label.place(x=0, y=10, w=90, h=35) # Sets the label's geometry.
671. self.new\_p\_label = tk.Label(self.master, text='New\npassword:', bg=bg\_colour, fg=text\_colour, font=(primary\_font, 10), anchor='e') # Defines a Tkinter label.
672. self.new\_p\_label.place(x=0, y=50, w=90, h=35) # Sets the label's geometry.
674. self.new\_p\_confirm\_label = tk.Label(self.master, text='Re-enter\npassword:', bg=bg\_colour, fg=text\_colour, font=(primary\_font, 10), anchor='e') # Defines a Tkinter label.
675. self.new\_p\_confirm\_label.place(x=0, y=90, w=90, h=35) # Sets the label's geometry.
677. ### TKINTER ENTRY BOXES
678. self.current\_p\_entry = tk.Entry(self.master, textvariable=self.current\_p\_var, fg=text\_colour, font=(secondary\_font, 9), show='\*') # Defines a Tkinter entry box.
679. self.current\_p\_entry.place(x=95, y=15, w=150, h=25) # Sets the entry boxes' geometry.
680. self.current\_p\_entry.bind('<Return>', **lambda** event: self.new\_p\_entry.focus()) # When pressing the return key while this widget is focused, will shift focus to the next widget.
681. self.current\_p\_entry.bind('<Escape>', **lambda** event: self.master.focus()) # Unfocuses the widget when the escape key is pressed while this widget is focused.
682. self.current\_p\_entry.focus()
684. self.new\_p\_entry = tk.Entry(self.master, textvariable=self.new\_p\_var, fg=text\_colour, font=(secondary\_font, 9), show='\*') # Defines a Tkinter entry box.
685. self.new\_p\_entry.place(x=95, y=55, w=150, h=25) # Sets the entry boxes' geometry.
686. self.new\_p\_entry.bind('<Return>', **lambda** event: self.new\_p\_confirm\_entry.focus()) # When pressing the return key while this widget is focused, will shift focus to the next widget.
687. self.new\_p\_entry.bind('<Escape>', **lambda** event: self.master.focus()) # Unfocuses the widget when the escape key is pressed while this widget is focused.
689. self.new\_p\_confirm\_entry = tk.Entry(self.master, textvariable=self.new\_p\_confirm\_var, fg=text\_colour, font=(secondary\_font, 9), show='\*') # Defines a Tkinter entry box.
690. self.new\_p\_confirm\_entry.place(x=95, y=95, w=150, h=25) # Sets the entry boxes' geometry.
691. self.new\_p\_confirm\_entry.bind('<Return>', **lambda** event: self.change\_password()) # When pressing the return key while this widget is focused, will shift focus to the next widget.
692. self.new\_p\_confirm\_entry.bind('<Escape>', **lambda** event: self.master.focus()) # Unfocuses the widget when the escape key is pressed while this widget is focused.
694. ### TKINTER BUTTONS
695. self.change\_p\_button = tk.Button(self.master, text='Change Password', command=**lambda**: self.change\_password(), bg=bg\_colour, fg=text\_colour, font=primary\_font) # Defines a tkinter button.
696. self.change\_p\_button.place(x=110, y=140, w=140, h=30) # Sets the button's geometry.
697. self.change\_p\_button.bind('<Return>', **lambda** event: self.change\_password()) # When pressing the return key while this widget is focused, will shift focus to the next widget.
698. self.change\_p\_button.bind('<Escape>', **lambda** event: self.master.focus()) # Unfocuses the widget when the escape key is pressed while this widget is focused.
700. self.cancel\_button = tk.Button(self.master, text='Cancel', command=**lambda**: [self.master.destroy(), settings\_window.master.focus\_set(), settings\_window.master.grab\_set()], bg=bg\_colour, fg=text\_colour, font=primary\_font) # Defines a tkinter button.
701. self.cancel\_button.place(x=30, y=140, w=65, h=30) # Sets the button's geometry.
702. self.cancel\_button.bind('<Return>', **lambda** event: [self.master.destroy(), settings\_window.master.focus\_set(), settings\_window.master.grab\_set()]) # When pressing the return key while this widget is focused, will shift focus to the next widget.
703. self.cancel\_button.bind('<Escape>', **lambda** event: self.master.focus()) # Unfocuses the widget when the escape key is pressed while this widget is focused.
705. ### 'CHANGE PASSWORD' CLASS FUNCTIONS
706. **def** change\_password(self):
707. '''''This function checks the if the inputted password matches the password in the database. If it does it will overwrite it with the new password. This function is called when the user presses the change password button in this window.'''
708. current\_p = self.current\_p\_var.get() # Get the string value in the tkinter string varaible.
709. new\_p = self.new\_p\_var.get() # Get the string value in the tkinter string varaible.
710. new\_p\_confirm = self.new\_p\_confirm\_var.get() # Get the string value in the tkinter string varaible.
712. **if** current\_p == '' **or** new\_p == '' **or** new\_p\_confirm == '': # If any entry boxes are empty.
713. messagebox.showerror('Error: Empty field(s)', 'Please ensure there are no empty field(s).') # Display a Tkinter error window (popup).
715. **elif** current\_p != user[3]: # If the current password does not match the one in the database.
716. messagebox.showerror('Error: Password Incorrect', 'The current password was incorrect. Please try again.') # Display a Tkinter error window (popup).
718. **elif** new\_p != new\_p\_confirm: # If the new password does not match in both entry boxes.
719. messagebox.showerror('Error: New passwords do not match', 'The new passwords do not match. Please re-enter the passwords.') # Display a Tkinter error window (popup).
721. **elif** current\_p == new\_p: # If the password is not new.
722. messagebox.showerror('Error: Password must be new', 'Your new password matches the old password. Please ensure it is different.') # Display a Tkinter error window (popup).
724. **elif** len(new\_p) < 8: # If the password's length is not atleast 8 characters.
725. messagebox.showerror('Error: Password too weak', 'The password must contain atleast 8 characters.') # Display a Tkinter error window (popup).
727. **else**:
728. **for** i **in** range(len(new\_p)): # For all characters in the new password.
729. **if** list(new\_p)[i].isalpha(): # If the character is alphabetical (a-z or A-Z).
731. **for** j **in** range(len(new\_p)): # For all characters in the new password.
732. **if** list(new\_p)[j].isdigit(): # If the character is a digit (0-9).
734. db\_open()
735. c.execute('UPDATE Staff SET Password=? WHERE StaffID=?', (new\_p, user[0])) # Update the password to the new password if the user id equals the one that's logged in.
736. conn.commit() # Commit the database change.
737. db\_close()
738. user[3] = new\_p # Update the password for the user logged in Python.
739. log('{} {} updated their password.'.format(user[1], user[2])) # Log the change.
741. self.master.destroy() # Destroy the change password window.
742. settings\_window.master.focus\_set() # Set focus back to the settings window.
743. settings\_window.master.grab\_set() # Set grab back to the settings window.
744. **return** # Exit the function early.
746. messagebox.showerror('Error: Password too weak', 'The password must contain atleast 1 digit.') # Display a Tkinter error window (popup).
747. **break** # Exit the for loop as no characters are digits therefore the password is already invalid.
749. messagebox.showerror('Error: Password too weak', 'The password must contain atleast 1 alphabetical character.') # Display a Tkinter error window (popup).
751. self.current\_p\_entry.delete(0, tk.END) # If unsuccessful, clear the entry box.
752. self.new\_p\_entry.delete(0, tk.END) # If unsuccessful, clear the entry box.
753. self.new\_p\_confirm\_entry.delete(0, tk.END) # If unsuccessful, clear the entry box.

756. #======================================================================================
757. #                                           TABLES WINDOW
758. #======================================================================================
759. **class** Tables():
760. '''''This window allows admins to access the database within Python. This function is called when a table is selected in the drop-down menu within the settings window. The window will then display the table selected.'''
761. **def** \_\_init\_\_(self, master, table):
762. '''''This function is run when the class is initialised.'''
763. ### VARIABLES
764. self.master = master # Sets the variable 'master' to an attribute of the class.
765. self.table = table # The table selected (between 0-2).
766. self.entry = None # The variable to hold the Tkinter entry box that will be defined when a field in the table is clicked.
767. self.entry\_var = tk.StringVar() # Defines a Tkinter string variable.
768. self.changes = [] # Defines an array.
770. ### TKINTER WINDOW CONFIG
771. self.master.title('Edit {} Tables (Version {})'.format(settings\_window.tables\_var.get(), version)) # Sets window's title.
772. self.master.geometry('540x360+{}+{}'.format(((screen\_width-540)//2), ((screen\_height-360)//2)))  # Sets window geometry & centers the window.
773. self.master.resizable(False, False) # Sets the window so it cannot be resized.
774. self.master.wm\_iconbitmap('resources/icon.ico') # Sets window's icon.
775. self.master.config(bg=bg\_colour) # Sets background colour of window.
776. self.master.protocol('WM\_DELETE\_WINDOW', **lambda**: self.cancel()) # Runs the following function when the window is exited.
777. self.master.bind('<Button-1>', **lambda** event: self.hide\_entry(click=True)) # Runs the following function when user left clickes anywhere within the window.
778. self.master.focus\_set() # Sets focus on the window.
779. self.master.grab\_set() # Sets grab on the window (so other windows cannot be focused).
781. ### TKINTER BUTTONS
782. self.add\_button = tk.Button(self.master, text='Add', command=**lambda**: self.add\_record(), bg=bg\_colour, fg=text\_colour, font=(secondary\_font, 9)) # Defines a Tkinter button.
783. self.add\_button.place(x=40, y=310, w=50, h=20) # Sets button's geometry.
784. self.add\_button.bind('<Return>', **lambda** event: self.add\_record()) # Runs the following function when the enter key is pressed while this widget is focused.
785. self.add\_button.bind('<Escape>', **lambda** event: self.master.focus()) # Unfocuses the widget when the escape key is pressed.
787. self.remove\_button = tk.Button(self.master, text='Remove', command=**lambda**: self.remove\_record(), bg=bg\_colour, fg=text\_colour, font=(secondary\_font, 9))  # Defines a Tkinter button.
788. self.remove\_button.place(x=90, y=310, w=50, h=20) # Sets button's geometry.
789. self.remove\_button.bind('<Return>', **lambda** event: self.remove\_record()) # Runs the following function when the enter key is pressed while this widget is focused.
790. self.remove\_button.bind('<Escape>', **lambda** event: self.master.focus()) # Unfocuses the widget when the escape key is pressed.
792. self.cancel\_button = tk.Button(self.master, text='Cancel', command=**lambda**: self.cancel(), bg=bg\_colour, fg=text\_colour, font=primary\_font)  # Defines a Tkinter button.
793. self.cancel\_button.place(x=220, y=320, w=75, h=30) # Sets button's geometry.
794. self.cancel\_button.bind('<Return>', **lambda** event: self.cancel()) # Runs the following function when the enter key is pressed while this widget is focused.
795. self.cancel\_button.bind('<Escape>', **lambda** event: self.master.focus()) # Unfocuses the widget when the escape key is pressed.
797. self.apply\_button = tk.Button(self.master, text='Apply', command=**lambda**: self.apply(), bg=bg\_colour, fg=text\_colour, font=primary\_font, relief=tk.SUNKEN)  # Defines a Tkinter button.
798. self.apply\_button.place(x=310, y=320, w=75, h=30) # Sets button's geometry.
799. self.apply\_button.bind('<Return>', **lambda** event: self.apply()) # Runs the following function when the enter key is pressed while this widget is focused.
800. self.apply\_button.bind('<Escape>', **lambda** event: self.master.focus()) # Unfocuses the widget when the escape key is pressed.
802. self.exit\_button = tk.Button(self.master, text='Apply & Exit', command=**lambda**: self.exit\_(), bg=bg\_colour, fg=text\_colour, font=primary\_font)  # Defines a Tkinter button.
803. self.exit\_button.place(x=400, y=320, w=120, h=30) # Sets button's geometry.
804. self.exit\_button.bind('<Return>', **lambda** event: self.exit\_()) # Runs the following function when the enter key is pressed while this widget is focused.
805. self.exit\_button.bind('<Escape>', **lambda** event: self.master.focus()) # Unfocuses the widget when the escape key is pressed.
807. ### GETTING RECORDS AND FIELDS OF TABLE
808. db\_open()
809. **if** self.table == 0: # If Staff table.
810. self.fields = ['StaffID', 'FirstName', 'LastName', 'Password', 'IsAdmin', 'IsHairdresser', 'Rate'] # Defines an array that contains the table's fields.
811. self.field\_descs = ['First name of the staff member.'] # Defines an array that contains the fields' descriptions.
812. self.field\_descs.append('Last name of the staff member.') # Adds a description to the array.
813. self.field\_descs.append('Password for the staff member to log into the booking system.') # Adds a description to the array.
814. self.field\_descs.append('Will this staff member have admin controls where False = 0 and True = 1.') # Adds a description to the array.
815. self.field\_descs.append('Will this staff member be a hairdresser. (Will clients be about to book appointments with this staff member.') # Adds a description to the array.
816. self.field\_descs.append('The rate of the hairdressers prices in percentage, where 100% is default. For example, 200% means the hairdresser charges double the normal rate/price.') # Adds a description to the array.
817. c.execute('SELECT \* FROM Staff') # Retrieves the data from the table.
818. self.records= list(c.fetchall()) # Fetches the data.
819. self.new\_record = [-1, '', '', '', 0, 0, 100] # Defines an array that contains the default values of a record within this table.
820. self.record\_required = [True, True, True, True, True, True] # Defines an array that contains booleans that represent if the field is required.
822. **elif** self.table == 1: # If Client table.
823. self.fields = ['ClientID', 'FirstName', 'LastName', 'Mobile', 'Home']  # Defines an array that contains the table's fields.
824. self.field\_descs = ['First name of client.'] # Defines an array that contains the fields' descriptions.
825. self.field\_descs.append('Last name of client.') # Adds a description to the array.
826. self.field\_descs.append('Mobile phone number to contact the client.') # Adds a description to the array.
827. self.field\_descs.append('Home phone number to contact the client.') # Adds a description to the array.
828. c.execute('SELECT \* FROM Clients') # Retrieves the data from the table.
829. self.records= list(c.fetchall()) # Fetches the data.
830. self.new\_record = [-1, '', '', '', ''] # Defines an array that contains the default values of a record within this table.
831. self.record\_required = [True, True, False, False] # Defines an array that contains booleans that represent if the field is required.
833. **elif** self.table == 2: # If Haircut table.
834. self.fields = ['HaircutID', 'Haircut', 'Price', 'EstimatedTime']  # Defines an array that contains the table's fields.
835. self.field\_descs = ['The name of the haircut.'] # Defines an array that contains the fields' descriptions.
836. self.field\_descs.append('The price of the haircut at normal rate, where £1 = 1.0') # Adds a description to the array.
837. self.field\_descs.append('Estimated time the appointment will take in factor of 15 where 15 minutes = 15.') # Adds a description to the array.
838. c.execute('SELECT \* FROM Haircuts') # Retrieves the data from the table.
839. self.records= list(c.fetchall()) # Fetches the data.
840. self.new\_record = [-1, '', '', ''] # Defines an array that contains the default values of a record within this table.
841. self.record\_required = [True, True, True] # Defines an array that contains booleans that represent if the field is required.
843. self.id\_labels = [] # Defines an array that will contain the ids of the records.
844. **for** i **in** range(len(self.records)): # For all records.
845. self.records[i] = list(self.records[i]) # Convert tuple to array/list.
846. **if** self.table == 0: # If Staff table.
847. c.execute('SELECT Rate FROM Hairdressers WHERE StaffID=?', (self.records[i][0],)) # Retrieve the hairdresser's rate for this staff member.
848. result = c.fetchone() # Fetch the data.
849. **if** result == None: # If this staff member isn't a hairdresser.
850. self.records[i].append(0) # Set the isHairdresser coloumn to False (0).
851. self.records[i].append('') # Set the rate to empty.
852. **else**:
853. self.records[i].append(1) # Set the isHairdresser coloumn to True (1).
854. self.records[i].append(result[0]) # Set the rate to the rate retrieved from the database.
855. self.id\_labels.append(tk.Label(self.master, text=self.records[i][0], bg=bg\_colour, fg=text\_colour, font=(primary\_font, 9), anchor='e')) # Add label that contains the id for this record.
856. **if** i <= 16: # If the record is within the first 16.
857. self.id\_labels[i].place(x=20, w=20, h=16, y=16\*i+40) # Place the label next to the corresponding record.
858. **else**:
859. self.id\_labels[i].place(x=20, w=20, h=16, y=360) # Place the label outside the window.
860. db\_close()
862. ### TKINTER FRAMES
863. self.table\_frame = tk.Frame(self.master) # Defines a Tkinter frame.
864. self.table\_frame.place(x=40, y=10, w=480, h=300) # Sets the frame's geometry.
865. self.table\_frame.rowconfigure(1, weight=1) # Adds a weight to the first row in the frame.
867. self.fields\_frame = tk.Frame(self.table\_frame, height=20) # Defines a Tkinter frame.
868. self.fields\_frame.grid(row=0, column=0, sticky='new', columnspan=len(self.fields)-1) # Sets the frame's geometry.
870. ### TKINTER LABEL
871. self.id\_label = tk.Label(self.master, bg=bg\_colour, fg=text\_colour, text='ID', font=(primary\_font, 9)) # Defines a Tkinter label.
872. self.id\_label.place(x=20, y=10, w=20, h=20) # Sets label's geometry.
874. self.remove\_info\_label = tk.Label(self.master, bg=bg\_colour, fg=text\_colour, font=(primary\_font, 7), text='', anchor='w') # Defines a Tkinter label.
875. self.remove\_info\_label.place(x=0, y=330, w=200, h=30) # Sets label's geometry.
877. ### SCROLLBAR
878. self.scroll = ttk.Scrollbar(self.table\_frame, orient='vertical', command=self.on\_scroll\_by\_bar) # Defines a Tkinter scrollbar.
879. self.scroll.grid(row=0, column = len(self.fields)-1, sticky='ns', rowspan = 2) # Set's the scrollbar's geometry.
881. ### POPULATING TKINTER FRAMES WITH LISTBOXES AND LABELS
882. self.field\_labels = [] # Defines an array that will contain the field headings.
883. self.lbs = [] # Defines an array that will contain the list boxes.
884. **for** i **in** range(len(self.fields)-1): # For each field excluding the id.
885. self.table\_frame.columnconfigure(i, weight=1) # Give it a weight.
886. self.fields\_frame.update\_idletasks() # Update the geometry of the frame.
887. self.field\_labels.append(tk.Label(self.fields\_frame, text=self.fields[i+1], bg=bg\_colour, fg=text\_colour, font=(primary\_font, 9))) # Appends a Tkinter label to the array.
888. self.field\_labels[i].place(x=self.fields\_frame.winfo\_width()\*i/(len(self.fields)-1), y=0, w=self.fields\_frame.winfo\_width()/(len(self.fields)-1), h=20) # Sets the label's geometry.
889. self.field\_labels[i].field = i # Adds an attribute to the label that contains the field.
890. self.field\_labels[i].bind('<Button-1>', **lambda** event: messagebox.showinfo(('Info: Help with ' + self.fields[event.widget.field+1]), self.field\_descs[event.widget.field])) # Displays a Tkinter information window (popup) when the field's heading it left clicked.
891. self.field\_labels[i].bind('<Enter>', **lambda** event: event.widget.configure(font=(primary\_font, 9, 'underline'))) # When the mouse cursor is hovering above this label, the text becomes underlined to indicate it's clickable.
892. self.field\_labels[i].bind('<Leave>', **lambda** event: event.widget.configure(font=(primary\_font, 9))) # When the mouse cursor is no longer hovering above this label, the text is reverted to its normal state.
894. ### TKINTER LISTBOXES
895. self.lbs.append(tk.Listbox(self.table\_frame, yscrollcommand=self.on\_scroll\_by\_wheel, fg=text\_colour, font=(secondary\_font, 9))) # Appends a Tkinter list box to the array.
896. self.lbs[i].grid(row=1, column=i, sticky='nesw') # Set's the list boxes' geometry.
897. self.lbs[i].bind('<<ListboxSelect>>', **lambda** event: self.force\_show\_entry(event)) # Run the following functions when the list box is left clicked.
898. self.lbs[i].bind('<FocusIn>', **lambda** event: self.focus\_fix()) # Run the 'focus\_fix' function when the list box is focused.
899. self.lbs[i].field = i # Adds an attribute to the list box that contains the field.
901. **for** j **in** range(len(self.records)): # For all records.
902. **if** self.records[j][i+1] == None: # If the field of this record is empty.
903. self.records[j][i+1] = '' # Set it to an empty string instead.
904. self.lbs[i].insert(tk.END, 'None') # Insert the string 'None' into the list box.
905. self.lbs[i].itemconfig(j, foreground=primary\_colour) # Change the colour of 'None' to show it's not a value.
906. **else**:
907. self.lbs[i].insert(tk.END, self.records[j][i+1]) # Otherwise insert the data into the list box.
909. ### TKINTER ENTRY BOXES
910. self.entry = tk.Entry(self.master, textvariable=self.entry\_var, fg=text\_colour, font=(secondary\_font, 9)) # Defines a Tkinter entry box.
911. self.entry.bind('<FocusOut>', **lambda** event: self.focus\_fix()) # Run the 'focus\_fix' function when the entry box loses focus.
912. self.entry.bind('<Escape>', **lambda** event: self.hide\_entry(cancel=True)) # Run the following function when the escape key is pressed while the entry box is focused.
913. self.entry.bind('<Return>', **lambda** event: self.hide\_entry()) # Run the following function when the return key is pressed while the entry box is focused.
914. self.entry.bind('<Enter>', **lambda** event: self.on\_hover(True)) # Run the following function when the cursor hovers over the entry box.
915. self.entry.bind('<Leave>', **lambda** event: self.on\_hover(False)) # Run the following function when the cursor leaves the entry box.
916. self.entry.hover = False # Defines a boolean.
917. self.entry.hidden = True # Defines a boolean.
918. self.entry.forced = None # Defines a boolean.
920. ### 'TABLES' CLASS FUNCTIONS
921. **def** on\_hover(self, hover):
922. '''''This function runs when the cursor moves over the entry box. It is required so when the user left clicks we know if they've left clicked the entry box.'''
923. self.entry.hover = hover
925. **def** focus\_fix(self):
926. '''''This function fixes which widget is focused depending on if the entry box is present. This function is a fix to a bug.'''
927. **if** self.entry.hidden == True:
928. self.master.focus()
929. **else**:
930. self.entry.focus()
932. **def** force\_show\_entry(self, event):
933. '''''This function forces the entry box to appear when hide\_entry is next called (which is called straight after by the left click binding). This is required so the entry box is not hidden straight away. This function is a fix to a bug.'''
934. **if** self.entry.hidden == True: # If the entry's attribute 'hidden' is True.
935. self.entry.forced = event # Set the entry's attribute 'forced' to the event of the widget that called this function.
937. **def** hide\_entry(self, cancel=False, click=False):
938. '''''This function hides the entry box and checks if the new value is valid data for this field. This function is called when the user clicks off the entry box or presses escape or enter.'''
939. **if** self.entry.forced != None: # If the entry needs to be forced to show.
940. lb = self.entry.forced.widget # Gets the widget from the event.
941. **if** **not**(self.table == 0 **and** lb.field == 5 **and** self.records[int(lb.curselection()[0])][5] == 0): # If the clicked listbox is not rates for a staff member that isn't a hairdresser.
942. self.show\_entry(self.entry.forced) # Calls the function to show the entry box.
943. lb.selection\_clear(0, tk.END) # Clears the selection of the listbox.
944. self.entry.forced = None # Makes the entry box no longer be forced to show.
946. **elif** (click == False **or** self.entry.hover == False) **and** self.entry.hidden == False: # If entry box is not hidden and the left click that called this function was not on the entry box.
947. self.entry.lower(self.table\_frame) # Hide the entry box under the table frame.
948. self.entry.hidden = True # Mark the entry box as hidden (change entry boxes' attribute).
949. new\_val = self.entry\_var.get() # Retrieve the string value from the Tkinter string variable.
950. **if** self.table != 0 **or** self.entry.field != 2: # If the selected field isn't the password.
951. new\_val = new\_val.replace(' ', '').lower() # Remove all space characters and then make all character lower case.
952. **for** lb **in** self.lbs: # For all list boxes.
953. lb.selection\_clear(0, tk.END) # Clear the selection.
954. self.entry.delete(0, tk.END) # Clear the contents of the entry box.
956. **if** cancel == True **or** new\_val == str(self.records[self.entry.record][self.entry.field+1]): # If the user clicked escape on the entry box or the value is equal to the value that was there before.
957. **return** # Exit function early.
958. **elif** new\_val == '': # If the new value is nothing.
959. self.records[self.entry.record][self.entry.field+1] = new\_val # Update the records with this new value.
960. self.lbs[self.entry.field].delete(self.entry.record) # Remove the old value from the list box.
961. **if** self.record\_required[self.entry.field] == True: # If this field is required.
962. self.lbs[self.entry.field].insert(self.entry.record, 'Required') # Add 'Required' to the list box in place of the old value.
963. self.lbs[self.entry.field].itemconfig(self.entry.record, foreground='red') # Make 'Required' red to show it's not a value.
964. **return** # Exit function early.
965. **else**:
966. self.lbs[self.entry.field].insert(self.entry.record, 'None') # Add 'None' to the list box in place of the old value.
967. self.lbs[self.entry.field].itemconfig(self.entry.record, foreground=primary\_colour) # Make 'None' a different colour to show it's not a value.
968. **return** # Exit function early.
969. ### VALIDATION
970. ### IF FIELD IS NAME
971. **elif** (self.table **in** {0, 1} **and** self.entry.field **in** {0, 1}) **or** (self.table == 2 **and** self.entry.field == 0): # If the field is a name.
972. **if** new\_val.isalpha(): # Check that the string only consists of alphabetical characters (a-z or A-Z).
973. new\_val = new\_val.title() # Capitalise the first character of the string.
974. **if** new\_val == self.records[self.entry.record][self.entry.field+1]: # If the new value is equal to the old value.
975. **return** # Exit function early.
976. **else**:
977. messagebox.showerror('Error: Incorrect data type', 'This field must only contain alphabetical characters.') # Displays a tkinter error window (popup) with the following message.
978. **return** # Exit function early.
979. ### IF FIELD IS PASSWORD
980. **elif** self.table == 0 **and** self.entry.field == 2: # If the field is the password.
981. **if** len(new\_val) < 8: # If length of the password is less than 8 characters.
982. messagebox.showerror('Error: Password too weak', 'The password must contain atleast 8 characters.') # Displays a tkinter error window (popup) with the following message.
983. **return** # Exit function early.
984. **else**:
985. alpha\_exists = False # Defines a boolean to keep track if there is atleast one alphabetical character.
986. digit\_exists = False # Defines a boolean to keep track if there is atleast one digit.
987. **for** i **in** range(len(new\_val)): # For all characters in the password.
988. **if** list(new\_val)[i].isalpha(): # If the character is alphabetical (a-z or A-Z).
989. alpha\_exists = True # Set the boolean to True.
990. **break** # End the loop early.
991. **for** j **in** range(len(new\_val)): # For all characters in the password.
992. **if** list(new\_val)[j].isdigit(): # If the character is a digit (0-9).
993. digit\_exists = True # Set the boolean to True.
994. **break** # End the loop early.
995. **if** alpha\_exists == False: # If there was no alphabetical characters.
996. messagebox.showerror('Error: Password too weak', 'The password must contain atleast 1 digit.') # Displays a tkinter error window (popup) with the following message.
997. **return** # Exit function early.
998. **elif** digit\_exists == False: # If there was no digits.
999. messagebox.showerror('Error: Password too weak', 'The password must contain atleast 1 alphabetical character.') # Displays a tkinter error window (popup) with the following message.
1000. **return** # Exit function early.
1001. ### IF FIELD IS BOOLEAN
1002. **elif** self.table == 0 **and** self.entry.field **in** {3, 4}: # If Staff Table and a boolean field.
1003. **if** new\_val **in** {'1', 'true', 'yes'}: # If value is one of the following.
1004. **if** self.records[self.entry.record][self.entry.field+1] == 1: # If the value has not changed.
1005. **return** # Exit function early.
1006. new\_val = 1 # Otherwise set new value to 1.
1007. **elif** new\_val **in** {'0', 'false', 'no'}: # If value is one of the following.
1008. **if** self.records[self.entry.record][self.entry.field+1] == 0: # If the value has not changed.
1009. **return** # Exit function early.
1010. **if** self.entry.field == 3: # If the field is 'isAdmin'.
1011. ### CHECK FOR ATLEAST ONE OTHER ADMIN
1012. admins = 0 # Counter for the amount of admins.
1013. **for** record **in** self.records: # For all records.
1014. **if** record[4] == 1: # If record is admin.
1015. admins += 1 # Add one to counter.
1016. **if** admins == 2: # When counter reaches two.
1017. **break** # End for loop early.
1018. **if** admins < 2: # If there is not atleast two admins.
1019. messagebox.showerror('Error: Atleast one admin must exist', 'There must be atleast one admin.') # Displays a tkinter error window (popup) with the following message.
1020. **return** # Exit function early.
1021. new\_val = 0 # If there is atleast two admins, then set new value to 0.
1022. **else**:
1023. messagebox.showerror('Error: Incorrect data type', 'This field must be a boolean.') # Displays a tkinter error window (popup) with the following message.
1024. **return** # Exit function early.
1025. ### IF FIELD IS NUMBER
1026. **elif** (self.table == 0 **and** self.entry.field == 5) **or** (self.table == 1 **and** self.entry.field **in** {2,3}) **or** (self.table == 2 **and** self.entry.field == 2): # If field consisting of only digits.
1027. **if** new\_val.isdigit(): # If new value consists of only digits.
1028. **if** self.table != 1 **or** **not** self.entry.field **in** {2,3}: # If field is not Clients' phone numbers.
1029. new\_val = int(new\_val) # Convert string to integer.
1030. **if** new\_val == self.records[self.entry.record][self.entry.field+1]: # If the value has not changed.
1031. **return** # Exit function early.
1032. **if** self.table == 2 **and** self.entry.field == 2 **and** new\_val % 15 != 0: # If the field is appointment time and isn't factor of 15.
1033. messagebox.showerror('Error: Incorrect data type', 'This field must be a factor of 15.') # Displays a tkinter error window (popup) with the following message.
1034. **return** # Exit function early.
1035. **else**:
1036. messagebox.showerror('Error: Incorrect data type', 'This field must only contain digits.') # Displays a tkinter error window (popup) with the following message.
1037. **return** # Exit function early.
1038. ### IF FIELD IS FLOAT/REAL
1039. **elif** self.table == 2 **and** self.entry.field == 1: # If table is Haircuts and field is price.
1040. **try**: # Try the following code.
1041. new\_val = float(new\_val) # Convert the new value to a float. If successful no errors will be produced.
1042. **if** new\_val == self.records[self.entry.record][self.entry.field+1]: # If the value has not changed.
1043. **return** # Exit function early.
1044. **if** new\_val < 0: # If value is negative.
1045. messagebox.showerror('Error: Incorrect data type', 'This float must be postive.') # Displays a tkinter error window (popup) with the following message.
1046. **return** # Exit function early.
1047. **except** TypeError: # If an error was produced when converting the new value to a float (In other words, the variable was not a float/real).
1048. messagebox.showerror('Error: Incorrect data type', 'This field must be a float.') # Displays a tkinter error window (popup) with the following message.
1049. **return** # Exit function early.
1050. ### IF NEW VALUE PASSES ALL VALIDATION
1051. self.records[self.entry.record][self.entry.field+1] = new\_val # Update records with new value.
1052. self.changes.append(['update', self.records[self.entry.record][0], self.entry.field, new\_val]) # Add the change to the changes array to be applied when the user chooses to.
1053. self.apply\_button.config(relief=tk.RAISED) # Set button's state to raised.
1054. self.lbs[self.entry.field].delete(self.entry.record) # Delete the old value from the list box.
1055. self.lbs[self.entry.field].insert(self.entry.record, new\_val) # Insert the new value into the list box in place of the old.
1056. **if** self.table == 0 **and** self.entry.field == 4 **and** new\_val == 1: # If Staff table, field is 'isHairdresser' and value is 1.
1057. self.records[self.entry.record][6] = 100 # Make 'Rate' field equal to 1.
1058. self.lbs[5].delete(self.entry.record) # Delete the old value from the list box.
1059. self.lbs[5].insert(self.entry.record, 100) # Insert the new value into the list box in place of the old.
1060. **elif** self.table == 0 **and** self.entry.field == 4 **and** new\_val == 0: # If Staff table, field is 'isHairdresser' and value is 0.
1061. self.records[self.entry.record][6] = '' # Make 'Rate' field equal to empty string.
1062. self.lbs[5].delete(self.entry.record) # Delete the old value from the list box.
1063. self.lbs[5].insert(self.entry.record, '') # Insert the new value into the list box in place of the old.

