OCR GCE A

COMPUTER SCIENCE

PROJECT

H446-03

Name: Farida Addo

Candidate Number: 1507

Ark Globe Academy: 10818

Title of Project: Hospital Database

H446-03 – PROJECT CONTENTS

TABLE OF CONTENTS

ABSTRACT

Millions of pieces of data are used by the NHS daily. Patient data is used to enable the NHS to analyse the health needs of the population, and provides avenues for future research. Furthermore, the acquisition of patient data enables the identification of patterns and trends, thus improving services and decision making.

ANALYSIS

OUTLINE OF THE PROJECT

This project will focus on the development of a database for storing the details of hospital patients. The database will be comprised of a series of inputs which consist of basic details about the patient, such as

their name, age, and NHS ID. Furthermore, a graph will be implemented since visualisation enables data to be analysed easier. This will enable the quick identification of pattens and trends.

This project needs to be undertaken in order for hospitals to keep a record of their patients, so when a doctor meets a patient, they are aware of their history and what they need to be treated with. A database makes it easy to access patient records and enables administration to operate effectively. Moreover, the database may be seen as a more secure way to store information, as paper records may result in vital information being lost easily through water/ink damage, ripping, with no form of backup or retrieval.

HOW THE PROBLEM CAN BE SOLVED BY THE COMPUTATIONAL APPROACH

PROBLEM RECOGNITION

The key issue is that doctors have numerous amounts of data to handle, but nowhere to secure to store it. This problem is solvable through the computational method of problem recognition, and in particular through the database that will be created. The inputs will be written to a text file which serves as the database for the patient information. In order for the database to be returned to the user, the contents of the text file will be read from, and then printed onto the screen. The creation of the database will provide a clear, comprehensible method of storing data. A further advantage of this is the fact that databases can be easily changed and edited, solving the problem that may arise due to doctors making mistakes in writing, or updated information requiring further paperwork.

DATA CAPTURE, RETRIEVAL AND SECURITY

The database system will ensure that all data stored is recorded in a consistent fashion. The system will also allow security to be applied to the storage and retrieval of data through the use of access levels and user authentication.

PROBLEM DECOMPOSITION

Decomposition refers to the breaking down a task into a series of sub-tasks and may involve solving each one individually. Decomposition is a vital part of the program as each part of the system will be built in turn. This will increase the speed at which the program can be developed. This is particularly important as I should be able to respond to Dr. Hussain's feedback, and implement his suggestions as quickly as possible to ensure that the system is being built in line with his requirements.

VISUALISATION

Data visualisation will be a common theme of the program. One way in which data visualisation will occur is through the creation of the scatter diagram. This is a form of data visualisation as a visual image will be presented to a user, from which they will be able to draw conclusions.

DATA MINING

Data from the system will be made available via various media: web download, file transfer, email, which can then be accessed by many different systems. By incorporating the graph, underlying relationships within the data can be revealed. Furthermore, data mining enables data-driven decisions to be made as to where time and money should be invested. For example, if the scatter diagram shows that high and low numbers of COVID-19 vaccines result in high numbers of COVID-19 cases regardless, scientists may conclude that COVID-19 rates are high due to inefficient vaccines. This may suggest that scientists should design newer, more efficient treatments. The effectiveness of new treatments may then be demonstrated in the scatter diagram by a negative relationship between the number of COVID-19 cases and the number of COVID-19 vaccinations had. This insight will be beneficial to doctors and scientists, and will provide empirical evidence for the effectiveness of the vaccinations, meaning that more of it can be distributed, thus improving the health, and living standards of many as they can become immune to the disease.

THINKING ABSTRACTLY

The diagram above was the first prototype for the initial start-up page. There were benefits to this design as the entry of a hospital number would narrow down the records shown to a single hospital. However, this design was also impractical as it does not account for the fact that some users may not have a login. Furthermore, it does not consider the use of access levels which would help filter out the functions

Thinking abstractly is a method of computation thinking whereby the unnecessary details relating to a problem are removed so that only the necessary information remains. Abstraction will be incorporated in the creation of the scatter diagram as it will be reduced to only including information about two co-variables – the number of COVID-19 cases and the number of COVID-19 vaccinations a patient has had. The removal of unnecessary information is crucial in order for doctors and researchers to find a clear relationship between the co-variables, and focus on the them solely. By removing irrelevant pieces of information, it becomes simpler for researchers to draw firm and robust conclusions regarding the efficiency of the vaccinations. For instance, a researcher may derive that because someone with 2 COVID-19 vaccinations has had 0 cases, and someone with 1 COVID-19 vaccination has had 2 COVID-19 cases, 2 COVID-19 vaccinations are most effective. This could become useful in relation to policy making as the government may then advise that individuals take 2 vaccines.

THINKING AHEAD

This section will focus on the inputs, processes and outputs of the program being identified. As a result, thinking ahead has been incorporated.

INPUTS

The inputs which will create a record to be stored in the database would be information such as the patient details. This may include the patient's name, surname, or address.

PROCESSES

Two processes may take place in the program. One process would be reading and writing the patient details to file. Another process would be creating the scatter diagram, through the use of the number of COVID-19 vaccinations, and number of COVID-19 cases provided as patient details.

OUTPUTS

The output would be the table used to present the patient data, as well as the diagram demonstrating the relationship between the number of COVID-19 vaccines and, the amount of COVID-19 cases a patient has. This would provide a foundation for researchers to analyse the relationships and their implications.

PRECONDITIONS

One example of a precondition in the program is that in order for a user to access a single patient's details, the patient's NHS number, which is used to uniquely identify the patient, must be correct.

By specifying this precondition, I am aware of what checks will need to be completed before a user's record is outputted, for example. This makes the process of coding easy as I will know which parts of the code should be developed first.

ADVANTAGE OF THINKING AHEAD

A benefit of thinking ahead is that predictions may be made about a user's actions. For example, a user who has frequently used the database may perform common actions, such as logging in and then accessing the entire database. As a result, caching may be implemented, where the frequently used instructions may be stored in cache for quick retrieval. This would improve the overall performance of the system, aiding good user experience with the database.

THINKING CONCURRENTLY

Thinking concurrently is implemented in the program. Thinking concurrently occurs when multiple tasks are run simultaneously, thus allowing multiple processes to take place at the same time. This enables greater efficiency in the program as it means that while users are inputting patient data, the data can be written to the text file storing patient data simultaneously, and the diagram can be formed. While it may take a longer time to complete processes when multiple users are trying to run programs, this slowdown may be offset by the fact that there would be increased program throughput. Furthermore, the time that would have been otherwise wasted by the processor waiting for the user to input data, would be used on another task, maximising processor use.

CONCLUSION

In conclusion, the program needs to be run immediately with constant availability. Because the proposed program is amenable to a computational approach with clearly defined inputs processes and outputs, it can run with greater efficiency, which is necessary when handling large amounts of important patient data. Overall, solving the problems entails a wide range of computational thinking approaches, which are very useful.

RESEARCH INTO SIMILAR PROGRAMS

An example of a similar program was devised by the National Center for Biotechnology Information[1]. It includes a range of patient data which would be useful for the stakeholders mentioned earlier. For example, it includes demographic data elements, such as the patient's address of residence. This is advantageous and may be deemed an essential feature as it enables doctors and nurses to send patients letters home regarding their necessary medication, and perhaps correlate the incidence of certain health disorders with the area in which the patient resides in. Furthermore, the database also includes information about patient allergies which is beneficial for doctors when prescribing treatments as they can ensure that the medicines handed out will not result in negative side effects, and that the body will not counteract the medication. While this database has many benefits, there may also be some problems faced when trying to implement these features. For example, patient allergies may be constantly changing. As a result, it may become outdated and may result in patients being deprived treatment due to past information stating that they may have a rection to the medication. In addition to this, if the allergies are not updated, patients may be prescribed a treatment which they suffer an allergic reaction from as it was not known to the doctor how they would react to the medication.

Another example of a similar database is the Hospital Episode Statistics[2] database. This database contains the details of all admissions, attendances, and outpatient appointments at NHS hospitals in England. For example, the database records information about the gender and ethnicity of patients. This is significant as it means that doctors may be able to detect trends in patients based on their gender and ethnicity, and prescribe treatments in order to prevent the acquisition of certain disorders on their onset. Some further benefits of this database is its ability to assess effective delivery of care and determine fair access to health care. A limitation of this database is that it may contain unnecessary patient information which could take up a large storage space. For instance, the database considers private patients treated in NHS hospitals. This is a limitation as where the patient is treated does not affect how they are prescribed treatment, so would be unnecessary to add to the database. This is an example of how thinking abstractly would be incorporated in the program in order to make the most efficient use of memory.

Having looked at these databases, the final product will aim to collect inputs regarding the address of the user, as well as their gender and ethnicity.

STAKEHOLDERS

[1] https://www.ncbi.nlm.nih.gov/books/NBK236556/

[2] https://digital.nhs.uk/data-and-information/data-tools-and-services/data-services/hospital-episode-statistics

IDENTIFYING STAKEHOLDERS

A stakeholder is an individual with an invested interest in something. In this case, the target end users are doctors, nurses, and patients. The database may also have implications for the government. The problem that these stakeholders have is the low security derived from using paper copies to store patient data. This form of storage can be lost easily, and is also susceptible to damage. In addition to this, another problem that the stakeholders have is the fact that it may be difficult to detect health trends as they do not have a comprehensive store of all patient data. As a result, the proposed solution will aim to overcome these problems.

The stakeholders mentioned are particularly important as they are who the database affects directly. There are several reasons why the proposed solution is appropriate to their needs. For instance, it is the patients' data which will be stored in the database. This makes patients important, as when attending appointments, they may require a comprehensive, easily accessible store of their health records. Doctors and nurses also need to keep track of patient data, so they know what medications to prescribe, and so that they can detect the incidence of certain disorders.
The reason there may be implications for the government is because it is integral that they keep a record of the health of their population, so that they can ensure that the workforce is fit for work and maximise output – this is particularly important during a time like this where there have been many fatalities as a result of COVID-19. Furthermore, knowledge of the population health – through using the database – is important as it will aid them when setting government policies, to ensure that the policies are in society's best interest.

In this instance, the doctor who will constantly be referenced, and updated regarding the program developments in order to ensure that the proposed solution meets all of their needs, is Dr. Hussain.

Overall, the provision of the database will provide an easily accessible and more secure medium of storage for the stakeholders mentioned, which can then be used by them in a variety of ways.

QUESTIONNAIRE

In order to gain a better understanding of my stakeholders' needs, a questionnaire will be issued, which will collect data regarding their needs and wants for the program. As a result, information will be provided regarding how the product should look and feel. Furthermore, the program will be able to take a more client-based approach and will be targeted towards providing features which directly benefit the client. For instance, the database will not consider a patient's favourite colour, as this knowledge does not aid doctors and nurses when it comes to treating patients, so would be pointless to add to the database.
In order to ensure that a wide range of viewpoints have been considered, the questionnaire sample will consist of 100 people, who are studying medicine or are already doctors. Dr. Hussain's responses will also be considered as a part of this.

QUESTIONS

1. What age bracket are you a part of?
2. Do you know how to use a database?
3. What kind of data do you wish to store in the database?

4. How satisfied are you with the current method used to store patient data?
5. Would the creation of the database be better than your current method of storing data?
6. What is your expected database capacity?
7. What are your security requirements?
8. Is data visualisation important to you?
9. Is having an easy-to-use software interface important to you?
10. Do you wish to connect this system to any other application i.e., Outlook?

The reasoning behind each question:

"What age bracket are you a part of?"
The purpose of asking this question is to assess the requirements for doctors and nurses across ages, and ensure the program meets all their needs. In addition, by knowing their ages, the technical capabilities of respondents can be associated with their ages. This is important as it may give an indication as to how easily users will be able to interact with the system.

"Do you know how to use a database?"
It is important to know how experienced the users are with a database, in order to determine how complex it should be. If they are unaware about the operations of a database, then a briefing of how to manage its operations may be required.

"What kind of data do you wish to store in the database?"
This information will help determine what the field names will be.

"How satisfied are you with the current method used to store patient data?"
If respondents are satisfied with their current method of storing data, then perhaps the provision of a database would be unnecessary.

"Would the creation of the database be better than your current method of storing data?"
This question will help determine the importance of the project by ensuring that it provides a better, more efficient means of storing data.

"What is your expected database capacity?"
The purpose of this question is to ensure that the database size does not compromise its performance, so that it operates as smoothly as possible.

"What are your security requirements?"
Important details will be stored in the database. This question is targeted at ensuring that the right security measures are put in place to protect this data.

"Is data visualisation important to you?"
If data visualisation is important, then perhaps more time should be spent on the design of the screens. However, if data visualisation is not important then more time can be spent on maintaining the functionality of the program.

"Is having an easy-to-use software interface important to you?"
This is question is aimed at determining whether an interface needs to be created which understands and fulfils user preferences, or whether this is unimportant, as other aspects of the program may require further attention.

"Do you wish to connect this system to any other application i.e. Outlook?"
By asking this question, I can find out how accessible the system needs to be to users, and whether it should have platform independence or not.

ANALYSING THE RESPONSES

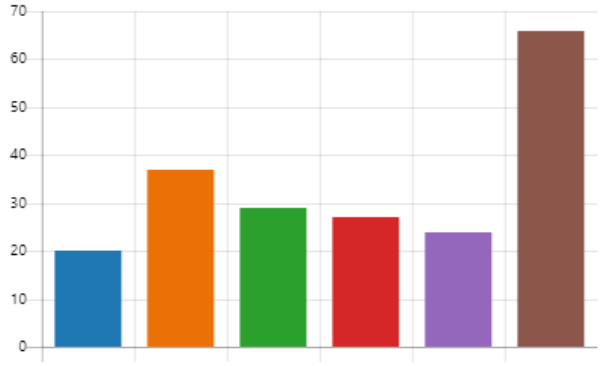
1. What age bracket are you a part of?

20% of respondents are under 21
39% of respondents are between ages 21-30
16% of respondents are between ages 31-40
15% of respondents are between ages 41-50
10% of respondents are over 50
From these results, it is clear that potential consumers of the product are between ages 21-30. This knowledge is beneficial as it suggests that complex features can be added to the program as 21-30 year olds would be at a stage of their life where they may be more willing to learn new things. On the other hand, if the majority of consumers were over 50 then perhaps more modern features would be omitted from the program development.

2. Do you know how to use a database?

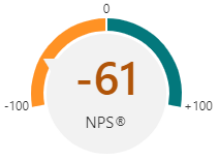
56% of respondents said Yes
44% of respondents said No
From this information, it is clear that the operations of the database will be familiar to the users, suggesting that there is not a requirement of spending extra time teaching them how to use it. This ensures that the system can be given to, and used by as many people as possible.

3. What kind of data do you wish to store in the database?



11% of respondents selected 'patient data'
16% of respondents selected 'the vaccination status of patients'
14% of respondents selected 'COVID cases of patients'
13% of respondents selected 'the medical history of patients'
12% of respondents selected 'patient allergies'
34% of respondents selected 'all of the above'
This data suggests that the database should incorporate patient data, such as patient allergies. In addition to this, because respondents selected to include COVID cases of patients, as well as the vaccination status of patients, it suggests that the database's incorporation of the scatter diagram will be useful.

4. How satisfied are you with the current method used to store patient data?



The net prompter score suggests that on average, current users are not satisfied with their current method used to store patient data – which in this case is likely to be pen and paper copies. As a result, this suggests that there is a requirement for the database.

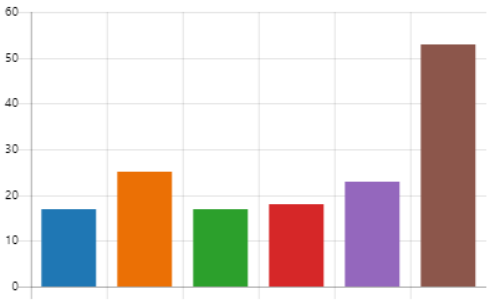
5. Would the creation of the database be better than your current method of storing data?

59% of respondents said 'yes'
19% of respondents said 'no'
23% of respondents said 'it wouldn't make a difference'
The findings suggest that the database would be better for users – perhaps due to the low security had by keeping paper copies of patient data, or because online records are easier to manage. In light of this, there may be greater confidence in the idea that the creation of the database will be benficial to users during their day to day operations with patient data.

6. What is your expected database capacity?

10% of respondents selected 'under 1TB'
13% of respondents selected '1TB-10TB'
11% of respondents selected '10TB-20TB'
66% of respondents selected 'over 20TB'
Knowing that most of the users will require a database storage of over 20TB is essential as it means that it can be determined whether existing hospital infrastructure will be able to sustain the anticipated workload.

7. What are your security requirements?



11% of respondents selected 'encryption'
16% of respondents selected 'password protection'
11% of respondents selected 'user access levels'
12% of respondents selected 'firewalls'
15% of respondents selected 'anti-virus'
35% of respondents selected 'all of the above'
Being aware of the security requirements of users enables the safe operation of applications implemented on the organisation's IT systems. Knowing that a lot of respondents require password protection suggests that it is a feature which will need to be incorporated into the system.

8. Is data visualisation important to you?

63% of respondents selected 'yes'
20% of respondents selected 'no'
18% of respondents selected 'it doesn't matter'
In light of this data, when developing the program, there may be time allocated to ensuring that that the data can be easily interpreted and understood by its users.

9. Is having an easy-to-use software interface important to you?

75% of respondents selected 'yes'
14% of respondents selected 'no'
11% of respondents selected 'it doesn't matter'
Because a majority of participants find a software interface important, a simple interface will be incorporated in order to disguise the complexities of managing and communicating with the computer's hardware. This will enable less time to be spent on understanding the operations of the system, allowing users to focus on maintaining and updating patient information – ensuring efficiency. Furthermore, the easy-to-use software interface will improve the usability of the database.

10. Do you wish to connect this system to any other application i.e., Outlook?

46% of respondents selected 'yes'
54% of respondents selected 'no'
Because users do not wish to connect the system to any other application, more time can be spent on creating the program as quickly as possible.

The findings of the research carried out with stakeholders has provided information which can be used to create the inputs of the program. The solution is appropriate to their needs as it is clear from the data that their current method of storing data is not ideal. Therefore, by providing a database which can be easily accessed, maintained, and backed up, the limitations of using pen and paper to store patient information can be overcome. As a result, a more reliable, efficient method of storing data will benefit stakeholders in the long run. For instance, doctors and nurses can easily access patient records, the government can easily track patient data, and patients themselves have a complete, concise record of their data, which can be accessed by them when required if needed for future reference.

FEATURES OF THE PROPOSED SOLUTION

FEATURES

The aim of the project is to create a database which contains patient information. Examples of this patient data have been derived from the responses to the questionnaire, and insights from conversations with Dr. Hussain. The data will take a tabular format, as displayed below:

Patient Name	Patient Surname	Patient Age	Patient Gender	Patient Ethnicity	COVID Cases of Patient	Patient Vaccination Status	Patient Allergies	Patient Address
X	X	X	X	X	X	X	X	X
X	X	X	X	X	X	X	X	X
X	X	X	X	X	X	X	X	X
X	X	X	X	X	X	X	X	X
X	X	X	X	X	X	X	X	X
X	X	X	X	X	X	X	X	X

The final project will include a more extensive list of patient data. For abstraction, the image displayed only includes a few inputs. The inputs displayed are essential as they enable the identification of patients. For example, the patient's name and surname is required in order to distinguish who they are from others. Their age is also required as it enables doctors to correlate age with the incidence of certain disorders, so they can be prepared to tackle the disorders amongst certain age groups. The ethnicity and gender of patients is also required for this same reason. Certain disorders may be seen mostly in certain genders/ethnicities. By being aware of this for each patient, doctors can once again prepare for its incidence within certain demographics. Having said previously, patient allergies are important inputs to have so doctors can ensure patients are not given medication which they will have a reaction to. The patient address is good knowledge for doctors to have to send them emails regarding their next appointment for example, thus also benefitting patients.

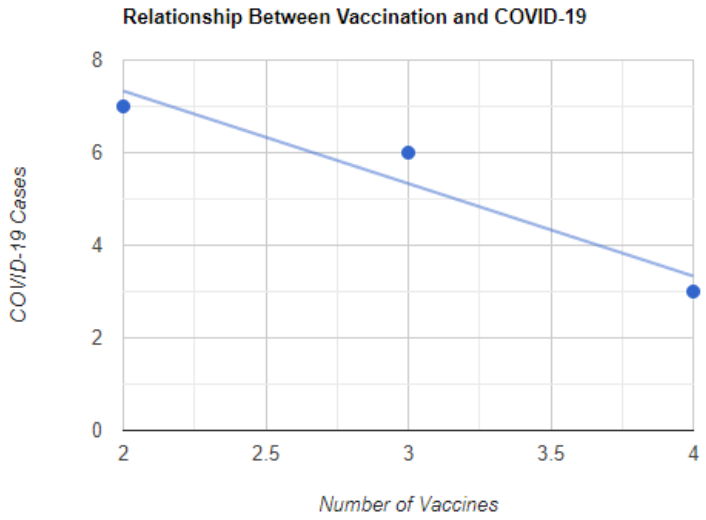


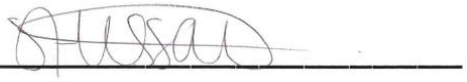
Figure 1

The program will also aim to include a diagram, correlating the number of vaccinations had with the number of COVID cases a patient has had. As a result, the COVID cases of the patient, and the number of vaccines they have had are essential inputs in order to create this diagram. This will determine how effective the vaccine dosage is in preventing the acquisition of COVID. As a result, doctors such as Dr. Hussain may be able to conclude that the vaccines are effective with greater certainty if a negative correlation is found.

LIMITATIONS OF PROPOSED SOLUTION

Having said this, there may also be some problems with the implementation of this program. For example, the way in which the database is implemented by stakeholders may pose an issue. If the database is used by a network, then it may be susceptible to hacking. This would breach the Data Protection Act, and may lead to important patient information being leaked, which may have negative consequences. In addition to this, the database may be reliant on an internet connection in some cases, which would make it unreliable in cases where internet connection is poor, as accessing data may be slow, or impossible. Furthermore, if many doctors are trying to access the database at once then it may act slowly, which may be problematic in cases where doctors require immediate access to patient data. Backing up the database is integral, as, if the computer system has a breakdown, the data cannot be accessed. Finally, the main limitation of the database is its cost. Thousands of data will require storage, which may incur excessive costs. In addition to this, there may be training costs as stakeholders are taught how to operate the system.

Overall, the program is targeted at ensuring that the responses provided from the questionnaire are incorporated as inputs, and are met throughout the development of the program. These features have also been discussed with Dr. Hussain, who has agreed that the features will meet the needs of himself, as well as other doctors. His signature of approval can be seen below:

Signed: 

SOLUTION REQUIREMENTS

HARDWARE AND SOFTWARE REQUIREMENTS

In order to create the program, there are a variety of hardware and software requirements. For example, the database will require over 20TB of storage, as recommended by questionnaire respondents. This is necessary as anything less may not be sufficient enough to store the vast amounts of patient data which require storing. There are also huge numbers of patients in a population, so the high storage will need to account for this.

An 8GHz clock speed is required as high storage is beneficial, but only if the operating system can perform them at a fast rate, thus improving throughput.

A quad-core processor may be required. This will ensure that several tasks can be run simultaneously. Parallel processing may be desirable, in order to allow the processor to work as efficiently as possible. As a result, patient data can be accessed quickly, and the overall performance of the database will improve.

Level 2 cache would be the most efficient to have. This is because level 3 cache may make the operating system too slow. On the other hand. Level 1 cache may not have sufficient memory. As a result, level 2 cache will provide a moderately fast, and yet moderately sufficient storage, which can be accessed during times when Random Access Memory (RAM) is too full. The reason it only needs to be moderate is because the other requirements of the system such as parallel processing means that there is not much

of a requirement for a large cache size. In light of this, 20GB of RAM would also be most suitable for the purpose of storing the data.

In order to ensure the security of the data, a variety of methods may be employed. For instance, an anti-virus subscription may be required in order to ensure that the data is not vulnerable to hacking. In addition to this, full backups will be required to ensure that copies of the data are made, and can be used in instances when data needs to be recovered from a disaster, such as in the event of a power failure. These backups should occur each time a change is made to the system in order to ensure that the most up to date copy is stored, so should be automated and continuous. The backup retention required will be determined by the service level agreement. As a result of this retention, if users make accidental changes to the data, they still have a copy that they can return to, to amend any mistakes made.

A 1024 x 768 resolution will be needed in order to ensure that the data is clear to those accessing the database. A lower resolution may mean that information is hard to decipher, which may lead to inaccuracies when doctors read patient details/prescriptions, which could then lead to further negative consequences if patients receive the wrong treatment etc.

A Graphics Processing Unit (GPU) may also be needed in order for data to be visualised for real-time analytics. As a result, doctors such as Dr. Hussain may be able to receive the analytic data they need, at the point of care, to make a more accurate diagnosis. Furthermore, the GPU will enable the processing of large amounts of data. This is particularly useful considering the vast amounts of data which will be stored.
A DirectX 10 graphics card will enable graphics hardware acceleration for the PC, which may be beneficial when doctors need to interpret the scatter diagram.

A Windows Operating System (OS) is also required. This is because it is commonly used, so provides an easy user interface as users are likely to already be familiar with the interface. From the questionnaire responses provided before, this is an important feature for 75% of respondents. In addition to this, a Windows OS is needed because it has a variety of available software, which can be of use to doctors, such as a word processor, and excel – which could be used for calculations. Using an OS with many applications (PowerPoint, Word, Excel etc.) also ensures consistency within the system as users do not have to switch between using Google Chrome, and then Microsoft Outlook, for example, which could lead to processing taking longer as they are not from the same OS. Windows also has support for new hardware which may reduce costs for hospitals as their machine is likely to be compatible with the Windows OS.
While a Windows OS is desirable due to its multiple applications, it is not essential. The system will still work with a macOS or Linux OS.

A text file will be needed in order to store the database information, such as the list of ethnicities, login details, and ‘Patients’ file.

The user will need a monitor, keyboard and mouse to run the system on. The purpose of the monitor is to display the contents of the system. The keyboard is so that the user can enter inputs and commands within the system. The mouse will help with functionality and usability as the user will be able to select different options i.e. whether they want to enter patient details of have the contents of the scatter diagram displayed.

Finally, because the program will be written using Python, a Python IDE is required to store the code. The IDE must include Tkinter as it forms the basis of the GUI.

SUCCESS CRITERIA

In order to ensure that the fundamental features of the program are met, the success criteria below will be used to measure the success of the solution and will also be regularly referred to and checked during the development process. The criteria is based off the responses produced in the questionnaire, and may be amended and changed as needs arise.

Success Criteria	Justification	Completed
Gather inputs about a patient.	The inputs will provide data used to create the database records. The field names for the inputs will be derived from the responses produced from the questionnaire i.e., patient allergies, the medical history of patients.	
Enables several patient's details to be inputted.	This would be beneficial when the doctor needs to input a set of data for multiple patients. If doctors are only able to enter details for one patient, then the usefulness of the system may decline.	
Create a diagram which represents the relationship between the number of vaccines had, and the number of COVID cases.	This will provide a diagram which can be referenced to by doctors when determining the effectiveness of the vaccine.	
Enable a record of a patient's data to be outputted.	This will allow doctors to gather all the details of a patient, which they can then use to access their medical history. This will provide the easy-to-use interface that 75% of respondents selected.	
Enable NHS IDs to be created.	This will provide patients with a unique number which will make it easier for doctors to distinguish between patients as some patients may have the same name, causing confusion. 11% of respondents selected patient data, which includes the NHS ID.	
Incorporate checks for NHS ID.	This ensures that multiple people do not have the same NHS ID, as it could lead to the wrong identification of patients. Therefore, the primary key (NHS ID) is required as a unique identifier for each patient record.	
Write the patient data to file.	This will allow patient details to all be stored in one place and will essentially create the database.	
Use data from file to create data visualisation screen.	The creation of a data visualisation screen file will enable the data to be displayed in a format which can be easily read. 63% of respondents in the questionnaire argued that data visualisation was important to them.	

Incorporate checks for addresses.	This ensures that the address given by a patient is authentic, allowing doctors to send letters to the correct address.	
Create home screen.	The creation of a home screen will provide users with a range of access levels to choose from. This is important as it meets the security requirement of 11% of questionnaire respondents. Additionally, home screens are beneficial as they provide users with a small introduction into the system, without being bombarded with many commands and functions. This is important as it improves the usability of the system, meeting the need of 75% of questionnaire respondents who deemed having an easy-to-use software interface as important.	
Create patient login screen.	This is important in meeting the security requirement of 16% of questionnaire respondents who deemed password protection as important. As a result, the system may have increased security as a login prevents unauthorised access to sensitive data. Moreover, a login screen is essential for patients to access their data. Without it, it would not be possible to determine which patient is who, and ensure that the correct information is provided to the users.	
Create login page.	This is important in meeting the security requirement of 16% of questionnaire respondents who deemed password protection as important. As a result, the system may have increased security as a login prevents unauthorised access to sensitive data.	
Separate main menu for 'Nurse' access level.	This introduces the concept of an access level, which is important to 11% of questionnaire respondents as a security requirement. It is also needed to meet the use case diagram requirements. This is because, apart from the patient, the nurse is the only other access level without rights to the scatter diagram. This means that a separate main menu is required for them.	

DESIGN

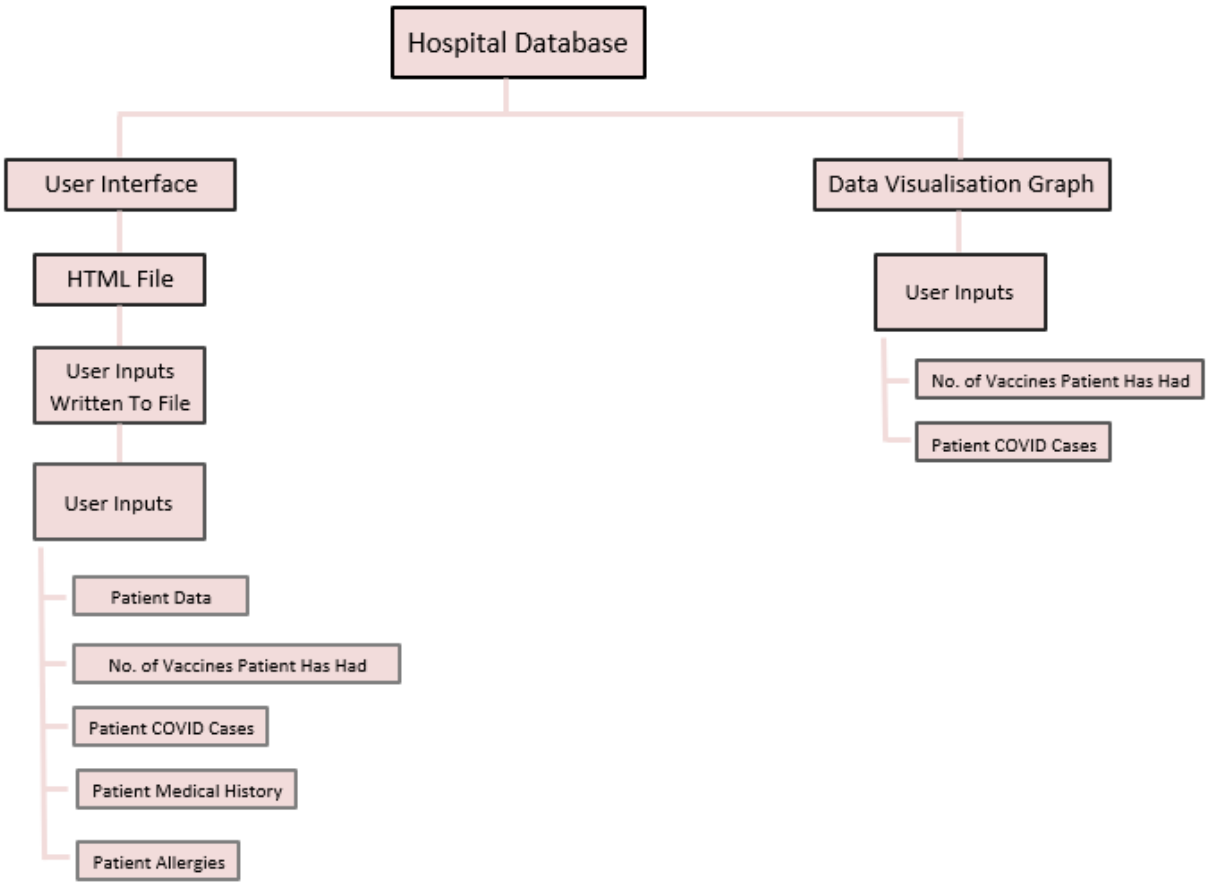
STRUCTURE OF THE SOLUTION

In order to make it easier to solve the problem, the top-down module design will be used in order to decompose the problem into subtasks. By implementing stepwise refinement, the program will become more manageable to solve, and thinking ahead may be incorporated.

When designing the program and receiving constant feedback from Dr. Hussain, the requirements of the program changed, hence the several diagram prototypes.

TOP-DOWN DIAGRAM – PROTOTYPE 1

The diagram below illustrates the subtasks which will be solved, which are derived from the success criteria developed previously. As a result of this decomposition, the program will be easier to test, maintain, and code, through logical and procedural thinking. Furthermore, changes can be made to individual modules without it affecting the rest of the program. In order to tackle each module independently, two branches have been made. The abstracted diagram ensures that only key elements will remain the focus of the program.



USER INTERFACE BRANCH

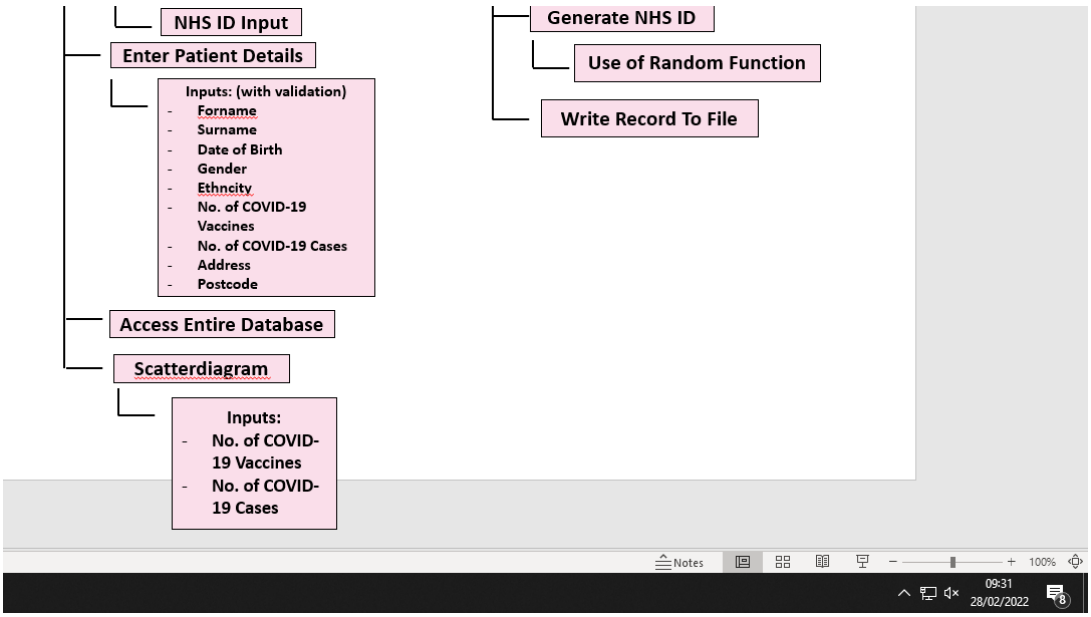
The user interface branch consists of the creation of a user interface, which will be implemented in order to visualise the database inputs written to file. The reason for this is because 63% of questionnaire respondents deemed it necessary. Furthermore, after consulting with Dr. Hussain, it became clear that this feature, which manages and simplifies the complexities of the hardware for the user, is required. Because of this, accessing, amending, and tracking patient records will be made easier.

DATA VISUALISATION GRAPH BRANCH

The data visualisation graph branch is a fundamental aspect of the proposed solution. It refers to the creation of a scatter diagram which will be derived from the user inputs which state the number of COVID-19 cases a patient has, as well as the number of COVID-19 vaccinations a patient has had. It is a feature that has been implemented because Dr. Hussain considers this important, since it will enable doctors like himself to have a simplified diagram which provides information regarding the effectiveness of the vaccine in preventing the acquisition of COVID-19. With insights from this diagram, doctors will be able to develop newer, more efficient vaccines if necessary.

TOP-DOWN DIAGRAM – PROTOTYPE 2

After designing the initial top-down diagram, amendments were later made. This is because the previous design did not account for a graphical user interface (GUI). The use of HTML would have been a substitute for a GUI, however I realised that, in order to ensure that the user is able to interact with the system well, and that the program was developed in line with the wants of 75% of questionnaire respondents who wanted to have an easy-to-use software interface, Tkinter must be incorporated. This led to a change to the problems which would need to be solved independently. The refined top-down diagram can be seen below:



GRPAHICAL USER INTERFACE BRANCH

The GUI branch will include the creation of the different screens for the user.

HOME SCREEN SUB-BRANCH

The home screen sub branch will include the development of 3 areas. Firstly, the patient login, which will require the provision of an NHS ID as an input. This meets the success criteria of generating an NHS ID, which is important as it provides a unique identifier for each patient record, improving the searching of patients. The reason why the NHS ID has been used as the input value is because it is unique to each individual. This means that it is a secure form of input as it is unlikely that anyone will have the same number. It is also secure as the ability for someone to guess a random 10-digit number is also unlikely. Secondly, logins will be created for the access levels, nurse doctor, scientist and government official as they are key stakeholders in the development of this program. This section will require the provision of username and password inputs for security reasons, emphasised by 16% of questionnaire respondents. Finally, the registration feature was added in light of feedback from Dr. Hussain, for reasons explained in the development section.

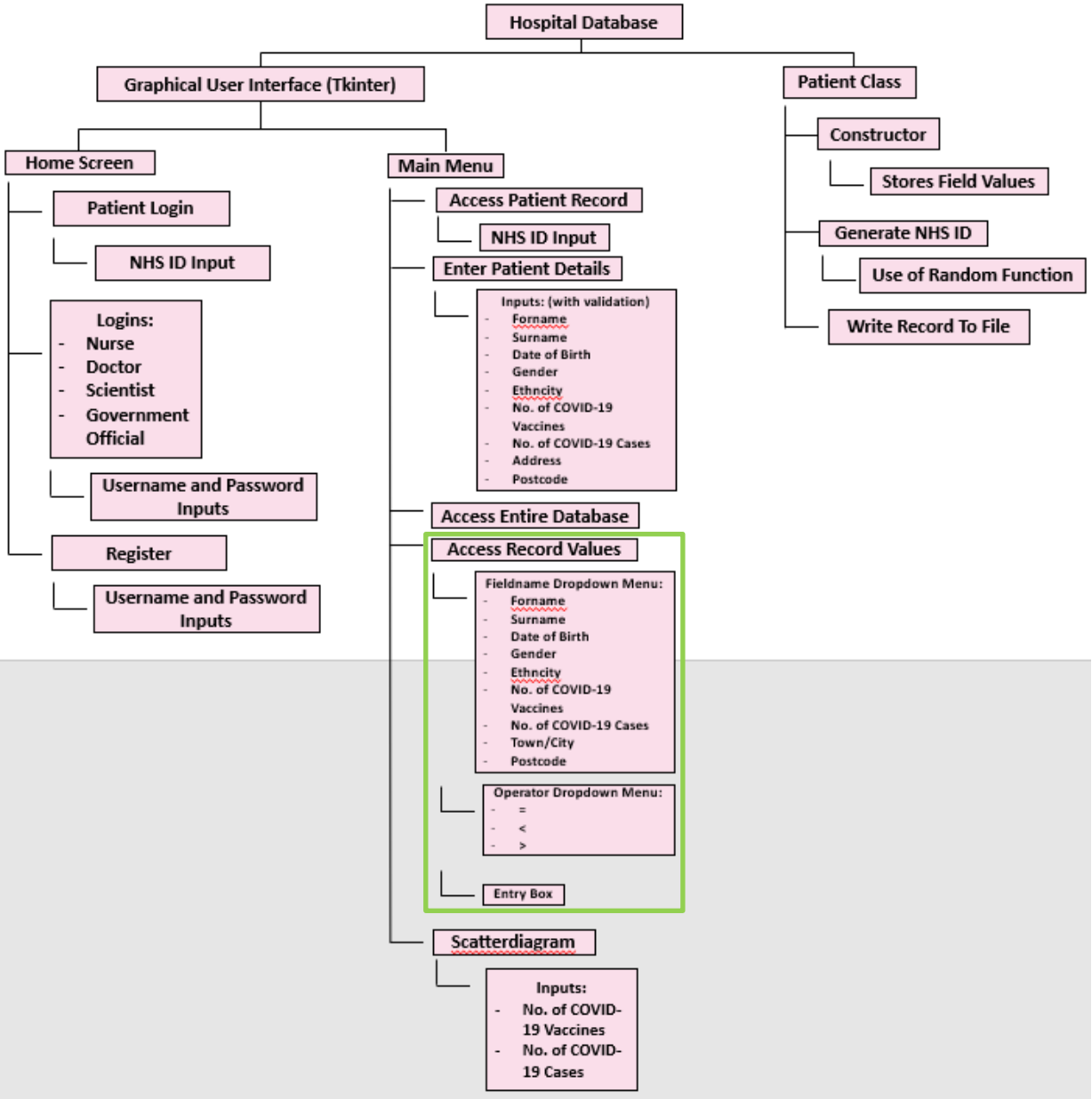
MAIN MENU SUB-BRANCH

Once a user logs in successfully, they will be introduced to a main menu screen, where they will have the options to access a patient's record, enter patient details, access the entire database, or output the scatter diagram. Instead of using the HTML to output the contents of the database, Tkinter features will be employed. By using Tkinter, the user will be able to see the database contents without requiring extra storage for a separate HTML file.

PATIENT CLASS BRANCH

The patient class branch will create a record for each user. This is not part of the GUI as it is a process which will occur without the user's knowledge. A constructor method will be used to collect the field values for each record, and the NHS ID will be randomly generated and then added as a field value for the record. Finally, the composed record will be written to database file.

TOP-DOWN DIAGRAM – PROTOTYPE 3

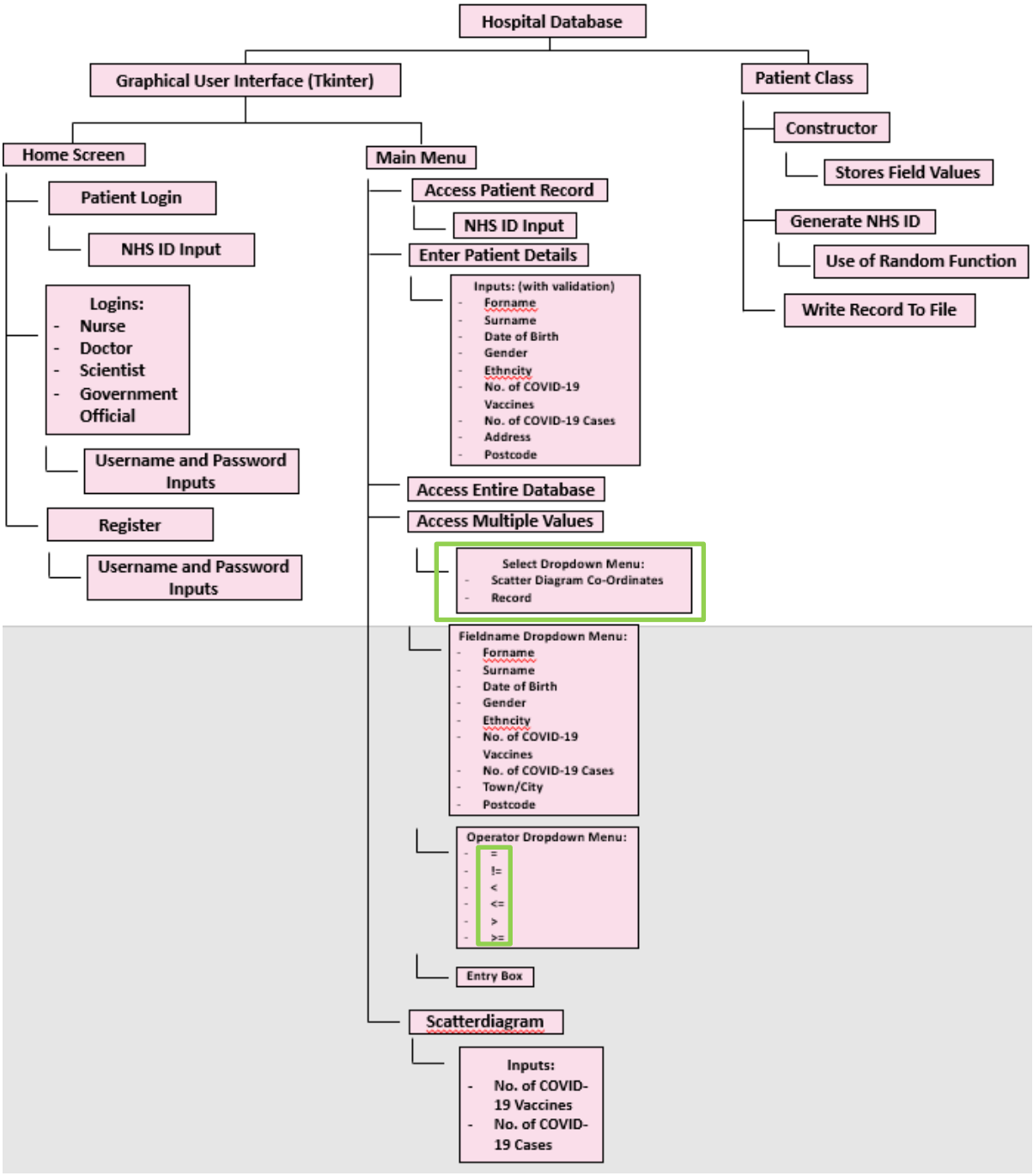


The top-down diagram was amended (additions in green), after the new 'Access Record Values' option was created. This screen will have a 'SELECT FROM WHERE' statement – to mimic SQL, and there will be 3 drop down menus.

DROPDOWN MENUS

The fieldname drop-down menu will allow users to select which field value they would like to check each record value against. The comparator will be determined by the user, who will have the drop-down menu options '=', '<' or '>'. Finally, an entry box will be incorporated that allows a user to specify what value must be met in order for a specific record to be outputted.

TOP-DOWN DIAGRAM – PROTOTYPE 4



The top-down diagram was amended (additions in green), following feedback from Dr. Hussain. The dropdown menu options are specified, and extra operator values were added in order to increase the number of tasks the user can perform.

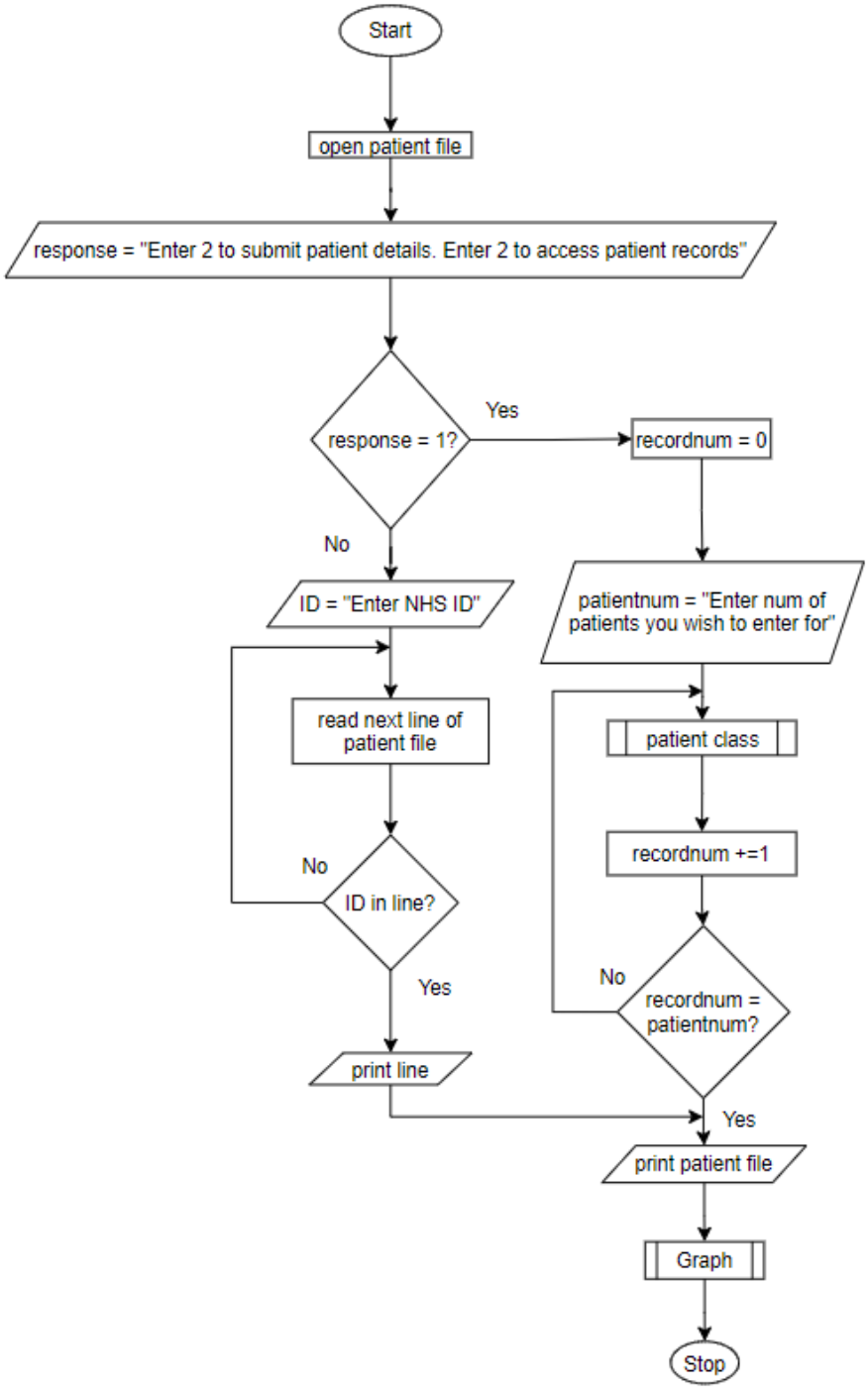
SELECT DROPDOWN MENU

The select drop-down menu will provide two options for the users, as suggested by Dr. Hussain – selecting scatter diagram co-ordinates, or record values.

ALGORITHMS

FLOWCHART FOR GENERAL PROGRAM

The flowchart below provides a representation of the logical flow of the general program, which brings together the subroutines graph, and patient class – which will be discussed later. By breaking down the flow of the program into subcomponents, writing the code will be made easier as each constituent part can be focused on at a time.



The flowchart above represents the main program. Initially, the patient file will be opened, so it can be used later on for reading or writing. Following this, the user will be asked to enter one of two options – whether they want to submit new patient details (1), or access existing patient files (2). If the user picks 1, they will then be asked the number of patients they would like to enter for. Afterwards, the patient class subroutine will run, whereby the data record can be entered. This process will repeat for n number of times, where n represents the number of patients they would like to enter for. This successfully meets the success criteria of enabling several patient's details to be inputted. On the other hand, if the user selects 2, they will then be

asked to enter the patient ID. The patient file will be checked line by line for this ID. If found, the record containing the specified patient's data will be printed. This successfully meets the success criteria of enabling a record of a patient's data to be outputted.

After these processes are carried out, the entire database will be outputted to the user, as well as the scatter diagram, which can then be interpreted by the user.

PSEUDOCODE FOR GENERAL PROGRAM

The flowchart only provides half of the solution. In order to provide a complete solution, pseudocode will be provided alongside the flow diagrams in order to provide an in-depth insight as to how the solution will be created in relation to the programming aspect.

```
html_file.write("<HTML>")

html_file.write("<table border = 1>")

index = 0

file = open(patientfile)

procedure accessrecords(id):

        for line in patientfile:

        if id in line then

                print line

task = input("Enter 1 to access patient records and 2 to add new patient data")

if task = 1 then

id = input("Enter NHS ID")

        accessrecords(id)

else if task = 2 then

        number = input("How many patients would you like to enter for?")

        for x in range (number):

                forename = input()

                surname = input()

                date of birth = input()

                nhsId = input()

                vaccineNo = input()

                covidCases = input()

                allergies = input()

                gender = input()

                ethnicity = input()

                address = input()

        patient(forename, surname, age, nhsId, vaccineNo, covidCases, allergies, gender, ethnicity,
address)

for j in file:

        html_file.write("<TR>")
```

```
        k = j.split(",")
        for j in k:
            if index == 0:
            html_file.write("<TD><b>" + j + "</b></TD>")
        else:
            html_file.write("<TD>" + j + "</TD>")
            html_file.write("</TR>")
            index+=1
print(file)
graph()
```

The pseudocode for the flowchart above starts by creating a table within a HTML file. The code then asks the users which of the two options to choose between. If the user selects option 1, they must provide the patient's NHS ID as an input. Following this, the subroutine 'accessrecords' is pointed to, which reads each line in the patient file, and outputs the line if the NHS ID is in the line. This meets the success criteria of enabling a record of a patient's data to be outputted.

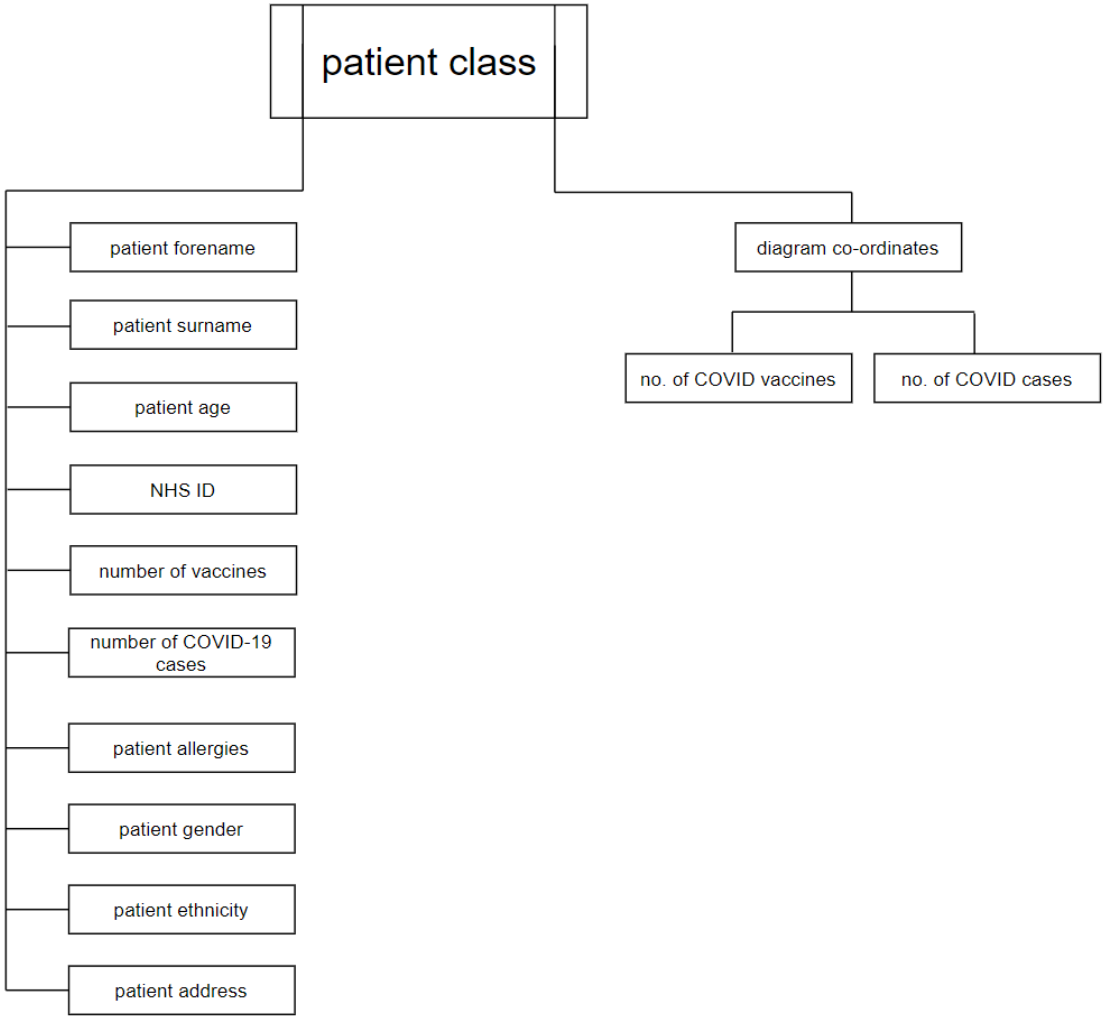
On the other hand, if the user chooses option 2, they are asked how many patients they would like to enter data for. The field values are then taken as inputs, meeting the success criteria of enabling inputs to be gathered about a patient. Following this, the patient class is pointed to, where the inputs provided by the user are passed as arguments. The workings of this class will be discussed in another section.

The patient file is then read. A new row is made, and the user inputs are added to each column, where this is represented by the comma separating the patient field values. If the first line is being read, the contents of the record will appear in bold. Otherwise, the values will simply be added to each column. This meets the success criteria where the data from the file is used to create a data visualisation screen.

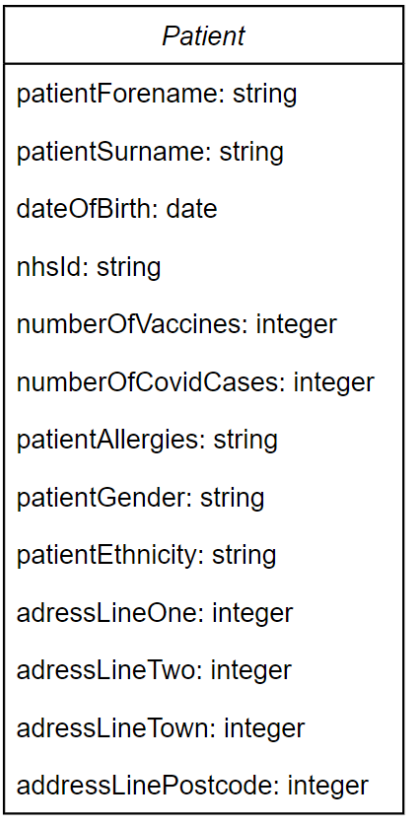
The algorithm then prints the patient file. The patient file printed is the comma separated values (CSV) file which will store the patient details. The reason why the HTML file is not printed is because this document is separate for those who prefer data visualisation (such as the 63% of questionnaire respondents). However, each record is printed where each user input such as a patient's forename and age, are separated by commas, with a new patient's details on a new line.

Afterwards, the contents of the scatter diagram are also displayed to the user.

CLASS DIAGRAM FOR PATIENT CLASS



The second flowchart represents the logical flow of class diagram. Because several records will be inputted, it would be most efficient to incorporate classes, as each object (patient) will have the same attributes. The attributes in the program have been derived from the responses on the prior questionnaire. Furthermore, after consulting Dr. Hussain, the following attributes were deemed appropriate. The attributes are the patient forename, surname, age, NHS ID, allergies, ethnicity, gender, address, the number of COVID-19 vaccines, and, the number of COVID-19 cases. In addition to this, the scatter diagram's coordinates will be derived from the inputs: number of COVID-19 vaccinations, and number of COVID-19 cases.



The class diagram designed previously was refined (see left) after showing the previous design to Dr. Hussain. For instance, the address has been broken down into sub components in order to make it easier to manage the address inputs. This is important as it allows patient address information to be derived separately. If the address was stored as one string, it would take a while to gather the first address line, and postcode, for example. However, by breaking the address down in this way, users can easily select the postcode value, making searching faster and improving the efficiency of the system.

In addition to this, the datatype of each attribute has also been specified.

Furthermore, the date of birth input has been improved. Initially, the patients age was taken – which is a static input. Dr. Hussain suggested that the date of birth should instead be taken, and the age will be calculated using that value. As a result, this ensures that the patient's age is kept as current as possible as comparisons can be made regarding the patient's age yearly.

PSEUDOCODE FOR PATIENT CLASS

```
import random

class Patient

    private procedure __init__(self, forename, surname, dateOfBirth, nhsId, vaccineNo, covidCases,
    allergies, gender, ethnicity, address)

        private forename

        private surname

        private dateOfBirth

        private nhsId

        private vaccineNo

        private covidCases

        private allergies

        private gender

        private ethnicity

        private address

    public procedure nhsId(nhsId)

        if nhsId == 0:

            unique_id = []

        for x in range 0 to 10

            number_generator = random.randint(1,9)

            unique_id.append(number_generator)

            result = ""

            for i in unique_id

                result += str(i)

            unique_id = result

            return unique_id

    writeToFile(self, forename, surname, dateOfBirth, nhsId, vaccineNo, covidCases, allergies,
gender, ethnicity, address):

        record =
forename+surname+dateOfBirth+nhsId+vaccineNo+covidCases+allergies+gender+ethnicity+address

        file.write(record)
```

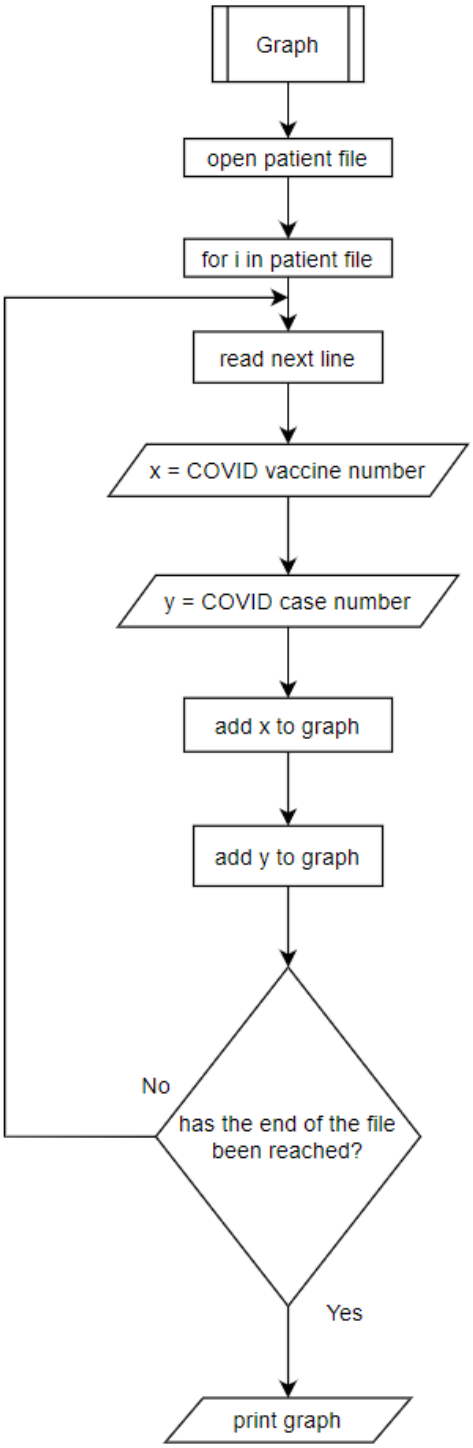
The following pseudocode is for the patient class. The code begins with a constructor method which takes in the attributes of a patient, which were previously inputted by the user. This essentially creates

the object for each patient. The use of a private method has been incorporated as a form of encapsulation, by preventing the original attributes from being amended.

A separate method called ‘nhsId’ takes in the inputted nhsId as a parameter. If this value is 0, this suggests that there has not been an nhsId provided by the user. Following this, the random function will be used to generate a random 10 digit number which forms the nhsId. This meets the success criteria of creating an NHS ID.

A final method joins the attributes together to create on record, and writes this to file. This meets the success criteria whereby the patient data is written to file.

FLOWCHART FOR SCATTERDIAGRAM



The final flowchart is a subroutine for the construction of the scatter diagram. Within this, the patient file will firstly be opened. Each line in the file will be read. Within each record, the number of COVID vaccinations a patient has had (x), as well as the number of COVID cases they have had (y) will be taken as x and y coordinates. Following this, the coordinates will be plotted onto the diagram. This process will continue until x and y coordinates have been derived from each record. Each time patient data is inputted, a new set of coordinates will be added to the diagram, thus providing an updated diagram for scientists to work from. Finally, after the patient file has been displayed on the screen, the scatter diagram will also be printed.

PSEUDOCODE FOR SCATTERDIAGRAM

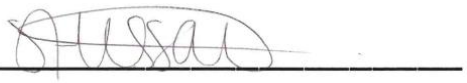
```
procedure graph
    for i in file
        x = pos[m]
        y = pos[n]
        diagram.plot(x,y)
```

The following pseudocode is for the scatter diagram. It checks through the file and reads each line/record. Each record imitates a list, and therefore m and n represent the position where the numbers of COVID vaccines, and COVID cases each patient has had are, which are assigned to variables x and y. M and n will be changed to actual values once the order of user inputs has been decided. Research will also be undertaken in order to determine how to plot the values on the scatter diagram. This is how the scatter diagram will form.

CONCLUDING THOUGHTS

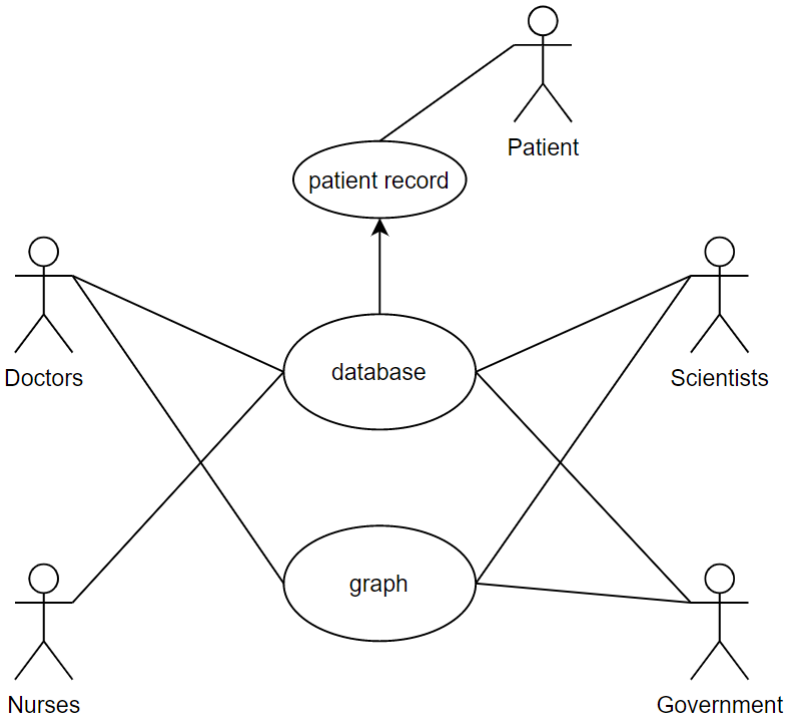
Overall, the flowcharts and pseudocode provided are highly abstracted, and form a very limited picture of what the final code will include. It was kept short because it is inly meant to represent the general flow of the program. However, extensive planning will be taken when programming the system, and factors such as validation will be taken into account. This feature is complicated as it will need to be done for each user input, and so it was not included as this section is not intended to be as detailed as the development section will be. Additionally, determining how each user input will be validated will require careful consolation with Dr. Hussain, which will occur during the iterative development phase. Therefore, more detailed features of the program will be accounted for whilst the code is being developed. Additionally, the testing of such algorithms can only occur in full code, as opposed to pseudocode, so will happen during development.

The simplified designs were also a key feature in order to ensure that Dr. Hussain was not bombarded with information about the proposed design. After consulting with Dr. Hussain about the pseudocode and flowcharts above, agreement was established, and his signature can be seen below:

Signed: 

USE-CASE DIAGRAM

The use case diagram below is a graphical representation of the interactions of the different users and elements of the system, which can be used in order to identify the system requirements. The reason why this has been added is because 11% of questionnaire respondents wanted user access levels as part of their security requirements.



Recognising that the database is highly private due to how it holds sensitive information explains why patients will not have access to the whole database. Having said this, a patient may be able to access their own record as it contains their own, personal data.

The jobs of nursers means that they are only required to access patient data. Additionally, analysing graphs is not a core part of a nurse's role, so the scatter diagram does not need to be seen by them. As a result, they will therefore only have access to the entire database – which is all that is necessary.

Scientists are only concerned with the scatter diagram as that is what they will analyse. However, if they see certain trends then they may wish to access patient data in order to investigate underlying causes of a disease, for instance, such as whether it's common in certain races or genders. This means that they are likely to require access to both the contents of the entire database, as well as the scatter diagram.

Initially, it was thought that doctors will only require access to the database. However, after consulting Dr. Hussain it became clear that doctors may also like to analyse trends in the scatter diagram to predict the incidence of certain disorders, for example. As a result, doctors will also have access to both the contents of the entire database, as well as the scatter diagram.

The government will also have access to both the database and scatter diagram as it is crucial that they monitor population health, and this entails having access to patient details. For instance, if after analysing the graph it is found that COVID-19 cases are rising drastically, policies may be set to increase vaccination rates, find more effective vaccines, or to enforce a lockdown.

After consulting with Dr. Hussain, his comments were as follows:

"The use case diagram is a good feature which clearly demonstrates the rights that different users will have. This is important as the sensitive nature of the data we deal with on a day-to-day basis cannot have unauthorised access. Additionally, it is nice to see the consideration of how the government may deal with data, as this is a factor that not many people consider. Overall, I am pleased with this concept."

After gaining Dr. Hussain's approval, there is increased confidence that the use of access levels will help meet the security requirement of 11% of questionnaire respondents.

The access levels are likely to be incorporated in relation to the options users have to select from. For instance, a scientist may have an option given to them where they can choose to display the contents of the scatter diagram, whereas a nurse would not have this option.

PROPOSED SCREEN DESIGNS AND USABILITY FEATURES

Ensuring that the program is accessible to a variety of users is important in providing a good user experience with the program. As a result of this, the number of individuals the program will be able to reach, and subsequently benefit will increase.

When developing the algorithms earlier, the use of Tkinter was not accounted for. This is explained further in the development section. However, essentially, whilst coding according to the algorithms above, I realised that I would need a GUI, and so the design section was referred back to as I realised that I would need to consider screen designs and usability features. In addition, the success criteria was amended. This change can be seen below:

Incorporate checks for addresses.	This ensures that the address given by a patient is authentic, allowing doctors to send letters to the correct address.
Create home screen.	The creation of a home screen will provide users with a range of access levels to choose from. This is important as it meets the security requirement of 11% of questionnaire respondents. Additionally, home screens are beneficial as they provide users with a small introduction into the system, without being bombarded with many commands and functions. This is important as it improves the usability of the system, meeting the need of 75% of questionnaire respondents who deemed having an easy-to-use software interface as important.
Create patient login screen.	This is important in meeting the security requirement of 16% of questionnaire respondents who deemed password protection as important. As a result, the system may have increased security as a login prevents unauthorised access to sensitive data.

Candidate Name: Farida Addo Candidate

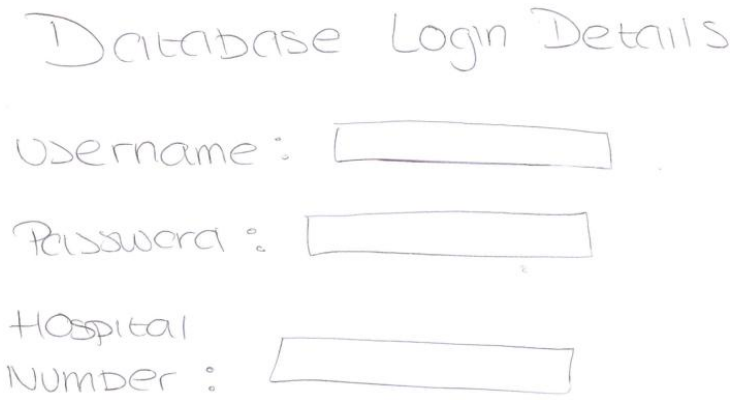
	Moreover, a login screen is essential for patients to access their data. Without it, it would not be possible to determine which patient is who, and ensure that the correct information is provided to the users.
Create login page.	This is important in meeting the security requirement of 16% of questionnaire respondents who deemed password protection as important. As a result, the system may have increased security as a login prevents unauthorised access to sensitive data.
Separate main menu for 'Nurse' access level.	This introduces the concept of an access level, which is important to 11% of questionnaire respondents as a security requirement. It is also needed to meet the use case diagram requirements. This is because, apart from the patient, the nurse is the only other access level without rights to the scatter diagram. This means that a separate main menu is required for them.

The screen designs also aimed to meet the success criteria.

PROPOSED SCREEN DESIGNS

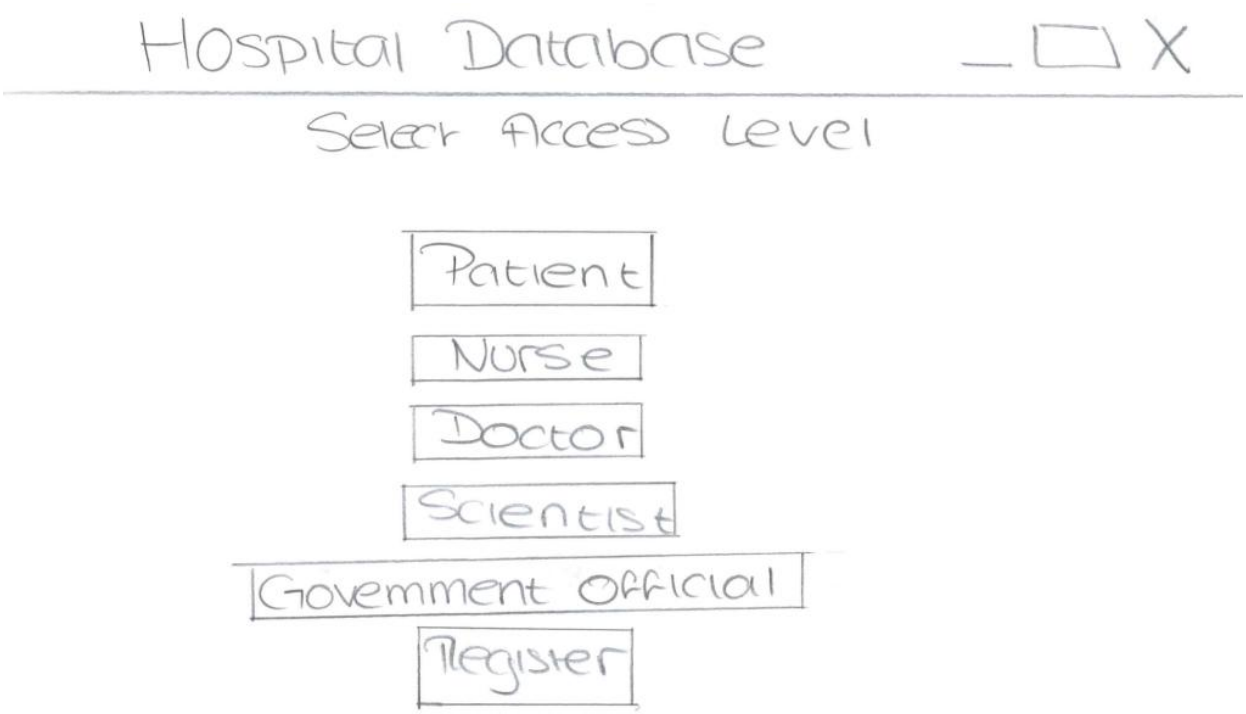
HOME SCREEN DESIGN

INITIAL HOME SCREEN DESIGN



This diagram above was the first prototype for the initial start-up page. There were benefits to this design as the entry of a hospital number would narrow down the records shown to users to a single hospital. After showing this to Dr. Hussain, he had some comments: “This design was is slightly impractical as it does not account for the fact that some users may not have a login. Furthermore, it does not consider the use of access levels which would help filter out the functions each user is able to do. For instance, I should be able to access more data than a patient.” In light of these problems, a second prototype for the initial start-up page was made, as shown below.

REFINED HOME SCREEN DESIGN



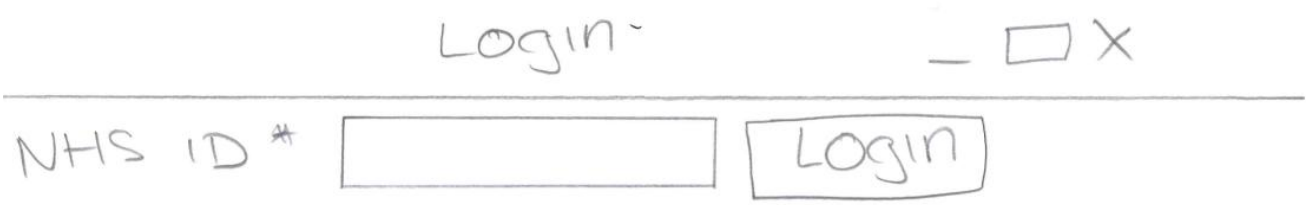
The diagram above represents the second, final prototype for the start-up page. It is an improvement from the previous screen as it accounts for different access levels, which represent this different

stakeholders, and gives users the chance to register if they do not have existing login details. The access levels generated were from the identified stakeholders, which were approved after consulting with Dr. Hussain. Furthermore, the above access levels were also chosen as they are the target end-users – who the program is being designed for. The user will have a range of options to choose from, which lead to different outcomes.

Using the use-case diagram as a guide, different user access levels will be implemented into the program. The reason for this is to ensure that security is maintained within the system, which was a security requirement for 11% of questionnaire respondents. For instance, a patient who solely needs to access their patient records should not have access to an entire database. With access to such sensitive information like this, a security threat may be posed if the data is misused. An example of this would be stealing credit card information and using it to purchase goods.

Overall, this screen design was created in a basic format which many people would be used to seeing, improving the usability of the system. Ultimately, the aim is to meet the success criteria of creating a home screen which is accessible to a range of users.

PATIENT LOGIN DESIGN

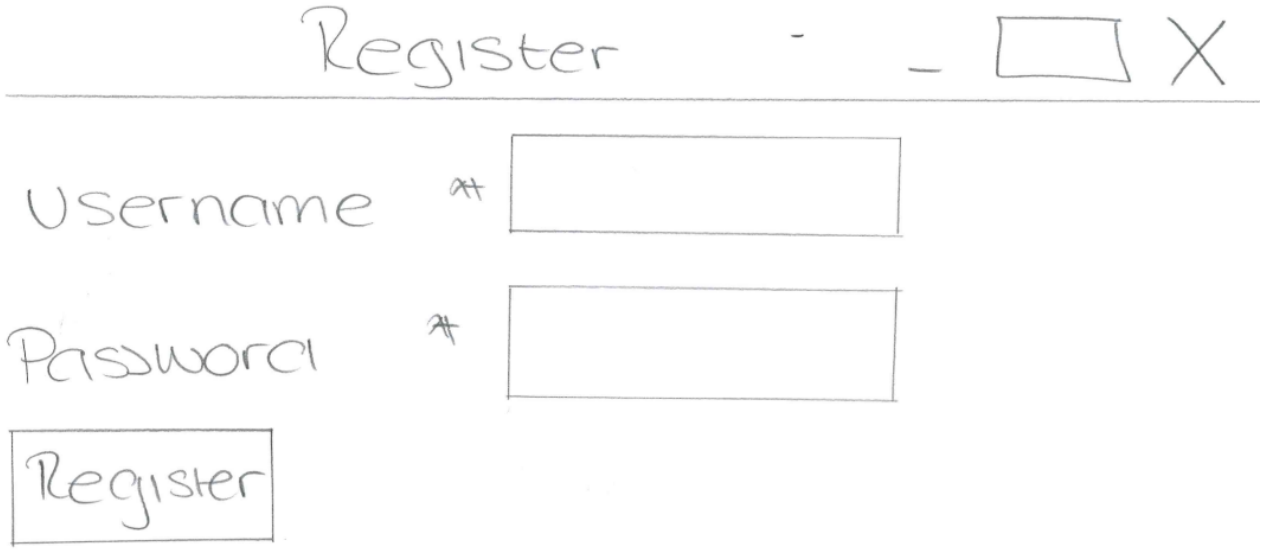


The diagram above represents the login screen for users who choose the patient option. The asterisk suggests that there must be an entry. Without an entry, the data cannot be accessed as the system will not know which NHS ID to provide details for, which is why it is important that the entries are not left blank. The asterisk features commonly throughout other login screen designs, indicating that a field is required. The asterisk is successful as it is a common indicator that an input is required. This will improve the learnability, and in turn usability of the program.

Users will only have to enter their NHS ID, where their record will then be displayed. The lack of entries available for input is because all that is required is an NHS ID to then find the user record. The requirement for any other entries would waste space, and would suggest that the database has partial dependencies (such as the requirement for both an NHS ID, and a particular date of birth before a record can be identified). This would be undesirable as the database should only hold one primary key (NHS ID) which uniquely defines each record. By keeping the interface and steps simple, the database will be accessible to a greater amount of people.

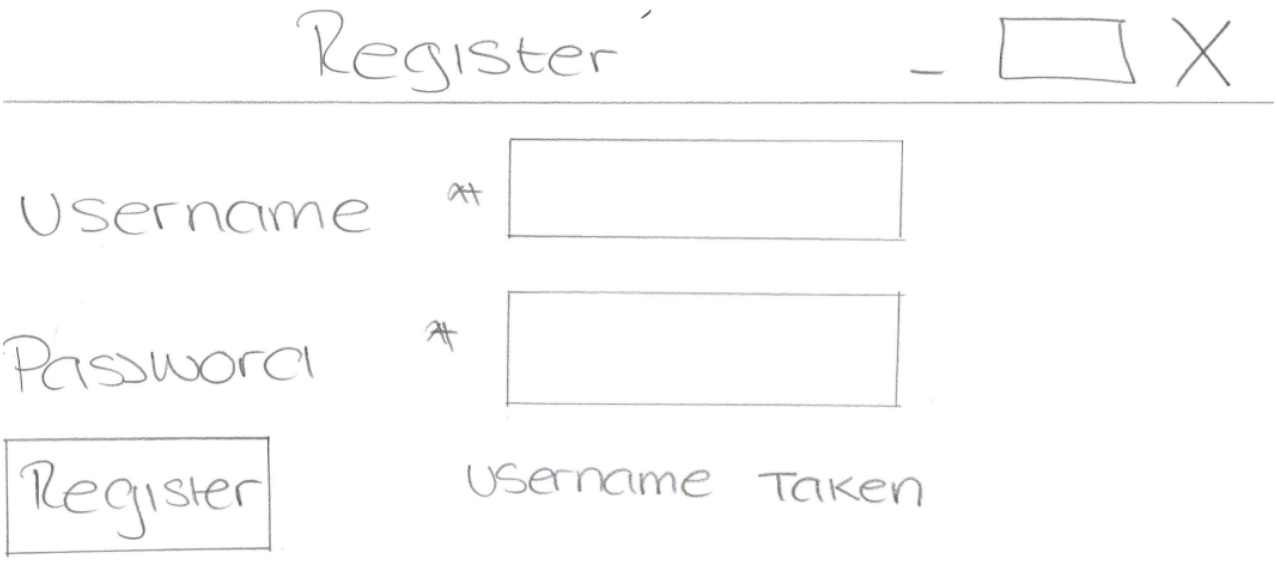
Overall, this screen design is intended to meet the success criteria of creating a patient login screen. All that is left is to implement this in relation to coding.

REGISTER DESIGN



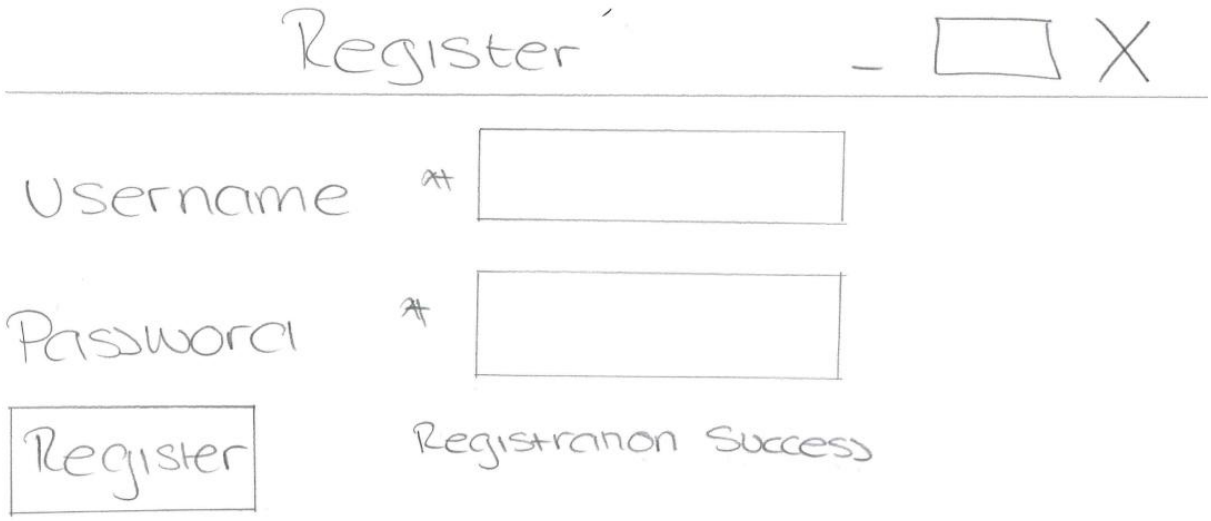
The diagram above represents the screen design for users who choose the register option. By allowing individuals to register new accounts, the database can be easily used by nurses and doctors as they can create new accounts if they don't have a log in. Having said this, it might be better for someone in administration to set the usernames and passwords as they could ensure that the passwords aren't easy to guess. Furthermore, some usernames may require a certain syntax which the administrator would be able to account for. The only entries required are a username and password. The reason for this is because it is a standard feature of login details. As expressed before, it is important that the system created is similar to applciations which users would have used in their daily lives, as it will improve the learnability of the system, thus improving its usability. Afterwards, users can click the register button where they will then be given a message, which is explained below.

USERNAME TAKEN SCREEN



The diagram above represents the error screen if the username has been taken. The reason for this is to ensure that multiple users are unable to use the same username. As a result, each user on the database can be uniquely identified. This also helps minimise security risks as it prevents others from posing as a trusted individual, which could result in hacking, and the theft of sensitive details.

REGISTRATION SUCCESS DESIGN



The diagram above represents the screen design for a registration which has been successful, where a success message will be outputted to the user. Following a successful entry, the entry boxes will be automatically cleared to allow another user to be registered. The reason for this is because it would take a while for users to manually clear this entry bar. This could be a source of frustration for some, so it is important that the user interface simplifies the complexity of the system as much as possible. This will therefore allow the system to have increased usability.

LOGIN DESIGN

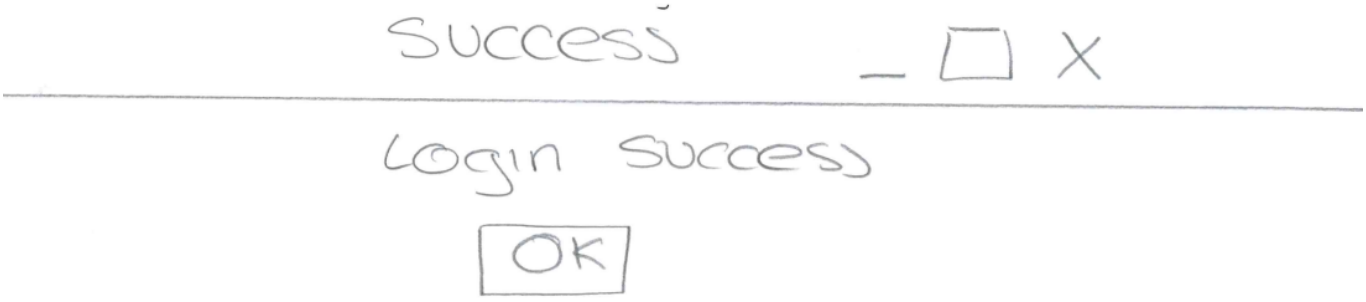
The diagram above represents the login screen design for a user who chooses any of the access levels apart from 'Patient'. This has been made different to the patient login screen as it was not deemed necessary by Dr. Hussain for a patient to have credentials on the system. This is because it would waste file space as it is unlikely that they would have regular contact with the system. As a result, the provision of an NHD ID is only required for users selecting the 'Patient' access level.

The functionality is similar to the register screen, where the user needs to enter a username and password. The requirement for both a username and password are to increase security. For instance, a hacker may be aware of a username, but without a password it cannot access the data. The password acts as an extra security layer and prevents unauthorised access to sensitive patient data.

When creating the screen design initially, the first thing a user would see was the initial home screen design. However, this was changed to the refined home screen design as users should not be able to login without selecting their access level. This is because each access level would go on to having different requirements for login inputs.

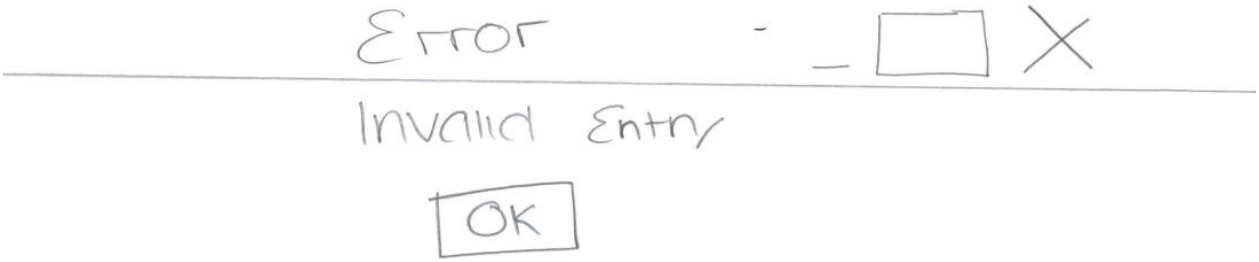
Having said this, this current login design will be compared to the initial home screen design as the initial home screen design was a login page. Looking at this design, it is clear that it is different to the initial home screen design, where a hospital number was required. Despite the fact that the addition of a hospital number would be beneficial, the reason for its removal is because it would take up extra file space, which would be unnecessary. Furthermore, the database system would be sold separately, and would therefore be unique to each individual hospital. As a result, there would be no need to include a hospital number as the username and password details stored would only be for each hospital regardless.

LOGIN SUCCESS SCREEN



The diagram above represents the screen design following the successful entry of a username and password. The reason for this is because it acts as an indicator that users will be able the move onto the next part of the program. It includes a simple design with a message, and an 'OK' button that the user can click to move onto the next screen.

INVALID USERNAME SCREEN

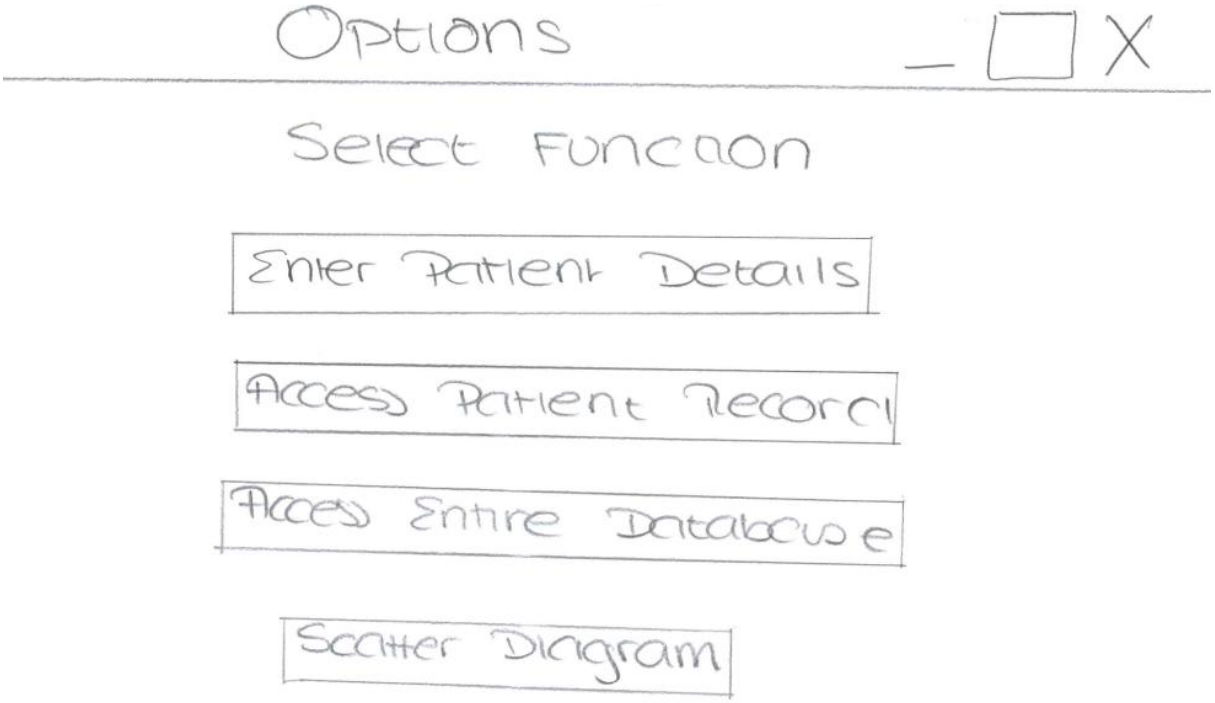


The diagram above represents the screen design for an unsuccessful login attempt. The reason for this is so users are aware that they have provided incorrect details. The provision of the message is also beneficial as after the incorrect entry of a username and password, the entries will go blank. As a result, the users need to be aware of why this has happened and why they have not been able to access the next part of the system. By adding this screen, users can go back and ensure they provide correct details.

CONCLUSION

Overall, this screen design is intended to meet the success criteria of creating a login page. All that is left is to implement this in relation to coding.

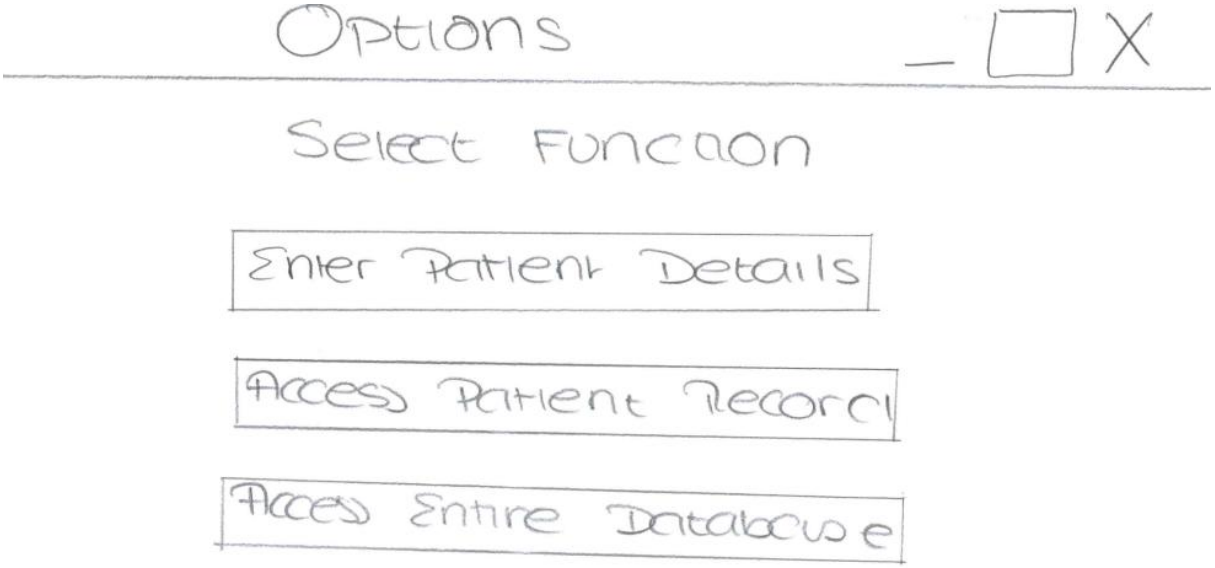
OPTIONS DESIGN



The diagram above represents the screen design for the user options. Once the use has successfully logged in, they'll be taken to the options screen, where they can choose what they would like to do. This

is where the use case diagram and user access levels are important. If the user chosen was 'Patient' then this page won't be displayed, instead they'll be given their record, such as the one displayed below.

The design implemented is a range of 'boxes' that users can choose from. This concept of users choosing from 'boxes' has been implemented consistently as it is important that users are not exposed to different designs, and that it is kept as simple as possible. In turn, the usability of the system will reach a wider range of users.



If a user chooses the 'Nurse' access level then they will be taken to a separate options page where the 'Scatter Diagram' option will not be shown. On the other hand, if the 'Doctor'. 'Scientist', or 'Government Official' users where chosen then the screen above will be displayed. The reasonings for this were displayed in the use case diagram section earlier.

This separate options page was created with the intention of meeting the succcess criteria where a separate main menu is created for the 'Nurse' access level. All that is left is to implement this in relation to coding this screen.

OUTPUT PATIENT RECORD DESIGN

Forename	Surname	NHS Number	Age	Number of VAccines	COVID-19 Cases
Farida	Addo	9131696573	17	Y	0

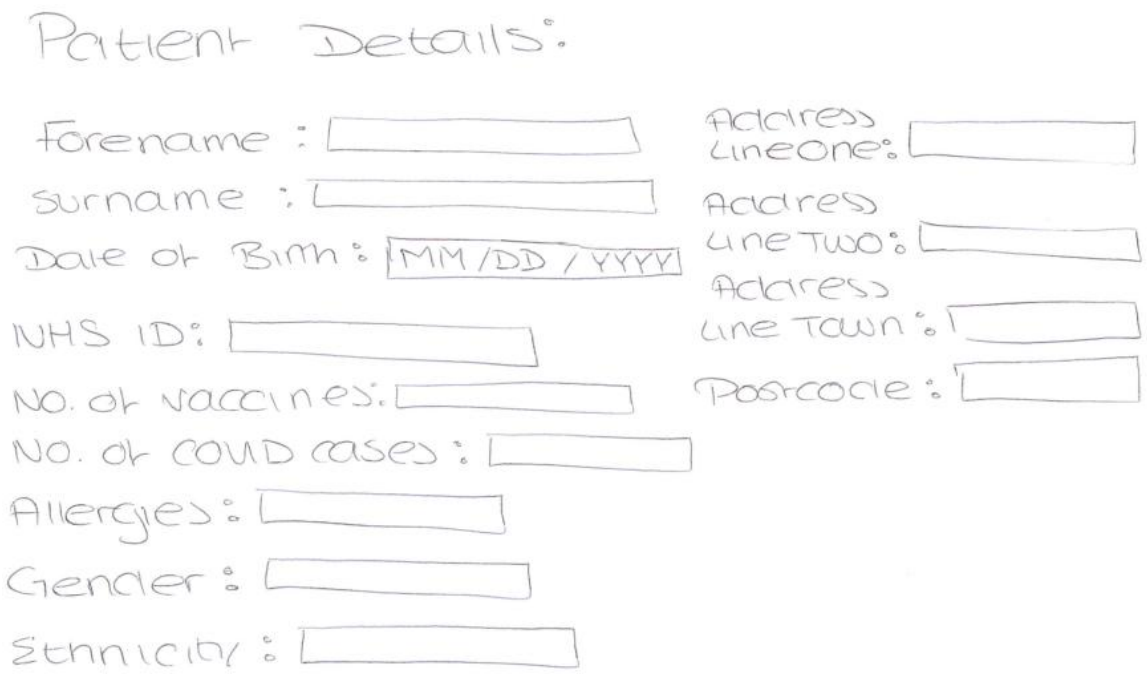
In addition to the diagram, a table will also be incorporated. The diagram being referenced is Figure 1, as shown in the analysis section.

This table will show the contents of a record when a patient has entered a valid NHS ID, or when a scientist, nurse of government official has entered a valid NHS ID after selecting the 'Access Patient Record' option. Thinking abstractly has played a fundamental role in creating the simple design of the

table. This simplicity is important in ensuring that the user can interpret every aspect of the table easily, thus allowing it to serve as a useful diagnostic tool.

INITIAL DATA INPUT DESIGN

Using the class diagram as a guideline, the initial prototype screen design for inputting patient details is displayed below:

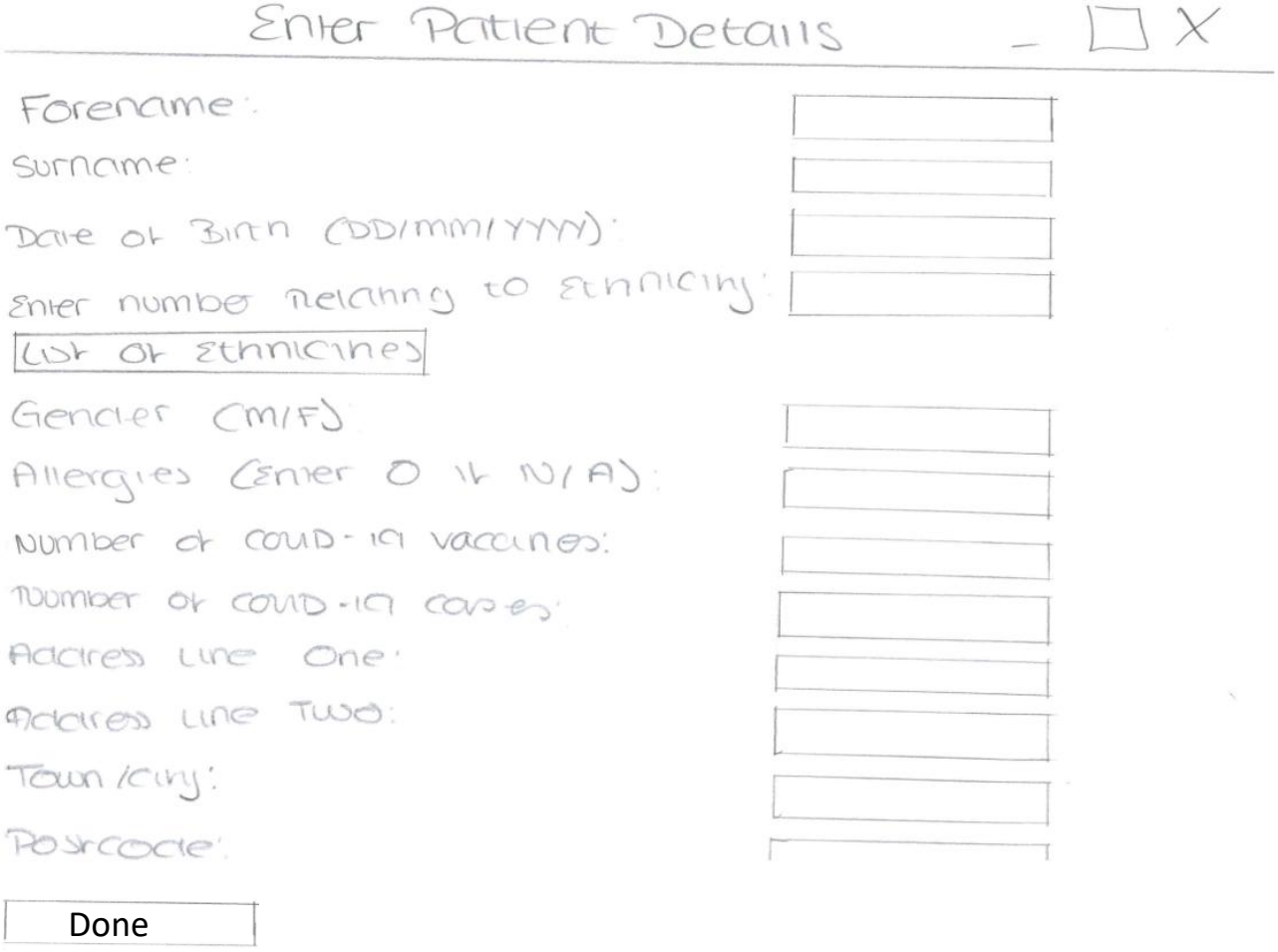


What was successful about this prototype was the simplicity. Simplicity is a feature which prevents users from being bombarded with unnecessary imagery which could deteriorate their experience with the system. This simplicity can also be seen in the finalised prototype below.

Another thing which was fundamental was formatting. For example, the format in which the date of birth should be inputted was clear to the user. The reason for this strict formatting was to ensure consistency within the database.

The format in which the date of birth is entered may vary from country to country. For instance, an American user will enter the month first, but a British user will enter the day first. However, the screen design specifically indicated that the month must be entered first. The strictness of this format ensures that the system can be used by users from varying cultures as they would know which format to write the date of birth in, even if it is not something they are used to. In turn, this increases the applicability of the system.

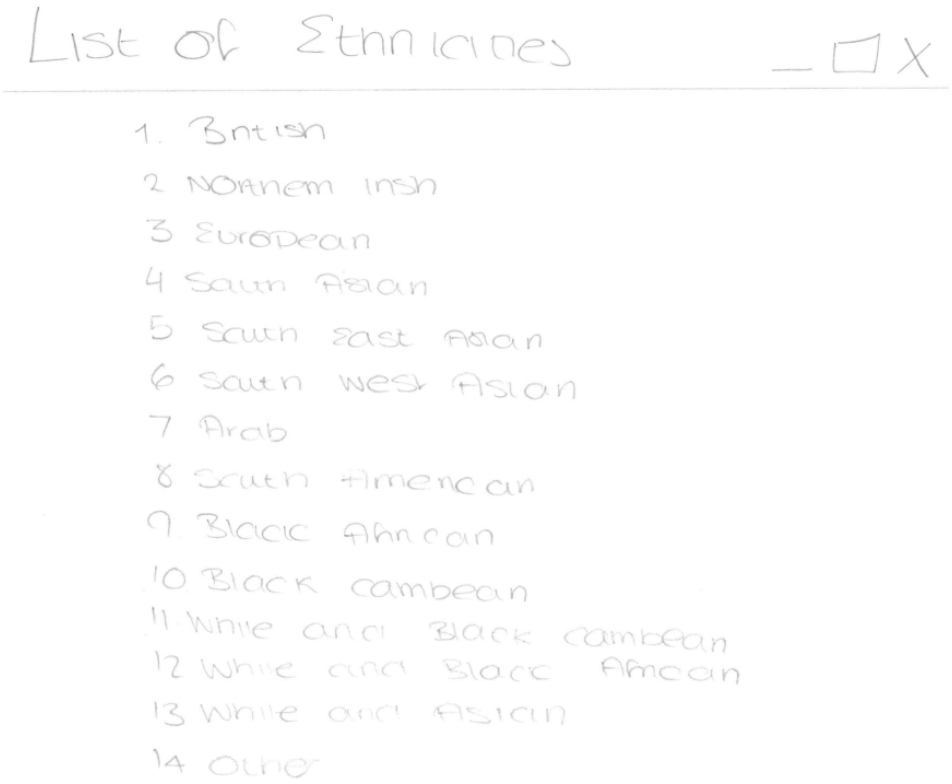
REFINED PATIENT INPUT DESIGN



The design above represents the second prototype for the screen which occurs if the user clicks the 'Enter Patient Details' option. The reason for the change is because this format is better structured. Having the inputs on both sides of the screen would create a visual mess. Furthermore, there has been a change to the ethnicity entry. For instance, users now have to enter a number from the box ('List of Ethnicities'). The reason for this is because ethnicities can have several different spellings and types so narrowing them down and standardising them helps maintain the consistency of the entries.

The screen design for the contents of the 'List of Ethnicities' can be seen below, where many of the ethnicities listed where determined through looking at the ethnicities listed on the gov.uk[3] website:

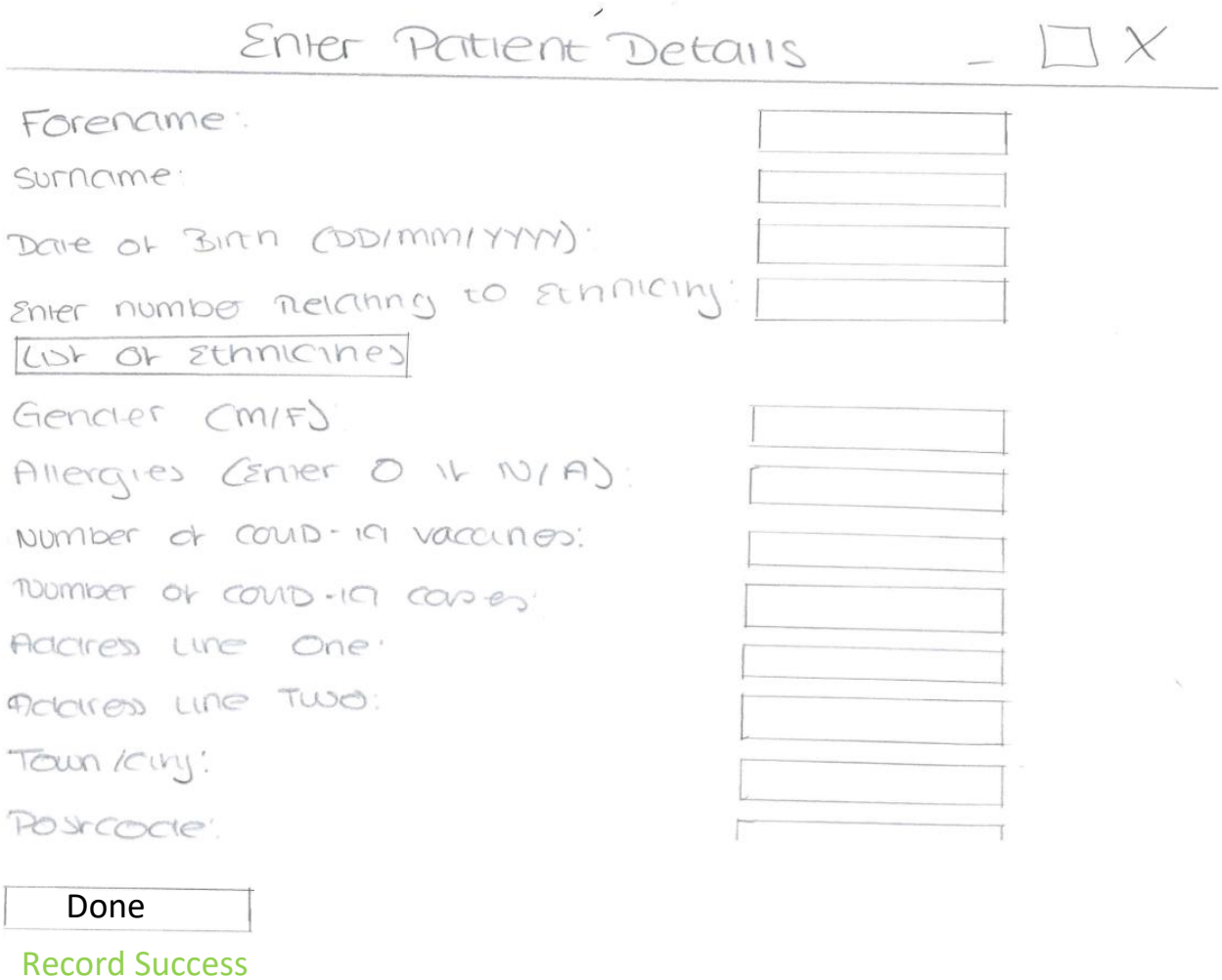
[3] https://www.ethnicity-facts-figures.service.gov.uk/style-guide/ethnic-groups



There is also the removal of the 'NHS ID' input. The reason for this is because, if a new patient is being added to the database, then the user will not be aware of their NHS ID as they do not have one. Instead, the patient's NHS ID will be randomly generated. This implementations of this do not need to be known to the user. All the user needs to know is that the final record will include the NHS ID, but how that has happened is not important. By doing this, the complexity of the system can be reduced, as well as unnecessary input entries.

Another change is the format in which the date of birth should be entered. After some investigation, it was found that the 'MM/DD/YYYY' date format is not commonly used, resulting in the change to a 'DD/MM/YYYY'. Once again, this enables the database to be accessible to a greater range of users. Furthermore, it is also beneficial as a key part of this system is ensuring that it is easy to use (which was deemed important by 75% of questionnaire respondents), and something which users can become familiar with quite quickly. When entering hundreds of data into the database, data entries should be made quickly. By ensuring that the format of the entries is something users are used to, this can increase the overall speed of the program as records can be saved quickly.

SUCCESSFUL RECORD ENTRY SCREEN



After entering record details in the correct format, the system will add the values into the database, output the message 'Record Success', and automatically clear the entry boxes. The reason for this is the same as stated before, it will increase the speed of working with the program as users won't have to manually remove entries – which is very time consuming. Once again, making the program easy to use for the users has been achieved.

UNSUCCESSFUL RECORD ENTRY SCREEN

On the other hand, if the user enters data which is not in the correct format, then the system will display an error message to the left of the specific entry box. This will enable data values to be changed to fit a specific format, enabling consistency to be maintained within the database. An example of an error is displayed below. The user is asked to enter 0 if the patient has no allergies. There is no indication that any other number should be used, so the user input of 3 has generated an error message, prompting the user to re-enter the data.

Gender (M/F): F

Allergies (Enter 0 if N/A): 3 Invalid Entry

USE OF COLOURS IN OUTPUT MESSAGES

The use of colours is also worth mentioning. For instance, 'Record Success' will be displayed in green as green is often associated with goodness – which in this case there is. If your record has been successfully added to the database, then you have done something well. On the other hand, 'Invalid Entry' will be displayed in red as red often has negative connotations. As a result, this will indicate to user that they have done something bad, which in this case they have. They have not entered the data in the desired format. As a society, we have automatically associate colours with meanings, so the use of the red/green colours means that a user will automatically know that they have done something wrong/right, and will automatically make any required changes. As a result, the use of colours is a feature which has improved the usability of the system.

ACCESS ENTRIE DATABASE DESIGN

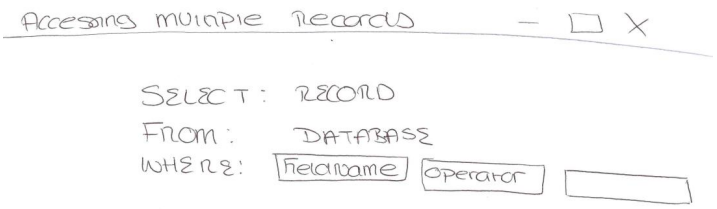
Forename	Surname	NHS Number	Age	Number of VAccines	COVID-19 Cases
Farida	Addo	9131696573	17	Y	0
Farida	Addo	6428473563	34	Y	0
July	Heather	5351331875	2	N	0
Sam	Johnson	9861212989	85	Y	6
Lola	Addo	3979872798	92	Y	34

The diagram above represents the screen design which will be shown if the user chooses the option 'Access Entire Database'. This tabular format will make the program easier to use as it can be easily understood. In addition, by keeping the format simple and not adding excessive, vibrant colours, the table can be viewed easily, thus improving user experience.

ACCESS MULTIPLE RECORDS DESIGN

INITIAL SCREEN DESIGN

The initial screen design for the 'Accessing Multiple Records' option can be seen below:



The statements 'SELECT' and 'FROM' lines will be pre-written onto the screen as they are unchanged labels. The reason why the user is defaulted to selecting a record is because only selecting a date of birth, for example, from each record would not be informative for a user. This is because the user would have no knowledge of who each patient is, as well as their allergies, which may be a more important factor in providing a user with a holistic knowledge of a patient.

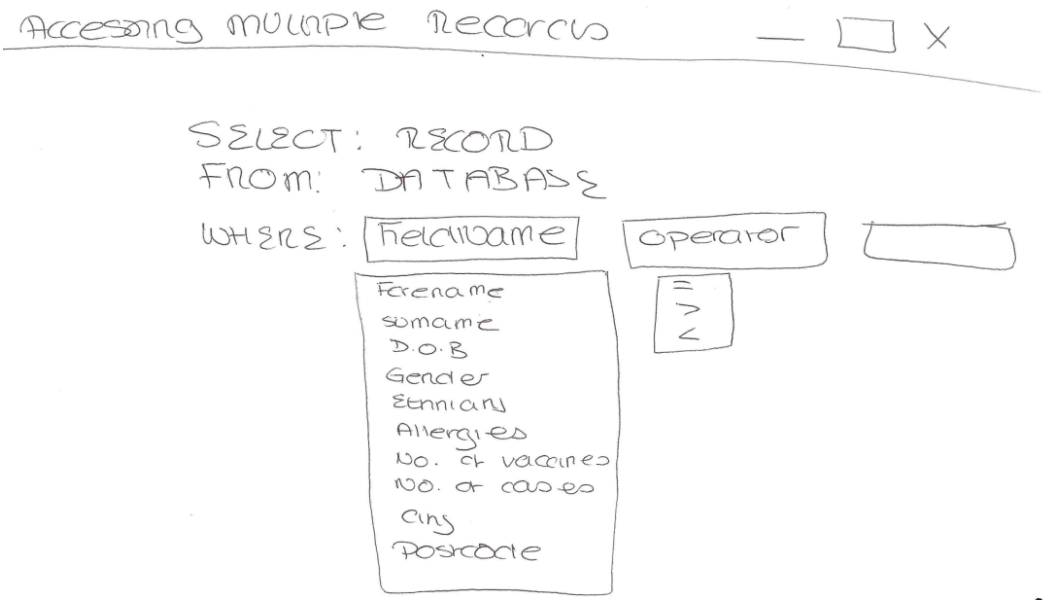
The 'WHERE' statement allows users to select both a fieldname, and operator. These features were derived from my knowledge on SQL as it allows users to determine specifically what they want to access.

A drop-down menu will be implemented in order to allow users to select which fieldname, or operator they want to use. The NHS ID is not a choice for fieldnames as if a user only wants to access a particular NHS ID, then they would only be able to access one record, as the NHS ID is unique to one patient. As a result, they should have selected the 'Access Patient Record' option.

Finally, a user will be given an entry box to enter what they would like to access in particular, i.e. where gender = 'F'.

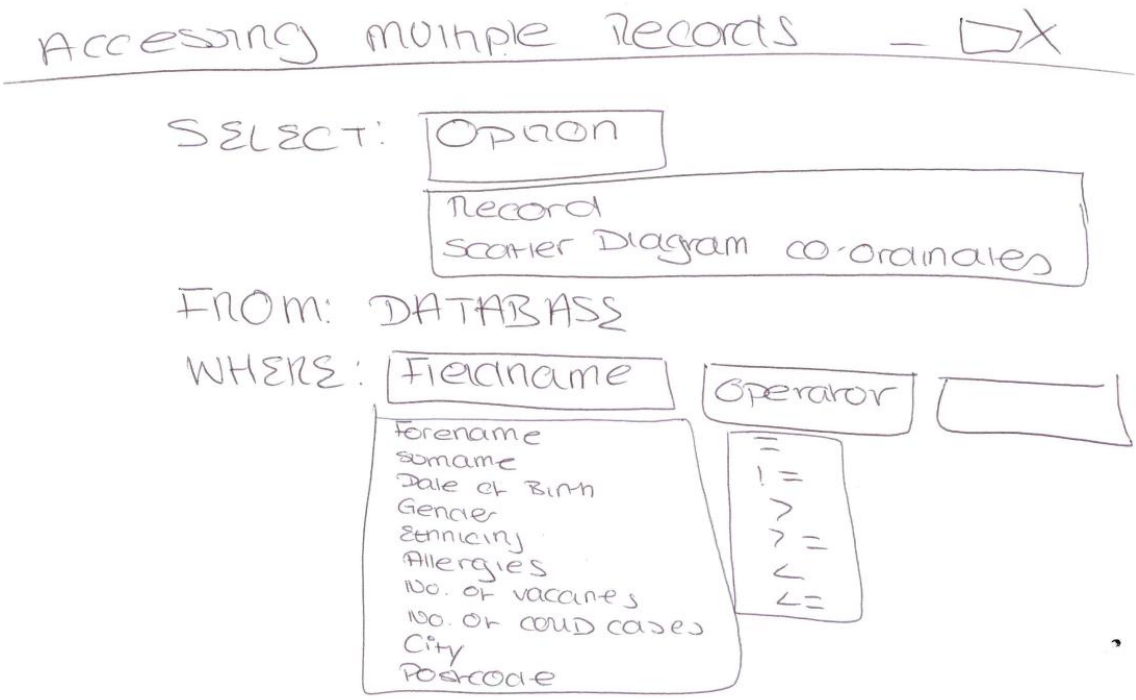
DROP-DOWN MENU SCREEN DESIGN

The screen design for the drop down menu can be seen below:



The ‘addressline’ fieldnames were removed as they are specific to one person. If a user wanted to see where an individual lived, they could use the ‘Access Patient Record’ option, as they would not need to access multiple records.

REFINED SCREEN DESIGN



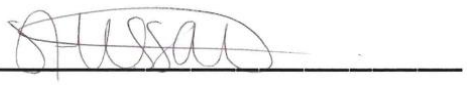
The screen design below represents the finalised design for the ‘Accessing Multiple Records’. After showing the designs to Dr. Hussain, he suggested that a user, such as himself, may want to see scatter diagram co-ordinate values for multiple users. As a result, the ‘select’ statement was amended in order to allow a drop-down menu to be incorporated – where the two options would be ‘Record’ or ‘Scatter Diagram Co-Ordinates’.

There have also been additions to the operator values that users can select from. This is because it is important that users are able to use a wider range of comparison values. Only being able to use ‘=’,’<’, or ‘>’ is very limited for the user, so it is important that they have high flexibility in the operations they want to perform within the system.

CONCLUSION

Overall, thinking abstractly is a recurring theme when deciding upon the screen design. It is this type of thinking which results in the program being suitable for all users. The screen designs have taken into account Dr. Hussain’s insights, as well as the questionnaire responses, as 75% of respondents deemed it necessary to have an easy-to-use interface. It is important to ensure that the system is being built in line with user requirements.

These screen designs were shown to, and approved by Dr. Hussain, who's signature can be seen below:

Signed: 

USABILITY FEATURES

It is important that the user interface is designed in a way which enables a range of users to interact with it easily.

BUTTON

A button is a common feature of a Graphical User Interface which provides the ability for users to trigger a certain event. Buttons will be embedded throughout the program to make it easier and more efficient for users to navigate through the system. For instance, after a user has entered patient details, they will be able to click a 'Done' button, which essentially converts their entries to records, which are then written to file. An example of this button can be seen below:

Buttons will also be added to allow users to pick a particular access level, where they will be taken to different main menus depending on their choice.

By implementing Tkinter, at the top right of each screen, 3 symbols will be present:

These 3 symbols are also 3 different buttons. They are important features as it means that users will be able to minimise, expand, or close a screen, which will improve the overall usability of the system as it provides users with greater functionality.

ENTRY BOXES

Entry boxes will also be incorporated into the program:

These are a fundamental feature of the program as it means that users will be able to enter data, which can then be interpreted by the system, and be utilised for different functions. For example, a user will be

able to enter their login details. This is important as without entry boxes, it will be difficult to verify a user's credentials, which could in turn pose a security threat to the data being stored in the database.

Keeping the entry box design simple is also important as it should be neutral enough for those who may have a problem with viewing bright things. It is also beneficial as someone who has never interacted with the system before will automatically be aware that the entry box must contain some kind of data due to being exposed to several entry boxes which look the same, in their daily computer operations.

LEARNABILITY

One feature incorporated into the system is learnability. As seen in the screen designs, the interface has a clear and consistent layout, which increases its usability and interacting with the software. This is important as the program should not be complex in a way in which prevents users from being able to utilise the systems functionality.

MEMORABILITY

Another feature is memorability, which refers to the ability for a user to return to the interface despite not having used it for a while. The use of Tkinter, where users solely need to click a button to perform a task, as well as the entry boxes indicative of user inputs, mean that a user with no experience of technology may be able to easily adapt to using the system. This means that the system will have high memorability as the use of buttons and entry boxes are all features seen in everyday computer tasks. As a result, long periods of time should not affect how easy it is for a user to interact with the system. This is crucial as users should not have to invest high amounts of time in re-learning the different outputs produced by certain user commands.

EFFICIENCY

Another feature is efficiency. It is important that the user is able to perform tasks in the least number of steps possible. The lack of menus to navigate between increases the efficiency of the program. This is an important aspect as a user should be able to perform tasks as quickly as possible. For instance, a doctor, needing to know about the allergies of a particular patient before administering them life changing medicine, should be able to access a patient record in the least possible time frame.

CUSTOMISABILITY

While the user may be unable to choose the font, or colours of the system, the requirements of all users have been considered in the final screen design. For instance, a font which strikes a balance being too small or big will be implemented. Furthermore, a black font colour will be implemented as black is a colour which often results in few viewing issues. Finally, in relation to displaying user records, the font Calibri will be used as it is a common font seen by most individuals. Using a cursive font may be difficult to read for some, perhaps acting as a source of frustration for those trying to interpret data. This may

prevent users from interacting with the system, slowing down system performance, reducing the efficiency, and usability of the system.

KEY VARIABLES AND VALIDATION

Having created the pseudocode for the program, the key variables and validation can now be identified. Validation is important in ensuring that all inputs can receive an appropriate outputted message. In order to create the validation features, the data type must first be identified, so it can then be recognised when an input breaks the rules of the syntax. By adding this feature, the code can be optimised to run quickly as it will not stop after encountering an input which is not desired. As the program is developed, the key variables and subroutines may be refined, changed, and added to.

As well as the key variables and validation, the time module will be imported in the program. This is in order to randomly generate and NHS number which is unique for each patient.

INITIAL VARIABLES AND VALIDATION

This table below reflects the variables and validation which were intended for the original code which did not implement the use of Tkinter. During the development of the program where the use of Tkinter was established, the required variables and validation changed – which can be seen in the following section. Having said this, many of the same variables identified below were still used as they were still applicable.

Name	**Type (Variable or Subroutine)**	**Description**	**Data Type**	**Validation**	**Scope**
Main Program					
option	Variable	Used to determine whether the user will access a record, or add a new record.	Integer	Must be either 1 or 2. The reason for this is because any other number is not assigned to a function. Therefore, the user should enter one of the two numbers so the system can provide the required response.	Global variable
entering	Subroutine	Allows the record to be created for x	Procedure	In order for the subroutine to be	N/A

		number of patients		accessed, the user input must have been 1.	
number	Variable	Used to store the number of patients the user would like to enter for	Integer	Must be an integer. An integer is the only way to determine the number of something, any other input would not allow the number of patients to be identified.	Local variable
records	Subroutine	Prints the patients record	Procedure	In order for the subroutine to be accessed, the user input must have been 2. Any other input would have pointed to a different subroutine.	N/A
id0	Variable	Allows the user to enter the NHS ID of the patient's record they would like to access	Integer	Must be a sequence of 10 numbers. This is because this is the standard format for an NHS ID.	Local
patients	Variable	Opens the patient file for read access only	File object	Text file must be present in memory to be opened.	Local variable
patient_file	Variable	Opens the patient file for appending	File object	Text file must be present in memory to be opened.	Global variable
html_file	Variable	Opens the html file for writing	File object	HTML file must be present in	Global variable

				memory to be opened.	
uniqueid	Subroutine	Returns NHS ID	Function	In order for the unique ID to be accessed, the user must have inputted "Y".	N/A
unique_id	Variable	NHS ID	String	Must be 10 digits in order to follow the format of a normal NHS ID.	Local variable
number_generator	Variable	Generates a random number between 1 and 9	Integer	No validation required – number is between 1 and 9	Local variable
result	Variable	Adds each digit in the NHS ID into a string	String	NHS ID must have 10 digits. It will not convert this into a string otherwise as the NHS ID would not have been created.	Local variable
Patient Class					
patient_name	Variable	Creates a patient forename	String	Must consist of letters in the alphabet. Names should not contain any other special characters.	Local variable
patient_surname	Variable	Creates a patient surname	String	Must consist of letters in the alphabet. Names should not contain any other special characters.	Local variable

age	Variable	Creates a patient age	Integer	Must be between 1 and 100 as it is unlikely that a patient is above 100.	Local variable
uniqueid1	Variable	Determines if patient has an NHS ID	String	Must be “Y” or “N” so the system can provide the correct response.	local variable
vaccines	Variable	Determines the number of COVID vaccines a patient has had	Integer	Must be between 0 and 10 as anything exceeding 10 may not be possible for doctors to administer due to health reasons and side effects.	Local variable
covid	Variable	Determines the number of COVID cases a patient has had	Integer	Must be between 0 and 10. It is unlikely a patient has had more than 10 COVID-19 cases.	Local variable
table	Variable	Creates table which is then written to file	Record	Variables used should be validated beforehand so no validation is required at this stage.	Local variable
questions	Subroutine	Returns table	Function	Variables in table should be validated beforehand so no validation is required at this stage.	N/A

REFINED VARIABLES AND VALIDATION

The table below includes the additions to the key variables and validation following the use of Tkinter during the development section. Also, when implementing the variables, I intend to use a consistent camel case as a constant format improves the readability of the variable names, and makes them easier to interpret as each new capital letter represents a new word.
Another change was to the input values for the 'Enter patient details' option. Initially, these would be created and validated in the patient class python file. However, now it will be done in the main program. The patient class python file will only focus on adding the field values together and writing it to the 'Patients' file.

Name	Type	Description	Data Type	Validation	Scope
Main Program					
delimiter	Constant	Used to add the field values to create a record separated by commas.	String (",")	N/A	Global
existenceOfFile	Boolean	Used to determine if the Patients file exists. Will create the file if not.	Boolean	Must be true as each record requires a storage space. If not, the file is created.	Global
numberOfLines	Variable	Used to store the number of lines (records) in the patient file. This will be important when creating a screen for outputting the entire database, for example.	Integer	N/A	Global
userNotFound	Subroutine	Used to design a popup for the value not found, whether that value is the incorrect username/password, or a record which has not been found (assuming that they have entered an invalid NHS ID).	Subroutine	N/A	N/A
title	Constant	Used to store the title for the screens displayed.	String	N/A	Local
message	String	Used to store the text for the label error message displayed to the user.	String	N/A	Local

plot	Subroutine	Used to create scatter diagram.	Subroutine	N/A	N/A
noOfVaccines	Variable	Used to store the number of vaccinations element from each record.	List	Must be an integer. For instance, the first line of the patients CSV file will be the field names, and so the element containing the number of vaccines will be a string. As a result, this first line should be skipped.	Local
noOfCovidCases	Variable	Used to store the number of covid cases element from each record.	List	Must be an integer as the number of COVID-19 cases would never take a Boolean value, for instance.	Local
repetitions	Variable	Determines the amount of time each co-ordinate appears.	List	N/A	Local
mainScreen	Subroutine	Creates main screen which will display access levels.	Subroutine	N/A	N/A
accessLevel	Variable	Used to store the access level selected by the user. Important as it will determine the main menu shown to the user.	String	N/A	Global
mainMenu	Variable	Stores the screen created for user options.	N/A Toplevel screen	N/A	Global
username	Variable	Stores the username input provided by the user.	String as a username may consist of a range of special characters.	If the username is taken, it cannot be used. It is important that uniqueness is maintained within the	Global

				system in relation to usernames. The username also cannot be empty, and cannot contain spaces as it is a common form of validation.	
password	Variable	Stores the password input provided by the user.	String as a password may consist of a range of special characters.	The password input cannot be empty. Also, cannot contain spaces as it is a common form of validation.	Global
invalidRegistrationOutput	Subroutine	Creates the screen when a user has not been found.	Subroutine	Subroutine will be pointed to when the user has provided an incorrect username and/or password.	N/A
usernameVerify	Variable	Stores the user input when a patient record is to be accessed.	String	NHS ID entered must exist in the database, or an error message will be outputted to the user.	Global
options	Variable	Stores the option value (scatter diagram co-ordinates or record) for users clicking the 'access multiple values' option.	List	N/A	Global
fieldNames	Variable	Stores the field name options for users clicking the 'access multiple values' option.	List	N/A	Global
fieldChoice	Variable	Stores the field name chosen by users from the drop down menu from clicking the	String	N/A – there are options for the users to choose from	Global

		'access multiple values' option.			
operators	Variable	Stores the operator options for users clicking the 'access multiple values' option.	List	N/A	Global
operatorChoice	Variable	Stores the operator chosen by users from the drop down menu from clicking the 'access multiple values' option.	String	N/A – there are options for the users to choose from	Global
inputEntry	Variable	Stores the input of the user who has selected the 'access multiple values' option.	String	N/A – the entry will be checked and whether or not the scatter diagram co-ordinates or record values are found will be displayed to the user.	Global
patientDetails	Variable	Used to create the screen that will display the labels and entry boxes for the 'enter patient details' option.	N/A – Toplevel screen	N/A – validation will be for each individual input.	Global
forename	Variable	Used to store the forename input provided by the user who selects the 'enter patient details' option.	String	Must be at least 2 characters as a forename is generally at least 2 characters long. This can be alphanumeric as names take different forms nowadays.	Global
surname	Variable	Used to store the surname input provided by the user who selects the 'enter patient details' option.	String	Must be at least 2 characters long as a surname is generally at least 2 characters long. This can be	Global

				alphanumeric as names take different forms nowadays.	
dateOfBirth	Variable	Used to store the date of birth input provided by the user who selects the 'enter patient details' option.	String	Day has to be from 1 to 31 as there are no more than 31 days in the month. Month has to be between 1 and 12 as there are no more than 12 months in a year. Using the 'strptime' function in python only limits the year to being 4 digits. While it would be better if the year could not be longer than the current year the user is using the system, this is a constraint which cannot be overcome.	Global
gender	Variable	Used to store the gender input provided by the user who selects the 'enter patient details' option.	Character	Must be either 'M' or 'F'. there is no need for the full 'Male' and 'Female' terms to be stored as it would waste storage.	Global
ethnicitiesList	Subroutine	Used to create the screen displaying the ethnicities for users to choose from.	N/A – Toplevel screen	N/A	Global
ethnicity	Variable	Used to store the ethnicity input provided by the user who selects the	Integer as the user will select a number	Number must be a whole number between 1 and	Global

		'enter patient details' option.	from the 'List of Ethnicities' screen.	the digit of the last ethnicity displayed. It cannot be more than the maximum digit as this does not correspond to an ethnicity, and thus an ethnicity cannot be derived from this for the record.	
validatedEthnicity	Variable	When a user enters a number from the 'List of Ethnicities' screen, this number will be converted into the actual ethnicity name.	String	N/A – the value would already be validated at this point.	Global
allergies	Variable	Used to store the allergies input provided by the user who selects the 'enter patient details' option.	Integer or String	If a patient has no allergies, 0 should be entered and no other number. Entering 1 or 4 without stating what the allergies are would not be of use to a doctor. On the other hand, if a user has not entered a number then a blank input cannot be entered (as is the case with the other inputs), any string cannot contain a number as alphanumeric words do not constitute to an	Global

				allergy. Finally, the length of the allergy has to be at least 2 characters as generally, there are no allergies less than 2 characters long.	
numberOfCovidVaccines	Variable	Used to store the number of covid vaccines input provided by the user who selects the 'enter patient details' option.	Integer	The only validation is that the number cannot be negative or decimal. This is because there is no such thing as having -1 or 3.4 number of vaccines, for example.	Global
numberOfCovidCases	Variable	Used to store the number of covid cases input provided by the user who selects the 'enter patient details' option.	Integer	The only validation is that the number cannot be negative or decimal. This is because there is no such thing as having -1 or 3.4 number of covid cases, for example.	Global
addressLineOne	Variable	Used to store the address line one input provided by the user who selects the 'enter patient details' option.	String	The address name can be alphanumeric, but has to be at least 2 characters long as it is unlikely that there is an address name of less than 2 characters.	Global
addressLineTwo	Variable	Used to store the address line two input provided by the user who selects the	String	The address name can be alphanumeric, but has to be at	Global

		'enter patient details' option.		least 2 characters long as it is unlikely that there is an address name of less than 2 characters.	
addressLineTown	Variable	Used to store the address line town input provided by the user who selects the 'enter patient details' option.	String	The town name can be alphanumeric, but has to be at least 2 characters long as it is unlikely that there is a town name of less than 2 characters.	Global
postcode	Variable	Used to store the postcode input provided by the user who selects the 'enter patient details' option.	String	It is hard to determine what validation to include for postcodes as they vary substantially. As a result, the only validation is that it must begin with a letter, as this is common across all postcodes. Additionally, the postcode has to be between 2 and 7 characters long – as this is the minimum and maximum length. The orders of letters and numbers vary so this will not be a form of validation.	Global
detailsVerify	Subroutine	Used to collect the input values from the entry boxes after a	Subroutine	N/A – user inputs will be validated in	N/A

		user has selected the 'Entire Patient Details' option and validate these by passing each input into different validation subroutines. At the end, the validated variables will be passed into the patient class.		their separate subroutines.	
submitDetails	Subroutine	Used to pass the record values into the patient class subroutine to be written to file.	Subroutine	N/A – values will already be validated.	N/A
Patient Class					
__init__	Constructor method	Assigns each value from the user inputs (assuming the 'Enter Patient Details' option has been selected) to a variable.	Method	N/A – values will already be validated.	N/A
forename	Attribute	Used to store the forename input provided by the user who selects the 'enter patient details' option.	String	N/A – value will already be validated.	Global
surname	Attribute	Used to store the surname input provided by the user who selects the 'enter patient details' option.	String	N/A – value will already be validated.	Global
dateOfBirth	Attribute	Used to store the date of birth input provided by the user who selects the 'enter patient details' option.	String	N/A – value will already be validated.	Global
gender	Attribute	Used to store the gender input provided by the user who selects the	Character	N/A – value will already be validated.	Global

		'enter patient details' option.			
ethnicity	Attribute	Used to store the ethnicity input provided by the user who selects the 'enter patient details' option.	String	N/A – value will already be validated.	Global
allergies	Attribute	Used to store the allergies input provided by the user who selects the 'enter patient details' option.	String (either 'N/A' or the allergies will be stored)	N/A – value will already be validated.	Global
numberOfCovidVaccines	Attribute	Used to store the number of covid vaccines input provided by the user who selects the 'enter patient details' option.	Integer	N/A – value will already be validated.	Global
numberOfCovidCases	Attribute	Used to store the number of covid cases input provided by the user who selects the 'enter patient details' option.	Integer	N/A – value will already be validated.	Global
addressLineOne	Attribute	Used to store the address line one input provided by the user who selects the 'enter patient details' option.	String	N/A – value will already be validated.	Global
addressLineTwo	Attribute	Used to store the address line two input provided by the user who selects the 'enter patient details' option.	String	N/A – value will already be validated.	Global
postcode	Attribute	Used to store the postcode input provided by the user who selects the 'enter patient details' option.	String	N/A – value will already be validated.	Global

generateNHSId	Method	Used to generate the NHS ID.	Method	N/A	N/A
nhsID	Variable	Used to store the generated NHS ID.	List	Must be 10 digits long, as it the standard NHS ID length. Also cannot be a pre-existing value.	Local
numberGenerator	Variable	Used to create a random number for the NHS ID which will be added to the list.	Integer	N/A	Local
newNhsId	Variable	Used to store the NHS ID as a string.	String	N/A	Global
saveRecord	Method	Used to create the record.	Method	N/A	Global
record	Variable	Adds each user input wit ha delimiter after each input.	Record	N/A	Global

TEST DATA FOR DEVELOPMENT

PRE-DEVELOPMENT TESTING TABLE

In order to test the behaviour of the system in response to various inputs, a test table will be created. The test table will also show the expected outputs to given inputs. Through this process of thinking ahead, the functionality of the program can be tested. Testing will be undergone continuously as the program is developed. A black box testing approach will be taken as test data is created in in line with the success criteria made previously. The test data below will ensure that this criteria is met.