1066. **def** show\_entry(self, event):
1067. '''''This function is called when the user wants to edit a record. It summons an entry box over the record so the user can edit the value.'''
1068. self.entry.field = event.widget.field # Get field of from list boxes' attribute.
1069. self.entry.record = int(event.widget.curselection()[0]) # Get the record that is selected in the list box.
1071. ### IF RECORD NEEDS TO BE REMOVED
1072. **if** self.remove\_button['relief'] == tk.SUNKEN: # If the button's state is sunken:
1073. self.remove\_button.config(relief=tk.RAISED) # Set the button's state to raised.
1074. self.remove\_info\_label.config(text='') # Empty the string containing information on how to remove.
1075. **for** i **in** range(len(self.records)): # For all records.
1076. **if** self.table != 0 **or** (self.records[i][4] == 1 **and** self.entry.record != i): # If atleast one other admin exists and it's not itself.
1077. self.id\_labels[self.entry.record].destroy() # Destroy the label that displays the record's primary key.
1078. self.id\_labels.remove(self.id\_labels[self.entry.record]) # Remove the label from the array that contained it.
1079. self.changes.append(['remove', self.records[self.entry.record][0]]) # Add the change to the changes array to be applied when the user chooses to.
1080. self.apply\_button.config(relief=tk.RAISED) # Set button's state to raised.
1081. self.records.remove(self.records[self.entry.record]) # Remove this record from the 2D array.
1082. **for** lb **in** self.lbs: # For all list boxes.
1083. lb.delete(self.entry.record) # Remove this record.
1084. **return** # End function early.
1085. messagebox.showerror('Error: Atleast one admin must exist', 'There must be atleast one admin.') # Displays a tkinter error window (popup) with the following message.
1086. ### IF RECORD IS BEING EDITED
1087. **else**:
1088. self.entry.lift(self.table\_frame) # Lift the entry box that was hidding behind the table frame.
1089. self.entry.hidden = False # Set it's attribute to no longer being hidden.
1090. self.entry.yview = self.lbs[0].yview() # Take note of the list box.
1091. self.entry\_var.set(self.records[self.entry.record][self.entry.field+1]) # Set the value of the entry box to the value of the record where the entry box will be placed.
1093. y = int(self.entry.record-self.entry.yview[0]\*len(self.records))\*16+30 # Calculate the y position that the entry box needs to be plaed at.
1094. self.entry.place(x=self.fields\_frame.winfo\_width()\*self.entry.field/(len(self.fields)-1)+40, y=y, w=self.fields\_frame.winfo\_width()/(len(self.fields)-1), h=16) # Set's the entry boxes' geometry.
1095. self.entry.icursor(tk.END) # Set's the cursor at the end of the entry box.
1097. **def** add\_record(self):
1098. '''''This function is called when the user clicks the add record button. It adds a record at the bottom of the table and adds it to the record array with that table's default values for each field.'''
1099. ### CALCULATES THE NEW PRIMARY KEY FOR THE RECORD.
1100. self.records.append([self.new\_record[0]]) # Adds an array to the records array (2D array).
1101. **for** i **in** range(len(self.records)-1): # For all records excluding self.
1102. **if** self.records[len(self.records)-1][0] < self.records[i][0]: # If this record has a higher primary key than the new record.
1103. self.records[len(self.records)-1][0] = self.records[i][0] # Set the new record's primary key equal to that record.
1104. self.records[len(self.records)-1][0] += 1 # Add one to the highest primary key to ensure it's unique.
1105. self.id\_labels.append(tk.Label(self.master, text=self.records[len(self.records)-1][0], bg=bg\_colour, fg=text\_colour, font=(primary\_font, 9), anchor='e')) # Defines a Tkinter label that contains this new record's primary key.
1106. self.id\_labels[len(self.id\_labels)-1].place(x=20, w=20, h=16, y=360) # Set's the record's geometry.
1107. self.changes.append(['add', self.records[len(self.records)-1][0]]) # Add the change to the changes array to be applied when the user chooses to.
1108. self.apply\_button.config(relief=tk.RAISED) # Set's the button's state as raised as there's unsaved changes.
1110. **for** i **in** range(len(self.fields)-1): # For all fields excluding the first.
1111. self.records[len(self.records)-1].append(self.new\_record[i+1]) # Add the field to the new record.
1112. **if** self.table == 0 **and** i == 5: # If Staff table and 'isHairdresser' field.
1113. self.lbs[i].insert(tk.END, '') # Insert empty string.
1114. **elif** self.new\_record[i+1] == '' **and** self.record\_required[i] == True: # If field is empty and required.
1115. self.lbs[i].insert(tk.END, 'Required') # Set list box to show 'Required'.
1116. self.lbs[i].itemconfig(len(self.records)-1, foreground='red') # Change the colour to red to show it's not a value.
1117. **elif** self.new\_record[i+1] == '': # If field is empty and not required.
1118. self.lbs[i].insert(tk.END, 'None') # Set list box to show 'None'.
1119. self.lbs[i].itemconfig(len(self.records)-1, foreground=primary\_colour) # Change the colour of this to show it's not a value.
1120. **else**:
1121. self.lbs[i].insert(tk.END, self.records[len(self.records)-1][i+1]) # Otherwise insert the record value into the list box.
1123. self.on\_scroll\_by\_wheel(1.0-16/len(self.records), 1.0) # Set the yview of all the listboxes down by a record to account for the new record at the bottom.
1125. **def** remove\_record(self):
1126. '''''This function is run when the remove button is clicked by the user. This will cause the next record to be pressed to be deleted. This button is toggleable.'''
1127. **if** self.remove\_button['relief'] == tk.RAISED: # If the button's in a raised state.
1128. self.remove\_button.config(relief=tk.SUNKEN) # Set the button's state to sunken.
1129. self.remove\_info\_label.config(text="The next record you click will be removed.\nClick 'Remove' again to cancel the removal.") # Make the label close to the button display infomation when the button is toggled on.
1130. **else**:
1131. self.remove\_button.config(relief=tk.RAISED) # Set the button's state to raised.
1132. self.remove\_info\_label.config(text='') # Remove the information displayed by making the displayed string empty.
1134. **def** on\_scroll\_by\_bar(self, \*args):
1135. '''''This function is run when the user scrolls with the scroll bar. It syncs all the listboxes to the new yview of the scroll bar.'''
1136. **if** self.entry.hidden == True: # Only scroll if the entry box is not present.
1137. **for** i **in** range(len(self.fields)-1): # For all fields excluding the first one.
1138. self.lbs[i].yview(\*args) # Sync the yview of this list box to the yview of the scroll bar.
1139. **if** float(args[1]) >=0 **and** float(args[1]) + 16/len(self.records) <= 1: # If the primary keys need re-syncing with the records.
1140. **for** i **in** range(len(self.id\_labels)): # For all primary key labels.
1141. **if** i - float(args[1]) \* len(self.records) <= 16: # If the corresponding record is on screen.
1142. self.id\_labels[i].place(y=int(i-float(args[1])\*len(self.records))\*16+30) # Place the primary key label next to the corresponding record.
1143. **else**:
1144. self.id\_labels[i].place(y=360) # Otherwise place the primary key label off screen (outside the window).
1146. **def** on\_scroll\_by\_wheel(self, \*args):
1147. '''''This function is triggered when the user scrolls using the scroll wheel on a list box. This sets all the other list boxes yview to the yview of the scrolled list box.'''
1148. **if** len(self.lbs) + 1 == len(self.fields): # If the list box is one of the fields.
1149. **if** self.entry.hidden == True: # If the entry box is currently hidden.
1150. yview = args[0] # Define a new float with the value of the first argument/parameter.
1151. self.scroll.set(\*args) # Set/sync the scrollbar's yview to the new yview of the list box.
1152. **else**:
1153. yview = self.entry.yview[0] # Else undo the change in yview of the list box that called this function.
1154. **for** i **in** range(len(self.fields)-1): # For all listboxes.
1155. self.lbs[i].yview\_moveto(yview) # Move the yview by the change in yview caused by the scroll.
1156. **for** i **in** range(len(self.id\_labels)): # For all primary key labels.
1157. line = int(i - float(yview) \* len(self.records)) # Calculate the line of the record where the first record is 1 and the nth record is n.
1158. **if** line >= 0 **and** line <= 16: # If in the first 16 records (In other words, if the corresponding record is no screen).
1159. self.id\_labels[i].place(y=line\*16+30) # Place the primary key label next to the corresponding label.
1160. **else**:
1161. self.id\_labels[i].place(y=360) # Otherwise, place the primary key label off screen (outside the window).


1165. **def** apply(self):
1166. '''''This function is called when the user clicks the apply button. It applies all the changes made in the table to the database.'''
1167. **if** self.apply\_button['relief'] == tk.RAISED: # If the apply button is raised (In other words, if there are any unsaved changes).
1168. **for** i **in** range(len(self.fields)-1): # For all fields excluding the first one.
1169. **for** j **in** range(len(self.records)): # For all records.
1170. **if** self.lbs[i].get(j) == 'Required' **and** self.records[j][i+1] == '': # If the listbox contains an empty field that is required.
1171. messagebox.showerror('Error: Required Fields', 'There are required fields are are empty. Please fill these in before continuing.') # Displays a Tkinter message window (popup).
1172. **return** False # End the function early with the boolean value False to show it was unsuccessful.
1174. **if** self.table == 0: # If Staff Table.
1175. table\_name = 'Staff' # Define a string with the following value.
1176. **elif** self.table == 1: # If Staff Table.
1177. table\_name = 'Clients' # Define a string with the following value.
1178. **elif** self.table == 2: # If Staff Table.
1179. table\_name = 'Haircuts' # Define a string with the following value.
1181. db\_open()
1182. **for** i **in** range(len(self.changes)): # For all changes made to the table.
1183. **if** self.changes[i][0] == 'add': # If the change is a new record.
1184. **if** self.table == 0: # If Staff Table.
1185. c.execute('INSERT INTO Staff (StaffID, FirstName, LastName, Password, IsAdmin) VALUES (?, "placeholder", "placeholder", "placeholder", 0)', (self.changes[i][1],)) # Insert a new record.
1186. **elif** self.table == 1: # If Client Table.
1187. c.execute('INSERT INTO Clients (ClientID, FirstName, LastName) VALUES (?, "placeholder", "placeholder")', (self.changes[i][1],)) # Insert a new record.
1188. **elif** self.table == 2: # If Haircut Table.
1189. c.execute('INSERT INTO Haircuts (HaircutID, Haircut, Price, EstimatedTime) VALUES (?, "placeholder", 1.0, 15)', (self.changes[i][1],)) # Insert a new record.
1190. **if** self.changes[i][0] == 'remove': # If the change is removing an existing record.
1191. c.execute('DELETE FROM {} WHERE {} = {}'.format(table\_name, self.fields[0], self.changes[i][1])) # Delete a record.
1192. **if** self.table == 0: # If Staff Table.
1193. c.execute('SELECT \* FROM Hairdressers WHERE StaffID = ?', (self.changes[i][1],)) # Delete a record.
1194. **if** c.fetchone() != None: # If this staff member is a hairdresser.
1195. c.execute('DELETE FROM Hairdressers WHERE StaffID = ?', (self.changes[i][1],)) # Delete a record.
1196. main\_window.update = True # Set the main\_window to be updated after returning back to the main window.
1197. **if** self.changes[i][0] == 'update': # If the change is a change in value of a field in a record.
1198. **if** self.table == 0 **and** self.changes[i][2] == 4: # If Staff table and field is 'isHairdresser'.
1199. **if** self.changes[i][3] == 0: # If the value of 'isHairdresser' is 0.
1200. c.execute('DELETE FROM Hairdressers WHERE StaffID = ?', (self.changes[i][1],)) # Delete a record.
1201. **elif** self.changes[i][3] == 1: # If the value of 'isHairdresser' is 1.
1202. c.execute('INSERT INTO Hairdressers (StaffID, Rate) VALUES (?, 100)', (self.changes[i][1],)) # Add a record.
1203. main\_window.update = True # Set the main\_window to be updated after returning back to the main window
1204. **elif** self.table == 0 **and** self.changes[i][2] == 5: # If Staff table and field is 'Rate'.
1205. c.execute('UPDATE Hairdressers SET Rate = ? WHERE StaffID = ?', (self.changes[i][3], self.changes[i][1])) # Change a value of the 'Rate' field where Staff ID another value.
1206. **else**:
1207. **if** isinstance(self.changes[i][3], str): # If the value is a instance of the class string/str (In other words, is this variable a string).
1208. self.changes[i][3] = ('"{}"'.format(self.changes[i][3])) # Surround the string in quotations so mysqlite3 knows it's a string.
1209. c.execute('UPDATE {} SET {} = {} WHERE {} = {}'.format(table\_name, self.fields[self.changes[i][2]+1], self.changes[i][3], self.fields[0], self.changes[i][1])) # Change a value of a field in a record.
1211. conn.commit() # Commit the changes to the database.
1212. db\_close()
1213. self.changes = [] # Defines an empty array.
1214. self.apply\_button.config(relief=tk.SUNKEN) # Set the button's state as sunken.
1215. log('Database was updated by {} {}.'.format(user[1], user[2])) # Log that the database was altered by an admin.
1216. **return** True # End the function early with the boolean value True to show it was successful.
1218. **def** exit\_(self): # This function is called 'exit\_' as 'exit' is a reserved function for Python.
1219. '''''This function is called when the user clicks the exit button. It applies changes and then closes the tables window.'''
1220. **if** self.apply(): # If apply is successful (returns True).
1221. self.master.destroy() # Destroy the tables window.
1222. settings\_window.master.focus\_set() # Set the focus to the settings window.
1223. settings\_window.master.grab\_set() # Set the grab to the settings window, so no other windows can take focus.
1225. **def** cancel(self):
1226. '''''This function is called when the user clicks the cancel button. It discards all changes and then closes the tables window.'''
1227. **if** self.apply\_button['relief'] == tk.RAISED: # If the button's state is raised.
1228. **if** **not** messagebox.askokcancel('Confirmation: Unsaved Changes', 'Do you want to quit? Any unsaved changes will be lost.'): # Display a Tkinter window (popup) to confirm the user wants to discard the changes.
1229. **return** # End the function early.
1230. self.master.destroy() # Destroy the tables window.
1231. settings\_window.master.focus\_set() # Set the focus to the settings window.
1232. settings\_window.master.grab\_set() # Set the grab to the settings window, so no other windows can take focus.
1234. #======================================================================================
1235. #                          ADDING / MANAGING APPOINTMENTS WINDOW
1236. #======================================================================================
1237. **class** Appointments():
1238. **def** \_\_init\_\_(self, master, lb, line):
1239. '''''This function is run when the class is initialised.'''
1240. ### VARIABLES
1241. self.master = master # Defines variable 'master' as attribute of the Appointments class.
1242. self.lb = lb # Defines variable 'lb' as attribute of the Appointments class.
1243. self.line = line # Defines variable 'line' as attribute of the Appointments class.
1244. **global** main\_window # Declare the variable 'main\_window' as global.
1245. self.day = main\_window.nb.index(main\_window.nb.select()) # Define an integer as the working day of the week selected (In other words, the tab from left to right).
1246. self.id = main\_window.lbs[self.day][self.lb+1].appointment\_id[self.line] # Retrieves the appointment id from the list boxes' attributes.
1248. ### TKINTER WINDOW CONFIG
1249. self.master.geometry('540x360+{}+{}'.format(((screen\_width-540)//2), ((screen\_height-360)//2)))  # Sets window geometry and centers the window.
1250. self.master.resizable(False, False) # Sets the window so it cannot be resized.
1251. self.master.wm\_iconbitmap('resources/icon.ico') # Sets the window's icon.
1252. self.master.config(bg=bg\_colour) # Sets background colour of window.
1253. self.master.focus\_set() # Sets the focus on this window.
1254. self.master.grab\_set() # Sets the grap on this window, so other windows cannot take the focus.
1255. self.master.protocol('WM\_DELETE\_WINDOW', **lambda**: [self.master.destroy(), main\_window.lbs[self.day][self.lb+1].selection\_clear(0, tk.END)]) # Run the following functions when the user closes the window.
1257. ### CONNECTION TO DATABASE
1258. db\_open()
1260. c.execute('SELECT Rate FROM Hairdressers') # Retrieve all the rate from the Hairdressers table.
1261. self.rate = c.fetchall()[self.lb][0] # Fetch the rate for this hairdresser from the query.
1263. c.execute('SELECT ClientID, FirstName, LastName FROM Clients') # Retrives the ClientID, FirstName, LastName from the Clients table.
1264. self.clients = c.fetchall() # Fetch the results from the query.
1266. c.execute('SELECT HaircutID, Haircut, Price, EstimatedTime FROM Haircuts') # Retrieve all records from the Haircuts table.
1267. self.haircuts = c.fetchall() # Fetch the results from the query.
1269. ### ADDING NEW APPOINTMENT
1270. **if** self.id == -1: # If there is no appointment in this appointment slot.
1271. db\_close()
1273. ### MORE TKINTER WINDOWS CONFIG
1274. self.master.title('Add Appointment (Version {})'.format(version)) # Sets window's title.
1276. ### VARIABLES
1277. self.client\_names = [] # Defines an array.
1278. self.haircut\_names = [] # Defines an array.
1279. self.durations = [15, 30, 45, 60, 75, 90, 105, 120] # Defines an array.
1281. self.client\_var = tk.StringVar() # Defines a Tkinter string variable.
1282. self.haircut\_var = tk.StringVar() # Defines a Tkinter string variable.
1283. self.duration\_var = tk.StringVar() # Defines a Tkinter string variable.
1285. ### POPULATING LISTS
1286. **for** i **in** range(len(self.clients)): # For all clients.
1287. self.client\_names.append('{} {}'.format(self.clients[i][1], self.clients[i][2])) # Put the client's first name and surname together to make it an option for a drop-down menu.
1288. self.client\_names.append('--- Add new client ---') # Add a last option to add a new client.
1290. **for** i **in** range(len(self.haircuts)): # For all haircuts.
1291. self.haircut\_names.append(self.haircuts[i][1] + ' (Estimated Price: £' + '{:0.2f}'.format(self.haircuts[i][2] \* self.rate \* 0.01) + ')') # Put's the haircut name, price and hairdresser's rate together to make it an option for a drop-down menu.
1293. **for** i **in** range(len(self.durations)): # For all durations.
1294. self.durations[i] = str(self.durations[i]) + ' minutes' # Makes each duration an option for a drop-down menu.
1296. ### TKINTER LABELS
1297. self.time\_label = tk.Label(self.master, text='Start Time:', bg=bg\_colour, fg=text\_colour, font=(primary\_font, 12), anchor='e') # Defines a Tkinter label.
1298. self.time\_label.place(x=0, y=20, w=165, h=30) # Sets the geometry of the label.
1299. self.time\_ans\_label = tk.Label(self.master, text=time.strftime('%M:%S', time.gmtime(first\_appointment\_time + self.line \* appointment\_intervals)), bg=bg\_colour, fg=text\_colour, font=(primary\_font, 12), anchor='w') # Defines a Tkinter label.
1300. self.time\_ans\_label.place(x=170, y=20, w=250, h=30) # Sets the geometry of the label.
1302. self.hairdresser\_label = tk.Label(self.master, text='Hairdresser:', bg=bg\_colour, fg=text\_colour, font=(primary\_font, 12), anchor='e') # Defines a Tkinter label.
1303. self.hairdresser\_label.place(x=0, y=70, w=165, h=30) # Sets the geometry of the label.
1304. self.hairdresser\_ans\_label = tk.Label(self.master, text=(main\_window.hairdressers[self.lb][1] + ' ' + main\_window.hairdressers[self.lb][3] + main\_window.hairdressers[self.lb][2]), bg=bg\_colour, fg=text\_colour, font=(primary\_font, 12), anchor='w') # Defines a Tkinter label.
1305. self.hairdresser\_ans\_label.place(x=170, y=70, w=250, h=30) # Sets the geometry of the label.
1307. self.client\_label = tk.Label(self.master, text='Client:', bg=bg\_colour, fg=text\_colour, font=(primary\_font, 12), anchor='e') # Defines a Tkinter label.
1308. self.client\_label.place(x=0, y=120, w=165, h=30) # Sets the geometry of the label.
1310. self.haircut\_label = tk.Label(self.master, text='Haircut:', bg=bg\_colour, fg=text\_colour, font=(primary\_font, 12), anchor='e') # Defines a Tkinter label.
1311. self.haircut\_label.place(x=0, y=170, w=165, h=30) # Sets the geometry of the label.
1313. self.duration\_label = tk.Label(self.master, text='Duration:', bg=bg\_colour, fg=text\_colour, font=(primary\_font, 12), anchor='e') # Defines a Tkinter label.
1314. self.duration\_label.place(x=0, y=220, w=165, h=30) # Sets the geometry of the label.
1316. ### TKINTER COMBOBOXES (DROPDOWN MENUS)
1317. self.client\_box = ttk.Combobox(self.master, textvariable=self.client\_var, font=(secondary\_font, 9), state='readonly') # Defines a Tkinter combo box (drop-down menu).
1318. self.client\_box['values'] = self.client\_names # Sets the options of this drop-down menu to the values in the following array.
1319. self.client\_box.place(x=170, y=120, w=250, h=30) # Sets the drop-down menu's geometry.
1320. self.client\_box.bind('<Return>', **lambda** event: self.client\_box.event\_generate('<Down>')) # Make the drop-down menu drop down when the user presses the enter/return key while this widget is focused.
1321. self.client\_box.bind('<Escape>', **lambda** event: self.master.focus()) # Unfocuses this widget when the user presses the escape key.
1322. self.client\_box.bind('<<ComboboxSelected>>', **lambda** event: self.open\_add\_client\_window(False)) # Runs the following function when the user presses the enter/return key while this widget is focused.
1324. self.haircut\_box = ttk.Combobox(self.master, textvariable=self.haircut\_var, font=(secondary\_font, 9), state='readonly') # Defines a Tkinter combo box (drop-down menu).
1325. self.haircut\_box['values'] = self.haircut\_names # Sets the options of this drop-down menu to the values in the following array.
1326. self.haircut\_box.place(x=170, y=170, w=250, h=30) # Sets the drop-down menu's geometry.
1327. self.haircut\_box.bind('<Return>', **lambda** event: self.haircut\_box.event\_generate('<Down>')) # Make the drop-down menu drop down when the user presses the enter/return key while this widget is focused.
1328. self.haircut\_box.bind('<Escape>', **lambda** event: self.master.focus()) # Unfocuses this widget when the user presses the escape key.
1329. self.haircut\_box.bind('<<ComboboxSelected>>', **lambda** event: self.duration\_box.current(int((self.haircuts[self.haircut\_box.current()][3])/appointment\_intervals)-1))  # Runs the following function when the user presses the enter/return key while this widget is focused.
1331. self.duration\_box = ttk.Combobox(self.master, textvariable=self.duration\_var, font=(secondary\_font, 9), state='readonly') # Defines a Tkinter combo box (drop-down menu).
1332. self.duration\_box['values'] = self.durations # Sets the options of this drop-down menu to the values in the following array.
1333. self.duration\_box.place(x=170, y=220, w=250, h=30) # Sets the drop-down menu's geometry.
1334. self.duration\_box.bind('<Return>', **lambda** event: self.duration\_box.event\_generate('<Down>')) # Make the drop-down menu drop down when the user presses the enter/return key while this widget is focused.
1335. self.duration\_box.bind('<Escape>', **lambda** event: self.master.focus()) # Unfocuses this widget when the user presses the escape key.
1337. ### TKINTER BUTTONS
1338. self.add\_client\_button = tk.Button(self.master, text='+', command= **lambda**: self.open\_add\_client\_window(True), bg=bg\_colour, fg=text\_colour, font=primary\_font) # Defines a Tkinter button.
1339. self.add\_client\_button.place(x=420, y=120, w=30, h=30) # Sets the button's geometry.
1340. self.add\_client\_button.bind('<Return>', **lambda** event: self.open\_add\_client\_window(True)) # Runs the following function when the user presses the enter/return key while this widget is focused.
1341. self.add\_client\_button.bind('<Escape>', **lambda** event: self.master.focus()) # Unfocuses this widget when the user presses the escape key.
1343. self.add\_appointment\_button = tk.Button(self.master, text='Add appointment', command=self.add\_appointment, bg=bg\_colour, fg=text\_colour, font=primary\_font) # Defines a Tkinter button.
1344. self.add\_appointment\_button.place(x=380, y=310, w=140, h=30) # Sets the button's geometry.
1345. self.add\_appointment\_button.bind('<Return>', **lambda** event: self.add\_appointment()) # Runs the following function when the user presses the enter/return key while this widget is focused.
1346. self.add\_appointment\_button.bind('<Escape>', **lambda** event: self.master.focus()) # Unfocuses this widget when the user presses the escape key.
1348. self.cancel\_appointment\_button = tk.Button(self.master, text='Cancel', command=**lambda**: [self.master.destroy(), main\_window.lbs[self.day][self.lb+1].selection\_clear(0, tk.END)], bg=bg\_colour, fg=text\_colour, font=primary\_font) # Defines a Tkinter button.
1349. self.cancel\_appointment\_button.place(x=290, y=310, w=75, h=30) # Sets the button's geometry.
1350. self.cancel\_appointment\_button.bind('<Return>', **lambda** event: self.master.destroy()) # Runs the following function when the user presses the enter/return key while this widget is focused.
1351. self.cancel\_appointment\_button.bind('<Escape>', **lambda** event: self.master.focus()) # Unfocuses this widget when the user presses the escape key.
1353. ### MANAGING EXISTING APPOINTMENT
1354. **else**:
1356. ### MORE TKINTER WINDOW CONFIG
1357. self.master.title('Manage Appointment (Version {})'.format(version)) # Sets the window's title.
1359. ### FETCHING DATA FROM DATABASE
1360. c.execute('SELECT ClientID, HaircutID, StartTime, Duration, Date, AmountPaid FROM Appointments WHERE AppointmentID=?', (self.id,)) # Retrieves the appointment information for the appointment with matching id.
1361. self.appointment = c.fetchone() # Fetches the result from the query.
1363. c.execute('SELECT FirstName, LastName FROM Clients WHERE ClientID=?', (self.appointment[0],)) # Retrieves the first and last name for the client with matching id.
1364. self.client = c.fetchone() # Fetches the result from the query.
1366. c.execute('SELECT Haircut, Price FROM Haircuts WHERE HaircutID=?', (self.appointment[1],)) # Retrieves the haircut name and price for the haircut with matching id.
1367. self.haircut = c.fetchone() # Fetches the result from the query.
1369. db\_close()
1371. ### TKINTER LABELS
1372. self.time\_label = tk.Label(self.master, text='Time:', bg=bg\_colour , fg=text\_colour, font=primary\_font, anchor='e') # Defines a Tkinter label.
1373. self.time\_label.place(x=0, y=20, w=165, h=30) # Sets the label's geometry.
1374. self.time\_ans\_label = tk.Label(self.master, text=(self.appointment[2] + ' to ' + time.strftime('%M:%S', time.gmtime(int(self.appointment[2].split(':')[0]) \* 60 + int(self.appointment[2].split(':')[1]) + self.appointment[3]))), bg=bg\_colour, fg=text\_colour, font=primary\_font, anchor='w') # Defines a Tkinter label.
1375. self.time\_ans\_label.place(x=170, y=20, w=250, h=30) # Sets the label's geometry.
1377. self.date\_label = tk.Label(self.master, text='Date:', bg=bg\_colour, fg=text\_colour, font=primary\_font, anchor='e') # Defines a Tkinter label.
1378. self.date\_label.place(x=0, y=50, w=165, h=30) # Sets the label's geometry.
1379. self.date\_ans\_label = tk.Label(self.master, text=self.appointment[4], bg=bg\_colour, fg=text\_colour, font=primary\_font, anchor='w') # Defines a Tkinter label.
1380. self.date\_ans\_label.place(x=170, y=50, w=250, h=30) # Sets the label's geometry.
1382. self.hairdresser\_label = tk.Label(self.master, text='Hairdresser:', bg=bg\_colour, fg=text\_colour, font=primary\_font, anchor='e') # Defines a Tkinter label.
1383. self.hairdresser\_label.place(x=0, y=80, w=165, h=30) # Sets the label's geometry.
1384. self.hairdresser\_ans\_label = tk.Label(self.master, text=(main\_window.hairdressers[self.lb][1] + ' ' + main\_window.hairdressers[self.lb][3] + main\_window.hairdressers[self.lb][2]), bg=bg\_colour, fg=text\_colour, font=primary\_font, anchor='w') # Defines a Tkinter label.
1385. self.hairdresser\_ans\_label.place(x=170, y=80, w=250, h=30) # Sets the label's geometry.
1387. self.client\_label = tk.Label(self.master, text='Client:', bg=bg\_colour, fg=text\_colour, font=primary\_font, anchor='e') # Defines a Tkinter label.
1388. self.client\_label.place(x=0, y=110, w=165, h=30) # Sets the label's geometry.
1389. self.client\_ans\_label = tk.Label(self.master, text=(self.client[0] + ' ' +  self.client[1]), bg=bg\_colour, fg=text\_colour, font=primary\_font, anchor='w') # Defines a Tkinter label.
1390. self.client\_ans\_label.place(x=170, y=110, w=250, h=30) # Sets the label's geometry.
1392. self.haircut\_label = tk.Label(self.master, text='Haircut:', bg=bg\_colour, fg=text\_colour, font=primary\_font, anchor='e') # Defines a Tkinter label.
1393. self.haircut\_label.place(x=0, y=140, w=165, h=30) # Sets the label's geometry.
1394. self.haircut\_ans\_label = tk.Label(self.master, text=self.haircut[0], bg=bg\_colour, fg=text\_colour, font=primary\_font, anchor='w') # Defines a Tkinter label.
1395. self.haircut\_ans\_label.place(x=170, y=140, w=250, h=30) # Sets the label's geometry.
1397. self.amount\_to\_pay\_label = tk.Label(self.master, bg=bg\_colour, fg=text\_colour, font=primary\_font, anchor='e') # Defines a Tkinter label.
1398. self.amount\_to\_pay\_label.place(x=0, y=170, w=165, h=30) # Sets the label's geometry.
1399. self.amount\_to\_pay\_ans\_label = tk.Label(self.master, bg=bg\_colour, fg=text\_colour, font=primary\_font, anchor='w') # Defines a Tkinter label.
1400. self.amount\_to\_pay\_ans\_label.place(x=170, y=170, w=250, h=30) # Sets the label's geometry.
1402. ### TKINTER BUTTONS
1403. self.cancel\_appointment\_button = tk.Button(self.master, text='Cancel Appointment', command=self.cancel\_appointment, bg=bg\_colour, fg=text\_colour, font=primary\_font) # Defines a Tkinter button.
1404. self.cancel\_appointment\_button.place(x=360, y=310, w=160, h=30) # Sets the geometry of the button.
1405. self.cancel\_appointment\_button.bind('<Return>', **lambda** event: self.cancel\_appointment()) # Runs the following function when the user presses the enter/return key while this widget is focused.
1406. self.cancel\_appointment\_button.bind('<Escape>', **lambda** event: self.master.focus()) # Unfocuses this widget when the user presses the escape key.
1408. self.dismiss\_button = tk.Button(self.master, text='Dismiss', command=**lambda**: [self.master.destroy(), main\_window.lbs[self.day][self.lb+1].selection\_clear(0, tk.END)], bg=bg\_colour, fg=text\_colour, font=primary\_font) # Defines a Tkinter button.
1409. self.dismiss\_button.place(x=270, y=310, w=75, h=30)# Sets the geometry of the button.
1410. self.dismiss\_button.bind('<Return>', **lambda** event: self.master.destroy()) # Runs the following function when the user presses the enter/return key while this widget is focused.
1411. self.dismiss\_button.bind('<Escape>', **lambda** event: self.master.focus()) # Unfocuses this widget when the user presses the escape key.
1413. ### IF APPOINTMENT NOT PAID
1414. **if** self.appointment[5] == None: # If appointment is not paid.
1416. ### MORE TKINTER LABELS
1417. self.amount\_to\_pay\_label.configure(text='Amount to pay:') # Defines a Tkinter label.
1418. self.amount\_to\_pay\_ans\_label.configure(text='£') # Configure an existing label's displayed text.
1420. ### TKINTER ENTRY BOX
1421. self.amount\_to\_pay\_var = tk.StringVar() # Defines a Tkinter string variable.
1422. self.amount\_to\_pay\_entry = tk.Entry(self.master, textvariable=self.amount\_to\_pay\_var, fg=text\_colour, font=secondary\_font) # Defines a Tkinter entry box.
1423. self.amount\_to\_pay\_entry.insert(tk.END, '{:0.2f}'.format(self.haircut[1] \* self.rate \* 0.01)) # Insert the amount to pay as a string into the entry box (in X.YY format where X is a number and Y are digits).
1424. self.amount\_to\_pay\_entry.place(x=185, y=170, w=235, h=30) # Sets the entry boxes' geometry.
1425. self.amount\_to\_pay\_entry.bind('<Return>', **lambda** event: self.pay\_appointment()) # Runs the following function when the user presses the enter/return key while this widget is focused.
1426. self.amount\_to\_pay\_entry.bind('<Escape>', **lambda** event: self.master.focus()) # Unfocuses this widget when the user presses the escape key.
1428. ### MORE TKINER BUTTONS
1429. self.mark\_as\_paid\_button = tk.Button(self.master, text='Mark appointment as paid', command=**lambda**: self.pay\_appointment(), bg=bg\_colour, fg=text\_colour, font=primary\_font) # Defines a Tkinter button.
1430. self.mark\_as\_paid\_button.place(x=170, y=210, w=250, h=30) # Sets the button's geometry.
1431. self.mark\_as\_paid\_button.bind('<Return>', **lambda** event: self.pay\_appointment()) # Runs the following function when the user presses the enter/return key while this widget is focused.
1432. self.mark\_as\_paid\_button.bind('<Escape>', **lambda** event: self.master.focus()) # Unfocuses this widget when the user presses the escape key.
1434. ### IF APPOINTMENT PAID
1435. **else**:
1436. self.amount\_to\_pay\_label.config(text='Amount paid:') # Defines a Tkinter label.
1437. self.amount\_to\_pay\_ans\_label.configure(text='£{:0.2f}'.format(self.appointment[5])) # Configure an existing label's displayed text.
1439. ### 'APPOINTMENT' CLASS FUNCTIONS
1440. **def** add\_appointment(self):
1441. '''''This function adds an appointment to the database. It will first check if any fields are empty and if the appointment overlaps with any other appointments.'''
1442. **if** self.client\_var.get() == '' **or** self.haircut\_var.get() == '': # If either of the drop-down menus are empty.
1443. messagebox.showinfo('Error: Empty Field(s)', 'Please ensure all the fields are filled to make an appointment.') # Displays a Tkinter error window (popup).
1444. **else**:
1445. haircut = self.haircut\_box.current() # Defines a new variable equal to the selected option's index in the array.
1446. duration = self.duration\_box.current() # Defines a new variable equal to the selected option's index in the array.
1447. client = self.client\_box.current() # Defines a new variable equal to the selected option's index in the array.
1449. **for** i **in** range(duration): # Loop for index of selected duration.
1450. **if** main\_window.lbs[self.day][self.lb+1].appointment\_id[self.line+i+1] != -1: # If the appointment slot is already booked.
1451. messagebox.showerror('Error: Overlapping Appointments', 'This appointment cannot be made because it overlaps with another. Please reschedule or if possible shorten the duration of the appointment.') # Displays a Tkinter error window (popup).
1452. **return** # End the function early.
1454. db\_open() # Creates a connection and cursor with the database.
1455. c.execute('INSERT INTO Appointments(StaffID, ClientID, HaircutID, StartTime, Duration, Date) VALUES(?, ?, ?, ?, ?, ?)', (main\_window.hairdressers[self.lb][0], self.clients[client][0], self.haircuts[haircut][0], time.strftime('%M:%S', time.gmtime(first\_appointment\_time + self.line \* appointment\_intervals)), ((duration+1)\*appointment\_intervals), working\_dates[self.day].strftime('%d/%m/%Y'))) # Insert a new appointment into the Appointments table.
1456. conn.commit() # Commit the change to the database.
1457. db\_close() # Closes the connection and cursor with the database.
1459. self.master.destroy() # Destroy the appointments window.
1460. main\_window.update\_table() # Update the table in main window to show the new appointment.
1461. log('{} {} booked the following appointment: {} for {} booked with {} {}{} at {} on {}.'.format(user[1], user[2], self.haircuts[haircut][1], self.client\_var.get(), main\_window.hairdressers[self.lb][1], main\_window.hairdressers[self.lb][3], main\_window.hairdressers[self.lb][2], time.strftime('%M:%S', time.gmtime(first\_appointment\_time + self.line \* appointment\_intervals)), working\_dates[self.day].strftime('%d/%m/%Y'))) # Log the new appointment in the console.
1463. **def** pay\_appointment(self):
1464. '''''This function is called when the user clicks the pay appointment button. It will mark the appointment paid with the amount in the entry box.'''
1465. amount\_to\_pay = self.amount\_to\_pay\_var.get() # Define a new string equal to the value in the corresponding entry box.
1466. **try**: # Try the following code.
1467. float(amount\_to\_pay) # Convert the variable to a float.
1468. **except** ValueError: # If the variable could not be converted to a float.
1469. messagebox.showerror('Error: Must be a float value', 'The value you entered is not number or a float value.') # Displays a Tkinter error window (popup).
1470. **return** # End the function early.
1472. **if** float(amount\_to\_pay) < 0: # If negative.
1473. messagebox.showerror('Error: Negative Payment', 'The amount to pay cannot be negative.') # Displays a Tkinter error window (popup).
1474. **return** # End the function early.
1476. db\_open() # Creates a connection and cursor with the database.
1477. c.execute('UPDATE Appointments SET AmountPaid=? WHERE AppointmentID=?', (amount\_to\_pay, self.id)) # Edit the record to show it's paid.
1478. conn.commit() # Commit the change to the database.
1479. db\_close() # Closes the connection and cursor with the database.
1481. self.amount\_to\_pay\_label.configure(text='Amount paid:') # Change the text in the label to show it's paid.
1482. self.amount\_to\_pay\_ans\_label.configure(text='£{:0.2f}'.format(float(amount\_to\_pay))) # Show the amount paid by a label.
1484. self.amount\_to\_pay\_entry.destroy() # Destroy the entry box as it's no longer needed.
1485. self.mark\_as\_paid\_button.config(relief=tk.SUNKEN) # Set the button's state to sunken as it's no longer needed.