Test	**Reason**	**Test Data**	**Expected Outcome**	**Type of Test**
Inputting alphanumeric characters when entering patient details	For some field values, alphanumeric characters are invalid inputs. Therefore, it is important that the program knows where to point to, following this input.	Y3S	The program will output “Invalid input. Please re-enter response.”	Erroneous for some inputs as an integer is used with letters.
Inputting an unrealistic	Realistically, a patient should not	1000	The program will output “Number	Erroneous as the data is outside a realistic range

number of COVID vaccines when entering patient details	have had over 5 vaccinations at this point in time. It may be too dangerous, so it is important to ensure that a patient has received a safe, and realistic number of vaccinations.		of vaccinations out of bounds. Please re-enter number of vaccinations."	
Entering an invalid NHS ID	Ensures that the program is able to read a separate text file and check the user input against it	Robot	The program will display an appropriate error screen message showing that the user has not been found.	Erroneous as the NHS ID is invalid
Entering a date of birth after the current date when entering patient details	Program must ensure that the date of birth entered is correct	03/05/2024	The program will output "Invalid birthdate. Please ensure date of birth precedes current date"	Erroneous as the date of birth exceeds the current time period
Inputting a valid age	Tests to ensure that valid details are acquired in the system.	16	Error message will be displayed.	Normal
Inputting a fake postcode when entering patient details	This ensures that the address given by a patient is authentic, allowing doctors to send letters to the correct address.	ABC DEF	The program will output "Invalid postcode. Please re-enter Postcode"	Erroneous as the data is not in the correct alphanumeric form
Checking to see if a randomly generated NHS ID is the same as a current one	This ensures that multiple people do not have the same NHS ID, as it could lead to the wrong identification of patients.	While this may only occur rarely, the generator may output "9131696573", which is already occupied	The generator will regenerate anther NHS ID, the user will not be notified of this whole process.	Boundary as it tests the bounds of the pre-existing data
Entering a valid NHS ID	Meets the success criteria which aims	An NHS ID which exists	The program will output the record	Normal as the input is expected

	to enable the record of a patient's data to be outputted	within the database.	for the specific patient.	

DEVELOPMENT

STRUCTURE OF THE SOLUTION

The following section will focus on coding the features described in the design section, as well as meeting the success criteria defined earlier. The main elements of the program are to display a scatter diagram, enable users to enter patient details which will create a record for the patient, and enable a single record to be accessed. Whilst designing the code, different user access levels will mean that certain features are restricted to certain users, thus enabling security of data. In turn, this will meet the requirement of 11% of questionnaire respondents who deemed having user access levels as a security requirement. Finally, these sections will be coded separately, and then brought together to complete the final project. The reason for coding the modules separately is to ensure that dedicated time can be spent on one module at a time, enabling the code to be efficient in the end. Having said this, an agile iterative development process will be undertaken, so any separately coded modules may be refined later on.

Initially, when designing the code for the program, Tkinter was not used as it was not something that I was familiar with. Later on in the coding process, I quickly realised that the program was being developed without a graphical user interface (GUI). This was problematic as it meant that I was unable to replicate the designs made before, which is a fundamental aspect of the system. As a result of this, much of the code had to be redone as the syntax for Tkinter is quite different. The reason for this change is because after having several users (such as Dr. Hussain), with no experience of Python, test the program, it became quite clear that the usability of the program was poor as it was not an interface that they were used to. An example of how the data was presented is displayed below.

```
Enter Patient's forename: farida
Enter Farida's surname: addo
Enter Farida's date of birth (DD/MM/YYYY): 25/08/2004
Enter Farida's gender (M/F): f
['1. British\n', '2. Northern Irish\n', '3. European\n', '4. South Asian\n', '5. South East A
sian\n', '6. South West Asian\n', '7. Arab\n', '8. Black African\n', '9. Black Carribean\n',
'10. White and Black Caribbean\n', '11. White and Black African\n', '12. White and Asian\n',
'13. Any other ethnic group']
Choose the number matching Farida's ethnicity: 2
How many allergies does Farida have? 0
Enter the number of COVID-19 vaccines Farida has had: 0
Enter the number of COVID-19 cases Farida has had: 0
Enter Farida's first address line: tertrtrt
Enter Farida's second address line: rttrrt
Enter Farida's town/city name: trrtr
```

Overall, the interface failed to provide a good user experience with the program. Having said this, the use of Tkinter has been integral in constructing an easy-to-use user interface.

The following sections will underpin how separate modules for the program were created using Tkinter, the problems I faced, and how I overcame them. There will also be testing done to ensure that the program is producing the desired outputs.

IMPORTING LIBRARIES

In order to be able to add system functionality into my program, such as generating a random number, libraries must first be imported. Libraries are ready-complied and tested programs that can be run when needed. The libraries which will be used in the program are defined initially. There are several benefits to using libraries. For example, using pre-built functions makes coding quicker. Furthermore, they are pre-tested, meaning that they are likely to be error-free. Developing the program in a short amount of time is important as it enables the acquisition of constant user feedback, ensuring that the program is being developed in a way in which benefits their needs. While I am unable to understand the implementation of such libraries, this is not a problem because the encapsulation and abstraction of this makes the coding easier to implement. In fact, knowing everything about the libraries may be a source of confusion, which would slow down the development process.

The code for the imported libraries is displayed below:

```
#importing libraries which will be referenced later
from Patient import Patient  #enables the patient class file to be linked to the main file
import matplotlib.pyplot as plt  #enables the creation of the scattergraph
from datetime import datetime
from collections import Counter
import tkinter as tk
from tkinter import *
import os.path
```

Time was initially imported in order to slow down the rate at which information was being presented to the user. However, due to using Tkinter, there was no longer a need to use it. This is beneficial as it saves lines of code, as well as memory.

One imported module is ‘Patient’, which has been added in order to link the main program to the class ‘Patient’. The class ‘Patient’ is what will be used to create a record for each database. Insights into how this is done and what this means specifically will be referenced in a later section.

Another library which has been imported is ‘mathplotlib.pyplot as plt’. The reason for this is to enable the creation of the scatter diagram to be made. ‘Counter’ was also imported in order to determine the number of times elements are present in a list. The purpose for this is to create coordinates for the scatter diagram, so will be used later.

Another imported library is ‘datetime’. This was imported in order to ensure that the input for a patient’s date of birth was in the correct format.

The next library is Tkinter. This sets the fundamental building blocks for the design of the program – the Graphical User Interface. When importing Tkinter, it is done in two ways. For example, the line ‘from

tkinter import *' is used as it comes with all the modules defined in Tkinter. In addition, in the line 'import tkinter as tk', the term 'tk' acts as an acronym that can be used to create an instance of widgets from Tkinter. As well as saving coding time, this improves code structure.

Finally, 'os.path' is imported as it will be used to code for determining if the 'Patients' file exists within the system. If not, it will be created.

This section gave an overview of why certain libraries were imported as well as the benefits of importing them. The ways in which the libraries will be implemented will further explained later in the document.

PATIENT FILE

Throughout the program, certain files will be needed in order the check data against, and validate data. Whilst some will be added later on in the program, I initialised the program by creating the 'Patients' file, where the contents of the database will be stored.

Before designing the program, the file was created. However, if the file hasn't been created, then the code below creates the file, along with the field headings. By importing 'os.path' earlier, I was able to determine the existence of the 'Patients' file, as shown below:

```
existenceOfFile = os.path.isfile("Patients.csv")  #determines if the patient file exists
if existenceOfFile == False:
    with open("Patients.csv", "w") as patientFile:
        string = "NHS ID, Forename, Surname, Age, Gender, Ethnicity, Allergies, Number of COVID-19 Vaccines, Number of COVID-19 Cases, Address Line One, Address Line Two,
Town/City, Postcode"  #sets the first line of the patient file
        patientFile.write(string)  #writes the first line to file
```

If the selection statement goes to the else statement, the code determines the number of lines in the 'Patients' file, assuming that it exists:

```
else:
  patientFile = open("Patients.csv", "r")
  global delimiter
  delimiter = ","  #sets the delimiter in the patient file, which will be
referenced later on
  global numberOfLines
  numberOfLines = 0  #setting the initial number of lines to 0
  for i in file:  #reads each line of the patient file
      numberOfLines += 1  #adds one to the variable each time there is an
additional number of lines
  patientFile.close()
```

The patient file is initially opened. 'delimiter' is then set as a global variable, and is defined as a comma. The reason why a delimiter is added is because it is more efficient. For example, when writing fields to file, the code below could be used.

```
print(x+","+y)
```
* x and y represent different field values i.e. username and surname

However, when there is a larger number of fields to be added, then following the syntax of using speech marks and a comma can become long. Furthermore, by setting the delimiter as a variable, a developer

can change the delimiter's value, which will automatically change every instance of it within code. This saves the programmer time from having to find each occurrence of the delimiter and manually changing it from a comma to an asterisk for example. As a result, the code below would is more efficient.

```
delimiter = ","
print(x+delimiter+y)
```

After defining the delimiter, 'numberOfLines' is also defined as a global variable so it can be used throughout the program. The purpose of 'numberOfLines' was to determine how many lines (or records) are in the database, which will become useful when validating data later in the program.

One thing to notice is the last line where the patient file is closed. Rather than having two lines for opening and closing the file, I thought that it would be better to replace the code with another statement. Changes to the code can be seen below:

```
else:
    global delimiter
    delimiter = ","  #sets the delimiter in the patient file, which will be referenced later on
    global numberOfLines
    numberOfLines = 0  #setting the initial number of lines to 0
    with open("Patients.csv","r") as file:  #opens the patient file for reading
        for i in file:  #reads each line of the patient file
            numberOfLines += 1  #adds one to the variable each time there is an additional number of lines
```

By using the 'with open("Patients.csv") as file' line, I have been able to reduce the amount of code needed in the program, saving memory. Furthermore, the line removes the hassle of having to remember to close the file.

After setting the number of lines in the database, I was able to begin coding my home screen.

HOME SCREEN

The following section focuses on the process of developing the program home screen.

INITIAL CODE

As mentioned before, the code did not initially incorporate a GUI. The initial code for the home screen is displayed below.

```
def main_menu():
  option = int(input("Enter 1 to submit the details of a new patient\nEnter 2 to access the records of a patient\nEnter 3 to exit\n"))
  while option != 1 and option != 2 and option != 3:
    print("Invalid input")
    time.sleep(0.5)
    main_menu()
  if option == 1:
    entering()
  elif option == 2:
    records()
  elif option == 3:
    print("Goodbye")
    sys.exit()
```

The code above also represents the initial 'main menu'. There were several limitations with this which resulted in my use of Tkinter. Firstly, an obvious problem is the lack of a GUI. Secondly, there were no access levels defined, the program went straight into asking the user what they wanted to do. This is problematic as it means that anyone would be able to access the data, even without a login – posing a serious security problem. This is problematic as it would have meant that the program was not being developed in line with the security requirements of questionnaire respondents who wanted user access levels, and password protection. Furthermore, it did not account for individual patients who wanted to interact with the system. For example, the three options highlighted above do not include an option which allows only one record to be accessed.

TKINTER CODE

The code for the refined home screen incorporating Tkinter is displayed below:

```
def mainScreen(): #creates mainscreen
  global accessLevel
  global mainScreen
  mainScreen = Tk() #creates screen
  mainScreen.geometry("300x250") #sets geometry of screen
  mainScreen.title("Hospital Database") #sets screen name
  Label(text="Select Access Level", width="300", height="2", font=("Calibri", 13)).pack()
  Label(text="").pack() #screen title
  #creating buttons for user access level options
  Button(text="Patient", height="2", width="30", command = patientLogin).pack()
  Label(text="").pack()
  Button(text="Nurse", height="2", width="30", command = login).pack()
  Label(text="").pack()
  Button(text="Doctor", height="2", width="30", command = login).pack()
  Label(text="").pack()
  Button(text="Scientist", height="2", width="30", command = login).pack()
  Label(text="").pack()
  Button(text="Government Official", height="2", width="30", command = login).pack()
  mainScreen.mainloop()
```

The code begins by declaring the global variables 'accessLevel' and 'mainScreen', which enables reusability throughout the code and saves some memory as multiple variables are not created for the same purpose. The main screen was then created, with the dimensions being set. The final look for the page was developed in line with the proposed screen design explained earlier. In order to achieve this design, it was important that each access level had a space between them, hence the addition of the 'Label(text= "").pack()' line several times. Furthermore, it was also important that the dimensions of

each button were equal – for visualisation purposes. In addition, each button was made unique though the assignment of different names.

After specifying the dimensions of the buttons, it was important that each button pressed would lead to another part of the program – the login. If the user chose the access level 'Patient' then they would be taken to a different login page than other access levels. This is because a username and password would only be for a user trying to access the entire database and other functions. A patient using the system is only concerned with their individual record, so doesn't need such credentials. The reason for implementing such access levels was in order to meet the security requirements defined by 11% of questionnaire respondents.

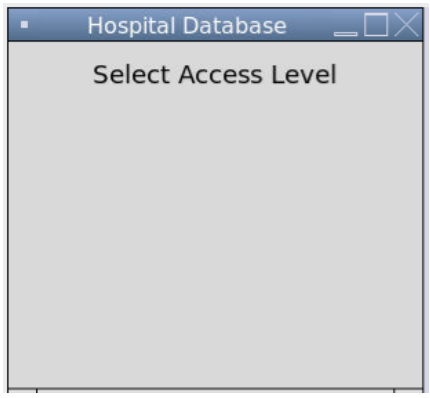
Finally, the use of 'mainScreen.mainloop()' enables the program to continue running until the user manually clicks the 'X' button. The initial program code included the line 'sys.exit()' which allowed the program to end once a user entered the number 3. The 'mainScreen.mainloop()' line acts as a replacement of this statement.

```
elif option == 3:
  print("Goodbye")
  sys.exit()
```

The method of clicking a the 'X' button is an improvement as clicking a button is less time consuming than having to enter a phrase or number. In addition, for those who are not familiar with the system, 'X' is a universally recognised symbol for deleting something. As a result, the assumption is made that if a user sees the button, then they know what it's used for, and how to use it, thus improving the usability of the system.

ERRORS

Whilst there was no form of testing values in this section, an error did occur. For example, when adding coding for the first button – the 'Patient' button – the home screen did not load fully and an error was reported, as shown below.

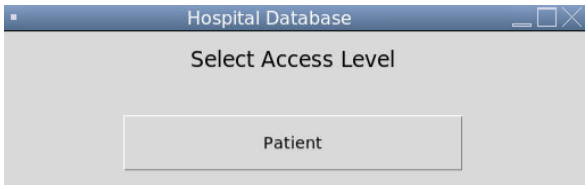



```
Traceback (most recent call last):
  File "main.py", line 501, in <module>
    mainScreen() #starts program
  File "main.py", line 59, in mainScreen
    Button(text="Patient", height="2", width="30", comman
d = patientLogin).pack()
NameError: name 'patientLogin' is not defined
```

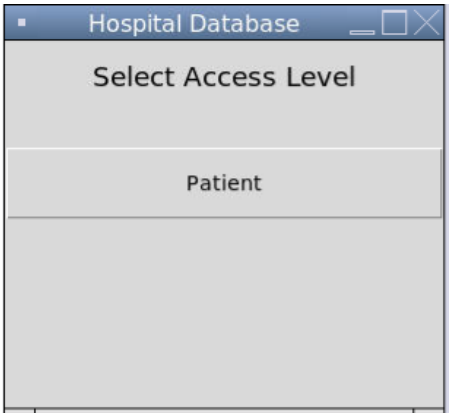
The reason for this was due to the 'command = patientLogin' line. The program was meant to go to a subroutine named 'patientLogin', however this subroutine did not exist as it had not yet been developed. In order to overcome this problem, I created a subroutine with the name 'patientLogin'

which returned a random number – this was done temporarily until it was time to code that module. The code can be seen below, as well as the desired output for the first button.

```
def patientLogin():
  return 0
```



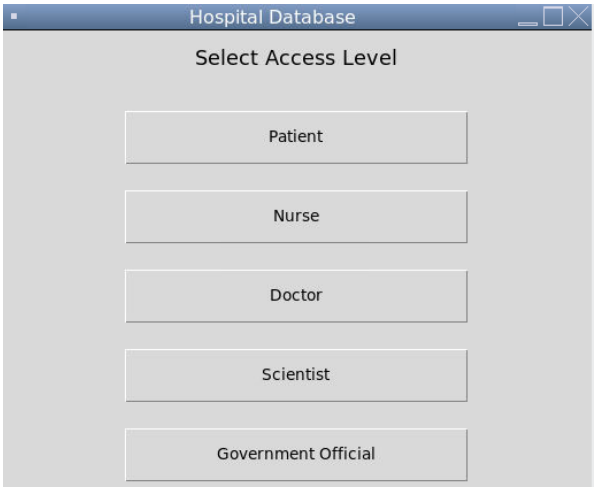
Once the rest of the buttons had been added, the same error occurred as the 'login' subruoutine did not exist. The error generated and incomplete home screen can be seen below.




```
Traceback (most recent call last):
  File "main.py", line 514, in <module>
    mainScreen() #starts program
  File "main.py", line 61, in mainScreen
    Button(text="Nurse", height="2", width="30", command
= login).pack()
NameError: name 'login' is not defined
```

The solution to the error was the same as above, a subroutine was made with the name 'login' and was given the function of randomly returning a value. Once again, this part of the code had not yet been completed so the code within the subroutine would be replaced later. The solution and desired output can be seen below.

```
def login():
  return 0
```

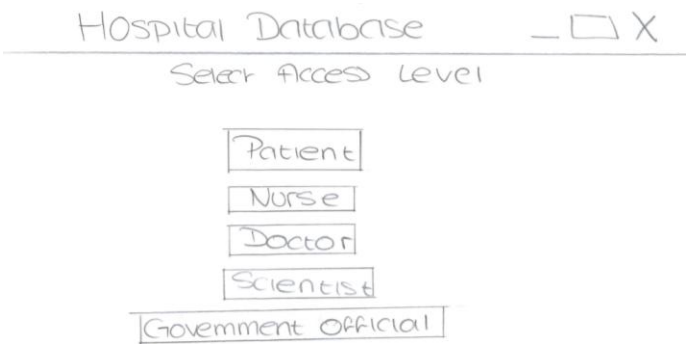


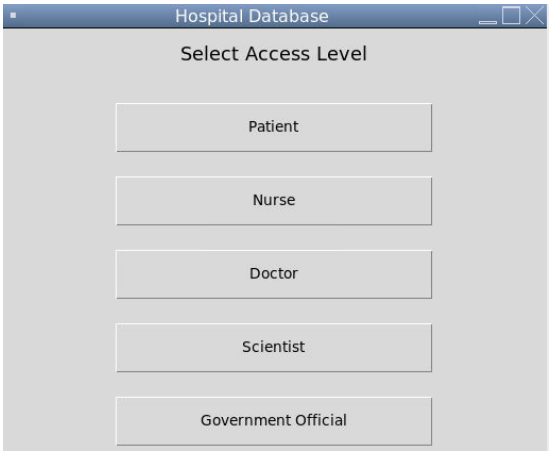
Now that the main screen had been developed and tested, both 'login' screens were developed.

REVIEW

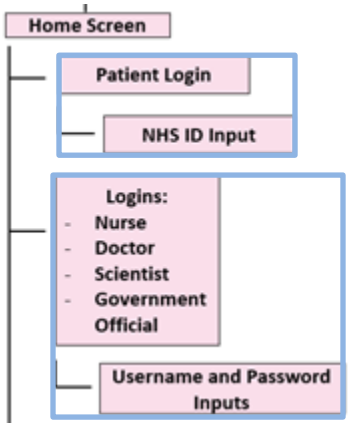
At the end of this section, a home screen was created which included access levels. This met the security requirement of 11% of questionnaire respondents who wanted access levels.

The final design matched the prototype created in the design section. Comparisons can be seen below:





After completing the section, I referenced the refined top-down diagram in order to ensure that the program was developed in line with it.



The home screen had successfully been created. Having said this, the logins had not yet been created. Therefore, it was time to ensure that the screen design for the different logins (shown in blue) were coded for so the user is displayed a screen after clicking a particular access level.

Overall, the success criteria of creating a home screen was met.

PATIENT LOGIN

The following section focuses on the process of developing the patient's login process. The concept of a login was incorporated for obvious security reasons, but also to stay in line with what users wanted. From the questionnaire responses, it was clear that password protection was an essential feature to 16% of respondents.

PATIENT LOGIN SCREEN CODE

The initial code for the patient login screen is displayed below:

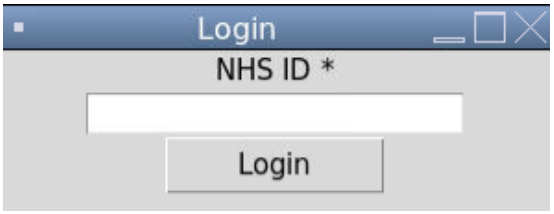
```
def patientLogin(): #creates login screen for patients
    global loginScreen
    loginScreen = Toplevel(mainScreen)
    loginScreen.title("Login")
    loginScreen.geometry("300x250")
    Label(loginScreen, text="Enter the details below") #enables user to input NHS ID
    global usernameVerify
    usernameVerify = StringVar()
    global usernameLoginEntry
    Label(loginScreen, text="NHS ID *").pack()
    usernameLoginEntry = tk.Entry(loginScreen, textvariable=usernameVerify)
    usernameLoginEntry.pack()
    Button(loginScreen, text="Login", width=9, height=1, command = nhsIdEntry).pack() #checks if NHS ID is valid
```

The 'Toplevel(mainScreen)' line enables the same screen from the home screen subroutine to be reused for the creation of other screens. Once again, the screen was given a title, dimensions, and label.

Entering a user's NHS ID was seen as a secure login credential because an individual's NHS ID is only known by oneself. Furthermore, the numbers are in a random order, so it is unlikely that someone else would be able to guess an individual's NHS ID. By using the code 'tk.Entry' an entry box for the user input was created. The purpose of the variable 'usernameVerify' was to store the data entered in the entry box. The final line points the program to another subroutine (nhsIdEntry), which would verify the patient input. In order to prevent the previous error from occurring, a 'fake' subroutine was made, where the code for it is shown below.

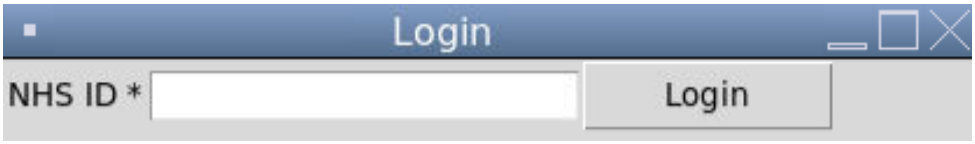
```
def nhsIdEntry(): #checks whether NHS ID is valid
  print("Hello")
```

Finally, I was left with the finished screen design, as shown below.



After looking at the screen design, Dr. Hussain did not want the label, entry box and button on separate lines. As a result, I switched from using the 'pack' code, to the 'grid' code, which enabled the positioning of each data item. This concept of positioning data by using the 'grid' function will be used when developing the rest of the code. The resulting code is displayed below, as well as the final look for the login screen.

```
def patientLogin(): #creates login screen for patients
    global loginScreen
    loginScreen = Toplevel(mainScreen) #uses the same screen defined in the mainscreen
    loginScreen.title("Login") #sets title
    loginScreen.geometry("300x250") #sets dimensions
    Label(loginScreen, text="Enter the details below") #enables user to input NHS ID
    global usernameVerify
    usernameVerify = StringVar() #stores user input
    global usernameLoginEntry
    Label(loginScreen, text="NHS ID *").grid(row = 0, column = 0) #creates label
    usernameLoginEntry = tk.Entry(loginScreen, textvariable=usernameVerify) #creates entry bod
    usernameLoginEntry.grid(row = 0, column = 1) #positions entry box
    Button(loginScreen, text="Login", width=9, height=1, command = nhsIdEntry).grid(row = 0, column= 2) #creates button which points to validation
```



After the patient login screen had been developed and approved by Dr. Hussain, the code for data validation was created.

VALIDATING PATIENT INPUT

The following code outlines how the NHS ID input validated was coded.

Previously declaring the 'usernameVerify' variable as global meant that it could be used in this part of the program. The code for the validation is below:

```
def nhsIdEntry(): #checks whether NHS ID is valid
  username1 = usernameVerify.get()
  usernameLoginEntry.delete(0, END)
  with open('Patients.csv') as file: #opens patient file for reading
    found = False #sets nhsId found as false
    for i in range(numberOfLines):
      line = file.readline() #reads each line in the patient file
      lineSplit = line.split(delimiter) #splits the line each time a delimited is identified
      if lineSplit[0] == username1: #if the first item in the line is the NHS ID
        global lineNumber
        lineNumber = i #print the record in the patient file
        found = True #changes found to true as nhsId has been found
        outputRecord(lineNumber) #prints record
        break
  if found != True:
    nhsIdNotFound() #displays error message
```

The code starts by taking collecting the input from the entry box (usernameVerify) and storing it in the variable 'username1'. The entry box was then automatically cleared to enable the next entry to be entered without manually having to delete the previous NHS ID. Each line of the file is then read, and then split when a comma is encountered. The reason for this is to enable each field to be treated as a separate element.

The following code demonstrates the subroutine which outputs the record to the user:

```
def outputRecord(lineNumber): #displays record
  with open("Patients.csv") as file:
    lines = file.readlines() #reads each line of the file
    header = lines[0] #uses first lien as headed
    record = lines[lineNumber] #finds record containing NHS ID
  header = header.split(delimiter)
  record = record.split(delimiter)
  root = Tk()
  root.geometry("300x250")
  root.title("{0}, {1}".format(record[2],record[1])) #sets the title of the screen to the users surname and forename
  outputValues = [(header),(record)] #sets output values
  for i in range(2):
    for j in range(13):
      addingValues = Entry(root, width=25, fg='black',font=('Calibri',8,'bold')) #sets display design
      addingValues.grid(row=i, column=j) #defines i and j
      addingValues.insert(END, outputValues[i][j]) #adds values to grid
```

The variable ‘lines’ is used to determine the first line of the database, which represents the field names. The variable ‘record’ then stores the record of the NHS ID entered by the patient. Both headers and the record are then split in order to enable each field value to be treated as a separate element.

A separate screen is then made to present the data. The title of the page is then defined as the surname, then forename of the patient. This is to indicate to the user the name of the individual’s record being accessed.

The variable ‘outputValues’ is then created in order to store the header record, and the record with the patient’s NHS ID. A loop was then made to iterate through each row and column. 2 rows were specified as only two records were required. Secondly, 13 columns were specified as it represents the number of fields in the database. The reason why ‘Calibri’ was chosen to display the values was because it is a font which can be easily understood, increasing the usability of the system. Choosing a font that is cursive may be harder for others to read, making it more difficult to access and interpret data.

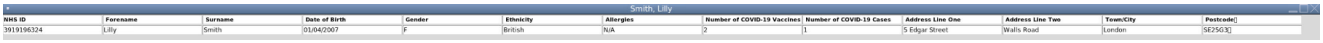
An example of the program’s output can be seen below:

Initially, all of the data was displayed in a bold format. However, upon careful consideration and feedback from Dr. Hussain this was changed. The following code includes the changes made:

```
def outputRecord(lineNumber): #displays record
  with open("Patients.csv") as file:
    lines = file.readlines() #reads each line of the file
    header = lines[0] #uses first lien as headed
    record = lines[lineNumber] #finds record containing NHS ID
  header = header.split(delimiter)
  record = record.split(delimiter)
  root = Tk()
  root.geometry("300x250")
  root.title("{0}, {1}".format(record[2],record[1])) #sets the title of the screen to the users surname and forename
  outputValues = [(header),(record)] #sets output values
  for i in range(2):
    for j in range(13):
      if i == 0:
        addingValues = Entry(root, width=20, fg='black',font=('Calibri',8,'bold')) #sets display design
      else:
        addingValues = Entry(root, width=20, fg='black',font=('Calibri',9)) #sets display design
      addingValues.grid(row=i, column=j) #defines i and j
      addingValues.insert(END, outputValues[i][j]) #adds values to grid
```

The change enabled only the header record to be bold, whilst the rest of the code would be represented in a normal format. This reason for this was to ensure that the heading can be differentiated from the remaining data. The user should clearly see that the value ‘F’, for example, is representative of the gender. It is also important that the header is included in the output as the user needs to be able to determine what each field item represents.

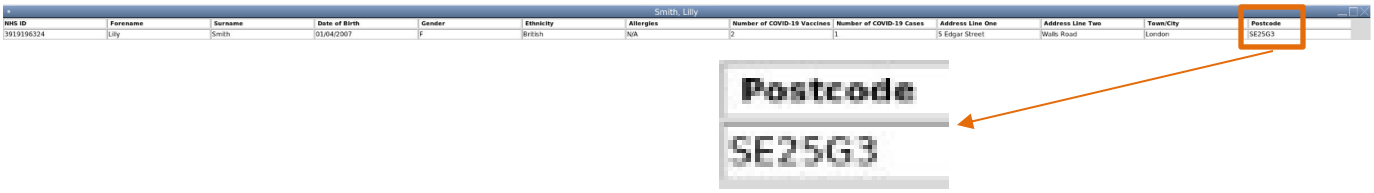
An example of the program’s output can be seen below:



Following this, a further change was made in the program. For instance, the ‘Postcode’ header and field values seemed to display a rectangular symbol, potentially due to the ‘\n’ statement saved next to it in the file. As a result, it was important that the ‘strip(“\n”)’ line of code was added to remove this gap.

The change can be seen in the code below, along with its desired output:

```
def outputRecord(lineNumber): #displays record
  with open("Patients.csv") as file:
    lines = file.readlines() #reads each line of the file
    header = lines[0].strip("\n") #uses first lien as headed
    record = lines[lineNumber].strip("\n") #finds record containing NHS ID
  header = header.split(delimiter) #separates line upon each instance of a delimiter
  record = record.split(delimiter)  #separates line upon each instance of a delimiter
  root = Tk() #creates a new screen
  root.geometry("300x250") #sets screen dimensions
  root.title("{0}, {1}".format(record[2],record[1])) #sets the title of the screen to the users surname and forename
  outputValues = [(header),(record)] #sets output values
  for i in range(2): #reads each record
    for j in range(13): #reads each field
      if i == 0:
        addingValues = Entry(root, width=25, fg='black',font=('Calibri',8,'bold')) #sets display design
      else:
        addingValues = Entry(root, width=25, fg='black',font=('Calibri',9)) #sets display design
      addingValues.grid(row=i, column=j) #positions each field + record
      addingValues.insert(END, outputValues[i][j]) #adds values to grid
```



The next section focuses on the code for the output if the NHS ID input was incorrect. The code is displayed below:

```
def nhsIdNotFound(): #displays message if NHS ID is not valid
    global userNotFoundScreen
    userNotFoundScreen = Toplevel(loginScreen) #uses the same screen as 'loginScreen'
    userNotFoundScreen.title("Error") #sets title
    userNotFoundScreen.geometry("150x100") #sets dimensions
    Label(userNotFoundScreen, text="Patient Not Found").pack() #displays error message
    Button(userNotFoundScreen, text="OK", command=deleteUserNotFoundScreen).pack() #closes screen when button is pressed
```

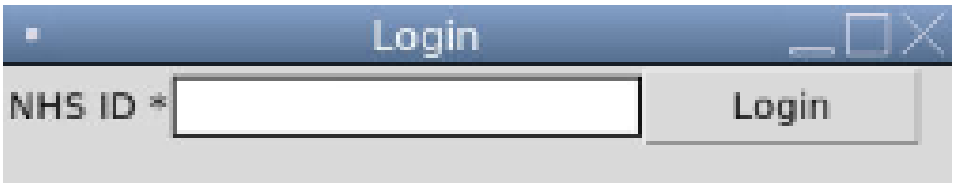
The screen is given the title ‘Error’ and contains the message ‘Patient Not Found’. There is also the addition of the button ‘OK’ which goes to another subroutine which will delete the screen once pressed, taking the user back to the login page.

```
#deleting popups
def deleteUserNotFoundScreen():
  userNotFoundScreen.destroy()
```

The code above is the subroutine pointed by the ‘nhsIDNotFound’ subroutine. It represents the code for deleting the screen.

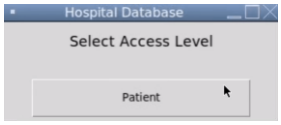
The screen for the error message is below:

The final display for the patient login screen is below:



TESTING

After completing the code, testing took place to ensure that the desired output to certain user inputs was produced. Some of the tests were used from the ones created in the pre-development testing table.

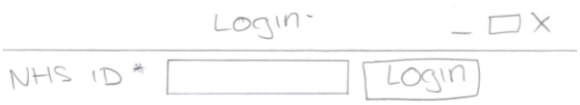
Test	Reason	Test Data	Expected Outcome	Does the expected outcome occur?	Type of Test
Entering nothing into the entry box	The program is unable to generate the record if there is no data to refer to.	Inputting empty string	The program must generate an error, indicating to the user that their input is invalid.	Yes 	Erroneous as there is no user input. Also, an integer value should have been entered.
Entering an invalid input	If the input is incorrect and the NHS ID is not found, then this should be reported back to the user.	Entering random letters “cvbvcbvv ”	The system should generate the message “Patient Not Found”	Yes 	Erroneous as the wrong data type has been used (string instead of integer).
Entering valid input	The system should be able to generate the correct patient record using the valid input given.	A valid NHS ID “3919196 324”	The patient record	Yes  (N.B Patient record displayed is the same as the output presented here.)	Normal as the data is the type that was expected

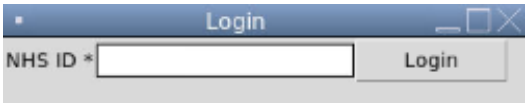
REVIEW

Overall, this part of the program was able to replicate the design made earlier.

At the end of this section, a patient login screen was created which accepted the NHS ID as an input. This met the security requirement of 16% of questionnaire respondents who wanted password protection, where the NHS ID is considered as the 'password' in this instance.

The final design matched the prototype created in the design section. Comparisons can be seen below:

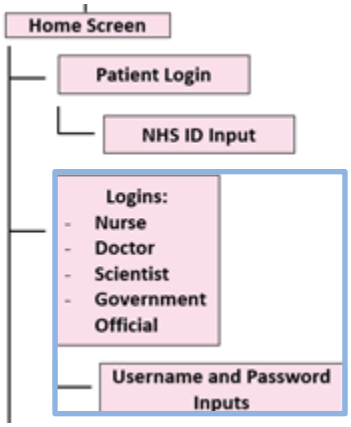




Different tests were conducted in order to ensure that the system provided the appropriate responses to valid, or invalid inputs. The tests were successful and substantiated the robustness of the system.

Overall, the success criteria of creating a patient login screen was met. Additionally, the success criteria of allowing a record of a patient's details to be outputted was successfully met.

After completing the section, I referenced the refined top-down diagram in order to ensure that the program was developed in line with it.



The patient login and NHS ID input subbranches had successfully been met. Having said this, the logins for the access levels had not yet been created. Therefore, after the development and testing of the screen was approved by Dr. Hussain, the login screen (shown in blue) for the access levels 'Nurse', 'Doctor', 'Scientist' and 'Government Official' was ready to be developed so that the user is displayed a screen after clicking any of the named access levels.

REGISTRATION

After the patient login system had been created, it was important that the login was created for those who chose the access levels 'Nurse', 'Scientist', 'Doctor' or 'Government Official'. Before developing the login screens, Dr. Hussain noticed a problem.

A login file would need to be created to store the login details of existing users. However, the issue was regarding how the username and passwords would have been created. A register option was not

included in the home screen. As a result, before creating the login page for the access levels 'Nurse', 'Scientist', 'Doctor' or 'Government Official', the registration page had to be developed.

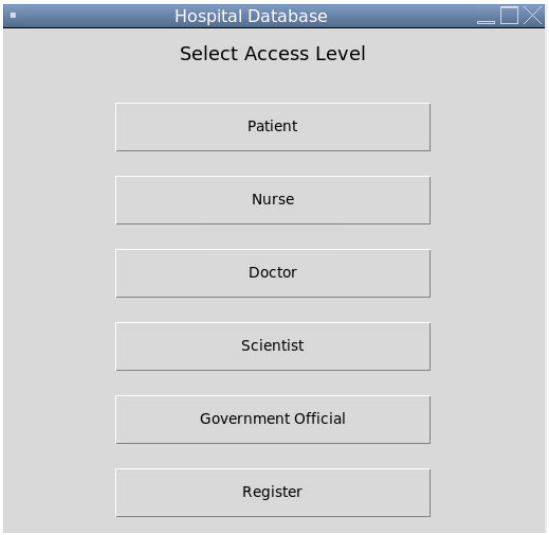
The following section focuses on the process of developing the registration screen.

The process of developing the system takes an agile approach. As a result, before creating the registration screen, it first had to be designed. The design section displays the registration screens that I intended to replicate in the code.

After the registration screen had been approved by Dr. Hussain, the programming began.

Firstly, the home screen design, and top down diagram was amended to add the 'Registration' option to it. As a result, the subroutine for the home screen was also amended to create a button which matched the one on the design. The amended code and output produced can be seen below:

```
def mainScreen(): #creates mainscreen
  global accessLevel
  global mainScreen
  mainScreen = Tk() #creates screen
  mainScreen.geometry("300x250") #sets geometry of screen
  mainScreen.title("Hospital Database") #sets screen name
  Label(text="Select Access Level", width="300", height="2", font=("Calibri", 13)).pack()
  Label(text="").pack() #screen title
  #creating buttons for user access level options
  Button(text="Patient", height="2", width="30", command = patientLogin).pack()
  Label(text="").pack()
  Button(text="Nurse", height="2", width="30", command =  login).pack()
  Label(text="").pack()
  Button(text="Doctor", height="2", width="30", command = login).pack()
  Label(text="").pack()
  Button(text="Scientist", height="2", width="30", command = login).pack()
  Label(text="").pack()
  Button(text="Government Official", height="2", width="30", command = login).pack()
  Label(text="").pack()
  Button(text="Register", height="2", width="30", command = register).pack() #displays register button
  mainScreen.mainloop()
```



The register button now had a command pointing to the 'register' subroutine, which was then coded.

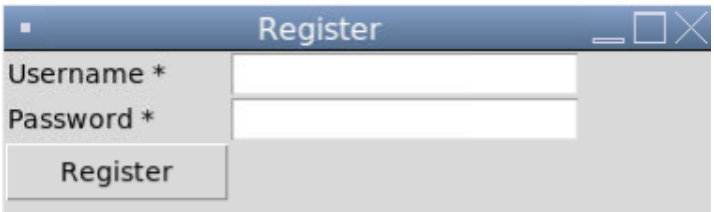
REGISTRATION CODE

The code and output of the registration screen is displayed below:

```
def register(): #creates register screen
    global register_screen
    register_screen = Toplevel(mainScreen)
    register_screen.title("Register")
    register_screen.geometry("300x250")
    global username
    global password
    global usernameEntry
    global passwordEntry
    username = StringVar()
    password = StringVar()
    usernameLabel = Label(register_screen, text="Username * ").grid(row = 0, column = 0) #sets title of user input
    usernameEntry = tk.Entry(register_screen, textvariable=username) #enables user to enter data
    usernameEntry.grid(row = 0, column = 1) #sets position of entry bar
    passwordLabel = Label(register_screen, text="Password * ").grid(row = 1, column = 0)
    passwordEntry = tk.Entry(register_screen, textvariable=password, show='*')
    passwordEntry.grid(row = 1, column = 1)
    Button(register_screen, text="Register", width=10, height=1,command = registerValidation).grid(row = 2, column = 0) #enables inputs to be validated
```

After showing the screen design to Dr. Hussain, he suggested that the 'Username' and 'Password' labels should be centred left. As a result, changes to the code were made in order to change the centring of the label, through the use of 'sticky'. This idea of centring the data left will continue during the development of later screens. The changed code and output can be seen below:

```
def register(): #creates register screen
    global register_screen
    register_screen = Toplevel(mainScreen)
    register_screen.title("Register")
    register_screen.geometry("300x250")
    global username
    global password
    global usernameEntry
    global passwordEntry
    username = StringVar() #stores username input provided by user
    password = StringVar() #stores password input provided by user
    usernameLabel = Label(register_screen, text="Username * ").grid(row = 0, column = 0, sticky = 'w') #sets title of user input
    usernameEntry = tk.Entry(register_screen, textvariable=username) #enables user to enter data
    usernameEntry.grid(row = 0, column = 1) #sets position of entry bar
    passwordLabel = Label(register_screen, text="Password * ").grid(row = 1, column = 0, sticky = 'w')
    passwordEntry = tk.Entry(register_screen, textvariable=password, show='*')
    passwordEntry.grid(row = 1, column = 1)
    Button(register_screen, text="Register", width=10, height=1,command = registerValidation).grid(row = 2, column = 0) #enables inputs to be validated
```



The last line of the code points the 'Register' button to another subroutine (registerValidation) which would validate the username and password entries.

After designing the registration screen to the standard which Dr. Hussain approved of, the registration input validation was developed.

REGISTRATION VALIDATION CODE

Using the registration error/success screens as a guide, and the validation highlighted in the key variables and validation section, the code for the registration validation is displayed below:

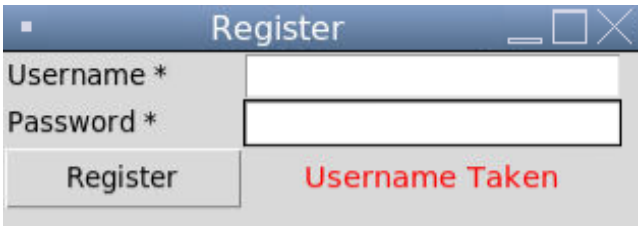
```
def registerValidation(): #validates register input
  usernameInfo = username.get() #returns username input
  passwordInfo = password.get() #returns password input
  validDetails = True #initialises register inputs as true
  with open("LoginDetails.csv") as file: #opens login details file
    for line in file: #reads each line in the file
      line = line.strip("\n").split(delimiter) #removes "\n" from each line and splits it so each element in the line can be accessed separately
      if usernameInfo == line[0]: #checks if the username is in the line
        Label(register_screen, text="                                                ").grid(row = 2, column = 1) #overwrites previous messages
        Label(register_screen, text="Username Taken", fg="red", font=("calibri", 11)).grid(row = 2, column = 1) #outputs error message
        usernameEntry.delete(0, END) #clears username entry box for re-entry
        passwordEntry.delete(0, END) #clears password entry box for re-entry
        validDetails = False #sets details as invalid
  containsSpaces = usernameInfo.isspace() #checks if username contains spaces
  containsSpaces1 = passwordInfo.isspace() #checks if password contains spaces
  if usernameInfo == "" or containsSpaces == True or passwordInfo == "" or containsSpaces1 == True: #checks if entry boxes are empty
    invalidRegistrationOutput() #outputs error message
    validDetails = False
  if validDetails == True: #performs operations if the inputs are valid
    with open("LoginDetails.csv", "a") as file: #opens login details file for appending
      file.write("\n") #writes a new line to the file
      file.write(usernameInfo + delimiter + passwordInfo) #writes the username and password to file
      usernameEntry.delete(0, END)
      passwordEntry.delete(0, END)
      Label(register_screen, text="                                ").grid(row = 2, column = 1) #overwrites previous messages
      Label(register_screen, text="Registration Success", fg="green", font=("calibri", 11)).grid(row = 2, column = 1) #prints success message for user
```

The code begins by creating a csv file to store login data in order to compare the registration details against. In order to have data to compare the inputs against, the login details file was given data. An example of this can be seen below:

LoginDetails.csv ×

```
1  Admin1,SQL$%RLS
2  Hello,No
3  gp1,london
```

The program then reads each line of the file and splits it into separate elements. If the first item in the line is equal to the username input given by the user, then it suggests that the username already exists, and would print an error to the user, stating that the username had been taken. The entry boxes are then cleared to allow the user to re-enter the data. An example of this message can be seen below:



The use of 'isSpace' (in green) is used to determine if there are any spaces in the user's inputs. If the user's entries have spaces, or are empty, then they will be presented with an invalid entry screen, which is another subroutine that will be developed after.

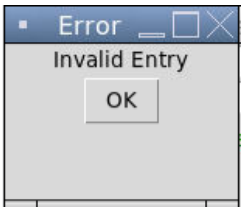
Something to note is that if the username is taken, then it is given a separate output compared to whether the username/password is empty or contains spaces. The reason for this is because there should be a clear distinction between whether a username is taken, or whether the input is invalid. This will make it clearer to the user about why they need to change their details. For instance, it would not make sense for the user to be presented with the 'Invalid Input' message when their input is valid. It would be better for them to know that they cannot use the credentials as they have been taken.

Finally, if the Boolean expression for 'validDetails' is 'True', then the new details are written to the login details file, and the entry boxes are cleared. The users are also presented with a success message. An example of this message can be seen below:

The final part of the validation process was to create the subroutine for the invalid entry screen. The code for this is displayed below:

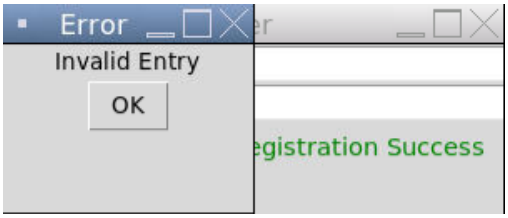
```
def invalidRegistrationOutput(): #outputs error message
    global userNotFoundScreen
    userNotFoundScreen = Toplevel(register_screen)
    userNotFoundScreen.title("Error")
    userNotFoundScreen.geometry("150x100")
    Label(userNotFoundScreen, text="Invalid Entry").pack() #outputs error message to user
    Button(userNotFoundScreen, text="OK", command=deleteUserNotFoundScreen).pack() #closes screen when button is pressed
```

The final line points to the subroutine ('deleteUserNotFoundScreen') which was created earlier. When the user clicks the 'OK' button, the screen is then deleted. An example of this screen is displayed below:



ERRORS

An error occurred after creating the invalid entry screen. If the entry before an invalid entry printed 'Registration Success' or 'Username Taken' then that message still remained when the invalid entry screen was outputted. An example of this can be seen below:



The reason why this had to be changed is because it could confuse a user. For instance, if a user's input was invalid yet they saw a message in the background saying 'Registration Success' then they may not understand what they did wrong. As a result, any previous messages had to be removed. This led an extra line of code being added, as shown below:

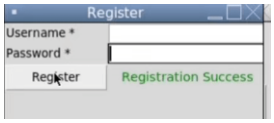
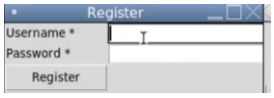
```
def registerValidation(): #validates register input
  usernameInfo = username.get() #returns username input
  passwordInfo = password.get() #returns password input
  validDetails = True #initialises register inputs as true
  with open("LoginDetails.csv") as file: #opens login details file
    for line in file: #reads each line in the file
      line = line.strip("\n").split(delimiter) #removes "\n" from each line and splits it so each element in the line can be accessed separately
      if usernameInfo == line[0]: #checks if the username is in the line
        Label(register_screen, text="                                        ").grid(row = 2, column = 1) #overwrites previous messages
        Label(register_screen, text="Username Taken", fg="red", font=("calibri", 11)).grid(row = 2, column = 1) #outputs error message
        usernameEntry.delete(0, END) #clears username entry box for re-entry
        passwordEntry.delete(0, END) #clears password entry box for re-entry
        validDetails = False #sets details as invalid
  containsSpaces = usernameInfo.isspace() #checks if username contains spaces
  containsSpaces1 = passwordInfo.isspace() #checks if password contains spaces
  if usernameInfo == "" or containsSpaces == True or passwordInfo == "" or containsSpaces1 == True: #checks if entry boxes are empty
    Label(register_screen, text="                                        ").grid(row = 2, column = 1)
    invalidRegistrationOutput() #outputs error message
    validDetails = False
  if validDetails == True: #performs operations if the inputs are valid
    with open("LoginDetails.csv", "a") as file: #opens login details file for appending
      file.write("\n") #writes a new line to the file
      file.write(usernameInfo + delimiter + passwordInfo) #writes the username and password to file
      usernameEntry.delete(0, END)
      passwordEntry.delete(0, END)
      Label(register_screen, text="                        ").grid(row = 2, column = 1) #overwrites previous messages
      Label(register_screen, text="Registration Success", fg="green", font=("calibri", 11)).grid(row = 2, column = 1) #prints success message for user
```

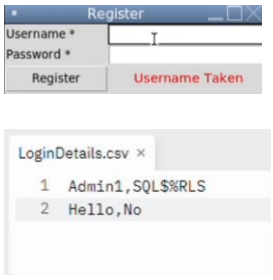
This line of code is seen commonly in the subroutine, as shown in purple. The purpose of this line is to overwrite any prior messages with empty string and will be commonly used throughout the program.

Ensuring that the changes made worked, required testing.

TESTING

After completing the code, testing took place to ensure that it produced the desired output to certain user inputs.

Test	Reason	Test Data	Expected Outcome	Does the expected outcome occur?	Type of Test
Entering nothing into the entry box	The program is unable to generate the record if there is no data to refer to.	Inputting empty string	The program must generate an error, indicating to the user that their input is invalid.	Yes 	Erroneous as there is no user input.
Entering nothing into the entry box after a previous entry generated a message	The program keeping the previous message may confuse the user as to what the problem is.	Inputting empty string	The program must clear the previous message and generate an error.	Yes 	Erroneous as there is no user input.
Entering a taken username	There should not be duplicate credentials in the login details file as it would remove the uniqueness and privacy of an individual having one account. Additionally, if two users have the same username then they can access each other's data, which is not secure.	Entering the username "Hello" and password "No" – these are already saved in the login details file.	The program should output that the username is taken.	Yes 	Whilst the data is normal, it is still erroneous as the name username has been taken.

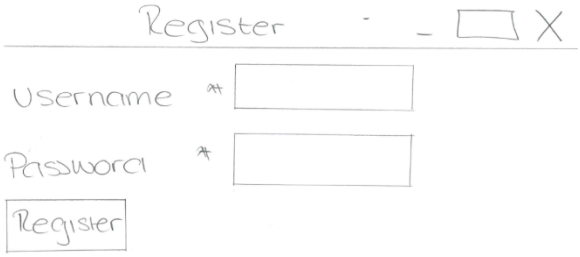
Entering a username and password which have not been taken	It is important that the system stores the credentials so they cannot be reused.	Entering the username “123£ and password “123”	The program should output ‘Registration Success’ and write the username and password to file.	Yes 	Normal as the data is what is expected

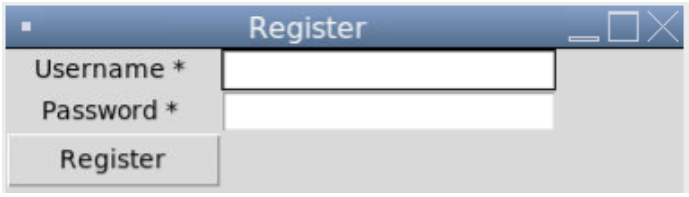
REVIEW

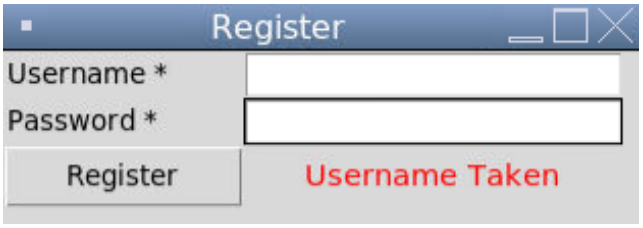
Overall, this part of the program was able to replicate the design made earlier.

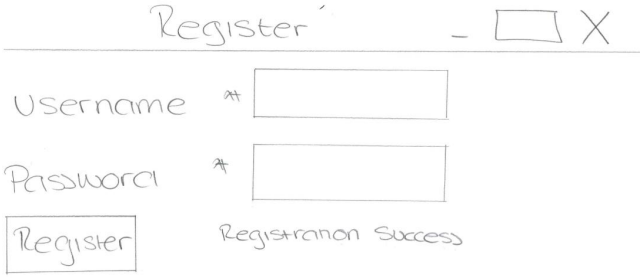
At the end of this section, a registration screen was created which collected username and password inputs. This met the security requirement of 16% of questionnaire respondents who wanted password protection.

The final designs matched the prototypes created in the design section. Comparisons can be seen below:



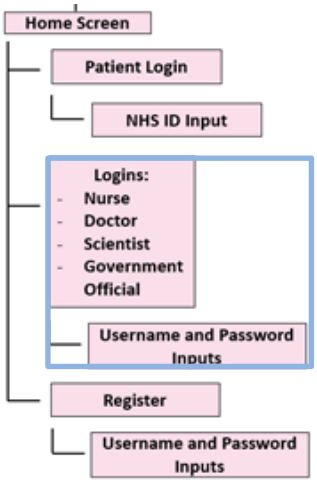






Different tests were conducted in order to ensure that the system provided the appropriate responses to valid, or invalid inputs. The tests were successful and substantiated the robustness of the system.

After completing the section, I referenced the top-down diagram in order to ensure that the program was developed in line with it.



The registration branch had been completed, along with the creation of entry boxes for the username and password inputs. Now that this section had been complete, the login screen (in blue), which would accept username and password inputs, could be developed.

LOGIN

The following section will focus on the process of developing the login for those who choose the access level ‘Nurse’, ‘Doctor’, ‘Scientist’, or ‘Government Official’.

CODE

The use-case diagram was used as a reference to determine the functions the different access levels had access to. Because the access level ‘Nurse’ was not intended to have the ‘Scatter Diagram’ function, amendments were made to the home screen code to take those who chose the access level ‘Nurse' to a different main menu. The changes can be seen below:

```
def mainScreen(): #creates mainscreen
  global accessLevel
  global mainScreen
  mainScreen = Tk() #creates screen
  mainScreen.geometry("300x250") #sets geometry of screen
  mainScreen.title("Hospital Database") #sets screen name
  Label(text="Select Access Level", width="300", height="2", font=("Calibri", 13)).pack() #screen title
  Label(text="").pack() #creates a space between labels
  #creating buttons for user access level options
  Button(text="Patient", height="2", width="30", command = patientLogin).pack() #creates button
  Label(text="").pack() #creates a space between labels
  Button(text="Nurse", height="2", width="30", command = lambda accessLevel = "Nurse":login(accessLevel)).pack() #creates button
  Label(text="").pack()#creates a space between labels
  Button(text="Doctor", height="2", width="30", command = lambda accessLevel = "Doctor":login(accessLevel)).pack() #creates button
  Label(text="").pack()#creates a space between labels
  Button(text="Scientist", height="2", width="30", command = lambda accessLevel = "Scientist":login(accessLevel)).pack() #creates button
  Label(text="").pack()#creates a space between labels
  Button(text="Government Official", height="2", width="30", command = lambda accessLevel = "Government Official":login(accessLevel)).pack() #creates button
  Label(text="").pack()#creates a space between labels
  Button(text="Register", height="2", width="30", command = register).pack() #displays register button
  mainScreen.mainloop() #keeps screen open until user clicks 'X'
```

The 'lambda' function was used was as it enabled the access levels to be passed as arguments into the 'login' subroutine. The use of this will become more apparent when programming the main screen.

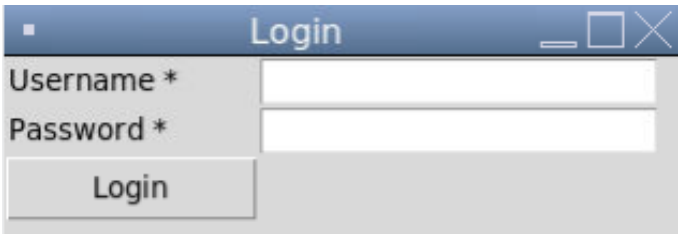
After defining the access levels of each button, the login screen was developed. The code is below:

```
def login(accessLevel): #creates login screen
  global loginScreen
  loginScreen = Toplevel(mainScreen) #uses the same login screen defined in the mainScreen subroutine
  loginScreen.title("Login") #sets title
  loginScreen.geometry("300x250") #sets dimensions
  global usernameVerify
  global passwordVerify
  usernameVerify = StringVar() #stores username input
  passwordVerify = StringVar() #stores password input
  global usernameLoginEntry
  global passwordLoginEntry
  Label(loginScreen, text="Username * ").grid(row = 0, column = 0, sticky = 'w') #sets username label
  usernameLoginEntry = tk.Entry(loginScreen, textvariable=usernameVerify) #creates entry box
  usernameLoginEntry.grid(row = 0, column = 1) #positions entry box
  Label(loginScreen, text="Password * ").grid(row = 1, column = 0, sticky = 'w') #sets password label
  passwordLoginEntry = tk.Entry(loginScreen, textvariable=passwordVerify, show= '*') #creates entry box and displays input as asterisk
  passwordLoginEntry.grid(row = 1, column = 1) #sets entry box position
  Button(loginScreen, text="Login", width=10, height=1, command = lambda accessLevel = accessLevel:loginVerify(accessLevel)).grid(row = 2, column = 0) #creates button which
onece pressed, enables user inputs to be verified
```

The code positions the 'Username' and 'Password' labels and entry boxes to the left due to the suggestion made by Dr. Hussain earlier. The last line creates a button which validates the inputs entered once pressed. Once again, the accessLevel is passed as an argument into the login validation subroutine 'loginVerify'. This was done so that it could be used when coding the main menu.

One factor to consider is the use of the code in red. This means that when a user enters the password, it will be displayed to them in the form of an asterisk. The use of sanitisation here protects the password and prevents it from being visible to anyone who may be looking at the screen, thus improving the security of the login system.

The final screen design, approved by Dr. Hussain is below:



VALIDATING LOGIN SCREEN

The following section focuses on the development of the login validation subroutine. The code is displayed below:

```
def loginVerify(accessLevel): #verifies login details
  username1 = usernameVerify.get() #gets username input
  password1 = passwordVerify.get() #gets password input
  usernameLoginEntry.delete(0, END) #clears username entry box
  passwordLoginEntry.delete(0, END) #clears password entry box
  found = False #states that the login has not been found
  with open("LoginDetails.csv") as file: #opens file for reading
    for line in file: #reads each line in the file
      line = line.strip("\n").split(delimiter) #removes "\n" from line
      if username1 == line[0] and password1 == line[1]: #checks if username and password inputs are in the file
        found = True #login credentials have been found
        break
  if found == False:
      userNotFound() #displays error message
  else:
    loginSuccess(accessLevel) #points to a success screen
```

The ‘Username’ and ’Password’ entry boxes are then cleared once accepted because the next part of the program built will be the main menu, and if the user clicks the ‘X’ button the main menu then they will be taken to the login screen. As a result, users will need to re-enter the credentials to access the database. This meets the security requirement of 16% of qustionnaire respondents to have password protection of data. Also, if a user moves away from the computer after clicking ‘X’ on the main menu then it is important that their login credentials do not remain as it could result in an unauthorised access to, and the subsequent security breach of data.

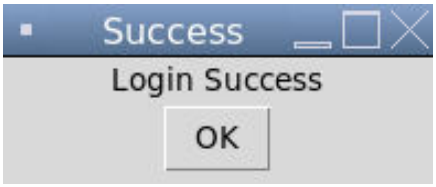
Each line is read with the “\n” being stripped from each line. Initially, the “\n” string was not stripped, which caused an undesired output. For example, even when the user entered valid credentials, they were presented with an invalid output message. I later found out that this was because when the lines were being read, the “\n” value was at the end of each line. As a result, even if the login credentials seemed valid, they were not processed by the program as so because they did not include new line value. This meant that the ‘strip’ function had to be used to prevent the program from seeing the “\n” value as part of the credential.

If the login credentials had not been found, the program points to the subroutine ‘userNotFound’ which displays an error message. The code for this can be seen below:

```
# Designing popup for user not found
def userNotFound(): #displays message if login details are not valid
    global userNotFoundScreen
    userNotFoundScreen = Toplevel(loginScreen) #uses the same screen defined in the 'loginScreen' subroutine
    userNotFoundScreen.title("Invalid Entry") #sets title
    userNotFoundScreen.geometry("150x100") #sets dimensions
    Label(userNotFoundScreen, text="Incorrect Username or Password").pack() #displays error message
    Button(userNotFoundScreen, text="OK", command=deleteUserNotFoundScreen).pack() #deletes the popup once the button is pressed
```

The code is similar as the ‘nhsIdNotFound’ subroutine, meaning that it could be reused with slight amendments. One change was setting the screen title to ‘Invalid Entry’, as opposed to ‘Error’. Another change was setting the error message to ‘Incorrect Username or Password’, instead of ‘Patient Not Found’. The last line then creates an ‘OK’ button which will delete the screen once pressed. The output can be seen below:

If the login credentials are valid and 'True', then the program will go the subroutine 'loginSuccess', which takes the access level as a parameter, so that it can be used when coding the main menu. The code and output for the subroutine can be seen below:




```
def loginSuccess(accessLevel): #displays success message if login details are valid
  global loginSuccessScreen
  loginSuccessScreen = Toplevel(loginScreen) #uses the same screen as 'loginScreen'
  loginSuccessScreen.title("Success") #sets title
  loginSuccessScreen.geometry("150x100") #sets dimensions
  Label(loginSuccessScreen, text="Login Success").pack() #displays success message
  Button(loginSuccessScreen, text="OK", command=deleteLoginSuccess).pack() #closes screen when button is pressed
  mainMenu1(accessLevel)
```

The code is similar to the 'userNotFound' screen, allowing it to be reused, with slight amendments. One change was setting the title as 'Success', and the label as 'Login Success'. There is also the addition of a button which once pressed goes to a subroutine which deletes the screen. The code for this is displayed below:

```
#deleting popups
def deleteLoginSuccess():
    loginSuccessScreen.destroy()
```

The code is similar to the one in the 'deleteUserNotFoundScreen' subroutine, so was reused. The only change was to the variable being 'destroyed' ('loginSuccesScreen' as opposed to 'userNotFoundScreen').

The final line of the code points to the main menu subroutine, where the access level was taken as an argument so that it could be used to determine which main menu to point to. Before developing this menu, it was important to test the login verification process first.

TESTING

The following tests were completed to ensure that the code produced the desired output to certain user inputs:

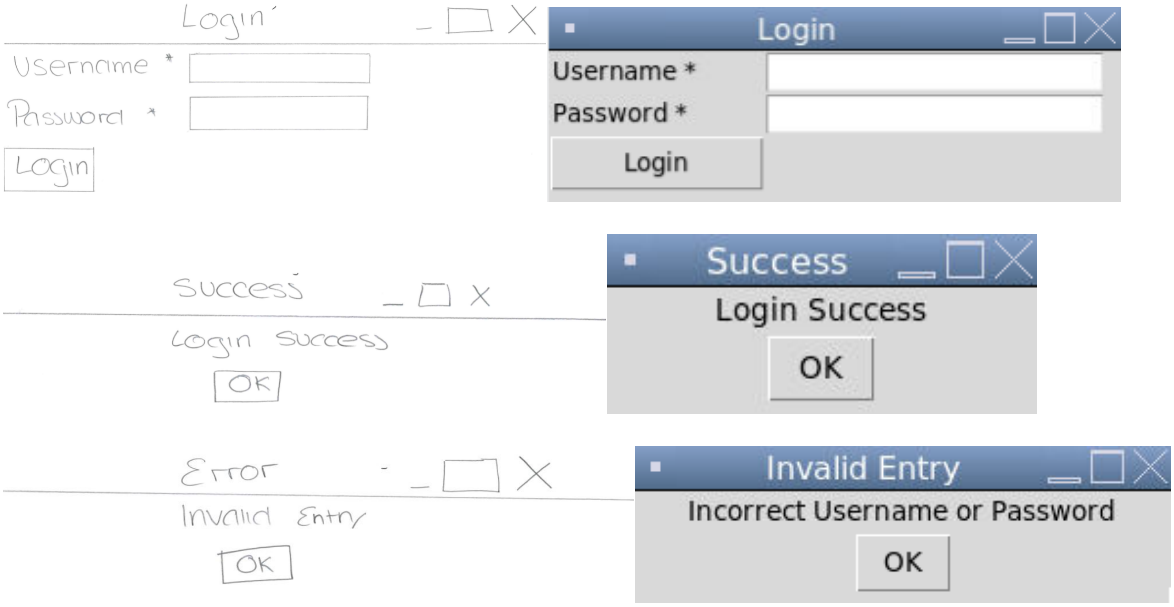
Test	Reason	Test Data	Expected Outcome	Does the expected outcome occur?	Type of Test
Entering nothing into the entry box	The program is unable to check empty credentials against the login details file.	Inputting empty string	The program must generate an error, indicating to the user that their input is invalid.	Yes 	Erroneous as there is no user input.
Entering invalid credentials	A user should not be able to gain access to the system if the data they enter isn't valid. This meets the requirements of 16% of questionnaire respondents who wanted password protection as a security requirement.	Data not stored in the login details file. The username is "565" and the password was "565".	An error message will be outputted and the previous credentials entered will be cleared.	Yes 	There is no specific syntax for username details so it may be considered normal. However, because the values are taken it may be considered to be erroneous data.
Entering valid data	The user should be able to enter the system if they provide valid credentials.	Entering data which exists in the login details file. Username "Hello" and password "No" – the same values used to validate the registration details.	The user should be presented with a success message, and taken to the next stage of the program – the main screen, which will be developed next.	Yes 	Normal data as the inputs are present in the login details file.

REVIEW

Overall, this part of the program was able to meet the success criteria of creating a login page.

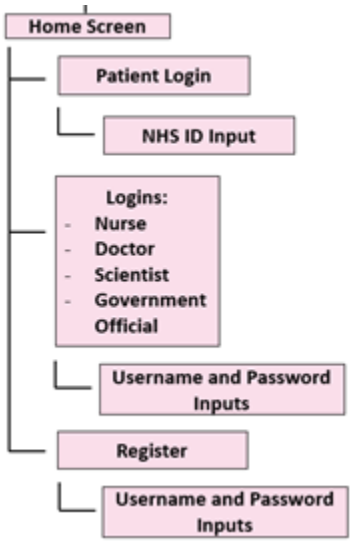
At the end of this section, a login screen was created which accepted only allowed users into the system following valid username and password credentials. This met the security requirement of 16% of questionnaire respondents who wanted password protection. Additionally, appropriate error messages were created when the credentials were incorrect.