1488. **def** cancel\_appointment(self):
1489. '''''This function is called when the user clicks the cancel appointment button. It will remove the appointment from the database.'''
1490. db\_open()  # Creates a connection and cursor with the database.
1491. c.execute('SELECT AmountPaid FROM Appointments WHERE AppointmentID=?', (self.id,)) # Retieve the amount paid for the appointment.
1492. **if** c.fetchone()[0] != 0: # If the appointment has been paid.
1493. db\_close() # Closes the connection and cursor with the database.
1494. **if** **not** messagebox.askyesno('Confimation: Appointment has been paid', 'This appointment has been paid for. Please refund the client before cancelling. Do you want to continue?'): # Displays a Tkinter confirmation window (popup).
1495. **return** # Ends the function early.
1496. **else**:
1497. db\_close() # Closes the connection and cursor with the database.
1498. **if** messagebox.askyesno('Confirmation: Cancel appointment', 'Are you sure you want to cancel this appointment? This change cannot be undone. Do you want to continue?'): # Displays a Tkinter confirmation window (popup).
1499. db\_open()  # Creates a connection and cursor with the database.
1500. c.execute('DELETE FROM Appointments WHERE AppointmentID=?', (self.id,)) # Deletes the appointment from the database.
1501. conn.commit() # Commits the change to the database.
1502. db\_close() # Closes the connection and cursor with the database.
1503. self.master.destroy() # Destroy the appointments window.
1504. main\_window.update\_table() # Update the table in the main window.
1505. log('{} {} cancelled the following appointment: {} for {} {} booked with {} {}{} at {} on {}.'.format(user[1], user[2], self.haircut[0], self.client[0], self.client[1], main\_window.hairdressers[self.lb][1], main\_window.hairdressers[self.lb][3], main\_window.hairdressers[self.lb][2], time.strftime('%M:%S', time.gmtime(first\_appointment\_time + self.line \* appointment\_intervals)), working\_dates[self.day].strftime('%d/%m/%Y'))) # Log who deleted the appointment in the console.
1507. **def** open\_add\_client\_window(self, called\_by\_button):
1508. '''''This function is called when the user clicks the button or selects the last option in the client drop-down menu. It opens the AddClient window.'''
1509. **if** self.client\_var.get() == '--- Add new client ---' **or** called\_by\_button == True: # If last option of the client drop-down menu is selected or the function was called by the button.
1510. self.client\_box.set('') # Unselect any options selected in the client's drop-down menu.
1512. popup = tk.Toplevel(self.master) # Defines a Tkinter toplevel window (popup window).
1513. **global** add\_client\_window # Declare the variable 'add\_client\_window' globally.
1514. add\_client\_window = AddClient(popup) # Create an instance of the AddClient class.
1515. popup.mainloop() # Wait until the add client window is closed until proceeding.
1517. #======================================================================================
1518. #                                   ADDING CLIENTS WINDOW
1519. #======================================================================================
1520. **class** AddClient():
1521. '''''This window consists of entry boxes allowing the user to add a client to the database. This is a quicker way to add clients without going into the client table. It also does not require admin rights to perform. This window can be opened in the appointments window by a button or by a drop-down menu.'''
1522. **def** \_\_init\_\_(self, master):
1523. '''''This function is run when the class is initialised.'''
1524. ### VARIABLES
1525. self.master = master # Defines the variable 'master' as an attribute to this class.
1526. self.first\_name\_var = tk.StringVar() # Defines a Tkinter string variable.
1527. self.last\_name\_var = tk.StringVar() # Defines a Tkinter string variable.
1528. self.mobile\_phone\_var = tk.StringVar() # Defines a Tkinter string variable.
1529. self.home\_phone\_var = tk.StringVar() # Defines a Tkinter string variable.
1531. ### TKINTER WINDOW CONFIG
1532. self.master.title('Add Client (Version {})'.format(version)) # Sets window's title.
1533. self.master.geometry('270x180+{}+{}'.format(((screen\_width-270)//2), ((screen\_height-180)//2)))  # Sets window geometry and centers the window.
1534. self.master.resizable(False, False) # Sets the window so it cannot be resized.
1535. self.master.wm\_iconbitmap('resources/icon.ico') # Sets window's icon.
1536. self.master.config(bg=bg\_colour) # Sets background colour of window.
1537. self.master.protocol('WM\_DELETE\_WINDOW', **lambda**: [self.master.destroy(), appointments\_window.master.focus\_set(), appointments\_window.master.grab\_set()]) # Runs the following functions when this window is closed.
1539. appointments\_window.master.grab\_release() # Release the grab of appointments window (to be transferred to this window).
1540. self.master.focus\_set() # Sets focus on this window.
1541. self.master.grab\_set() # Sets grab on this window, so other windows cannot take the focus.
1543. # TKINTER LABELS
1544. self.first\_name\_label = tk.Label(self.master, text='First Name:', bg=bg\_colour, fg=text\_colour, font=(primary\_font, 10), anchor='e') # Defines a Tkinter label.
1545. self.first\_name\_label.place(x=0, y=10, w=90, h=25) # Sets the label's geometry.
1547. self.last\_name\_label = tk.Label(self.master, text='Last Name:', bg=bg\_colour, fg=text\_colour, font=(primary\_font, 10), anchor='e') # Defines a Tkinter label.
1548. self.last\_name\_label.place(x=0, y=40, w=90, h=25) # Sets the label's geometry.
1550. self.mobile\_phone\_label = tk.Label(self.master, text='Mobile Phone:', bg=bg\_colour, fg=text\_colour, font=(primary\_font, 10), anchor='e') # Defines a Tkinter label.
1551. self.mobile\_phone\_label.place(x=0, y=70, w=90, h=25) # Sets the label's geometry.
1553. self.home\_phone\_label = tk.Label(self.master, text='Home Phone:', bg=bg\_colour, fg=text\_colour, font=(primary\_font, 10), anchor='e') # Defines a Tkinter label.
1554. self.home\_phone\_label.place(x=0, y=100, w=90, h=25) # Sets the label's geometry.
1556. # TKINTER ENTRY BOXES
1557. self.first\_name\_entry = tk.Entry(self.master, textvariable=self.first\_name\_var, fg=text\_colour, font=(secondary\_font, 9)) # Defines a Tkinter entry box.
1558. self.first\_name\_entry.place(x=95, y=10, w=150, h=25) # Sets the entry boxes' geometry.
1559. self.first\_name\_entry.bind('<Return>', **lambda** event: self.last\_name\_entry.focus()) # When pressing the return key while this widget is focused, will shift focus to the next widget.
1560. self.first\_name\_entry.bind('<Escape>', **lambda** event: self.master.focus()) # Unfocuses this widget when the user presses the escape key.
1561. self.first\_name\_entry.focus()
1563. self.last\_name\_entry = tk.Entry(self.master, textvariable=self.last\_name\_var, fg=text\_colour, font=(secondary\_font, 9)) # Defines a Tkinter entry box.
1564. self.last\_name\_entry.place(x=95, y=40, w=150, h=25) # Sets the entry boxes' geometry.
1565. self.last\_name\_entry.bind('<Return>', **lambda** event: self.mobile\_phone\_entry.focus()) # When pressing the return key while this widget is focused, will shift focus to the next widget.
1566. self.last\_name\_entry.bind('<Escape>', **lambda** event: self.master.focus()) # Unfocuses this widget when the user presses the escape key.
1568. self.mobile\_phone\_entry = tk.Entry(self.master, textvariable=self.mobile\_phone\_var, fg=text\_colour, font=(secondary\_font, 9)) # Defines a Tkinter entry box.
1569. self.mobile\_phone\_entry.place(x=95, y=70, w=150, h=25) # Sets the entry boxes' geometry.
1570. self.mobile\_phone\_entry.bind('<Return>', **lambda** event: self.home\_phone\_entry.focus()) # When pressing the return key while this widget is focused, will shift focus to the next widget.
1571. self.mobile\_phone\_entry.bind('<Escape>', **lambda** event: self.master.focus()) # Unfocuses this widget when the user presses the escape key.
1573. self.home\_phone\_entry = tk.Entry(self.master, textvariable=self.home\_phone\_var, fg=text\_colour, font=(secondary\_font, 9)) # Defines a Tkinter entry box.
1574. self.home\_phone\_entry.place(x=95, y=100, w=150, h=25) # Sets the entry boxes' geometry.
1575. self.home\_phone\_entry.bind('<Return>', **lambda** event: self.add\_client()) # When pressing the return key while this widget is focused, run the following function.
1576. self.home\_phone\_entry.bind('<Escape>', **lambda** event: self.master.focus()) # Unfocuses this widget when the user presses the escape key.
1578. # TKINTER BUTTONS
1579. self.add\_client\_button = tk.Button(self.master, text='Add Client', command=**lambda**: self.add\_client(), bg=bg\_colour, fg=text\_colour, font=primary\_font) # Defines a Tkinter button.
1580. self.add\_client\_button.place(x=110, y=140, w=140, h=30) # Sets the button's geometry.
1581. self.add\_client\_button.bind('<Return>', **lambda** event: self.add\_client()) # When pressing the return key while this widget is focused, run the following function.
1582. self.add\_client\_button.bind('<Escape>', **lambda** event: self.master.focus()) # Unfocuses this widget when the user presses the escape key.
1584. self.cancel\_button = tk.Button(self.master, text='Cancel', command=**lambda**: [self.master.destroy(), appointments\_window.master.focus\_set(), appointments\_window.master.grab\_set()], bg=bg\_colour, fg=text\_colour, font=primary\_font) # Defines a Tkinter button.
1585. self.cancel\_button.place(x=30, y=140, w=65, h=30) # Sets the button's geometry.
1586. self.cancel\_button.bind('<Return>', **lambda** event: [self.master.destroy(), appointments\_window.master.focus\_set(), appointments\_window.master.grab\_set()]) # When pressing the return key while this widget is focused, run the following functions
1587. self.cancel\_button.bind('<Escape>', **lambda** event: self.master.focus()) # Unfocuses this widget when the user presses the escape key.

1590. ### 'ADDCLIENT' CLASS FUNCTIONS
1591. **def** add\_client(self):
1592. '''''This function validates the values entered into the entry boxes. If valid, will add the client to the database. This function is called by the add client button.'''
1593. ### VARIABLES
1594. first\_name = self.first\_name\_var.get().replace(' ', '') # Defines a new string with the value in the corresponding entry box. It also removes any space characters in the process.
1595. last\_name = self.last\_name\_var.get().replace(' ', '') # Defines a new string with the value in the corresponding entry box. It also removes any space characters in the process.
1596. mobile\_phone = self.mobile\_phone\_var.get().replace(' ', '') # Defines a new string with the value in the corresponding entry box. It also removes any space characters in the process.
1597. home\_phone = self.home\_phone\_var.get().replace(' ', '') # Defines a new string with the value in the corresponding entry box. It also removes any space characters in the process.
1599. ### EMPTY FIELD(S)
1600. **if** first\_name == '' **or** last\_name == '': # If any of the name fields are empty.
1601. messagebox.showerror('Error: Empty Field(s)', 'You cannot have any empty fields') # Displays a Tkinter error window (popup).
1602. **return** # Ends the function early.
1604. ### CHECK NAMES ARE ALPHABETICALLY
1605. **if** **not** ((first\_name.isalpha() **or** first\_name == '') **and** (last\_name.isalpha()) **or** last\_name ==''): # If the names only consist of alphabetically characters (or are blank).
1606. messagebox.showerror('Error: Incompatible characters', 'Names must contain only alphabetically characters. Please try again.') # Displays a Tkinter error window (popup).
1607. **return** # Ends the function early.
1609. ### CHECKS PHONE NUMBER CONTAINS ONLY DIGITS AND CORRECT AMOUNT OF DIGITS
1610. **if** **not** ((mobile\_phone.isdigit() **or** mobile\_phone == '') **and** (home\_phone.isdigit()) **or** home\_phone ==''): # If the phone numbers consist of only digits (or are blank).
1611. messagebox.showerror('Error: Incompatible characters', 'Phone number must contain only digits. Please try again.') # Displays a Tkinter error window (popup).
1612. **return** # Ends the function early.
1614. **if** **not**((len(mobile\_phone) **in** {10, 11} **or** mobile\_phone == '') **and** (len(home\_phone) **in** {10, 11} **or** home\_phone == '')): # If the phone numbers consist of 10 or 11 characters (or are blank).
1615. **if** **not** messagebox.askyesno('Confirmation: Phone Number(s) length(s)', 'Phone number should contain 10 or 11 digits. Do you want to continue?'): # Displays a Tkinter confirmation window (popup).
1616. **return** # Ends the function early.
1617. ### MORE EMPTY FIELD(S)
1618. **if** mobile\_phone == '' **and** home\_phone == '': # If any of the phone number fields are empty.
1619. **if** **not** messagebox.askyesno('Confirmation: No Phone Number', 'We recommended atleast one phone number to contact and inform clients of cancellations. Are you sure you want to continue?'): # Displays a Tkinter confirmation window (popup).
1620. **return** # Ends the function early.
1622. ### CHECK CLIENT DOES NOT ALREADY EXIST ON DATABASE
1623. db\_open()
1624. c.execute('SELECT Mobile, Home FROM Clients WHERE (FirstName=? AND LastName=? AND (Mobile=? OR Home=?))', (first\_name, last\_name, mobile\_phone, home\_phone)) # Retrieve any records of clients that have matching names and at least one matching phone number.
1625. **if** c.fetchone() != None: # If any matching records.
1626. **if** **not** messagebox.askyesno('Confirmation: Duplicates may exist!', 'There is already a client with this name who shares the same phone number(s). Please ensure this new client is not already on the database. Do you want to continue?'): # Displays a Tkinter confirmation window (popup).
1627. **return** # Ends the function early.
1628. db\_close()
1630. ### ADD CLIENT TO DATABASE
1631. db\_open()
1632. **if** mobile\_phone == '' **and** home\_phone == '': # If no phone numbers were given.
1633. c.execute('INSERT INTO Clients(FirstName, LastName) VALUES(?, ?)', (first\_name, last\_name)) # Add a new client record with only the names provided.
1634. **elif** mobile\_phone == '': # If only the home phone number was given.
1635. c.execute('INSERT INTO Clients(FirstName, LastName, Home) VALUES(?, ?, ?)', (first\_name, last\_name, home\_phone)) # Add a new client record with the names and the home phone number provided.
1636. **elif** home\_phone == '': # If only the mobile phone number was given.
1637. c.execute('INSERT INTO Clients(FirstName, LastName, Mobile) VALUES(?, ?, ?)', (first\_name, last\_name, mobile\_phone)) # Add a new client record with the names and the mobile phone number provided.
1638. **else**: # # If both phone numbers were given.
1639. c.execute('INSERT INTO Clients(FirstName, LastName, Mobile, Home) VALUES(?, ?, ?, ?)', (first\_name, last\_name, mobile\_phone, home\_phone)) # Add a new client record with the names and phone numbers provided.
1640. conn.commit() # Commit the change to the database.
1642. c.execute('SELECT ClientID, FirstName, LastName FROM Clients') # Retrive the new client records.
1643. appointments\_window.clients = c.fetchall() # Update the client records of the 2D array stored as an attribute of the appointments window.
1644. db\_close()
1646. appointments\_window.client\_names = [] # Redefines the variable as an empty array.
1647. **for** i **in** range(len(appointments\_window.clients)): # For all clients.
1648. appointments\_window.client\_names.append('{} {}'.format(appointments\_window.clients[i][1], appointments\_window.clients[i][2])) # Put the client's first name and surname together to make it an option for a drop-down menu.
1649. appointments\_window.client\_names.append('--- Add new client ---') # Add a last option to add a new client.
1650. appointments\_window.client\_box['values'] = appointments\_window.client\_names # Resets the options of this drop-down menu to the values in the following array.
1651. appointments\_window.client\_box.current(len(appointments\_window.client\_names)-2) # Set the currently selected client as the one just added.
1653. self.master.destroy() # Destroys the add client window.
1654. appointments\_window.master.focus\_set() # Sets focus back on the appointments window.
1655. appointments\_window.master.grab\_set() # Sets the grab back on the appointments window, so other windows cannot take focus.

1658. **if** \_\_name\_\_ == '\_\_main\_\_': # Only runs the following code if the program is the main program and is not imported into another.
1659. main() # Calls the main function.

# Prototypes

I created two prototypes during the program stage which allowed me to test some of the suggestions I had got from the investigation. After the prototypes were made, I was able to meet with Jim to discuss if we could further improve the design and functionality. It was important to prototype to ensure my program was at the standard the client expected. I feel like the layout of the program was the most important thing as it is not clear of what the client of the company expects the program to look like. However, the functionality the client wanted was quite clear and is less subject to change.

## Prototype 1

The first prototype had little functionality but had the basic indented layout. It features a table with a scroll bar and example values to show where they would be displayed. The functional settings button and a search bar were in place too. The client was then able to see what the intended GUI would look like.

The table displayed the appointment records where each column represents a field in the record. It featured information for all the appointments for that day. The size of the screen does not limit the amount of data being shown. The scroll bar allows the user to scroll vertically through the table so all the appointments can be displayed.

## Prototype 2

This prototype featured a more advanced GUI design. It features a Tkinter notebook widget with tabs that can change the day of appointments that are being view. This prototype also includes the login window that gives the program the basic security required. The basics of the settings window is also working. It allows the user to change check boxes and other widgets. If changed, the window is recognising it has unsaved changes and will notify the user if they try to close the settings window down with a confirmation window. The user can discard the changes or cancel closing the window by discarding and click apply and exit instead to apply the changes. This feature will be carried forward to the tables window once implemented so if admins make changes which they are not sure about none of the changes will take effect until they click that apply or apply and exit button.

There is also validation in place and communication to the database. The login window validates the Staff ID by checking it’s a digit. It will also check if either of the entry boxes are blank and cancel the login. If both values are valid it will retrieve the password for that Staff ID from the database and compare it with the entered password. If they match, then the user will be allowed access to the main window, else they will be denied and asked to log in again.

## Prototype Refinement

### Prototype 1

After presenting my first prototype with my client a few improvements became apparent. These included the:

#### Improving the design of the program

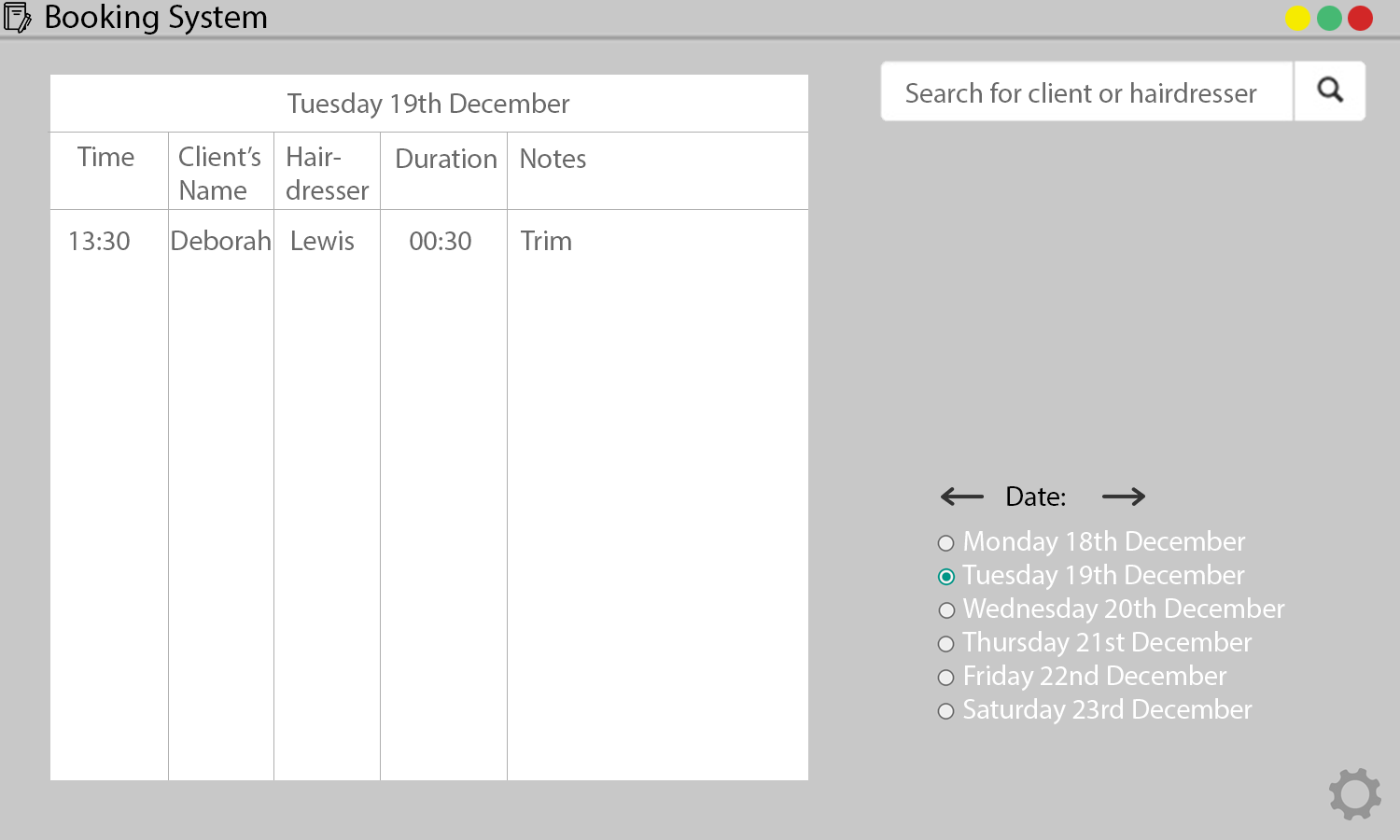
The layout of the program was not very clean and the table was not big enough. The colour was bland and the overall program required another colour. This could easily be partially achieved by changing the variables that hold the colour in hexadecimal.

#### Introducing navigation

The program needed more functionality in terms of navigation. The program needed a method of navigating to future days so clients of the hairdressers could book appointments in advance and not only on the same day. The program should allow the user to move through days or even weeks quickly to be efficient. This can be achieved by having buttons that go back and forward a week. We can also introduce tabs, radio buttons or a drop-down menu to select a day of the week to view.

#### Improving the GUI

From these improvements I was able to create a new and improved GUI design with Photoshop that I was then able to show my client. This meant I could show my intentional changes without going through the process of potentially wasting time making the changes in Python. I annotated the GUI explaining fully explaining what each element of the GUI and its purpose. This GUI can be found on the next page.



**Search bar** for searching. Once clicked, search settings will appear below allowing for a bigger time range, enabling hairdressers, days of the week, etc. Once something is searched the table on the left will be replaced with the results. 

**Buttons** to control the geometry of the window. This is a standard across all applications. The program will support resizing the main window and will result in the widgets resizing accordingly.

An additional window will appear which will allow **bookings** of appointments.

**Icon** and **Program Name**

Table that **displays information** about the current day’s **appointments**. The radio buttons on the bottom right allow users to quickly scan through the new few days appointments.

Users can quickly **select date** that want to appear on screen. The arrows next to the heading allow weeks to be cycled.

Book an Appointment

**Settings**. This will allow Admins to add / remove hairdressers and change prices and other details at wish. Details about clients can also be changed. This will require an administrator password or PIN.

### Prototype 2

My client was overall happy with the changes. There were a few suggestions and changes that were discussed during our meeting.

#### Displaying appointments

The original plan was to have an entire row for each record displaying an appointment. However, it was obvious that this is not space efficient and tends to lead to a lot of scrolling when searching through the system. Therefore, we concluded that each cell would instead display an appointment. Each row would represent a certain time where the rows had a time difference of 15 minutes. And each column would represent a hairdresser. I suggested that clicking on the cell would display more information about that appointment in that cell which was agreed on. This would therefore lead to the table having much less rows and the table being much less busy. Overloading the user with information is not always good as it would mean the user would be having to scan through more information to find the appointment there are looking for.

#### Introducing a price calculator

I was struggling on how to set prices for haircuts. He informed me that each hairdresser had a certain rate and each haircut had a set price. Therefore, we can multiple the set prices and the rate of the hairdresser to generate a price. We would have the rate 1.00 as the standard hairdresser rate and make all the haircuts’ prices relative to that hairdresser.

If we had a haircut with the lowest experienced hairdresser at £12, then we have a hairdresser that is more experienced and therefore charges 75% more. We can set their rate at 1.75.

£12 × 1.75 = £21. Therefore, a haircut with the more experienced hairdresser would be £21.

This means higher quality hairdressers such as Jim can be priced more appropriately compared to new less experienced hairdressers.

#### Improving the GUI

With the new changes to displaying appointments the table have to go through major changes. The first column would now contain times of the appointments so the way the program populated the table had to be adjusted. The changes were again made into a second new and improved GUI which again was annotated. This GUI can be found on the next page.



An **appointment**. This appointment with the hairdresser James is expected to take 30 minutes so takes two **15 minute slots**. The client’s name is displayed in the slot.

This appointment is empty so when the user clicks it the user will be given a window to add an appointment to this slot. The appointment would be with Jim starting at 10:45.

Each appointment in this row will have the corresponding time in the beginning of the row.

Each column represents a hairdresser’s appointment for this day.

The table will have different **tabs** where each tab changes the table for that working day of the week.

**Scroll bar** allows the user to **navigate** through the times for this day. It will scroll all the appointments down while making sure the hairdresser’s names are fixed in place.

User can use the arrows to **navigate** to different weeks quickly at the click of a button.

**Buttons** to control the geometry of the window. This is a standard across all applications. The program will support resizing the main window and will result in the widgets resizing accordingly.

**Icon** and **Program Name**

**Settings**. This will allow Admins to add / remove hairdressers and change prices and other details at wish. Details about clients can also be changed. This will require an administrator password or PIN.

## Prototype Testing

|  |  |  |  |
| --- | --- | --- | --- |
| Test Description | Expected Outcome | Actual Outcome | Passed?  Linked to figure |
| Test to see what happens when the Staff ID or password field is left blank when logging in. | The program detects the blank field and tells the user it’s invalid. | Successful. | Figure 1a, Figure 1b, Figure 1c |
| Test to see what happens when the Staff ID doesn’t exist in the database. | The program cannot find the Staff ID in the database and tells the user one of the fields is incorrect. | Successful. | Figure 2 |
| Test to see what happens when the password entered is not the correct password for that corresponding Staff ID. | The program will detect the entered password does not match the password in the database and tells the user one of the fields is incorrect. | Successful. | Figure 3 |
| Test to see what happens when the Staff ID is a invalid datatype. | The program detects the Staff ID is not a digit and tell the user one of the fields is incorrect. | Successful. | Figure 4a, Figure 4b |
| Test to see if the interactive labels work correctly (if the user can click | The program should change the appearance of the label to indicate it’s interactable (clickable). Once clicked, the label should show some sort of response. | Successful. | Figure 5a, Figure 5b |
| The user should be able to open the settings window. | The program should open the settings window when the user clicks the settings window. | Successful. | Figure 18a, Figure 18b |

# Maintenance Documentation

## Table of Classes

Each class represents a type of Tkinter window within the program. Classes can be initialised in the format <class name>(<window>) where <class name> is the name of the class and <window> is the name of the Tkinter window variable being used.

|  |  |
| --- | --- |
| Class Name | Description |
| Login | This window will be the window seen at launch. This is the basic security for the program and will require a staff ID and password in order to launch the main window. |
| Main | This is the main window of the program. It contains the appointment table, settings button, logout button and buttons to navigate through the table. |
| Settings | This window is a basic window that includes all the settings for the program. It contains a drop-down menu where the user can select a table to edit. This requires admin access. It also allows the user to edit their password. This window is created when the user clicks the settings button in the main window. |
| ChangePassword | This window is created when the user clicks the change password button in the settings window. This window simply contains three entries which askes for the current password, the new password and to re-enter the new password. |
| Tables | This window is created when the user selects a table from the dropdown menu from the settings window. This window contains the selected table and all corresponding records from the database. From here the user can add, remove and edit records. |
| Appointments | This window is created when the user clicks on an appointment slot. This class has two states depending on the appointment slot. If the slot is empty the window will consist of drop-down menus so the user can book an appointment. If the appointment slot is occupied, then the window will consist of labels instead so the user can review the appointment information. The user will also be able to mark the appointment as paid in case the client wants to pay ahead of time. |
| AddClient | This window consists of entry boxes allowing the user to add a client to the database. This is a quicker way to add clients without going into the client table. It also does not require admin rights to perform. This window can be opened in the appointments window by a button or by a drop-down menu. |

## Table of Routines/Functions

|  |  |  |  |
| --- | --- | --- | --- |
| Routine Name | Description | Class the routine belongs to | Variable(s) used within the routine |
| main | The function called when the program is first run. It defines a Tkinter window and creates an instance of the Login class. | *None* | login\_window, root |
| add\_date\_suffix | This function adds ‘st’, ‘nd’, ‘rd’ or ‘th’ to the end of a number where the number represents a day of the month and is between 1 – 31. | *None* | day, suffix |
| log | This function simply prints a string with the date and time as a prefix. | *None* | statement |
| db\_open | This function opens the connection and cursor to the database. | *None* | c, conn |
| db\_close | This function closes the connection and cursor to the database. | *None* | c, conn |
| \_\_init\_\_ | This function is called whenever a new instance of any class is created. This function contains all the code the class needs to run when initialised. | \* (*All classes*) | *\** (*Depends on class, but usually almost every variable*) |
| log\_in | The function ran when the user has inputted a Staff ID and password and then pressed enter or clicked the log in button. The function compares the inputted values to the ones in the database and then performs accordingly. | Login | id, id\_var, p, p\_var, c, user, main\_window, root, id\_entry, p\_entry |
| on\_resize | The function called when the geometry of the main window is altered. This function adjusts the geometry any widgets that depend on the height and width of the main window. A lot of different widgets’ geometry are altered here therefore lots of variables are involved. | Main | event, width, height, master, \*\_label, \*\_button, names\_frame, nb, i, hairdressers, names |
| on\_hover | This function is called when an interactable **label** is hovered over. Depending on the parameter given the function will change the colour of the label to show it’s interactable. These labels perform like buttons. | Main | hover, \*\_button, \*\_image |
| log\_out | This function is called when the user clicks the log out button. The function prepares the window to be converted back into a log in window and then converts it. | Main | widget, master |
| move\_day | This function is called when the tab is changed or needs to be changed. This function has one argument which is a boolean that determines if the function will move back a day (to left tab) or forward a day (to right tab). | Main | direction, nb, working\_days, master |
| move\_week | This function allows the user to navigate through the weeks in the table. This function has one argument which is a boolean that determines if the function will move back or forward a week. | Main | direction, i, working\_dates |
| tick | This recursive function compares real time and date to the displayed time and date and updates them where necessary. This function calls itself after every 0.2 seconds. | Main | time, time\_label, date, date\_label |
| on\_scroll\_by\_bar | When scrolling with the scrollbar this function is called. All the list boxes on that page will change their y view to match the scrollbar. This allows all list boxes to be scrolled through with a single scrollbar. | Main | i, hairdressers, lbs, nb |
| on\_scroll\_by\_ wheel | When scrolling by hovering over a list box it only scrolls that single list box. Therefore, we have to call this function that will move all the list boxes to this y view. The scrollbar also has to be readjusted to match the list boxes. | Main | i, hairdressers, lbs, scrolls, nb |
| update\_table | This function is responsible for importing all appointments from the database into the tables. This function is called when the class is initialised, when the user changes the week and adds or removes an appointment or hairdressers. | Main | i, j, k, l, hairdressers, lbs, nb, appointment\_id, working\_dates, padding, tabs, week\_label, first\_appointment\_ time, last\_appointment\_ time, appointment\_intervals, start\_time, c, client |
| open\_settings\_ window | This function opens a popup (Tkinter top level) window so the user can config the settings. It is triggered when the user clicks the settings button. | Main | popup, master, settings\_window |
| open\_ appointment\_ window | This function is triggered when the user clicks on a list box. It works out the column and row that was clicked then creates an instance of Appointments with that information. | Main | event, popup, master, appointment\_window |
| open\_change\_ password\_ window | This function opens the Tkinter window that allows the user to change their password. This function is called when the user clicks the change password button. | Settings | popup, master, change\_password\_ window |
| open\_tables\_ window | This function opens the Tkinter window that allows the user to edit the database. This function is called when the user selects an option from the drop-down menu. | Settings | user, popup, master, tables\_window, tables\_box |
| update\_main\_ window | This function is called when the settings window is closed and the main window is refocused/re-grabbed. If any hairdressers have been added or removed from the database, the window will need to be rebuild to account for the change of list boxes, canvases, etc. | Settings | update, master, tab, tabs |
| change\_ password | This function checks the if the inputted password matches the password in the database. If it does it will overwrite it with the new password. This function is called when the user presses the change password button in this window. | Change-Password | current\_p, current\_p\_var, new\_p, new\_p\_var, new\_p\_confirm, new\_p\_confirm\_var, i, j, c, conn, user, master |
| on\_hover | This function runs when the cursor moves over the entry box. It is required so when the user left clicks we know if they've left clicked the entry box. | Tables | entry, hover |
| focus\_fix | This function fixes which widget is focused depending on if the entry box is present. This function is a fix to a bug. | Tables | entry, hidden, master |
| force\_show\_ entry | This function forces the entry box to appear when hide\_entry is next called (which is called straight after by the left click binding). This is required so the entry box is not hidden straight away. This function is a fix to a bug. | Tables | event, entry, hidden, forced |
| hide\_entry | This function hides the entry box and checks if the new value is valid data for this field. This function is called when the user clicks off the entry box or presses escape or enter. | Tables | cancel, click, entry, forced, table, lb, field, records, hover, hidden, table\_frame, new\_val, entry\_var, lbs, record, record\_required, alpha\_exists, digit\_exists, i, j, admins |
| show\_entry | This function is called when the user wants to edit a record. It summons an entry box over the record so the user can edit the value. | Tables | entry, field, record, remove\_button, remove\_info\_label, i, records, table, records, id\_labels, changes, apply\_button, lb, lbs, hidden, yview, entry\_var, y |
| add\_record | This function is called when the user clicks the add record button. It adds a record at the bottom of the table and adds it to the record array with that table's default values for each field. | Tables | records, new\_record, id\_labels, master, bg\_colour, text\_colour, id\_labels, apply\_button, i, fields, table, lbs, primary\_colour |
| remove\_record | This function is run when the remove button is clicked by the user. This will cause the next record to be pressed to be deleted. This button is toggleable. | Tables | remove\_button, remove\_info\_label |
| on\_scroll\_by\_bar | This function is run when the user scrolls with the scroll bar. It syncs all the list boxes to the new y view of the scroll bar. | Tables | entry, hidden, i, fields, lbs, records, id\_labels |
| on\_scroll\_by\_ wheel | This function is triggered when the user scrolls using the scroll wheel on a list box. This sets all the other list boxes y view to the y view of the scrolled list box. | Tables | lbs, fields, entry, hidden, yview, i, scroll, id\_labels, line |
| apply | This function is called when the user clicks the apply button. It applies all the changes made in the table to the database. | Tables | apply\_button, i, j, fields, records, lbs, table, table\_name, changes, c, conn, update, user |
| exit\_ | This function is called when the user clicks the exit button. It applies changes and then closes the tables window. | Tables | master, settings\_window |
| cancel | This function is called when the user clicks the cancel button. It discards all changes and then closes the tables window. | Tables | apply\_button, master, settings\_window |
| add\_ appointment | This function adds an appointment to the database. It will first check if any fields are empty and if the appointment overlaps with any other appointments. | Appoint-ments | client\_var, haircut\_var, haircut, duration, client, i, lbs, day, appointment\_id, c, conn, master |
| pay\_ appointment | This function is called when the user clicks the pay appointment button. It will mark the appointment paid with the amount in the entry box. | Appoint-ments | amount\_to\_pay, amount\_to\_pay\_var, c, conn, id, amount\_to\_pay\_label, amount\_to\_pay\_ans\_ label, amount\_to\_pay\_entry, mark\_as\_paid\_button |
| cancel\_ appointment | This function is called when the user clicks the cancel appointment button. It will remove the appointment from the database. | Appoint-ments | c, conn, master, main\_window, user, haircut, client, hairdressers, lb, first\_appointment\_ time, line, appointment\_intervals, working\_dates, day |
| open\_add\_ client\_window | This function is called when the user clicks the button or selects the last option in the client drop-down menu. It opens the AddClient window. | Appoint-ments | called\_by\_button, client\_var, client\_box, popup, add\_client\_window |
| add\_client | This function validates the values entered into the entry boxes. If valid, will add the client to the database. This function is called by the add client button. | Appoint-ments | first\_name, first\_name\_var, last\_name, last\_name\_var, mobile\_phone, mobile\_phone\_var, home\_phone, home\_pone\_var, c, conn, appointments\_ window, clients, client\_names, client\_box, master |