The final designs somewhat matched the prototypes created in the design section. Comparisons can be seen below:



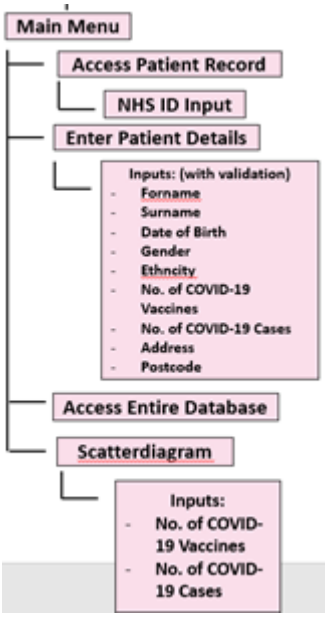
Different tests were conducted in order to ensure that the system provided the appropriate responses to valid, or invalid inputs. The tests were successful and substantiated the robustness of the system.

After completing the section, I referenced the top-down diagram in order to ensure that the program was developed in line with it.



I was able to create login pages for the user access levels 'Nurse', 'Doctor', 'Scientist', or 'Government Official' which accepted username and password inputs.

Following this, the home screen branch had been successfully completed. Now was time to move onto coding the next branch – the main menu.



MAIN MENU

The following section focuses on the process of developing the main menu of the program. The coding was designed with the intention of replicating the design made earlier.

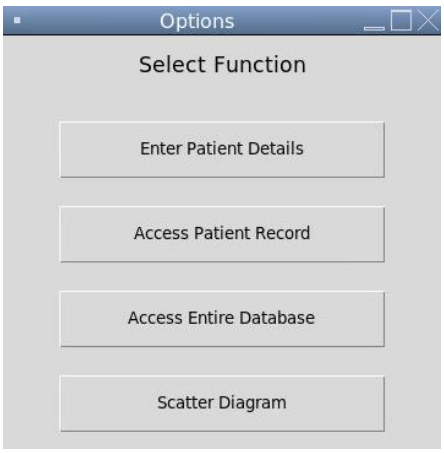
CODE

The code for the main menu is displayed below:

Change in code referenced in the next section

```
def mainMenu1(accessLevel): #creates first main menu
  global mainMenu
  mainMenu = Toplevel(mainScreen) #creates screen using the one defined in the subroutine 'mainScreen'
  mainMenu.geometry("300x250") #sets dimesnison
  mainMenu.title("Options") #sets title
  Label(mainMenu, text="Select Function", width="300", height="2", font=("Calibri", 13)).pack() #sets screen title
  Label(mainMenu,text="").pack() #creates empty space between labels
  #creating buttons for user options
  Button(mainMenu,text="Enter Patient Details", height="2", width="30", command = patientDetails).pack() #creates button for function
  Label(mainMenu,text="").pack() #creates empty space between labels
  Button(mainMenu,text="Access Patient Record", height="2", width="30", command = patientLogin).pack() #creates button for function
  Label(mainMenu,text="").pack() #creates empty space between labels
  Button(mainMenu,text="Access Entire Database", height="2", width="30", command=entireDatabase).pack() #creates button for function
  Label(mainMenu,text="").pack() #creates empty space between labels
  if accessLevel != "Nurse": #does not show this button if the access level is 'Nurse'
    Button(mainMenu,text="Scatter Diagram", height="2", width="30", command=plot).pack() #creates button for function
```

The code begins by designing the screen and its title/dimensions, using the design as a guidance. Whilst looking over the use-case diagram, I noticed that the 'Scatter Diagram' option was not to be made available to access level 'Nurse'. By defining the access levels as arguments earlier, I was able to use it as a parameter to this procedure. As a result, the if statement makes use of the access level passed into the subroutine and only provides the 'Scatter Diagram' button if the access level chosen by the user is not 'Nurse'. The output of the code can be seen below:



Nurse Main Menu *Main Menu For Doctor, Scientist or Government Official*

ERRORS AND TESTING

Initially, two separate subroutines for the main menus were created, the code is displayed below.

```
def mainMenu(): #creates first main menu
  global mainMenu
  mainMenu = Toplevel(mainScreen)
  mainMenu.geometry("300x250")
  mainMenu.title("Options")
  Label(mainMenu, text="Select Function", width="300", height="2", font=("Calibri", 13)).pack() #sets screen title
  Label(mainMenu,text="").pack()
  #creating buttons for user options
  Button(mainMenu,text="Enter Patient Details", height="2", width="30", command = patientDetails).pack()
  Label(mainMenu,text="").pack()
  Button(mainMenu,text="Access Patient Record", height="2", width="30", command = patientLogin).pack()
  Label(mainMenu,text="").pack()
  Button(mainMenu,text="Access Entire Database", height="2", width="30", command=entireDatabase).pack()
  Label(mainMenu,text="").pack()
  Button(mainMenu,text="Scatter Diagram", height="2", width="30", command=plot).pack()

def mainMenu1(): #creates main menu for nurse
  global mainMenu1
  mainMenu1 = Toplevel(mainScreen)
  mainMenu1.geometry("300x250")
  mainMenu1.title("Options")
  Label(mainMenu1, text="Select Function", width="300", height="2", font=("Calibri", 13)).pack()
  Label(mainMenu1,text="").pack()
  Button(mainMenu1,text="Enter Patient Details", height="2", width="30", command = patientDetails).pack()
  Label(mainMenu1,text="").pack()
  Button(mainMenu1,text="Access Patient Record", height="2", width="30", command = patientLogin).pack()
  Label(mainMenu1,text="").pack()
  Button(mainMenu1,text="Access Entire Database", height="2", width="30", command=entireDatabase).pack()
```

The code below also demonstrates the selection statement used to point to different menus depending on the access level:

```
def loginSuccess(accessLevel): #displays success message if login details are valid
  global loginSuccessScreen
  loginSuccessScreen = Toplevel(loginScreen)
  loginSuccessScreen.title("Success")
  loginSuccessScreen.geometry("150x100")
  Label(loginSuccessScreen, text="Login Success").pack() #displays success message
  Button(loginSuccessScreen, text="OK", command=deleteLoginSuccess).pack() #closes screen when button is pressed
  if accessLevel == "Nurse":
    mainMenu1() #goes to a different main menu if the user is a nurse
  else:
    mainMenu()
```

This had to be changed because the duplicated subroutines took up unnecessary lines of code, so it would be more efficient to combine them together. After deleting the second subroutine from the program, and amending the last time of the 'loginSuccess' subroutine, I was left with the code below:

```
def loginSuccess(accessLevel): #displays success message if login details are valid
  global loginSuccessScreen
  loginSuccessScreen = Toplevel(loginScreen)
  loginSuccessScreen.title("Success")
  loginSuccessScreen.geometry("150x100")
  Label(loginSuccessScreen, text="Login Success").pack() #displays success message
  Button(loginSuccessScreen, text="OK", command=deleteLoginSuccess).pack() #closes screen when button is pressed
  mainMenu(accessLevel)
```

However, when running the code, the following error was generated:

```
Exception in Tkinter callback
Traceback (most recent call last):
  File "/usr/lib/python3.8/tkinter/__init__.py", line 1883, in __call__
    return self.func(*args)
  File "main.py", line 179, in <lambda>
    Button(loginScreen, text="Login", width=10, height=1, command = lambda accessLevel = accessLevel:loginVeri
fy(accessLevel)).grid(row = 2, column = 0) #enables user inputs to be verified
  File "main.py", line 248, in loginVerify
    loginSuccess(accessLevel) #proceeds to main menu
  File "main.py", line 257, in loginSuccess
    mainMenu(accessLevel)
TypeError: 'Toplevel' object is not callable
```

I realised that this was because the subroutine shared the same name as the variable used to create the screen (shown in purple).

```
def mainMenu(): #creates first main menu
  global mainMenu
  mainMenu = Toplevel(mainScreen)
  mainMenu.geometry("300x250")
  mainMenu.title("Options")
  Label(mainMenu, text="Select Function", width="300", height="2", font=("Calibri", 13)).pack() #sets screen title
  Label(mainMenu,text="").pack()
  #creating buttons for user options
  Button(mainMenu,text="Enter Patient Details", height="2", width="30", command = patientDetails).pack()
  Label(mainMenu,text="").pack()
  Button(mainMenu,text="Access Patient Record", height="2", width="30", command = patientLogin).pack()
  Label(mainMenu,text="").pack()
  Button(mainMenu,text="Access Entire Database", height="2", width="30", command=entireDatabase).pack()
  Label(mainMenu,text="").pack()
  Button(mainMenu,text="Scatter Diagram", height="2", width="30", command=plot).pack()
```

As a result, the subroutine name and call in the 'loginSuccess' subroutine was changed from 'mainMenu' to 'mainMenu1'. This change in code to the main menu subroutine can be seen in the previous section (shown in orange). The change of code in the 'loginSuccess' subroutine can be seen below (in orange):

```
def loginSuccess(accessLevel): #displays success message if login details are valid
  global loginSuccessScreen
  loginSuccessScreen = Toplevel(loginScreen) #uses the same screen as 'loginScreen'
  loginSuccessScreen.title("Success") #sets title
  loginSuccessScreen.geometry("150x100") #sets dimensions
  Label(loginSuccessScreen, text="Login Success").pack() #displays success message
  Button(loginSuccessScreen, text="OK", command=deleteLoginSuccess).pack() #closes screen when button is pressed
  mainMenu1(accessLevel)
```

After fixing this problem, a test was completed which ensured that clicking each button would lead to a separate screen. Because the screens had not been made, the program was unable to display a new

screen. However, there were no errors generated when the buttons were pressed, indicating that the code had worked. Examples can be seen below:

Nurse Main Menu

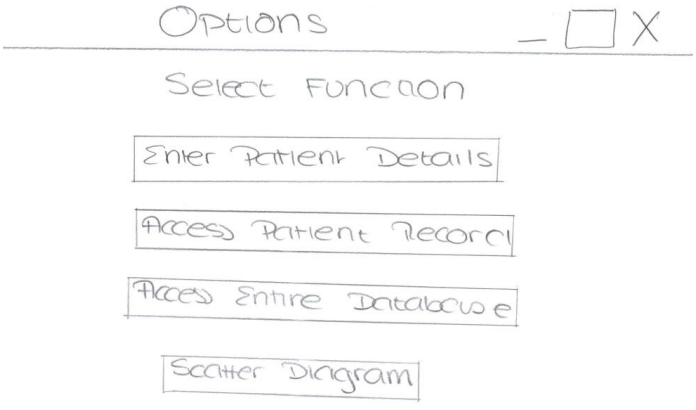
Main Menu For Doctor, Scientist or Government Official

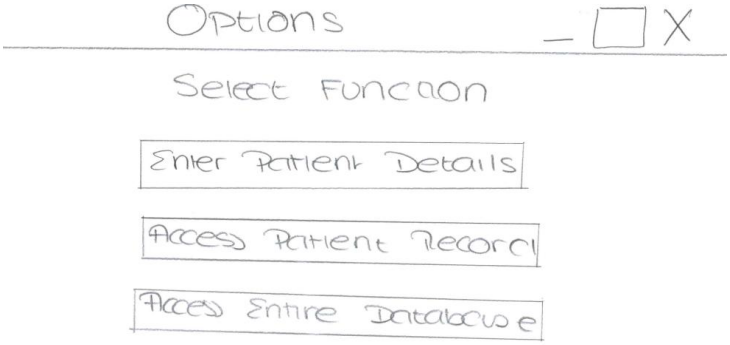
REVIEW

Overall, this part of the program was able to meet the success criteria of creating a main menu, as well as creating a separate main menu for the 'Nurse' access level.

At the end of this section, the use case diagram was implemented by giving the different access levels different rights to data. This use of access levels meets the security requirement for 11% of questionnaire respondents.

The final design matched the prototype created in the design section. Comparisons can be seen below:

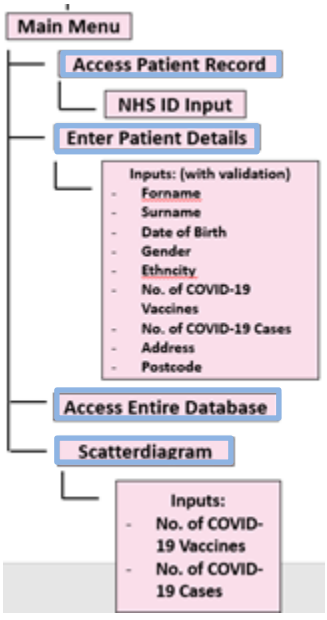




Testing was conducted to ensure that the buttons worked, and the tests were successful. As a result, there may be increased certainty regarding the button as a usability feature which points the user to another screen.

After completing the section, I referenced the top-down diagram in order to ensure that the program was developed in line with it.

After completing the section, I referenced the top-down diagram in order to ensure that the program was developed in line with it.



The labels for each option had successfully been devised and the ones created in the top-down diagram. After the main screen had been developed, tested and approved by Dr. Hussain, the screns were developed for each option.

OPTIONS

The following section focuses on the process of developing the option screens for the options referenced in the main menu.

ACCESS PATIENT RECORD

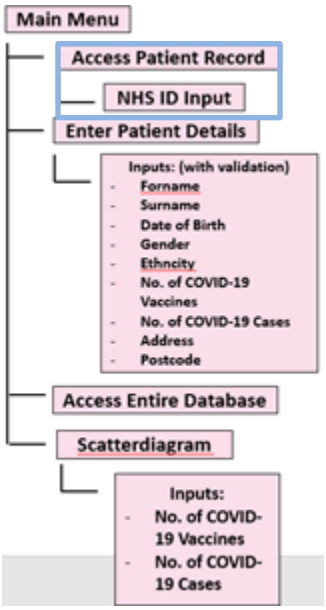
CODE

The first option screen that was developed was the 'Access Patient Record' option. I realised that the 'patientLogin' subroutine created earlier had the same functionality that I intended for this section. As a result, when creating the command function, it was made to point to the same 'patientLogin' subroutine. A reason why this worked was because the 'patientLogin' screen design did not specifically state that it was for the access level 'Patient's use only. As a result, the subroutine could be used in this section. An example of the code pointing to this subroutine can be seen below:

```
def mainMenu1(accessLevel): #creates first main menu
  global mainMenu
  mainMenu = Toplevel(mainScreen) #creates screen using the one defined in the subroutine 'mainScreen'
  mainMenu.geometry("300x250") #sets dimesnison
  mainMenu.title("Options") #sets title
  Label(mainMenu, text="Select Function", width="300", height="2", font=("Calibri", 13)).pack() #sets screen title
  Label(mainMenu,text="").pack() #creates empty space between labels
  #creating buttons for user options
  Button(mainMenu,text="Enter Patient Details", height="2", width="30", command = patientDetails).pack() #creates button for function
  Label(mainMenu,text="").pack() #creates empty space between labels
  Button(mainMenu,text="Access Patient Record", height="2", width="30", command = patientLogin).pack() #creates button for function
  Label(mainMenu,text="").pack() #creates empty space between labels
  Button(mainMenu,text="Access Entire Database", height="2", width="30", command=entireDatabase).pack() #creates button for function
  Label(mainMenu,text="").pack() #creates empty space between labels
  if accessLevel != "Nurse": #does not show this button if the access level is 'Nurse'
    Button(mainMenu,text="Scatter Diagram", height="2", width="30", command=plot).pack() #creates button for function
```

By using the pre-existing subroutine, testing did not have to be carried out as it had been pre-tested. All that was left was to ensure that clicking the button 'Access Patient Record' would open the 'patientLogin' screen. This can be seen below:

REVIEW



The top-down diagram was referenced. It was clear that the access patient record screen had been created, taking in the NHS ID as an input. By creating this function, the success criteria of allowing a record of a patient's details to be outputted was successfully met.

The final design matched the prototype created in the design section. Comparisons can be seen below:

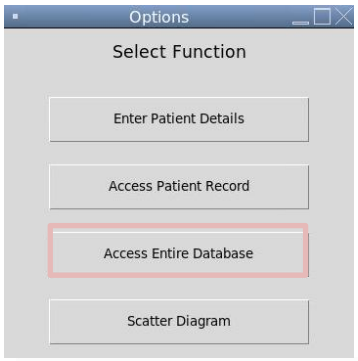
Forename	Surname	NHS Number	Age	Number of VAccines	COVID-19 Cases
Farida	Addo	9131696573	17	Y	0

An obvious change to the initial design is the order of the fields, as well as the fact that the date of birth is provided to the user instead of the age. Another thing worth mentioning is the fact that when planning the program initially, one of the scatter diagram co-ordinates was about whether or not a patient had received a vaccine, as opposed to the number of vaccines they had received. After thinking logically and thinking ahead, I realised that a 'Y' or 'N' value (which can be seen in the initial table as it had not been changed) would not form a compelling scatter diagram which inferences could be made from.

After this part of the program was tested and was approved by Dr. Hussain, the next option was coded.

ACCESS ENTIRE DATABASE

POINTING TO THE SUBROUTINE



The next option that was developed was the 'Access Entire Database' option. The part of the code which points to the subroutine which will display the screen is below:

```
def mainMenu1(accessLevel): #creates first main menu
  global mainMenu
  mainMenu = Toplevel(mainScreen) #creates screen using the one defined in the subroutine 'mainScreen'
  mainMenu.geometry("300x250") #sets dimesnison
  mainMenu.title("Options") #sets title
  Label(mainMenu, text="Select Function", width="300", height="2", font=("Calibri", 13)).pack() #sets screen title
  Label(mainMenu,text="").pack() #creates empty space between labels
  #creating buttons for user options
  Button(mainMenu,text="Enter Patient Details", height="2", width="30", command = patientDetails).pack() #creates button for function
  Label(mainMenu,text="").pack() #creates empty space between labels
  Button(mainMenu,text="Access Patient Record", height="2", width="30", command = patientLogin).pack() #creates button for function
  Label(mainMenu,text="").pack() #creates empty space between labels
  Button(mainMenu,text="Access Entire Database", height="2", width="30", command=entireDatabase).pack() #creates button for function
  Label(mainMenu,text="").pack() #creates empty space between labels
  if accessLevel != "Nurse": #does not show this button if the access level is 'Nurse'
    Button(mainMenu,text="Scatter Diagram", height="2", width="30", command=plot).pack() #creates button for function
```

CODE

The code was written to replicate the design made earlier, and can be seen below:

```
def entireDatabase(): #displaying database
  database = [] #creates an empty list
  with open("Patients.csv") as file: #opens file for reading
    lines = file.readlines() #reads all records in file
  for record in lines: #goes through each record
    database.append(record.split(delimiter)) #adds each record to the separate file
  root = Tk() #creates screen
  root.geometry("300x250") #sets dimensions
  root.title("Database") #sets title
  for i in range(numberOfLines): #loops for the amount of record in the file
    for j in range(12): #loops for the number of fields in the file
      if i == 0:
        addingValues = Entry(root, width=20, fg='black',font=('Arial',8,'bold')) #makes header bold
      else:
        addingValues = Entry(root, width=20, fg='black',font=('Arial',9))
      addingValues.grid(row=i, column=j) #determines position of values
      addingValues.insert(END, database[i][j]) #adds values to screen
```

The empty list 'database' is made in order to store all the lines in the file, which is acquired by using the 'readlines' function in the database. When coding, I found that I was unable to use the 'split' function alongside the 'readlines' function. As a result, the reason why a list was made was because it was easier to then split each line upon the occurrence of a delimiter so they could be treated as separate elements.

The code to display the fields and records was then programmed. Writing this section of code was made easier by the fact that most of the code from outputting patient records had the same functionality, and had been pre-tested. One of the differences is that the first for loop iterates 'numberOfLines' amount of times instead of '2' as it is now going through the whole patient database file. I used a selection (if)

statement in order to allow the header value to be bold, whilst the rest of the records remain in a normal font. This was done in order to make it easier to see what each field indicates.

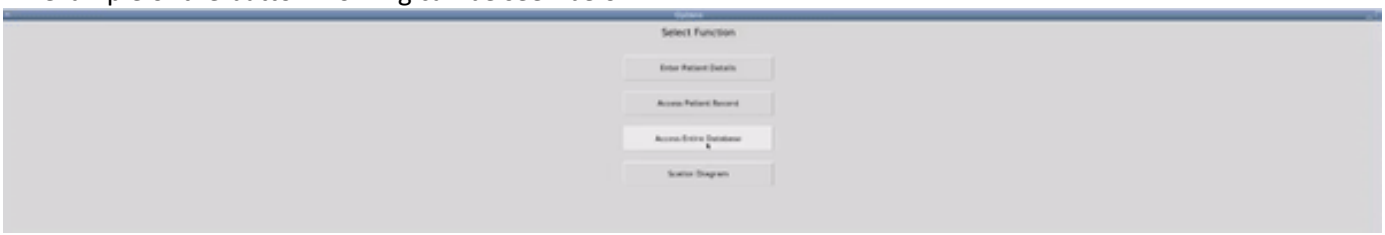
TESTING

Because this section does not require user inputs, the only testing required was to ensure that clicking the 'Access Entire Database' button would output the entire database. Test data was added to the 'Patients' csv file in order to determine if the program would work. The state of the csv file can be seen below:

Patients.csv ×

```
NHS ID, Forename, Surname, Date of Birth, Gender, Ethnicity, Allergies, Number of COVID-19 Vaccines, Number of COVID-19 Cases, Address Line One, Address Line Two, Town/City, Postcode
1762883873,Farida,Addo,25/08/2004,F,Black African,N/A,1.0,0.0,8 Pine Building,Scots Road,London,SE93EF
4813824874,Sholeye,Olajide,14/03/2004,M,White and Black African,Nuts,0,1,Flat 26 Woodrow Court,69 Camberwell Station Road,London,SE59AZ
8967463139,Habibullah,Beverley,28/04/2010,M,White and Black African,Pollen|Water|Seafood,0,2,Potato Road Number 10,Chocolate House,Iraq,GHT13
6388236637,Timmy,Tom,15/08/1555,M,British,N/A,3,1,Timmytomville,Tomicity Town,Tomdon,TIY6
9848197844,Susan,Sally,03/04/2005,F,Northern Irish,Rasperries|Jam,4,22,3 Walkers Street,Avenue Close,Manchester,SW5690L
3919196324,Lilly,Smith,01/04/2007,F,British,N/A,2,1,5 Edgar Street,Walls Road,London,SE25G3
3752656695,Susan,Rose,04/02/2005,F,British,Apples|Chocolates|Strawberries,5,3,39 Hammersmith Town,Ealing Lane,Birmingham,SW194JW
9554744818,Rafaela,Mendes-Da-Silva,31/12/2003,F,Black African,N/A,0,0,17 Wilshaw House,Creekside Road,London,SE83YY
1249766451,Ryan,Akay,13/11/2005,F,Black African,Peanut,0,20,3 Meadow Lane,Right Street,Chestershire,SWKPLR
7529882792,Fajmaela,Aldi,12/09/2003,F,Black Carribean,Peanut,2,5,18 Falter House,Creative Road,London,SE91JM
1914236296,Mockaroo,Tonic,03/04/2005,M,European,N/A,32,3,9 Nice Road,Ivy Lane,Ealing,SE0OI2
4854166997,Kayla,Smith,01/04/2006,F,Black African,N/A,2,0,28 Martins Court,Thompsons Road,London,SE25AY
5564528166,Taylor,Blue,01/03/1992,F,South American,Pollen|Apples,5,2,3 Cool House,Highway Lane,Birmingham,SE4BW3
```

An example of the button working can be seen below:



A clearer screenshot of the output can be seen below:

NHS ID	Forename	Surname	Date of Birth	Gender	Ethnicity
1762883873	Farida	Addo	25/08/2004	F	Black African
4813824874	Sholeye	Olajide	14/03/2004	M	White and Black African
8967463139	Habibullah	Beverley	28/04/2010	M	White and Black African
6388236637	Timmy	Tom	15/08/1555	M	British
9848197844	Susan	Sally	03/04/2005	F	Northern Irish
3919196324	Lilly	Smith	01/04/2007	F	British
3752656695	Susan	Rose	04/02/2005	F	British
9554744818	Rafaela	Mendes-Da-Silva	31/12/2003	F	Black African
1249766451	Ryan	Akay	13/11/2005	F	Black African
7529882792	Fajmaela	Aldi	12/09/2003	F	Black Carribean
1914236296	Mockaroo	Tonic	03/04/2005	M	European
4854166997	Kayla	Smith	01/04/2006	F	Black African
5564528166	Taylor	Blue	01/03/1992	F	South American

Database

Allergies	Number of COVID-19 Vaccines	Number of COVID-19 Cases	Address Line One	Address Line Two	Town/City	Postcode
N/A	1.0	0.0	8 Pine Building	Scots Road	London	SE93EF
Nuts	0	1	Flat 26 Woodrow Court	69 Camberwell Station Road	London	SE59AZ
Pollen|Water|Seafood	0	2	Potato Road Number 10	Chocolate House	Iraq	GHT13
N/A	3	1	Timmytomville	Tomicity Town	Tomdon	TIY6
Rasperries|Jam	4	22	3 Walkers Street	Avenue Close	Manchester	SW5690L
N/A	2	1	5 Edgar Street	Walls Road	London	SE25G3
Apples|Chocolates|Strawberries	5	3	39 Hammersmith Town	Ealing Lane	Birmingham	SW194JW
N/A	0	0	17 Wilshaw House	Creekside Road	London	SE83YY
Peanut	0	20	3 Meadow Lane	Right Street	Chestershire	SWKPLR
Peanut	2	5	18 Falter House	Creative Road	London	SE91JM
N/A	32	3	9 Nice Road	Ivy Lane	Ealing	SE0OI2
N/A	2	0	28 Martins Court	Thompsons Road	London	SE25AY
Pollen|Apples	5	2	3 Cool House	Highway Lane	Birmingham	SE4BW3

REVIEW

The program successfully replicating the csv file in a clearer way met the requirement for 63% of questionnaire respondents who believed that data visualisation was important to them, as well as the success criteria to incorporate data visualisation.
An important note is that when initially designing the program, HTML was going to be incorporated as a form of data visualisation. After learning about Tkinter, there was no longer a need for HTML. This was beneficial as it prevented the need for there to be a separate file to store the HTML, which may have been confusing to operate as the HTML file would have to then be saved onto the computer and then opened in a browser. As a result, the usability of the program increased as there was now less work to do for the user. An additional benefit was the reduced memory requirements.

The final design matched the prototype created in the design section. Comparisons can be seen below:

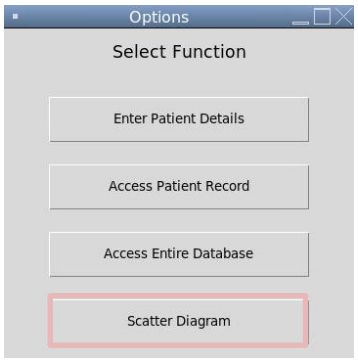
Forename	Surname	NHS Number	Age	Number of VAccines	COVID-19 Cases
Farida	Addo	9131696573	17	Y	0
Farida	Addo	6428473563	34	Y	0
July	Heather	5351331875	2	N	0
Sam	Johnson	9861212989	85	Y	6
Lola	Addo	3979872798	92	Y	34

NHS ID	Forename	Surname	Date of Birth	Gender	Ethnicity
1762883873	Farida	Addo	25/08/2004	F	Black African
4813824874	Sholeye	Olajide	14/03/2004	M	White and Black African
8967463139	Habibullah	Beverley	28/04/2010	M	White and Black African
6388236637	Timmy	Tom	15/08/1555	M	British
9848197844	Susan	Sally	03/04/2005	F	Northern Irish
3919196324	Lilly	Smith	01/04/2007	F	British
3752656695	Susan	Rose	04/02/2005	F	British
9554744818	Rafaela	Mendes-Da-Silva	31/12/2003	F	Black African
1249766451	Ryan	Akay	13/11/2005	F	Black African
7529882792	Fajmaela	Aldi	12/09/2003	F	Black Carribean
1914236296	Mockaroo	Tonic	03/04/2005	M	European
4854166997	Kayla	Smith	01/04/2006	F	Black African
5564528166	Taylor	Blue	01/03/1992	F	South American

After ensuring that this part of the program worked and was approved by Dr. Hussain, the next option was coded.

SCATTER DIAGRAM

POINTING TO THE SUBROUTINE



The next option that was developed was the 'Scatter Diagram' option. By importing the 'matplotlib.pyplot' function earlier, its functions were used to create the scatter diagram. This option is only accessible to the any user access level apart from 'Nurse', which is why the selection statement was used below. The part of the code which points to the subroutine which will display the scatter diagram is below:

```
def mainMenu1(accessLevel): #creates first main menu
  global mainMenu
  mainMenu = Toplevel(mainScreen) #creates screen using the one defined in the subroutine 'mainScreen'
  mainMenu.geometry("300x250") #sets dimesnison
  mainMenu.title("Options") #sets title
  Label(mainMenu, text="Select Function", width="300", height="2", font=("Calibri", 13)).pack() #sets screen title
  Label(mainMenu,text="").pack() #creates empty space between labels
  #creating buttons for user options
  Button(mainMenu,text="Enter Patient Details", height="2", width="30", command = patientDetails).pack() #creates button for function
  Label(mainMenu,text="").pack() #creates empty space between labels
  Button(mainMenu,text="Access Patient Record", height="2", width="30", command = patientLogin).pack() #creates button for function
  Label(mainMenu,text="").pack() #creates empty space between labels
  Button(mainMenu,text="Access Entire Database", height="2", width="30", command=entireDatabase).pack() #creates button for function
  Label(mainMenu,text="").pack() #creates empty space between labels
  if accessLevel != "Nurse": #does not show this button if the access level is 'Nurse'
    Button(mainMenu,text="Scatter Diagram", height="2", width="30", command=plot).pack() #creates button for function
```

CODE

The code for this subroutine can be seen below:

```
def plot(): #creates scatter diagram
  plt.figure(figsize=(3,3)) #sets dimensions of scatter diagram
  column = 0
  noOfVaccines = [] #sets an empty array for the number of covid vaccines
  noOfCovidCases = [] #sets an empty array for the number of covid cases
  with open("Patients.csv") as file: #opens the patient file
    record = [line.split(delimiter) for line in file] #splits the record each time a delimiter is encountered
    for element in record: #reads each element in the record
      column+=1
      if column == 1:
        continue #ignore header row
      noOfVaccines.append(element[7]) #adds the element number of vaccines to the array
      noOfCovidCases.append(element[8]) #adds the element number of covid cases to the array
  combinedLists = zip(noOfVaccines,noOfCovidCases)  #combines the two lists
  convertToList = list(combinedLists) #converts variable to list
  repetitions = Counter(convertToList) #determines how many times coordinates are repeated
  combinedLists = list(repetitions.keys()) #returns the values in order of insertion
  noOfVaccines = [] #creating a new array
  noOfCovidCases = [] #creating a new array
  for i in combinedLists: #goes through each pair
    noOfVaccines.append(i[0]) #adds the value in the zeroth position to the number of vaccines
    noOfCovidCases.append(i[1]) #adds the value in the zeroth position to the number of covid cases
  repetitions = list(repetitions.values()) #determines the number of coordinates for each coordinate
  x = [noOfVaccines]; y = [noOfCovidCases]; s = [repetitions] #sets scatter values
  plt.scatter(x,y,s) #adds plots to diagram
  plt.title("Relationship Between The Number of COVID-19 Vaccines and The Number of COVID-19 Cases") #setting title
  plt.xlabel("Number of COVID-19 Vaccines") #setting x axis
  plt.ylabel("Number of COVID-19 Cases") #setting y axis
  plt.show() #displays scatter diagram
```

Empty lists were created to store the values which are needed to create the diagram – the number of COVID-19 vaccinations, as well as the number of COVID-19 cases. The patient file is then read (with the same test data as before), with each record being split upon the occurrence of a delimiter (a comma in this case, but can be changed by future programmers), in order to treat each field item as a separate element. The use of the variable 'column' is to avoid the potential error where the first line is read. This would cause an error because the plot values are meant to be numerical, however the header value is a string value. As a result, the first line of the text file is dismissed as it does not contain any values which would be plotted on the diagram. In relation to the rest of the lines which would contain the desired values, the 7th and 8th elements are extracted, and appended to the designated lists. The zip function is then used to create co-ordinates. This is done by adding two values together. For instance, if the number of COVID-19 vaccines was 2, and the number of COVID-19 cases was 0, then this would be changed to '(2,0)'. The variable 'convertToList' then converts these values into a list so that their values can be manipulated. An example of what this looks like can be seen below:

'2,0' occurs twice

```
convert to list = [('1.0', '0.0'), ('0', '1'), ('0', '2'), ('3', '1'), ('4', '22'), ('2', '1'),
 ('5', '3'), ('0', '0'), ('0', '20'), ('2', '5'), ('2', '0'), ('2', '0'), ('5', '2')]
```

The 'Counter' function is then used to determine the number of times a co-ordinate occurs. It also removes reoccurring coordinates (see annotations in red). An example of what this looks like can be seen below:

'2,0' occurs once

```
repetitions = Counter({('2', '0'): 2, ('1.0', '0.0'): 1, ('0', '1'): 1, ('0', '2'): 1, ('3', '1'): 1, ('4', '22'): 1
, ('2', '1'): 1, ('5', '3'): 1, ('0', '0'): 1, ('0', '20'): 1, ('2', '5'): 1, ('5', '2'): 1})
```

The use of the 'key' function essentially puts the coordinates back into their original order after temporary rearrangement by the 'Counter' function. An example of what this looks like can be seen below:

```
combined lists = [('1.0', '0.0'), ('0', '1'), ('0', '2'), ('3', '1'), ('4', '22'), ('2', '1'), ('5', '3'), ('0', '0'),
 ('0', '20'), ('2', '5'), ('2', '0'), ('5', '2')]
```

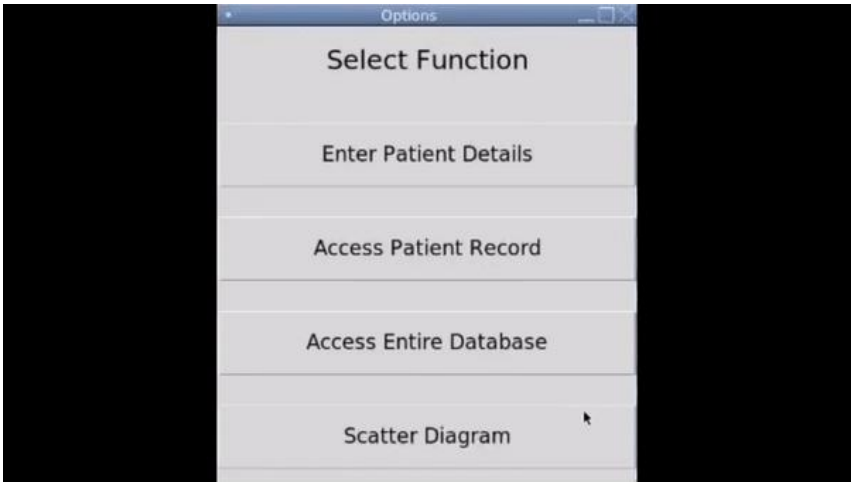
After the repetitions had been removed, 'combinedLists' was then iterated through, and the COVID-19 vaccination and case coordinates were added to new lists. This was because there was no need to store repeated co-ordinate values. The purpose of the second 'repetitions' variable was to store the number of occurrences of each value in the coordinates. The reason why repetition values were stored was because Dr. Hussain suggested that a good feature would be if the plot size could increase each time co-ordinates appeared more than once. This would allow analysts to see common patterns. For example, they might find that it was common for those with 2 vaccines to have 0 COVID-19 cases, demonstrating the effectiveness of receiving two doses of the vaccine. An example of what the 'repetitions' variable stores can be seen below:

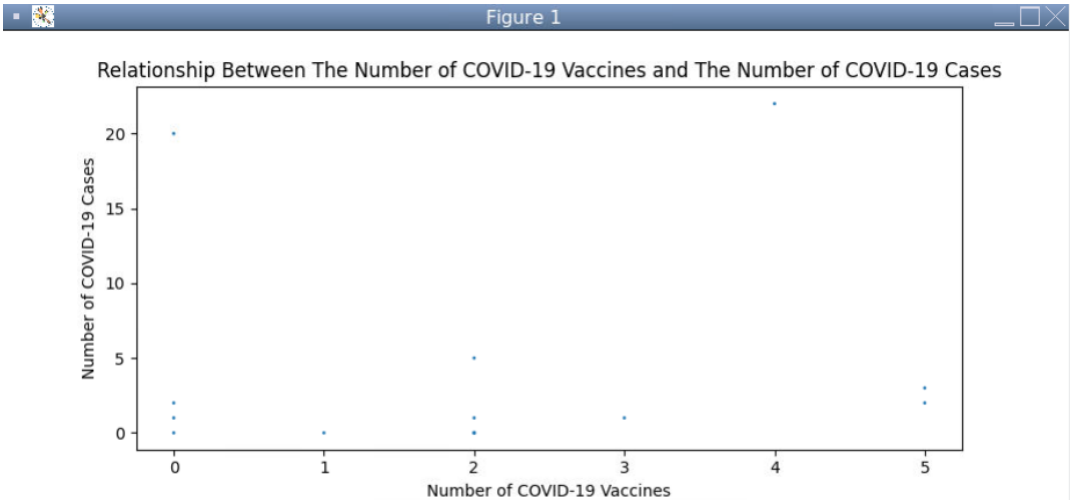
```
repetitions = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1]
```

After the coordinate and repetition values were collected, they were assigned values, as seen in the code above, in orange. The plot values were then added to the diagram, the x and y axes were labelled, and the screen was assigned a title. The last line 'plt.show()' was used to display the diagram.

TESTING

Once the coding was complete, a test was conducted to ensure that clicking the 'Scatter Diagram' option would result in the diagram being shown. The successful output screen can be seen below:

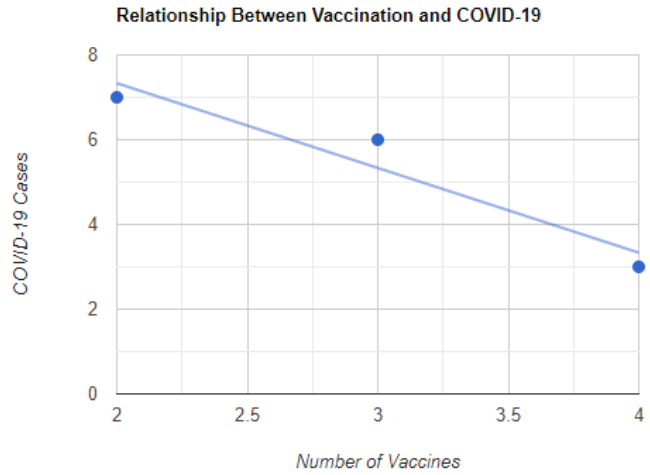


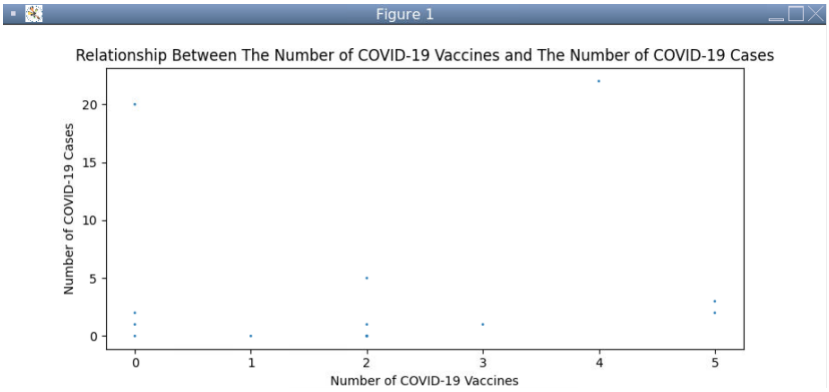


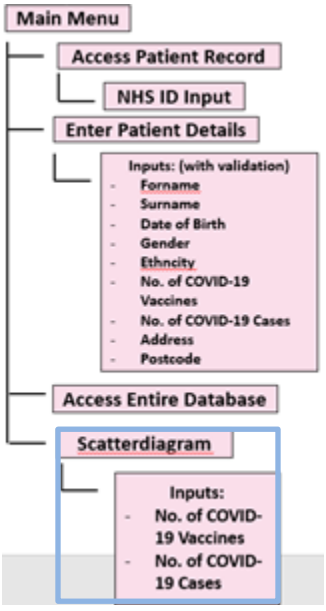
REVIEW

The creation of the scatter diagram met the requirement for 63% of questionnaire respondents who believed that data visualisation was important to them, as well as the success criteria to incorporate data visualisation. Another success criteria met was the creation of a diagram which represents the relationship between the number of COVID vaccines and the number of COVID cases had.

The final design matched the prototype created in the design section. Comparisons can be seen below:







The top-down diagram was referenced. The scatter diagram screen had been successfully created, which took the number of COVID-19 vaccinations and cases as inputs to plot the diagram.

After this part of the program was tested and approved by Dr. Hussain, the next option was coded.

```
procedure graph

    for i in file

        x = pos[m]

        y = pos[n]

        diagram.plot(x,y)
```

Finally, the pseudocode to the left displays the original code created in the design section. Whilst the final code somewhat resembles this, in relation to the code logic, it is also clear how simplified the pseudocode is when compared to the final code.

ENTER PATIENT DETAILS

POINTING TO THE SUBROUTINE

The next option that was developed was the 'Enter Patient Details' option. By selecting this function, users are able to create a record, setting the foundation for the database.

The part of the code which points to the subroutine which will display the screen is below:

```
def mainMenu1(accessLevel): #creates first main menu
  global mainMenu
  mainMenu = Toplevel(mainScreen) #creates screen using the one defined in the subroutine 'mainScreen'
  mainMenu.geometry("300x250") #sets dimesnison
  mainMenu.title("Options") #sets title
  Label(mainMenu, text="Select Function", width="300", height="2", font=("Calibri", 13)).pack() #sets screen title
  Label(mainMenu,text="").pack() #creates empty space between labels
  #creating buttons for user options
  Button(mainMenu,text="Enter Patient Details", height="2", width="30", command = patientDetails).pack() #creates button for function
  Label(mainMenu,text="").pack() #creates empty space between labels
  Button(mainMenu,text="Access Patient Record", height="2", width="30", command = patientLogin).pack() #creates button for function
  Label(mainMenu,text="").pack() #creates empty space between labels
  Button(mainMenu,text="Access Entire Database", height="2", width="30", command=entireDatabase).pack() #creates button for function
  Label(mainMenu,text="").pack() #creates empty space between labels
  if accessLevel != "Nurse": #does not show this button if the access level is 'Nurse'
    Button(mainMenu,text="Scatter Diagram", height="2", width="30", command=plot).pack() #creates button for function
```

CREATING SCREEN

The code for the screen design can be seen below:

```
def patientDetails(): #gathering patient detials
  global patientDetails
  patientDetails = Toplevel(mainScreen) #creates screen using screen from subroutine 'mainScreen'
  patientDetails.geometry("300x250") #sets dimensions
  patientDetails.title("Enter Patient Details") #sets title
  global forename
  forename = StringVar() #stores input
  Label(patientDetails, text="Forename:").grid(row = 0, column = 0, sticky = 'w') #creates label
  forename = tk.Entry(patientDetails, textvariable=forename) #creates entry box
  forename.grid(row = 0, column = 1) #sets position of entry box
  global surname
  surname = StringVar() #stores input
  Label(patientDetails, text="Surname:").grid(row =1, column =0, sticky = 'w') #creates label
  surname = tk.Entry(patientDetails, textvariable=surname) #creates entry box
  surname.grid(row = 1, column = 1) #sets position of entry box
  global dateOfBirth
  dateOfBirth = StringVar() #stores input
  Label(patientDetails, text="Date of Birth (DD/MM/YYYY):").grid(row =2, column = 0, sticky = 'w') #creates label
  dateOfBirth = tk.Entry(patientDetails, textvariable=dateOfBirth) #creates entry box
  dateOfBirth.grid(row = 2, column = 1) #sets position of entry box
  global gender
  gender = StringVar() #stores input
  Label(patientDetails, text="Gender (M/F):").grid(row = 3, column = 0, sticky = 'w') #creates label
  gender = tk.Entry(patientDetails, textvariable=gender) #creates entry box
  gender.grid(row =3, column = 1) #sets position of entry box
  global ethnicity
  ethnicity = StringVar() #stores input
  Label(patientDetails, text="Enter Number Relating To Ethnicity:").grid(row = 4, column = 0, sticky = 'w') #creates label
  Button(patientDetails, text="List of Ethnicities", width="12", height="1", command = ethnicitiesList).grid(row = 5, column = 0, sticky = 'w') #creates button to display ehtnicies to choose from
  ethnicity = tk.Entry(patientDetails, textvariable=ethnicity) #creates entry box
  ethnicity.grid(row = 4, column = 1) #sets position of entry box
  global allergies
  allergies = StringVar() #stores input
  Label(patientDetails, text="Allergies (Enter 0 If N/A):").grid(row = 6, column = 0, sticky = 'w') #creates label
  allergies = tk.Entry(patientDetails, textvariable=allergies) #creates entry box
  allergies.grid(row =6, column = 1) #sets position of entry box
  global numberOfCovidVaccines
  numberOfCovidVaccines = StringVar() #stores input
  Label(patientDetails, text="Number Of COVID-19 Vaccines:").grid(row = 7, column = 0, sticky = 'w') #creates label
  numberOfCovidVaccines = tk.Entry(patientDetails, textvariable=numberOfCovidVaccines) #creates entry box
  numberOfCovidVaccines.grid(row = 7, column = 1) #sets position of entry box
  global numberOfCovidCases
  numberOfCovidCases = StringVar() #stores input
  Label(patientDetails, text="Number of COVID-19 Cases:").grid(row =8, column = 0, sticky = 'w') #creates label
  numberOfCovidCases = tk.Entry(patientDetails, textvariable=numberOfCovidCases) #creates entry box
  numberOfCovidCases.grid(row = 8, column = 1) #sets position of entry box
  global addressLineOne
  addressLineOne = StringVar() #stores input
  Label(patientDetails, text="Address Line One:").grid(row = 9, column = 0, sticky = 'w') #creates label
  addressLineOne = tk.Entry(patientDetails, textvariable=addressLineOne) #creates entry box
  addressLineOne.grid(row = 9, column = 1) #sets position of entry box
  global addressLineTwo
  addressLineTwo = StringVar() #stores input
  Label(patientDetails, text="Address Line Two:").grid(row = 10, column = 0, sticky = 'w') #creates label
  addressLineTwo = tk.Entry(patientDetails, textvariable=addressLineTwo) #creates entry box
  addressLineTwo.grid(row = 10, column = 1) #sets position of entry box
  global addressLineTown
  addressLineTown = StringVar() #stores input
  Label(patientDetails, text="Town/City:").grid(row = 11, column = 0, sticky = 'w') #creates label
  addressLineTown = tk.Entry(patientDetails, textvariable=addressLineTown) #creates entry box
  addressLineTown.grid(row = 11, column = 1) #sets position of entry box
  global postcode
  postcode = StringVar() #stores input
  Label(patientDetails, text="Postcode:").grid(row = 12, column = 0, sticky = 'w') #creates label
  postcode = tk.Entry(patientDetails, textvariable=postcode) #creates entry box
  postcode.grid(row = 12, column = 1) #sets position of entry box
  buttonRefresh = Button(patientDetails, text="Done", width=10, height=1, command = detailsVerify).grid(row = 13, column = 0, sticky = 'w') #creates button which verifies details once pressed
```

The code consists of the same 5 repeated lines for each field value – made to replicate the design made earlier. Because the screenshot above is blurry due to the long lines of code, a specific section will be used to explain what the 5 lines do, as shown below:

```
def patientDetails(): #gathering patient detials
  global patientDetails
  patientDetails = Toplevel(mainScreen) #creates screen using screen from subroutine 'mainScreen'
  patientDetails.geometry("300x250") #sets dimensions
  patientDetails.title("Enter Patient Details") #sets title
  global forename
  forename = StringVar() #stores input
  Label(patientDetails, text="Forename:").grid(row = 0, column = 0, sticky = 'w') #creates label
  forename = tk.Entry(patientDetails, textvariable=forename) #creates entry box
  forename.grid(row = 0, column = 1) #sets position of entry box
```

The code begins by setting the screen geometry and title. It then creates the first label – forename. The variable ‘forename’ stores the user input. The same variable name is then used to store the entry box,

and its position. The box highlighted was copied and pasted to create the other entry boxes for the other inputs where the only difference was the variable names. Additionally, each variable was declared as global so they could then be used as arguments for input validation.

The code below is for the ethnicity entry box section:

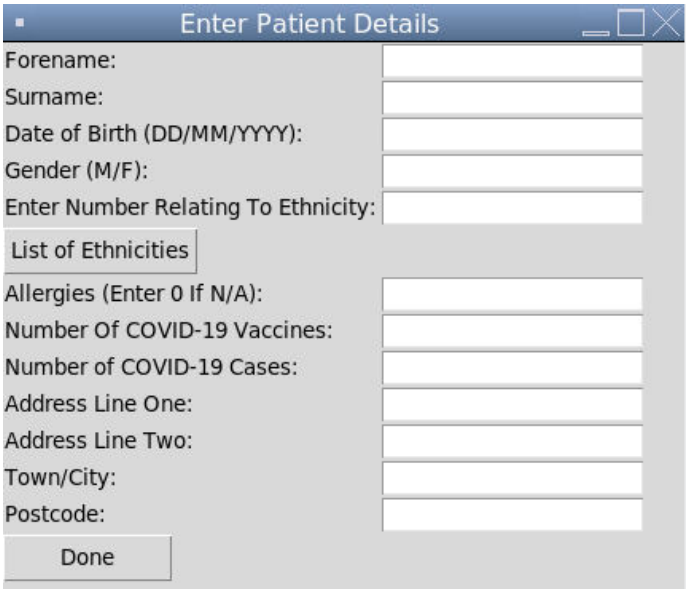
```
global ethnicity
ethnicity = StringVar() #stores input
Label(patientDetails, text="Enter Number Relating To Ethnicity:").grid(row = 4, column = 0, sticky = 'w') #creates label
Button(patientDetails, text="List of Ethnicities", width="12", height="1", command = ethnicitiesList).grid(row = 5, column = 0, sticky = 'w')
creates button to display ehtnicies to choose from
ethnicity = tk.Entry(patientDetails, textvariable=ethnicity) #creates entry box
ethnicity.grid(row = 4, column = 1) #sets position of entry box
```

As aforementioned, the code lines are same, except with different variable names and grid positions. The ethnicities section was pointed out because it includes an additional line – the creation of a button. The button is created so the user can see the list of eligible ethnicities they can choose from.

Finally, the last line of the subroutine creates a button, as shown below:

```
buttonRefresh = Button(patientDetails, text="Done", width=10, height=1, command = detailsVerify).grid(row = 13, column = 0, sticky = 'w')
#creates button which verifies details once pressed
```

The button points to the subroutine ‘detailsVerify’ which will verify the user inputs. Before making the subroutine, it was important to ensure that the screen design met the expectations of Dr. Hussain. The final screen design can be seen below:



After ensuring the ‘Enter Patient Details’ option pointed to the screen above, the entry validations were coded.

INPUT VALIDATION

CODE

The code for the data validation can be seen below:

```
def detailsVerify(): #validating patient details
  validateForename = nameValidate(forename.get().title(),1) #stores results of input validation
  validateSurname = nameValidate(surname.get().title(),2) #stores results of input validation
  validateDateOfBirth = dateOfBirthValidate(dateOfBirth.get()) #stores results of input validation
  validateGender = genderValidate(gender.get().capitalize()) #stores results of input validation
  validateEthnicity = ethnicityValidate(ethnicity.get()) #stores results of input validation
  validateAllergies = allergiesValidate(allergies.get().title()) #stores results of input validation
  validateNumberOfCovidVaccines = numberValidate(numberOfCovidVaccines.get(),1) #stores results of input validation
  validateNumberOfCovidCases = numberValidate(numberOfCovidCases.get(),2) #stores results of input validation
  validateAddressLineOne = nameValidate(addressLineOne.get().title(),3) #stores results of input validation
  validateAddressLineTwo = nameValidate(addressLineTwo.get().title(),4) #stores results of input validation
  validateAddressLineTown = nameValidate(addressLineTown.get().title(),5) #stores results of input validation
  validatePostcode = postcodeValidate(postcode.get().upper()) #stores results of input validation
  if validateForename == False or validateSurname == False or validateDateOfBirth == False or validateGender == False or validateEthnicity == False or
validateAllergies == False or validateNumberOfCovidVaccines == False or validateNumberOfCovidCases == False or validateAddressLineOne == False or
validateAddressLineTwo == False or validateAddressLineTown == False or validatePostcode == False: #if any of the inputs aren't valid
    Label(patientDetails, text="                              ").grid(row = 13, column = 1) #overwrites prior messages
  else:
   record = [validateForename,validateSurname,validateDateOfBirth,validateGender,validateEthnicity,validateAllergies,validateNumberOfCovidVaccines,
validateNumberOfCovidCases,validateAddressLineOne,validateAddressLineTwo,validateAddressLineTown,validatePostcode] #creating record
    Label(patientDetails, text="Record Added", fg="green", font=("calibri", 10)).grid(row = 13, column = 1) #displaying success message
    submitDetails(record) #passing details to patient class to sdd to csv file
    forename.delete(0, END) #clears entry box
    surname.delete(0, END) #clears entry box
    dateOfBirth.delete(0, END) #clears entry box
    gender.delete(0, END) #clears entry box
    ethnicity.delete(0, END) #clears entry box
    allergies.delete(0, END) #clears entry box
    numberOfCovidVaccines.delete(0, END) #clears entry box
    numberOfCovidCases.delete(0, END) #clears entry box
    addressLineOne.delete(0, END) #clears entry box
    addressLineTwo.delete(0, END) #clears entry box
    addressLineTown.delete(0, END) #clears entry box
    postcode.delete(0, END) #clears entry box
```

For each field entry, the code performs a validation on the entry and stores it in its corresponding variable. The use of functions like ‘title’, ‘upper’, and ‘capitalize’ are written so that each line of the record has the same format, sterilising the inputs. The use of formatting, i.e., upper for postcode, was created due to suggestions made by Dr. Hussain. This means that it doesn’t matter how the user inputs their data as it will be stored in the desired format.

The first condition of the if statement overwrites the message ‘Record Success’ (which would be there assuming that the user had previously entered patient details which were valid and written to file) with empty string, assuming that the user had previously entered a valid record. This is done because any invalid inputs would have an error message next to it, so it would not make sense for the previous success message to remain.

If the inputs are valid, the values are saved as the list ‘record’ and passed into a separate subroutine ‘submitDetails’ to be added into the patient file. A ‘Record Success’ message will also be displayed to the user. Afterwards, the contents of each entry box are cleared in order to allow the user to re-enter patient details without manually having to remove previous entries, thus saving time for the user.

NAME VALIDATION

CODE

The following section outlines how the validation for the forename, surname and address names inputs were coded.

Nowadays forenames and address names can take alphanumeric forms. Because of this, the program accounts for name inputs which may include numbers. If the program only allowed for string input, then

it may have limited applicability to users who need to enter patient details with alphanumeric characters. As a result, the only form of validation I had to ensure is that the entry boxes were not empty and were greater than 2 characters.

The subroutine was used to validate the forename, surname, addressLineOne, addressLineTwo, and town name inputs as they can all take on alphanumeric characters, even if it is not always the case. By using the subroutine to validate all of these inputs, coding time was shorter.

However, there was a problem. If the same subroutine was going to be used, then I had to ensure that there was some way to differentiate between the variable being validated. This is because the error message (for when the length is too short) needs to be placed next to the entry box for the specific input. As a result, I decided that the program would take two parameters: the variable being validated, as well as its corresponding number. The number is just used to distinguish between the input. The code can be seen below:

```
def nameValidate(variableBeingValidated,numberOfVariableBeingValidated): #validating the inputs: forename, surname, addressLineOne, addressLineTwo and
addressLineTown
  messsage = "Too Short" #error message
  if len(variableBeingValidated) < 2: #while the variable being validated has less than two characters
    if numberOfVariableBeingValidated == 1: #forename
      Label(patientDetails, text=messsage, fg="red", font=("calibri", 10)).grid(row = 0, column =3) #error message outputted
    elif numberOfVariableBeingValidated == 2: #surname
      Label(patientDetails, text=messsage, fg="red", font=("calibri", 10)).grid(row = 1, column =3) #error message outputted
    elif numberOfVariableBeingValidated == 3: #addressLineOne
      Label(patientDetails, text=messsage, fg="red", font=("calibri", 10)).grid(row = 9, column =3) #error message outputted
    elif numberOfVariableBeingValidated == 4: #addressLineTwo
      Label(patientDetails, text=messsage, fg="red", font=("calibri", 10)).grid(row = 10, column =3) #error message outputted
    elif numberOfVariableBeingValidated == 5: #Town/City name
      Label(patientDetails, text=messsage, fg="red", font=("calibri", 10)).grid(row = 11, column =3) #error message outputted
    return False
  if numberOfVariableBeingValidated == 1: #forename
    Label(patientDetails, text="                         ").grid(row = 0, column = 3) #overwrites previous messages
  elif numberOfVariableBeingValidated == 2: #surname
    Label(patientDetails, text="                         ").grid(row = 1, column = 3) #overwrites previous messages
  elif numberOfVariableBeingValidated == 3: #addressLineOne
    Label(patientDetails, text="                         ").grid(row = 9, column = 3) #overwrites previous messages
  elif numberOfVariableBeingValidated == 4: #addressLineTwo
    Label(patientDetails, text="                         ").grid(row = 10, column = 3) #overwrites previous messages
  elif numberOfVariableBeingValidated == 5: #Town/City name
    Label(patientDetails, text="                         ").grid(row = 11, column = 3) #overwrites previous messages
  return variableBeingValidated #returns the value of the validated variable as an element in the record
```

The number being passed as a parameter was used in the selection statement to determine the positioning of the error message. For example, if the current variable being validated is ‘1’ then the ‘Too Short’ message will be displayed beside the first entry box. The variable ‘message’ contains the error message. The reason for this is because it was easier to reuse the same message since the error would be the same, saving coding time.

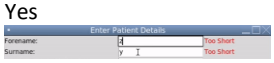
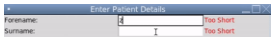
If the input is deemed as invalid, the value False is returned, which in turn will indicate that the value must be changed.

On the other hand, if the input length is at least 2 characters then any previous error messages in that position is overwritten with blank text, indicating that the input does not need to be changed.

The value of the variable being validated is then returned to the ‘detailsVerify’ subroutine for later use.

TESTING

After completing the code, it was time to ensure that it produced the desired output to certain user inputs. All of the input boxes the validation related to were tested at once.

Test	Reason	Test Data	Expected Outcome	Does the expected outcome occur?	Type of Test
Entering nothing into the entry box	The program is unable to create the record if there is no data given.	Inputting empty string	The program should output a 'Too Short' message, indicating to the user that their input is invalid.	Yes 	Erroneous as there is no user input.
Entering a 2 character input	The program should accept the inputs given as a field.	2 character string	No error messages. Any previous error messages should be removed.	Yes 	Boundary/extreme as 2 is the minimum number of characters which can be used.
Entering a 5 character input	The program should accept the inputs given as a field.	5 character string	No error messages. Any previous error messages should be removed.	Yes 	Normal/valid as an average of 5 characters may be expected.
Entering alphanumeric string	Both alphabetical and numerical values should be accepted.	Alphanumeric string	No error messages. Any previous error messages should be removed.	Yes 	Normal as the data is of an expected type.
Entering 1 character	It is an invalid length which should be changed	1 character string	The program should output a 'Too Short' message, indicating to the user that their input is invalid.	Yes 	Erroneous/invalid as the data is an invalid length.

After testing the code, the next validation section was developed.

DATE OF BIRTH VALIDATION

CODE

The following section outlines how the validation for the date of birth input was coded. The code can be seen below:

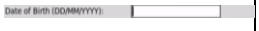
```
def dateOfBirthValidate(dateOfBirth): #validating date of birth input
  try:
      datetime.strptime(dateOfBirth, '%d/%m/%Y') #determines if the date of birth given is in the format '%d/%m/%Y'
  except ValueError or UnboundLocalError: #if the input is not in the correct format
    Label(patientDetails, text="Invalid Entry", fg="red", font=("calibri", 10)).grid(row = 2, column = 3) #displays and positions error message
    return False #the input is not valid
  Label(patientDetails, text="                         ").grid(row = 2, column = 3) #overwrites previous message
  return dateOfBirth #returns the value of the validated variable as an field in the record
```

The date of birth input is first passed as a parameter so that its value can be checked. The 'strptime' function is used to ensure that the input is in the format DD/MM/YYYY. If not, the 'Invalid Entry' message is outputted to the user.

By specifying the format required, there is a lower chance that a user will need to re-enter the input as they know the format for entering the data.

TESTING

After completing the code, it was tested to ensure that it produced the desired output to certain user inputs.

Test	Reason	Test Data	Expected Outcome	Does the expected outcome occur?	Type of Test
Entering nothing into the entry box	The program is unable to create the record if there is no data given.	Inputting empty string	The program should output a 'Invalid Entry' message, indicating to the user that their input is invalid.	Yes 	Erroneous as there is no user input.
Using dashes instead of slashes	The format has been explicitly states so anything not equal to it should not be accepted.	'25-08-2004'	An 'Invalid Entry' message, indicating to the user that their input is invalid.	Yes 	Erroneous as the wrong format is used.
Entering a day out of bounds	A date of birth with a day which does not exist should not be accepted as it is invalid. It would make it impossible for doctors to make predictions about a patients health based on a non-existent date of birth.	'32/08/2008'	An 'Invalid Entry' message, indicating to the user that their input is invalid.	Yes 	Erroneous an out of bounds value has been entered.
Entering a month out of bounds	A date of birth with a month which does not exist should not be accepted as it is invalid.	'25/32/2004'	An 'Invalid Entry' message, indicating to the user that their input is invalid.	Yes 	Erroneous an out of bounds value has been entered.
Entering a date of birth on the verge of being out of bounds	The values are not out of bounds so the system should not generate an error.	'31/12/2004'	The input should be accepted with any pre-existing messages cleared.	Yes 	Boundary as adding an extra number to the day or month would be erroneous.
Entering a normal date of birth	A valid input should be accepted by the system without generating errors.	'25/08/2004'	The input should be accepted with any pre-existing messages cleared.	Yes 	Normal as the value is what is expected
Entering a one digit month without a leading 0	Without the leading 0, the value is still valid so should be accepted as such.	'25/8/2004'	The input should be accepted with any pre-existing messages cleared.	Yes 	Normal as the value is what is expected

After testing the code, the next validation section was developed.

GENDER VALIDATION

CODE

The following section outlines how the validation for the gender input was coded. The code can be seen below:

```
def genderValidate(gender): #validate gender input
  if gender != "M" and gender != "F": #while the gender is not equal to male or female
    Label(patientDetails, text="Invalid Entry", fg="red", font=("calibri", 10)).grid(row = 3, column = 3) #displays and positions error message
    return False  #returns the value of the gender variable as an element in the record
  Label(patientDetails, text="                    ").grid(row = 3, column = 3) #overwrites previous message
  return gender #returns the value of the validated variable as a field in the record
```

The code starts by passing the gender input as a parameter. By previously calling the subroutine using the 'capitalize' function, the lines of code required to determine if the gender input was invalid was reduced. This is because the selection statement did not need to calculate whether the input was unequal to "m" or "f". `validateGender = genderValidate(gender.get().capitalize())`

By specifying the format required, there is a lower chance that a user will need to re-enter the input as they know the format for entering the data. Furthermore, only requiring one letter as an input saves the 'Patients' file memory space. Gender (M/F):

TESTING

After completing the code, it was tested to ensure that it produced the desired output to certain user inputs.

Test	Reason	Test Data	Expected Outcome	Does the expected outcome occur?	Type of Test
Entering nothing into the entry box	The program is unable to create the record if there is no data given.	Inputting empty string	The program should output a 'Invalid Entry' message, indicating to the user that their input is invalid.	Yes	Erroneous as there is no user input.
Entering an invalid input	The system should only accept the input in the format specified - "(M/F)".	'Male'	An 'Invalid Entry' message, indicating to the user that their input is invalid.	Yes	Erroneous as the wrong input is provided.
Entering a valid input	The system should accept the input when written in the specified format.	'F'	The input should be accepted and returned to the 'detailsVerify' subroutine, with any prior messages being removed.	Yes	Normal as the input is in the expected format.

After testing the code, next validation section was developed.

ALLERGIES VALIDATION

CODE

The following section outlines how the validation for the allergies input was coded. The code can be seen below:

```
def allergiesValidate(allergies): #validate allergy input
  numbers = ["1","2","3","4","5","6","7","8","9","0"] #creating a list to store numbers
  number = False #initialising number in allergy input as false
  for element in allergies: #reads each element in the input
    if element in numbers: #checks if the element is a number
      number = True #returns true if allergy input contains a number
      break
  if allergies == "0": #if the patient has no allergies
    Label(patientDetails, text="                         ").grid(row = 6, column = 3) #overwrites previus error message
    return "N/A" #the value in the allergy filed should be set to not applicable
  elif number == True or allergies == "" or len(allergies) < 2: #checks if entry is blank, contains a number or has a length of less than 2
    Label(patientDetails, text="Invalid Entry", fg="red", font=("calibri", 10)).grid(row = 6, column = 3) #displays error message
    return False #indicates that the value is invalid
  allergies = allergies.replace(" ,","|").replace(", ","|").replace("  ","") #replacing delimiter so the allergy input takes up one element in the list
  Label(patientDetails, text="                         ").grid(row = 6, column = 3) #overwrites previus error message
  return allergies #returns the value of the validated variable as a field in the record
```

The code begins by passing the allergy input as a parameter. The list ‘number’ is then used to store numerical values so that the allergies input could be iterated through to determine if it contained a numerical value present in the list ‘number’. The allergies input is then tested to determine if the input is ‘0’, where it would then be set to ‘N/A’ and be returned as a field value.

If the input is valid then each occurrence of a comma would be replaced by the symbol ‘|’. This was done because when acquiring the scatter diagram plots, the 7th and 8th element position is taken. However, if the allergy values contain commas, this could result in the wrong element, and in turn values being taken, leading to an error as the value collected would be a string and not integer.

The value of the validated input is then returned to the ‘detailsVerify’ subroutine for later use.

TESTING

After completing the code, it was tested to ensure that it produced the desired output to certain user inputs.

Test	Reason	Test Data	Expected Outcome	Does the expected outcome occur?	Type of Test
Entering nothing into the entry box	The program is unable to create the record if there is no data given.	Inputting empty string	The program should output a ‘Invalid Entry’ message, indicating to the user that their input is invalid.	Yes	Erroneous as there is no user input.
Entering a number	A number should not be inputted as it does not explain what the patient’s allergies are in particular.	‘3’	The program should output a ‘Invalid Entry’ message, indicating to the user that their input is invalid.	Yes 	Erroneous as the data is in the wrong format
Entering 0	Despite the fact that other numbers should not be accepted, 0 should be an accepted input.	‘0’	The value “N/A” is written to file and any previous error messages are removed.	Yes	Normal as the data is an example of what is expected
Entering allergy values	I had to ensure that the ‘|’ symbol would replace comma values.	‘Pineapples, Oranges’	The program should add the value ‘Pineapples|Oranges’ to the patient file	Yes 	Normal as the data is an example of what is expected.

After testing the code, the next validation section was developed.

ETHNICITY VALIDATION

CODE

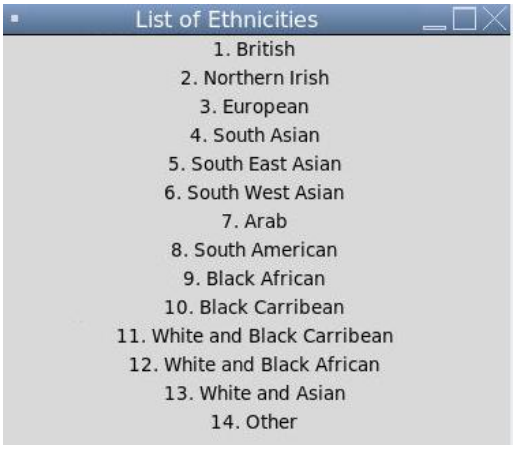
The following section outlines how the validation for the ethnicities input was coded.

First, a button which displayed the list of ethnicities to choose from was coded. The reason why the ethnicity choices are narrow is because different users may have different spellings for ethnicities, so having a specific value to choose from reduces the chance of inconsistencies in the database. It also means that ethnicities stored in the database can be universally understood, reducing chances of the misinterpretation of data.