## Table of Variables

|  |  |  |  |
| --- | --- | --- | --- |
| Variable Name | Description | Type | `Scope |
| i, j, k, l | This variable is used as a counter. | Integer | Local |
| version | This variable holds the version number of the program. | String | Global |
| first\_appointment\_time | This variable holds the time of the first appointment in ‘hh:mm’ format. | String | Global |
| last\_appointment\_time | This variable holds the time of the last appointment in ‘hh:mm’ format. | String | Global |
| appointment\_intervals | The variable holds the time in minutes of each appointment slot. | Integer | Globa |
| working\_days | This array holds integers, where each integer corresponds to a day of the week where 1 is Monday and 7 is Sunday. | Array of Integers | Global |
| working\_dates | This array holds the dates as an instance of the datetime class. It is the working\_days converted to dates. | Array of datetimes | Global |
| bg\_colour | The variable holds the colour of the background in hexidecimal. | String |  |
| text\_colour | The variable holds the colour of the text in hexidecimal. | String |  |
| primary\_colour | The variable holds the primary theming colour in hexidecimal. | String |  |
| secondary\_colour | The variable holds the secondary theming colour in hexidecimal. | String |  |
| primary\_font | The variable holds the name of the primary font used. | String |  |
| secondary\_font | The variable holds the name of the secondary font used. | String |  |
| tertiary\_font | The variable holds the name of the tertiary font used. | String |  |
| database\_file | The variable holds the name of the database file located in the /db/ folder in the format ‘<name>.db’. | String |  |
| main\_window\_zoomed | The variable holds if the main window will launch zoomed or unzoomed. | Boolean |  |
| main\_window\_width\_ relative\_size | The variable holds the decimal percentage of the relative width of the main window to the screen width. Only has an effect when unzoomed. | Float |  |
| main\_window\_height\_ relative\_size | The variable holds the decimal percentage of the relative height of the main window to the screen height. Only has an effect when unzoomed. | Float |  |
| root | The variable holds the Tkinter window used in the Login class and later in the Main class. | Tkinter window | Global |
| popup | The variable holds the Tkinter window used for the Settings, Tables, AddClient and Appointments classes. Multiple popups may exist at one time because they are local. | Tkinter toplevel | Local |
| screen\_width | The sceen width in number of pixels | Integer | Global |
| screen\_height | The sceen width in number of pixels | Integer | Global |
| main\_window\_width | The initial width of the main window in number of pixels. | Integer | Global |
| main\_window\_height | The initial height of the main window in number of pixels. | Integer | Global |
| login\_window | The variable that holds the instance of the Login class. | Login (class instance) | Global |
| main\_window | The variable that holds the instance of the Main class. | Main (class instance) | Global |
| settings\_window | The variable that holds the instance of the Settings class. | Settings (class instance) | Global |
| appointments\_window | The variable that holds the instance of the Appointments class. | Appoint-ments (class instance) | Global |
| change\_password\_ window | The variable that holds the instance of the ChangePassword class. | Change-Password (class instance) | Global |
| tables\_window | The variable that holds the instance of the Tables class. | Tables (class instance) | Global |
| add\_client\_window | The variable that holds the instance of the AddClient class. | Add-Client (class instance) | Global |
| day | This varible holds the day of the month and later holds the day of the week. | Integer | Local |
| suffix | This variable holds the suffix of the day of the month as a string. | String | Local |
| statement | This variable holds the string that’s printed into the console after the prefix is added. | String | Local |
| c | The variable that holds an instance of the cursor class from the mysqlite3 module. | SQL cursor | Global |
| conn | The variable that holds an instance of the connection class from the mysqlite3 module. | SQL connection | Global |
| master | The variable that the root or popup is referred to inside a class defined in the program. It is stored as an attribute of that class. | Tkinter window | Local |
| \*\_var | These variables holds the value stored within Tkinter widgets (entry boxes or combo boxes). A lot of these widgets / variables exist within the program so this refers to any variable that ends is ‘\_var’. This variable is defined as an attribute of a defined class. | Tkinter String | Local |
| \*\_label | These variables hold Tkinter labels that are displayed within a Tkinter window. A lot of these widgets / variables exist within the program so this refers to any variable that ends is ‘\_var’. This variable is defined as an attribute of a defined class. | Tkinter label | Local |
| \*\_entry | These variables hold Tkinter entry boxes that are displayed within a Tkinter window. A lot of these widgets / variables exist within the program so this refers to any variable that ends is ‘\_var’. This variable is defined as an attribute of a defined class. | Tkinter entry | Local |
| \*\_button | These variables hold Tkinter buttons that are displayed within a Tkinter window. A lot of these widgets / variables exist within the program so this refers to any variable that ends is ‘\_var’. This variable is defined as an attribute of a defined class. | Tkinter button / Tkinter label | Local |
| \*\_image | These variables hold Tkinter photo images that hold a .png image found in the /resources/ folder and makes them displayable with a Tkinter window. A lot of these widgets / variables exist within the program so this refers to any variable that ends is ‘\_var’. This variable is defined as an attribute of a defined class. | Tkinter photo image | Local |
| id | This variable holds the values stored within the first Tkinter entry box in the login window. | Integer | Local |
| p | This variable holds the values stored within the second Tkinter entry box in the login window. | String | Local |
| user | This array stores the record of the user that has a StaffId equal to the value in the id variable. | Array of strings | Global |
| width | This variable holds the width of the main window and dynamically changes value when the width of the main window is changed. | Integer | Local |
| height | This variable holds the height of the main window and dynamically changes value when the height of the main window is changed. | Integer | Local |
| update | This variable holds a boolean that results in the main window being re-initilised when exiting the settings window, if true. | Boolean | Local |
| widget | This variable refers to any Tkinter widget. Used very short term. | Tkinter widget (label or button or list box, etc.) | Local |
| style | This variable hold the Tkinter style of text. It allows the text in the tabs in the Tkinter notebook to be changed. | Tkinter style | Local |
| current\_theme | This variable holds the current them being used. It is stored as a variable to apply the style to the tabs in the Tkinter notebook. | Tkinter theme | Local |
| time | This variable holds the time in ‘hh:mm:ss’ format of the current time. | String | Local |
| date | This variable holds the date in ‘dd:mm:yy’ format of the current date. | String | Local |
| day\_of\_the\_week | This variable hold the day of the week as an Integer where 1 is Monday and 7 is Sunday. | Integer | Local |
| hairdressers | This 2D array holds all the records of the hairdressers. | 2D array of strings | Local |
| surname\_clash | This array holds the index in the hairdresser array of all records that have clashing surnames. | Array of integers | Local |
| nb | This variable holds the Tkinter notebook that is dispalyed in the main window. | Tkinter notebook | Local |
| tabs | This array holds the frames displayed in each tab in nb. | Array of Tkinter frames | Local |
| lbs | This 2D array holds all the list boxes displayed in each tab for each hairdresser. | 2D array of list boxes | Local |
| names\_frame | This array holds the frame that holds all the canvases that display the hairdressers’ names for each tab. | Array of Tkinter frame | Local |
| names | This 2D array holds all the canvases displayed for each tab for each hairdresser. | 2D array of Tkinter canvases | Local |
| scrolls | This array holds the Tkinter scroll bar for each tab of nb. | Array of Tkinter scroll bars | Local |
| hover | This integer holds the parameter of the on\_hover function. The label that has its image changed depends on this integer. | Integer | Local |
| appointment\_id | This array holds all the appointment ids for each list box. It is defined as an attribute of a list box. The first index holds the appointment id of the first appointment and the nth index holds the appointment id of the nth appointment. | Array of integers | Local |
| start\_time | This variable holds the appointment time for an appointment before being appended in the list box that displays times. | String | Local |
| appointments | This 2D array holds all the records of appointments from the database. | 2D array of strings | Local |
| client | This array holds the record for the client for the appointment booked. | Array of strings | Local |
| \*\_box | This array holds the Tkinter combo boxes (drop-down menus) that are displayed within a Tkinter window. A lot of these widgets / variables exist within the program so this refers to any variable that ends is ‘\_var’. This variable is defined as an attribute of a defined class. | Tkinter combo box | Local |
| table | This variable holds the table being editted by an admin where 0 is the Staff table, 1 is the Client table and 2 is the Haircut table. | Integer | Local |
| entry | This variable holds the Tkinter entry box used to edit the tables within the database. It will appear when the user clicks a record and displays when clicked away. | Tkinter entry box | Local |
| changes | This 2D array holds all the pending changes made to the database before they are applied. | 2D array of strings and integers | Local |
| fields | This array holds the field names as strings. | Array of strings | Local |
| fields\_descs | The array holds the descriptions of the fields so the user can click the field name to find out more. | Array of strings | Local |
| records | This 2D array holds all the records for the selected table. | 2D array of integers and strings | Local |
| new\_record | This array holds the initial values for all the fields when a new record is added. | Array of integers and strings | Local |
| record\_required | This array contains the boolean values for if the field is required for all the fields | Array of booleans | Local |
| result | This variable holds the result from a SQL query. | Array of strings and integers | Local |
| scroll | This variable holds the Tkinter scroll bar used to scroll through the list boxes in the tables window. | Tkinter scroll bar | Local |
| hover | This variable holds if the user’s cursor is currently over the entry box. It is stored as an attribute of the entry box and is required so the program knows where the user clicked. | boolean | Local |
| hidden | This varable holds if the entry box in the tables window is currently hidden or showing. | boolean | Local |
| forced | This variable holds a boolean value which will force the entry box to appear next time the user clicks a list box. | boolean | Local |
| new\_val | This variable holds the value of the entry box just before it is hidden. The variable is validated and then updated to the record. | String | Local |
| alpha\_exists | This variable simply keeps track if atleast one character in the password is alphabetical. | boolean | Local |
| digit\_exists | This variable simply keeps track if atleast one character in the password is a digit. | boolean | Local |
| admins | This variable counts how many users are admins. | Integer | Local |
| field | This varible holds what field the entry box is located in. This variable is defined as an attribute to the entry box. | Integer | Local |
| record | This variable holds what reocrd the entry box is located in. This variable is defined as an attribute to the entry box. | Integer | Local |
| yview | This variable holds what the yview of the list boxes was before the entry box was unhidden. It returns the y view of the list boxes to this value after. | Float | Local |
| y | This variable holds the y value for the entry box. | Integer | Local |
| lb | This variable holds which list box the user clicked to open the appointments window. | Integer | Local |
| line | This variable holds which line of the list box the user clicked to open the appointments window. | Integer | Local |
| id | This variable holds the id of the appointment the user clicked to open the appointments window. | Integer | Local |
| rate | This variable holds the rate of the hairdresser that the new appointment is being made for. | Integer | Local |
| clients | This 2D array holds all the clients’ details from the Clients table. | 2D array of integers and strings | Local |
| haircuts | This 2D array holds all the haircuts’ details from the Haircuts table. | 2D array of integers and strings | Local |
| client\_names | This array holds all the clients’ names. | Array of strings | Local |
| haircut\_names | This array holds all the names of the haircuts. | Array of strings | Local |
| durations | This array holds the different durations the haircut can be. | Array of strings | Local |
| appointment | This array holds appointment record from the databade where the appointment id equals the id variable. | Array of integers and strings | Local |
| client | This variable holds the client’s name that had the client id found in the appointment record. | String | Local |
| haircut | This variable holds the haircut’s name that had the client id found in the appointment record. | String | Local |
| duration | This variable holds the duration of the appointment from the appointment record. | Integer | Local |
| amount\_to\_pay | This variable holds the value of the amount the client is paying for the appointment. This value depends on what the user entered into the entry box found when clicking an existing appointment. | Float | Local |
| first\_name | This variable holds the value the user entered into the first name entry box found in the AddClient window. | String | Local |
| last\_name | This variable holds the value the user entered into the last name entry box found in the AddClient window. | String | Local |
| mobile\_phone | This variable holds the value the user entered into the mobile phone entry box found in the AddClient window. | String | Local |
| home\_phone | This variable holds the value the user entered into the home phone entry box found in the AddClient window. | String | Local |

# Testing

## Testing strategy

During my implementation of my program, I preformed black box testing regularly when adding any new code. During the development of the program a lot of variables were printed at different stages so I could tell if variables were holding the expected values and if any errors occurred, I could debug the error using the printed variable values in the console.

I tested each part of the program with varying inputs. I used expected inputs as well as unexpected inputs in order to test if the program can avoid any errors or falsely validate values. If the inputted data is not valid the program should notify the user of any errors they have made. For example, on the login window when the user is prompted to enter a staff ID, if they enter a non-integer value such as a string composing of letters, then the program will notify the user.

## Pre- Planned Tests

The evidence for these tests can be found in Appendix E or by clicking the linked figure next to the test.

|  |  |  |  |
| --- | --- | --- | --- |
| Test Description | Expected Outcome | Actual Outcome | Passed?  Linked to figure |
| Test to see what happens when the Staff ID or password field is left blank when logging in. | The program detects the blank field and tells the user it’s invalid. | Successful. | Figure 1a, Figure 1b, Figure 1c |
| Test to see what happens when the Staff ID doesn’t exist in the database. | The program cannot find the Staff ID in the database and tells the user one of the fields is incorrect. | Successful. | Figure 2 |
| Test to see what happens when the password entered is not the correct password for that corresponding Staff ID. | The program will detect the entered password does not match the password in the database and tells the user one of the fields is incorrect. | Successful. | Figure 3 |
| Test to see what happens when the Staff ID is a invalid datatype. | The program detects the Staff ID is not a digit and tell the user one of the fields is incorrect. | Successful. | Figure 4a, Figure 4b |
| Test to see if the interactive labels work correctly (if the user can click | The program should change the appearance of the label to indicate it’s interactable (clickable). Once clicked, the label should show some sort of response. | Successful. | Figure 5a, Figure 5b |
| Test to see if the tabs change day and the appointments for that day. | The program should change the appointments that are being displayed in the table. | Successful. | Figure 6a, Figure 6b |
| Test to see if the arrows next to the week heading can navigate through weeks. | The program should move back a week when the user clicks the left arrow and move forward a week when the user clicks the right arrow. | Succesful. | Figure 7a, Figure 7b, Figure 7c |
| Test to see if the log out button returns the user to the login window. | The program should log the user out of the program. The user should see the same window that they would have saw when launching the program. | Successful. | Figure 8a, Figure 8b |
| Test to see if the program allows the user to view existing appointments in the table. | The program should open a new, smaller window that includes all the relevant information for that appointment. | Successful. | Figure 9 |
| Test to see if the user can mark an appointment as paid and alter the amount the user pays. | The program should allow the user to change the value in an entry box. Once the user is happy with the value they should be able to mark the appointment as paid. | Successful. | Figure 10a, Figure 10b, Figure 10c |
| Test to see if the program allows the user to add an appointment to the table. | The program should open a new, smaller window that allows the user to enter the details required to make a new appointment. | Successful. | Figure 11 |
| Test to see if the program allows the user to select options in the drop-down menus found in the appointment and settings window. | The program should show a list of options when the user clicks a drop-down menu. The user should then be able to select an option in the menu. The drop-down menu should show that value selected. | Successful. | Figure 12a, Figure 12b, Figure 12c, Figure 12d, Figure 12e |
| Test to see if the program allows the user get open the window that allows the user to add a client through the appointments window. | The user should be able to click the button next to the client drop-down menu to add a client. Alternatively, the user can select the add client option in the drop-down menu. | Successful. | Figure 13a, Figure 13b, Figure 13c |
| Test to see if the user can add a client to the database through the add client window. | The user should be able to fill in the four entry boxes with values. | Successful. | Figure 14a, Figure 14b |
| Test if the program checks for any empty fields. | The program should detect if any fields are empty and act accordingly. The first name and surname fields are required but the phone number fields are optional. If both phone number fields are empty the program will warn the user. | Successful. | Figure 15a, Figure 15b, Figure 15c |
| Test if the first name and surname values are the correct datatype. | The program should check if the firstname and surname contain only alphabetically characters. | Succesful. | Figure 16a, Figure 16b |
| The user should be able to add a client. | The program should add the client to the database. It should also return the user back to the add appointments window with the new client as the selected client. | Sucessful. | Figure 17a, Figure 17b, Figure 17c |
| The user should be able to open the settings window. | The program should open the settings window when the user clicks the settings window. | Successful. | Figure 18a, Figure 18b |
| The user can open the change password window. | When the user clicks the change password button in the settings window. The program should open the change password window. | Successsful. | Figure 19 |
| The user should be able to enter values into the entry boxes in the change password window. | The user should be able to enter values into the entry boxes. | Successful. | Figure 20 |
| The program should not let the user change their password if the current password they entered does not match the one in the database. | The program should tell the user they have entered their current password incorrectly. | Successful. | Figure 21 |
| The program should not let the user change their password if they have typed their new password incorrectly. | The program should compare the enter new password and re-enter new password entry boxes to check if they match. If they do not match, it should tell the user. | Successful. | Figure 22 |
| The program should not let the user change their password if it’s not strong enough. | The program should check if the password contains atleast 8 characters, with atleast one alphabetical character and one digit. | Successful. | Figure 23a, Figure 23b, Figure 23c |
| The user should be able to change their password. | If the old password matches the password in the database and the new password is valid, then change the password for the user logged in in the database. | Successful. | Figure 24a, Figure 24b, Figure 24c |
| The user should be able to open the tables through the the settings window, only if they have admin rights. | The program should check if the user has admin rights. If they do the program should open the selected table in a new window. | Successful. | Figure 25a, Figure 25b, Figure 25c, Figure 25d, Figure 25e, Figure 25f |
| An admin should be able to add a record to a table. | The program should add a record to the table inside the tables window, when the user clicks the add record button. The user should then be able to enter values into the record and apply it. | Successful. | Figure 26a, Figure 26b, Figure 26c, Figure 26d |
| An admin should be able to remove an existing record from the database. | The user clicks the remove button and it cause the next record that’s pressed to be removed. When the user clicks apply it will be removed from the database. | Successful. | Figure 27a, Figure 27b, Figure 27c, Figure 27d, Figure 27e |
| An admin should be able to edit existing records. | The user can click a value in the record. It should allow the user to edi the value. When the user clicks apply it should update the database. | Successful. | Figure 28a, Figure 28b, Figure 28c, Figure 28d, Figure 28e |
| The main window’s widgets should dynamically change size and position when the window is resized. | The program should re-position all widgets in the main window that depend on the windows width and height. | Successful. | Figure 29a, Figure 29b |
| The program should log all important changes made with the program. | The program should print the log to the console with the date and time it happened. If necessary it should also include the user who made the change and any other important details. | Successful. | Figure 30 |

# Evaluation

## Test results

From intensive testing I have concluded everything performed as expected and no errors were encountered. The figures attached prove the program meeting the success criteria. The testing also proves the program validating inputted data. The program did not let any unexpected values to be accepted and therefore the program will not have to deal with unexpected datatypes and data format because of this. From these results, I can say my program is ready to be shown as a final version to the client.

## Objectives and my experience while programming

My aim was to create a booking system with a GUI that would allow users to quickly and easily make appointments was successful. Once logged on the user can locate an existing appointment in a few seconds and add a new appointment in a few seconds. With these two common tasks, the user has already saved massively on relative time spent on this task compared to the old system. Also because of the logging, any appointments removed are traceable by the admins back to the user. So any unauthoried appointment removals, will be able to be traced back to the user who removed it. Also the program logs all the appointment information during the removal so the appointment can be re-added back to the appointments table if necessary.

My objectives to learn and use the two Python modules Tkinter and SQLite3 were successful. I learnt about a lot of different Tkinter widgets and there different uses. Working out how to implement the notebook widget with frames as tabs and more widgets inside those frames was challenging. I decided to use vertically stacked list boxes instead of the tree view widget as it provided more customisation to allow me to meet my success criteria. The biggest challenge with Tkinter was creating a database browser/editor, while keeping the GUI user-friendly. I felt the end result was simple but functional. Working out how to edit the list boxes required experimenting, but I ended up using a temporary entry box over the cell to give the user the illusion they were editing the table itself. I also challenged my Python skills by working out how to not apply the changes made in the table until the user wanted to. This required storing the changes in a 2D array and then making all the changes to the database once the user clicked apply.

SQLite3 was rather simple and only required a few functions to be learnt. You would open a connection followed by a cursor and then make a SQL query with the cursor. Once finished you simply close the cursor and connection down. This module challenged my SQL skills the most. Finding a way to retrieve all the staff member’s information from the Staff table who had there Staff ID in the Hairdressers table was difficult.

A new modules called datetime was also used. It allowed my to get the dates instead of the days of the week which meant the program could tell the user more information. Also datetime was used to make a functional clock which ran behind a recursive function that called itself every 200ms.

Success Criteria  
The program needed to achieve the following criteria to be successful and meet my client’s requirements. It should:

* Provide a basic security system so unauthorised people cannot access the system while unoccupied.

The program features a login window that locks the main program behind a Staff ID and password. The password is stored not within the program but in a database. This login window stops any strangers from accessing the system and messing with appointments. It also allows the admins to keep of track of when and who is logged in at a certain time. In future releases I would like to introduce encryption to the security system. Or instead of storing the password in the database store a hash so the password cannot be retrieved from the database.

* Allow users to view all existing appointments.

Once logged in, the user can click any existing appointment within the appointments table. This will bring up a new window with all the relevant details to this appointment including the client who booked it, the hairdresser its booked with, the time it starts and finishes, the data, the type of service / haircut and the amount to pay.

* Allow users to add appointments and ask only the relevant information. This includes any day or week.

The user can click any empty appointment slot in the appointments table to open a new window. This window will have three drop-down menus where the user can enter the client, haircut and duration of the appointment. When the user enters the haircut, the estimated duration is automatically filled in, but can be changed by the user. The hairdresser and time of the appointment is already known due to the slot that the user clicked. The user can press the left or right arrow keys to navigate back and forward a day or click a corresponding tab at the top of the table. The user can also navigate quicker through weeks instead of days using the arrow buttons above the table.

* Display to the user the price of the appointment in advance, taking into account the type of service being provided and which hairdresser is providing it.

The user can see the price of an appointment during the process of making an appointment. The price of the appointment is displayed next to each type of service / haircut when the user opens the drop-down menu. These prices have taken into account the rates of the hairdressers. The price of the appointment can also be seen for existing appointments by simply clicking on an existing appointment. The window opened will display this price of the appointment and all other appointment information.

* Allow users to mark appointments as paid.

The user can click on an existing appointment to open up the information window for that appointment. They will then have an option to mark as paid as well as customise the amount to pay. The amount to pay can be altered in case the client has a family-discount or promotion discount from the company.

* Allow users to remove existing appointments.

The user can click on an existing appointment to open up the information window for that appointment. From here they can click the cancel button. The user will get a confirmation window confirming their choice to cancel. If the user clicks yes, the appointment will be removed from the database and the appointments table will be updated.

* Allow users to add clients to the database in a user-friendly way.

The user can add a client from the add appointment menu. After clicking an appointment slot, the user will be presented a drop-down menu to select from clients within the database. The user can either click the button to the right of the drop-down menu or alternatively click the last option in the drop-down menu that reads ‘Add client’. Either method will bring up a new window that contains four entry boxes. The user then is required to enter the first name and surname for this client. The other two entry boxes are for phone numbers and are optional, but recommended. Once the user clicks add the client will be added to the database.

* Allow users to change their password.

The user can click the settings button top left of the main window then click change password. The user will then have to enter their old password and then enter their new password twice. If the old password matches the one in the database and the new passwords match the user’s password is updated in the database.

* Allow admins to access and view the tables located the database within the Python program.

The user can click the settings button top left of the main window then select one of the tables in the drop-down menu. If the user has admin rights, they will be taken to the tables window where the selected table will be displayed alongside all the records within that table.

* Allow admins to add, edit and remove records in the database within the Python program.

The user can edit any fields within the table by clicking on the value’s cell. The user can enter a new value in the cell. If the value is valid it will update the table. The user can also add records by clicking the add record button. The user must fill all required cells before being allowed to apply this new record. The user can also click the toggleable remove button which will result in the new record the user clicking being removed from the table. Any changes made in the table are only applied to the database when the user clicks apply. The user can discard changes if they are not happy with them before clicking apply.

## Using Python over other programming languages

Python was the most ideal language for me to program this booking system in as I already had prior knowledge. Although I had never used Tkinter and SQLite3 modules before they were an educational challenge for me. I selected Python because of it’s a scripting language that is easy to use and learn. Alternative programming language like C++ would have allowed me to make a similar booking system but in much more code and therefore would have been more time consuming. Python is easy to debug as it is an interpreter and runs code line by line. Therefore, when my program encountered errors, Python would point to which line and section of that line was causing the error. There are drawbacks with it being an interpreter, such as always needing Python installed to run your program. Although in this situation the program was only being run by me and my client so it’s a very small problem and there are programs that can make the Python program a standalone executable. More advanced programs such as C++ would also cause the programming experience overall more stressful as it’s an unfamiliar and harder language to learn.

## Existing solutions

The program is much more user-friendly compared to Microsoft Excel and is much cheaper and affordable for a small company compared to systems like SystemBookings. It is more efficient that the previous booking system with a pencil and appointments book. The navigation improvements from the pencil and appointment book are hugely beneficial to the company and should shorten call times and let save staff and clients time.

## Future Improvements

There are many improvements I could make to my program in the future and improvements to the methods I used to make my program.

### What I could have done differently

The appointments table had a lot of re-designs while making it. Overall, I’m happy with the final design however it was very challenging to implement. I decided to use stacked list boxes vertically where each list box represented a column. However, I wish I had just stuck to using the Tkinter tree view widget instead as it was an already formed table.

### What I could improve in the future

There are lots of improvements I would like to make to future improvements of my program. These include adding a system that could let admins mark users as ill on certain days to stop appointments from being booked on that day. Also add lunchbreak slots so the hairdressers are forced to get some relaxing time, as legally required.

There could be automatic daily backups of the database file. This could be as simple as copying the database file and renaming it in a date format so admins know which date it is a backup of. And also make the program delete backups after a period of time, such as 90 days. This means if in the rare chance the database corrupts, the company will at least have a recent backup of the system, and check logs for any missing appointments. I would also have liked to save logs as a text file where a text file is generated every time the program closes. Therefore, the logs are more accessible to admins.

Introducing encryption for the database could further improve the security. The database could instead store a hash for each password so people cannot just find users passwords with access to the database.

A feature that was originally meant to be included but had to be removed due to time was a search bar. This was an entry box that would allow the user to type in keywords and the SQL could find any records with that word in the database. It would benefit the user when extremely busy or just struggling to find an appointment. However, I am still happy with the current navigation and feel finding existing appointments is already quite easy.

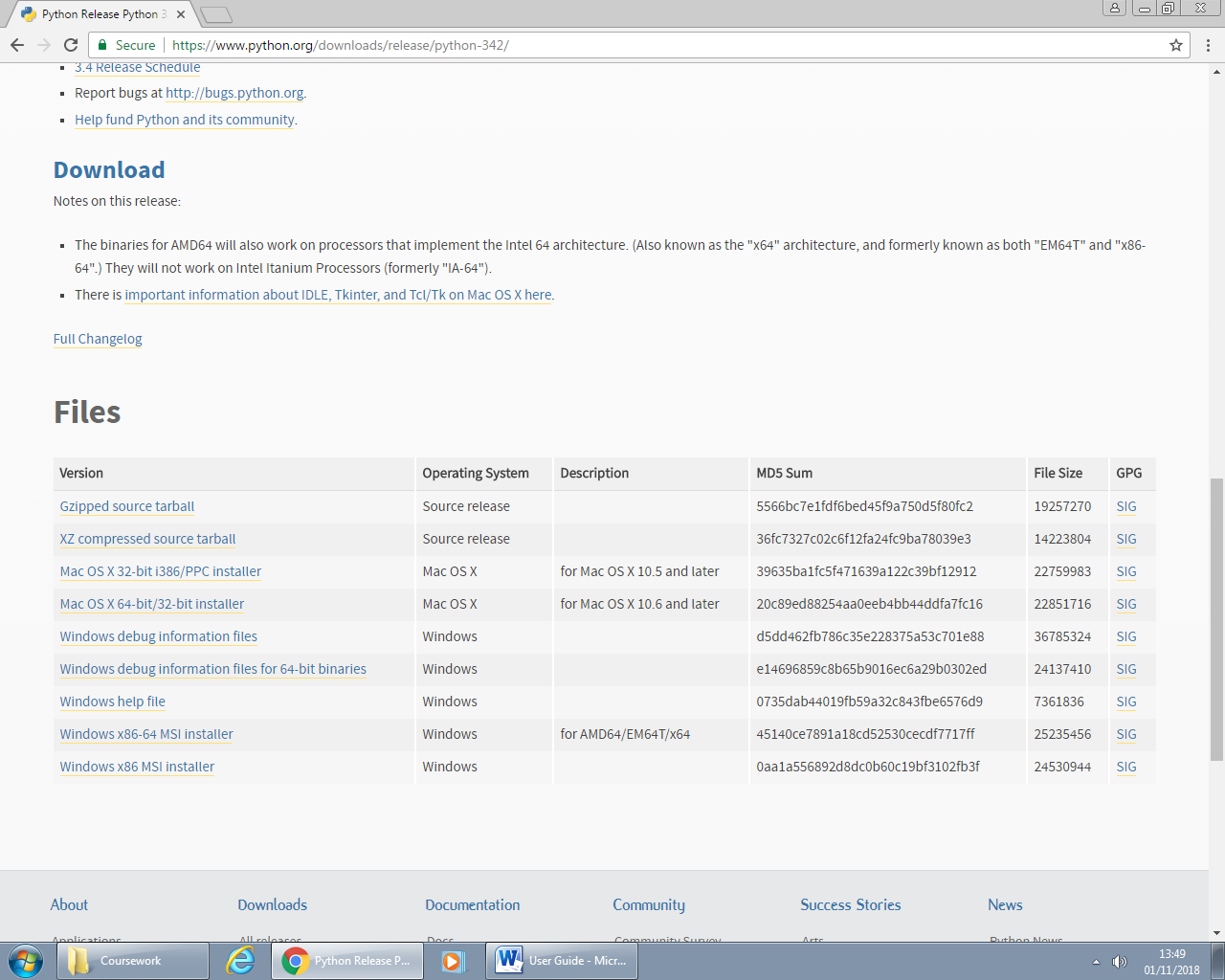
Overall there are a few minor changes I would like to make such as cleaning up some code (especially in the tables class) to make it clearer of what’s happening. Also a few very minor bugs such as the y view of the tables not being correct when adding a record, but these are very minor issues that was have little to no effect on the user. But the program is working as intended and has no major bugs through testing so is ready to be delivered to the client to be used.

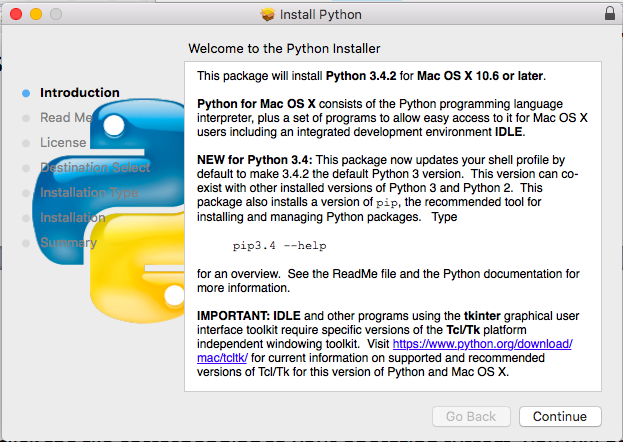
# User Documentation

## Python

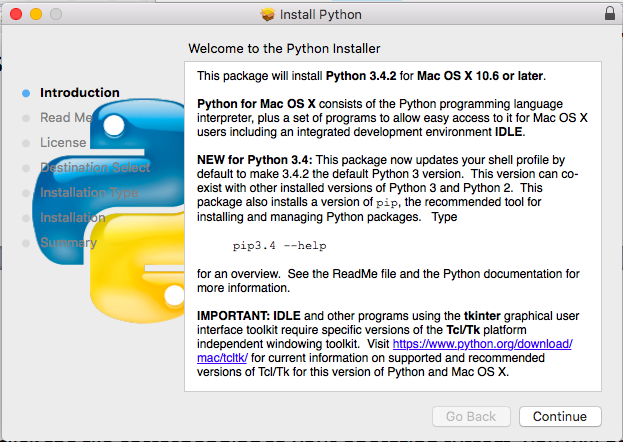
The Python software is required as the interpreter for the program. Therefore, it is required to run the program. We **highly recommend** you use the version 3.4.2, as it is the version the program was designed for.

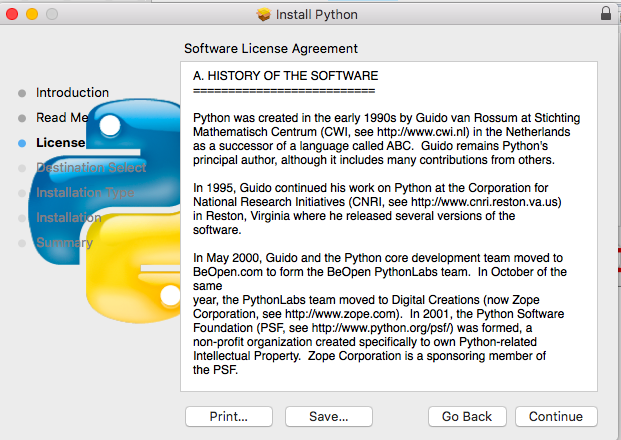
### Installation

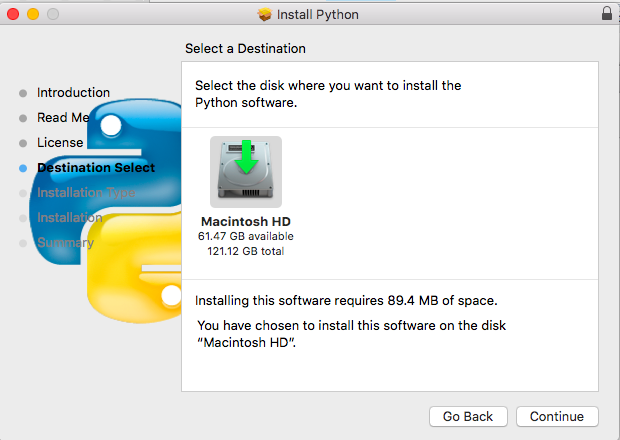
1. Make your way to the Python download page and scroll down until you find Python version 3.4.2, or click [here](https://www.python.org/downloads/release/python-342/) to directly get to the page.
2. Scroll down a page and a half until you get to the section shown below.
3. Click the file ‘Mac OS X 64-bit/32-bit installer’ if you are on a Mac OS X or the file ‘Windows x86-64 MSI installer’ if you are on a Windows.
4. Once you click the file corresponding to your operating system, you will have a new file within your downloads folder or a custom directory, if set. Navigate to your downloads folder and open the file labelled ‘Python-3.4.2.XXXXX.msi’.



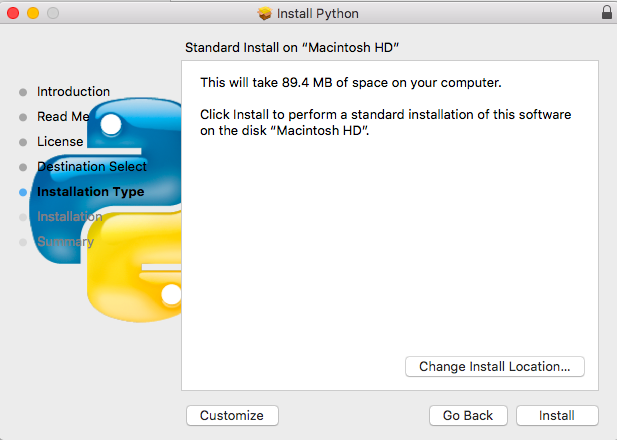
1. Click ‘Continue’.



1. Read the ‘Read Me’ then again click ‘Continue’.
2. Read the ‘Software License Agreement’ then again click ‘Continue’. If a prompt is displayed just click ‘Agree’ if you agree.



1. Select your preference of drive. We recommend you keep it as your default drive. After just click ‘Continue’.



1. Select your preference of install location. We again recommend you stay for your default location. After just click ‘Install’. You may be prompted to enter your computer’s password in order to make changes.
2. After the installation, wait until the program informs you it’s complete. Just click ‘Done’.

Now Python is installed there should be a folder next to the original file. You may now delete the original executable installer file, as it’s no longer needed.

### How to open/edit the program

In this case the only reason for Python to be install is to use it as an interpreter.

1. Navigate to the ‘program.py’ file. It should be under the directory ‘/Essentials Booking System/’ in your download folder.
2. Once you’ve located the ‘program.py’, right-click on the file. Hover over ‘Open with’ and then click ‘Python IDLE’.
3. *If you wish to just run/use the program skip this step.* Here you can see the code for the program. If you would like to make any changes to the program by altering the default values of any variables or adding your own code. Ensure you always **backup** the file first and know what you are doing when making changes.
4. Click ‘F5’ on your keyboard to run the program.

The booking system is now running.

## SQL DB Browser (optional)

If you would like to open the database outside the program you will need to install an external program that’s capable of opening database (.db) files. This is not necessary to use the program but recommended if you need to perform maintenance. The suggested program to use is SQLite Browser as it’s lightweight and user-friendly.

### Installation

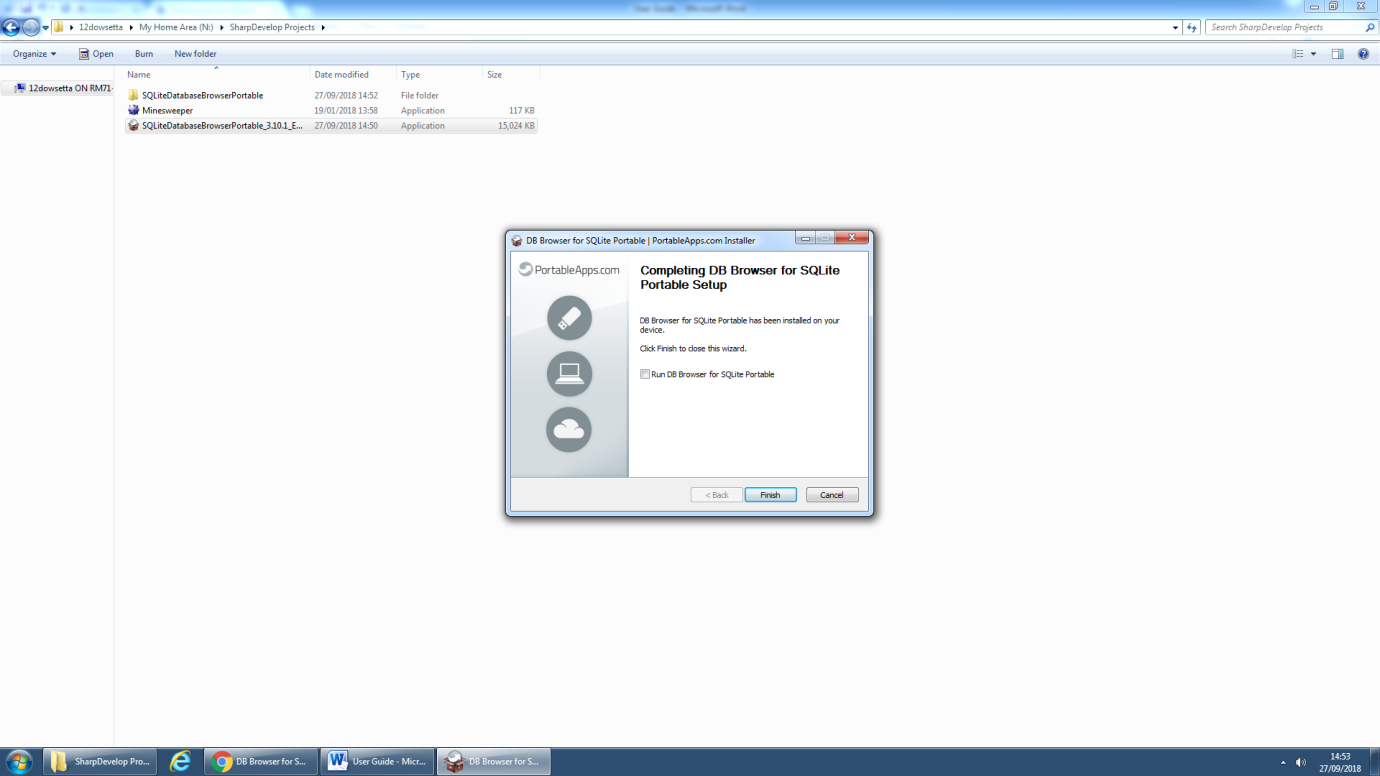
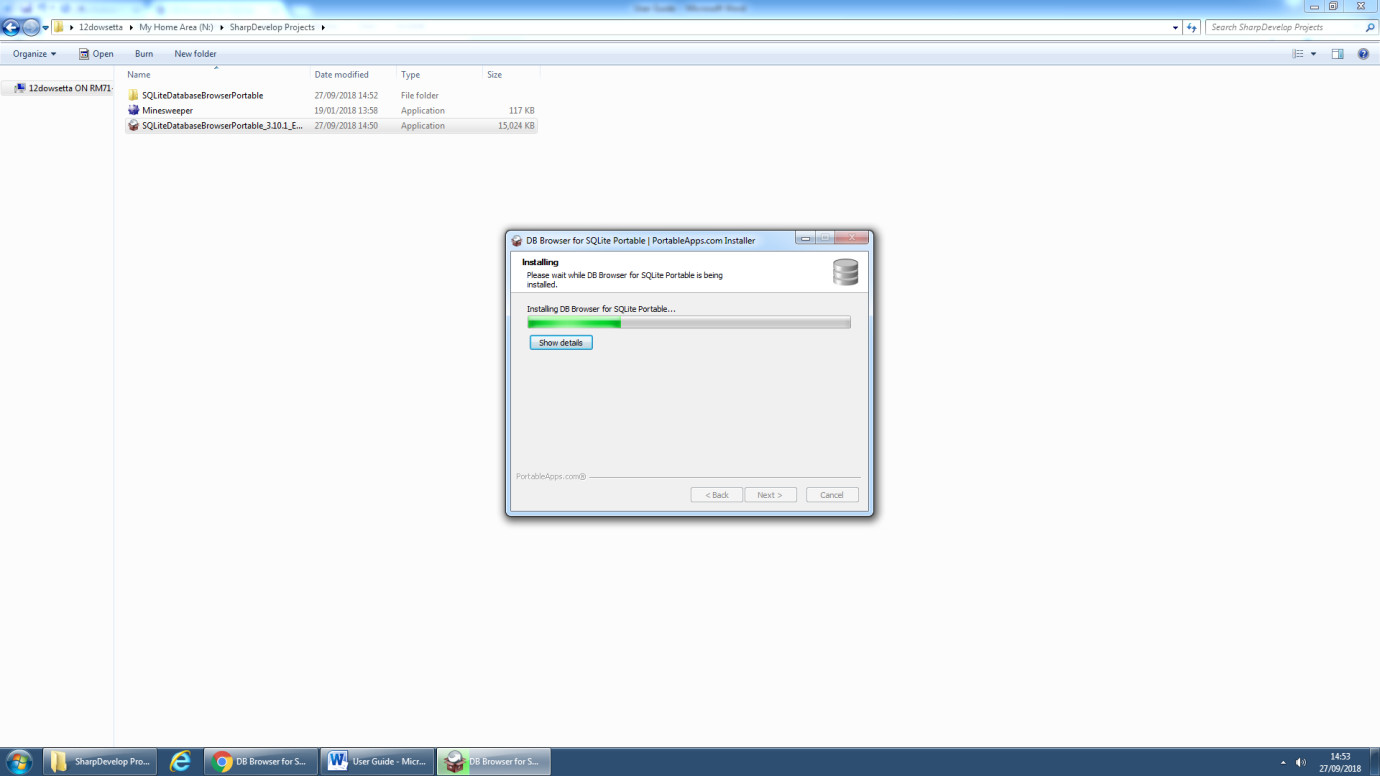
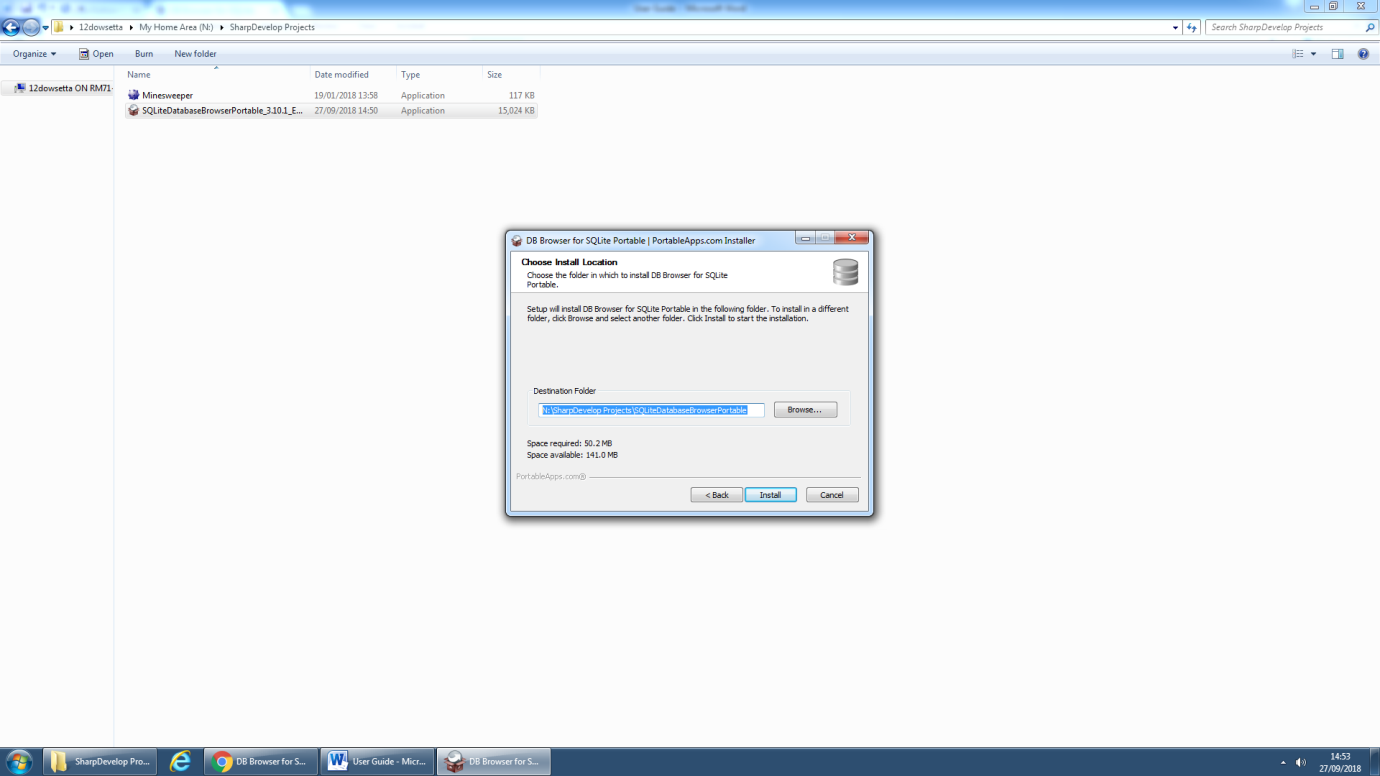
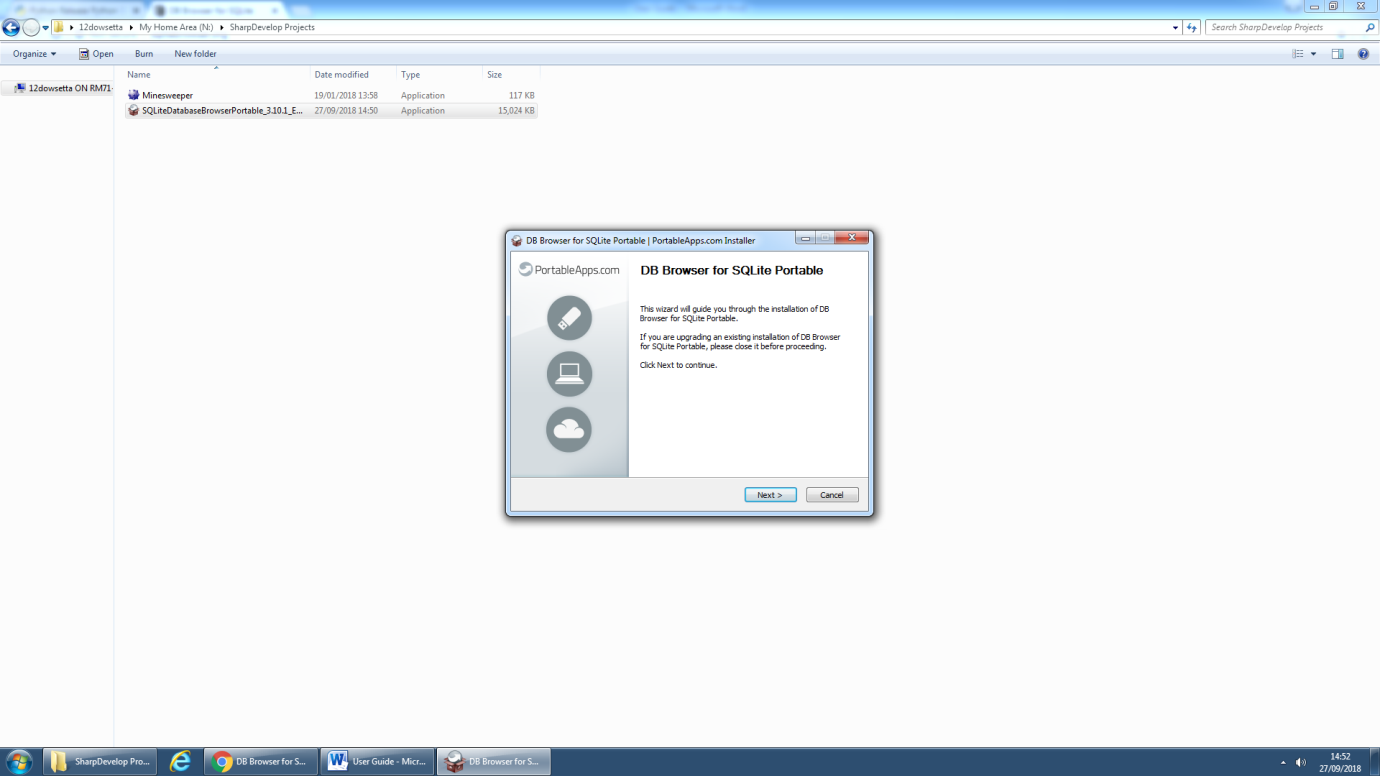
1. Make your way to the SQLite Browser website. You can either google ‘SQLite Browser’ and click the first result or more simply click [here](http://sqlitebrowser.org/). It should take you to a page that looks similar to below.



1. Click on the ‘PortableApp’ version and it should start your download.

***NOTE:*** *If you will to install a different version read the following.* If you want the program installed onto your program files instead of an isolated folder click ‘Download 64-bit Windows .exe’. If you are on a mac operating system click ‘Download Mac .dmg’. The next steps may very slightly differ.

1. You will have a new file within your downloads folder or a custom directory, if set. Navigate to your downloads folder and double-click to open the file labelled ‘SQLLiteDatabasePortable\_XXX.exe’.



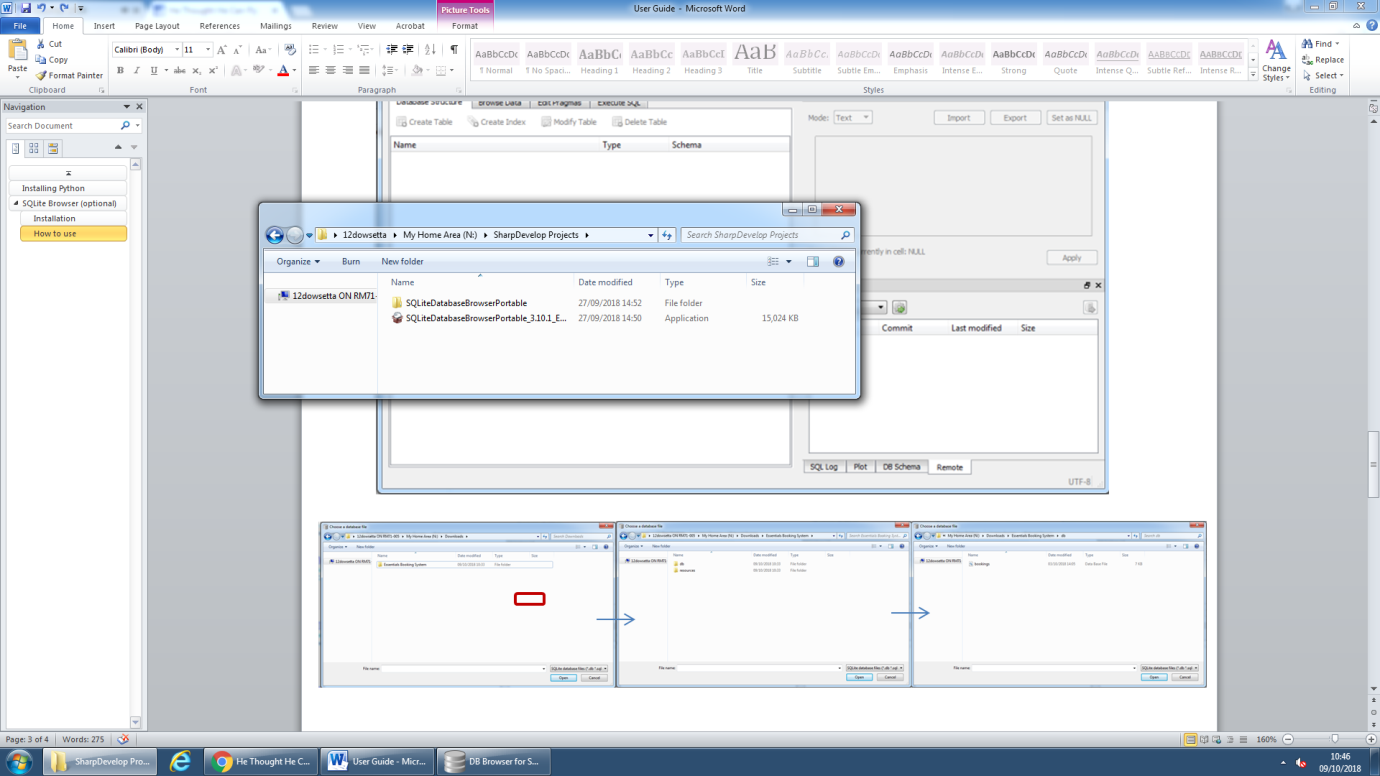
1. Click ‘Run’ on the popup window.
2. Click ‘Next’.
3. Click ‘Install’.
4. Wait for the program to install. This may take some time depending on your computer’s speed.
5. Click ‘Finish’.

*If you installed the PortableApp version of the program there should be a generated folder next to the installer file. Keep this folder safe as this contains the SQL Browser program and files.*

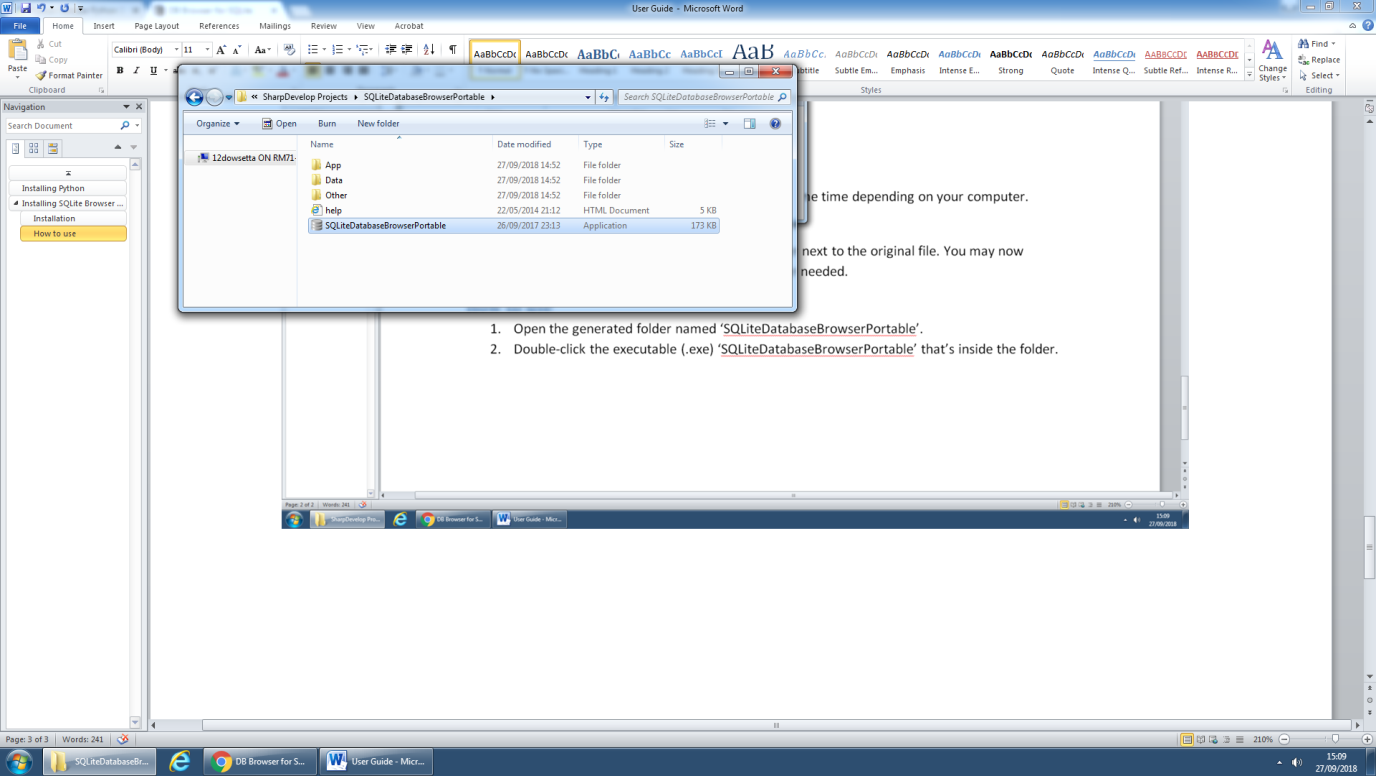
Now SQLite Browser is installed there should be a folder next to the original file. You may now delete the original executable installer file, as it’s no longer needed.

### Opening SQL DB Browser

1. Open the generated folder named ‘SQLiteDatabaseBrowserPortable.

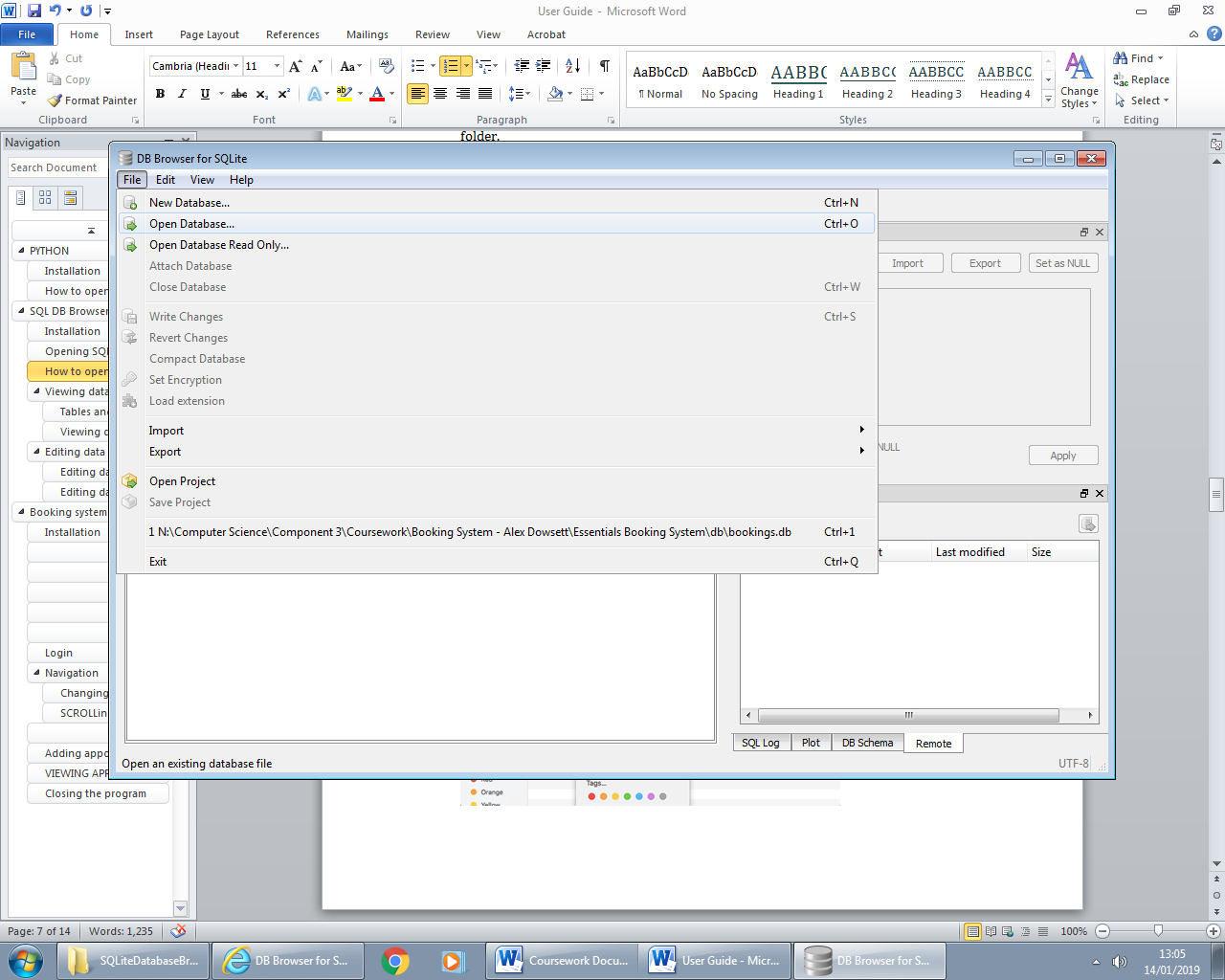


1. Double-click the executable (.exe) ‘SQLiteDatabaseBrowserPortable’ that’s inside the folder.



### How to open database

1. Open the DB SQL Browser. *See above steps if necessary.*
2. Click ‘File’ then from the dropdown click ‘Open Database...’.

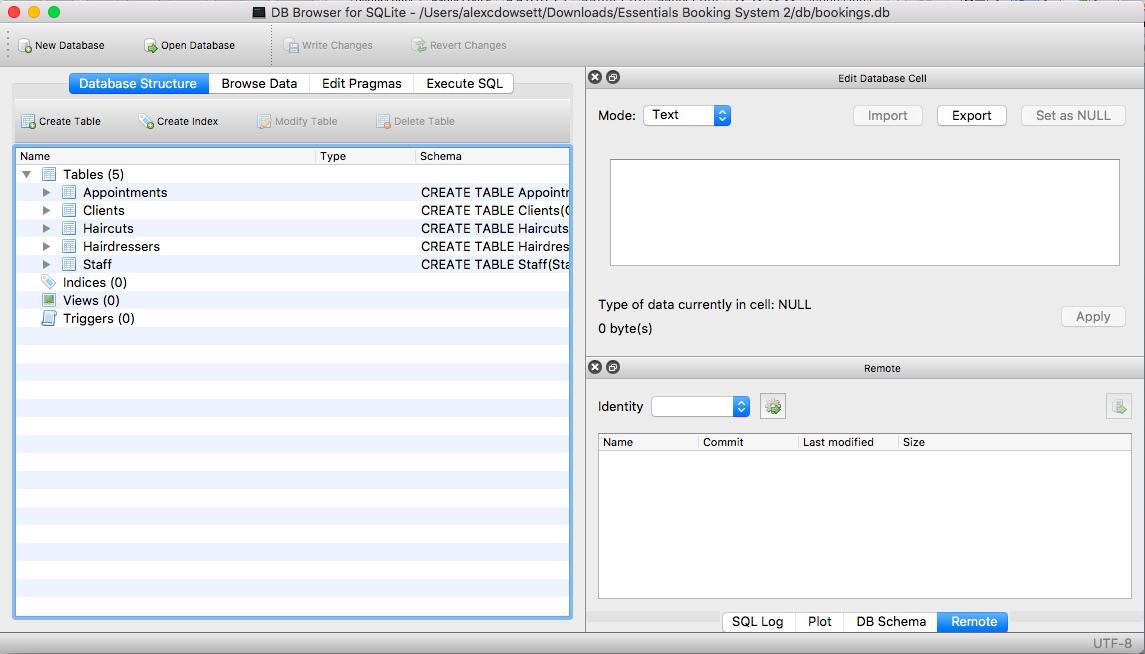


1. Navigate to your downloads or a custom directory, if set. Find the database file in the directory, ‘/Essentials Booking System/db/bookings.db’.
2. Double-click on the file ‘bookings.db’.