The code for displaying the ethnicity button can be seen below:

```
def ethnicitiesList(): #displays list of ethnicities
  global ethnicitiesList
  ethnicitiesList = Toplevel(mainScreen) #creates screen using the one defined in 'mainScreen'
  ethnicitiesList.geometry("300x250") #sets timensions
  ethnicitiesList.title("List of Ethnicities") #sets title
  global validEthnicity
  validEthnicity = [] #sets the ethnicity as an array
  with open("Ethnicities.txt") as f: #opens the ethncities file
    for line in f: #reads through each line in the ethnicities file
      validEthnicity.append(line.replace("\n","")) #adds each line to the array declared earlier, and replaces the new line (at the end of each line)
with empty string to remove prevent "\n" from being added to the array
  global lengthEthnicity
  lengthEthnicity = len(validEthnicity) #finds the length of the array
  for i in range(lengthEthnicity): #continues to iterate for the length of the the ethnicity list
    Label(ethnicitiesList, text=(str(i+1)+". "+validEthnicity[i])).pack() #prints each line in validEthnicity, with a number and fulvaluesop before it
```

The screen design for ‘ethnicitiesList’ can be seen below, replicating the design made earlier:



The code for the ethnicity validation can be seen below:

```
def ethnicityValidate(ethnicity): #validating ethnicity
  if ethnicity == "":
    Label(patientDetails, text="Invalid Entry", fg="red", font=("calibri", 10)).grid(row = 4, column = 3) #displays error message if ethnicity is an empty string
    return False #indicates that the value is invalid
  lastDigit = int(repr(float(ethnicity))[-1]) #determines the last digit of the float number
  ethnicityLength = lengthEthnicity #stores ethnicity length
  ethnicity1 = validEthnicity #stored list of ethnicities
  if lastDigit != 0 or int(ethnicity) < 1 or int(ethnicity) > ethnicityLength:
    Label(patientDetails, text="Invalid Entry", fg="red", font=("calibri", 10)).grid(row = 4, column = 3) #error message
    return False #returns false if the input is not equal to one of the values specified in 'List of Ethnicities'
  validatedEthnicity = ethnicity1[int(ethnicity)-1] #returns ethnicity value
  Label(patientDetails, text="                              ").grid(row = 4, column = 3) #overwrites previous error messages
  return validatedEthnicity #returns the value of the validated variable as a field in the record
```

The number was stored as a float in order to account for float user inputs. By using the 'repr' function to determine the last digit, the system could ensure that no decimal numbers were valid inputs. The options to choose from were not decimal, so a decimal number should not be provided by the user.

Furthermore, the 'validatedEthnicity' variable uses the number inputted by the patient to collect the correct ethnicity from the ethnicities file. The use of '-1' is because arrays start from 0, so the number entered by the user would be one number above the ethnicity's position. This prevents the program from storing the wrong ethnicity.

The value of the validated input is then returned to the 'detailsVerify' subroutine for later use.

ERRORS

The following section explains an error which occurred when writing the ethnicity input into the patients file.

Whilst coding, I made sure to print the state of each record that was being written into the patient file. For the user input ethnicities, the field value can be seen below:

```
'2. Northern Irish\n',
```

Initially, the values in the ethnicities file were stored as seen below:

Ethnicities.txt ×

```
1. British
2. Northern Irish
3. European
4. South Asian
5. South East Asian
6. South West Asian
7. Arab
8. South American
9. Black African
10. Black Carribean
11. White and Black Carribean
12. White and Black African
13. White and Asian
14. Other
```

The old code for displaying the ethnicities can also be seen below:

```
num = validEthnicity[ethnicity-1]
```

The logic error here, was that the line printed included the number with it, as well as the '\n' text. As a result, I changed the state of the values in the ethnicities file to the one below:

```
British
Northern Irish
European
South Asian
South East Asian
South West Asian
Arab
South American
Black African
Black Carribean
White and Black Carribean
White and Black African
White and Asian
Other
```

By removing the numbers, I could ensure that the database would not store the number beside it. This was necessary as a user accessing a patient record may not know what the number beside the ethnicity corresponds to, and may think that it is important, when in reality it is only there to determine which ethnicity should be written to the patients file. A key feature of the program was ensuring that it was usable, so the workings of its implementation did not have to be of knowledge to the user.

After the number issue had been resolved, it was important to also remove the '\n' text that would occur when reading each line. This was done through the code below:

```
validEthnicity.append(line.replace("\n",""))
```

Each '\n' text was replaced with empty string.

As a result of these amendments, an example of the desired ethnicity value stored in a record can be seen below:

```
British
```

TESTING

After completing the code, it was tested to ensure that it produced the desired output to certain user inputs.

Test	Reason	Test Data	Expected Outcome	Does the expected outcome occur?	Type of Test
Entering nothing into the entry box	The program is unable to create the record if there is no data given.	Inputting empty string	The program should output a 'Invalid Entry' message, indicating to the user that their input is invalid.	Yes 	Erroneous as there is no user input.
Entering a decimal number	The list of ethnicities to choose from do not contain decimals, so the input should not be accepted.	'3.4'	The program should output a 'Invalid Entry' message, indicating to the user that their input is invalid.	Yes 	Erroneous as the data is of the wrong type.
Entering a number not in the list	The list of ethnicities to choose from have a maximum number limit of 14, so any input above this should not be accepted.	'56'	The program should output a 'Invalid Entry' message, indicating to the user that their input is invalid.	Yes 	Erroneous as the number is out of bounds
Entering a number on the upper boundary	The system should accept the number as a valid input, despite the fact that a number above it would generate an error message.	'14'	The program should add the value 'Other' to the patients file, and previous error messages should be removed.	Yes 	Boundary as the number is on the maximum which can be entered

Entering a number from 'ethnicitiesList'	The program should accept the input as it is valid.	'3'	The program should add the value 'European' to the patients file, and previous error messages should be removed.	Yes 	Normal as the data is of the expected value

After testing the code, develop the next validation section was developed.

POSTCODE VALIDATION

CODE

The following section outlines how the validation for the postcode input was coded. The code can be seen below:

```
def postcodeValidate(postcode): #validate postcode
  postcode = postcode.replace(" ","") #removes spaces in postcode
  if len(postcode) < 2  or len(postcode) > 7: #continues the loop until the postcode has a standard length
    Label(patientDetails, text="Invalid Length", fg="red", font=("calibri", 10)).grid(row = 12, column = 3) #displays error message
    return False #indicates that the value is invalid
  for x in range (len(postcode)): #goes through each value in the postcode
    if x == 0: #if the value is the first value
      isAlpha = postcode[x].isalpha() #determines if the first value is in the alphabet
      if isAlpha == False: #loops until the first value of the postcode is in the alpahbet
        Label(patientDetails, text="Invalid Format", fg="red", font=("calibri", 10)).grid(row = 12, column = 3) #displays error message
        return False #indicates that the value is invalid
  Label(patientDetails, text="                         ").grid(row = 12, column = 3)
  return postcode #returns the value of the validated variable as a field in the record
```

The 'replace' function was used to remove spaces. This is because it would then make it easier to determine the minimum and maximum length of string. After researching about the shortest and longest postcode lengths, the values '2' and '7' were chosen as the minimum and maximum valid length. An issue with validating the postcode input is the fact that they can take many forms, so it was difficult to determine what would make it invalid. However, one thing that was found was that each postcode begins with an alphabetical character, so this was the only form of validation, assuming that the string entered was within the accepted length range. As a result, the 'isalpha' function was used to determine if the first element of the input is an alphabetical character.

Furthermore, instead of the error message being 'Invalid Entry', it is 'Invalid Length' or 'Invalid Format'. This makes it clearer to the user about what they should rectify.

The value of the validated input is then returned to the 'detailsVerify' subroutine for later use.

TESTING

After completing the code, it was tested to ensure that it produced the desired output to certain user inputs.

Test	Reason	Test Data	Expected Outcome	Does the expected outcome occur?	Type of Test
Entering nothing into the entry box	The program is unable to create the record if there is no data given.	Inputting empty string	The program should output a 'Invalid Length' message, indicating to the user that their input is invalid.	Yes 	Erroneous as there is no user input.

Entering a 2-digit postcode	The program should allow a postcode, which is at the lower bound limit of characters, to be accepted.	'F3'	No error messages. Any previous error messages should be removed.	Yes 	Boundary as the number of characters is the minimum amount
Entering a 10-digit postcode	The program should not accept inputs longer than 7 characters.	'ABCDEFG HIJ'	The program should output a 'Invalid Length' message, indicating to the user that their input is invalid.	Yes 	Erroneous as the input exceeds the maximum length
Starting the postcode with a numerical character	A postcode which begins with a number does not meet the conditions of a valid postcode, so should not be treated as such.	'2EJ'	The program should output a 'Invalid Format' message, indicating to the user that their input is invalid.	Yes 	Erroneous as the input is of the wrong format
Entering a normal postcode	The program should allow a valid postcode to be entered. Furthermore, a space has been used to check if the input will be accepted. The space makes the length 8 characters, but with the removal of spaces, the input should be processed as valid.	'SE13 9EQ'	No error messages. Any previous error messages should be removed.	Yes 	Normal as the data is of the expected type

After testing the code, it was time to develop the next validation section.

NUMBERS VALIDATION

CODE

The following section will discuss how the validation for the number of COVID cases and vaccines input was coded.

The subroutine 'numberValidate' was used to validate the 'numberOfCovidVaccines' and 'numberOfCovidCases' inputs as they both take on numerical characters. By using the same subroutine to validate these inputs, coding time was shorter.

However, there was a problem. If the same subroutine was going to be used, then I had to ensure that there was some way to differentiate between the variables being validated. This is because the error message for an invalid input needs to be placed next to the entry box for the specific variable. As a result, I decided that the program would take two parameters: the variable being validated, as well as its corresponding number. The code can be seen below:

```
def numberValidate(variableBeingValidated,numberOfVariableBeingValidated): #validating the inputs: numberOfCovidVaccines and numberOfCovidCases
  string = False #sets string occurences as false
  if variableBeingValidated == "":
    if numberOfVariableBeingValidated == 1: #numberOfCovidVaccines
      Label(patientDetails, text="Invalid Entry", fg="red",font= ("calibri", 10)).grid(row = 7, column = 3) #displays error message
    else: #numberOfCovidVaccines
      Label(patientDetails, text="Invalid Entry", fg="red", font=("calibri", 10)).grid(row = 8, column = 3) #displays error message
    return False #returns false if input is empty
  for index in range (len(variableBeingValidated)):
    if variableBeingValidated[index].isalpha() == True:
      string = True #sets string equal to true if input contains an alphabetical character
      if numberOfVariableBeingValidated == 1: #numberOfCovidVaccines
        Label(patientDetails, text="Invalid Entry", fg="red",font= ("calibri", 10)).grid(row = 7, column = 3) #displays error message
      else: #numberOfCovidCases
        Label(patientDetails, text="Invalid Entry", fg="red", font=("calibri", 10)).grid(row = 8, column = 3) #displays error message
      return False #returns false if the input contains a string
  lastDigit = int(repr(float(variableBeingValidated))[-1]) #provides gives the last digit of the variable "validatedNumber"
  if lastDigit != 0 or int(variableBeingValidated) < 0: #checks if input is a float or greater than 1
    if numberOfVariableBeingValidated == 1: #numberOfCovidVaccines
      Label(patientDetails, text="Invalid Entry", fg="red", font=("calibri", 10)).grid(row = 7, column = 3) #displays error message
    else: #numberOfCovidCases
      Label(patientDetails, text="Invalid Entry", fg="red", font=("calibri", 10)).grid(row = 8, column = 3) #displays error message
    return False #indicates that the value is invalid
  if numberOfVariableBeingValidated == 1:#numberOfCovidVaccines
    Label(patientDetails, text="                         ").grid(row = 7, column = 3) #overwrites previous error message
  else: #numberOfCovidCases
    Label(patientDetails, text="                         ").grid(row = 8, column = 3) #overwrites previous error message
  return variableBeingValidated #returns the value of the validated variable as a field in the record
```

If the input does not contain a letter (determined through the use of the 'isalpha' function) and isn't empty, then the value is converted into a float in order to ensure that the number does not have a number other than '0' after the decimal point, indicating that it is a whole number. This is important as there is no such thing as having, for example, 2 and a half vaccines or 1 and a half COVID-19 cases.

In relation to length validation, the number only has to be greater than 0 - there is no maximum number which can be inputted. This is because, although it may be highly unlikely that someone has received 1000 vaccines, for example, it is not impossible, and therefore, the program should be able to account for this.

The value of the validated input is then returned to the 'detailsVerify' subroutine for later use.

TESTING

After completing the code, it was time to ensure that it produced the desired output to certain user inputs.

Test	Reason	Test Data	Expected Outcome	Does the expected outcome occur?	Type of Test
Entering nothing into the entry box	The program is unable to create the record if there is no data given.	Inputting empty string	The program should output a 'Invalid Entry' message, indicating to the user that their input is invalid.	Yes 	Erroneous as there is no user input.
Entering a negative number	The program should not accept a number below 0.	'-12'	The program should output a 'Invalid Entry' message, indicating to the user that their input is invalid.	Yes 	Erroneous as the input is of the wrong format
Entering 0	The program should accept 0 as the minimum number	'0'	No error messages. Any previous error messages should be removed.	Yes 	Boundary as the input is the minimum amount which

	that can be entered.				can be entered.
Entering a decimal number	The program should be able to detect that the number is not whole, through the use of the 'repr' function, and that the input is invalid.	'5.6'	The program should output a 'Invalid Entry' message, indicating to the user that their input is invalid.	Yes 	Erroneous as the data is of the wrong type.
Entering an alphabetic character	The program should not accept an alphabetical character as an input.	'abc'	The program should output a 'Invalid Entry' message, indicating to the user that their input is invalid.	Yes 	Erroneous as the data is of the wrong type.
Entering a whole number above 0	The program should accept a valid input	'15'	No error messages. Any previous error messages should be removed.	Yes 	Normal as the input is of the expected type.

After testing the code, it was time to develop the next validation section.

TESTING THE SCREEN

Now that the variables were validated, it was time to test the 'Enter Patient Details' screen. In order to ensure that the correct validated variables were being passed into the 'submitDetails' subroutine, the line 'print(record)' was temporarily added to check this. The code and output can be seen below:

```
    record = [validateForename,validateSurname,validateDateOfBirth,validateGender,validateEthnicity,validateAllergies,validateNumberOfCovidVaccines,
validateNumberOfCovidCases,validateAddressLineOne,validateAddressLineTwo,validateAddressLineTown,validatePostcode] #creating record
    Label(patientDetails, text="Record Added", fg="green", font=("calibri", 10)).grid(row = 13, column = 1) #displaying success message
    print(record)
```

All that was left to do was to create a valid record. The process of doing so can be seen below, with all the correct error messages being displayed, as well as removed:

One success of the code was that it enabled the lower case 'f' to be stored in a capitalised format, suggesting that the code below worked.

```
validateGender = genderValidate(gender.get().capitalize())
```

ERRORS

An error I found when testing the screen was that if a user attempted to enter a number relating to an ethnicity without having clicked the 'List of Ethnicities' button, the following message would appear:

```
    validateEthnicity = ethnicityValidate(ethnicity.get()) #stores results of input validation
  File "main.py", line 780, in ethnicityValidate
    ethnicityLength = lengthEthnicity #stores ethnicity length
NameError: name 'lengthEthnicity' is not defined
```

This is problematic as someone who is used to the system (due to its high memorability perhaps) may no longer need reminding of the ethnicities that the different numbers correspond to. As a result, I amended the ethnicities validation code so that this error did not persist.

I tried adding a call to the 'ethnicitiesList1' subroutine (which creates the 'List of Ethnicities' screen) into the 'ethnicityValidate' subroutine but found that the 'List of Ethnicities' screen would appear each time 'Done' was clicked, following an entry into the ethnicity number entry box. In order to resolve this issue, I started by separating the 'ethnicitiesList1' subroutine into two. This was because, I needed to gather information about the ethnicity length, and the valid ethnicities without the user seeing the list of ethnicities screen.

The separated subroutines can be seen below:

```
def determiningEthnicities():
  global validEthnicity
  validEthnicity = [] #sets the ethnicity as an array
  with open("Ethnicities.txt") as f: #opens the ethncities file
    for line in f: #reads through each line in the ethnicities file
      validEthnicity.append(line.replace("\n","")) #adds each line to the array declared earlier,
and replaces the new line (at the end of each line) with empty string to remove prevent "\n" from
being added to the array
  global lengthEthnicity
  lengthEthnicity = len(validEthnicity) #finds the length of the array
  return lengthEthnicity, validEthnicity #returns two values for use in 'ethnicityValidate' function

def ethnicitiesList1(): #displays list of ethnicities
  global ethnicitiesList
  ethnicitiesList = Toplevel(mainScreen) #creates screen using the one defined in 'mainScreen'
  ethnicitiesList.geometry("300x250") #sets timensions
  ethnicitiesList.title("List of Ethnicities") #sets title
  determiningEthnicities()
  for i in range(lengthEthnicity): #continues to iterate for the length of the the ethnicity list
    Label(ethnicitiesList, text=(str(i+1)+". "+validEthnicity[i])).pack() #prints each line in
validEthnicity, with a number and fulvaluesop before it
```

A benefit of making two subroutines is that despite making the subroutines separate, the list of ethnicities screen still remained the same, as by declaring 'validEthnicity' and 'lengthEthnicity' as global, they could be used beyond the scope of the subroutine to display the ethnicities for the users to choose from.

After creating the 'determiningEthnicities' subroutine, I attempted to use the global variables in my 'ethnicityValidate' subroutine, but was still presented with the error message:

```
NameError: name 'lengthEthnicity' is not defined
```

As a result, I included the return line (in green) in the 'determiningEthnicities' function in order to pass the values into the 'ethnicityValidate' subroutine.

Finally, all I had to do was amend the 'ethnicityValidate' code so that it would store the length of the ethnicity, as well as the validated ethnicity values. This can be seen below, in purple:

```
def ethnicityValidate(ethnicity): #validating ethnicity
  if ethnicity == "":
    Label(patientDetails, text="Invalid Entry", fg="red", font=("calibri", 10)).grid(row = 4, column = 3) #displays
error message if ethnicity is an empty string
    return False
  for character in ethnicity: #goes through each character in the ethnciity input
    if character.isalpha() == True: #checks if character is an alphabetical character
      Label(patientDetails, text="Invalid Entry", fg="red", font=("calibri", 10)).grid(row = 4, column = 3)
#displays error message
      return False #indicates that the value is invalid
  lastDigit = int(repr(float(ethnicity))[-1]) #determines the last digit of the float number
  lengthEthnicity, validEthnicity = determiningEthnicities() #stores ethnicityLength and validated
  if lastDigit != 0 or int(ethnicity) < 1 or int(ethnicity) > lengthEthnicity:
    Label(patientDetails, text="Invalid Entry", fg="red", font=("calibri", 10)).grid(row = 4, column = 3) #error
message
    return False #returns false if the input is not equal to one of the values specified in 'List of Ethnicities'
  validatedEthnicity = validEthnicity[int(ethnicity)-1] #returns ethnicity value
  Label(patientDetails, text="                          ").grid(row = 4, column = 3) #overwrites previous error messages
  return validatedEthnicity #returns the value of the validated variable as a field in the record
```

After adding this, there was no longer an error generated when a user entered an ethnicity number without having clicked the 'List of Ethnicities' button.

After that the 'Enter Patients Details' screen had been developed, tested, error checked, and approved by Dr. Hussain, the code was developed for writing the patient details to file.

SUBMITTING DETAILS

The following code will explain what happens in the 'submitDetails' subroutine, referenced in the 'detailsVerify' subroutine, as seen below:

```
def detailsVerify(): #validating patient details
  validateForename = nameValidate(forename.get().title(),1) #stores results of input validation
  validateSurname = nameValidate(surname.get().title(),2) #stores results of input validation
  validateDateOfBirth = dateOfBirthValidate(dateOfBirth.get()) #stores results of input validation
  validateGender = genderValidate(gender.get().capitalize()) #stores results of input validation
  validateEthnicity = ethnicityValidate(ethnicity.get()) #stores results of input validation
  validateAllergies = allergiesValidate(allergies.get().title()) #stores results of input validation
  validateNumberOfCovidVaccines = numberValidate(numberOfCovidVaccines.get(),1) #stores results of input validation
  validateNumberOfCovidCases = numberValidate(numberOfCovidCases.get(),2) #stores results of input validation
  validateAddressLineOne = nameValidate(addressLineOne.get().title(),3) #stores results of input validation
  validateAddressLineTwo = nameValidate(addressLineTwo.get().title(),4) #stores results of input validation
  validateAddressLineTown = nameValidate(addressLineTown.get().title(),5) #stores results of input validation
  validatePostcode = postcodeValidate(postcode.get().upper()) #stores results of input validation
  if validateForename == False or validateSurname == False or validateDateOfBirth == False or validateGender == False or validateEthnicity == False or
validateAllergies == False or validateNumberOfCovidVaccines == False or validateNumberOfCovidCases == False or validateAddressLineOne == False or
validateAddressLineTwo == False or validateAddressLineTown == False or validatePostcode == False: #if any of the inputs aren't valid
    Label(patientDetails, text="                    ").grid(row = 13, column = 1) #overwrites prior messages
  else:
    record = [validateForename,validateSurname,validateDateOfBirth,validateGender,validateEthnicity,validateAllergies,validateNumberOfCovidVaccines,
validateNumberOfCovidCases,validateAddressLineOne,validateAddressLineTwo,validateAddressLineTown,validatePostcode] #creating record
    Label(patientDetails, text="Record Added", fg="green", font=("calibri", 10)).grid(row = 13, column = 1) #displaying success message
    submitDetails(record) #passing details to patient class to sdd to csv file
    forename.delete(0, END) #clears entry box
    surname.delete(0, END) #clears entry box
    dateOfBirth.delete(0, END) #clears entry box
    gender.delete(0, END) #clears entry box
    ethnicity.delete(0, END) #clears entry box
    allergies.delete(0, END) #clears entry box
    numberOfCovidVaccines.delete(0, END) #clears entry box
    numberOfCovidCases.delete(0, END) #clears entry box
    addressLineOne.delete(0, END) #clears entry box
    addressLineTwo.delete(0, END) #clears entry box
    addressLineTown.delete(0, END) #clears entry box
    postcode.delete(0, END) #clears entry box
```

Once the subroutine has been called, it is assumed that the data entries are all valid. As a result, all that is left to do is add the record to the 'Patients.csv' file.

The code for the submitting the details can be seen below:

```
def submitDetails(detailsVerify): #enables user to submit the details of a new patient
  record = detailsVerify #jumps to another subroutine in order to collect inputs which will form the record
  patient = Patient(record) #provides the record to the Patient class, which will then combine the NHS ID with the other elements in the record
  patient.saveRecord() #jumps to a subroutine in the Patient class where the record is written to file
```

The validated variables are taken as a parameter, and are given the variable name 'record'. This is the point where the Patient class imported at the beginning of the program is now used. The 'patient' variable is used to store the creation of an object, or is an instantiation of the patient class. The final line of code saves the record to the Patients file. The code for how all of this works will be explained in the following section.

CREATING DATABASE – PATIENT CLASS

This section will refer to a separate python file made, which is linked to the main python file. This file is referred to as 'Patient.py'.

IMPORTING LIBRARIES

```
#importing library which can be referenced later
import random
```

The following library is initially imported, for later use in the program.

CLASS PATIENT

```
class Patient: #creates a class for each patient
```

The following code referenced will be subroutines in to the class 'Patient'. The reason why a separate class file has been made is because the process of creating the NHS ID, and adding the record to the Patients file is easier to handle in a separate python file. The main program file was long enough with the validations, so adding more to this may have made the code more difficult to manage for future developers. Furthermore, storing each instance of a Patient as an object also made logical sense.

CONSTRUCTOR

The following section will focus on the process of developing the constructor method. The code can be seen below:

```
def __init__(self, record): #creates a record where each field is derived from the record variable decleared in the main program
  self.forename = record[0] #stores forename input
  self.surname = record[1] #stores surename input
  self.dateOfBirth = record[2] #stores date of birth input
  self.gender = record[3] #stores gender input
  self.ethnicity = record[4] #stores ethnicity input
  self.allergies = record[5] #stores allergies input
  self.numberOfCovidVaccines = record[6] #stores number of COVID-19 vaccines input
  self.numberOfCovidCases = record[7] #stores number of COVID-19 cases input
  self.addressLineOne = record[8] #stores address line one input
  self.addressLineTwo = record[9] #stores address line two input
  self.addressLineTown = record[10] #stores town/city name input
  self.postcode = record[11] #stores postcode input
```

The purpose of the '__init__' method is to initialise the attributes of the patient class, so they can be used later. The 'record' value is passed as a parameter. The method then takes each element from the record list, and assigns it to attributes which represent the different field values.

Adding the constructor method enables instantiations to be made for the Patient class, and also makes it easier to determine, and keep track of the attributes of each Patient record.

GENERATING NHS ID

The following section will focus on the process of creating an NHS ID for a patient record. Having an NHS ID is important as it serves as a primary key for each record. This unique identifier ensures that each patient can be identified individually, so makes searching for patients easier.

INITIAL CODE

When developing the code initially and the user chose the option to enter patient details, the code would point to the subroutine 'uniqueid' to gather NHS ID details. This can be seen below:

```
def patient_record(): #creates record
  nhsId = uniqueid() #points to subroutine which generates NHS ID
```

The code for the subroutine can be seen below:

```
def uniqueid():
  nhsId = input("Does Patient have an NHS ID? (Y/N) ").upper()
  while nhsId != "Y" and nhsId != "N":
      print("Invalid input") #generates error message while user input is not "Y" or "N"
      nhsId = input("Does Patient have an NHS ID? (Y/N) ")
  if nhsId == "N" or nhsId == "NO": #creates NHS ID if it does not pre-exist
    nhsId = [] #creates empty list
    for x in range (0,10): #generates 10 digit NHS ID
      number_generator = random.randint(0,9) #generates a random number from 0 to 9
      nhsId.append(number_generator) #adds number to list
    result = "" #creates empty string
    for i in nhsId:
      result += str(i) #converts each number in generates NHS ID to string
    nhsId = result #stores remaining value as 'nhsID'
  else:
    nhsId = int(input("Enter NHS ID: ")) #user enters NHS ID if it is already known
  return nhsId #returns nhsId as a field in the record
```

The use of the imported random library can be seen to gather a random number between 0 and 9. This is important as the NHS ID should not be easily guessed, hence the use of 'randint' to generate a random combination.

After showing this code to Dr. Hussain, an issue arose. If a user is entering patient details, then they should not be asked if a patient has an NHS ID because the user does not have an existing record. As a result, the NHS ID should automatically be randomly generated. This meant that there was no need to add an 'Enter NHS ID' option to the list of user inputs required on the 'Enter Patient Details' screen.

UPDATED CODE

The updated code can be seen below:

```
#generates unique 10 digit ID
def generateNHSId(self):
  nhsId = [] #creates an empty array for the NHS ID
  for x in range (0,10): #generates a 10 digit number
    number_generator = random.randint(0,9) #generates a random number from 0 to 9
    nhsId.append(number_generator) #adds the value to the array
  newNhsId = "" #creates an empty string
  self.delimiter = "," #sets a delimiter
  for element in nhsId: #goes through each element in the nhs id
   newNhsId += str(element) #converts each element into a string as the value does not need to be treated as an integer
  with open("Patients.csv") as file: #opens patient file
    for line in file: #reads each line (record)
      line = line.split(self.delimiter) #splits the line upon an occurence of a delimiter so each element can be treated as a separate value
      if line[0] == newNhsId: #if the first element is equal to the NHS ID generated
        self.generateNHSId() #generates new nhs id if it has been taken
      else:
        continue
  return newNhsId #returns the nhs id as a field value for the record
```

The line 'newNhsId += str(element)' is used to convert each value into string. This is because appending to the list saves the values separately. An example can be seen below:

```
[7, 0, 1, 2, 3, 7, 2, 9, 3, 1]
```

However, the final NHS ID should all be concatenated, so they are converted into string so they can all be concatenated and form a whole 'number'. An example of this can be seen below:

```
7012372931
```

Validation in this section occurs where the 'Patients' file is opened and the first element of each line is read. If the generated value is equal to a pre-existing record's NHS ID, then the method will repeat until this is no longer the case. This is important, as having a unique primary key (NHS ID) for each patient record enables the unique identification of each patient, a point that was regarded to be important by Dr. Hussain.

After this section had been developed and approved by Dr. Hussain, the code for saving the record was developed.

SAVING RECORD

CODE

The following section will focus on writing the record into the 'Patients' file.

The code can be seen below:

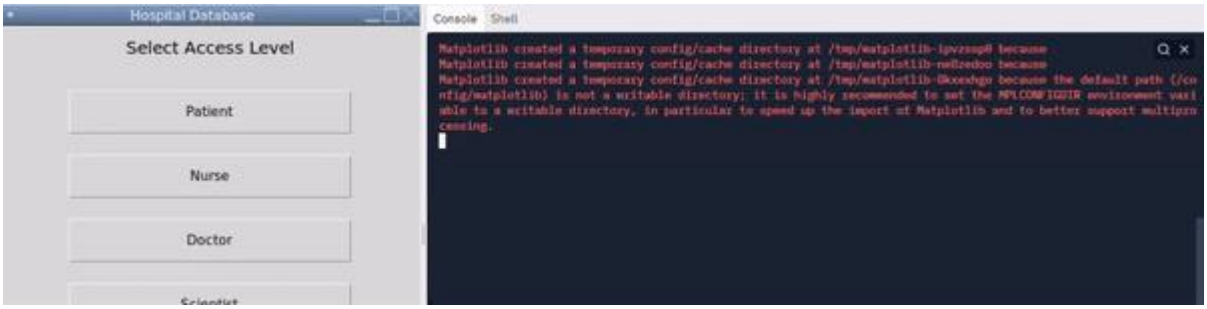
```
#save record
def saveRecord(self):
  self.nhsID = self.generateNHSId() #collects NHS ID from a separate subroutine
  self.delimiter = "," #sets a delimiter
  self.record = self.nhsID + self.delimiter + self.forename + self.delimiter + self.surname + self.delimiter +
self.dateOfBirth + self.delimiter + self.gender + self.delimiter + self.ethnicity + self.delimiter + self.allergies +
self.delimiter + self.numberOfCovidVaccines + self.delimiter + self.numberOfCovidCases + self.delimiter +
self.addressLineOne +  self.delimiter + self.addressLineTwo + self.delimiter + self.addressLineTown + self.delimiter +
self.postcode #adds each field to create a record
  with open ("Patients.csv", "a") as patientFile: #opens the patient file for appending
    patientFile.write("\n") #writes a new line to the file
    patientFile.write(self.record) #writes the record to the file
```

The NHS ID value is collected, and a delimiter value is set. The 'self.record' variable then stores each field value (defined in the constructor algorithm), with the NHS ID first, where there is a delimiter concatenated to each value. The patient file is then opened for appending with a new line value written to file, to make storing each record clearer on separate lines. The created record value is then written to file.

TESTING AND ERRORS

After coding this section, it was important the user inputs would be written to file. The 'Patients.csv' file I made manually was deleted to test whether a new file would be made, which would store the field values (from the code made at the start).

I started the program from the beginning, but when attempting to login, the following error occurred:




```
Exception in Tkinter callback
Traceback (most recent call last):
  File "/usr/lib/python3.8/tkinter/__init__.py", line 1883, in __call__
    return self.func(*args)
  File "main.py", line 172, in <lambda>
    Button(loginScreen, text="Login", width=10, height=1, command = lambda accessLevel = accessLevel:loginVerif
y(accessLevel)).grid(row = 2, column = 0) #creates button which onece pressed, enables user inputs to be verifi
ed
  File "main.py", line 234, in loginVerify
    line = line.strip("\n").split(delimiter) #removes "\n" from line
NameError: name 'delimiter' is not defined
```

Despite declaring ‘delimiter’ as global initially, the error message stated that it was not defined. As a result, I went back to the beginning of the code, and removed the following two lines from the else loop:

```
global delimiter
delimiter = "," #sets the delimiter in the patient file, which will be referenced later on

existenceOfFile = os.path.isfile("Patients.csv")  #determines if the patient file exists
if existenceOfFile == False:
    with open("Patients.csv", "w") as patientFile:
        string = "NHS ID, Forename, Surname, Age, Gender, Ethnicity, Allergies, Number of COVID-19 Vaccines, Number of COVID-19 Cases, Address Line One
Address Line Two, Town/City, Postcode"  #sets the first line of the patient file
        patientFile.write(string)  #writes the first line to file
else:
  global numberOfLines
  numberOfLines = 0  #setting the initial number of lines to 0
  with open("Patients.csv","r") as file:  #opens the patient file for reading
    for i in file:  #reads each line of the patient file
      numberOfLines += 1  #adds one to the variable each time there is an additional number of lines
```

After making this change, I was able to login as normal, and the record being written to file can be seen below:

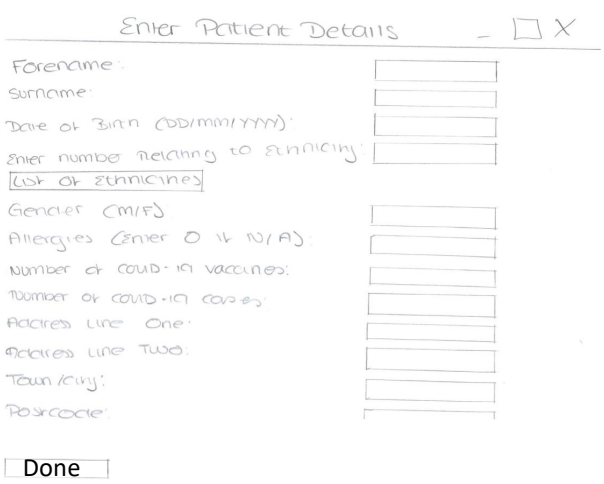
REVIEW

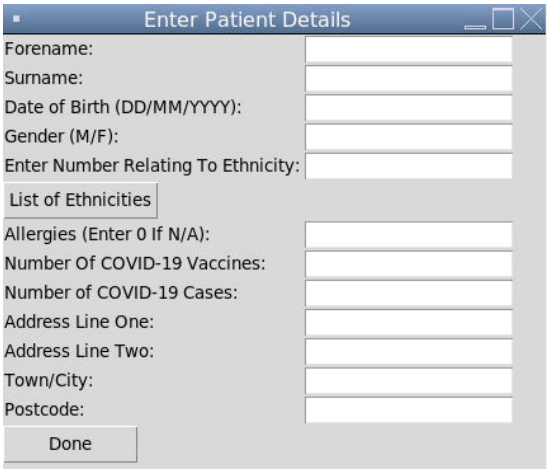
By the end of this section, the following success criteria were met:

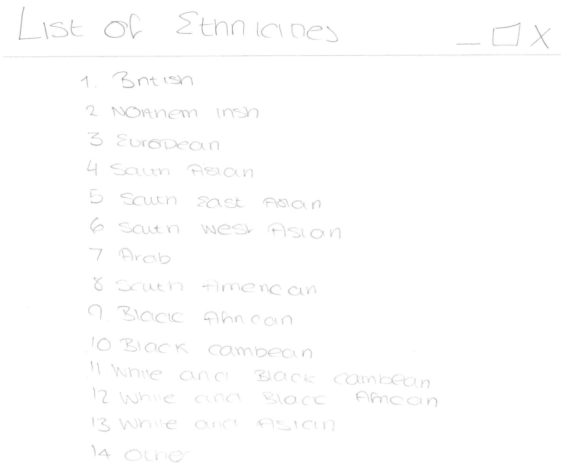
Gather inputs about a patient.
Enables several patient's details to be inputted.
Enable NHS IDs to be created.
Incorporate checks for NHS ID.
Write the patient data to file.
Incorporate checks for addresses.

Enabling several patient's details to be inputted was met as the done button 'refreshing' the entry boxes allows multiple data entries to be made. Additionally, the built-in Tkinter 'X' button means that the screen will not close until a user manually closes it.

The final design matched the prototype created in the design section. Some comparisons can be seen below:

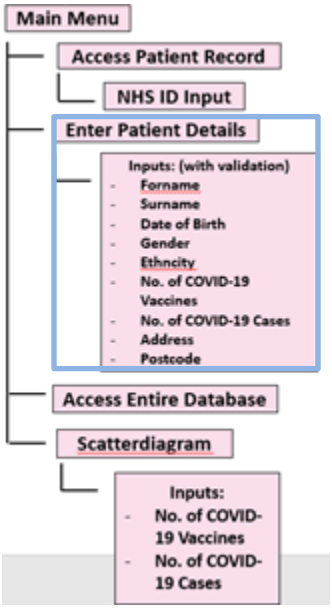




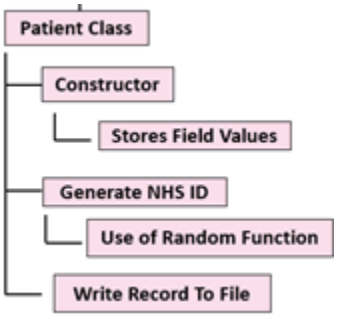


After completing the section, I referenced the top-down diagram in order to ensure that the program was developed in line with it.

Each of the branches had been successfully coded in line with the top-down diagram.



The 'Enter Patient Details' sub-branch had been coded. Within this, the forename, surname, date of birth, gender, number of COVID vaccines and cases, address and postcode inputs were coded for through the use of entry boxes, as well as their validation.



The patient class branch had been coded for. It consisted of a constructor which stored the field values as attributes. It also contained a method which generated an NHS ID through the use of the python random function. Finally, a method was included which writes the record to file.

```
import random

class Patient

    private procedure __init__(self, forename, surname, dateOfBirth, nhsId, vaccineNo, covidCases,
    allergies, gender, ethnicity, address)

        private forename

        private surname

        private dateOfBirth

        private nhsId

        private vaccineNo

        private covidCases

        private allergies

        private gender

        private ethnicity

        private address

    public procedure nhsId(nhsId)

        if nhsId == 0:

            unique_id = []

        for x in range 0 to 10

            number_generator = random.randint(1,9)

            unique_id.append(number_generator)

            result = ""

            for i in unique_id

                result += str(i)

            unique_id = result

            return unique_id

    writeToFile(self, forename, surname, dateOfBirth, nhsId, vaccineNo, covidCases, allergies,
gender, ethnicity, address):

        record =
forename+surname+dateOfBirth+nhsId+vaccineNo+covidCases+allergies+gender+ethnicity+address

        file.write(record)
```

The pseudocode for the patient class generally reflects the final code. The slight differences are regarding how the patient file is written. In the pseudocode, a method was called which takes in the attributes as parameters. However, in the final code there is no such thing – the 'saveRecord' subroutine reuses the attributes without having to pass them as parameters.

```
html_file.write("<HTML>")
html_file.write("<table border = 1>")
index = 0
file = open(patientfile)
procedure accessrecords(id):
        for line in patientfile:
        if id in line then
                print line
task = input("Enter 1 to access patient records and 2 to add new patient data")
if task = 1 then
id = input("Enter NHS ID")
        accessrecords(id)
else if task = 2 then
        number = input("How many patients would you like to enter for?")
        for x in range (number):
                forename = input()
                surname = input()
                date of birth = input()
                nhsId = input()
                vaccineNo = input()
                covidCases = input()
                allergies = input()
                gender = input()
                ethnicity = input()
                address = input()
        patient(forename, surname, age, nhsId, vaccineNo, covidCases, allergies, gender, ethnicity,
address)
for i in file:
        html_file.write("<TR>")
```

The main code pseudocode clearly underwent a significant change. For instance, the use of HTML was removed so the pseudocode regarding HTML was not required. Additionally, the iteration was not implemented as Tkinter was used, so inputs were gathered differently. Overall, the pseudocode for the main code was not of much use when developing the program due to how much the system requirements changed – i.e. change to Tkinter.

SELECTING VALUES FROM DATABASE

After creating the database, I consulted Dr. Hussain, who recommended a feature which would improve the usability of the database. Currently, in order to access multiple database records, the NHS ID must be entered multiple times, with the output screen also being closed each time. Dr. Hussain suggested that the user should be able to select certain records based on a common feature, such as all patients being under 18 for example. This would have high practical applications as the scope for doctors' searches increased. As a result, additions to the code were made.

AMENDING TOP-DOWN DIAGRAM

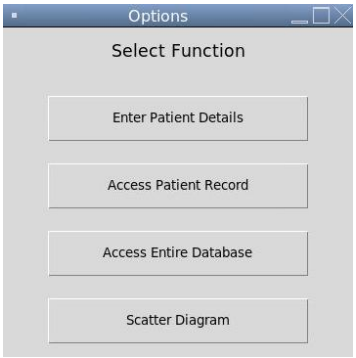
Before I could code this option, I needed to amend the top down diagram so that I knew which modules would be tackled at a time, independently. Amendments to the diagram can be seen in the design section.

POINTING TO THE SUBROUTINE

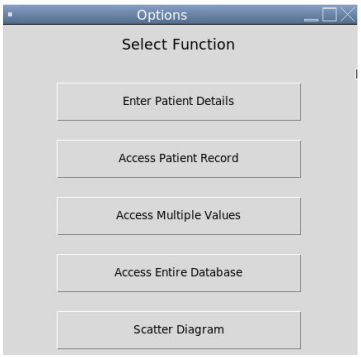
The first step was to add amend the 'options' page, so it would allow users to select an option which let them select multiple records at once. The additional code for adding the button can be seen below, in green:

```
def mainMenu1(accessLevel): #creates first main menu
  global mainMenu
  mainMenu = Toplevel(mainScreen) #creates screen using the one defined in the subroutine 'mainScreen'
  mainMenu.geometry("300x250") #sets dimesnison
  mainMenu.title("Options") #sets title
  Label(mainMenu, text="Select Function", width="300", height="2", font=("Calibri", 13)).pack() #sets screen title
  Label(mainMenu,text="").pack() #creates empty space between labels
  #creating buttons for user options
  Button(mainMenu,text="Enter Patient Details", height="2", width="30", command = patientDetails).pack() #creates button for function
  Label(mainMenu,text="").pack() #creates empty space between labels
  Button(mainMenu,text="Access Patient Record", height="2", width="30", command = patientLogin).pack() #creates button for function
  Label(mainMenu,text="").pack() #creates empty space between labels
  Button(mainMenu,text="Access Multiple Records", height="2", width="30", command = multipleRecords).pack() #creates button for function
  Label(mainMenu,text="").pack() #creates empty space between labels
  Button(mainMenu,text="Access Entire Database", height="2", width="30", command=entireDatabase).pack() #creates button for function
  Label(mainMenu,text="").pack() #creates empty space between labels
  if accessLevel != "Nurse": #does not show this button if the access level is 'Nurse'
    Button(mainMenu,text="Scatter Diagram", height="2", width="30", command=plot).pack() #creates button for function
```

Old home screen:

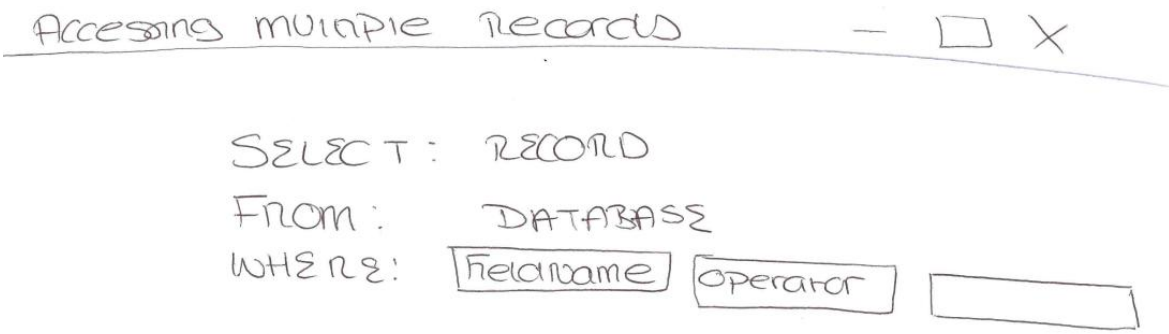


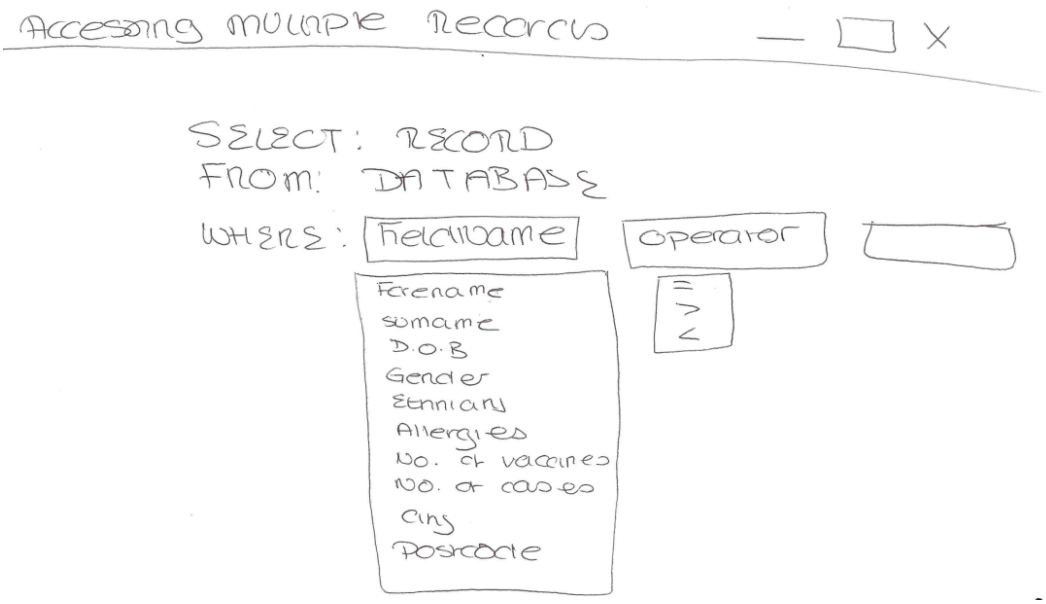
New home screen:



DESIGNING THE SCREEN

Before coding for the subroutine 'multipleRecords', which would handle the implementation of this option, the screen design had to be created. This can be seen below:





Further explanations for this design can be seen in the design section.

After showing the designs to Dr. Hussain, he suggested a further addition. He suggested that a user, such as himself, may want to see scatter diagram co-ordinate values for multiple users. As a result, the 'select' statement was amended in order to allow a drop-down menu to be incorporated – where the two options would be 'Record' or 'Scatter Diagram Co-Ordinates'.

This resulted in additions to the top down diagram, as well as the design of the screen, which is referenced and explained in the design section.

After approving the designs with Dr. Hussain, the screen for accessing multiple values was coded.

ACCESSING MULTIPLE VALUES SCREEN

CODE

```
def multipleValues():
```

The first thing to notice is the change in variable names from 'multipleRecords' (seen in the pointing to the subroutine section) to 'multipleValues'. Now that this option was dealing with multiple functionalities for the user, its name was changed, and made broader, to match this. This meant that the code pointing to the subroutine, as well as the option name, was changed (to match the new design), as seen in green:

```
Button(mainMenu,text="Access Multiple Values", height="2", width="30", command = multipleValues).pack() #creates button for function
```

```
def multipleValues():
  global screen
  screen = tk.Tk() #creates screen
  screen.geometry("300x250") #sets dimensions
  screen.title("Access Multiple Values") #sets title
  global inputEntry
  global input
  input = StringVar() #stores input value
  inputEntry = tk.Entry(screen, textvariable=input)
  inputEntry.grid(row = 3, column = 3, sticky = 'w')
  selectLabel = Label(screen, text="SELECT:").grid(row = 1, column = 0, sticky = 'w') #designs select label
  fromLabel = Label(screen, text="FROM:").grid(row = 2, column = 0, sticky = 'w') #designs from label
  databaseLabel = Label(screen, text="DATABASE").grid(row = 2, column = 1, sticky = 'w') #designs database label
  whereLabel = Label(screen, text="WHERE:").grid(row = 3, column = 0, sticky = 'w') #designs where label
```

The code begins by designing the screen and setting the labels for the 'SELECT', 'FROM', 'DATABASE', and 'WHERE' values. The user input entry box is also created, with the input stored in the variable 'input'.

```
global fieldNames
fieldNames = [
        "Forename",
        "Surname",
        "Date of Birth",
        "Gender",
        "Ethnicity Number",
        "Allergies",
        "Number of COVID-19 Vaccines",
        "Number of COVID-19 Cases",
        "City",
        "Postcode"
    ] #dropdown menu options

global fieldName
fieldName = tk.StringVar(screen) #sets datatype of menu text
fieldName.set("FieldName") #sets initial menu text
```

The values for the drop-down menu for the field names are then defined. The initial menu label is also set and stored as a string variable so its value can be used later on.

```
  global option
  option = tk.StringVar(screen) #sets datatype of menu text
  option.set("Options") #sets initial menu text

  options = [
  "Record",
  "Scatter Diagram Co-Ordinates"
  #dropdown menu options
]
  (OptionMenu(screen , option , *options,command=selected)).grid(row = 1, column = 1, sticky = 'w') #positions options menu button

  (OptionMenu(screen , fieldName , *fieldNames, command = selected ).grid(row = 3, column = 1, sticky = 'w') #positions fieldnames menu button

  operators1 = [
    "=",
    "!=",
    "<",
    "<=",
    ">",
    ">=",
  ] #sets menu options

  global operator
  operator = tk.StringVar(screen) #sets datatype of menu text
  operator.set("Operator") #sets initial menu text

  #creates dropdown menu
  (OptionMenu(screen , operator , *operators1 )).grid(row = 3, column = 2, sticky = 'w') #positions operators menu button

  Button(screen,text="Done",command=verifyEntry).grid(row=4,column=0,sticky='w') #validates user input
```

The ‘options’ and ‘operators’ menu buttons were then designed. The drop-down menus were then created and positioned onto the screen. There were some additions i.e. “!=” to the operators values as Dr. Hussain suggested that the “=”, “<”, and “>” symbols were too limited. Therefore, the use of other operators, such as “>=” provides greater functionality for a user. For instance, it allows users to select a record where the date of birth is greater than or equal to 25/08/2004. Finally, a ‘Done’ button is created which validates user entries once clicked, by pointing to another subroutine (in blue).

A command function was used in order to store the values of each menu item clicked by the user. This resulted in the creation of a subroutine ‘selected’, where its referencing can be seen in green.

```
def selected(choice): #determines value chosen by user
  operators = operators1 #stores operator vaue
  global selectChoice
  selectChoice = option.get() #stores option value
  if selectChoice == "Scatter Diagram Co-Ordinates":
    fieldNames = [
      "Number of COVID-19 Vaccines",
      "Number of COVID-19 Cases"
    ] #reduces fieldname choice if user wants to access scatter diagram co-ordinates
```

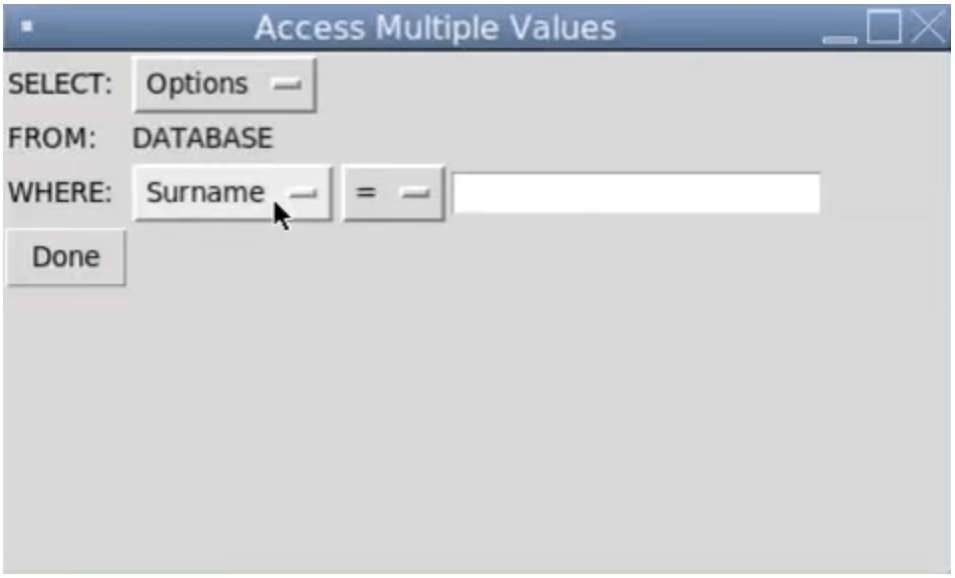
The ‘selected’ procedure takes in the user’s menu selection choice as a parameter. It stores the operator and option choice values. When a user’s option choice value is ‘Scatter Diagram Co-Ordinates’, the fields that the user could search for were initially limited to ‘Number of COVID-19 Vaccines’ and ‘Number of COVID-19 Cases’. However, after showing this to Dr. Hussain, he argued that a doctor may want to study

the correlation between COVID-19 cases and vaccines within a specific gender, for instance. As a result, this if statement was removed in order for doctors to be able to spot trends within certain demographics.

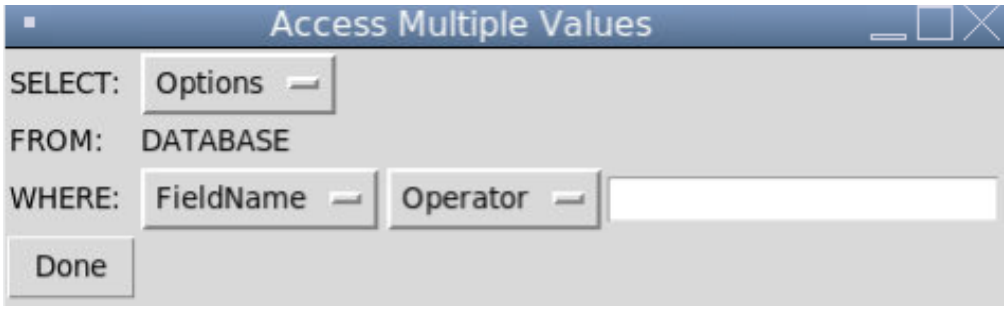
```
  def selected(choice): #determines value chosen by user
    operators = operators1 #stores operator vaue
    global selectChoice
    selectChoice = option.get() #stores option value
    global fieldChoice
    fieldChoice = fieldName.get() #stores fieldname choice
    if fieldChoice == "Forename" or fieldChoice == "Surname" or fieldChoice == "Gender" or
fieldChoice == "Allergies" or fieldChoice == "Town/City" or fieldChoice == "Postcode":
      operators = [
      "=",
      "!="
    ] #dropdown menu options
    if choice == "Ethnicity Number":
      ethnicitiesList1() #displays ethnicites list if ethnicity number field name is chosen
    global operatorChoice
    operatorChoice = operator.get() #stores operator choice
    (OptionMenu(screen , operator , *operators,command=selected)).grid(row = 3, column = 2, sticky
 = 'w') #replaces previous operator drop-down menu
```

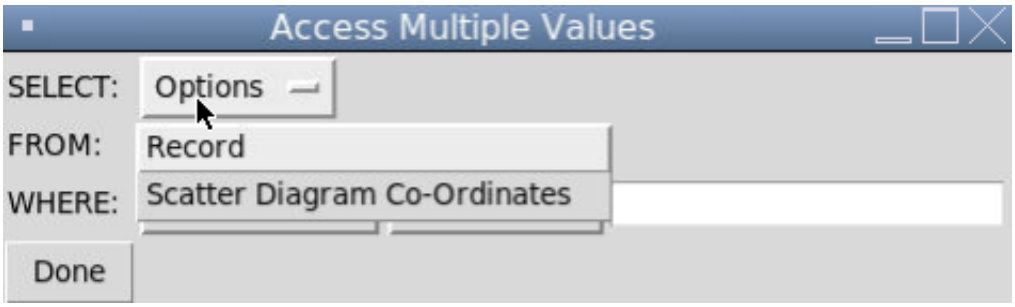
The field choice value is stored because it enables the operator values to be limited to an '=' and "!=" if the field value chosen is 'Forename', 'Surname', 'Gender', 'Allergies', or 'City'. Because these values would not be integers, there is no need for a user to be able to select '>'. For instance, a forename greater than 3 is hard to quantify. The 'operators' variable stores the operator menu options. If the selection statement is met, these operator values displayed will change. The final operator choice is stored in the variable 'operatorChoice'.

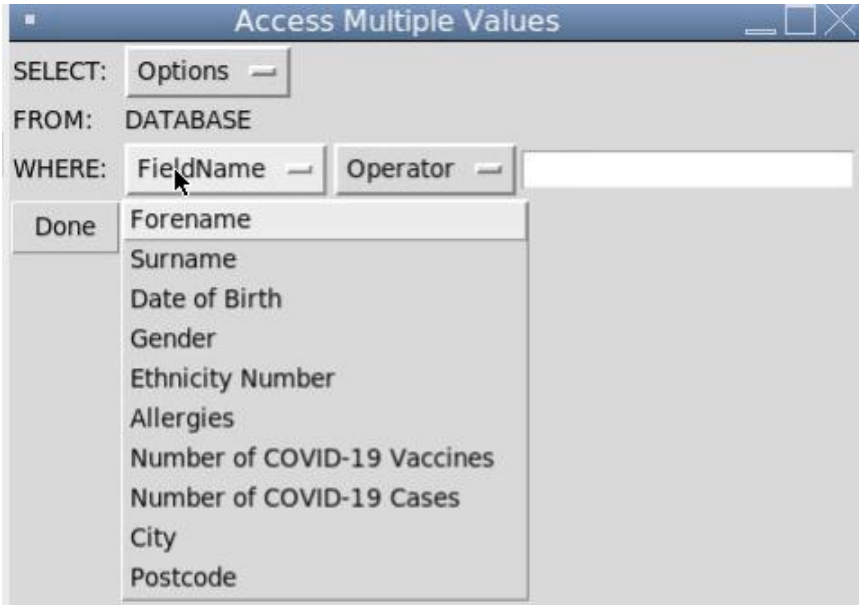
Furthermore, if the choice is 'Ethnicity Number', the ethnicities list will be displayed. Once again, this is because there are many variations of ethnicity spellings, so providing users with a number to enter which is equivalent to a particular ethnicity removes such problems. This output can be seen below:

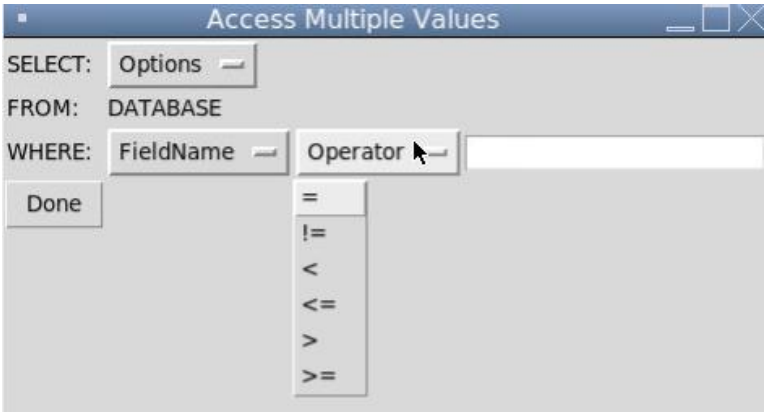


The final designs for the 'Accessing Multiple Values' screen can be seen below:



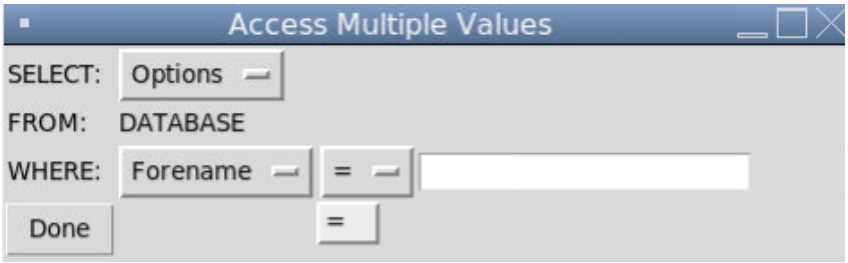






TESTING

The only ‘test’ carried out was ensuring that the inputs ‘Forename’, ‘Surname’, ‘Gender’, ‘Allergies’, or ‘City’ would only have the ‘=’ operator. An example of this being successful can be seen below.



ERRORS

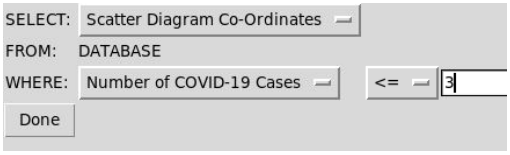
```
  def selected(choice): #determines value chosen by user
    operators = operators1 #stores operator vaue
    global selectChoice
    selectChoice = option.get() #stores option value
    global fieldChoice
    fieldChoice = fieldName.get() #stores fieldname choice
    if fieldChoice == "Forename" or fieldChoice == "Surname" or fieldChoice == "Gender" or
fieldChoice == "Allergies" or fieldChoice == "Town/City" or fieldChoice == "Postcode":
      operators = [
      "=",
      "!="
    ] #dropdown menu options
    if choice == "Ethnicity Number":
      ethnicitiesList1() #displays ethnicites list if ethnicity number field name is chosen
    global operatorChoice
    operatorChoice = operator.get() #stores operator choice
    (OptionMenu(screen , operator , *operators,command=selected)).grid(row = 3, column = 2, sticky
 = 'w') #replaces previous operator drop-down menu
```

The 'command=selected' line (in orange) was required as without it, the value of the operator would not update. It is important that it updates so the user can see what they selected. I discovered this because I had incorporated a line which printed the stored values each time they were clicked, as seen below:

```
print(selectChoice, operatorChoice, fieldChoice)
```

The initial code and printed statement can be seen below:

```
    (OptionMenu(screen , operator , *operators)).grid(row = 3, column = 2, sticky
= 'w')
```




```
Scatter Diagram Co-Ordinates Opera
tor Number of COVID-19 Cases
```

Should say '<=' not 'Operator'

The amended code and printed statement can be seen below:

Addition of 'command' line

```
    (OptionMenu(screen , operator , *operators,command=selected)).grid(row = 3,
column = 2, sticky = 'w')
```


```
Record < Number of COVID-19 Cases
```

Correct updated operator symbol

After I tried to call the ethnicitiesList screen subroutine (by selecting 'Ethnicity Number' as a field choice) I encountered another error. The code and output can be seen below:

```
if fieldd == "Ethnicity Number":
  ethnicitiesList()
```

```
TypeError: 'Toplevel' object is not cal
lable
```

Because I had given both the screen, and subroutine the same name, the screen could not be displayed. As a result, I changed the name of the 'ethnicitiesList' subroutine by adding a one to it, and amended this to the call statement. This can be seen below:

```
def ethnicitiesList1(): #displays list of ethnicities
  global ethnicitiesList
```

```
if choice == "Ethnicity Number":
  ethnicitiesList1() #displays ethnicites list if ethnicity number field name is chosen
```

As a result of the change in variable name, I also had to change the call to this variable in the 'Enter Patients Details' screen.

```
  global ethnicity
  ethnicity = StringVar() #stores input
  Label(patientDetails, text="Enter Number Relating To Ethnicity:").grid(row = 4,
column = 0, sticky = 'w') #creates label
  Button(patientDetails, text="List of Ethnicities", width="12", height="1", command =
ethnicitiesList1 .grid(row = 5, column = 0, sticky = 'w') #creates button to display
ehtnicies to choose from
```

After the code had matched the design, and had been tested and approved by Dr. Hussain, the validation of the inputted data was developed.

VALIDATION

CODE

The 'Done' button pointed to a 'verifyEntry' subroutine, as shown below:

```
Button(screen,text="Done",command=verifyEntry).grid(row=4,column=0,sticky='w') #validates user input
```

The code for the subroutine will be broken down.

```
def verifyEntry(): #validates menu/entry values
  global inputValue
  inputValue = inputEntry.get() #stores user input
  error = False #sets flag as false
```

Initially, the input value is collected, and the variable 'error' is set as a flag to determine whether the input is valid or not.

```
try: #stores values
  selectChoice == "Options"
  fieldChoice == "FieldName"
  operatorChoice == "Operator"
except NameError: #reports error if values have not been changed
  message = "Menu Value Must Be Amended" #sets error message
  error = True #sets error to true
```

A try/except statement is then used. This is because, the first time a user clicks 'Done', assuming that they have not changed any values of the drop-down menus, a NameError would appear, such as the one below:

```
NameError: name 'selectChoice' is not defined
```

As a result, the try/expect statement is used to output an error message to the user. The error flag is also set to true for later use.

```
  else:
    if selectChoice == "Options" or fieldChoice == "FieldName" or operatorChoice == "Operator":
#checks menu values
      message = "Menu Value Must Be Amended" #reports error if menu values have not been changed
      error = True #sets error to true
    if fieldChoice == "Forename" or fieldChoice == "Surname" or fieldChoice == "Gender" or
fieldChoice == "Allergies" or fieldChoice == "Town/City" or fieldChoice == "Postcode":
      if operatorChoice != "=" and operatorChoice != "!=":
        message = "Operator Value Must Be Amended" #reports error if menu values have not been
changed
        error = True #sets error to true
    if inputValue == "": #checks if entry box is empty
      message = "Invalid Entry" #sets error message
      error = True #sets error to true
```

The try/except statement was extended. I found that if the drop-down menu values have not been changed the second time 'Done' is pressed, it does not generate a name error. This is incorrect as the values must be changed so the program knows what to output to the user. As a result, the first selection statement within the 'else' statement checks if the drop-down menu values have not been amended, where an error will be generated. Additionally, a check was incorporated to ensure that a fieldChoice of 'Forename', 'Surname', 'Gender', 'Town/City', or 'Allergies' had the operator choice '=' or '!=' – the only accepted operators. Furthermore, if the input entry contains empty string, then an error message is set.

```
if error == True: #performs code if error is found
  Label(screen, text="                                                    ").grid(row = 3, column =4) #overwrites previous messages
  Label(screen, text=message,fg='red').grid(row = 3, column =4) #displays error message
  return False #ends function
```

If an error is present, then an error message is displayed beside the entry box. The reason why the 'message' variable was created is because it shortened the lines of code. If I had not done this, then the same error output statement would have been repeated for the different errors which occurred. 'False'

is then returned to leave the function. This is because the input is invalid, so it should not be used within the subroutine any longer to access multiple values, and the subroutine can be left. This statement ('return False') is featured commonly within the subroutine.

```
if fieldChoice == "Number of COVID-19 Cases" or fieldChoice == "Number of COVID-19 Vaccines" or fieldChoice == "Ethnicity Number":
  try: #determines if number is integer if specific fieldChoice which requires an integer is chosen
    int(inputValue)
  except ValueError:
    Label(screen, text="                                                    ").grid(row = 3, column =4) #overwrites previous messages
    Label(screen, text="Must Be A Whole Number",fg='red').grid(row = 3, column =4) #displays error message
    return False #ends function
  if fieldChoice == "Number of COVID-19 Cases" or fieldChoice == "Number of COVID-19 Vaccines":
    if int(inputValue) < 0: #checks if input is less than 0, which would be invalid
      Label(screen, text="                                                    ").grid(row = 3, column =4) #overwrites previous messages
      Label(screen, text="Entry Must Be Greater Than One",fg='red').grid(row = 3, column =4) #displays error message
      return False #ends function
  else:
    if int(inputValue) <= 0 or int(inputValue) > numberOfLines-1: #checks if input is within range of list of ethnicites options
      Label(screen, text="                                                    ").grid(row = 3, column =4) #overwrites previous messages
      Label(screen, text="Entry Must Be Between 1 and "+str(numberOfLines-1),fg='red').grid(row = 3, column =4) #displays error message
      return False #ends function
```

The field values 'Number of COVID-19 Vaccines', 'Number of COVID-19 Cases', and 'Ethnicity Number' all require inputs which are numbers. As a result, another try/except statement is used to ensure that an integer value above 0 is entered. If the fieldChoice is 'Ethnicity Number', then the input should be derived from the list of ethnicities, which only has values from 1 to the maximum number of ethnicities listed. As a result, any numbers other than this will result in an error message being returned. The use of 'numberOfLines' is in order to account for changes to the number of ethnicities listed in the 'List of Ethnicities'. If someone decided to add ethnicities, then this would be added onto the 'List of Ethnicities' screen, so the maximum number allowed would be changed. Therefore, 'numberOfLines' is used as opposed to a predetermined number. Subtracting 'numberOfLines' by 1 (in green) was due to python storing indexes from 0, whereas the user input would start from 1.

```
  global inputValue2
  if fieldChoice == "Ethnicity Number": #performs selection if fieldChoice is
ethnicity number
    with open("Ethnicities.txt","r") as file: #opens ethnicities file
      lines = file.readlines() #reads all lines of file
      inputValue2 = lines[int(inputValue)-1] #determines input value (subtracted from
1 due to index starting from 0)
```

If the field choice is 'Ethnicity Number', then the ethnicities file is read, and the number for the corresponding ethnicity value is determined and stored in the 'inputValue2' variable.

```
  elif fieldChoice == "Allergies": #performs selection if fieldChoice is allergies
    inputValue = inputValue.title() #sets format of input
    inputValue2 = inputValue.replace(" ,","|").replace(", ","|").replace("  ","")
#converts input into format of the value which would be stored in the patients file
```

If the field choice is 'Allergies' then the user input is converted into the format in which allergies inputted into the database would be stored in.

```
  elif fieldChoice == "Date of Birth": #performs selection if fieldChoice is date of
birth
    try:
      datetime.strptime(inputValue, '%d/%m/%Y') #determines if the date of birth
given is in the format '%d/%m/%Y'
    except ValueError or UnboundLocalError: #if the input is not in the correct format
      Label(screen, text="
").grid(row = 3, column =4) #overwrites previous messages
      Label(screen, text="Invalid Entry", fg="red").grid(row = 3, column =4)
#displays error message
      return False #ends function
```

If the field choice is 'Date of Birth' then its value is checked against the 'datetime' function, to ensure that the date is in the correct format. Otherwise, an error message is returned.

```
if fieldChoice == "Forename" or fieldChoice == "Surname" or fieldChoice == "Town/City": #performs selection dependent on fieldChoice
  inputValue = inputValue.title() #sets format of input
elif fieldChoice == "Gender" or fieldChoice == "Postcode": #performs selection dependent on fieldChoice
  inputValue = inputValue.upper() #sets format of input
```

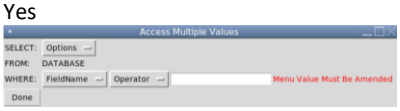
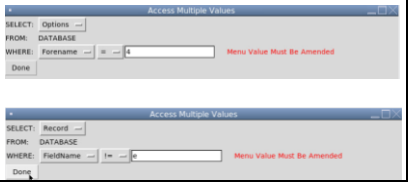
Depending on the field choice, the user input is then converted into the format in which the data would be stored in the database. For example, if 'f' is entered in the entry box for the gender field choice, it is automatically converted into 'F' because this is the format for gender values in the database.