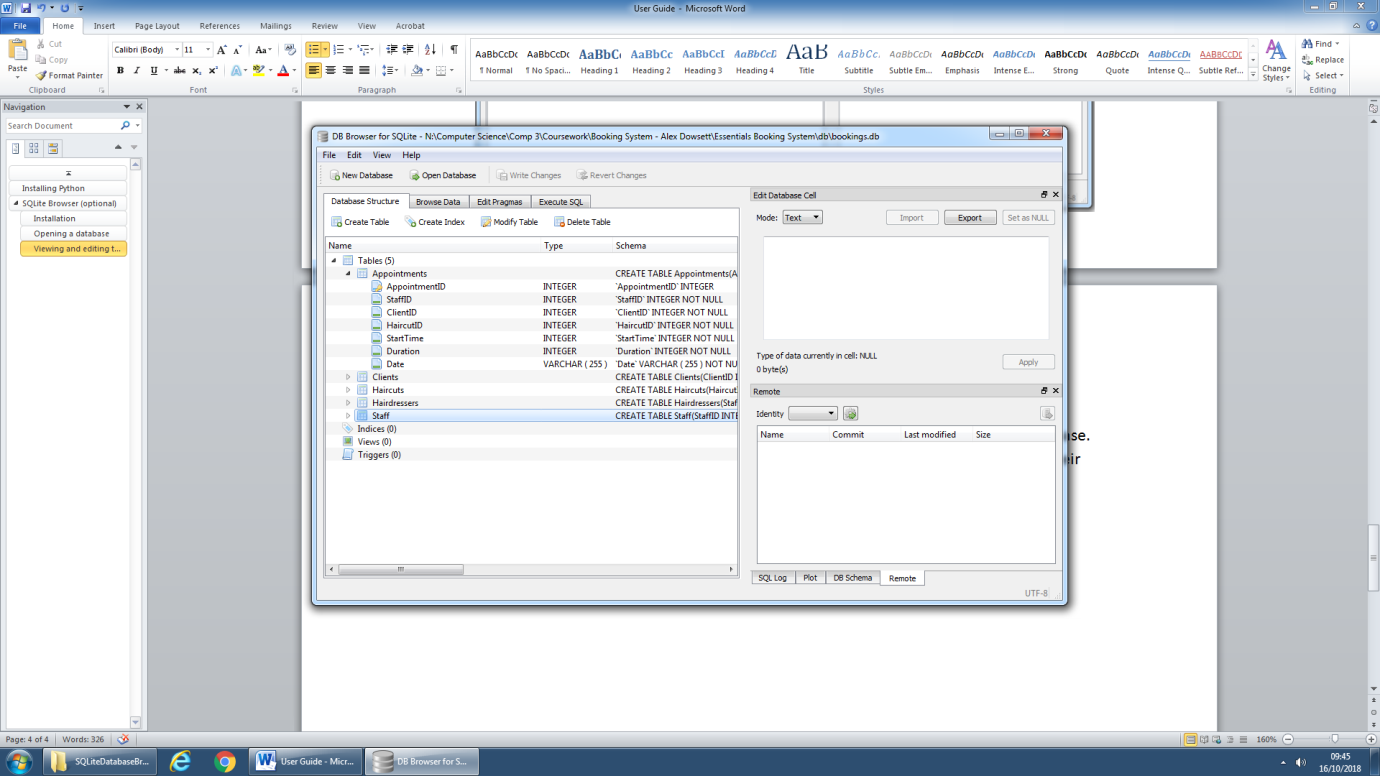
Now you have ‘bookings.db’ open in SQL DB Browser.

### Viewing data within the database

#### Tables and their properties

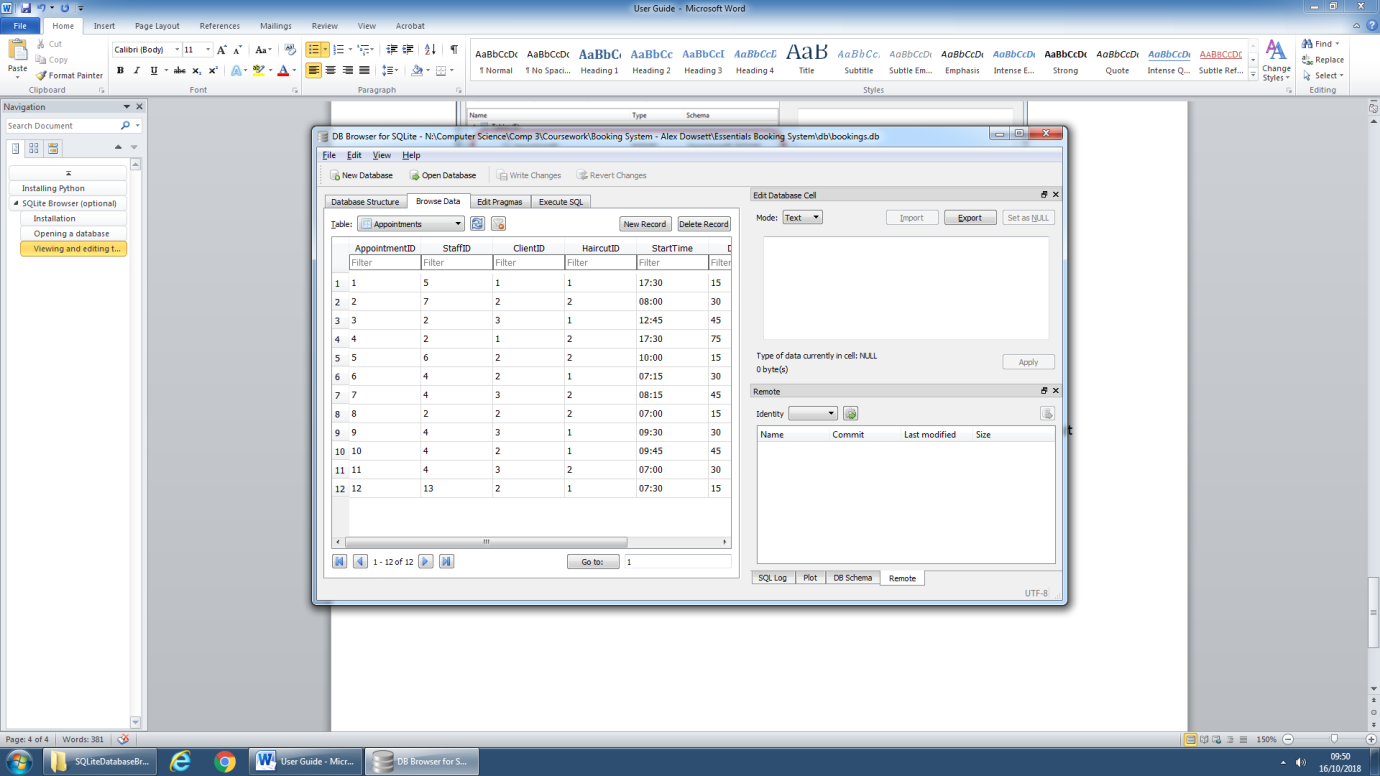


Once the database is opened you will be represented with a list of tables inside the database. Double-clicking on a table will expand the view allowing you to see all the fields and their properties within that table.



#### Viewing data within tables

You can browse the data within the database on the ‘Browse Data’ tab. You can then select a table underneath the tab to view. The application allows you to filter data and edit records.

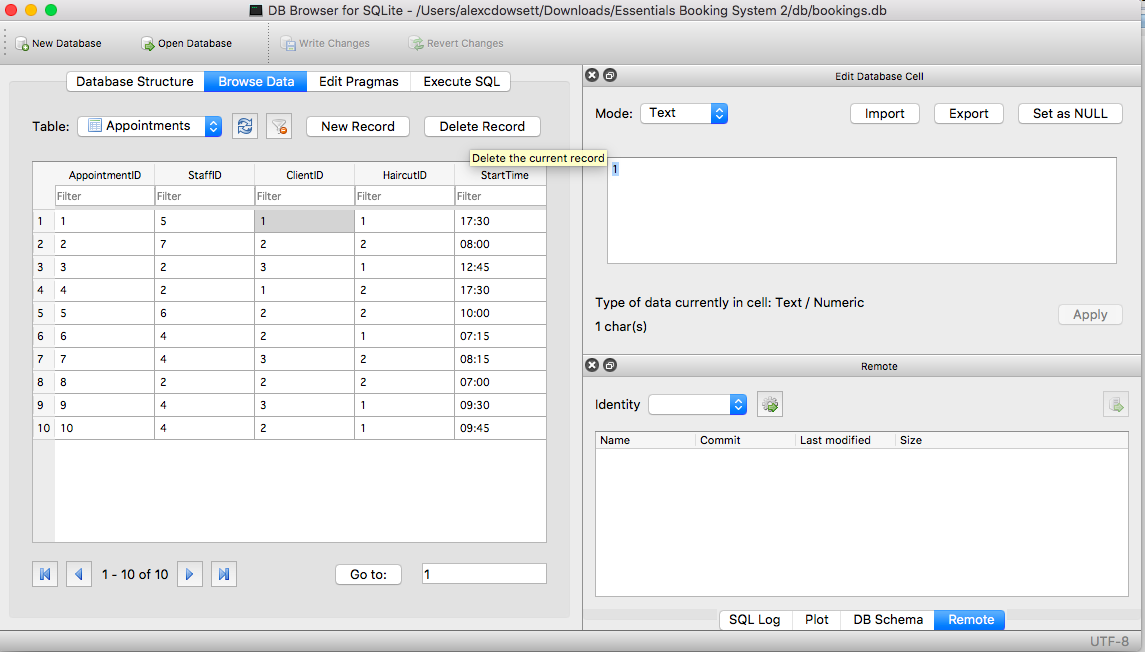


### Editing data within database

**NOTE:** Please do not edit the database without knowledge of what you are doing as you risk corrupting the file. Also always make a **backup** first.

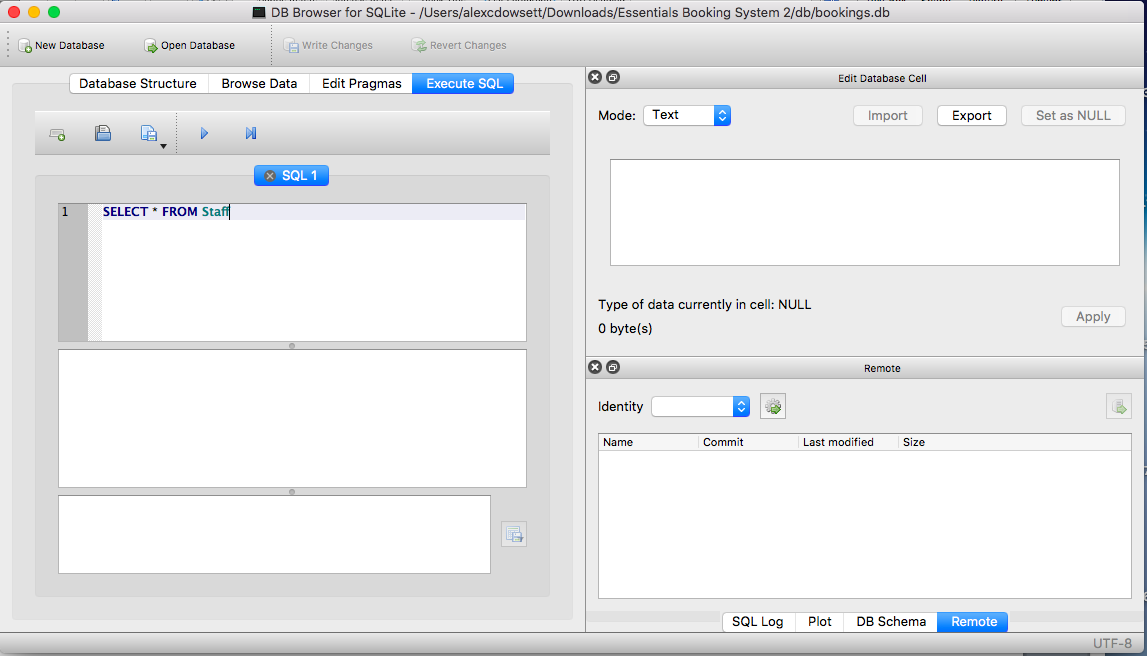
#### Editing data within tables

1. Click ‘Browse Data’
2. Double-click a cell that you wish to edit.
3. Edit the value in the text box on the right.



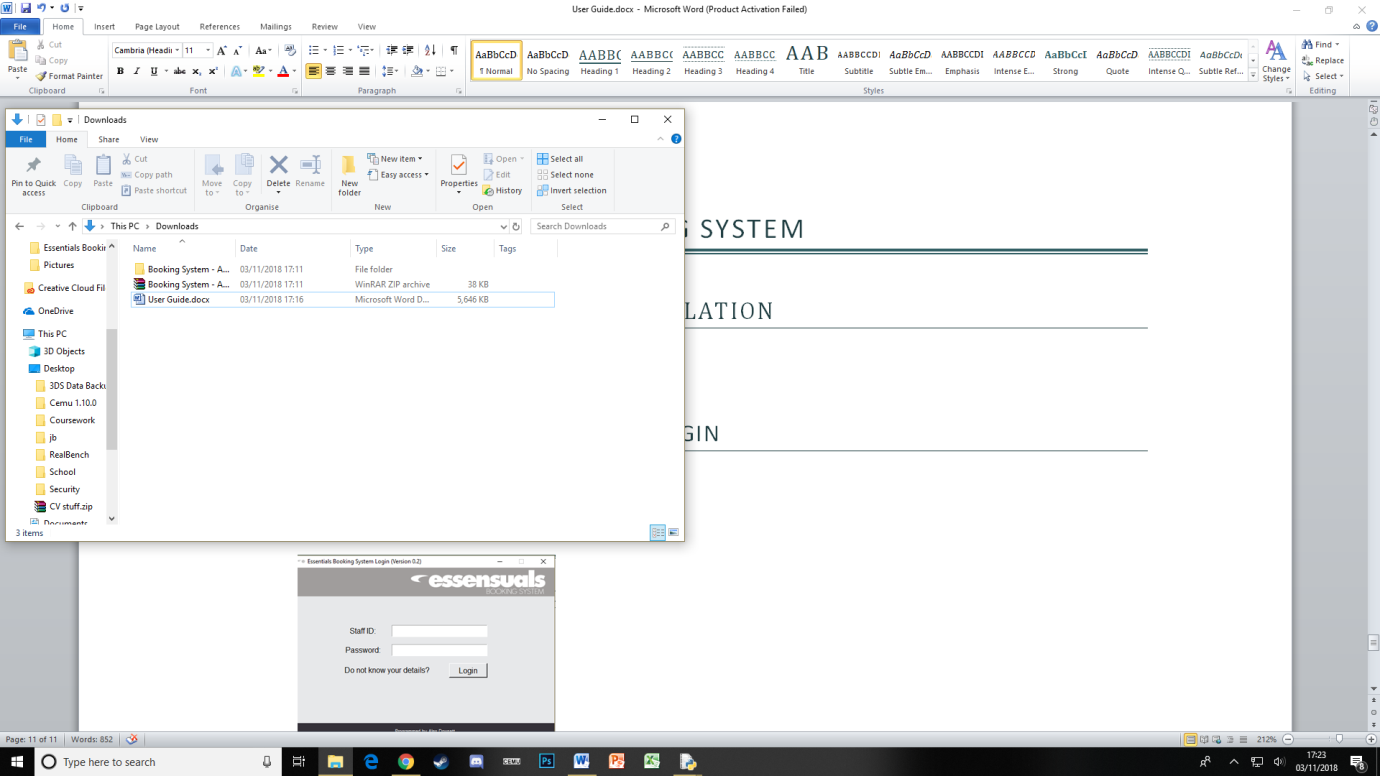
#### Editing data with SQL

1. Select the tab ‘Execute SQL’
2. Type SQL Statement(s) into the box below.
3. Execute the statements using the execute button.



## Booking system

### Installation



1. Extract the .zip file provided.

‘program.py’ is the file to be opened by Python. See ‘how to open/edit the program’ in the Python section of the user guide.

### Login

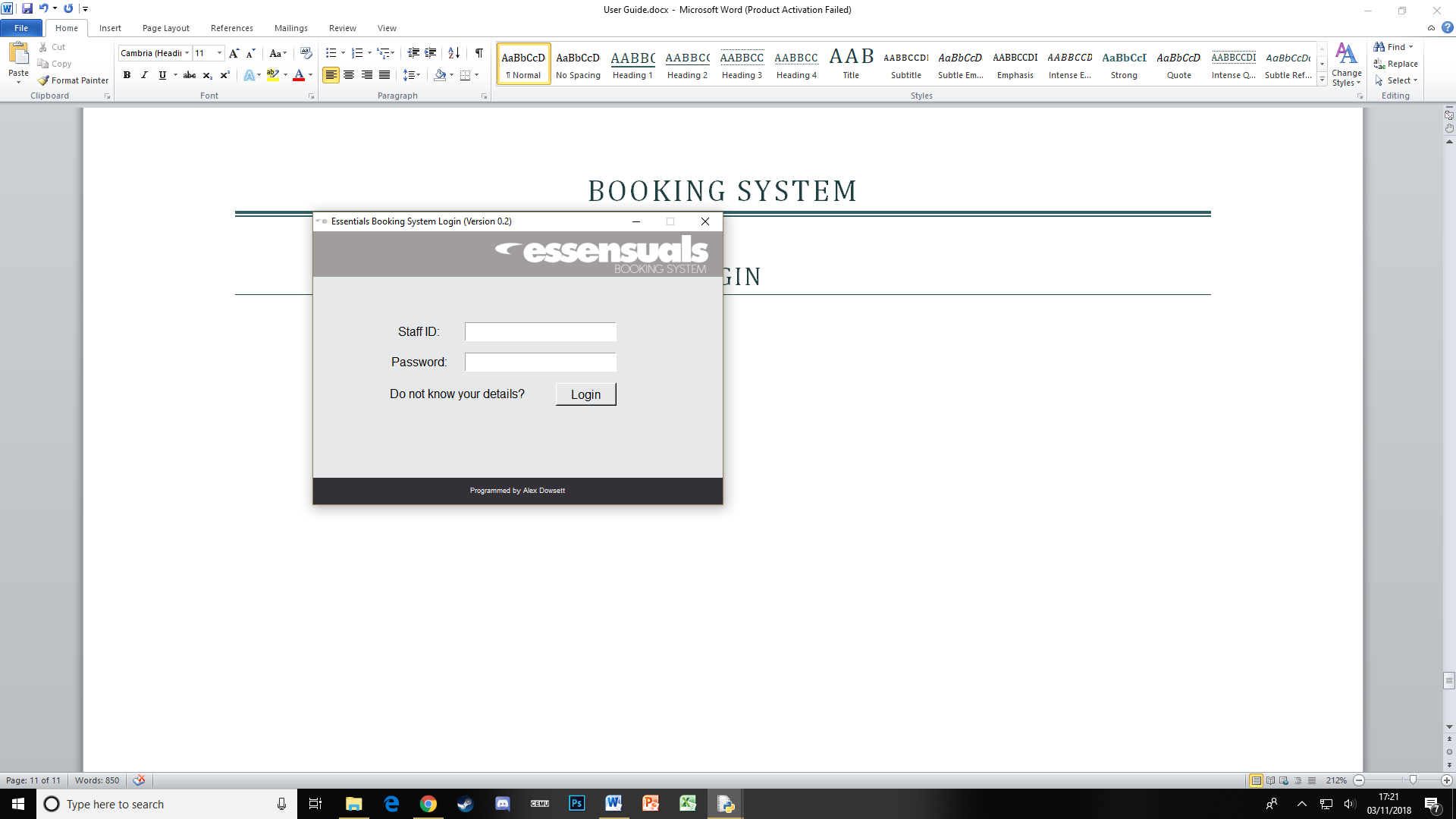
*If this is the first instance of running the program the default login details will be:*

**Staff ID: 1**

**Password: ess19**

*Please change the default password after login for security.*

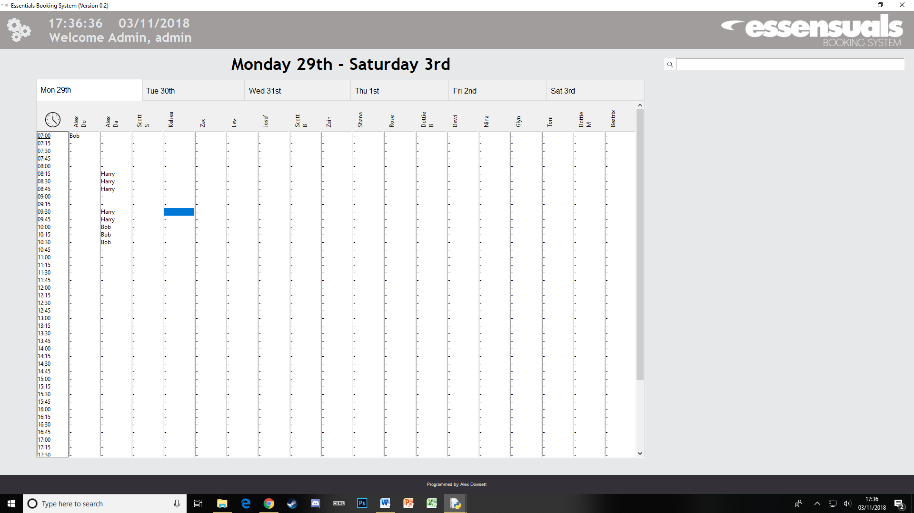
1. Enter your staff ID in the first entry box
2. Enter your password in the second entry box
3. Click ‘Login’ button or Enter key.



### Navigation

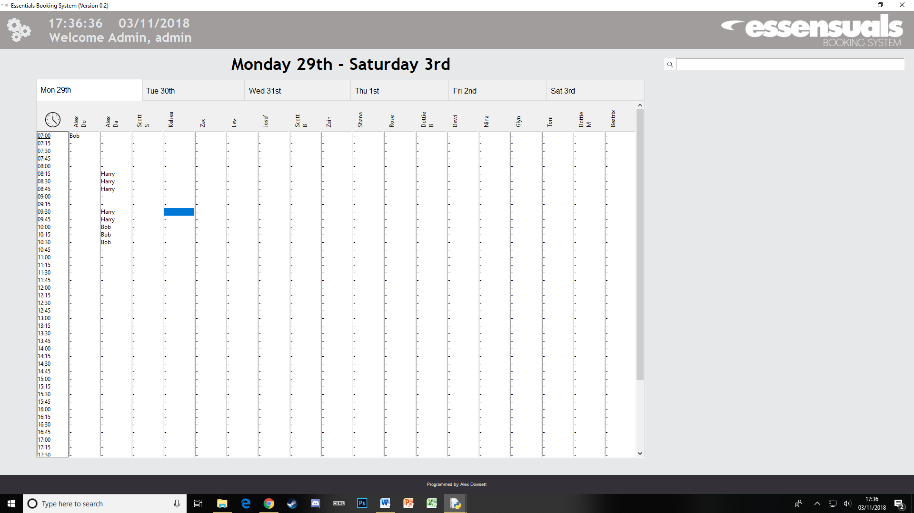
#### Changing the day

1. Click between the day of the week with the notebook tabs at the top of the table.

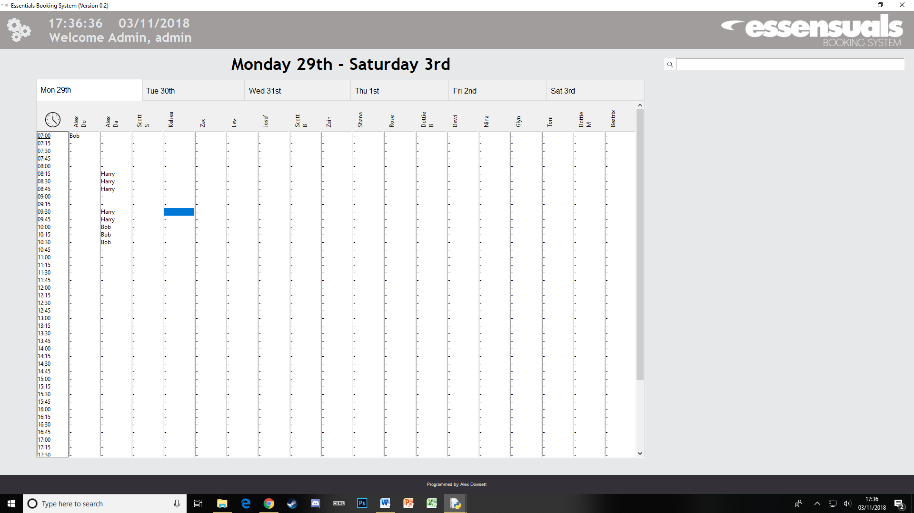
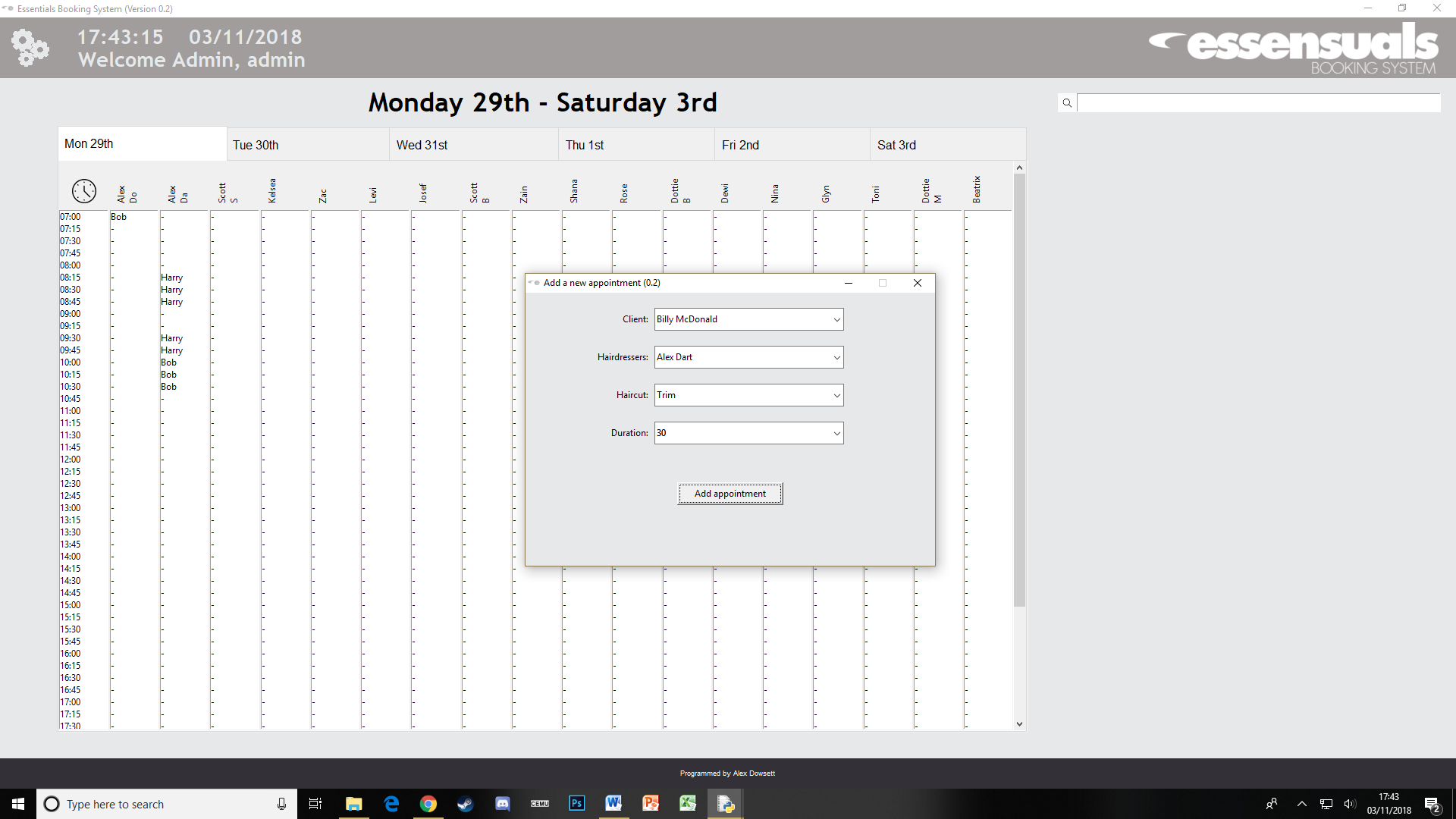


#### scrolling through appointments

1. Use your mouse wheel or the scrollbar on the right of the table to navigate through the appointments



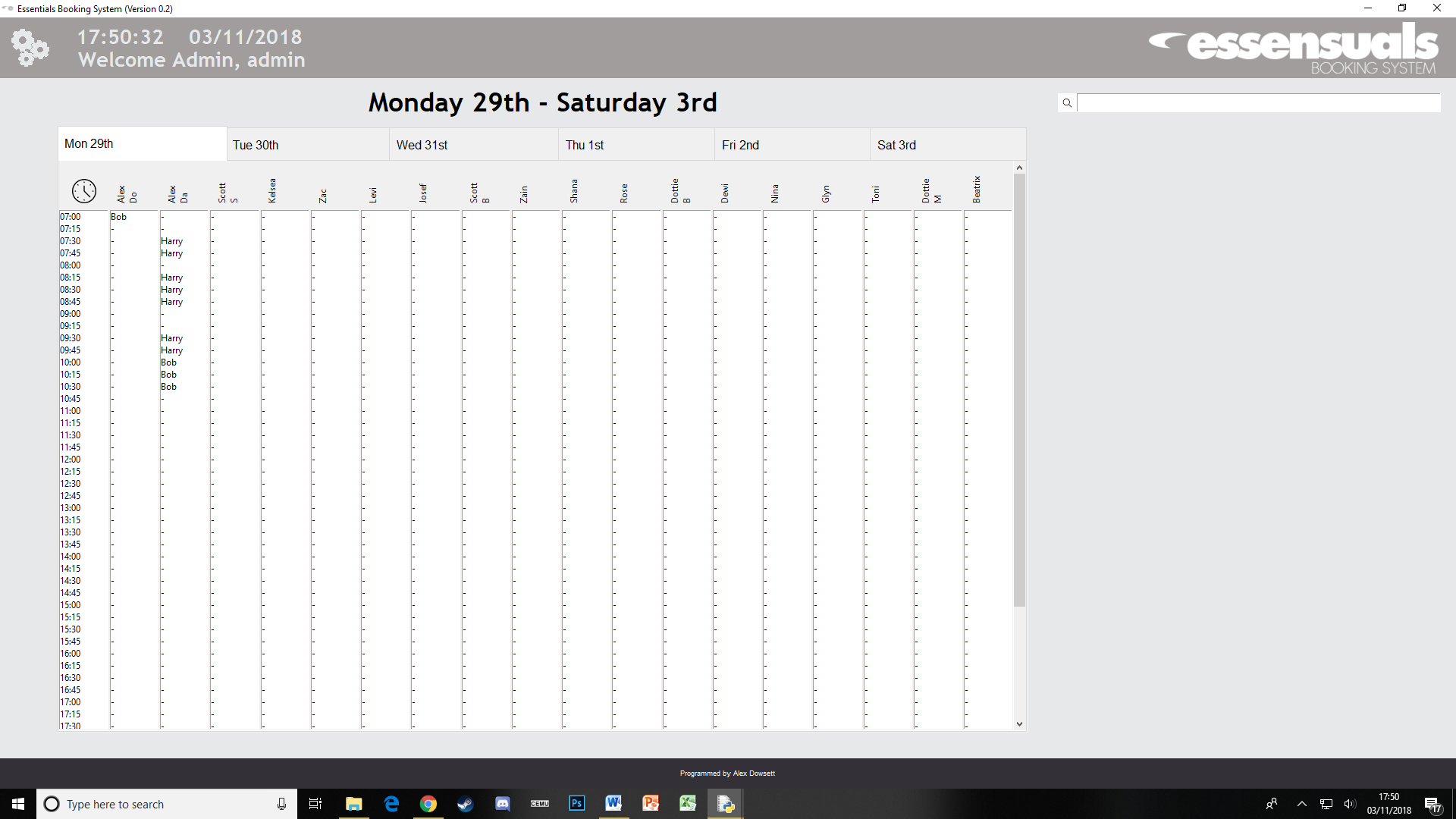
#### Adding appointments

1. Click on a cell that corresponds to the time and hairdresser you would like to book an appointment with
2. Enter the clients name, hairdresser, haircut and duration of the appointment.

#### View appointments

1. Clicking on an appointment slot that is already occupied will display the details of that appointment.

#### Closing the program

1. Just like any other program, clicking the top right ‘X’ will close the booking system.
2. You may need to close the Python shell (console) after.
3. You also may need to close the program.py IDLE window.

*If you accidently edit the program.py file and it asks to save after closing,* ***do not*** *save, to avoid corruption of the file.*

# Appendix

## Appendix Order

|  |  |
| --- | --- |
| Appendix A | Investigation – Observation |
| Appendix B | Investigation – Questionnaire |
| Appendix C | Investigation – Interview |
| Appendix D | Investigation – Existing booking system |
| Appendix E | Testing – Screenshots |

## Appendix A

Notes from observation:

* The pencil and book system is time-consuming. Lots of wasted time for staff member and client on phone.
* Messy book due to clients cancelling appointments.
* If clients called to find out when their appointment is it was extremely difficult for the staff member to find the appointment.
* Staff members sometimes could not read other staff member’s writing leading to the company appearing to be unorganised.
* It is difficult to fit client’s information in the book.

## Appendix B

With the current booking system in mind, please answer the following questions:

1. I feel comfortable with adding appointments to the book.

☐ Strongly disagree ☐ Disagree ☐ Agree ☐ Strongly agree

1. I can easily navigate through the book to find dates.

☐ Strongly disagree ☐ Disagree ☐ Agree ☐ Strongly agree

1. I can easily remind clients what date they have booked their appointments.

☐ Strongly disagree ☐ Disagree ☐ Agree ☐ Strongly agree

1. I can easily remove and edit appointments to the book.

☐ Strongly disagree ☐ Disagree ☐ Agree ☐ Strongly agree

1. I feel comfortable with calculating prices for haircuts.

☐ Strongly disagree ☐ Disagree ☐ Agree ☐ Strongly agree

1. I favour a computerised booking system over a book and pen.

☐ Strongly disagree ☐ Disagree ☐ Agree ☐ Strongly agree

*Please circle a number on the scale below*

*where 1 is extremely unhappy and 10 is extremely happy.*

1. Overall, how happy are you with using a book to record bookings.

1 2 3 4 5 6 7 8 9 10

**Thank you.**

## Appendix C

1. **Do you feel there would be any potential limitations with a computerised booking system?**

* Some people automatically dislike the idea of using a computer.
* Reliance on technology.

1. **What features that are not in the current booking system would you like to see in the new one?**

* Calculation of price.
* Much easier to navigate and find appointments with a search bar.
* Consistent formatting.
* Bigger record, a computer database is more timeless than booking books.
* Easier cancellation and rebooking in same slot.
* A secure system.

1. **What is one of the biggest problems you have with the current booking system and how could this be improved with a computerised system.**

* Cancelling multiple appointments can become messy and confusing.
* Also finding client’s lost appointments times is extremely time-consuming and can take over minutes to find their appointment.
* Can be improved by allowing searching client names with a search bar.

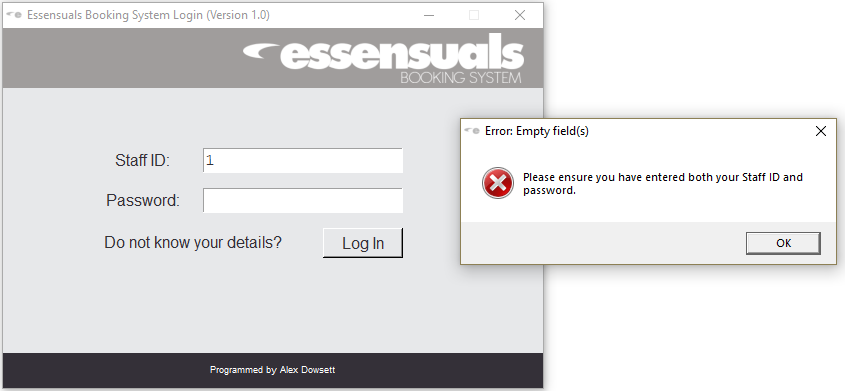
## Appendix D

The company used a very similar daily appointments book to the one below, with client names on the line with the corresponding time. Next to the client’s name was the hairdresser’s initials to save on space.