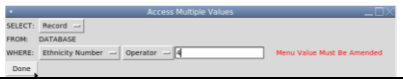
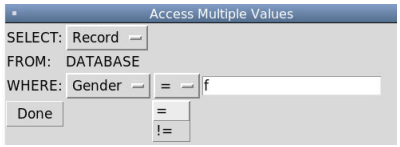
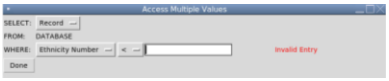
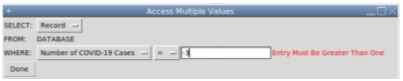
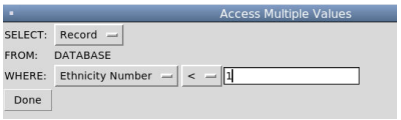
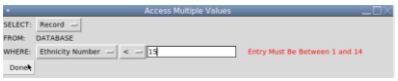
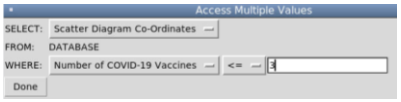
```
Label(screen, text="                                                    ").grid(row = 3, column =4) #overwrites previous messages
if selectChoice == "Scatter Diagram Co-Ordinates":
  plot(1) #points to subroutine if user wants to see plot values
```

A blank line is used to overwrite any previous error messages, so the user does not see error messages when they are not applicable.

TESTING

After developing the code, I had to ensure that the validation worked as intended.

Test	Reason	Test Data	Expected Outcome	Does the expected outcome occur?	Type of Test
Changing none of the values	An error should be generated as the program is unable to determine option, field name, operator, or input entry values	Clicking 'Done'	The program should output an error message.	Yes 	Erroneous as there is no user input.
Not changing one of the menu options	An error should be generated as the program is unable to determine menu option value	Clicking 'Done'	The program should output an error message.	Yes 	Erroneous

					
Choosing surname field value	The only comparison variable that can work with string is '=' or '!=', so no other options should be presented.	Clicking 'Surname' field value	The operator value options should automatically change to '=' and '!='. There should be no other options	Yes 	Normal
Entering decimal number for 'Number of COVID-19 Vaccines'	A person cannot have 'half' a vaccine, so decimal numbers should not be accepted.	4.3	Error message	Yes 	Erroneous
Entering nothing into the entry box	Without an input value, it is difficult to make comparisons.	Clicking 'Done'	Error message	Yes 	Erroneous
Entering number less than 0 for 'Number of COVID-19 Cases'	It is not possible to have had -2 vaccines so negative numbers should not be accepted.	-3	Error message	Yes 	Erroneous
Entering lowest ethnicity number	The lower bound of acceptable ethnicity numbers should be accepted	1	Removal of previous error messages	Yes 	Boundary
Entering number above highest ethnicity option	The number wouldn't correspond to any ethnicity so shouldn't be accepted.	15	Error message	Yes 	Erroneous
Amending all menu values and entering text into the entry box	Values should be interpreted as valid	Option: 'Scatter Diagram Co-Ordinates' Field Name: 'Number of COVID-19 Vaccines' Operator: '<=' Entry Value: '3'	Removal of previous error messages	Yes 	Normal

After testing had occurred, the scatter-diagram co-ordinates screen was designed.

SCATTER DIAGRAM CO-ORDINATES SCREEN

CODE

Because this part of the program would output the scatter diagram, the same 'plot' subroutine was used.

Firstly, the plot subroutine was changed so it would require a parameter:

```
def plot(value): #creates scatter diagram
```

This was important as it ensured that the functionality of the same subroutine could be differentiated. The same subroutine should be able to output the co-ordinates for the entire database, as well as the co-ordinates for field values specified by the user.

For the 'Access Multiple Values' option, the argument was set to one:

```
if selectChoice == "Scatter Diagram Co-Ordinates":
    plot(1) #points to subroutine if user wants to see plot values
```

As a result, I also had to change the button command for the 'Main Menu' screen, where 0 was the argument for outputting the entire scatter diagram:

```
    if accessLevel != "Nurse": #does not show this button if the access level is 'Nurse'
        Button(mainMenu,text="Scatter Diagram", height="2", width="30", command=lambda:
plot(0)).pack() #creates button for function
```

```
if value == 1: #performs if statment if specific values are being plotted
  fieldChoice1 = fieldChoice #stores field choice
  operatorChoice1 = operatorChoice #stores operator choice
  inputValue1 = inputValue #stores input value
  if fieldChoice1 == "Forename":
    comparisonValue = 1 #sets element to be checked
  elif fieldChoice1 == "Surname":
    comparisonValue = 2 #sets element to be checked
  elif fieldChoice1 == "Date of Birth":
    comparisonValue = 3 #sets element to be checked
    inputValue1 = time.strptime(inputValue, "%d/%m/%Y")
  elif fieldChoice1 == "Gender":
    comparisonValue = 4 #sets element to be checked
  elif fieldChoice1 == "Ethnicity Number":
    comparisonValue = 5 #sets element to be checked
    inputValue1 = inputValue2.strip() #stores input value
  elif fieldChoice1 == "Allergies":
    comparisonValue = 6 #sets element to be checked
    inputValue1 = inputValue2 #stores input value
  elif fieldChoice1 == "Number of COVID-19 Vaccines":
    comparisonValue = 7 #sets element to be checked
  elif fieldChoice1 == "Number of COVID-19 Cases":
    comparisonValue = 8 #sets element to be checked
  elif fieldChoice1 == "Town/City":
    comparisonValue = 11 #sets element to be checked
  elif fieldChoice1 == "Postcode":
    comparisonValue = 12 #sets element to be checked
```

If the ‘Access Multiple Values’ option is selected, then the field choice, operator choice, and input values are stored in variables. The variable ‘comparisonValue’ represents the index of each record that will be checked against, and is based on the field choice.

For the ‘Date of Birth’ field choice, the ‘strptime’ function is used to convert the input into a value which can be compared against the date values in each record.

If the field choice is ‘Ethnicity Number’ or ‘Allergies’, the input value is based on the validated variable, which was declared as global in the ‘verifyEntry’ subroutine:

```
global inputValue2
```

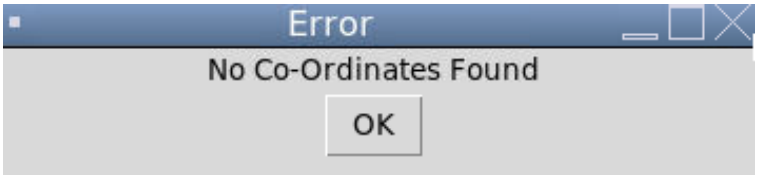
The ‘strip’ function (in green) is used for the ethnicity number input because the ethnicity value is likely to have a new line after it, which should be removed, otherwise the value being compared against will not be accurate.

```
for element in record: #reads each element in the record
  column+=1
  if column == 1:
    continue #ignore header row
  if value == 1:
    if fieldChoice1 == "Date of Birth":
      element[comparisonValue] = time.strptime(element[comparisonValue], "%d/%m/%Y") #makes date variable comparable
    if operatorChoice1 == "=":
      if element[comparisonValue] == inputValue1:
        noOfVaccines.append(element[7]) #adds the element number of vaccines to the array
        noOfCovidCases.append(element[8]) #adds the element number of covid cases to the array
    elif operatorChoice1 == "!=":
      if element[comparisonValue] != inputValue1:
        noOfVaccines.append(element[7]) #adds the element number of vaccines to the array
        noOfCovidCases.append(element[8]) #adds the element number of covid cases to the array
    elif operatorChoice1 == "<":
      if element[comparisonValue] < inputValue1:
        noOfVaccines.append(element[7]) #adds the element number of vaccines to the array
        noOfCovidCases.append(element[8]) #adds the element number of covid cases to the array
    elif operatorChoice1 == "<=":
      if element[comparisonValue] <= inputValue1:
        noOfVaccines.append(element[7]) #adds the element number of vaccines to the array
        noOfCovidCases.append(element[8]) #adds the element number of covid cases to the array
    elif operatorChoice1 == ">":
      if element[comparisonValue] > inputValue1:
        noOfVaccines.append(element[7]) #adds the element number of vaccines to the array
        noOfCovidCases.append(element[8]) #adds the element number of covid cases to the array
    elif operatorChoice1 == ">=":
      if element[comparisonValue] >= inputValue1:
        noOfVaccines.append(element[7]) #adds the element number of vaccines to the array
        noOfCovidCases.append(element[8]) #adds the element number of covid cases to the array
  else:
    noOfVaccines.append(element[7]) #adds the element number of vaccines to the array
    noOfCovidCases.append(element[8]) #adds the element number of covid cases to the array
```

The second part of the subroutine to be amended was the iteration statements. Once again, if the option selected is ‘Access Multiple Values’ - which would be determined by the parameter in the subroutine being 1 - then the selection statement (‘if value == 1’) is run. Firstly, if the field choice is ‘Date of Birth’, the date of birth value in each record is converted into a comparable value. Because I was unable to use the operator variable in replacement of an operator in an if statement (i.e. ‘if element[comparisonValue] operatorChoice inputValue1’), I used a nested if statement which would determine the operator to be used. Within these if statements was the element, determined by the field choice, of each record in the database being compared against the input value. For instance, if the field choice was ‘Forename’, the first index of each record will be compared against the forename input value, where the comparator is based on the operator chosen. If the condition was true then the number of vaccines and cases within the record were taken as coordinate values.

The final addition to the ‘plot()’ subroutine was to display an error message if no co-ordinates are found. The code and screen can be seen below:

```
if not noOfVaccines: #determines if list is empty
  userNotFound(0) #points to subroutine
  return #exits subroutine
```



The existing 'userNotFound' subroutine was reused and amended (in green) to allow the title and message to be changed:

Old code:

```
# Designing popup for user not found
def userNotFound(): #displays message if login details are not valid
    global userNotFoundScreen
    userNotFoundScreen = Toplevel(loginScreen) #uses the same screen defined in the 'loginScreen' subroutine
    userNotFoundScreen.title("Invalid Entry") #sets title
    userNotFoundScreen.geometry("150x100") #sets dimensions
    Label(userNotFoundScreen, text="Incorrect Username or Password").pack() #displays error message
    Button(userNotFoundScreen, text="OK", command=deleteUserNotFoundScreen).pack() #deletes the popup once the button is pressed
```

New code:

```
# Designing popup for value not found
def userNotFound(value):
  global userNotFoundScreen
  userNotFoundScreen = Toplevel(loginScreen) #uses the same screen defined in the 'loginScreen' subroutine
  if value == 1: #displays message if login details are not valid
    title = "Invalid Entry" #sets title
    message = "Incorrect Username or Password" #sets output text
  else: #displays message if co-ordinates haven't been found
    title = "Error" #sets title
    message = "No Co-Ordinates Found" #sets output text
  userNotFoundScreen.title(title) #sets title
  userNotFoundScreen.geometry("150x100") #sets dimensions
  Label(userNotFoundScreen, text=message).pack() #displays error message
  Button(userNotFoundScreen, text="OK", command=deleteUserNotFoundScreen).pack() #deletes the popup once the button is pressed
```

A parameter ('value') is passed in order to determine which title and message to output now that the subroutine is being used to output different messages. This resulted in a change (in red) to the 'loginVerify' subroutine so the argument could be added:

Old code:

```
if found == False:
    userNotFound() #displays error message
```

New code:

```
if found == False:
    userNotFound(1) #displays error message
```

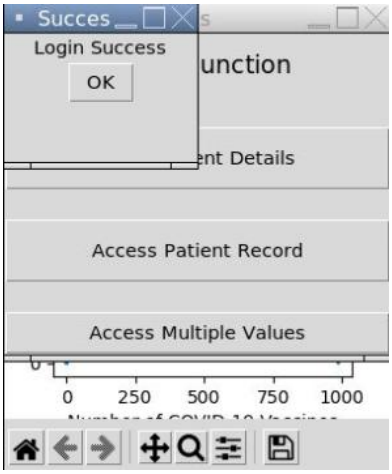
The rest of the ‘plot()’ code remained the same as it did not require changing.

ERORRS

When updating the button for the ‘Scatter Diagram’ option, some errors surfaced.

```
  if accessLevel != "Nurse": #does not show this button if the access level is 'Nurse'
    Button(mainMenu,text="Scatter Diagram", height="2", width="30",
command=plot(0)).pack() #creates button for function
```

Initially, the command function pointed to the subroutine ‘plot’ with the arguments ‘0’. However, when running the code, the scatter diagram would appear as soon as the ‘Login’ button was pressed:

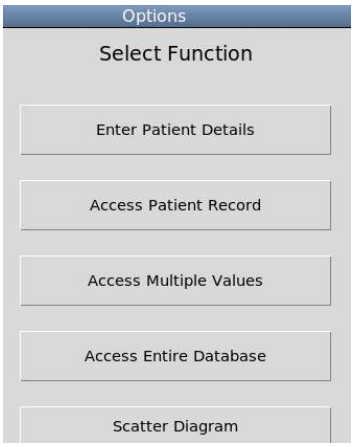


After closing the windows so that I was left with the main screen, I noticed that the ‘Scatter Diagram’ option did not appear:

I realised that by using the brackets, I was calling the subroutine. As a result, the ‘lambda’ function was used to pass the value ‘0’ into the subroutine:

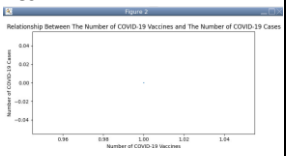
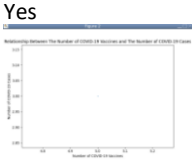
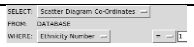
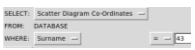
```
if accessLevel != "Nurse": #does not show this button if the access level is 'Nurse'
  Button(mainMenu,text="Scatter Diagram", height="2", width="30", command=lambda: plot(0)).pack() #creates button for function
```

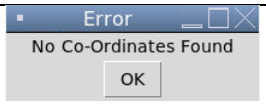
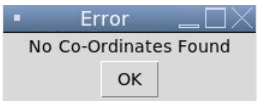
After adding this line, the 'Scatter Diagram' option appeared:



TESTING

After the code had been completed and errors had been solved, the code was tested to ensure that it produced the correct outputs.

Test	Reason	Test Data	Expected Outcome	Does the expected outcome occur?	Type of Test
Acquiring co-ordinates depending on date of birth.	By using the 'time.strptime' function, I need to ensure that the dates are comparable and produce the correct output.		One scatter diagram co-ordinate	Yes 	Normal
Entering allergies values	The input value should have been converted into the same format stored in the database	 Entry box: "apples, chocolates, strawberries"	One scatter diagram co-ordinate	Yes 	Normal
Entering ethnicity number value	The input value should be used as the index of the ethnicity value, where this value will be compared against the ethnicity in each record.		Several scatter-diagram co-ordinates	Yes 	Normal
Entering value which doesn't exist	The program should still check for the		Error message	Yes	Erroneous

	co-ordinates, but report an error message if such so-ordinate values do not exist.				
Entering valid postcode	The postcode input should be converted into uppercase and have its co-ordinated displayed.		tiy6	No 	Normal

Following the failed postcode test, I added an additional line to the code. This was because I realised that because the postcode is the last field value, it is likely to have a '\n' statement at the end of it, which means that the input value is not being compared against the true postcode value.

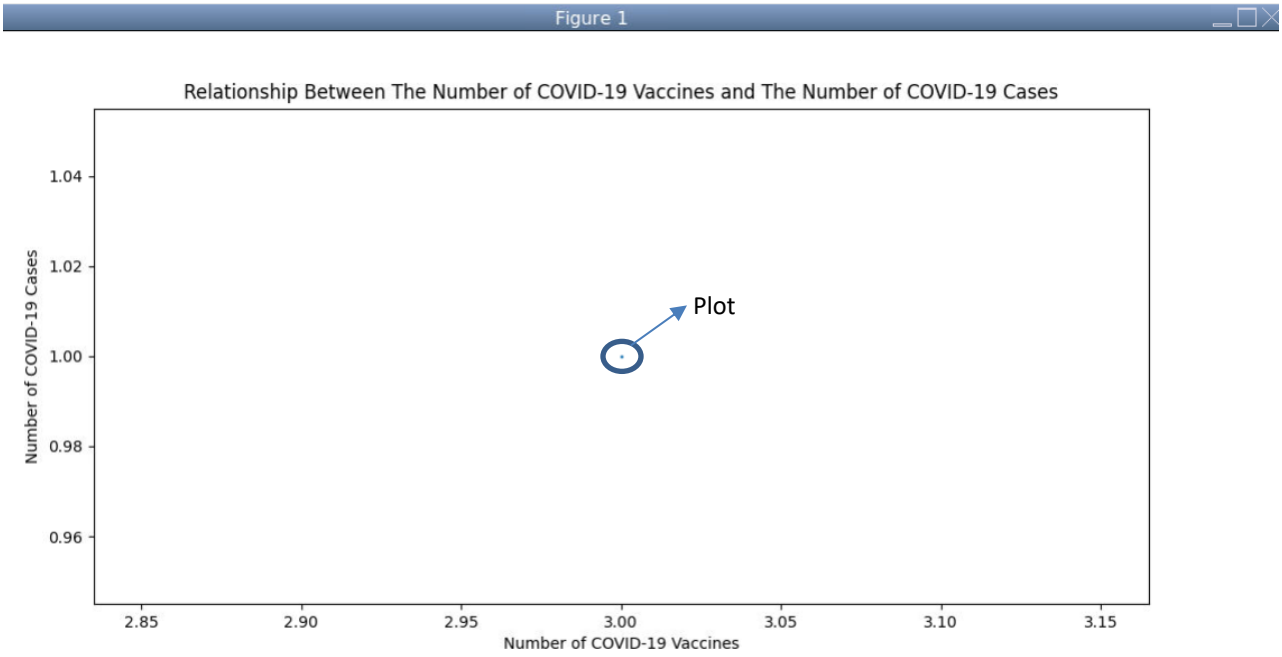
Old code:

```
if value == 1:
  if fieldChoice1 == "Date of Birth":
    element[comparisonValue] = time.strp
```

New code: (amendments in green)

```
if value == 1:
  element[comparisonValue] = element[comparisonValue].strip() #removes new line statement
  if fieldChoice1 == "Date of Birth":
    element[comparisonValue] = time.strptime(element[comparisonValue], "%d/%m/%Y") #makes
```

After this addition, the correct output was produced:



After testing had been complete, the final screens matched the initial screen design, and the section had been approved by Dr. Hussain, the code for the multiple records screen was developed.

MULTIPLE RECORDS SCREEN

CODE

Because this part of the program would output records, the same subroutine ('outputRecord') which outputted records (from the option 'Access Patient Record') was used.

Firstly, the 'outputRecord' subroutine was changed so it would require parameters:

```
def outputRecord(c,a,b): #displays record
```

This was important as it ensured that the functionality of the same subroutine could be differentiated. The same subroutine should be able to output one patient record, as well as multiple patient records.

The part of the 'verifyEntry' subroutine which pointed to this can be seen below:

```
if selectChoice == "Scatter Diagram Co-Ordinates":
  plot(1) #points to subroutine if user wants to see plot values
else:
  outputRecord(fieldChoice, operatorChoice, inputEntry)
```

The values 'fieldChoice', 'operatorChoice' and 'inputEntry' were assigned as arguments for later use.

This also meant that the 'nhsIdEntry' subroutine which would point to the same 'outputRecord' subroutine had to have additional arguments:

```
outputRecord(lineNumber,2,b) #prints record
```

The 2 would have use later on in the subroutine, but the 'b' did not mean anything – it was only added as the subroutine required 3 arguments (due to the three menu option values).

After adding the b, an error occurred:

```
NameError: name 'b' is not defined
```

```
outputRecord(lineNumber,2,1) #prints record
```

Because b was not a variable referenced within the 'outputRecord' subroutine, I changed it to 1 – which also had no meaning but solved the error.

The parameters in the 'outputRecord' subroutine were given the names 'c', 'a' and 'b'. Random letters were assigned because it was difficult to find parameter names which suited both the 'verifyEntry' and 'nhsIdEntry' subroutines.

```
def outputRecord(c,a,b): #displays record
  if a != 2: #performs selection for outputting multiple records
    recordsToBeDisplayed = [] #empty list for records to be displayed
    fieldChoice1 = fieldChoice #stores field choice
    operatorChoice1 = operatorChoice #stores operator choice
    inputValue1 = inputValue #stores input value
    if fieldChoice1 == "Forename":
      comparisonValue = 1 #sets element to be checked
    elif fieldChoice1 == "Surname":
      comparisonValue = 2 #sets element to be checked
    elif fieldChoice1 == "Date of Birth":
      comparisonValue = 3 #sets element to be checked
      inputValue1 = time.strptime(inputValue, "%d/%m/%Y")
    elif fieldChoice1 == "Gender":
      comparisonValue = 4 #sets element to be checked
    elif fieldChoice1 == "Ethnicity Number":
      comparisonValue = 5 #sets element to be checked
      inputValue1 = inputValue2.strip() #stores input value
    elif fieldChoice1 == "Allergies":
      comparisonValue = 6 #sets element to be checked
      inputValue1 = inputValue2 #stores input value
    elif fieldChoice1 == "Number of COVID-19 Vaccines":
      comparisonValue = 7 #sets element to be checked
    elif fieldChoice1 == "Number of COVID-19 Cases":
      comparisonValue = 8 #sets element to be checked
    elif fieldChoice1 == "Town/City":
      comparisonValue = 11 #sets element to be checked
    elif fieldChoice1 == "Postcode":
      comparisonValue = 12 #sets element to be checked
```

The first change to the 'outputRecord' subroutine was the addition of a selection statement. If the subroutine was being called to output multiple records, an empty list ('recordsToBeDisplayed') was created to store the record values. Additionally, the field choice, operator choice, and input values were assigned to variables. Finally, depending on the field choice, the comparison value was set. This was a feature copied from the 'plot' subroutine, and was important in ensuring that the right element of each record is checked against.

```
with open("Patients.csv") as file:
  lines = file.readlines() #reads each line of the file
  header = lines[0].strip("\n") #uses first line as header
  if a == 2: #performs selection for outputting multiple records
    record = lines[c].strip("\n") #finds record containing NHS ID
  else:
    recordsToBeDisplayed.append(lines[0].strip("\n").split(delimiter)) #adds fieldname line to list
    for line in lines[1:]: #reads each line of the database
      line = line.strip("\n")
      line = line.split(delimiter)
      if fieldChoice1 == "Date of Birth":
        line[comparisonValue] = time.strptime(line[comparisonValue], "%d/%m/%Y") #makes date variable comparable
      if operatorChoice1 == "=": #determines operator which compares data
        if line[comparisonValue] == inputValue1: #compares values
          recordsToBeDisplayed.append(line) #adds record to records to be displayed
      elif operatorChoice1 == "!=": #determines operator which compares data
        if line[comparisonValue] != inputValue1: #compares values
          recordsToBeDisplayed.append(line) #adds record to records to be displayed
      elif operatorChoice1 == "<": #determines operator which compares data
        if line[comparisonValue] < inputValue1: #compares values
          recordsToBeDisplayed.append(line) #adds record to records to be displayed
      elif operatorChoice1 == "<=": #determines operator which compares data
        if line[comparisonValue] <= inputValue1: #compares values
          recordsToBeDisplayed.append(line) #adds record to records to be displayed
      elif operatorChoice1 == ">": #determines operator which compares data
        if line[comparisonValue] > inputValue1: #compares values
          recordsToBeDisplayed.append(line) #adds record to records to be displayed
      elif operatorChoice1 == ">=": #determines operator which compares data
        if line[comparisonValue] >= inputValue1: #compares values
```

The patient file is then read, where the record appended to the 'recordsToBeDisplayed' list is dependent on whether the conditions set by the user is true for each record.

The initial for loop began like this:

```
for line in lines
```

This produced the error below when the fieldChoice was 'date of birth':

```
  File "main.py", line 326, in outputRecord
    line[comparisonValue] = time.strptime(line[comparisonValue], "%d/%m/%Y") #makes d
ate variable comparable
  File "/usr/lib/python3.8/_strptime.py", line 562, in _strptime_time
    tt = _strptime(data_string, format)[0]
  File "/usr/lib/python3.8/_strptime.py", line 349, in _strptime
    raise ValueError("time data %r does not match format %r" %
ValueError: time data ' Date of Birth' does not match format '%d/%m/%Y'
```

Because the first line of the 'Patients' file was being iterated through, the date of birth index was checked. However, because the first line is the title, it did not contain a date of birth value. As a result, the loop was changed to start from the first indexed item in the array, and the error was removed:

```
for line in lines[1:]
```

Despite the fact that the first line was 'skipped', I still wanted the first line to be added to the list 'recordsToBeDisplayed' because each record shown required a field name to identify what each element means. As a result, the following line was added before the loop:

```
recordsToBeDisplayed.append(lines[0].strip("\n").split(delimiter)) #adds fieldname line to list
```

```
header = header.split(delimiter) #separates line upon each instance of a delimiter
if a == 2: #performs selection for outputting multiple records
  record = record.split(delimiter)  #separates line upon each instance of a delimiter
  columns = 2 #sets record columns
else:
  columns = len(recordsToBeDisplayed)  #determines number of columns
  if columns == 1:
    userNotFound(2) #produces error if no records were added
    return #exits subroutine
screen = Tk() #creates a new screen
screen.geometry("300x250") #sets screen dimensions
if a == 2: #performs selection for outputting multiple records
  screen.title("{0}, {1}".format(record[2],record[1])) #sets the title of the screen to the users surname and forename
  outputValues = [(header),(record)] #sets output values
else:
  screen.title("Records") #sets the title of the screen to 'Records'
```

The use of the 'a' parameter is used to determine which function the subroutine is performing – outputting one, or multiple records.

The 'columns' variable is set to 2 if one record is being outputted because only the header and record values will be printed. If multiple records and being printed, then the length of the 'recordsToBeDisplayed' list becomes the number of columns to be outputted, as that determines how many records there will be. The purpose of this will become clearer later on.

If the column number is 1, then an error screen will be printed as it suggests that the only value stored is the header record, which means that no other records were found.

Finally, depending on the function the subroutine is performing, the titles will differ.

In relation to the error screen message, the 'userNotFound' subroutine was amended:

Old code:

```
# Designing popup for value not found
def userNotFound(value):
  global userNotFoundScreen
  userNotFoundScreen = Toplevel(loginScreen) #uses the same screen defined in the 'loginScreen' subroutine
  if value == 1: #displays message if login details are not valid
    title = "Invalid Entry" #sets title
    message = "Incorrect Username or Password" #sets output text
  else: #displays message if co-ordinates haven't been found
    title = "Error" #sets title
    message = "No Co-Ordinates Found" #sets output text
  userNotFoundScreen.title(title) #sets title
  userNotFoundScreen.geometry("150x100") #sets dimensions
  Label(userNotFoundScreen, text=message).pack() #displays error message
  Button(userNotFoundScreen, text="OK", command=deleteUserNotFoundScreen).pack() #deletes the popup once the button is pressed
```

New code:

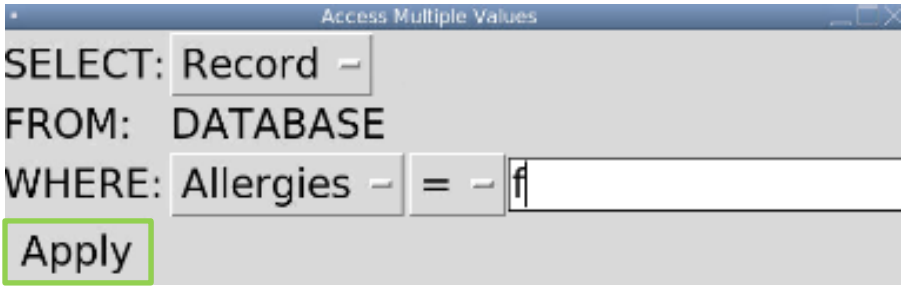
```
# Designing popup for value not found
def userNotFound(value):
  global userNotFoundScreen
  userNotFoundScreen = Toplevel(loginScreen) #uses the same screen defined in the 'loginScreen' subroutine
  if value == 1: #displays message if login details are not valid
    title = "Invalid Entry" #sets title
    message = "Incorrect Username or Password" #sets output text
  else: #displays message if co-ordinates haven't been found
    title = "Error" #sets title
    if value == 0: #displays message if co-ordinates haven't been found
      message = "No Co-Ordinates Found" #sets output text
    else: #displays message if records haven't been found
      message = "No Records Found" #sets output text
  userNotFoundScreen.title(title) #sets title
  userNotFoundScreen.geometry("150x100") #sets dimensions
  Label(userNotFoundScreen, text=message).pack() #displays error message
  Button(userNotFoundScreen, text="OK", command=deleteUserNotFoundScreen).pack() #deletes the popup once the button is pressed
```

The selection statement was extended to account for the new call to the subroutine. For instance, if ‘value’ is not 0, then the error message will differ. By keeping the title of the screen as ‘Error’ regardless of whether the error screen is loading for there being no record or scatter diagram values, coding time and memory was saved.

The error screen, matching the one is the design section, can be seen below:

At this point, a small amendment was made to the ‘Done’ button, where it was changed to ‘Apply’. This was because Dr. Hussain thought that ‘Apply’ was more specific to function being performed. The amended code and output can be seen below:

```
Button(screen,text="Apply",command=verifyEntry).grid(row=4,column=0,sticky='w') #validates user input
```




```
for i in range(columns): #reads each record
  for j in range(13): #reads each field
    if i == 0:
      addingValues = Entry(screen, width=26, fg='black',font=('Calibri',9,'bold')) #sets display design
    else:
      addingValues = Entry(screen, width=26, fg='black',font=('Calibri',9)) #sets display design
    addingValues.grid(row=i, column=j) #positions each field + record
    if a == 2:
      addingValues.insert(END, outputValues[i][j]) #adds values to grid
    else:
      addingValues.insert(END, recordsToBeDisplayed[i][j]) #adds values to grid
```

Used for 'Access Patient Record' option

Used for accessing multiple records option

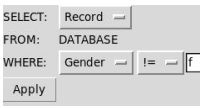
The last part of the code dealt with adding the values onto the screen. The columns value was used to determine how many records would be outputted. Initially, this value was 2, but after the subroutine had increased functionality, this value was dependent on the function being performed, so 'columns' was more appropriate.

The values inserted onto the grids were also dependent on the function being performed. It could either be the 'outputValues' list with a header and the specific record, or the 'recordsToBeDisplayed' list, which contains all the records to be displayed.

TESTING/ERRORS

After the code had been completed, it was tested to ensure that it produced the correct outputs.

Test Number	Test	Reason	Test Data	Expected Outcome	Does the expected outcome occur?	Type of Test
1	Finding the records for all females in the database	I need to ensure that the "=" comparator operator works effectively		10 records	Yes 	Normal

2	Finding the records for all males in the database	I need to ensure that the "!=" comparator operator works effectively		4 records	Yes 	Normal
3	Finding records for those born before, or on 25/08/2004	I need to ensure that the "<=" comparator operator works effectively		7 records	Yes 	normal

After conducting test number 3, I realised that the date of birth output was incorrect:

Date of Birth
2004 8 25 0 0 0 2 238 -1
2004 3 14 0 0 0 6 74 -1
1555 8 15 0 0 0 0 227 -1
2003 12 31 0 0 0 2 365 -1
2003 9 12 0 0 0 4 255 -1
1992 3 1 0 0 0 6 61 -1
2004 3 2 0 0 0 1 62 -1

The output displayed the converted value of the original date of birth value, which was intended to make the date of birth comparable. The code for this can be seen below:

```
line[comparisonValue] = time.strptime(line[comparisonValue], "%d/%m/%Y") #makes date variable comparable
```

Because this line was overwriting the date of birth field values, I realised that it would be better if the selection statements were comparing variables:

```
if fieldChoice1 == "Forename":
  comparisonValue = line[1] #sets element to be checked
elif fieldChoice1 == "Surname":
  comparisonValue = line[2] #sets element to be checked
elif fieldChoice1 == "Date of Birth":
  comparisonValue = line[3] #sets element to be checked
  inputValue1 = time.strptime(inputValue, "%d/%m/%Y")
elif fieldChoice1 == "Gender":
  comparisonValue = line[4] #sets element to be checked
elif fieldChoice1 == "Ethnicity Number":
  comparisonValue = line[5] #sets element to be checked
  inputValue1 = inputValue2.strip() #stores input value
elif fieldChoice1 == "Allergies":
  comparisonValue = line[6] #sets element to be checked
  inputValue1 = inputValue2 #stores input value
elif fieldChoice1 == "Number of COVID-19 Vaccines":
  comparisonValue = line[7] #sets element to be checked
elif fieldChoice1 == "Number of COVID-19 Cases":
  comparisonValue = line[8] #sets element to be checked
elif fieldChoice1 == "Town/City":
  comparisonValue = line[11] #sets element to be checked
elif fieldChoice1 == "Postcode":
  comparisonValue = line[12] #sets element to be checked
```

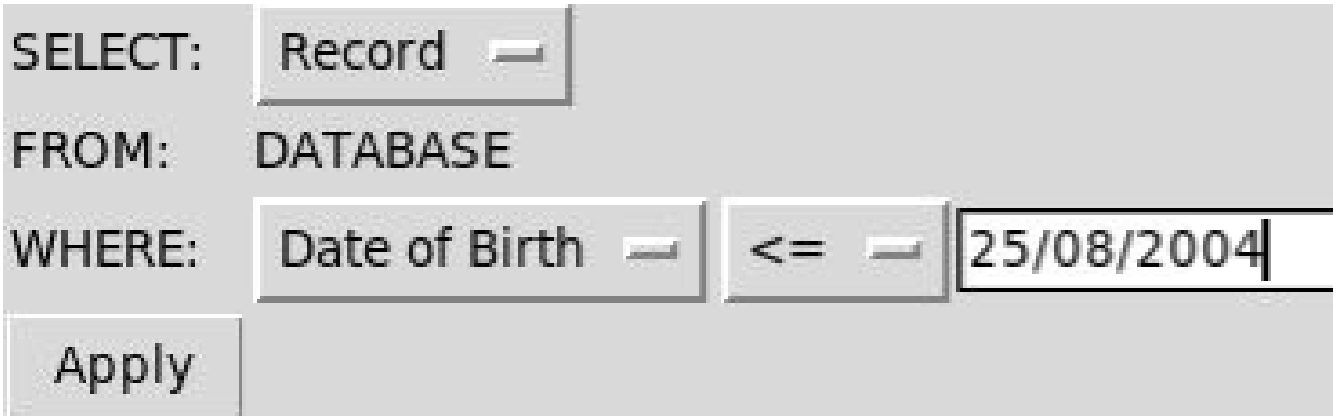
Rather than the comparison value being given a number, it stored the index of the fieldChoice for each record.

```
if fieldChoice1 == "Date of Birth":
  comparisonValue = time.strptime(comparisonValue, "%d/%m/%Y") #makes date variable comparable
```

As a result, the actual date of birth value for each record was no longer changed, as the original line was left unchanged, and only the variable was being compared against:

```
if operatorChoice1 == "=": #determines operator which compares data
  if comparisonValue == inputValue1: #compares values
    recordsToBeDisplayed.append(line) #adds record to records to be displayed
elif operatorChoice1 == "!=": #determines operator which compares data
  if comparisonValue != inputValue1: #compares values
    recordsToBeDisplayed.append(line) #adds record to records to be displayed
elif operatorChoice1 == "<": #determines operator which compares data
  if comparisonValue < inputValue1: #compares values
    recordsToBeDisplayed.append(line) #adds record to records to be displayed
elif operatorChoice1 == "<=": #determines operator which compares data
  if comparisonValue <= inputValue1: #compares values
    recordsToBeDisplayed.append(line) #adds record to records to be displayed
elif operatorChoice1 == ">": #determines operator which compares data
  if comparisonValue > inputValue1: #compares values
    recordsToBeDisplayed.append(line) #adds record to records to be displayed
elif operatorChoice1 == ">=": #determines operator which compares data
  if comparisonValue >= inputValue1: #compares values
    recordsToBeDisplayed.append(line) #adds record to records to be displayed
```

Following this change, the same test was performed:



And the correct output, with the unchanged date of birth values can be seen below:

NHS ID	Forename	Surname	Date of Birth
1762883873	Farida	Addo	25/08/2004
4813824874	Sholeye	Olajide	14/03/2004
6388236637	Timmy	Tom	15/08/1555
9554744818	Rafaela	Mendes-Da-Silva	31/12/2003
7529882792	Fajmaela	Aldi	12/09/2003
5564528166	Taylor	Blue	01/03/1992
8583335398	Sally	Lucy	02/03/2004

After developing the subroutine, I added an extra code within the selection statement:

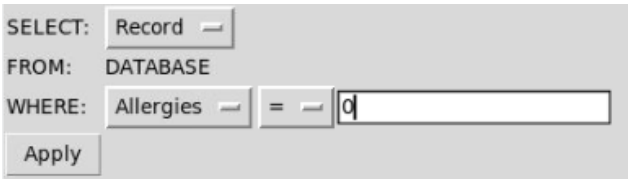
```
elif fieldChoice1 == "Allergies":
  comparisonValue = line[6] #sets element to be checked
  if inputValue1 == '0':
    inputValue1 = "N/A" #replaces input value to whats stored in database
  else:
    inputValue1 = inputValue2 #stores input value
```

This was added regarding feedback from Dr. Hussain who suggested that if a user is searching for patients with 0 allergies, for example, then they should be able to type in 0. With the original code, the user would have to provide the input value ‘N/A’ in order to see the records/scatter diagram co-ordinates for those with 0 allergies.

As a result, I changed the code so it would enable people to input 0. The input would be converted into ‘N/A’ by the system – without the user’s knowledge – so that the correct value stored in the database could be compared against. By adding this feature, the efficiency of the algorithm has been ensured as users don’t have or type ‘N/A’, which is longer than typing 0.

The successful testing of this addition can be seen below:

Input: (normal data)



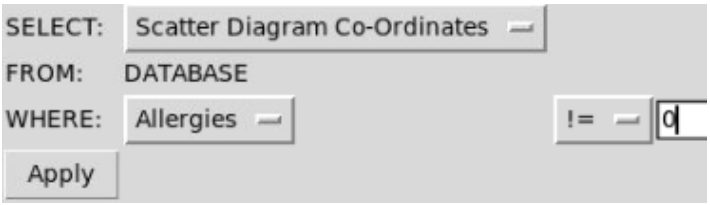
Output:

The code was also amended in the 'plot' subroutine:

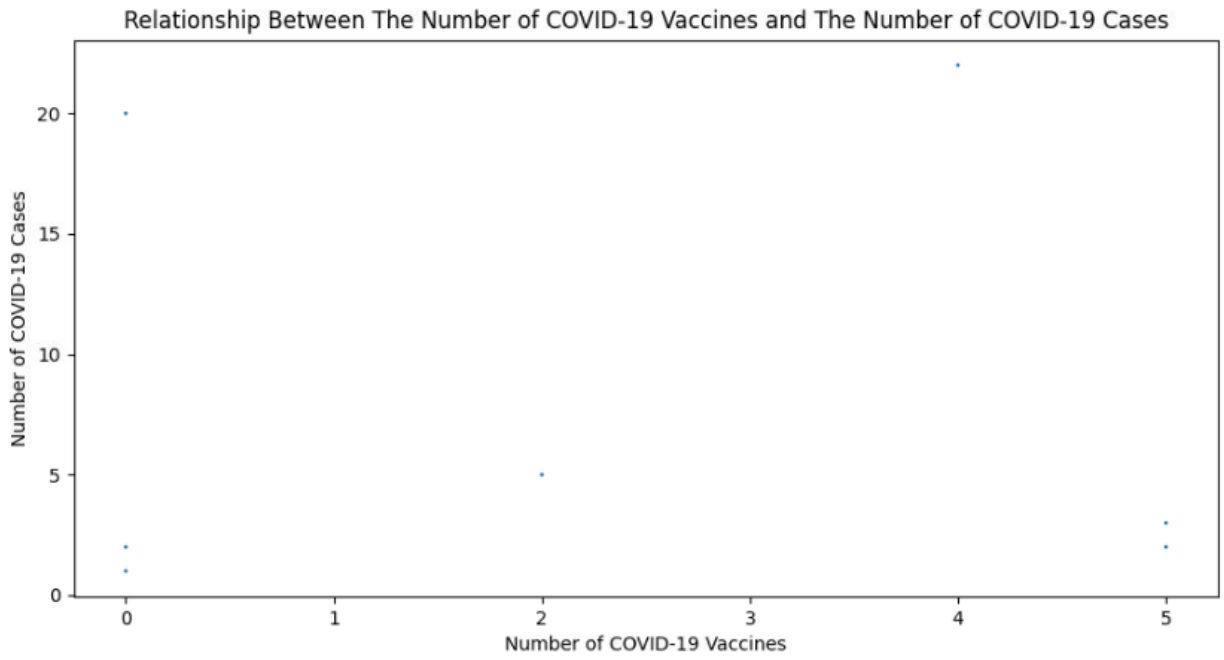
```
elif fieldChoice1 == "Allergies":
  comparisonValue = 6 #sets element to be checked
  if inputValue1 == '0':
    inputValue1 = "N/A" #replaces input value to whats stored in database
  else:
    inputValue1 = inputValue2 #stores input value
```

The testing can be seen below:

Input: (normal data)



Output:



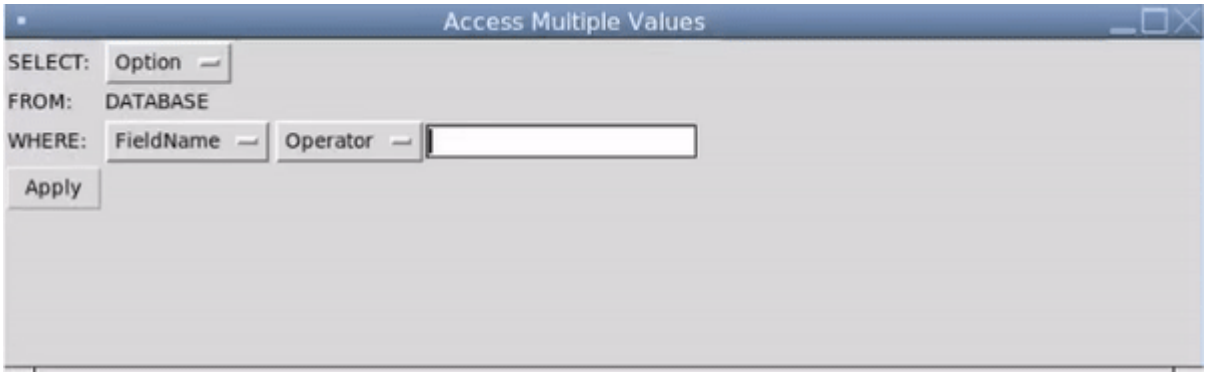
FINAL CODE LINES

The following code is referencing the 'multipleValues' subroutine.

```
fieldName.set("FieldName") #sets original menu text
operator.set("Operator") #sets original menu text
option.set("Option") #sets original menu text
inputEntry.delete(0, END) #clears entry box
```

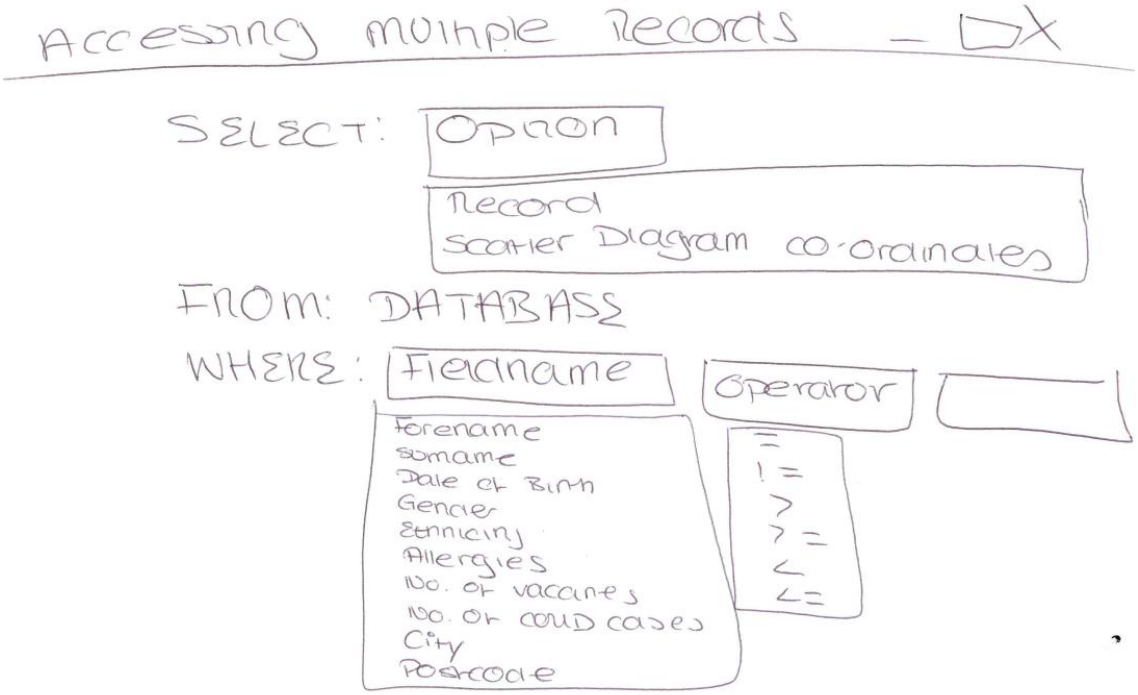
After either of the operations have been performed, the input value is cleared to save users time in having to remove previous data entries. Additionally, the option, field name and operator values are set to their original state. This was done so that the user is aware that they are providing details for a new function.

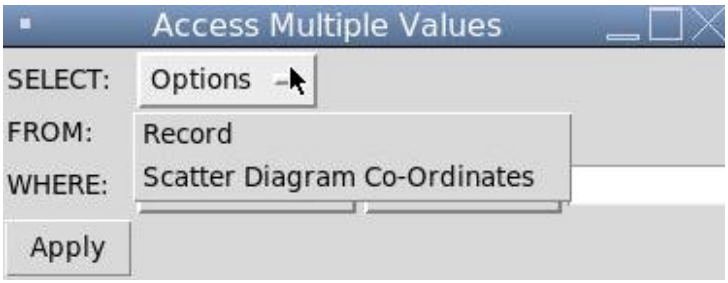
This can be seen working below:

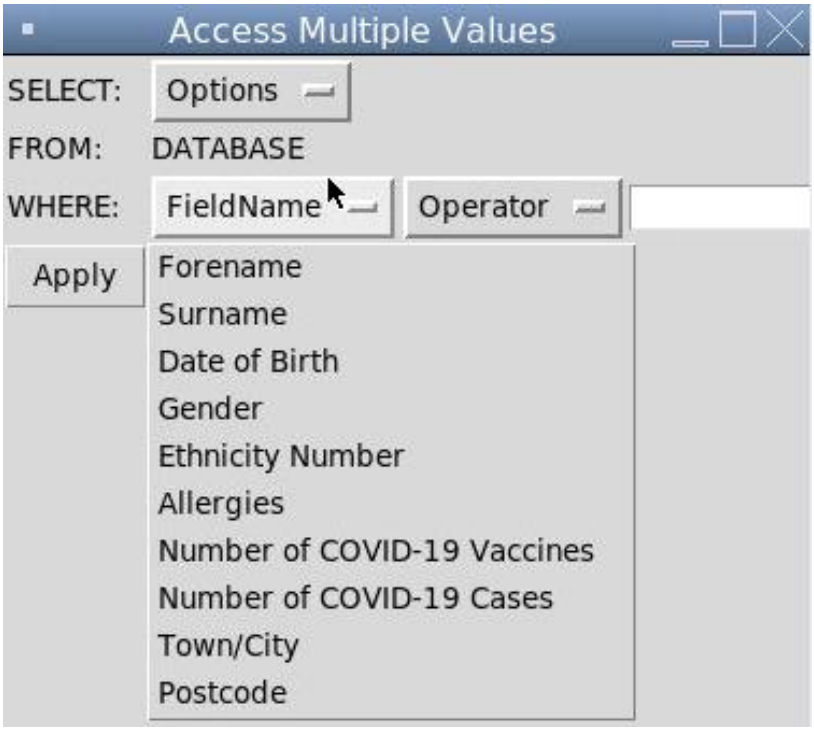


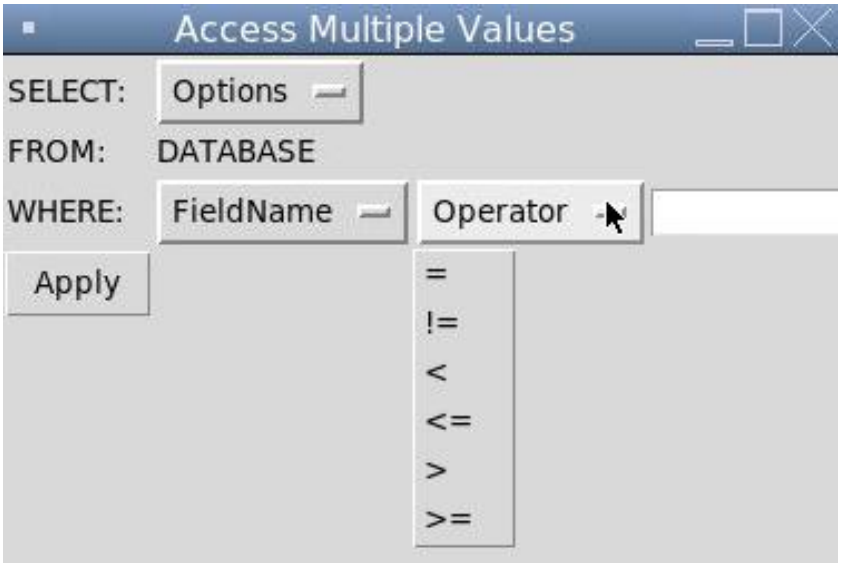
Overall, this part of the program was able to successfully meet Dr. Hussain's requirement of an SQL-like option.

The final design matched the prototype created in the design section. Comparisons can be seen below:



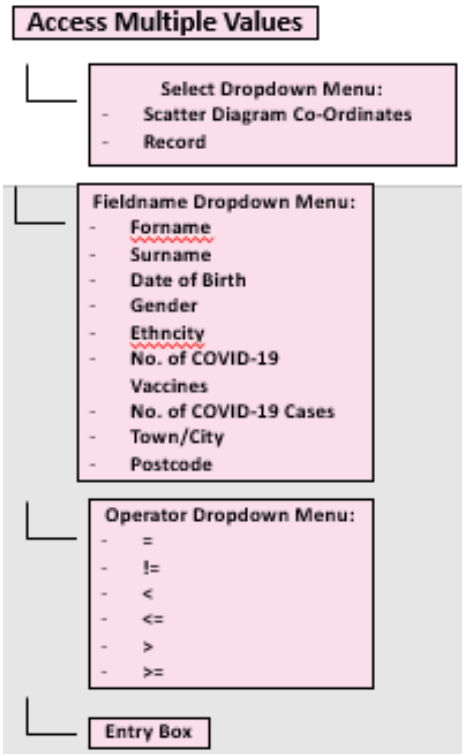






Testing was conducted to ensure that the buttons worked, and the tests were successful, with any errors being rectified.

After completing the section, I referenced the top-down diagram in order to ensure that the program was developed in line with it.



The 'Access Multiple Values' option was complete. I created a drop-down menu for users to select either a record or scatter-diagram co-ordinates, from the different fieldnames, where operator values were equal to a user input. The sub-branches had been successfully coded for.

After completing the code for the options, the last line of code needed was to call the main screen.

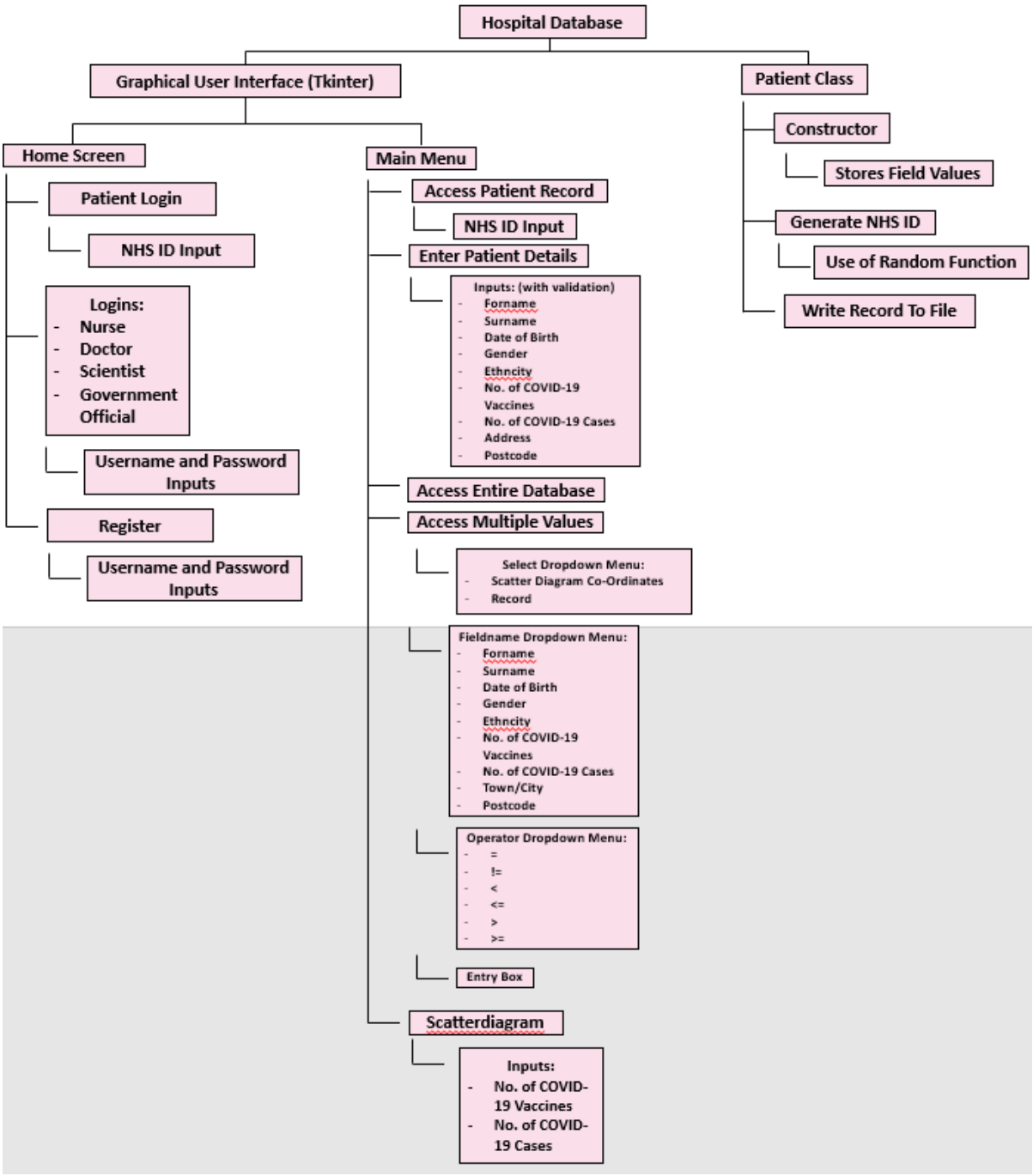
CALLING THE PROGRAM

```
mainScreen() #starts program
```

After all the validation, testing, error resolving, and writing to file had been completed and approved by Dr. Hussain, all that was left to do was to call the 'mainScreen' subroutine which would start the program.

REVIEW

A key part of creating the subroutine was the use of the top-down diagram, in order to guide the development process.



By breaking down the problem into subtasks, I was able to code each part independently, and then bring them together to create the final system. Ensuring that the final code met each branch was also important as the diagram was simplified into the main components which set the foundations for the program.

Additionally, the program was developed with the aim of meeting the success criteria, as well as meeting the questionnaire respondents from stakeholders who may have to utilise the system.

By the end of this iterative development process, I was able to produce a code which reflected the initial designs and system requirements.

After completing the program, I showed its entirety to Dr. Hussain. His response can be seen below:

To: You

Hi Farida,

Overall, I am very impressed with the outcome of this program. Although I was not initially used to 'Tkinter', it was very easy to adapt to. The system you created reflects one in which I encounter daily. Well done!

Kind regards,
Dr. Hussain

From this response, it can be assumed that the program matched his requirements, increasing my certainty that the development of this program will benefit other doctors in their daily operations.

EVALUATION

TESTING TO INFORM EVALUATION

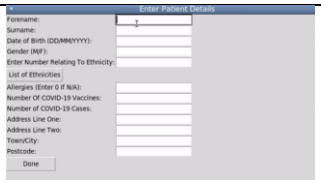
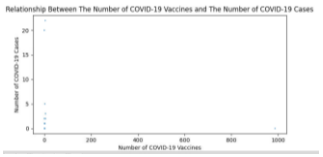
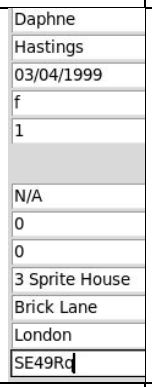
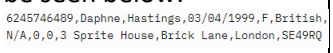
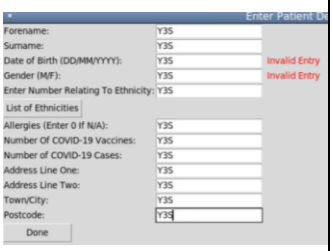
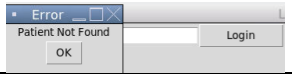
As well as testing the code myself, Dr. Hussain also tested the program. In order to ensure that the program meets a wide range of people's needs, I did not limit the beta-testing to just Dr. Hussain, I found some other stakeholders who were willing to test the program. These were students who were currently studying Medicine, as well as Biology and Chemistry A-Level students who were aspiring medics. This was important as it ensured that the program was tested by those who were likely to encounter such a database in their later life. It was also insightful as I was able to see how those who had never had an experience with the system interacted with it, and the errors which may have surfaced.

POST-DEVELOPMENT TESTING

SELF-TESTING AGAINST PRE-DEVELOPMENT TESTING TABLE

The following section will focus on the post-development testing of the program. This is to ensure that it performs all the necessary functions, and is robust. The pre-development testing table will be used as a guide for testing, in order to ensure that the program meets the initial requirements of the solution.

Test Number	Test	Test Data	Expected Outcome	Does the expected outcome occur?	Type of Test
1	Gathering inputs about a patient	User inputs for	Forename: 'sally' Surname: 'lucy'	Yes	Normal as the data entered

		entry boxes	Date of Birth: '02/03/2004' Gender: 'f' Ethnicity: '3' Allergies: '0' Number of vaccines: '987' Number of cases: '0' Address line one: 'flat 99' Address line two: 'castle lane' City: 'nottingham' Postcode: 'se789q'		is of the expected type
2	Create a diagram which represents the relationship between the number of COVID vaccines had, and the number of COVID cases had.	N/A – uses values from the database	Scatter diagram	Yes 	N/A
3	Write the patient data to file		Record written to 'Patients' file	Yes. A screenshot of the inputs written to file can be seen below: 	Normal as it is of the expected type
4	Inputting alphanumeric characters when entering patient details	Y3S	An 'invalid entry' message should appear next to the entry boxes for the user inputs: date of birth, gender, ethnicity, number of COVID-19 vaccines, and number of COVID-19 cases.	Yes, but only for some inputs 	Erroneous as the string is in the wrong format for some entries
5	Entering an invalid NHS ID	Robot	Error message	Yes 	Erroneous as the input is in

					the wrong format
6	Entering a valid NHS ID	3919196324	Patient record screen outputted	Yes, same as figure	Normal as it is of the expected type
7	Entering an unrealistic amount of COVID-19 vaccines	1000	Invalid entry message should be outputted	No 	Erroneous as the data is of an unrealistic limit
8	Entering a Date of Birth after the current date	10/03/3276	Invalid entry message	No 	Erroneous as the data exceeds the current year. Also boundary as 4 digits is the maximum year digit limit.

RESOLVING TESTING ERRORS

When testing the code according to the pre-testing data, a few errors surfaced which required changes to the code.

TEST NUMBER 4

When performing the alphanumeric input test, the 'invalid entry' message did not appear beside all of the entry bars it was supposed to, such as the ethnicity, number of COVID-19 vaccines, and number of COVID-19 cases entry boxes. Furthermore, I was presented with an error, as shown below:

```
  File "/usr/lib/python3.8/tkinter/__init__.py", line 1883, in __call__
    return self.func(*args)
  File "main.py", line 369, in detailsVerify
    validateEthnicity = ethnicityValidate(ethnicity.get()) #stores results
 of input validation
  File "main.py", line 475, in ethnicityValidate
    lastDigit = int(repr(float(ethnicity))[-1]) #determines the last digit
 of the float number
ValueError: could not convert string to float: 'Y3S'
```

The problem was that the ethnicity input had been accepted, and the error arose when the 'lastDigit' could not be determined. However, the input should never have been accepted in the first place. As a result, I made some amendments to the ethnicity validation code to check if the user input contained any alphabetical characters.

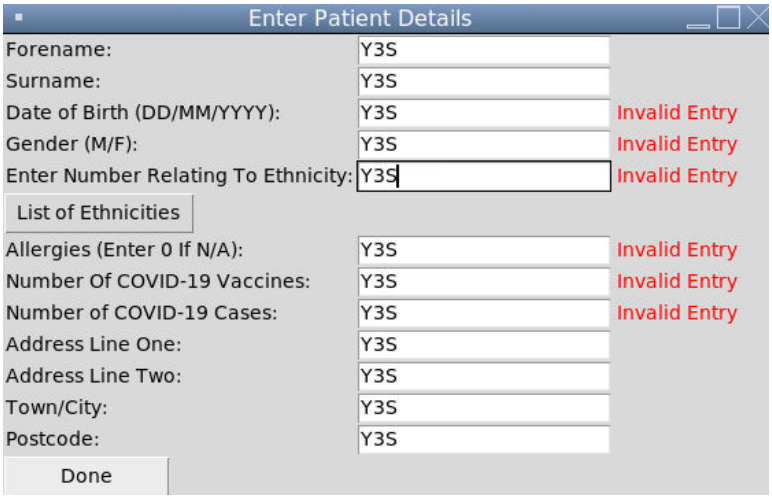
Initially, I tried to test the whole ethnicity input with the code below:

```
if ethnicity.isalpha() == True:
    Label(patientDetails, text="Invalid Entry", fg="red", font=("calibri", 10)).grid(row = 4, column = 3) #displays error message
    return False #indicates that the value is invalid
```

However, I realised that 'False' would be returned by the 'isalpha' method if the whole string did not contain alphabetical characters, which was not the intended output as it should only take one alphabetical character for 'True' to be returned. As a result, I implemented a for loop that went through each character and would return 'True' as soon as an alphabetical character was encountered. The changed code can be seen below in orange:

```
def ethnicityValidate(ethnicity): #validating ethnicity
    if ethnicity == "":
        Label(patientDetails, text="Invalid Entry", fg="red", font=("calibri", 10)).grid(row = 4, column = 3) #displays error message if
        return False
    for character in ethnicity: #goes through each character in the ethnciity input
        if character.isalpha() == True: #checks if character is an alphabetical character
            Label(patientDetails, text="Invalid Entry", fg="red", font=("calibri", 10)).grid(row = 4, column = 3) #displays error message
            return False #indicates that the value is invalid
```

After making this change and running the program with the same tests, the error messages automatically displayed beside the number of COVID-19 vaccines, and number of COVID-19 cases entry boxes, indicating that there was no error with the validation code for those sections, and that only the ethnicity validation needed to be amended. The successful test output can be seen below:



OMITTING SOME TESTS FROM THE PRE-DEVELOPMENT TESTING TABLE

When comparing the tests outlined in the pre-development test table to the final code, some tests were omitted.

INPUTTIING AN UNREALISTIC NUMBER OF COVID-19 VACCINES

This particular test was not conducted. This is because the only validation for inputting COVID-19 vaccines is ensuring that the input is a whole number greater than 1. Even though it would be practical to add a validation which sets a realistic limit on the number of COVID-19 vaccines someone has had, it

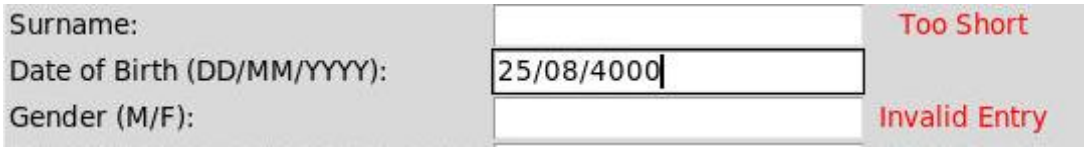
is also hard to quantify. It would be too subjective for me to say that 1000 vaccines is the limit, when in reality someone may have had 1000 COVId-19 vaccines – although, highly unlikely.

INPUTTING A VALID AGE

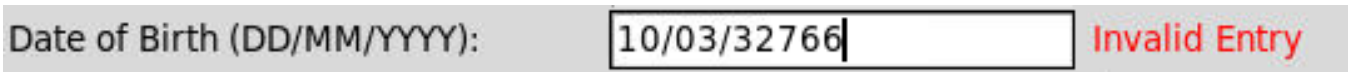
When creating the program initially, the age input was required. However, this changed during the development process, and was replaced with a date of birth input. Therefore, there was no need to test for this anymore.

ENTERING A DATE OF BIRTH AFTER THE CURRENT DATE

Initially, the date of birth validation was intended to prevent the year from exceeding the current year the data was being inputted in. However, the 'strptime' function does not account for this. It only ensures that the day and month values are in the correct format. As a result, testing this feature was omitted as entering a year, such as 4000 would still produce a valid input, which is technically not incorrect as that is what the validation was designed to allow. An example of the input being incorrectly accepted without generating an error message can be seen below:



However, when completing this test, I noticed that the digit limit for the year value using the 'strptime' function was 4. A digit limit of 5 would produce an error, as shown below:



While entering a date of birth after the current date wouldn't produce an error, it is still good to note that there is a 4-digit limit, and thus there is some validation for this input.

INPUTTING A FAKE POSTCODE WHEN ENTERING PATIENT DETAILS

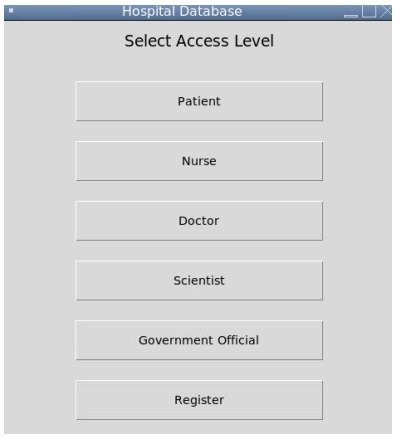
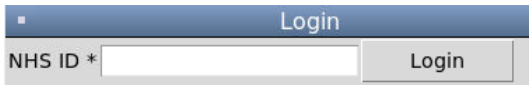
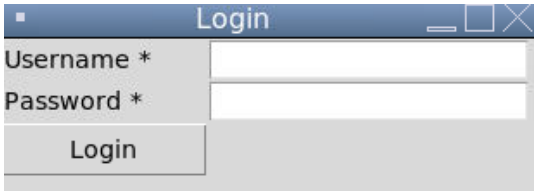
When thinking about validation for the postcode initially, the postcode would not allow a random sequence of characters i.e. ABC DEF to be entered. However, whilst developing the code the validation enabled any inputs to be valid as long as they began with an alphabetical character. As a result, the test was no longer needed as it would not provide the desired error. Furthermore, the code validation was not amended to allow a sequence of alphabetical characters to be invalid because in reality, there may be a real postcode which is in the form of ABC DEF, for example. It is important that the system accepts, and accounts for all types of user input. An example of the input being incorrectly accepted without generating an error message can be seen below:

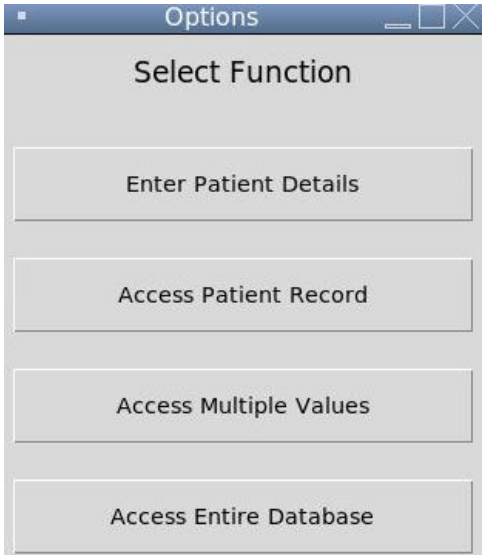
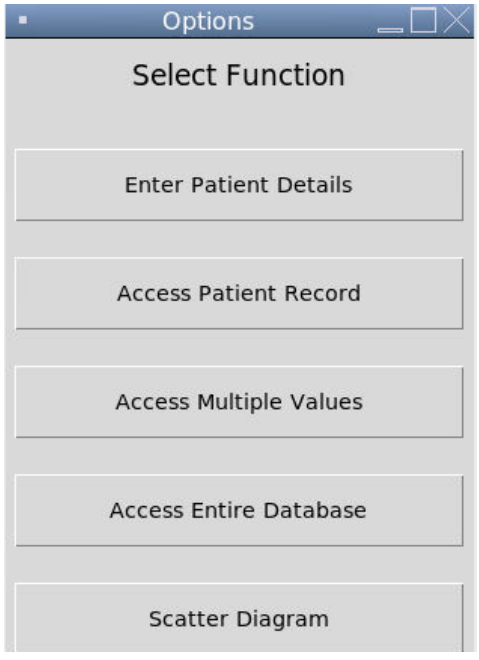
CHECKING TO SEE IF A RANDOMLY GENERATED NHS ID IS THE SAME AS A CURRENT ONE

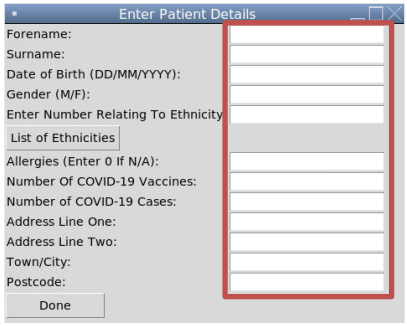
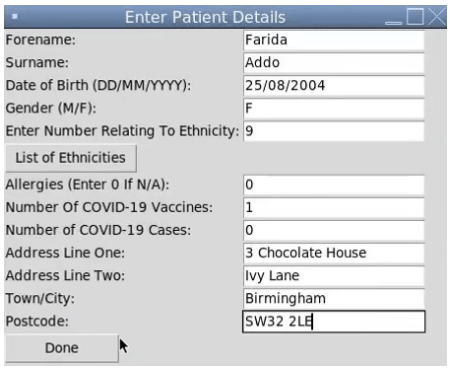
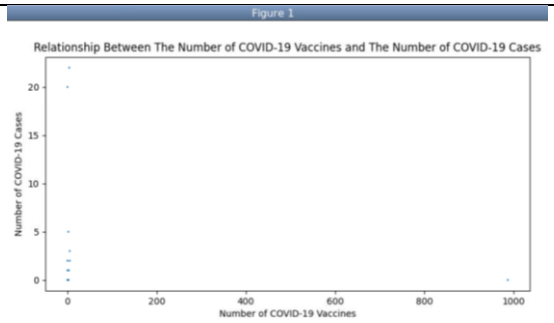
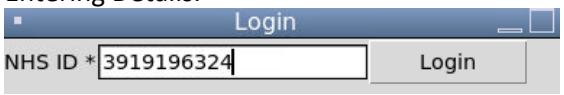
This test was omitted because the chances of the same unique 10 digit number being generated is low. Furthermore, if this occurred then it would be automatically regenerated. Therefore, this test was removed as it would be difficult to conduct.

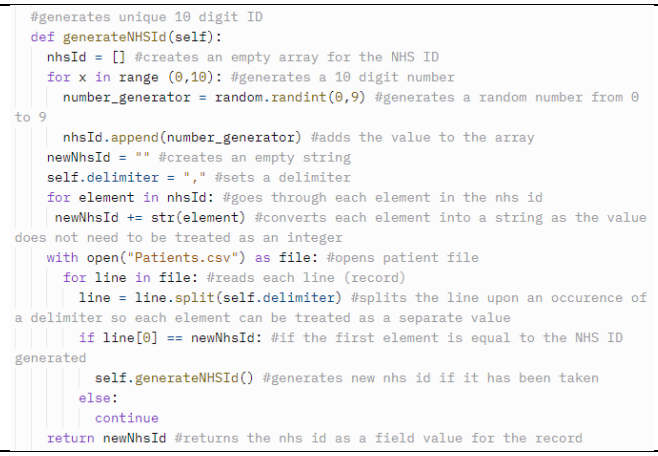
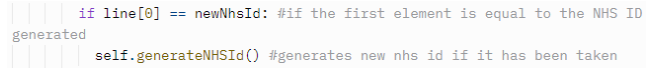
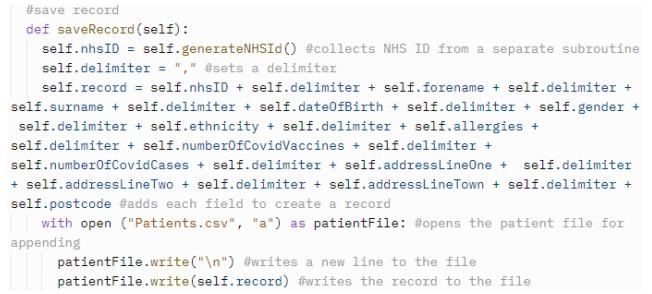
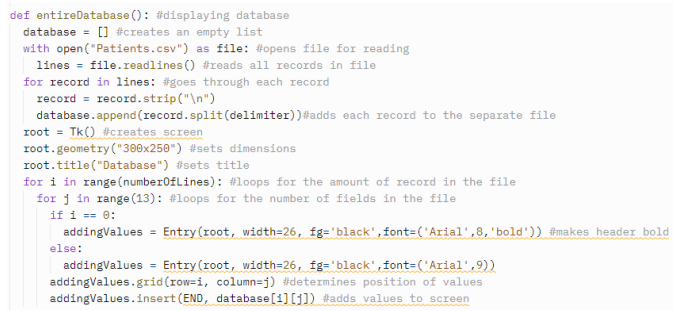
SELF-TESTING AGAINST SUCCESS CRITERIA

As well as testing the system against the pre-development testing table, it was also tested against the success criteria defined earlier. As the program progressed, the success criteria changed to meet the changing functionality of the system. The updated success criteria, and testing can be seen below:

Success Criteria	Justification	Evidence of Success Criteria Being Met
Create home screen.	Users should be able to select an access level. This feature is important as it allows the functionality to differ depending on what the user intends to use the system for. Access levels were also deemed as necessary by 11% of questionnaire respondents, so should be included.	
Create patient login screen.	It is important that a patient who interacts with the system is able to access their data by providing valid credentials.	
Create login page.	It is important that before accessing sensitive data, the authenticity of the user is ensured. This feature is necessary due to being suggested by 16% of questionnaire respondents about have	

	password protection as a security requirement.	
Separate main menus for 'Nurse' access level.	After creating the use-case diagram, it became clear that the Nurse would need a different main menu.	Nurse Main Menu:  Main Menu for Doctor, Scientist, and Government Official: 

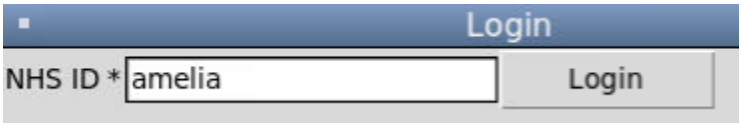
Gather inputs about a patient.	The inputs will provide data used to create the database. These inputs will be derived from the responses produced from the questionnaire i.e. patient allergies, the medical history of patients.	
Enables several patient's details to be inputted.	It is important that a user can enter data for more than one patient if they need to.	
Create a diagram which represents the relationship between the number of vaccines had, and the number of COVID cases.	This will provide a diagram which can be referenced to by doctors when determining the effectiveness of the vaccine.	
Enable a record of a patient's data to be outputted.	This will allow doctors to gather all the details of a patient, which they can then use to access their medical history. This will provide the easy-to-use interface that 75% of respondents selected.	Entering Details:  Output: 

Enable NHS IDs to be created.	This will provide patients with a unique number which will make it easier for doctors to distinguish between patients as some patients may have the same name, causing confusion. 11% of respondents selected patient data, which includes the NHS ID.	
Incorporate checks for NHS ID.	This ensures that multiple people do not have the same NHS ID, as it could lead to the wrong identification of patients	
Write the patient data to file.	This will allow patient details to all be stored in one place, and will essentially create the database.	
Incorporate checks for addresses.	This ensures that the address given by a patient is authentic, allowing doctors to send letters to the correct address.	
Allow entire database to be outputted.	It is important that a suer is able to access the contents of the entire database, so that they can spot potential trends and patterns.	Code:  Output:

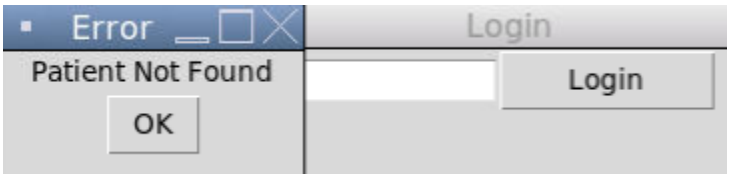
		Database NHS ID | Forename | Surname 1762883873 | Farida | Addo 4813824874 | Sholeye | Olajide 8967463139 | Habibullah | Beverley 6388236637 | Timmy | Tom 9848197844 | Susan | Sally 3919196324 | Lilly | Smith 3752656695 | Susan | Rose 9554744818 | Rafaela | Mendes-Da- 1249766451 | Ryan | Akay 7529882792 | Fajmaela | Aldi 1914236296 | Mockaroo | Tonic 4854166997 | Kayla | Smith 5564528166 | Taylor | Blue 8583335398 | Sally | Lucy
Implement SQL-like statements to allow users to access multiple values	A user may want to access several data items at once, so it is important that they have the ability to do so.	Access Multiple Values SELECT: Options FROM: DATABASE WHERE: FieldName Operator Apply

STAKEHOLDER TESTING

AMELIA

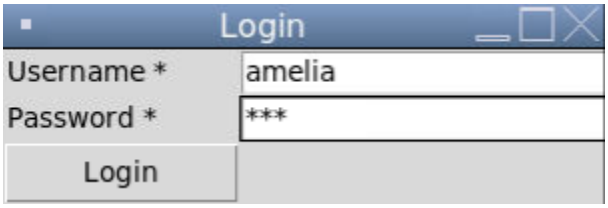


The functionality of the program was proven when the following message was outputted to Amelia:

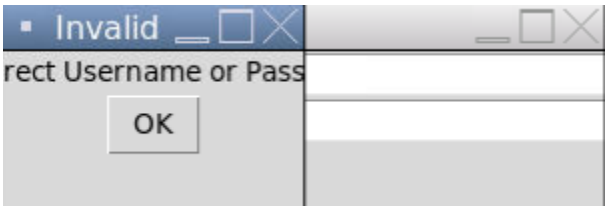


This is advantageous as it demonstrates how the validation coded earlier meant that any inputs which do not relate to an existing NHS ID produced an error message.

Furthermore, when Amelia first started the program and had not registered, she attempted to sign in with credentials (Username: Amelia, Password: 123) which did not exist, which can be seen below:



Yet again, the functionality and robustness of the system was proven as the following message was outputted:

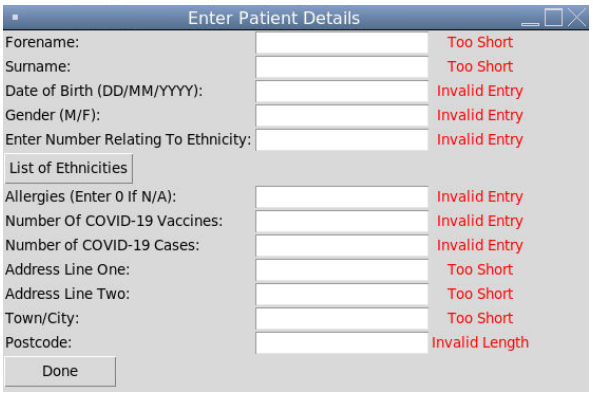


This is advantageous as it demonstrates that the checks within the system to determine if credentials exist in the 'LoginDetails' file worked as intended.

HUMAYRA

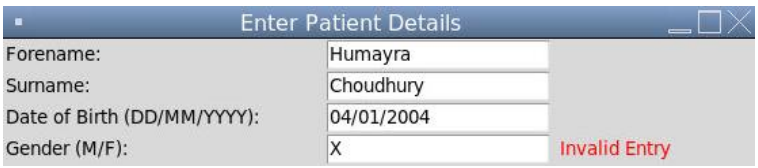
Another stakeholder, Humayra tested out the program.

When she attempted to click 'Done' without adding any input values, the following error messages were displayed.



By not allowing empty entry boxes to be accepted, the program's function works as intended, and it is robust against invalid inputs.

Additionally, when Humayra attempted to enter the gender 'X', an error was produced, as seen below:



By not allowing the invalid input to be accepted, the program's function works as intended, and it is robust against invalid inputs.

Humayra's successful registration, login and entry of patient details following valid inputs is shown below:

ERRORS

After adding her patient entry. Humayra attempted to select records where the allergies contained 'Nuts'. This can be seen below:

When entering the allergies on the 'Enter Patient Details' option, Humayra included 'Nuts'. However, her inability to see this displayed on the screen after adding it was a problem.

This problem reoccurred when Humayra attempted to access the entire database:

After adding her new record, she was unable to see it on the screen.

USABILITY TESTING

Aside from testing for functionality and robustness, it was also important that the usability features outlined earlier (i.e., learnability, memorability, efficiency, customisability) were met, as a huge part of making this system a success is the ability for individuals to interact with the system.

USABILITY FEATURES

LEARNABILITY

When getting stakeholders to interact with the program, they did not require help about what to click, or how to select certain options. For instance, Julie (a stakeholder) knew right away that after she clicked the 'Doctor' option and did not have login details, she was meant to click the register option, where she was then able to use those credentials to log into the system. An example of Julie interacting with the system can be seen below:

MEMORABILITY

Dr. Hussain's feedback featured heavily in the development of the program. As a result, he constantly had to start the program from the beginning each time additions were made. I found that each time he interacted with the system, he was able to perform actions instantaneously as he was used to the system, and it was not difficult for him to remember which options led to which output.

EFFICIENCY

One way to measure efficiency is through success rate, which refers to the ability of users to compete their tasks. All stakeholders were able to perform tasks each time they clicked an option. As a result, the program appears to have good usability due to its high efficiency.

Time taken to complete tasks may also be a determinant of efficiency. I also found that all the users were able to perform tasks quickly – the system provided a quick response time to user inputs, such as a click. It can also be noted that Sophie (a stakeholder) praised the program for providing quick responses.

CUSTOMISABILITY

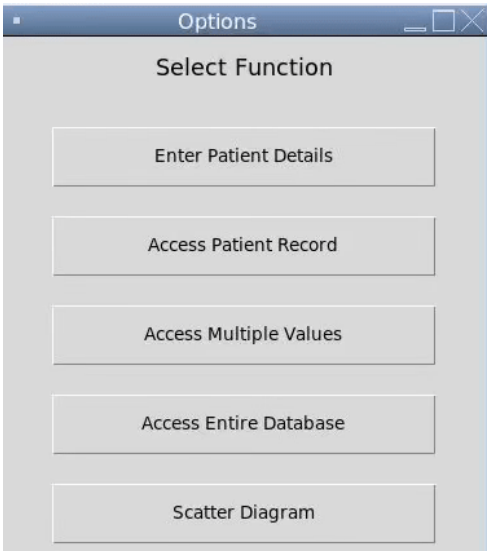
Customisability is another feature which determines the usability of code. As explained earlier, customisability is not a feature of the system, particularly because there are not many user actions which require customisation. Having said this, the screen designs have been developed with neutral designs, in order to benefit everyone.

BUTTONS

Throughout the program, several buttons have many different functions.

CLOSING SCREEN

The test below demonstrates how clicking the 'X' button closes a screen. It is important that this feature works for users who want to close a screen, perhaps because they have received all the data they need. Closing a screen is also important as it prevents data being displayed on a screen for too long, which could in turn be intercepted and be vulnerable to unauthorised access to sensitive data.



MINIMISING SCREEN

Additionally, the '-' button was tested to ensure that it minimised a screen. This feature is also important as users may want to close a screen temporarily so they can access another function. Furthermore, minimising a screen is also another way of preventing the unauthorised viewing of sensitive data, which in turn may pose a security threat. The button working can be seen below:

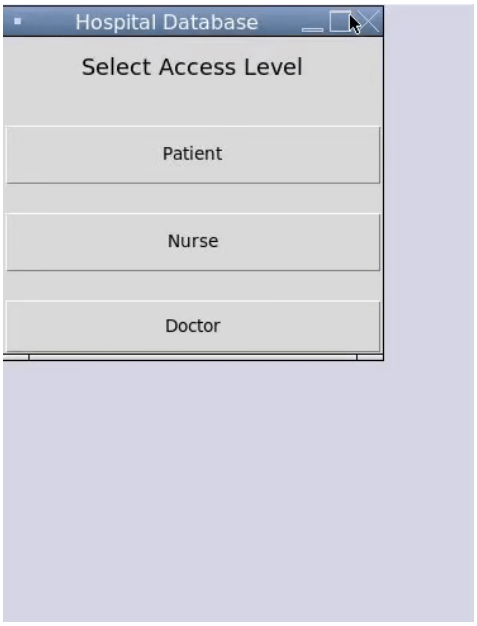
SCREEN CONFIGURATION

Another button which plays an important role, and is a key usability feature, is the window maximiser button. In general, most of the screens are given a size of '300x250'. The code for this can be seen below:

```
mainScreen.geometry("300x250")
```

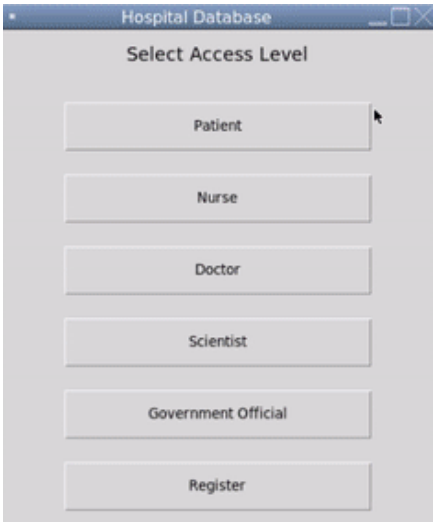
Because the screens are given sizes which tend to be smaller than what is required to display the full contents of a screen, the maximisation button plays a role in ensuring that users are able to see the entire screen where they may need to input data, for instance.

An example of the maximisation button being tested can be seen below:



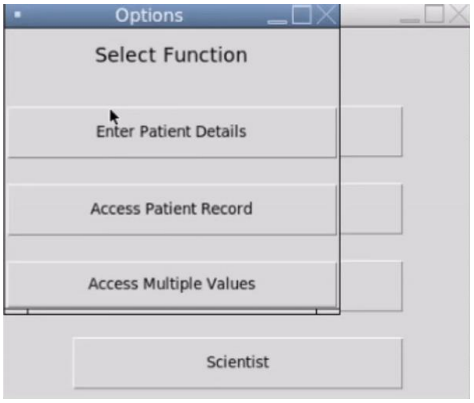
ACCESS LEVEL

When opening the home screen of the page, users are presented with a range of access levels to choose from. Each access level has a corresponding page. As a result, I had to ensure that I tested each access level option, to make sure that it produced the correct screen. For example, clicking the 'Register' button should've taken users to a screen where they can enter login credentials. The testing of each access level button can be seen below:

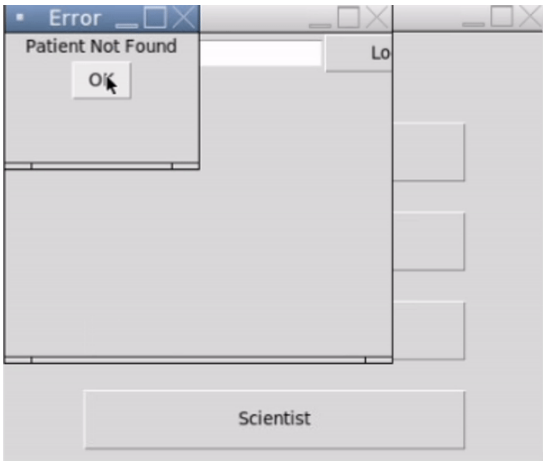


DESTROYING SCREEN

After a user logs in and the login success screen is displayed to them, it is important to ensure that the code which deletes the login success screen once the 'OK' button is pressed, works. The successful testing can be seen below:



Additionally, any error message screens displayed to the user i.e. 'Patient Not Found' also have an 'OK' button which should delete the screen once pressed. Testing that these buttons work can be seen below:

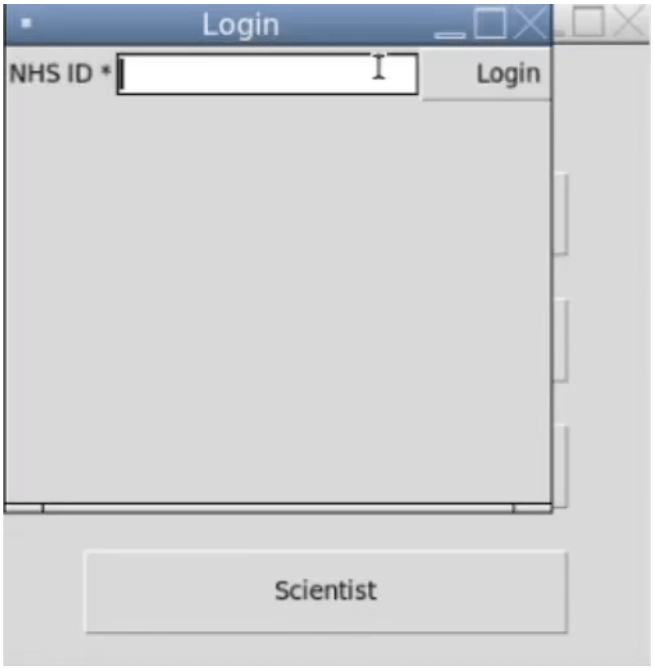


FUNCTION

When accessing the main menu, it is important that the buttons repressing the different functions, produce the different functions. As a result, I conducted tests to ensure that each button would result in the appropriate screen being presented to the user. This can be seen below:

ENTRY BOX

As explained in the design section, it is important that the values a user enters the program is interpreted correctly. For example, when being presented with a login screen where a user enters their credentials, if a user clicks 'Login' then they expect that the boxes they see will be used to enter information that, assuming that the credentials they have entered is correct, will be seen as valid and will transfer them onto the main screen of the program. Tests that some of the entry boxes allow information to be entered in them can be seen below:



CODER

Firstly, I had another coder check the readability of the code that was developed. This was important as the code would most likely be maintained by the admin of a hospital, who would need to understand the code and make improvements and changes to the code if necessary.

The feedback from the coder was that separating the different functions i.e. graph, GUI, into separate python files may make it easier to identify problems in the code, and read the code. Currently, the only separate python file was the class file, so by combining everything in the main file, it may be difficult for a hospital administrator to change certain features as they may not be aware of where the problems reside in the code.

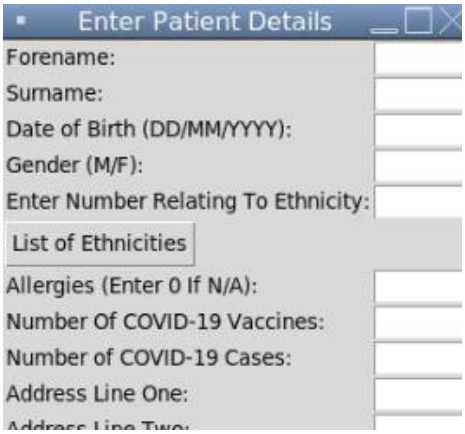
STAKEHOLDER TESTING

The general feedback from the stakeholders was that the program was easy to interact with, however, there were a few points mentioned.

A problem which occurred for some of my stakeholders was the fact that they were unable to enter details into the entry box on their phones – it would only work on a laptop/computer.

DEFAULT SCREEN

When starting the program, each screen is initially minimised. One of the stakeholders, Sophie, found that when she tried to click the 'Nurse' option, she was unaware of what to write for the login details. However, if the screen had been maximised, then she would have been able to see the 'Register' button first.



This problem also persisted with the 'Enter Patient Details' option. The minimised screen meant that users were unable to click 'Done' after an entry, so were forced to maximise the screen. This may be problematic if some users prefer a smaller screen, such as a split screen.

MINIMISING HOME SCREEN

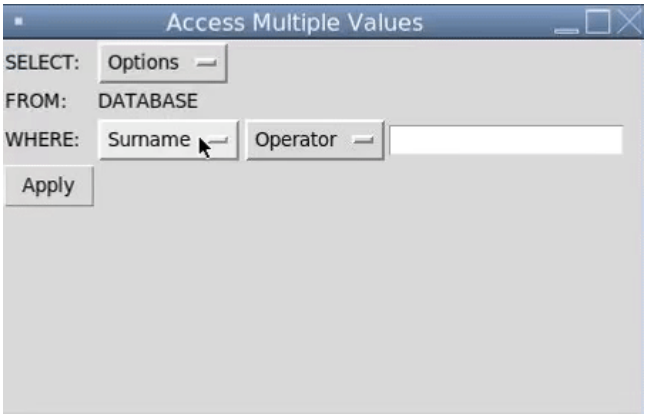
A different stakeholder, and student, Luke faced a problem. For instance, he found that minimising the main menu resulted in the screen not being retrievable. As a result, he had to restart the program each time he minimised the screen. An example of this can be seen below:

JULIE

Another stakeholder who tested my program was Julie.

After choosing the option 'Access Entire Database', she looked at the field values and suggested that the details for each patient were too simplistic. She argued that perhaps it would be better if other patient data was included, such as medical history. As a result, the usability of the program was arguably limited as not much could be done with the data from the field values – they don't give enough insight into the patients and their history very well.

She also found that when using the 'Access Multiple Values' option, and clicking a drop-down menu, the options would disappear instantly at times. As a result, she had to hold the drop-down menu when clicking it. She stated that this was a bit inconvenient, as someone attempting to access multiple data values should be able to click a button easily, without its contents disappearing. An example of what this means can be seen below:



MARK

An additional stakeholder was Mark. He found that in general, the usability of the program was fine. However, he also suggested that the look of the system should be improved to give it a more 'modern' feel.

EVALUATION OF SOLUTION

The following section will focus on the successes, limitations, and potential improvements to the solution.

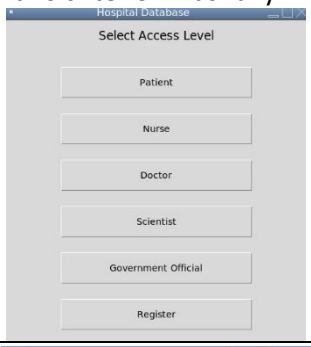
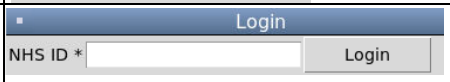
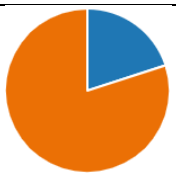
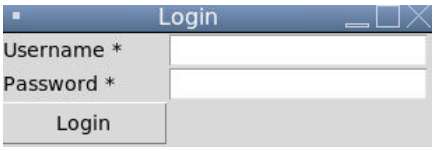
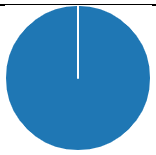
MEETING THE SUCCESS CRITERIA

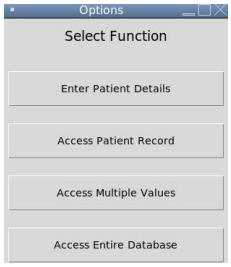
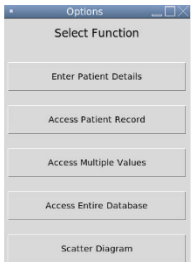
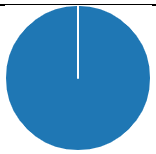
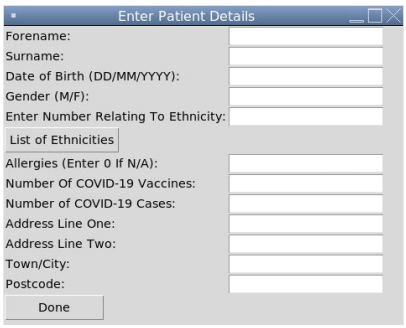
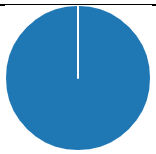
This section will outline the extent to which each success criteria was met.

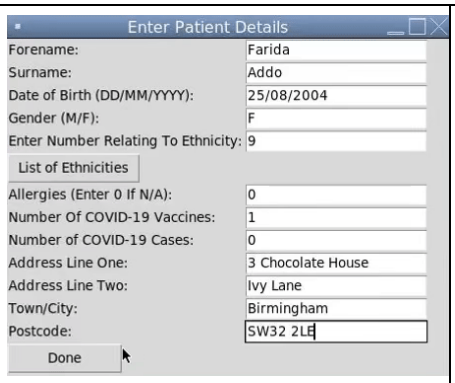
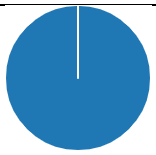
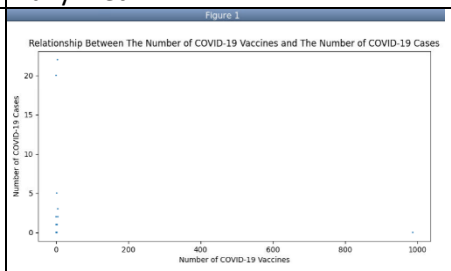
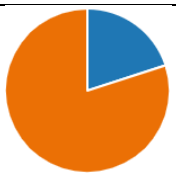
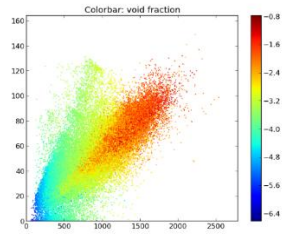
In order to determine whether the final solution matched Dr. Hussain's requirements, I showed him the evidence of the success criteria being met and asked him if it represented his initial requirements.

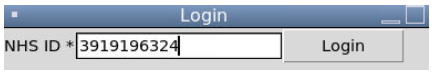
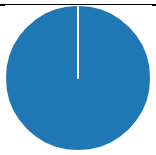
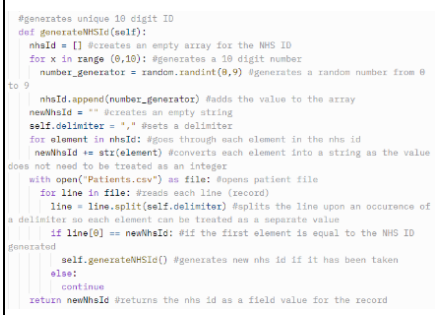
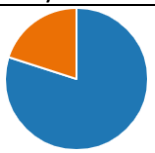
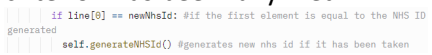
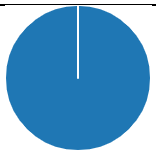
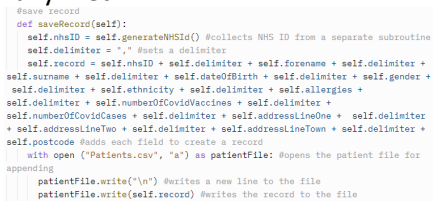
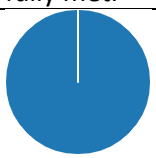
Additionally, I sent a questionnaire to my stakeholders to ask them about how well they believed I met the success criteria.

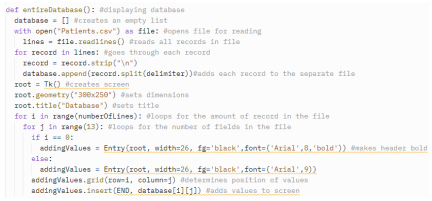
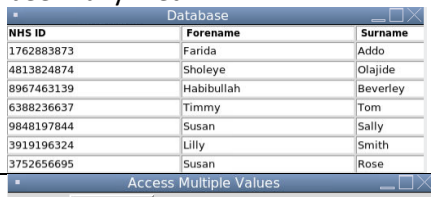
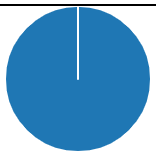
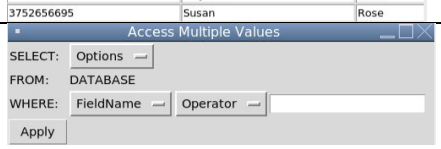
Success Criteria	Extent To Which It Was Met	Questionnaire Responses	Potential Improvements

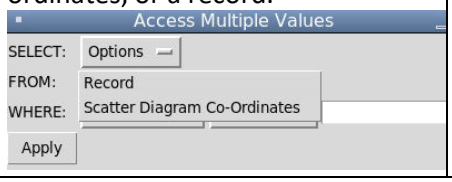
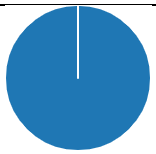
Create home screen.	After asking Dr. Hussain, he suggested that the final design met the initial home screen requirements. I also believe that this was the case because the home screen was meant to have clearly defined access levels that users identified with. As a result, this criterion was fully met. 	 100% of stakeholders selected the option that the success criteria was 'fully met'. This supports the idea that the success criteria was fully met.	Due to the probem posed earlier, where the minimised screen displays only the first three access levels, it may be better to make 'Register' the first option for new users who do not see the entirety of the screen. This means that although they may be unable to see all the access levels, at least they will have some functionality by being able to register.
Create patient login screen.	 The patient login screen may be argued to have been partially met. This is because the login screen was met in the sense that a user would be able to enter credentials, which would then be checked for authenticity. However, a random 10 digit number could be entered, and would provide an external user access to sensitive patient information. A login page is intended to prevent unauthorised access to data, but the lack of reliable credentials required may pose a security threat.	 20% of stakeholders selected the option that the success criteria was 'fully met'. 80% of stakeholders selected the option that the success criteria was 'partially met'. This supports the idea that the success criteria was partially met.	Some form of two factor authentication could be implemented in order to ensure that the user entering the data is who they say they are. For example, most individuals have both a patient number, as well as NHS ID. Therefore, the requirement of a patient number may be able to increase the system's security.
Create login page.	 The login screen was considered to be fully met by both myself, as well as Dr. Hussain. A login screen usually requires a username and password – which is what I designed. Additionally, by preventing users from having the	 100% of stakeholders selected the option that the success criteria was 'fully met'. This supports the idea that the success criteria was fully met.	N/A

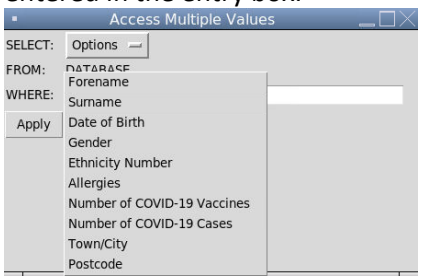
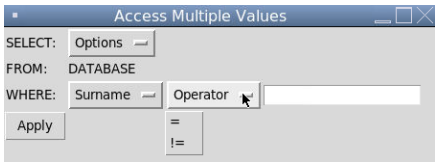
	same username, the login system may be considered to be secure.		
Separate main menu for 'Nurse' access level.	By using the lambda function on Python, I was able to create two separate main menus, where the 'Nurse' main menu does not include the 'Scatter Diagram' function as a nurse is unlikely to require scatter diagram information as part of their daily operations. Overall, Dr. Hussain and I both reached the conclusion that this criterion was fully met.  Nurse Main Menu  Main Menu for Doctor, Scientist, and Government	 100% of stakeholders selected the option that the success criteria was 'fully met'. This supports the idea that the success criteria was fully met.	Having a separate main menu was a requirement of Dr. Hussain's, however, in the long run it may be better for the same main menu to be had for all the access levels (apart from 'Patient') because Nurses not usually utilising the scatter diagram does not mean that there will not be times where their role extends, and that they would need to access the scatter diagram in order to draw conclusions about data.
Gather inputs about a patient.	 The use of entry boxes enables users to enter data which could then be returned into the program as inputs. Therefore, both Dr. Hussain and I agreed that this crietrion was fully met.	 100% of stakeholders selected the option that the success criteria was 'fully met'. This supports the idea that the success criteria was fully met.	N/A

Enables several patient's details to be inputted.	 After a record has been successfully added, the entry boxes are cleared. This means that users are able to enter new data values for another patient. Therefore, this criterion can be said to have been fully met.	 100% of stakeholders selected the option that the success criteria was 'fully met'. This supports the idea that the success criteria was fully met.	N/A
Create a diagram which represents the relationship between the number of vaccines had, and the number of COVID cases.	 The criterion has been perceived to be fully met by both myself, and Dr. Hussain. This is because a clear diagram has been created which plots the COVID-19 vaccination and case values for each record.	 20% of stakeholders selected the option that the success criteria was 'fully met'. 80% of stakeholders selected the option that the success criteria was 'partially met'. This sugggests that overall, the success criteria was considered to be partially met.	The implementation of a colour box could be added. This could set plots as different colours, depending on how common a plot value is. For instance, the plot values will get bigger each time certain values are repeated. However, this increase in size may not be picked up by the human eye very well. Therefore, adding colour changes depending on how common co-ordinates are can make spotting trends easier. An example of such a colour bar can be seen below: 

Enable a record of a patient's data to be outputted.	Entering a valid NHS ID, such as the one below  results in the following record being produced.  Therefore, this criterion has been fully met.	 100% of stakeholders selected the option that the success criteria was 'fully met'. This supports the idea that the success criteria was fully met.	N/A
Enable NHS IDs to be created.	 The code for developing the NHS ID can be seen above. The code results in the creation of a 10-digit string, also known as the NHS ID. Therefore, this criterion has been fully met.	 20% of stakeholders selected the option that the success criteria was 'partially met'. 80% of stakeholders selected the option that the success criteria was 'fully met'. This supports the idea that the success criteria was fully met.	N/A
Incorporate checks for NHS ID.	Validation has been added to ensure that an NHS ID which matches an existing one is re-generated, to ensure that the NHS ID is truly a unique identifier – a primary key. Therefore, this criterion has been fully met. 	 100% of stakeholders selected the option that the success criteria was 'fully met'. This supports the idea that the success criteria was fully met.	N/A
Write the patient data to file.	The code below demonstrates how each field value is added together to create a record, which is then written to the 'Patients' file. Therefore, this criterion has been fully met. 	 100% of stakeholders selected the option that the success criteria was 'fully met'. This supports the idea that the success criteria was fully met.	N/A

<table>
<tr>
<td>Incorporate checks for addresses.</td>
<td>The code below checks the length of the addresses, and ensures that it is equal to, or above 2. As a result, a check has been created and this criterion has been partially met.
  </td>
<td> 
20% of stakeholders selected the option that the success criteria was ‘fully met’.
80% of stakeholders selected the option that the success criteria was ‘partially met’.
This suggests that this success criteria was percieved to be partially met.</td>
<td>An addition to the address check could have been ensuring that no symbols or special characters were able to be considered valid. The only check regards the length of the input, however it does not account for the values themselves that the user will enter. For instance, an address which contains a semicolon is not valid, so should not be accepted (but would be in this system due to lack of validation).</td>
</tr>
<tr>
<td>Allow entire database to be outputted.</td>
<td>The code below reads through the ‘Patients’ file and outputs each record in separate boxes, with the header field names in bold.
 
This code results in the output seen below. Therefore, this criterion has been fully met.
 
</td>
<td> 
100% of stakeholders selected the option that the success criteria was ‘fully met’. This supports the idea that the success criteria was fully met.</td>
<td>Following Humayra’s test where she attempted to access the entire database and was unable to see her most recent addition of the patient record, the code could be changed to ensure that new additions to the ‘Patients’ file are accounted for.</td>
</tr>
<tr>
<td>Implement SQL-like statements to allow users to access multiple values.</td>
<td> 

The common ‘SELECT FROM WHERE’ SQL statement has been added to the program. Users are able to determine whether they want to select scatter diagram co-ordinates, or a record.
 
</td>
<td> 
100% of stakeholders selected the option that the success criteria was ‘fully met’. This supports the idea that the success criteria was fully met.</td>
<td>Due to the issue regarding Humayra’s testing earlier where new patient records would not be displayed, the code should be updated in a way in which allows it to display all values relating to the user options selected, regardless of how new the user entry was.
There could also be an additional improvement to the ‘WHERE’ statement.</td>
</tr>
</table>

	They are also able to determine the field name, as well as the operator, which they will use to compare the field value of each record against the input value, which would be entered in the entry box.   Therefore, this criterion has been fully met.		For instance, Humayra found that when she attempted to find records containing the allergy 'Nuts', she only found records where the allergies were only 'Nuts'. When entering patient details, she created a patient with several allergies, where 'Nuts' was a part of this. However, this was not displayed to her as she used the operator '=' which means that only records containing exactly the inputted value will be displayed. In light of this, the 'WHERE' statement could be extended to allow a statement such as 'WHERE 'Nuts' IN Allergies' to be accepted. The use of terms like 'IN' will extend the functionality of this option as a user can access records where a certain value is present, without having to state all of the details for that particular field element.

Overall, the success criteria were met effectively, and can be confirmed by the stakeholders, and Dr. Hussain's responses. The final solution was able to meet the original requirements, which is a success of the program.

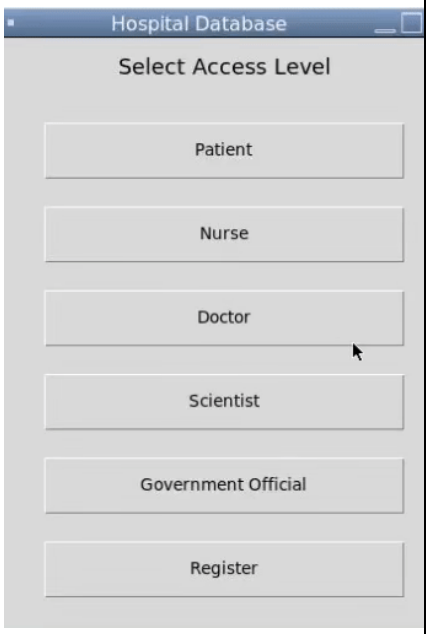
MEETING THE USABILITY FEATURES

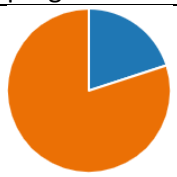
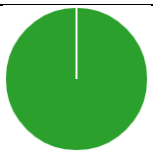
This section will outline the extent to which the usability features were met.

In order to determine whether the final solution matched Dr. Hussain's requirements, I showed him the evidence of the usability features being met and asked him if it represented his initial requirements.

Additionally, I sent a questionnaire to my stakeholders to ask them about how well they believed I met the usability features.

Usability Feature	Extent To Which It Was Met	Questionnaire Responses	Potential Improvements

Learnability	The video below represents how Julie, one of my stakeholders, interacted with the program the first time she used it.  This demonstrates how learnable the program is, as someone who had only used the program for the first time was able to understand that not being able to login after selecting an access level meant that they should register. Additionally, the program's use of buttons and entry boxes were widely understood by the stakeholders. Therefore, the learnability of the program was successful due to the absence of complex commands and a complex design.	 100% of stakeholders selected that the learnability of the program was successful. This supports the idea that learnability was a successful usability feature in the program.	N/A
Memorability	As a result of the simple screen designs, and use of buttons and entry boxes which are used in everyday operations, the memorability of the system was successful. Dr. Hussain also credited the system's memorability.	 100% of stakeholders selected that the memorability of the program was successful. This supports the idea that learnability was a	N/A

		successful feature in the program.	
Efficiency	The final system was considered very efficient by Dr. Hussain, and myself. For instance, clicking each button would result in a quick system response. Additionally, performing tasks did not take long. For instance, a user clicking 'Access Entire Database' did not have to wait long to see the entire database. Therefore, efficiency has been successful in the program.	 20% of stakeholders selected the option that the usability feature was successful. 80% of stakeholders selected the option that this usability feature was 'partially successful'. This suggests that the usability feature was partially met.	An additional potential improvement to the usability features would be the inclusion of a scroll bar. This would overcome the issue of users not being able to view the whole screen without maximising it. As a result, the usability will improve as users will be able to use the program whether its maximised, or minimised. It also gives them the control of having the screen contents which are most relevant to them displayed. This would be especially efficient in the case of outputting a record, or entire database, where the numerous amounts of records cannot be viewed unless a user maximised the screen heavily.
Customisability	The criteria of customisability was unmet. This was because there were no features which allowed users to change the font type, or colours of the screen, for instance. Dr. Hussain also critiqued the lack of customisability in the program.	 100% of stakeholders selected that the customisability of the program was unmet. This reiterates the idea that customisability was an unsuccessful feature in the program.	The code should have a settings page where users can change the appearance of the screen, such as from light to dark mode. Additions should also be made which allow users to change text size and colour. Furthermore, code should be added that allows users to choose between different themes. For instance, a user may choose a default theme – what I developed – as well as others with a range of colours.

Button	The addition of the button features was highly successful. The buttons ranged from minimising/maximising screens, closing screens, to performing functions, such as leading users to login screens, or allowing them to enter patient details. The tests conducted earlier are reflective of how a user clicking one button would lead them to a specific output. Therefore, the button feature may be classed as successful.	 100% of stakeholders selected that the button of the program was met. This reiterates the idea that button was a successful feature in the program.	N/A
Entry Box	The addition of the entry boxes was deemed as successful by both me, and Dr. Hussain. This is because the purpose of the entry box was to allow information to be entered, which can then be processed by the computer. For instance, entering login details would result in the inputs being checked against the existing login details to ensure that the data. Therefore, this usability feature was successful.	 100% of stakeholders selected that the entry boxes of the program was met. This reiterates the idea that entry boxes were a successful feature in the program.	N/A

Overall, the implemented usability features were deemed as successful by both me, Dr. Hussain and the stakeholders. By creating a system with high usability, the needs of 75% of questionnaire respondents who deemed having an easy-to-use software interface as important has been met.

POTENTIAL IMPROVEMENTS

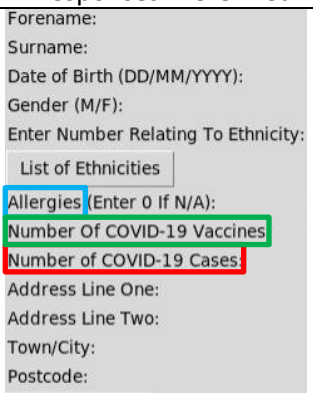
An additional feature which would improve the usability of the system would be the addition of a help button. This could be where each screen has a '?' button, which they can click. This would tell them what functions can be performed, as well as how to perform them. As a result, those who are new to the system, such as the 44% of questionnaire respondents who said that they do not know how to use a database, will not have problems with using new functions. Additionally, it could help improve the memorability of the program as users may get so used to clicking the help button, that there may come a time where they no longer need it, as they have memorised how the system works.

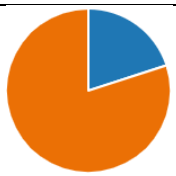
MEETING THE QUESTIONNAIRE RESPONSES

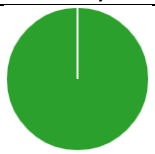
At the beginning of the program development, a questionnaire was given out to a range of individuals to provide a guide as to what the program should include. The following section will outline the extent to

which some of the features proposed by the questionnaire respondents were met. Determining this extent will be done by myself and Dr. Hussain.

Additionally, I sent a questionnaire to my stakeholders to ask them about how well they believed these questionnaire responses were met.

Question	Questionnaire Responses	Extent To Which Responses Were Met	Stakeholder Questionnaire Responses	Potential Improvements
What kind of data do you wish to store in the database?	- Patient Data - Vaccination status - COVID cases - Medical history - Allergies - All of the above	 The final solution provided fields for 'patient data', 'COVID cases' (in red), and 'allergies' (in blue). The 'vaccination status of patients' was changed to the number of COVID-19 vaccines (in green). This is because creating a scatter diagram is difficult with a qualitative 'Y' or 'N' value. Furthermore, having a more precise value for the number of COVID-19 vaccination is better, in relation to drawing conclusions. Overall, the addition of the different patient details followed the questionnaire responses. However, the 'medical history of patients' was unaccounted for as it would be hard to maintain when coding. For instance, people could have different	 46% of stakeholders selected that the questionnaire responses were fully met, whereas 54% selected that this questionnaire response was partially met. of the program was successful. Additionally, Dr. Hussain believed that these questionnaire responses were partially met.	The program should be amended so that it enables an input for the medical history of patients to be added. This is an essential feature, that even one of the stakeholders, Julie, brought up.

		spellings, or, when entering data it may not be in the correct format. Because of this, this feature wasn't included. Therefore, the questionnaire responses were arguably partially met.		
What are your security requirements?	- Encryption - Password protection - User access levels - Firewalls - Anti-virus - All of the above	The password protection was incorporated through the user of logins, registrations and NHS ID provision. User access levels are also evident through the use of 'Patient, 'Nurse', 'Doctor', 'Scientist', 'Government Official'. However, encryption, firewalls and anti-virus was not incorporated. Therefore, the questionnaire responses were partially met.	 20% of stakeholders selected the option that the all the security requirements were fully met. 80% of stakeholders selected the option that this usability feature was was 'partially successful'. This suggets that the security requirements was partially met. Additionally, Dr. Hussain believed that these questionnaire responses were partially met.	Encryption could be incorporated through the use of hashing. Hashing is a one-way form of encryption, which could be applied to the passwords. This would make it difficult for hackers to access the login credentials, even if they accessed the system. The hashing algorithm could also be applied to the database contents. Anti-virus and firewall systems are out of my control, and should be something added by the user separately. However, the program could be adapted, where a message is displayed when a user first uses the system, which states that the system cannot run without an

				appropriate firewall or anti-virus installed. This would help ensure the security of the system.
Is data visualisation important to you?	63% - Yes 20% - No 18% - Doesn't matter	The formation of the scatter diagram is an example of data visualisation, which has been successful. Therefore, the requirement of 63% of questionnaire respondents has been met.	 100% of stakeholders selected that data visualisation was met. This supports the idea that data visualisation was a successful feature in the program. Additionally, Dr. Hussain believed that this questionnaire response was fully met.	N/A
Do you wish to connect this system to any other application?	46% - Yes 54% - No	This requirement was deemed as unnecessary as 54% of questionnaire respondents did not require it. Therefore, the implementation of this requirement was unsuccessful.	 100% of stakeholders selected that this requirement was unmet. Additionally, Dr. Hussaim believed that this was unmet.	The system could be improved so that it is imported into the cloud. As a result, it may become more accessible from a range of locations. This would improve the efficiency of the program for users. This is especially important for a user, such as a patient who would only need to hold one record. However, the system would take up a large proportion of their memory unnecessarily.

Overall, the system was successfully built in line with the requirements set by the users from their questionnaire responses. This conclusion is also supported by Dr. Hussain and the stakeholders.

MEETING THE VARIABLES AND VALIDATION

The following section will provide evidence for how the proposed variables and validation are implemented in the final code:

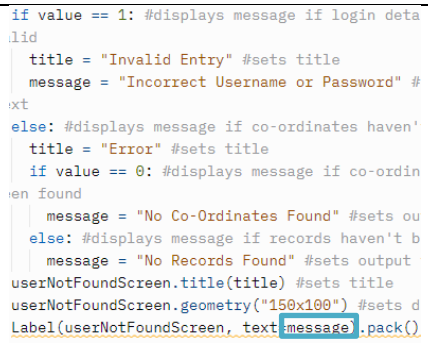
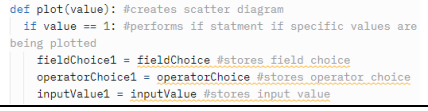
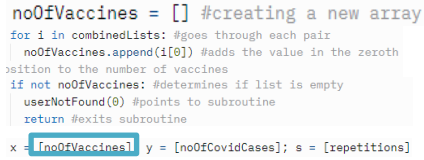
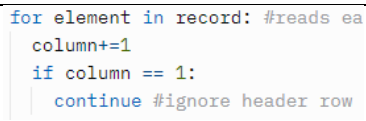
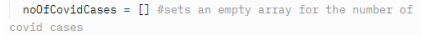
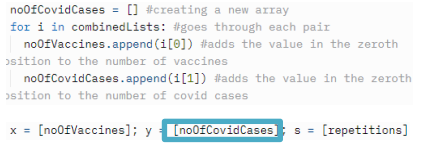
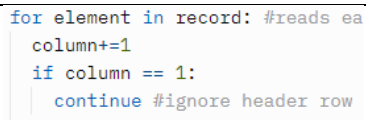
Name/Type	Evidence of Creation	Validation	Evidence of Validation
Main Program			
delimiter - constant	`global delimiter` `delimiter = ","`	N/A	N/A

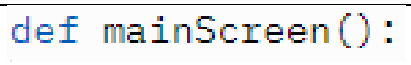
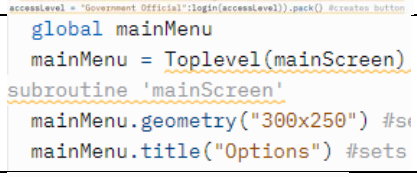
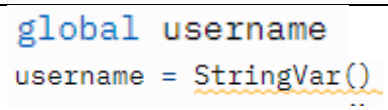
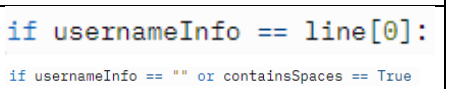
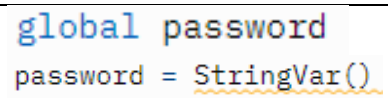
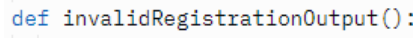
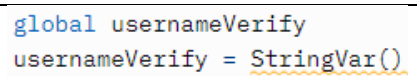
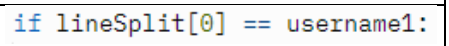
existenceOfFile – Boolean	`existenceOfFile = os.path.isfile("Patients.csv")`	Must be true as each record requires a storage space. If not, the file is created.	`if existenceOfFile == False:`
			`  with open("Patients.csv", "w") as patientFile:`
			`  string = "NHS ID, Forename, Surname, Age, Gender, Ethnicity, Allergies, Number of COVID-19 Vaccines, Number of COVID-19 Cases, Address Line One, Address Line Two, Town/City, Postcode" #sets the first line of the patient file`
			`  patientFile.write(string) #writes the first line to file`
			Creation of file can be seen, along with the field names being written to the file.

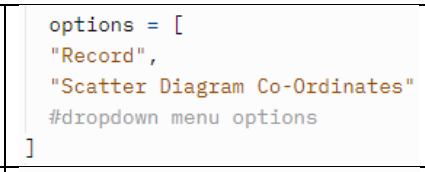
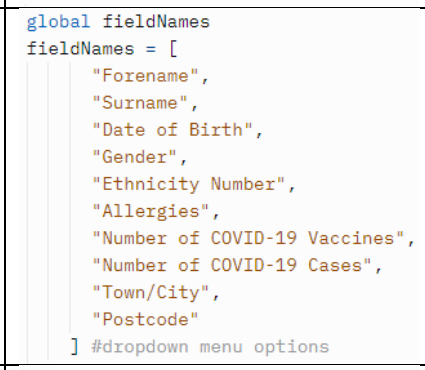
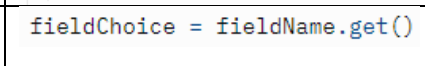
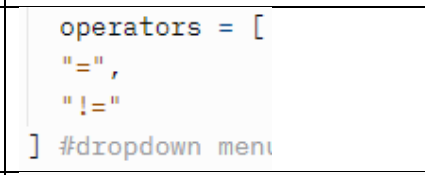
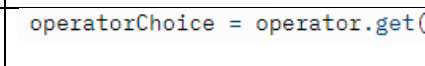
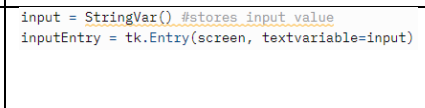
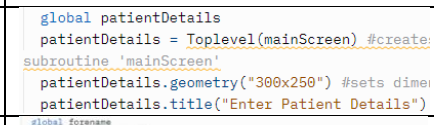
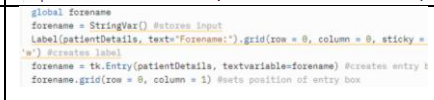
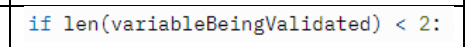
numberOfLines – variable	`global numberOfLines`
	`numberOfLines = 0 #setting the initial number of lines to 0`
	`with open("Patients.csv","r") as file: #opens the patient file for reading`
	`  for i in file: #reads each line of the patient file`
	`  numberOfLines += 1 #adds one to the variable each time there is an additional number of lines`
	Creation of the variable can be seen as it reads the 'Patients' file and determines how many lines it has.
		N/A	N/A

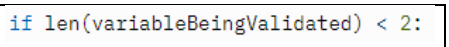
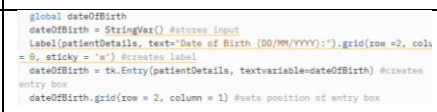
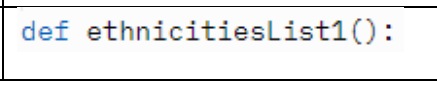
userNotFound – subroutine	`def userNotFound(value):`
	`  global userNotFoundScreen`
	`  userNotFoundScreen = Toplevel(loginScreen) #screen defined in the 'loginScreen' subroutine`
	`  if value == 1: #displays message if login details valid`
	`  title = "Invalid Entry" #sets title`
	`  message = "Incorrect Username or Password" #sets message text`
	`  else: #displays message if co-ordinates haven't been found`
	`  title = "Error" #sets title`
	`  if value == 0: #displays message if co-ordinates haven't been found`
	`  message = "No Co-Ordinates Found" #sets message`
	`  else: #displays message if records haven't been found`
	`  message = "No Records Found" #sets output message`
	`  userNotFoundScreen.title(title) #sets title`
	`  userNotFoundScreen.geometry("150x100") #sets size`
	`  Label(userNotFoundScreen, text=message).pack() #displays error message`
	`  Button(userNotFoundScreen, text="OK", command=deleteUserNotFoundScreen).pack() #deletes screen once the button is pressed`
		N/A	N/A

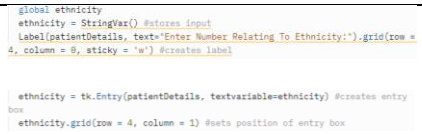
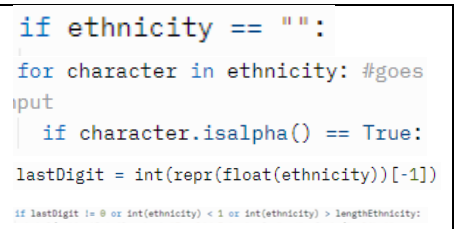
title - constant	`if value == 1: #displays message if login details valid`
	`  title = "Invalid Entry" #sets title`
	`  message = "Incorrect Username or Password" #sets message text`
	`else: #displays message if co-ordinates haven't been found`
	`  title = "Error" #sets title`
	`  if value == 0: #displays message if co-ordinates haven't been found`
	`  message = "No Co-Ordinates Found" #sets message`
	`  else: #displays message if records haven't been found`
	`  message = "No Records Found" #sets output message`
	`userNotFoundScreen.title(title) #sets title`
	'title' is based on the parameter 'value'. Determining the value of 'title' is done through a selection statement.
		N/A	N/A

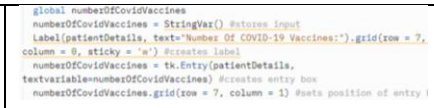
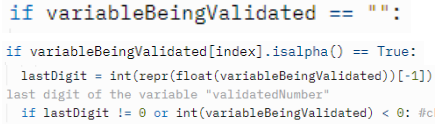
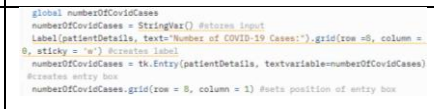
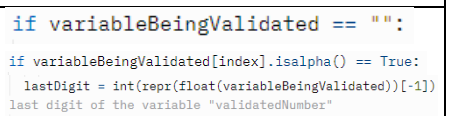
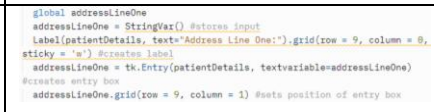
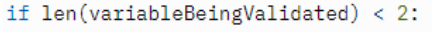
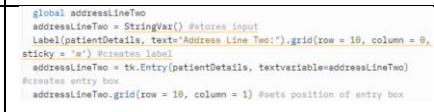
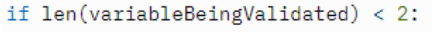
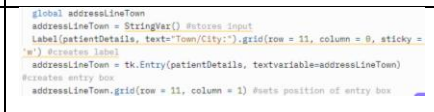
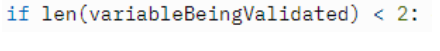
	The use of 'title' can be seen in blue.		
message - string	 'message' is based on the parameter 'value'. Determining the value of 'message' is done through a selection statement. The use of 'title' can be seen in blue.	N/A	N/A
plot – subroutine	The beginning of the subroutine can be seen below: 	N/A	N/A
noOfVaccines – array	This variable is made twice. The first time can be seen below:  The second time can be seen below:  The variable is used as a plot value, seen in blue.	Must be an integer. For instance, the first line of the patients CSV file will be the field names, and so the element containing the number of vaccines will be a string. As a result, this first line should be skipped.	 By skipping the first column in each record, which contains string (the field names), the values accepted into the array are integers.
noOfCovidCases - array	This variable is made twice. The first time can be seen below:  The second time can be seen below:  The variable is used as a plot value, seen in blue.	Must be an integer as the number of COVID-19 cases would never take a Boolean value, for instance.	 By skipping the first column in each record, which contains string (the field names), the values accepted into the array are integers
repetitions – variable		N/A	N/A

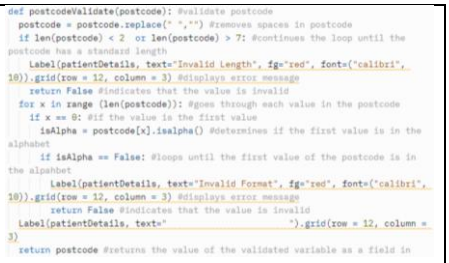
	The variable is used to determine the size of the plot values, seen in blue.		
mainScreen – subroutine		N/A	N/A
accessLevel – variable		N/A	N/A
mainMenu – variable		N/A	N/A
username - variable		If the username is taken, it cannot be used. It is important that uniqueness is maintained within the system in relation to usernames. The username also cannot be empty, and cannot contain spaces as it is a common form of validation.	
password - variable		The password input cannot be empty. Also, cannot contain spaces as it is a common form of validation.	
invalidRegistrationOutput – subroutine		Subroutine will be pointed to when the user has provided an incorrect username and/or password.	N/A
usernameVerify – variable		NHS ID entered must exist in the database, or an error message will be outputted to the user.	

options – variable		N/A	N/A
fieldNames – variable		N/A	N/A
fieldChoice – variable		N/A – there are options for the users to choose from	N/A
operators – variable		N/A	N/A
operatorChoice – variable		N/A – there are options for the users to choose from	N/A
inputEntry – variable		N/A – the entry will be checked and whether or not the scatter diagram co-ordinates or record values are found will be displayed to the user.	N/A
patientDetails – variable		N/A – validation will be for each individual input.	N/A
forename – variable		Must be at least 2 characters as a forename is generally at least 2 characters long. This can be alphanumeric as names take different forms nowadays.	

Surname – variable		Must be at least 2 characters long as a surname is generally at least 2 characters long. This can be alphanumeric as names take different forms nowadays.	
dateOfBirth – variable		Day has to be from 1 to 31 as there are no more than 31 days in the month. Month has to be between 1 and 12 as there are no more than 12 months in a year. Using the 'strptime' function in python only limits the year to being 4 digits. While it would be better if the year could not be longer than the current year the user is using the system, this is a constraint which cannot be overcome.	
Gender – variable		Must be either 'M' or 'F'. there is no need for the full 'Male' and 'Female' terms to be stored as it would waste storage.	
ethnicitiesList – subroutine		N/A	N/A

ethnicity – variable		Number must be a whole number between 1 and the digit of the last ethnicity displayed. It cannot be more than the maximum digit as this does not correspond to an ethnicity, and thus an ethnicity cannot be derived from this for the record.	
validatedEthnicity – variable		N/A – the value would already be validated at this point.	N/A
Allergies – variable		If a patient has no allergies, 0 should be entered and no other number. Entering 1 or 4 without stating what the allergies are would not be of use to a doctor. On the other hand, if a user has not entered a number then a blank input cannot be entered (as is the case with the other inputs), any string cannot contain a number as alphanumeric words do not constitute to an allergy. Finally, the length of the allergy has to be at least 2 characters as generally, there are no allergies less than 2 characters long.	

numberOfCovid Vaccines – variable		The only validation is that the number cannot be negative or decimal. This is because there is no such thing as having -1 or 3.4 number of vaccines, for example.	
numberOfCovid Cases – variable		The only validation is that the number cannot be negative or decimal. This is because there is no such thing as having -1 or 3.4 number of covid cases, for example.	
addressLineOne – variable		The address name can be alphanumeric, but has to be at least 2 characters long as it is unlikely that there is an address name of less than 2 characters.	
addressLineTwo – variable		The address name can be alphanumeric, but has to be at least 2 characters long as it is unlikely that there is an address name of less than 2 characters.	
addressLineTown – variable		The town name can be alphanumeric, but has to be at least 2 characters long as it is unlikely that there is a town name of less than 2 characters.	

Postcode – variable		It is hard to determine what validation to include for postcodes as they vary substantially. As a result, the only validation is that it must begin with a letter, as this is common across all postcodes. Additionally, the postcode has to be between 2 and 7 characters long – as this is the minimum and maximum length. The orders of letters and numbers vary so this will not be a form of validation.	
detailsVerify - subroutine	`def detailsVerify():`	N/A – user inputs will be validated in their separate subroutines.	N/A
submitDetails - subroutine	`def submitDetails(detailsVerify):`	N/A – values will already be validated.	N/A
Patient Class			
__init__ - constructor method	`def __init__(self, record):`	N/A – values will already be validated.	N/A
forename - attribute	`self.forename = record[0]`	N/A – value will already be validated.	N/A
surname - attribute	`self.surname = record[1]`	N/A – value will already be validated.	N/A
dateOfBirth - attribute	`self.dateOfBirth = record[2]`	N/A – value will already be validated.	N/A
gender - attribute	`self.gender = record[3]`	N/A – value will already be validated.	N/A

ethnicity - attribute	`self.ethnicity = record[4]`	N/A – value will already be validated.	N/A
allergies - attribute	`self.allergies = record[5]`	N/A – value will already be validated.	N/A
numberOfCovid Vaccines - attribute	`self.numberOfCovidVaccines = record[6]`	N/A – value will already be validated.	N/A
numberOfCovid Cases - attribute	`self.numberOfCovidCases = record[7]`	N/A – value will already be validated.	N/A
addressLineOne - attribute	`self.addressLineOne = record[8]`	N/A – value will already be validated.	N/A
addressLineTwo - attribute	`self.addressLineTwo = record[9]`	N/A – value will already be validated.	N/A
postcode - attribute	`self.addressLineTown = record[10]`	N/A – value will already be validated.	N/A
generateNHSId - subroutine	`def generateNHSId(self):`	N/A	N/A

nhsID – variable	`nhsId = []`
	`newNhsId = ""`
	`for element in nhsId: #go`
	`newNhsId += str(element)`
		Must be 10 digits long, as it the standard NHS ID length. Also cannot be a pre-existing value.	`for x in range (0,10): #generates a 10 d`
			`number_generator = random.randint(0,9)`
			`with open("Patients.csv") as file: #op`
			`for line in file: #reads each line (`
			`line = line.split(self.delimiter)`
			`elimiter so each element can be treated`
			`if line[0] == newNhsId: #if the fi`
			`erated`
			`self.generateNHSId() #generates`

numberGenerator – variable	`number_generator = random.randint(0,9)` `nhsId.append(number_generator) #adds t`	N/A	N/A

newNhsId – variable	`newNhsId = ""`
	`for element in nhsId: #go`
	`newNhsId += str(element)`
		N/A	N/A

saveRecord - method	`def saveRecord(self):`	N/A	N/A
record – variable	`self.record = self.nhsID + self.delimiter + self.forename + self.delimiter + self.surname + self.delimiter + self.dateOfBirth + self.delimiter + self.gender + self.delimiter + self.ethnicity + self.delimiter + self.allergies + self.delimiter + self.numberOfCovidVaccines + self.delimiter + self.numberOfCovidCases + self.delimiter + self.addressLineOne + self.delimiter + self.addressLineTwo + self.delimiter + self.addressLineTown + self.delimiter + self.postcode #adds each field to create a record`	N/A	N/A

Overall, I was able to effectively replicate the key variables and validation.