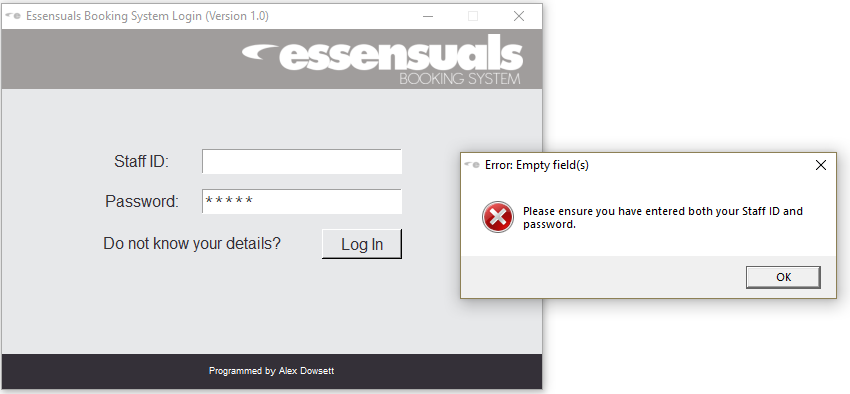


## Appendix E

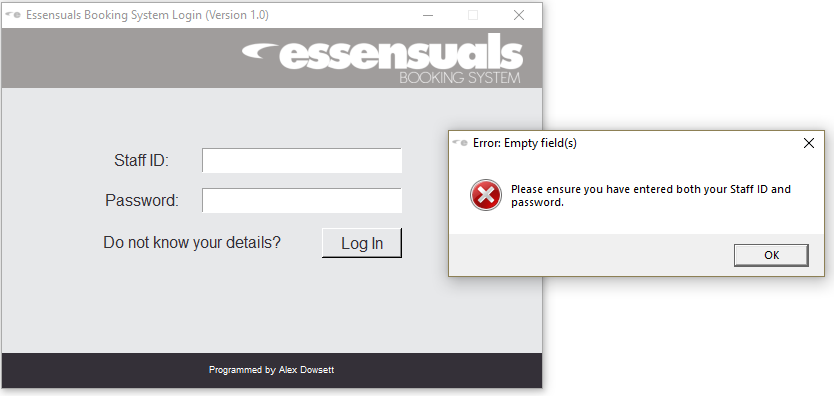
#### Figure 1a



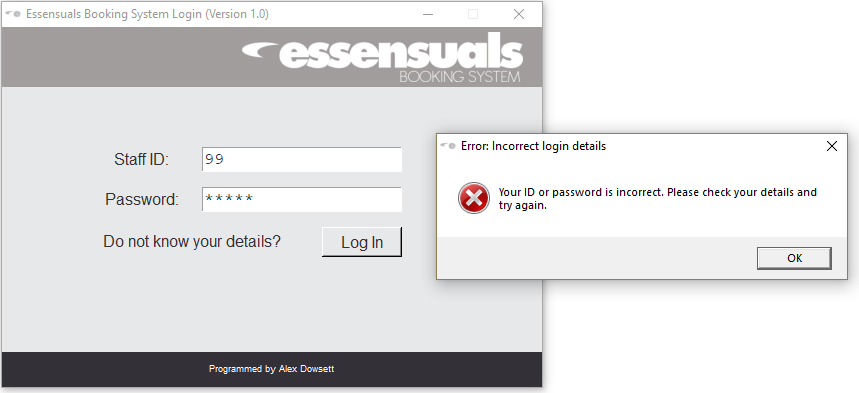
#### Figure 1b



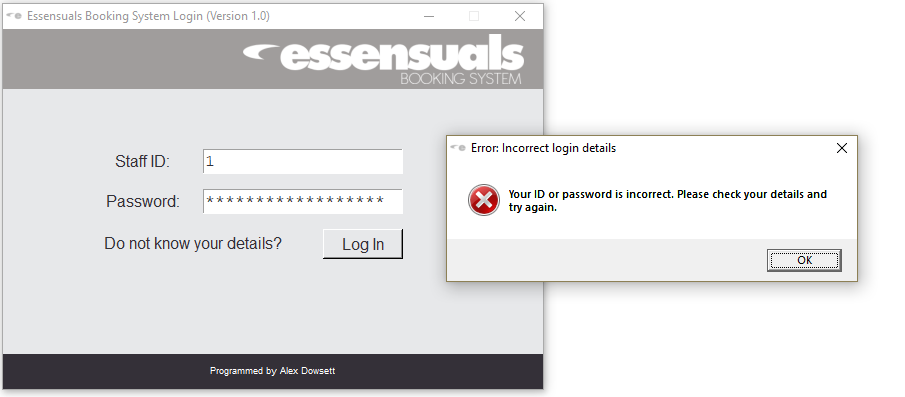
#### Figure 1c



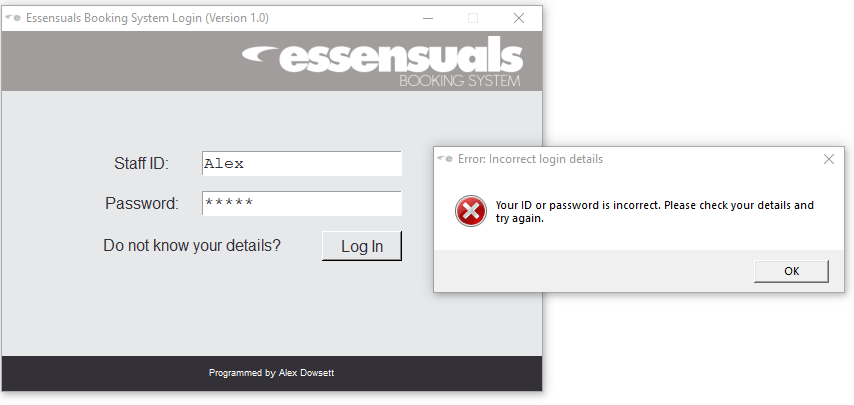
#### Figure 2



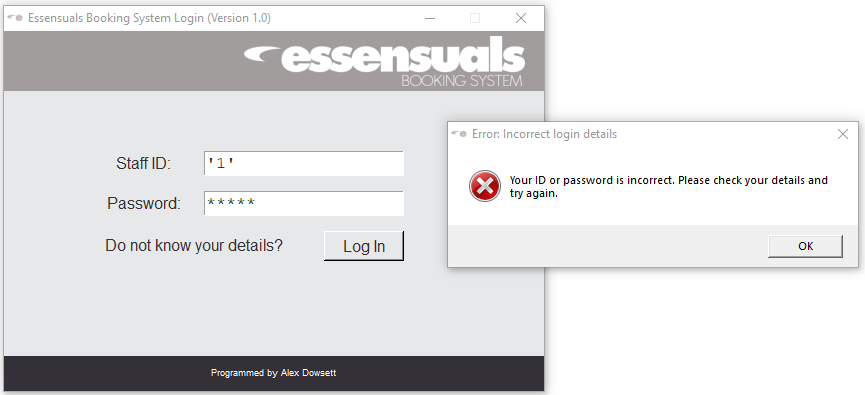
#### Figure 3

****

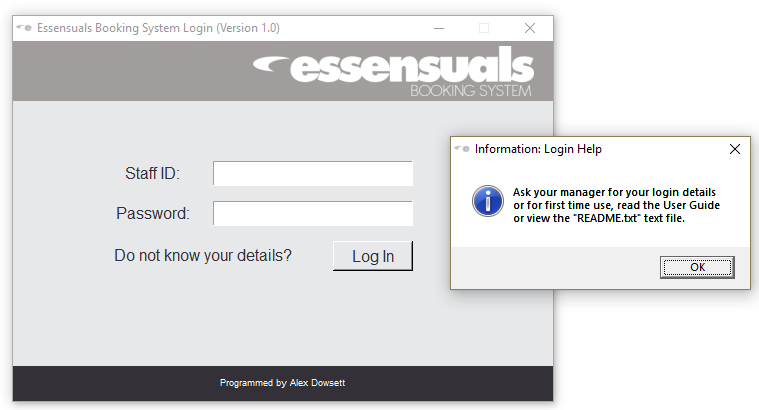
#### Figure 4a

****

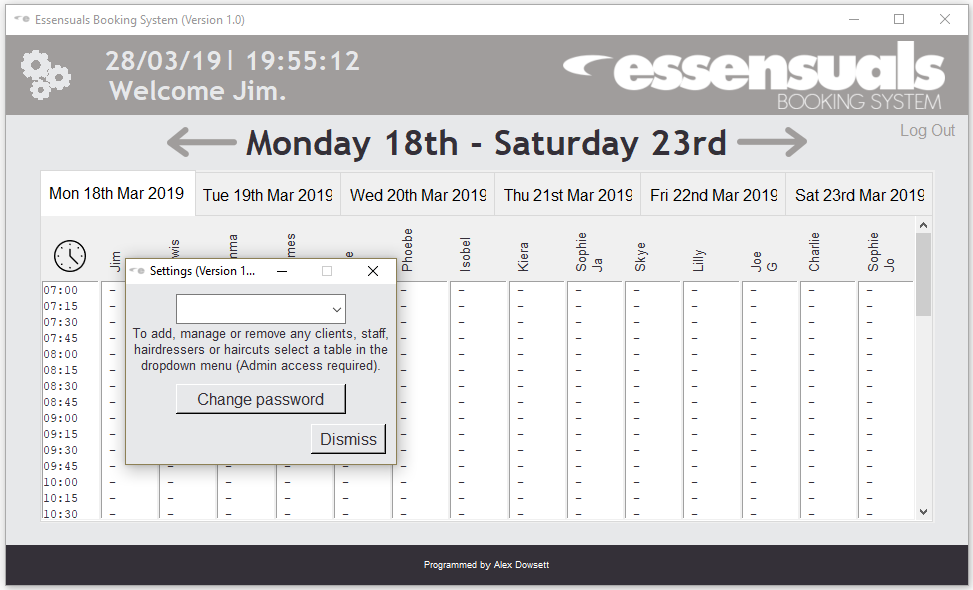
#### Figure 4b

****

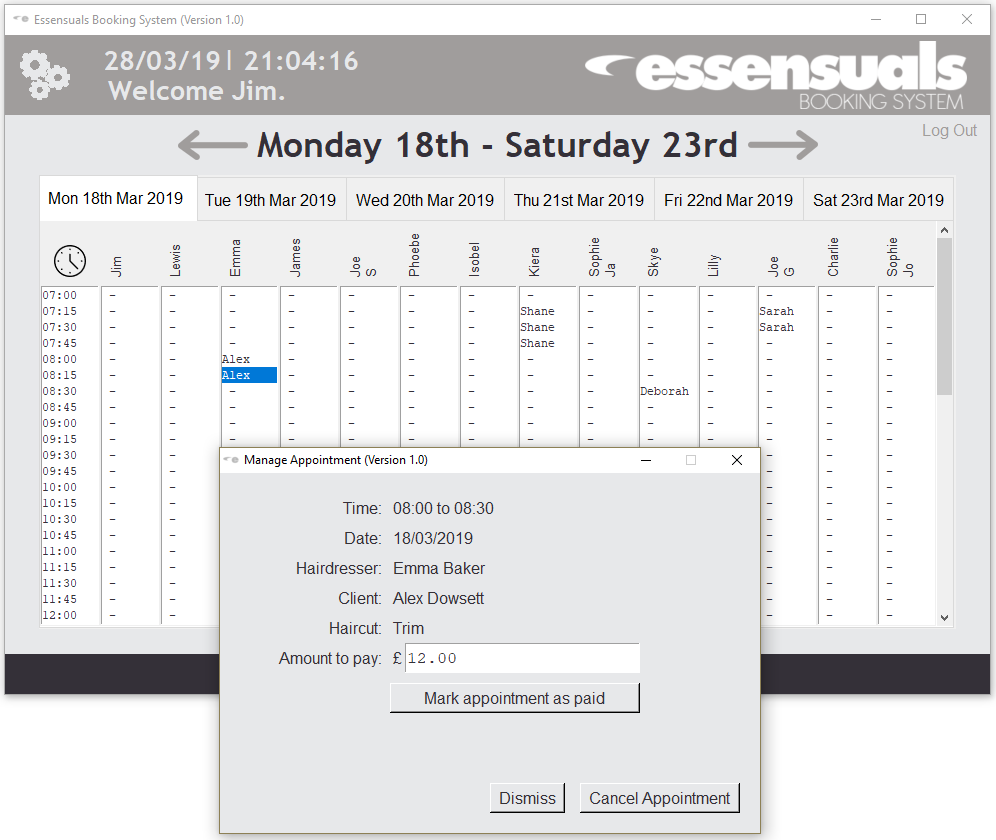
#### Figure 5a



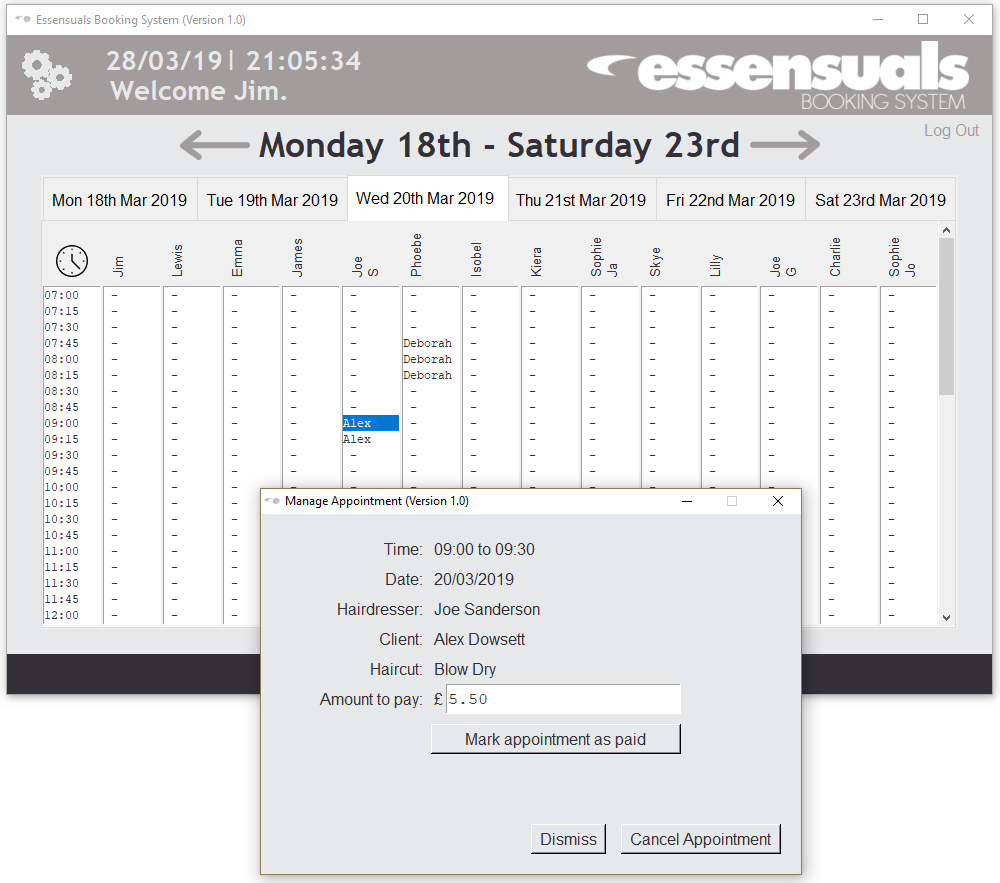
#### Figure 5b



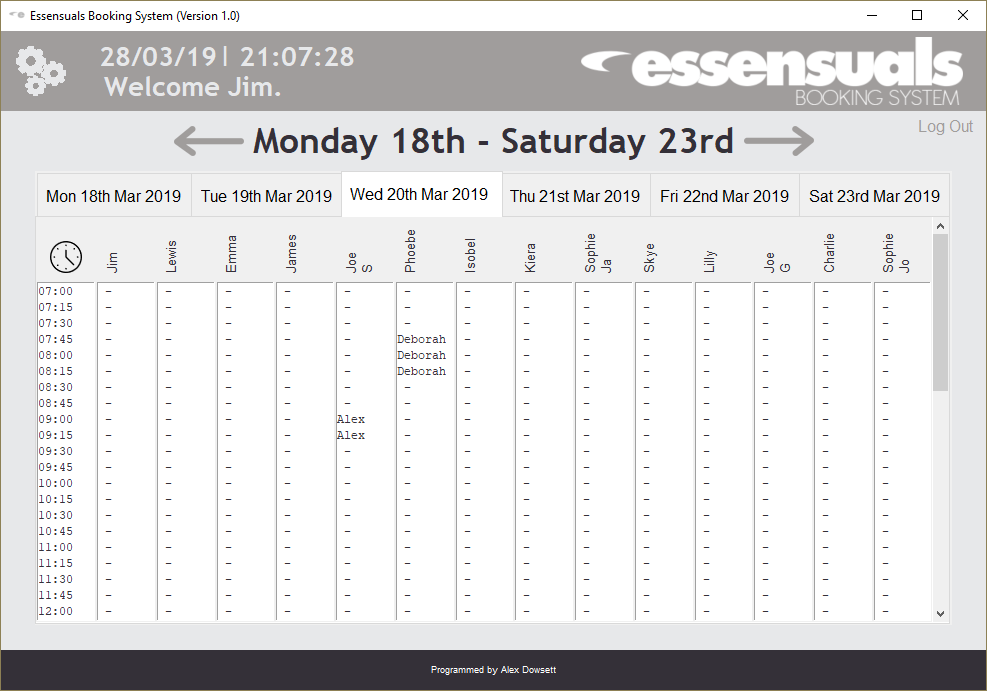
#### Figure 6a



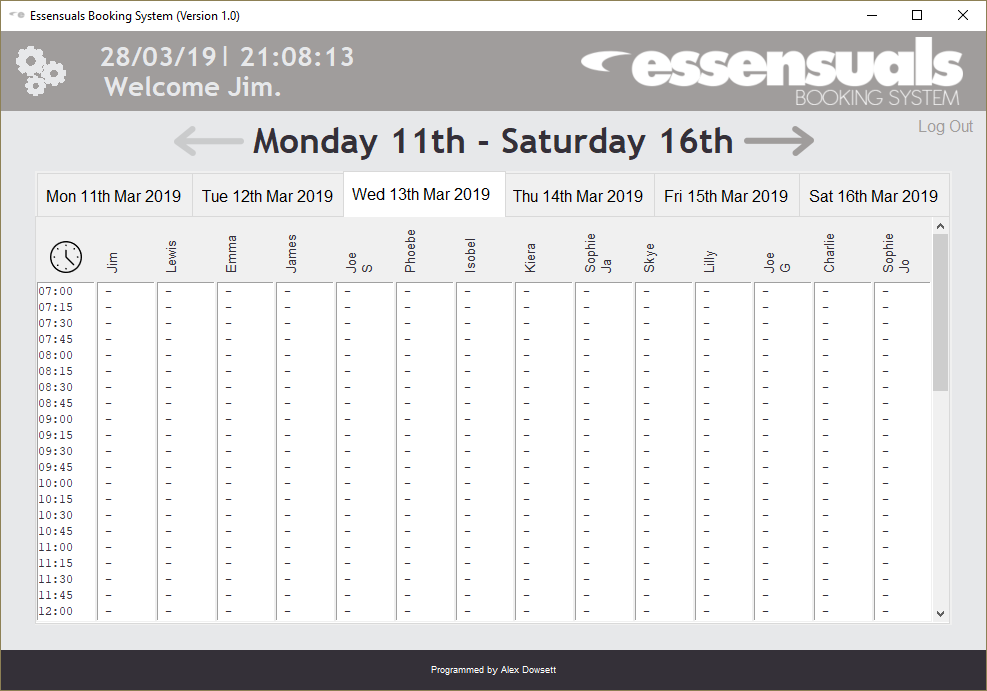
#### Figure 6b



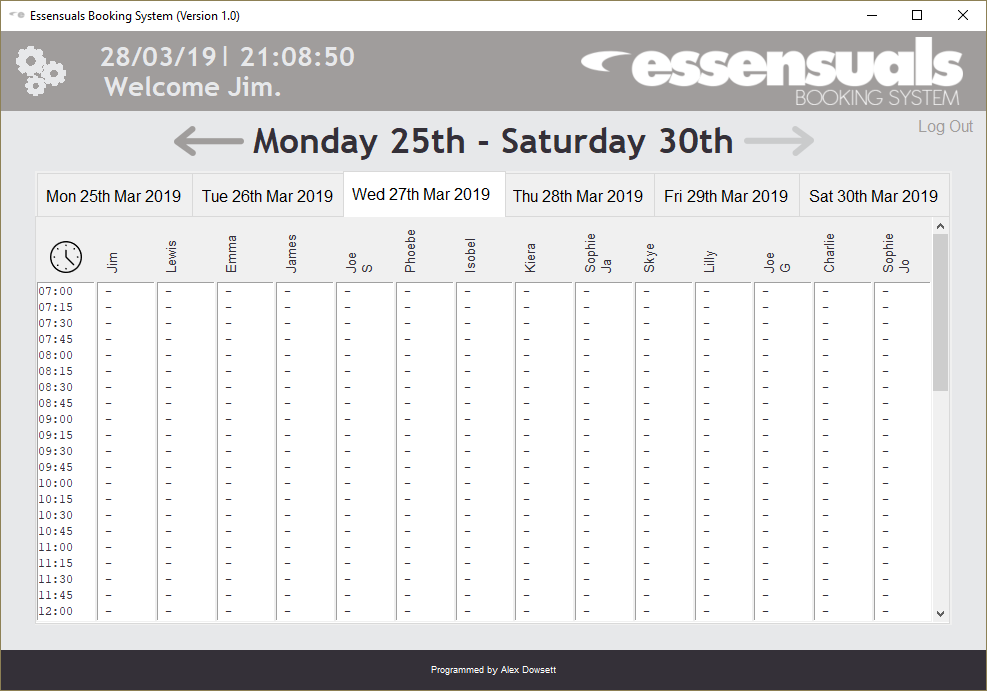
#### Figure 7a



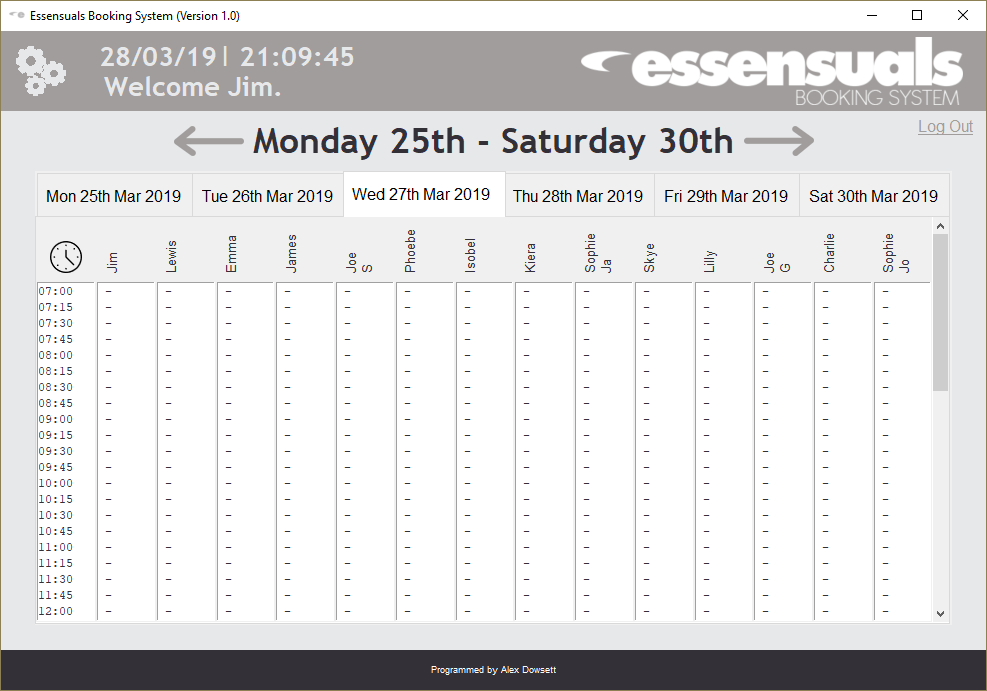
#### Figure 7b



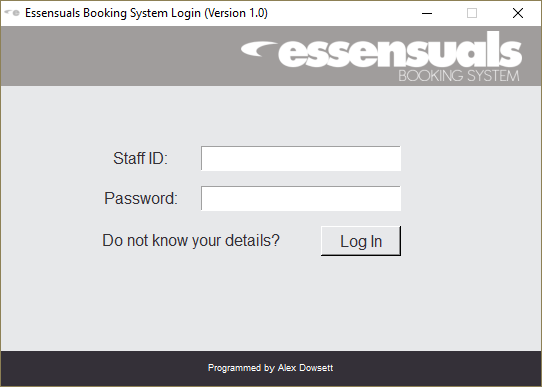
#### Figure 7c



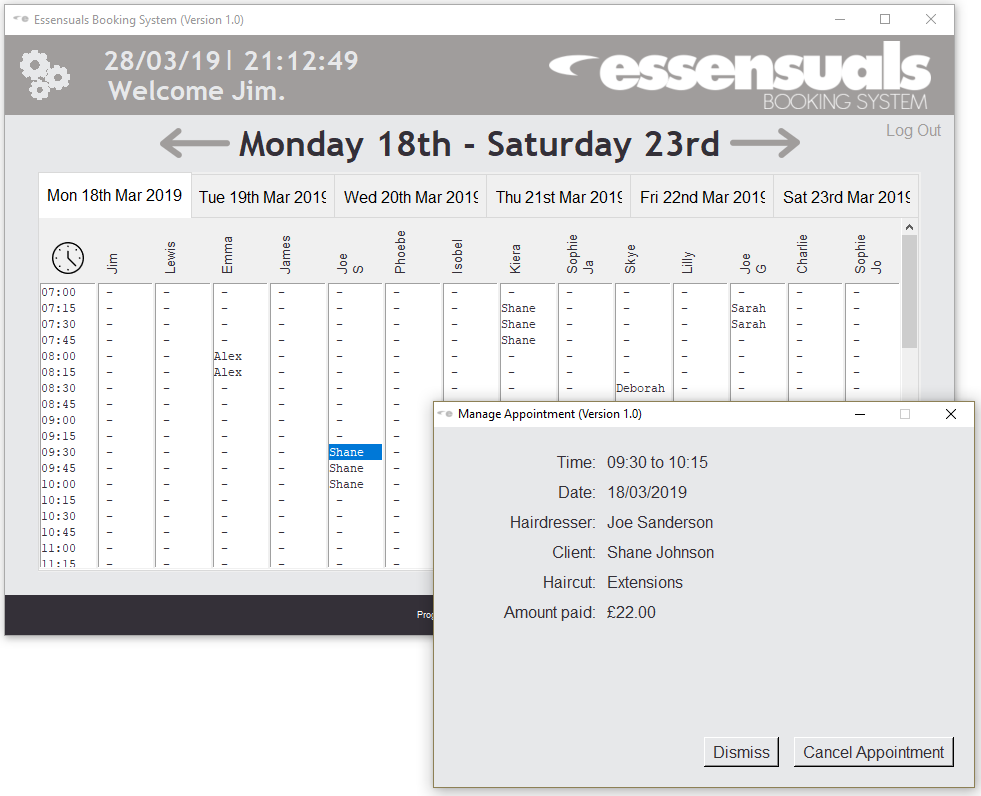
#### Figure 8a



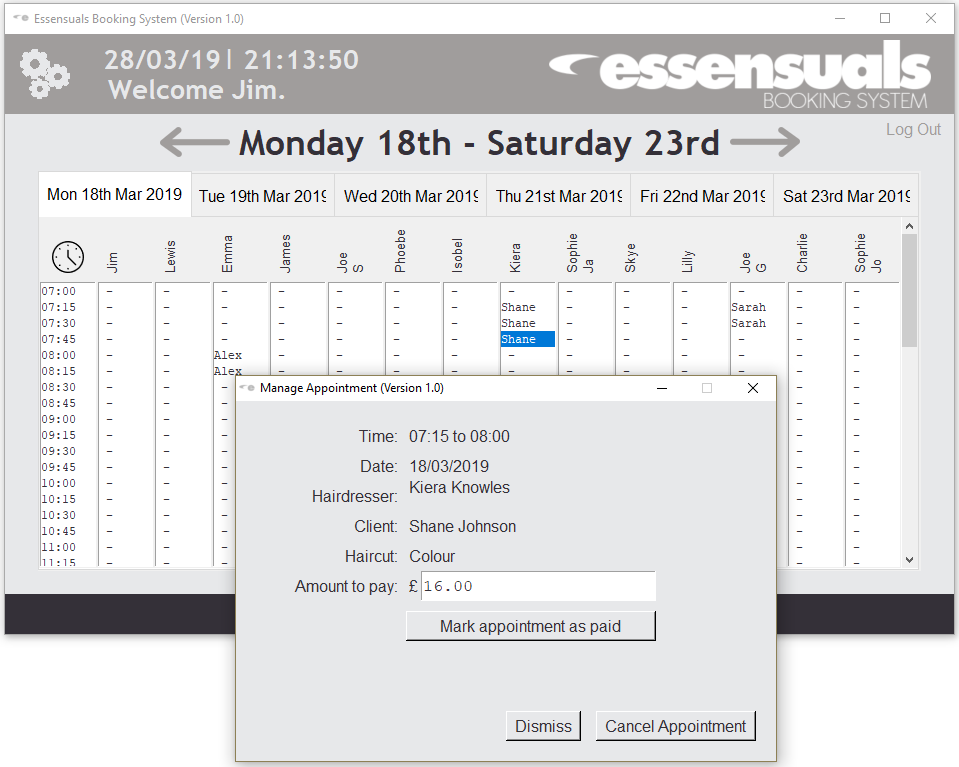
#### Figure 8b



#### Figure 9



#### Figure 10a



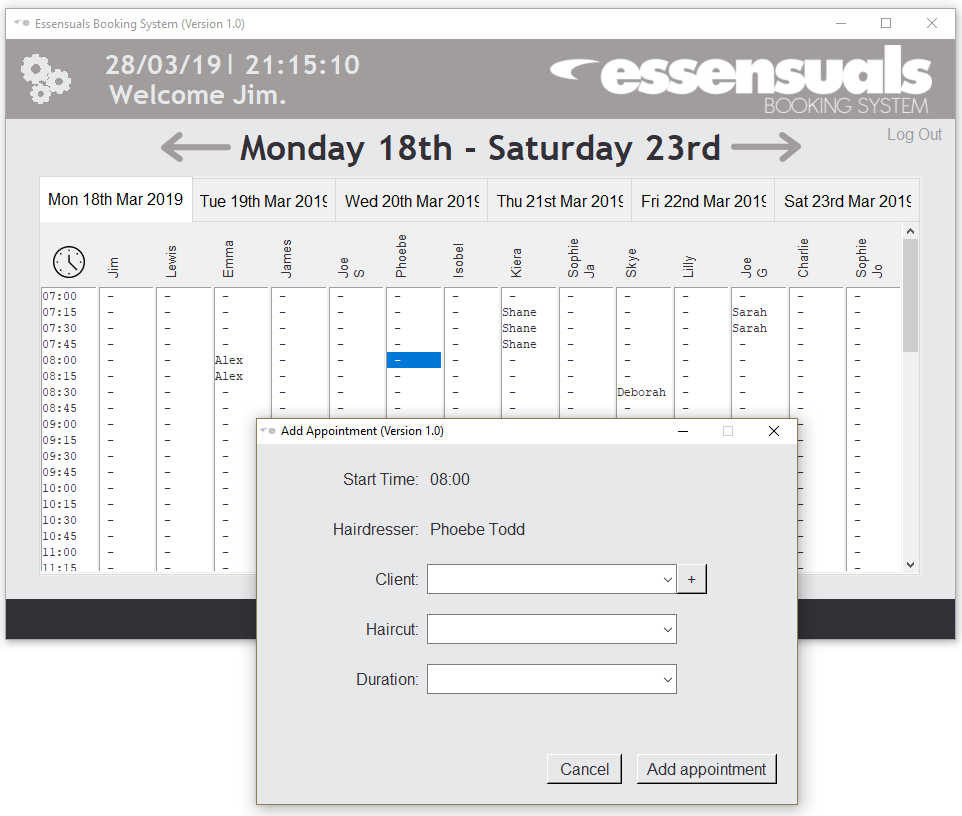
#### Figure 10b



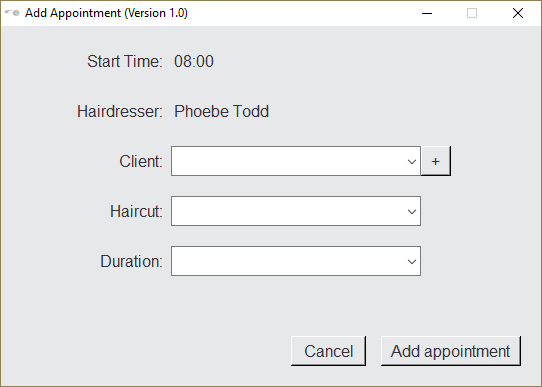
#### Figure 10c



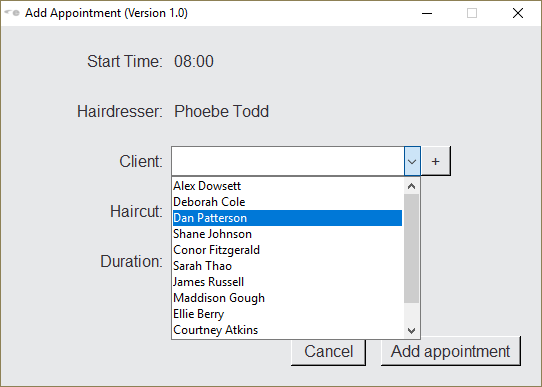
#### Figure 11



#### Figure 12a



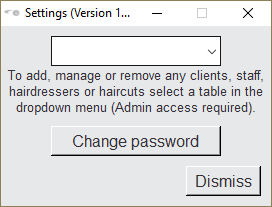
#### Figure 12b



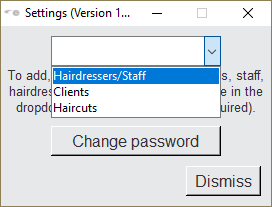
#### Figure 12c



#### Figure 12d



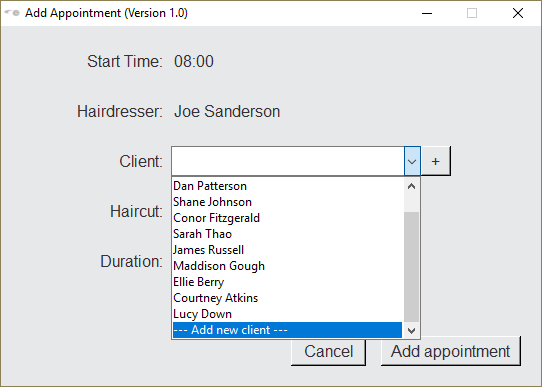
#### Figure 12e



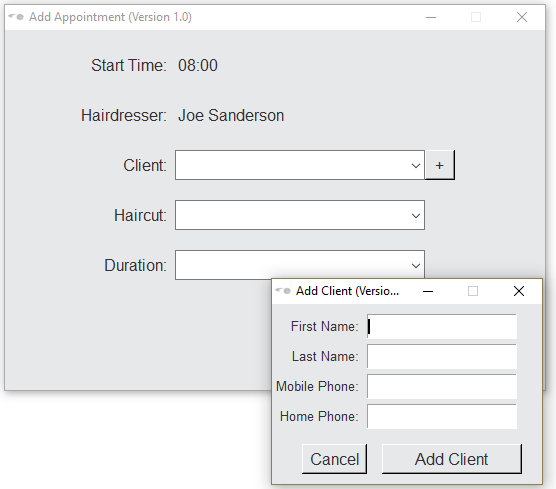
#### Figure 13a



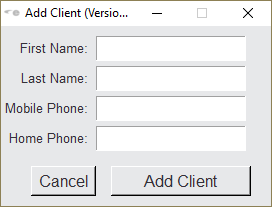
#### Figure 13b



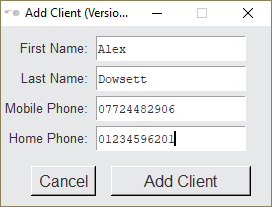
#### Figure 13c



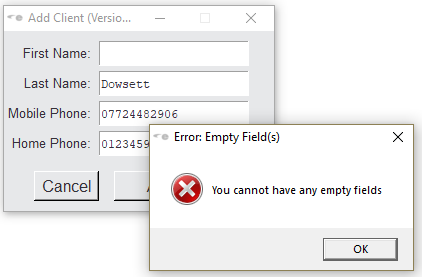
#### Figure 14a



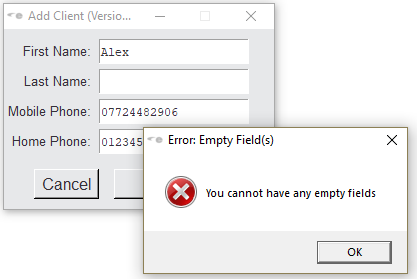
#### Figure 14b



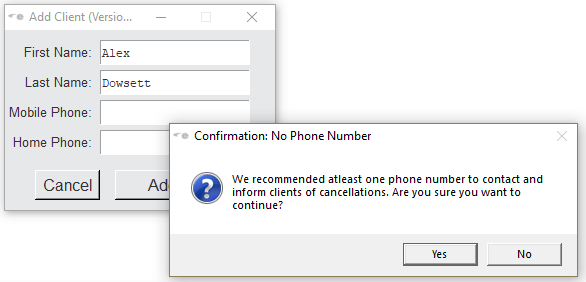
#### Figure 15a



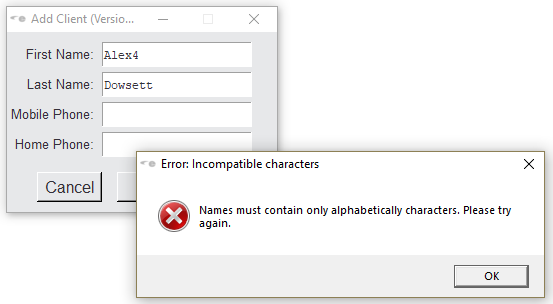
#### Figure 15b



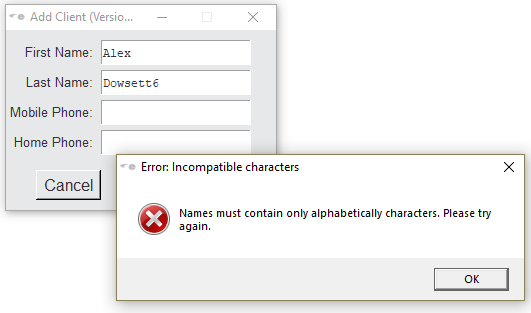
#### Figure 15c



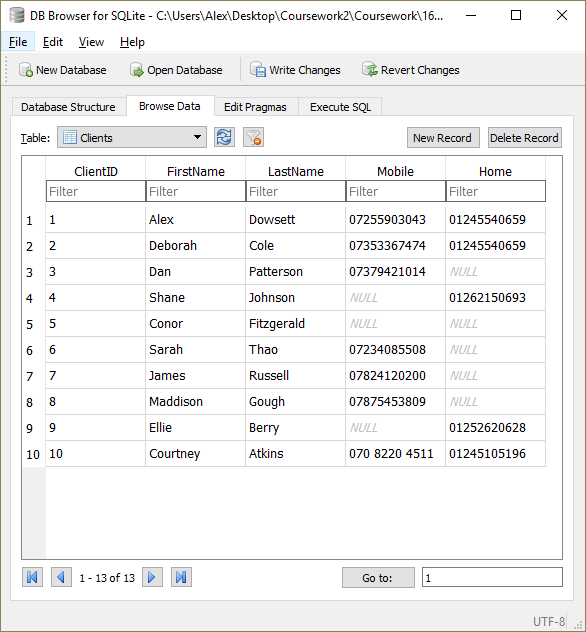
#### Figure 16a



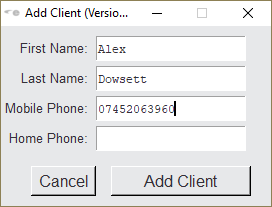
#### Figure 16b



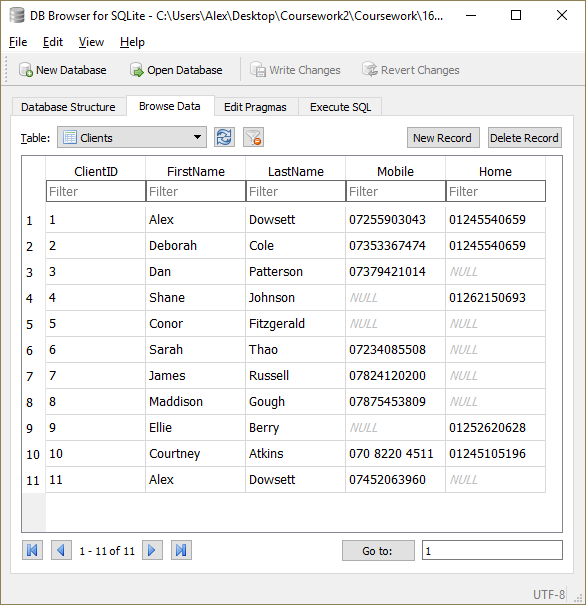
#### Figure 17a



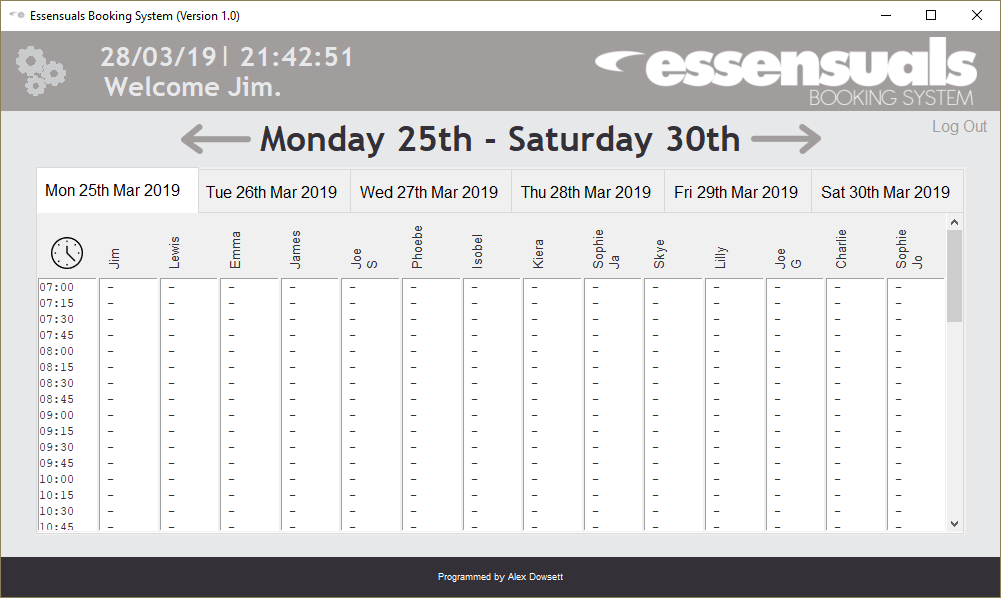
#### Figure 17b



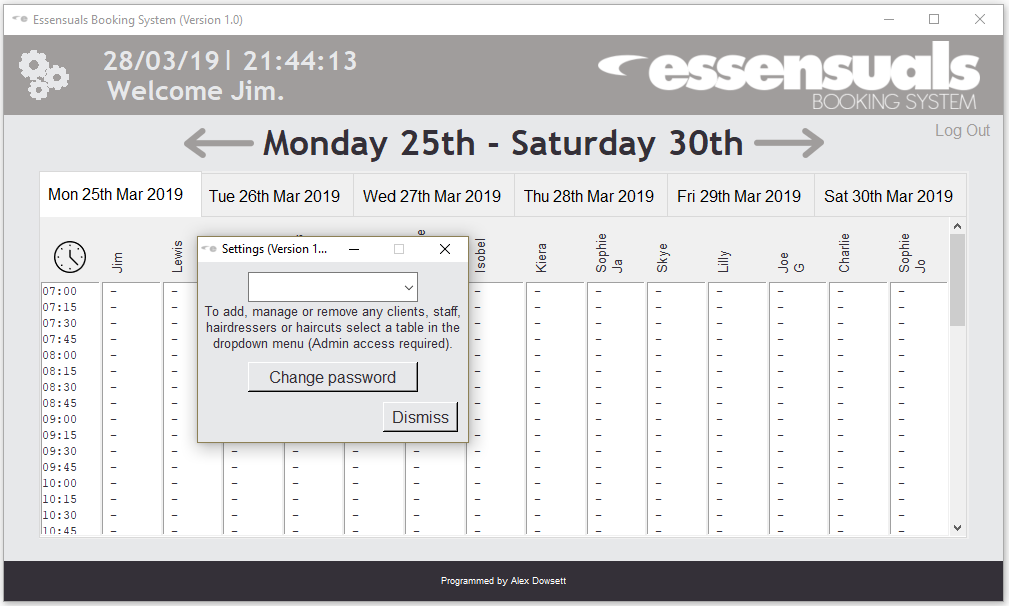
#### Figure 17c



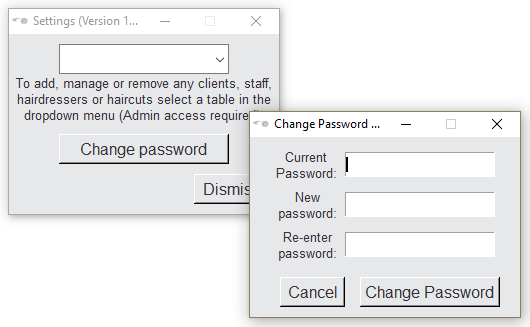
#### Figure 18a



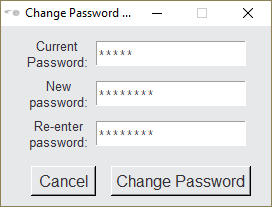
#### Figure 18b



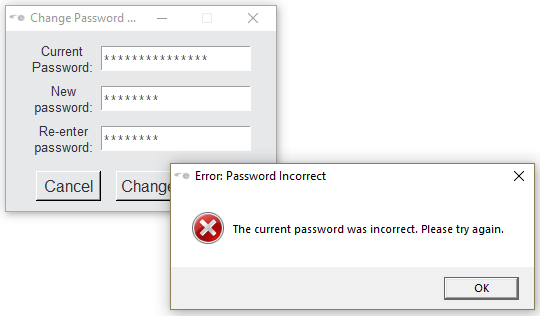
#### Figure 19



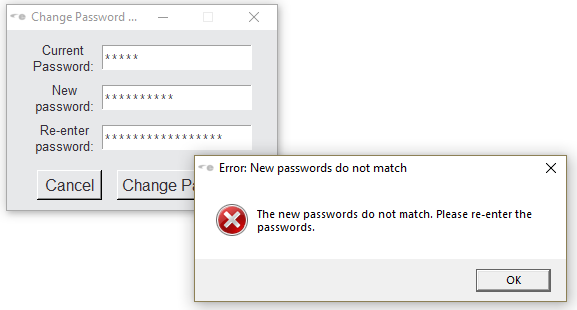
#### Figure 20



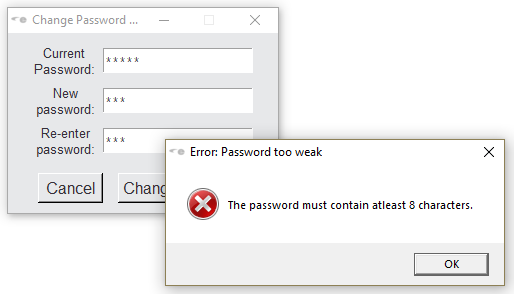
#### Figure 21



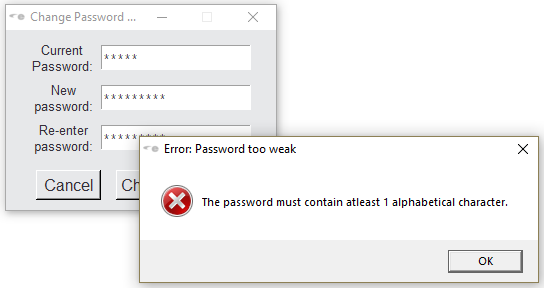
#### Figure 22



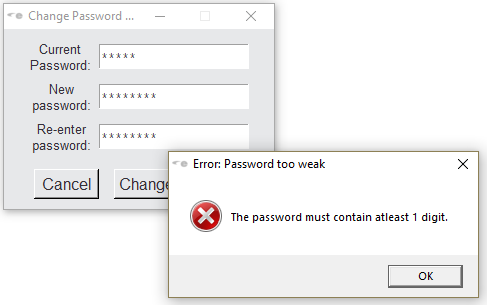
#### Figure 23a



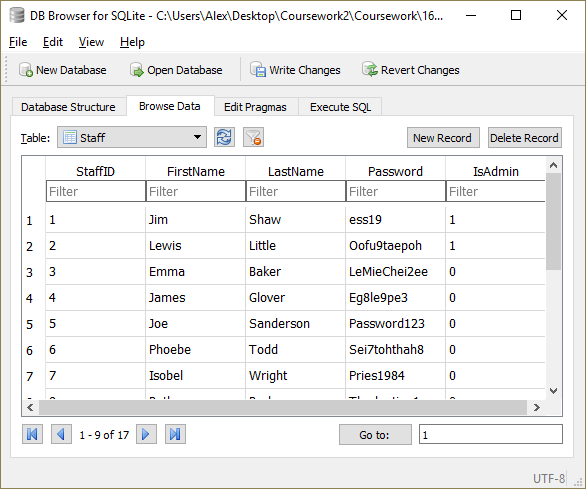
#### Figure 23b



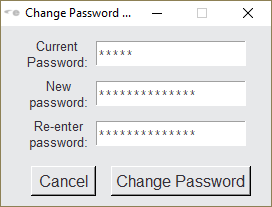
#### Figure 23c



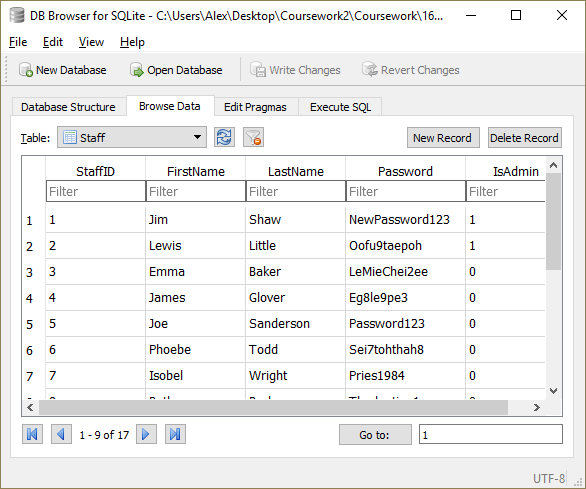
#### Figure 24a



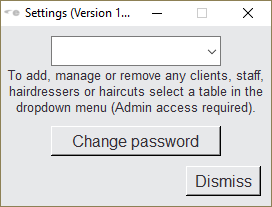
#### Figure 24b



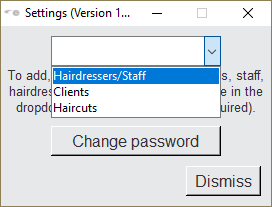
#### Figure 24c



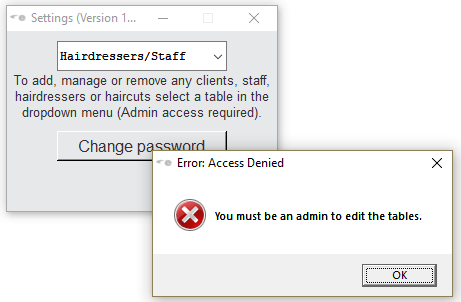
#### Figure 25a



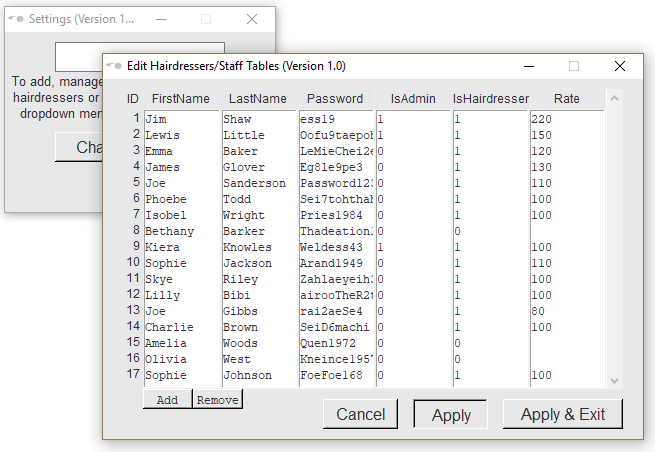
#### Figure 25b



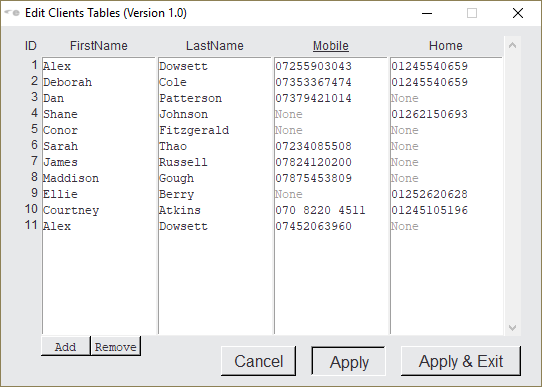
#### Figure 25c



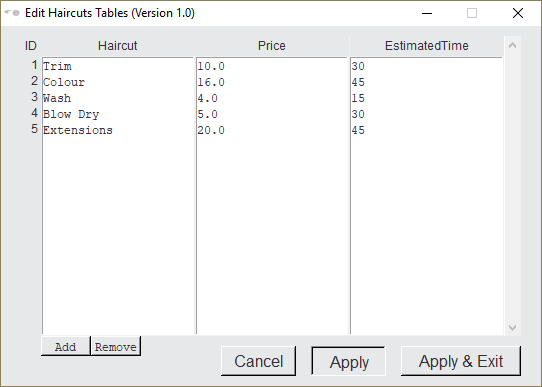
#### Figure 25d



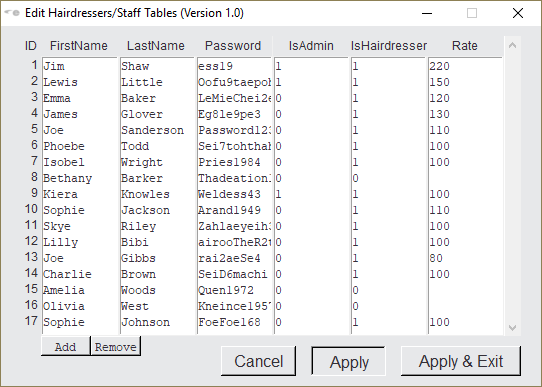
#### Figure 25e



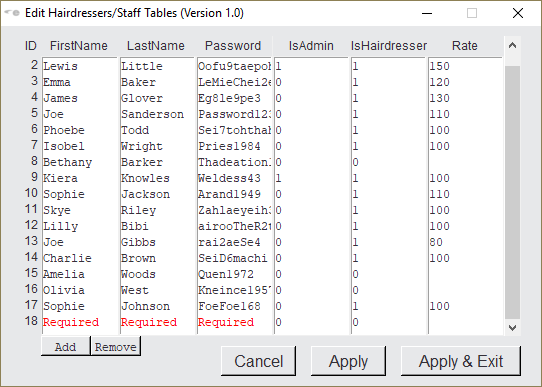
#### Figure 25f



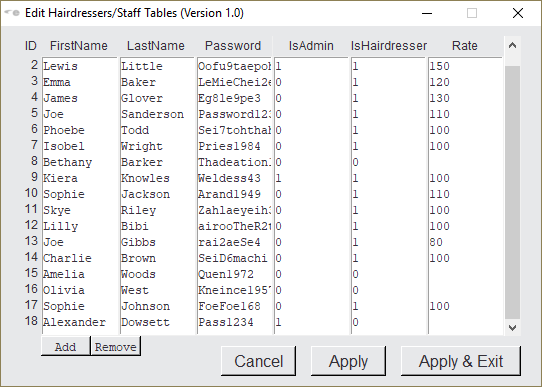
#### Figure 26a



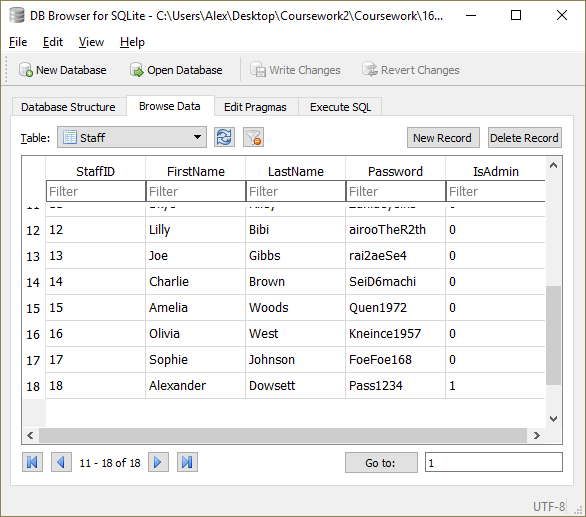
#### Figure 26b



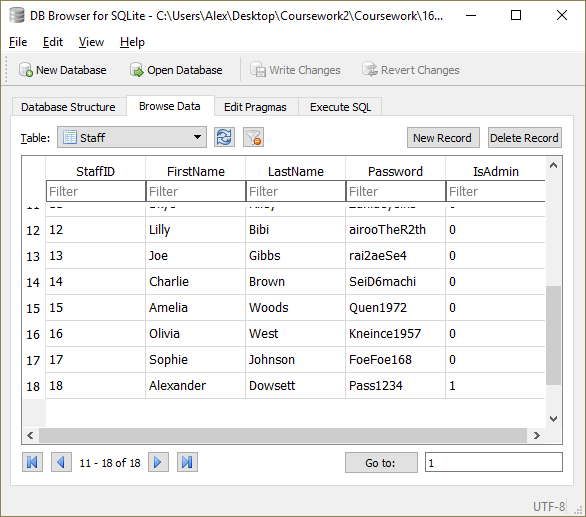
#### Figure 26c



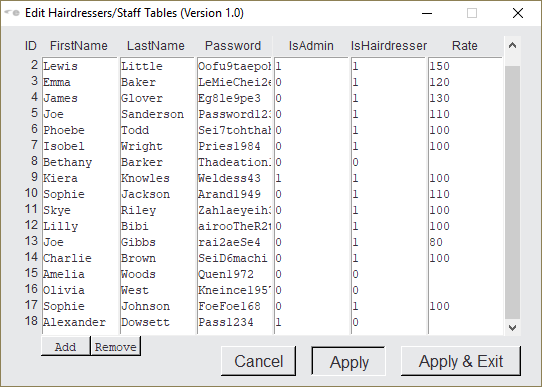
#### Figure 26d



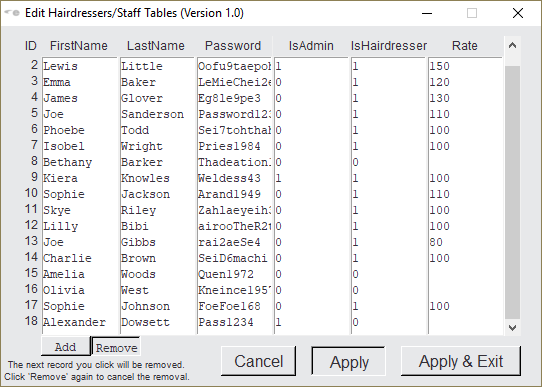
#### Figure 27a



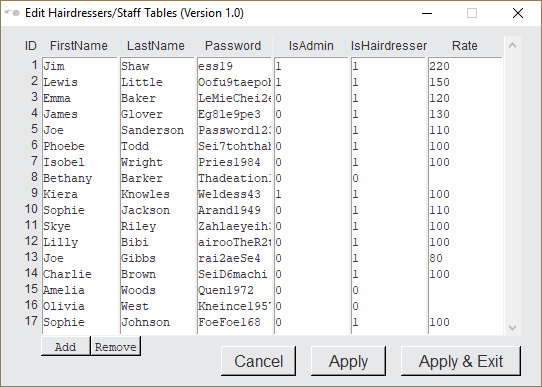
#### Figure 27b



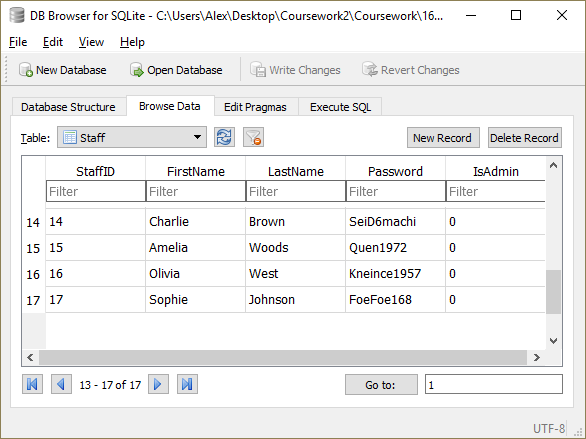
#### Figure 27c



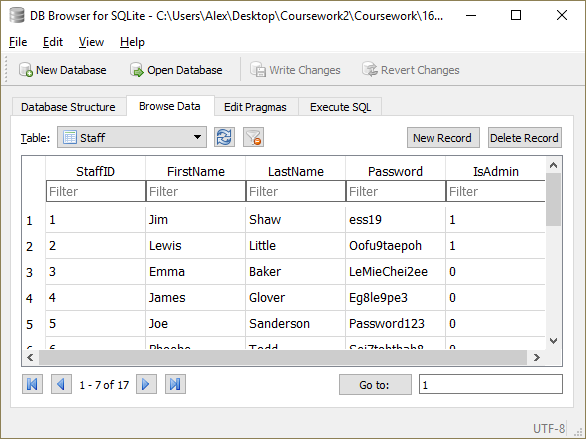
#### Figure 27d



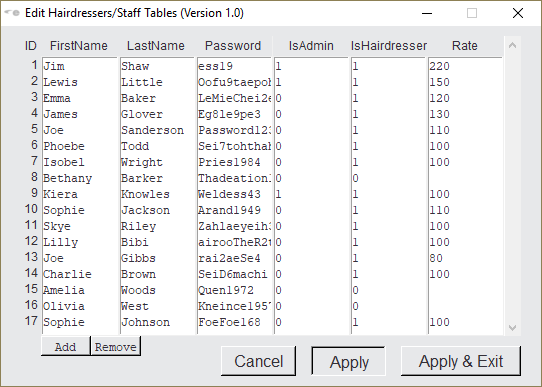
#### Figure 27e



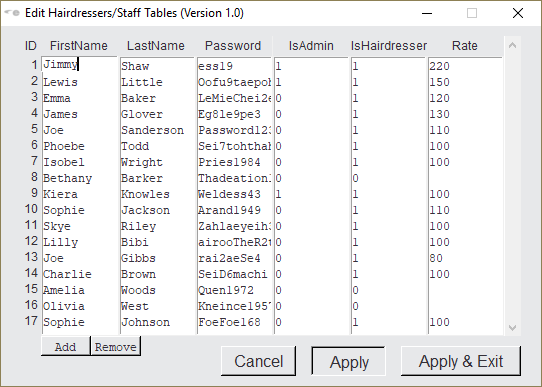
#### Figure 28a



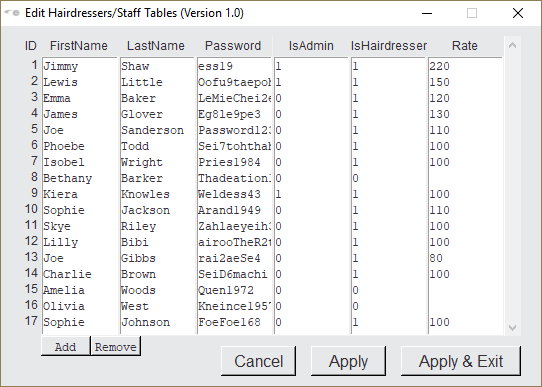
#### Figure 28b



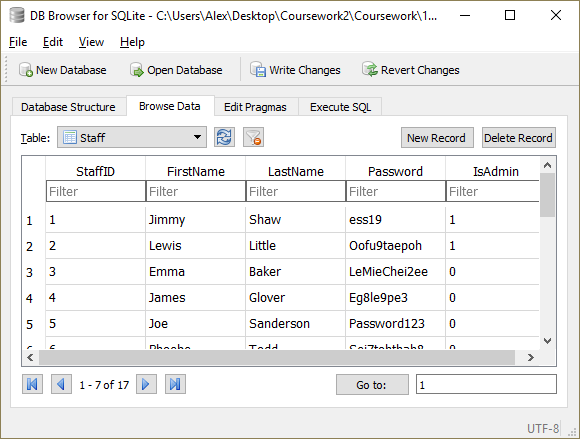
#### Figure 28c



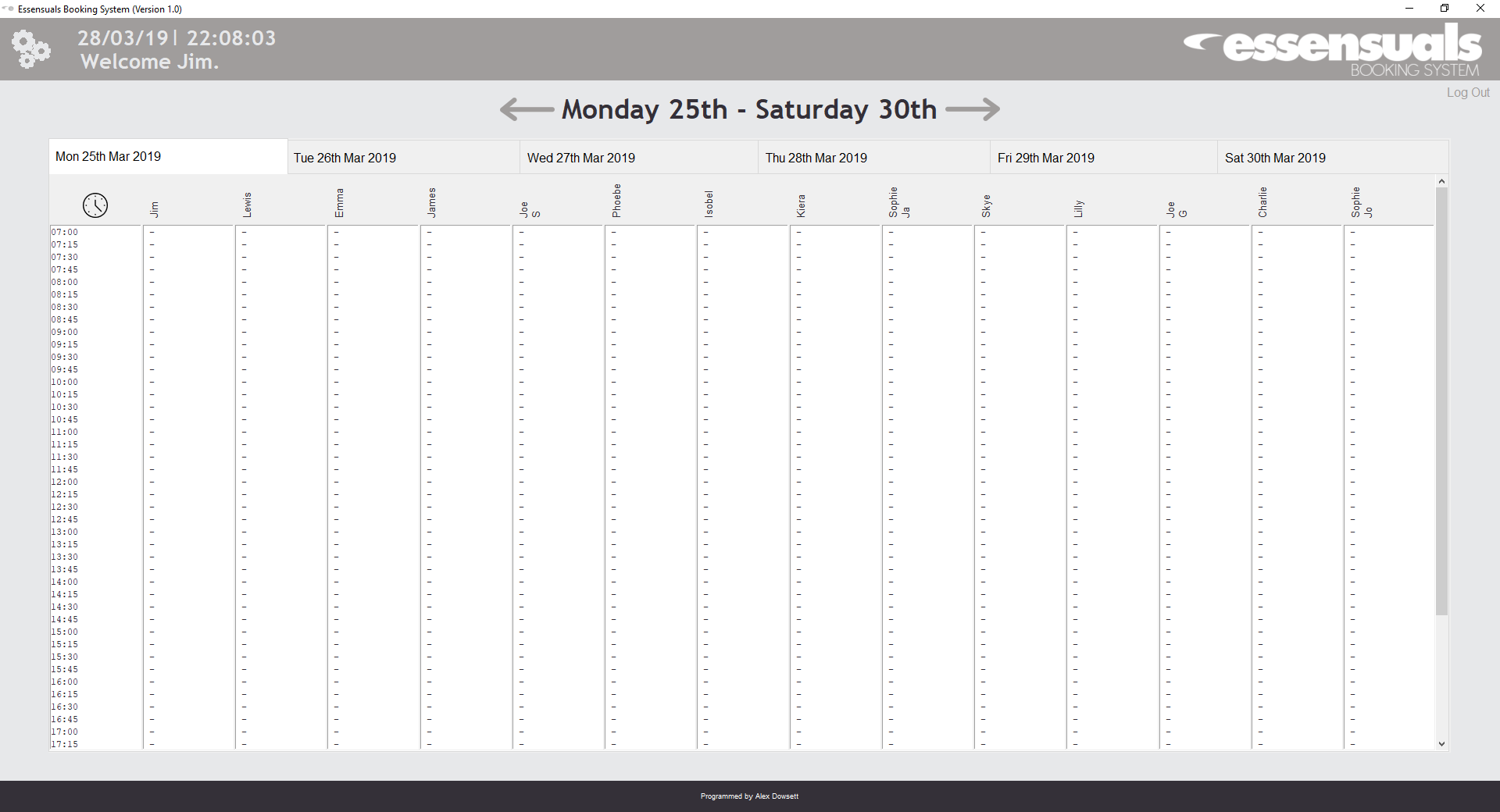
#### Figure 28d



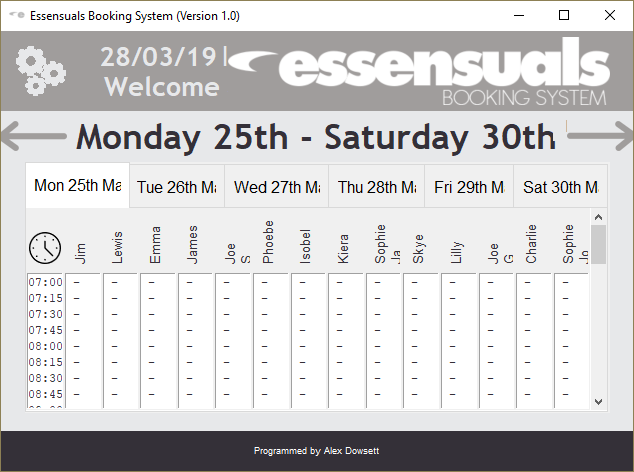
#### Figure 28e



#### Figure 29a



#### Figure 29b



#### Figure 30