MAINTENANCE ISSUES OF SOLUTION

CURRENT MAINTENANCE

In relation to maintaining the system, there is not necessarily a system built in place. However, when developing the code, comments were added so that a developer could see the purpose of each line. The variables were also given meaningful names. These features make it easier for developers to identify problems in the code, and solve it accordingly.

Aside from this maintenance feature, there is a small limitation with the code. For example, most of the code resides in a 'main' file. This poses maintenance issues as it means that a developer would have to go through 900 lines of code to determine where the code for validating the allergies input is, for example. This makes the program time consuming to maintain as looking for errors in the code for modification will take a while.

FUTURE MAINTENANCE

One way to overcome the time-consuming maintenance nature of the code is by splitting up the main python file into separate files which each have an individual purpose. For instance, one python file could be for the entering patient details option, and another subroutine could be for creating the scatter diagram. Whilst this may increase the file size of the system, it will also improve its maintainability.

Additional maintenance features may relate to the security of the data. For instance, due to the sensitive nature of the data being stored, full backups will be performed automatically, each time a change is made to the database. By incorporating this mechanism, if the system shuts down or temporarily stops working, the data can be retrieved. This is particularly important as information about each patient will be time consuming and costly to retrieve.

The system will also be made to work offline. This is important as it overcomes any issues that stem from relying on the internet in order to run the system. As such, the program will have increased usability. It also means that the program can be maintained in the case of an outage.

As more pieces of data are stored, it is important that there is enough space to capture such data. In light of this, mechanisms should be put in place to enable users to expand the data they capture, such as increased storage, as specified in the system hardware requirements.

A useful feature to be incorporated would be turning the system into an application. Currently, it is likely to be used on a browser. However, digitalisation will enable the system to be accessible to the masses. As a result, it may potentially be able to reach the billions of people who have a phone. This will mean that the system is not limited to use in the UK. Additionally, nowadays everyone has their phone/device with them everywhere they go. As a result, it would be useful for users to be able to open an application and access patient details quickly. What this will mean is that the compatibility with different devices and software (i.e. iOS) will need to be considered. Ultimately, the system should be extended for a mobile platform. In order for this to work, device integration may be a feature considered for future maintenance. The different devices used by one user should be connected to enable seamless data exchange. As a result, users will be able to continue from where they left off, increasing the efficiency of transaction processing. It also allows users to be more flexible in their work operations, improving user

experience with the system. By incorporating this feature, portability is incorporated in the program, thus optimising workflows.

When issuing the system to individuals, a user guide may also be given to them simultaneously. This will allow users to understand how to interact with the system. Although the system was developed with the intention of being highly usable by all individuals, it would still be useful to give users a guide as to what they can do. This will make teaching others how to use the system easier.

Updates will be released every quarter, with updated version of the system. This is because development may take a while as user feedback is everchanging. By releasing an update every quarter, there is sufficient time for developers to collect information about the additions users would like, which can then be implemented. These updates will also take into consideration any errors reported by users, and ensure that changes are made accordingly. In addition, beta testing of the system may be implemented commonly after each new update occurs. This is because it gives users a chance to interact with the system and give insights into what went well and what could be improved, resulting in subsequent amendments. Having constant user feedback is important as new updates are made of the system. This is because a user will have constant interactions with the system, so will know what they want. If it was down to solely the developer to make changes, they may add features of no use. Therefore, it is important that the suggestions of users are replicated, once again extending the use of the system. Moreover, a menu bar may be created where one of the options are to scan for updates in the system. Users can change their setting to automate, or manually perform these updates. Upon updation, users will be presented with a screen which outlines new features and changes made to the system.

EVALUATING THE SOLUTION CODE

The following section will outline the strengths and limitations of each section of code that was written.

HOME SCREEN CODE

STRENGTHS

The home screen code was efficient in designing the user access level, and registration buttons in accordance with the initial design – it achieved what was intended.

LIMITATIONS

Limitation	How The Program Could Be Developed To Deal With The Limitations
The amount of lines of code which were used. For instance, the continuous 'Label(text="").pack()' lines were repeated too much, and shorter code could have been implemented to have the same functionality.	Perhaps a for loop which added each user access level button with a space above it could have been more efficient and saved memory.

PATIENT LOGIN CODE

STRENGTHS

When creating a successful login page for the user, several subroutines were used and incorporated together. The benefits of doing this was that it enabled each module to be devised independently. This meant that coding became easier as there was a short task to do within a short time frame – much like a 'sprint' often occurring in the Rapid Application Development systems analysis method.

LIMITATIONS

Limitation	How The Program Could Be Developed To Deal With The Limitations
It may have been more efficient, saved coding time, and memory space if the header and record data items were collected in the 'nhsIdEntry' subroutine. This is because the 'nhsIdEntry' subroutine involved opening the file and extracting data, but this same process was repeated in the 'outputRecord' subroutine.	It would be more efficient to define the variables within one opened file, as opposed to opening the same file twice.
A final limitation is regarding the deletion of the 'Patient Not Found' screen. The 'nhsIdNotFound' subroutine points to the 'deleteUserNotFoundScreen' subroutine.	It may not have been necessary if one line of code could have been written within the command line `Button(userNotFoundScreen, text="OK", command=deleteUserNotFoundScreen).pack() #` with the same functionality as the subroutine made (i.e. the code deleted the screen). Yet again, this may have been more efficient, saved coding time, as well as memory space.

REGISTRATION CODE

STRENGTHS

The registration code effectively replicated the screen design made earlier. Additionally, the error messages which occurred when a user attempted to use a pre-existing username was also efficient as it ensured that each user on the system had unique credentials, making identifying users on the system easier.

LIMITATIONS

Limitation	How The Program Could Be Developed To Deal With The Limitations

The code for gridding each label is on a separate line, an example is below:
`usernameEntry = tk.Entry(register_screen, textvariable=username)`
`usernameEntry.grid(row = 0, column = 1) #sets position of entry !`
This is problematic as it increases the number of lines which require memory, once again increasing the amount of storage required to store the python file.
There is also a line which is constantly repeated throughout the program, and can be seen below:
`Label(register_screen, text="                                    ").grid(row = 2, column = 1)`
The purpose of this line is to overwrite previous error messages with blank text. However, this is problematic as this may increase the file size, and in turn storage required by the user.
	Therefore, the code should be updated in a way in which allows new labels to be added, which automatically overwrite previous messages.

Waste of memory space in the register validation code.
`containsSpaces = usernameInfo.isspace() #checks if username contains spaces`
`containsSpaces1 = passwordInfo.isspace() #checks if password contains spaces`
`if usernameInfo == "" or containsSpaces == True or passwordInfo == "" or containsSpaces1 == True:`
For example, the variables 'containsSpaces' and 'containsSpaces1' were added to determine if the username and password inputs contained spaces, where this is then referenced in the selection statement above.
	However, two lines of code could be saved if, instead of 'containsSpaces == True', 'usernameInfo.isspace() == True' was used instead.

DELETING MESSAGE SCREEN

LIMIATIONS

Limitation	How The Program Could Be Developed To Deal With The Limitations

Whether a user has entered a successful, or unsuccessful login screen, they are presented with an 'OK' button which deletes the message. At the moment, this takes the form of two separate subroutines, which can be seen below:
`def deleteLoginSuccess():`
`    loginSuccessScreen.destroy()`
`def deleteUserNotFoundScreen():`
`  userNotFoundScreen.destroy()`
	Because they both perform the same functionality, it would be better if only one subroutine was made, which would take the screen as a parameter and delete it.
	Even if this was made into one subroutine, perhaps it would still be better if the command which calls the subroutines was built into the command. What this means is that instead of the code pointing to a subroutine which deletes the subroutine, perhaps the code which deleted the subroutine could be written into the command button instead, once again freeing up memory.

LOGIN CODE

STRENGTHS

The login code was effective in providing a login screen to a user. An additional good feature was the fact that the password input was displayed to a user in the form of asterisk. This is important as it means that anyone looking over the shoulder of a user entering their credentials is unable to determine what the password is, increasing the security of the system.

The use of the 'lambda' function was also beneficial in allowing the access level to be stored, which would then be beneficial in creating the separate main menus.

LIMITATIONS

Limitation	How The Program Could Be Developed To Deal With The Limitations
When the login subroutine was called, following the user pressing the button of an access level, the command passed in the access level as an argument. This access level was once again passed as an argument into the loginVerify subroutine. The access level variable is only used during the main menu subroutine. This wasted code, and in turn memory.	Continuously passing the access level was unnecessary, so this could be removed by using the 'accessLevel' variable once after defining it as global.

MAIN MENU

STRENGTHS

The access level chosen by the user was able to be used in this subroutine. As a result, the 'Scatter Diagram' option was not displayed to users who chose the 'Nurse' access level, as intended.

A further strength is the fact that once each option was clicked, the correct corresponding function would be performed. For instance, if a user clicked the 'Access Entire Database' option, they would be displayed the entire database.

LIMITATIONS

Limitation	How The Program Could Be Developed To Deal With The Limitations
The continuous use of the 'pack' lines may have taken up additional memory, which is problematic as the system should only take up the necessary amount of memory.	The pack lines could be removed and replaced with code which can create another button with a space above it.

ACCESS ENTIRE DATABASE

STRENGTHS

Much like with the access patient record option, a strength of the access entire database function is that it displays the entire contents of the database, as intended. A further strength is the fact that the header line is in bold. This is important as it indicates the field names to user.

LIMITATIONS

Limitation	How The Program Could Be Developed To Deal With The Limitations

One limitation that may have occurred is the way in which the allergies field value is presented. When deriving the scatter diagram co-ordinates, the elements of each record is taken. This is determined upon each occurrence of a comma. As a result, if more than one allergy is entered, the commas are turned into '|' in order to prevent the allergy commas from being interpreted as separate elements. As a result, the way in which the allergies values are stored in the database may not be visually pleasing to a user, which could be a limitation.	When displaying each allergies value for the patient record, the '|' symbols should be replaced with commas, through the use of an if statement.
Another limitation is that the code does not account for new entries if a user has entered a new patient record and attempts to display the entire database.	The code should be amended to account for the newest record added, which should be included in the output.

Allergies
N/A
Nuts
Pollen|Water|Seafood
N/A
Rasperries|Jam
N/A
Apples|Chocolates|Strawberries
N/A
Peanut
Peanut
N/A
N/A
Pollen|Apples
N/A
Fish|Nuts|Sesame Seeds|Kiwi|Cheri

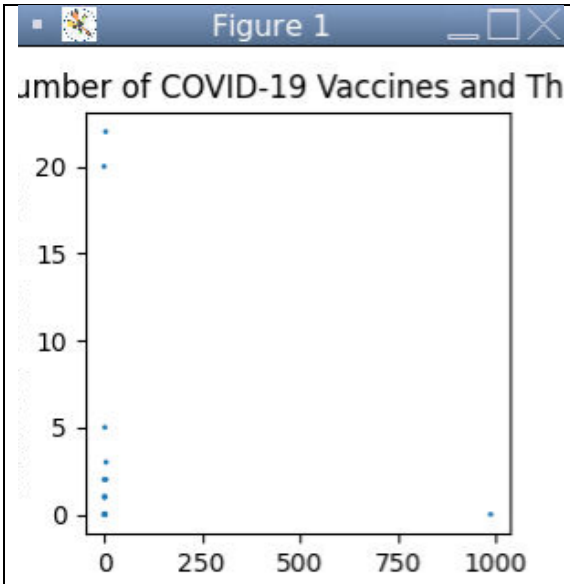
SCATTER DIAGRAM

STRENGTHS

The scatter diagram was effectively created, demonstrating the relationship between the number of COVID-19 vaccines, and cases. By doing so, doctors and scientists are able to effectively identify patterns and trends

LIMITATIONS

Limitation	How The Program Could Be Developed To Deal With The Limitations
The title of the scatter diagram is 'Figure 1'.	Code should be written which enables the title of the diagram to be changed to something more meaningful.

 When attempting to change this, it wasn't possible. This is a limitation as whilst the diagram is a figure, I don't think that it's the most appropriate name choice.	
It does not account for new entries. If a user has entered a new patient record and attempts to display the entire database, the new record is not accounted for in the diagram.	The code should be amended to account for the newest record added, which should be included in the final diagram.
A further limitation regards the size of the plot values. For instance, if a co-ordinate is repeated then the size of the plot increases. However, this is only done by a small amount. This is a limitation as it may be difficult for someone to identify patterns easily. As a result, the use of the scatter diagram may not be as high.	The implementation of a colour bar would be effective for users. This is because changes in colour would allow users to see how common a certain co-ordinate is more easily.

ENTER PATIENT DETAILS

STRENGTHS

I was able to effectively create a screen which allowed users to enter patient details.

Additionally, by clearing the user entry boxes each time a successful record was entered, time was saved in interacting with the system as users did not have to manually delete the contents of each box. Furthermore, the fact that the entry boxes are only cleared after the successful entry of a record is advantageous as it means that if a user makes any errors, they are able to rectify the entry box, rather than its entire contents being deleted, as they may forget certain patient details and it may take a long time to retrieve it.

LIMITATIONS

Limitation	How The Program Could Be Developed To Deal With The Limitations
An obvious issue is the amount of code which is used to display the entry boxes. As explained in the development section, the same 5 lines are repeated for each new field name.	It would be better if a for loop was incorporated instead as it would save the lines of code which are taking up a large amount of memory.
An additional error is in relation to the comprehension of the labels. For instance, the allergies label may be misled by users. A user may enter the number of allergies a patient has, as opposed to the allergies themselves specifically. Whilst any number other than 0 would return an error, the lack of clarity regarding what should be inputted for the 'allergies' value is a limitation as it may result in confusion, making the program more difficult to interact with.	The label should be changed so that it states that the allergy name should be entered. As a result, the user may not face a misconception where they think that the allergy number should be inputted. Therefore, this modification may improve the clarity of the entry label.
A limitation of the code which validates the allergies input is that it creates a list of numbers and then goes through each character of the user input and checks if it has a number. This is a limitation as the number of lines used takes up a lot of memory.	It would be better if there was one line of code which could determine if a string has an integer in it.
As explained earlier, a limitation of the ethnicities list presented to the user is that it is arguably limited. This is problematic as it could limit the user's ability to interact with the system if they are unable to find an ethnicity which best describes a patient.	The ethnicities displayed should be increased. For example, if a user enters the ethnicity number 'Other' then there could be the option to add this new ethnicity onto the 'List of Ethnicities' file, therefore allowing the ethnicity list to be more inclusive. Whilst this would be a good addition, and one that I was going to include, there is a problem. People may spell a new ethnicity in many different ways. As a result, it may be better if a user who wants to add an ethnicity spoke to an admin who would then research and ensure that the most universal spelling for the ethnicity is added. As a result, this would allow multiple ethnicities to be added which can be understood globally, thus extending the use of the system.
A final limitation is regarding the forename, surname, and address inputs. The only form of validation for these inputs is ensuring that their length is above two. However, the validation does not account for the use of special characters, as well as spaces, in the patient's input. This is problematic as a user may enter an unrealistic value into the database, compromising the validity of the database's contents. As a result, this makes relying upon the database	The validation code for these inputs should be increased so that special characters are not accepted as valid inputs.

contents difficult due to the doubt over the authenticity of the data. This problem also persists with other inputs.	

ACCESSING MULTIPLE VALUES

STRENGTHS

A code was created with the ability to successfully replicate the 'SELECT FROM WHERE' SQL statement. This is advantageous as it increases the extent to which the system is able to have the same functionality as a real life database. By replicating this, the usability of the program can increase as it may be more learnable

LIMITATIONS

Limitation	How The Program Could Be Developed To Deal With The Limitations
The system arguably has limited functionality.	For instance, a user should be able to select names (as opposed to a whole record) where the allergies are equal to 0, for example. Without functionality like this, the application and utility of the system may decrease. Additionally, Boolean operators such as 'AND' could be implemented in order to allow multiple conditions to be met before a record is outputted, for example. This would overcome the problem whereby the functionality of this feature is arguably limited.

The final limitation is regarding the length of the code. For instance, the comparisonValue is defined within each variable, an example can be seen below:
`if fieldChoice1 == "Forename":`
`  comparisonValue = 1 #sets element to be checked`
`elif fieldChoice1 == "Surname":`
`  comparisonValue = 2 #sets element to be checked`
`elif fieldChoice1 == "Date of Birth":`
`  comparisonValue = 3 #sets element to be checked`
`  inputValue1 = time.strptime(inputValue, "%d/%m/%Y")`
`elif fieldChoice1 == "Gender":`
`  comparisonValue = 4 #sets element to be checked`
`elif fieldChoice1 == "Ethnicity Number":`
`  comparisonValue = 5 #sets element to be checked`
`  inputValue1 = inputValue2.strip() #stores input value`
`elif fieldChoice1 == "Allergies":`
`  comparisonValue = 6 #sets element to be checked`
`  inputValue1 = inputValue2 #stores input value`
`elif fieldChoice1 == "Number of COVID-19 Vaccines":`
`  comparisonValue = 7 #sets element to be checked`
`elif fieldChoice1 == "Number of COVID-19 Cases":`
`  comparisonValue = 8 #sets element to be checked`
`elif fieldChoice1 == "Town/City":`
`  comparisonValue = 11 #sets element to be checked`
`elif fieldChoice1 == "Postcode":`
`  comparisonValue = 12 #sets element to be checked`
However, this wastes memory.
	In order to overcome this, perhaps the comparisonValues themselves should be passed as parameters into the subroutine.

Overall, there may be improved functionality within the code.

A recurring limitation, however, is the lines of code used. By having unnecessarily long lines of code, lots of memory is taken up. As a result, those attempting to use the system must be willing to dedicate a lot of memory for this one application. A memory intensive system may be difficult for users to implement, considering the fact that backups will need to be stored.

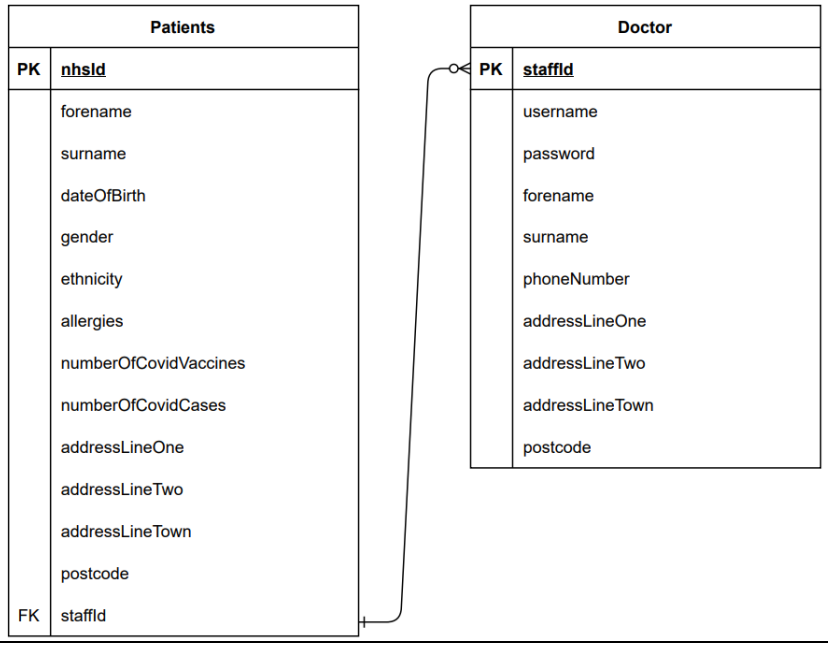
Having said this, shortening the code too much may make it more difficult to understand by other coders. For instance, recursion is a common concept used in programming, but it can often be hard to understand. This could make it difficult for programmers wanting to amend the code and solve errors.

LIMITATIONS OF SOLUTION

Aside from limitations to the code, which may be addressed in future maintenance, there are also limitations to the overall solution:

Limitation	How The Program Could Be Developed To Deal With The Limitations
The inability for users to amend the state of records. For instance, a patient's allergies may increase. As a result, it is important that these changing allergies are reflected in the database.	The way in which updating a record would work is that a patient's NHS ID will need to be entered, and then a screen may be displayed with options allowing a user to select which field value needs updating. This is important as it will ensure that data kept about each patient is in its most updated form, increasing the reliability of the data stored. However, this may result in a problem occurring, where multiple users attempt to update the same record. This could be solved by having a timestamp. The operation with the earlier timestamp should be applied first. However, the user whose amendments have not been applied should also be given a message which indicates to them that their change was not applied. Following this, the second user can re-try amending the record.
Lack of field values. As mentioned before, the lack of field values, such as the patient's medical history has not been incorporated.	Field values should be added to the header line of the CSV file. Following this, the enter patient details option should be amended to account for the new patient values which are required. By adding this feature, there can be greater insights made from the data seen by users.
Potential security threat as only a username and password are required.	A form of two factor authentication should be incorporated which allows users to have to answer a second question once they've logged in. The question could be personal, such as the name of a user's first school.
Anyone with login credentials is able to see the entire contents of the database, which poses a security threat.	In further development, the login details file should be amended. There should be two login details files – one for the doctors/nurses, and the other for the administrators/scientists/government officials. The administrators/scientists/government officials will have a login

which gives them access to the entire contents of the database. On the other hand. The doctors/nurses will have a file which consists of a staff ID (primary key), their username, password, name and more. When checking to see if the credentials are correct, the elements which store the username and password details will be checked. Following this, in the patients file, an additional field will be added which stores the doctor ID. This means that the staff ID will be a foreign key in the patients file. As a result, when a doctor logs in and selects the access entire database option, they would only be able to see the records for the patients that they have a corresponding foreign key for in the patients file. This prevents doctors from seeing sensitive data about other patients, and also keeps the data they observe relevant to them, thus improving the overall security and applicability of the database. This will also result in a one to many relationship being formed, where one doctor has many patients:



Overall the features outlined in this section will be considered in relation to the future maintenance of the system.

CONCLUSION

Overall, the creation of the system underwent an extensive iterative development process, where the designs were constantly amended to account for new user options. After consulting with Dr. Hussain and the other stakeholders, the program may be effectively considered to be a hospital database. Additionally, there are other features (outlined in the future maintenance section) which may also be incorporated which will improve the overall use of the system, and are important features to be considered for future development and maintenance.

FINAL CODE

PATIENTS CSV FILE

The Patients csv file which is used to store and extract data can be seen below, where the initial line sets the field names:

Patients.csv ×

```
NHS ID, Forename, Surname, Date of Birth, Gender, Ethnicity, Allergies, Number of COVID-19 Vaccines, Number of COVID-19 Cases, Address Line One, Address Line Two, Town/City, Postcode
1762883873,Farida,Addo,25/08/2004,F,Black African,N/A,1,0,8 Pine Building,Scots Road,London,SE93EF
4813824874,Sholeye,Olajide,14/03/2004,M,White and Black African,Nuts,0,1,Flat 26 Woodrow Court,69 Camberwell Station Road,London,SE59AZ
8967463139,Habibullah,Beverley,28/04/2010,M,White and Black African,Pollen|Water|Seafood,0,2,Potato Road Number 10,Chocolate House,Iraq,GHT13
6388236637,Timmy,Tom,15/08/1555,M,British,N/A,3,1,Timmytomville,Tomicity Town,Tomdon,TIY6
9848197844,Susan,Sally,03/04/2005,F,Northern Irish,Rasperries|Jam,4,22,3 Walkers Street,Avenue Close,Manchester,SW5690L
3919196324,Lilly,Smith,01/04/2007,F,British,N/A,2,1,5 Edgar Street,Walls Road,London,SE25G3
3752656695,Susan,Rose,04/02/2005,F,British,Apples|Chocolates|Strawberries,5,3,39 Hammersmith Town,Ealing Lane,Birmingham,SW194JW
9554744818,Rafaela,Mendes-Da-Silva,31/12/2003,F,Black African,N/A,0,0,17 Wilshaw House,Creekside Road,London,SE83YY
1249766451,Ryan,Akay,13/11/2005,F,Black African,Peanut,0,20,3 Meadow Lane,Right Street,Chestershire,SWKPLR
7529882792,Fajmaela,Aldi,12/09/2003,F,Black Carribean,Peanut,2,5,18 Falter House,Creative Road,London,SE91JM
1914236296,Mockaroo,Tonic,03/04/2005,M,European,N/A,2,0,9 Nice Road,Ivy Lane,Ealing,SE0OI2
4854166997,Kayla,Smith,01/04/2006,F,Black African,N/A,2,0,28 Martins Court,Thompsons Road,London,SE25AY
5564528166,Taylor,Blue,01/03/1992,F,South American,Pollen|Apples,5,2,3 Cool House,Highway Lane,Birmingham,SE4BW3
8583335398,Sally,Lucy,02/03/2004,F,European,N/A,987,0,Flat 99,Castle Lane,Nottingham,SE789Q
3632233817,Humayra,Choudhury,04/01/2004,F,South Asian,Fish|Nuts|Sesame Seeds|Kiwi|Cherries|Jackfruit|Cats|Dust,1,1,23 Forest View,Henllys,Cwmbran,NP446ED
2893417490,Najib,Hussain,28/02/2004,M,Arab,N/A,2,1,Globe Academy,Harper Road,London,SE16AG
3893123045,Abdulsalam,Giwa,09/03/2004,M,Black African,N/A,0,1,419 Street,Nsuo House,Agbobloshie,GHANA12
```

ETHNICITIES TEXT FILE

The text file which stores the ethnicities values and displays the ethnicities can be seen below:

Ethnicities.txt ×

```
British
Northern Irish
European
South Asian
South East Asian
South West Asian
Arab
South American
Black African
Black Carribean
White and Black Carribean
White and Black African
White and Asian
Other
```

LOGIN DETAILS CSV FILE

The file which stores the login details can be seen below:

LoginDetails.csv ×

```
1  Admin1,SQL$%RLS
2  Hello,No
3  123,123
4  h,h
5  Hussain,Doctor
6  Addo,Farida
7  JulieS,jud3201
8  Humayra4,Oreo
```

MAIN CODE

The final main code can be seen below:

main.py ×

```
#importing libraries which will be referenced later
import time
from Patient import Patient #enables the patient class file to be linked to the main file
import matplotlib.pyplot as plt #enables the creation of the scattergraph
from datetime import datetime
from collections import Counter
import tkinter as tk
from tkinter import *
#from functools import partial
#import os
#import re
import os.path

global delimiter
delimiter = ","  #sets the delimiter in the patient file, which will be referenced later on

existenceOfFile = os.path.isfile("Patients.csv")  #determines if the patient file exists
if existenceOfFile == False:
    with open("Patients.csv", "w") as patientFile:
        string = "NHS ID, Forename, Surname, Age, Gender, Ethnicity, Allergies, Number of COVID-19 Vaccines, Number of COVID-19 Cases, Address Line One, Address Line Two, Town/City, Postcode"  #sets the first line of the patient file
        patientFile.write(string)  #writes the first line to file
else:
  global numberOfLines
  numberOfLines = 0  #setting the initial number of lines to 0
  with open("Patients.csv","r") as file:  #opens the patient file for reading
    for i in file:  #reads each line of the patient file
      numberOfLines += 1  #adds one to the variable each time there is an additional number of lines

def deleteUserNotFoundScreen():
  userNotFoundScreen.destroy()
```

```
# Designing popup for value not found
def userNotFound(value):
    global userNotFoundScreen
    userNotFoundScreen = Toplevel(loginScreen) #uses the same screen defined in the 'loginScreen' subroutine
    if value == 1: #displays message if login details are not valid
        title = "Invalid Entry" #sets title
        message = "Incorrect Username or Password" #sets output text
    else: #displays message if co-ordinates haven't been found
        title = "Error" #sets title
        if value == 0: #displays message if co-ordinates haven't been found
            message = "No Co-Ordinates Found" #sets output text
        else: #displays message if records haven't been found
            message = "No Records Found" #sets output text
    userNotFoundScreen.title(title) #sets title
    userNotFoundScreen.geometry("150x100") #sets dimensions
    Label(userNotFoundScreen, text=message).pack() #displays error message
    Button(userNotFoundScreen, text="OK", command=deleteUserNotFoundScreen).pack() #deletes the popup once the button is pressed
```

```
def plot(value): #creates scatter diagram
  if value == 1: #performs if statment if specific values are being plotted
    fieldChoice1 = fieldChoice #stores field choice
    operatorChoice1 = operatorChoice #stores operator choice
    inputValue1 = inputValue #stores input value
    if fieldChoice1 == "Forename":
      comparisonValue = 1 #sets element to be checked
    elif fieldChoice1 == "Surname":
      comparisonValue = 2 #sets element to be checked
    elif fieldChoice1 == "Date of Birth":
      comparisonValue = 3 #sets element to be checked
      inputValue1 = time.strptime(inputValue, "%d/%m/%Y")
    elif fieldChoice1 == "Gender":
      comparisonValue = 4 #sets element to be checked
    elif fieldChoice1 == "Ethnicity Number":
      comparisonValue = 5 #sets element to be checked
      inputValue1 = inputValue2.strip() #stores input value
    elif fieldChoice1 == "Allergies":
      comparisonValue = 6 #sets element to be checked
      if inputValue1 == '0':
        inputValue1 = "N/A" #replaces input value to whats stored in database
      else:
        inputValue1 = inputValue2 #stores input value
    elif fieldChoice1 == "Number of COVID-19 Vaccines":
      comparisonValue = 7 #sets element to be checked
    elif fieldChoice1 == "Number of COVID-19 Cases":
      comparisonValue = 8 #sets element to be checked
    elif fieldChoice1 == "Town/City":
      comparisonValue = 11 #sets element to be checked
    elif fieldChoice1 == "Postcode":
      comparisonValue = 12 #sets element to be checked
  plt.figure(figsize=(3,3)) #sets dimensions of scatter diagram
  column = 0
  noOfVaccines = [] #sets an empty array for the number of covid vaccines
  noOfCovidCases = [] #sets an empty array for the number of covid cases
  with open("Patients.csv") as file: #opens the patient file
    record = [line.split(delimiter) for line in file] #splits the record each
time a delimiter is encountered
```

```
87      for element in record: #reads each element in the record
88        column+=1
89        if column == 1:
90          continue #ignore header row
91        if value == 1:
92          element[comparisonValue] = element[comparisonValue].strip() #removes new
    line statement
93          if fieldChoice1 == "Date of Birth":
94            element[comparisonValue] = time.strptime(element[comparisonValue],
    "%d/%m/%Y") #makes date variable comparable
95          if operatorChoice1 == "=":
96            if element[comparisonValue] == inputValue1:
97              noOfVaccines.append(element[7]) #adds the element number of vaccines
    to the array
98              noOfCovidCases.append(element[8]) #adds the element number of covid
    cases to the array
99          elif operatorChoice1 == "!=":
100           if element[comparisonValue] != inputValue1:
101             noOfVaccines.append(element[7]) #adds the element number of vaccines
    to the array
102             noOfCovidCases.append(element[8]) #adds the element number of covid
    cases to the array
103         elif operatorChoice1 == "<":
104           if element[comparisonValue] < inputValue1:
105             noOfVaccines.append(element[7]) #adds the element number of vaccines
    to the array
106             noOfCovidCases.append(element[8]) #adds the element number of covid
    cases to the array
107         elif operatorChoice1 == "<=":
108           if element[comparisonValue] <= inputValue1:
109             noOfVaccines.append(element[7]) #adds the element number of vaccines
    to the array
110             noOfCovidCases.append(element[8]) #adds the element number of covid
    cases to the array
```

```
111         elif operatorChoice1 == ">":
112           if element[comparisonValue] > inputValue1:
113             noOfVaccines.append(element[7]) #adds the element number of vaccines to the array
114             noOfCovidCases.append(element[8]) #adds the element number of covid cases to the array
115         elif operatorChoice1 == ">=":
116           if element[comparisonValue] >= inputValue1:
117             noOfVaccines.append(element[7]) #adds the element number of vaccines to the array
118             noOfCovidCases.append(element[8]) #adds the element number of covid cases to the array
119       else:
120         noOfVaccines.append(element[7]) #adds the element number of vaccines to the array
121         noOfCovidCases.append(element[8]) #adds the element number of covid cases to the array
122   combinedLists = zip(noOfVaccines,noOfCovidCases)  #combines the two lists
123   convertToList = list(combinedLists) #converts variable to list
124   repetitions = Counter(convertToList) #determines how many times coordinates are repeated
125   combinedLists = list(repetitions.keys()) #returns the values in order of insertion
126   noOfVaccines = [] #creating a new array
127   noOfCovidCases = [] #creating a new array
128   for i in combinedLists: #goes through each pair
129     noOfVaccines.append(i[0]) #adds the value in the zeroth position to the number of vaccines
130     noOfCovidCases.append(i[1]) #adds the value in the zeroth position to the number of covid cases
131   if not noOfVaccines: #determines if list is empty
132     userNotFound(0) #points to subroutine
133     return #exits subroutine
134   repetitions = list(repetitions.values()) #determines the number of coordinates for each coordinate
135   x = [noOfVaccines]; y = [noOfCovidCases]; s = [repetitions] #sets scatter values

136   plt.scatter(x,y,s) #adds plots to diagram
137   plt.title("Relationship Between The Number of COVID-19 Vaccines and The Number of COVID-19 Cases") #setting title
138   plt.xlabel("Number of COVID-19 Vaccines") #setting x axis
139   plt.ylabel("Number of COVID-19 Cases") #setting y axis
140   plt.show() #displays scatter diagram
```

```
142 def mainScreen(): #creates mainscreen
143     global accessLevel
144     global mainScreen
145     mainScreen = Tk() #creates screen
146     mainScreen.geometry("300x250") #sets geometry of screen
147     mainScreen.title("Hospital Database") #sets screen name
148     Label(text="Select Access Level", width="300", height="2", font=("Calibri",
    13)).pack() #screen title
149     Label(text="").pack() #creates a space between labels
150     #creating buttons for user access level options
151     Button(text="Patient", height="2", width="30", command = patientLogin).pack()
    #creates button
152     Label(text="").pack() #creates a space between labels
153     Button(text="Nurse", height="2", width="30", command = lambda accessLevel =
    "Nurse":login(accessLevel)).pack() #creates button
154     Label(text="").pack()#creates a space between labels
155     Button(text="Doctor", height="2", width="30", command = lambda accessLevel =
    "Doctor":login(accessLevel)).pack() #creates button
156     Label(text="").pack()#creates a space between labels
157     Button(text="Scientist", height="2", width="30", command = lambda accessLevel =
    "Scientist":login(accessLevel)).pack() #creates button
158     Label(text="").pack()#creates a space between labels
159     Button(text="Government Official", height="2", width="30", command = lambda
    accessLevel = "Government Official":login(accessLevel)).pack() #creates button
160     Label(text="").pack()#creates a space between labels
161     Button(text="Register", height="2", width="30", command = register).pack()
    #displays register button
162     mainScreen.mainloop() #keeps screen open until user clicks 'X'
```

```
165 def mainMenu1(accessLevel): #creates first main menu
166     global mainMenu
167     mainMenu = Toplevel(mainScreen) #creates screen using the one defined in the
    subroutine 'mainScreen'
168     mainMenu.geometry("300x250") #sets dimesnison
169     mainMenu.title("Options") #sets title
170     Label(mainMenu, text="Select Function", width="300", height="2", font=
    ("Calibri", 13)).pack() #sets screen title
171     Label(mainMenu,text="").pack() #creates empty space between labels
172     #creating buttons for user options
173     Button(mainMenu,text="Enter Patient Details", height="2", width="30", command =
    patientDetails).pack() #creates button for function
174     Label(mainMenu,text="").pack() #creates empty space between labels
175     Button(mainMenu,text="Access Patient Record", height="2", width="30", command =
    patientLogin).pack() #creates button for function
176     Label(mainMenu,text="").pack() #creates empty space between labels
177     Button(mainMenu,text="Access Multiple Values", height="2", width="30", command =
     multipleValues).pack() #creates button for function
178     Label(mainMenu,text="").pack() #creates empty space between labels
179     Button(mainMenu,text="Access Entire Database", height="2", width="30",
    command=entireDatabase).pack() #creates button for function
180     Label(mainMenu,text="").pack() #creates empty space between labels
181     if accessLevel != "Nurse": #does not show this button if the access level is
    'Nurse'
182         Button(mainMenu,text="Scatter Diagram", height="2", width="30",
    command=lambda: plot(0)).pack() #creates button for function
```

```
185 def register(): #creates register screen
186     global register_screen
187     register_screen = Toplevel(mainScreen)
188     register_screen.title("Register")
189     register_screen.geometry("300x250")
190     global username
191     global password
192     global usernameEntry
193     global passwordEntry
194     username = StringVar()
195     password = StringVar()
196     usernameLabel = Label(register_screen, text="Username * ").grid(row = 0,
    column = 0, sticky = 'w') #sets title of user input
197     usernameEntry = tk.Entry(register_screen, textvariable=username) #enables
    user to enter data
198     usernameEntry.grid(row = 0, column = 1) #sets position of entry bar
199     passwordLabel = Label(register_screen, text="Password * ").grid(row = 1,
    column = 0, sticky = 'w')
200     passwordEntry = tk.Entry(register_screen, textvariable=password, show='*')
201     passwordEntry.grid(row = 1, column = 1)
202     Button(register_screen, text="Register", width=10, height=1,command =
    registerValidation).grid(row = 2, column = 0) #enables inputs to be validated
```

```
204 def registerValidation(): #validates register input
205     usernameInfo = username.get() #returns username input
206     passwordInfo = password.get() #returns password input
207     validDetails = True #initialises register inputs as true
208     with open("LoginDetails.csv") as file: #opens login details file
209         for line in file: #reads each line in the file
210             line = line.strip("\n").split(delimiter) #removes "\n" from each line and
    splits it so each element in the line can be accessed separately
211             if usernameInfo == line[0]: #checks if the username is in the line
212                 Label(register_screen, text="
    ").grid(row = 2, column = 1) #overwrites previous messages
213                 Label(register_screen, text="Username Taken", fg="red", font=("calibri",
    11)).grid(row = 2, column = 1) #outputs error message
214                 usernameEntry.delete(0, END) #clears username entry box for re-entry
215                 passwordEntry.delete(0, END) #clears password entry box for re-entry
216                 validDetails = False #sets details as invalid
217     containsSpaces = usernameInfo.isspace() #checks if username contains spaces
218     containsSpaces1 = passwordInfo.isspace() #checks if password contains spaces
219     if usernameInfo == "" or containsSpaces == True or passwordInfo == "" or
    containsSpaces1 == True: #checks if entry boxes are empty
220         Label(register_screen, text="
    ").grid(row = 2, column = 1)
221         invalidRegistrationOutput() #outputs error message
222         validDetails = False
223     if validDetails == True: #performs operations if the inputs are valid
224         with open("LoginDetails.csv", "a") as file: #opens login details file for
    appending
225             file.write("\n") #writes a new line to the file
226             file.write(usernameInfo + delimiter + passwordInfo) #writes the username
    and password to file
227             usernameEntry.delete(0, END)
228             passwordEntry.delete(0, END)
229             Label(register_screen, text="                          ").grid(row = 2, column
     = 1) #overwrites previous messages
230             Label(register_screen, text="Registration Success", fg="green", font=
    ("calibri", 11)).grid(row = 2, column = 1) #prints success message for user
```

```
232 def invalidRegistrationOutput(): #outputs error message
233     global userNotFoundScreen
234     userNotFoundScreen = Toplevel(register_screen)
235     userNotFoundScreen.title("Error")
236     userNotFoundScreen.geometry("150x100")
237     Label(userNotFoundScreen, text="Invalid Entry").pack() #outputs error message
    to user
238     Button(userNotFoundScreen, text="OK", command=deleteUserNotFoundScreen).pack()
     #closes screen when button is pressed
239
240
241 def login(accessLevel): #creates login screen
242   global loginScreen
243   loginScreen = Toplevel(mainScreen) #uses the same login screen defined in the
    mainScreen subroutine
244   loginScreen.title("Login") #sets title
245   loginScreen.geometry("300x250") #sets dimensions
246   global usernameVerify
247   global passwordVerify
248   usernameVerify = StringVar() #stores username input
249   passwordVerify = StringVar() #stores password input
250   global usernameLoginEntry
251   global passwordLoginEntry
252   Label(loginScreen, text="Username * ").grid(row = 0, column = 0, sticky = 'w')
    #sets username label
253   usernameLoginEntry = tk.Entry(loginScreen, textvariable=usernameVerify)
    #creates entry box
254   usernameLoginEntry.grid(row = 0, column = 1) #positions entry box
255   Label(loginScreen, text="Password * ").grid(row = 1, column = 0, sticky = 'w')
    #sets password label
256   passwordLoginEntry = tk.Entry(loginScreen, textvariable=passwordVerify, show=
    '*') #creates entry box and displays input as asterisk
257   passwordLoginEntry.grid(row = 1, column = 1) #sets entry box position
258   Button(loginScreen, text="Login", width=10, height=1, command = lambda
    accessLevel = accessLevel:loginVerify(accessLevel)).grid(row = 2, column = 0)
    #creates button which onece pressed, enables user inputs to be verified
```

```
260 def patientLogin(): #creates login screen for patients
261     global loginScreen
262     loginScreen = Toplevel(mainScreen) #uses the same screen defined in the
    mainscreen
263     loginScreen.title("Login") #sets title
264     loginScreen.geometry("300x250") #sets dimensions
265     Label(loginScreen, text="Enter the details below") #enables user to input NHS
    ID
266     global usernameVerify
267     usernameVerify = StringVar() #stores user input
268     global usernameLoginEntry
269     Label(loginScreen, text="NHS ID *").grid(row = 0, column = 0) #creates label
270     usernameLoginEntry = tk.Entry(loginScreen, textvariable=usernameVerify)
    #creates entry bod
271     usernameLoginEntry.grid(row = 0, column = 1) #positions entry box
272     Button(loginScreen, text="Login", width=9, height=1, command =
    nhsIdEntry).grid(row = 0, column= 2) #creates button which points to validation
273
274 def nhsIdEntry(): #checks whether NHS ID is valid
275   username1 = usernameVerify.get()
276   usernameLoginEntry.delete(0, END)
277   with open('Patients.csv') as file: #opens patient file for reading
278     found = False #sets nhsId found as false
279     for i in range(numberOfLines):
280       line = file.readline() #reads each line in the patient file
281       lineSplit = line.split(delimiter) #splits the line each time a delimited is
    identified
282       if lineSplit[0] == username1: #if the first item in the line is the NHS ID
283         global lineNumber
284         lineNumber = i #print the record in the patient file
285         found = True #changes found to true as nhsId has been found
286         outputRecord(lineNumber,2,1) #prints record
287         break
288   if found != True:
289     nhsIdNotFound() #displays error message
```

```
291 def outputRecord(c,a,b): #displays record
292   with open("Patients.csv") as file:
293     lines = file.readlines() #reads each line of the file
294     header = lines[0].strip("\n") #uses first line as header
295     if a == 2: #performs selection for outputting multiple records
296       record = lines[c].strip("\n") #finds record containing NHS ID
297     else:
298       recordsToBeDisplayed = [] #empty list for records to be displayed
299       recordsToBeDisplayed.append(lines[0].strip("\n").split(delimiter)) #adds
    fieldname line to list
300       for line in lines[1:]: #reads each line of the database
301         line = line.strip("\n")
302         line = line.split(delimiter)
303         fieldChoice1 = fieldChoice #stores field choice
304         operatorChoice1 = operatorChoice #stores operator choice
305         inputValue1 = inputValue #stores input value
306         if fieldChoice1 == "Forename":
307           comparisonValue = line[1] #sets element to be checked
308         elif fieldChoice1 == "Surname":
309           comparisonValue = line[2] #sets element to be checked
310         elif fieldChoice1 == "Date of Birth":
311           comparisonValue = line[3] #sets element to be checked
312           inputValue1 = time.strptime(inputValue, "%d/%m/%Y")
313         elif fieldChoice1 == "Gender":
314           comparisonValue = line[4] #sets element to be checked
315         elif fieldChoice1 == "Ethnicity Number":
316           comparisonValue = line[5] #sets element to be checked
317           inputValue1 = inputValue2.strip() #stores input value
318         elif fieldChoice1 == "Allergies":
319           comparisonValue = line[6] #sets element to be checked
320           if inputValue1 == '0':
321             inputValue1 = "N/A" #replaces input value to whats stored in database
322           else:
323             inputValue1 = inputValue2 #stores input value
```

```
        elif fieldChoice1 == "Number of COVID-19 Vaccines":
          comparisonValue = line[7] #sets element to be checked
        elif fieldChoice1 == "Number of COVID-19 Cases":
          comparisonValue = line[8] #sets element to be checked
        elif fieldChoice1 == "Town/City":
          comparisonValue = line[11] #sets element to be checked
        elif fieldChoice1 == "Postcode":
          comparisonValue = line[12] #sets element to be checked
        if fieldChoice1 == "Date of Birth":
          comparisonValue = time.strptime(comparisonValue, "%d/%m/%Y") #makes date variable comparable
        if operatorChoice1 == "=": #determines operator which compares data
          if comparisonValue == inputValue1: #compares values
            recordsToBeDisplayed.append(line) #adds record to records to be displayed
        elif operatorChoice1 == "!=": #determines operator which compares data
          if comparisonValue != inputValue1: #compares values
            recordsToBeDisplayed.append(line) #adds record to records to be displayed
        elif operatorChoice1 == "<": #determines operator which compares data
          if comparisonValue < inputValue1: #compares values
            recordsToBeDisplayed.append(line) #adds record to records to be displayed
        elif operatorChoice1 == "<=": #determines operator which compares data
          if comparisonValue <= inputValue1: #compares values
            recordsToBeDisplayed.append(line) #adds record to records to be displayed
        elif operatorChoice1 == ">": #determines operator which compares data
          if comparisonValue > inputValue1: #compares values
            recordsToBeDisplayed.append(line) #adds record to records to be displayed
        elif operatorChoice1 == ">=": #determines operator which compares data
          if comparisonValue >= inputValue1: #compares values
            recordsToBeDisplayed.append(line) #adds record to records to be displayed
```

```
352     header = header.split(delimiter) #separates line upon each instance of a
    delimiter
353 v   if a == 2: #performs selection for outputting multiple records
354       record = record.split(delimiter)  #separates line upon each instance of a
    delimiter
355       columns = 2 #sets record columns
356 v   else:
357       columns = len(recordsToBeDisplayed)  #determines number of columns
358 v     if columns == 1:
359         userNotFound(2) #produces error if no records were added
360         return #exits subroutine
361     screen = Tk() #creates a new screen
362     screen.geometry("300x250") #sets screen dimensions
363 v   if a == 2: #performs selection for outputting multiple records
364       screen.title("{0}, {1}".format(record[2],record[1])) #sets the title of the
    screen to the users surname and forename
365       outputValues = [(header),(record)] #sets output values
366 v   else:
367       screen.title("Records") #sets the title of the screen to 'Records'
368 v   for i in range(columns): #reads each record
369 v     for j in range(13): #reads each field
370 v       if i == 0:
371           addingValues = Entry(screen, width=26, fg='black',font=
    ('Calibri',9,'bold')) #sets display design
372 v       else:
373           addingValues = Entry(screen, width=26, fg='black',font=('Calibri',9))
    #sets display design
374         addingValues.grid(row=i, column=j) #positions each field + record
375 v       if a == 2:
376           addingValues.insert(END, outputValues[i][j]) #adds values to grid
377 v       else:
378           addingValues.insert(END, recordsToBeDisplayed[i][j]) #adds values to grid
```

```
381 def loginVerify(accessLevel): #verifies login details
382     username1 = usernameVerify.get() #gets username input
383     password1 = passwordVerify.get() #gets password input
384     usernameLoginEntry.delete(0, END) #clears username entry box
385     passwordLoginEntry.delete(0, END) #clears password entry box
386     found = False #states that the login has not been found
387     with open("LoginDetails.csv") as file: #opens file for reading
388         for line in file: #reads each line in the file
389             line = line.strip("\n").split(delimiter) #removes "\n" from line
390             if username1 == line[0] and password1 == line[1]: #checks if username and
    password inputs are in the file
391                 found = True #login credentials have been found
392                 break
393     if found == False:
394         userNotFound(1) #displays error message
395     else:
396         loginSuccess(accessLevel) #points to a success screen
```

```
398 def loginSuccess(accessLevel): #displays success message if login details are
    valid
399     global loginSuccessScreen
400     loginSuccessScreen = Toplevel(loginScreen) #uses the same screen as
    'loginScreen'
401     loginSuccessScreen.title("Success") #sets title
402     loginSuccessScreen.geometry("150x100") #sets dimensions
403     Label(loginSuccessScreen, text="Login Success").pack() #displays success message
404     Button(loginSuccessScreen, text="OK", command=deleteLoginSuccess).pack()
    #closes screen when button is pressed
405     mainMenu1(accessLevel)
406
407 def nhsIdNotFound(): #displays message if NHS ID is not valid
408     global userNotFoundScreen
409     userNotFoundScreen = Toplevel(loginScreen) #uses the same screen as
    'loginScreen'
410     userNotFoundScreen.title("Error") #sets title
411     userNotFoundScreen.geometry("150x100") #sets dimensions
412     Label(userNotFoundScreen, text="Patient Not Found").pack() #displays error
    message
413     Button(userNotFoundScreen, text="OK", command=deleteUserNotFoundScreen).pack()
     #closes screen when button is pressed
414
415
416
417 #deleting popups
418 def deleteLoginSuccess():
419     loginSuccessScreen.destroy()
```

```
422 def entireDatabase(): #displaying database
423     database = [] #creates an empty list
424     with open("Patients.csv") as file: #opens file for reading
425         lines = file.readlines() #reads all records in file
426     for record in lines: #goes through each record
427         record = record.strip("\n")
428         database.append(record.split(delimiter))#adds each record to the separate file
429     root = Tk() #creates screen
430     root.geometry("300x250") #sets dimensions
431     root.title("Database") #sets title
432     for i in range(numberOfLines): #loops for the amount of record in the file
433         for j in range(13): #loops for the number of fields in the file
434             if i == 0:
435                 addingValues = Entry(root, width=26, fg='black',font=('Arial',8,'bold'))
    #makes header bold
436             else:
437                 addingValues = Entry(root, width=26, fg='black',font=('Arial',9))
438             addingValues.grid(row=i, column=j) #determines position of values
439             addingValues.insert(END, database[i][j]) #adds values to screen
```

```
441 def multipleValues():
442     global screen
443     screen = tk.Tk() #creates screen
444     screen.geometry("300x250") #sets dimensions
445     screen.title("Access Multiple Values") #sets title
446     global inputEntry
447     global input
448     input = StringVar() #stores input value
449     inputEntry = tk.Entry(screen, textvariable=input)
450     inputEntry.grid(row = 3, column = 3, sticky = 'w')
451     selectLabel = Label(screen, text="SELECT:").grid(row = 1, column = 0, sticky =
    'w') #designs select label
452     fromLabel = Label(screen, text="FROM:").grid(row = 2, column = 0, sticky = 'w')
    #designs from label
453     databaseLabel = Label(screen, text="DATABASE").grid(row = 2, column = 1, sticky
    = 'w') #designs database label
454     whereLabel = Label(screen, text="WHERE:").grid(row = 3, column = 0, sticky =
    'w') #designs where label
455
456     global fieldNames
457     fieldNames = [
458             "Forename",
459             "Surname",
460             "Date of Birth",
461             "Gender",
462             "Ethnicity Number",
463             "Allergies",
464             "Number of COVID-19 Vaccines",
465             "Number of COVID-19 Cases",
466             "Town/City",
467             "Postcode"
468         ] #dropdown menu options
469
470     global fieldName
471     fieldName = tk.StringVar(screen) #sets datatype of menu text
472     fieldName.set("FieldName") #sets initial menu text
```

```
474   def selected(choice): #determines value chosen by user
475     operators = operators1 #stores operator vaue
476     global selectChoice
477     selectChoice = option.get() #stores option value
478     global fieldChoice
479     fieldChoice = fieldName.get() #stores fieldname choice
480     if fieldChoice == "Forename" or fieldChoice == "Surname" or fieldChoice ==
    "Gender" or fieldChoice == "Allergies" or fieldChoice == "Town/City" or
    fieldChoice == "Postcode":
481       operators = [
482       "=",
483       "!="
484     ] #dropdown menu options
485     if choice == "Ethnicity Number":
486       ethnicitiesList1() #displays ethnicites list if ethnicity number field
    name is chosen
487     global operatorChoice
488     operatorChoice = operator.get() #stores operator choice
489     (OptionMenu(screen , operator , *operators,command=selected)).grid(row = 3,
    column = 2, sticky = 'w') #replaces previous operator drop-down menu
490
491   global option
492   option = tk.StringVar(screen) #sets datatype of menu text
493   option.set("Options") #sets initial menu text
494
495   options = [
496   "Record",
497   "Scatter Diagram Co-Ordinates"
498   #dropdown menu options
499 ]
500   (OptionMenu(screen , option , *options,command=selected)).grid(row = 1, column
     = 1, sticky = 'w') #positions options menu button
501
502   (OptionMenu(screen , fieldName , *fieldNames, command = selected)).grid(row =
    3, column = 1, sticky = 'w') #positions fieldnames menu button
```

```
  operators1 = [
    "=",
    "!=",
    "<",
    "<=",
    ">",
    ">=",
  ] #sets menu options

  global operator
  operator = tk.StringVar(screen) #sets datatype of menu text
  operator.set("Operator") #sets initial menu text

  #creates dropdown menu
  (OptionMenu(screen , operator , *operators1 )).grid(row = 3, column = 2, sticky = 'w') #positions operators menu button


Button(screen,text="Apply",command=verifyEntry).grid(row=4,column=0,sticky='w') #validates user input
```

```
522 def verifyEntry(): #validates menu/entry values
523     global inputValue
524     inputValue = inputEntry.get() #stores user input
525     error = False #sets flag as false
526     try: #stores values
527         selectChoice == "Options"
528         fieldChoice == "FieldName"
529         operatorChoice == "Operator"
530     except NameError: #reports error if values have not been changed
531         message = "Menu Value Must Be Amended" #sets error message
532         error = True #sets error to true
533     else:
534         if selectChoice == "Options" or fieldChoice == "FieldName" or operatorChoice
     == "Operator": #checks menu values
535             message = "Menu Value Must Be Amended" #reports error if menu values have
    not been changed
536             error = True #sets error to true
537         if fieldChoice == "Forename" or fieldChoice == "Surname" or fieldChoice ==
    "Gender" or fieldChoice == "Allergies" or fieldChoice == "Town/City" or
    fieldChoice == "Postcode":
538             if operatorChoice != "=" and operatorChoice != "!=":
539                 message = "Operator Value Must Be Amended" #reports error if menu
    values have not been changed
540                 error = True #sets error to true
541         if inputValue == "": #checks if entry box is empty
542             message = "Invalid Entry" #sets error message
543             error = True #sets error to true
544     if error == True: #performs code if error is found
545         Label(screen, text="
    ").grid(row = 3, column =4) #overwrites previous messages
546         Label(screen, text=message,fg='red').grid(row = 3, column =4) #displays
    error message
547         return False #ends function
```

```
548   if fieldChoice == "Number of COVID-19 Cases" or fieldChoice == "Number of COVID-19 Vaccines" or fieldChoice == "Ethnicity Number":
549     try: #determines if number is integer if specific fieldChoice which requires an integer is chosen
550       int(inputValue)
551     except ValueError:
552       Label(screen, text="").grid(row = 3, column =4) #overwrites previous messages
553       Label(screen, text="Must Be A Whole Number",fg='red').grid(row = 3, column =4) #displays error message
554       return False #ends function
555     if fieldChoice == "Number of COVID-19 Cases" or fieldChoice == "Number of COVID-19 Vaccines":
556       if int(inputValue) < 0: #checks if input is less than 0, which would be invalid
557         Label(screen, text="").grid(row = 3, column =4) #overwrites previous messages
558         Label(screen, text="Entry Must Be Greater Than One",fg='red').grid(row = 3, column =4) #displays error message
559         return False #ends function
560     else:
561       if int(inputValue) <= 0 or int(inputValue) > numberOfLines-1: #checks if input is within range of list of ethnicites options
562         Label(screen, text="").grid(row = 3, column =4) #overwrites previous messages
563         Label(screen, text="Entry Must Be Between 1 and "+str(numberOfLines-1),fg='red').grid(row = 3, column =4) #displays error message
564         return False #ends function
565   global inputValue2
566   if fieldChoice == "Ethnicity Number": #performs selection if fieldChoice is ethnicity number
567     with open("Ethnicities.txt","r") as file: #opens ethnicities file
568       lines = file.readlines() #reads all lines of file
569       inputValue2 = lines[int(inputValue)-1] #determines input value (subtracted from 1 due to index starting from 0)
```

```
570     elif fieldChoice == "Allergies": #performs selection if fieldChoice is
    allergies
571         inputValue = inputValue.title() #sets format of input
572         inputValue2 = inputValue.replace(" ,","|").replace(", ","|").replace("
    ","") #converts input into format of the value which would be stored in the
    patients file
573     elif fieldChoice == "Date of Birth": #performs selection if fieldChoice is
    date of birth
574         try:
575             datetime.strptime(inputValue, '%d/%m/%Y') #determines if the date of
    birth given is in the format '%d/%m/%Y'
576         except ValueError or UnboundLocalError: #if the input is not in the correct
    format
577             Label(screen, text="
    ").grid(row = 3, column =4) #overwrites previous messages
578             Label(screen, text="Invalid Entry", fg="red").grid(row = 3, column =4)
    #displays error message
579             return False #ends function
580     if fieldChoice == "Forename" or fieldChoice == "Surname" or fieldChoice ==
    "Town/City": #performs selection dependent on fieldChoice
581         inputValue = inputValue.title() #sets format of input
582     elif fieldChoice == "Gender" or fieldChoice == "Postcode": #performs
    selection dependent on fieldChoice
583         inputValue = inputValue.upper() #sets format of input
584     Label(screen, text="
    ").grid(row = 3, column =4) #overwrites previous messages
585     if selectChoice == "Scatter Diagram Co-Ordinates":
586         plot(1) #points to subroutine if user wants to see plot values
587     else:
588         outputRecord(fieldChoice, operatorChoice, inputEntry)
589     fieldName.set("FieldName") #sets original menu text
590     operator.set("Operator") #sets original menu text
591     option.set("Option") #sets original menu text
592     inputEntry.delete(0, END) #clears entry box
```

```
595 def patientDetails(): #gathering patient detials
596     global patientDetails
597     patientDetails = Toplevel(mainScreen) #creates screen using screen from
    subroutine 'mainScreen'
598     patientDetails.geometry("300x250") #sets dimensions
599     patientDetails.title("Enter Patient Details") #sets title
600     global forename
601     forename = StringVar() #stores input
602     Label(patientDetails, text="Forename:").grid(row = 0, column = 0, sticky =
    'w') #creates label
603     forename = tk.Entry(patientDetails, textvariable=forename) #creates entry box
604     forename.grid(row = 0, column = 1) #sets position of entry box
605     global surname
606     surname = StringVar() #stores input
607     Label(patientDetails, text="Surname:").grid(row =1, column =0, sticky = 'w')
    #creates label
608     surname = tk.Entry(patientDetails, textvariable=surname) #creates entry box
609     surname.grid(row = 1, column = 1) #sets position of entry box
610     global dateOfBirth
611     dateOfBirth = StringVar() #stores input
612     Label(patientDetails, text="Date of Birth (DD/MM/YYYY):").grid(row =2, column
    = 0, sticky = 'w') #creates label
613     dateOfBirth = tk.Entry(patientDetails, textvariable=dateOfBirth) #creates
    entry box
614     dateOfBirth.grid(row = 2, column = 1) #sets position of entry box
615     global gender
616     gender = StringVar() #stores input
617     Label(patientDetails, text="Gender (M/F):").grid(row = 3, column = 0, sticky =
     'w') #creates label
618     gender = tk.Entry(patientDetails, textvariable=gender) #creates entry box
619     gender.grid(row =3, column = 1) #sets position of entry box
620     global ethnicity
621     ethnicity = StringVar() #stores input
622     Label(patientDetails, text="Enter Number Relating To Ethnicity:").grid(row =
    4, column = 0, sticky = 'w') #creates label
```

```
623   Button(patientDetails, text="List of Ethnicities", width="12", height="1", command = ethnicitiesList1).grid(row = 5, column = 0, sticky = 'w') #creates button to display ehtnicies to choose from
624   ethnicity = tk.Entry(patientDetails, textvariable=ethnicity) #creates entry box
625   ethnicity.grid(row = 4, column = 1) #sets position of entry box
626   global allergies
627   allergies = StringVar() #stores input
628   Label(patientDetails, text="Allergies (Enter 0 If N/A):").grid(row = 6, column = 0, sticky = 'w') #creates label
629   allergies = tk.Entry(patientDetails, textvariable=allergies) #creates entry box
630   allergies.grid(row =6, column = 1) #sets position of entry box
631   global numberOfCovidVaccines
632   numberOfCovidVaccines = StringVar() #stores input
633   Label(patientDetails, text="Number Of COVID-19 Vaccines:").grid(row = 7, column = 0, sticky = 'w') #creates label
634   numberOfCovidVaccines = tk.Entry(patientDetails, textvariable=numberOfCovidVaccines) #creates entry box
635   numberOfCovidVaccines.grid(row = 7, column = 1) #sets position of entry box
636   global numberOfCovidCases
637   numberOfCovidCases = StringVar() #stores input
638   Label(patientDetails, text="Number of COVID-19 Cases:").grid(row =8, column = 0, sticky = 'w') #creates label
639   numberOfCovidCases = tk.Entry(patientDetails, textvariable=numberOfCovidCases) #creates entry box
640   numberOfCovidCases.grid(row = 8, column = 1) #sets position of entry box
641   global addressLineOne
642   addressLineOne = StringVar() #stores input
643   Label(patientDetails, text="Address Line One:").grid(row = 9, column = 0, sticky = 'w') #creates label
644   addressLineOne = tk.Entry(patientDetails, textvariable=addressLineOne) #creates entry box
645   addressLineOne.grid(row = 9, column = 1) #sets position of entry box
```

```
646     global addressLineTwo
647     addressLineTwo = StringVar() #stores input
648     Label(patientDetails, text="Address Line Two:").grid(row = 10, column = 0,
    sticky = 'w') #creates label
649     addressLineTwo = tk.Entry(patientDetails, textvariable=addressLineTwo)
    #creates entry box
650     addressLineTwo.grid(row = 10, column = 1) #sets position of entry box
651     global addressLineTown
652     addressLineTown = StringVar() #stores input
653     Label(patientDetails, text="Town/City:").grid(row = 11, column = 0, sticky =
    'w') #creates label
654     addressLineTown = tk.Entry(patientDetails, textvariable=addressLineTown)
    #creates entry box
655     addressLineTown.grid(row = 11, column = 1) #sets position of entry box
656     global postcode
657     postcode = StringVar() #stores input
658     Label(patientDetails, text="Postcode:").grid(row = 12, column = 0, sticky =
    'w') #creates label
659     postcode = tk.Entry(patientDetails, textvariable=postcode) #creates entry box
660     postcode.grid(row = 12, column = 1) #sets position of entry box
661     buttonRefresh = Button(patientDetails, text="Done", width=10, height=1,
    command = detailsVerify).grid(row = 13, column = 0, sticky = 'w') #creates
    button which verifies details once pressed
```

```
663 def detailsVerify(): #validating patient details
664     validateForename = nameValidate(forename.get().title(),1) #stores results of
    input validation
665     validateSurname = nameValidate(surname.get().title(),2) #stores results of
    input validation
666     validateDateOfBirth = dateOfBirthValidate(dateOfBirth.get()) #stores results
    of input validation
667     validateGender = genderValidate(gender.get().capitalize()) #stores results of
    input validation
668     validateEthnicity = ethnicityValidate(ethnicity.get()) #stores results of
    input validation
669     validateAllergies = allergiesValidate(allergies.get().title()) #stores
    results of input validation
670     validateNumberOfCovidVaccines = numberValidate(numberOfCovidVaccines.get(),1)
    #stores results of input validation
671     validateNumberOfCovidCases = numberValidate(numberOfCovidCases.get(),2)
    #stores results of input validation
672     validateAddressLineOne = nameValidate(addressLineOne.get().title(),3) #stores
    results of input validation
673     validateAddressLineTwo = nameValidate(addressLineTwo.get().title(),4) #stores
    results of input validation
674     validateAddressLineTown = nameValidate(addressLineTown.get().title(),5)
    #stores results of input validation
675     validatePostcode = postcodeValidate(postcode.get().upper()) #stores results
    of input validation
676     if validateForename == False or validateSurname == False or
    validateDateOfBirth == False or validateGender == False or validateEthnicity ==
    False or validateAllergies == False or validateNumberOfCovidVaccines == False or
     validateNumberOfCovidCases == False or validateAddressLineOne == False or
    validateAddressLineTwo == False or validateAddressLineTown == False or
    validatePostcode == False: #if any of the inputs aren't valid
677         Label(patientDetails, text="                          ").grid(row = 13, column
     = 1) #overwrites prior messages
```

```
678     else:
679         record =
    [validateForename,validateSurname,validateDateOfBirth,validateGender,validateEth
    nicity,validateAllergies,validateNumberOfCovidVaccines,
    validateNumberOfCovidCases,validateAddressLineOne,validateAddressLineTwo,validat
    eAddressLineTown,validatePostcode] #creating record
680         Label(patientDetails, text="Record Added", fg="green", font=("calibri",
    10)).grid(row = 13, column = 1) #displaying success message
681         submitDetails(record) #passing details to patient class to sdd to csv file
682         forename.delete(0, END) #clears entry box
683         surname.delete(0, END) #clears entry box
684         dateOfBirth.delete(0, END) #clears entry box
685         gender.delete(0, END) #clears entry box
686         ethnicity.delete(0, END) #clears entry box
687         allergies.delete(0, END) #clears entry box
688         numberOfCovidVaccines.delete(0, END) #clears entry box
689         numberOfCovidCases.delete(0, END) #clears entry box
690         addressLineOne.delete(0, END) #clears entry box
691         addressLineTwo.delete(0, END) #clears entry box
692         addressLineTown.delete(0, END) #clears entry box
693         postcode.delete(0, END) #clears entry box
```

```
695 def nameValidate(variableBeingValidated,numberOfVariableBeingValidated):
    #validating the inputs: forename, surname, addressLineOne, addressLineTwo and
    addressLineTown
696   messsage = "Too Short" #error message
697   if len(variableBeingValidated) < 2: #while the variable being validated has
    less than two characters
698     if numberOfVariableBeingValidated == 1: #forename
699       Label(patientDetails, text=messsage, fg="red", font=("calibri",
    10)).grid(row = 0, column =3) #error message outputted
700     elif numberOfVariableBeingValidated == 2: #surname
701       Label(patientDetails, text=messsage, fg="red", font=("calibri",
    10)).grid(row = 1, column =3) #error message outputted
702     elif numberOfVariableBeingValidated == 3: #addressLineOne
703       Label(patientDetails, text=messsage, fg="red", font=("calibri",
    10)).grid(row = 9, column =3) #error message outputted
704     elif numberOfVariableBeingValidated == 4: #addressLineTwo
705       Label(patientDetails, text=messsage, fg="red", font=("calibri",
    10)).grid(row = 10, column =3) #error message outputted
706     elif numberOfVariableBeingValidated == 5: #Town/City name
707       Label(patientDetails, text=messsage, fg="red", font=("calibri",
    10)).grid(row = 11, column =3) #error message outputted
708     return False
709   if numberOfVariableBeingValidated == 1: #forename
710     Label(patientDetails, text="                          ").grid(row = 0, column =
     3) #overwrites previous messages
711   elif numberOfVariableBeingValidated == 2: #surname
712     Label(patientDetails, text="                          ").grid(row = 1, column =
     3) #overwrites previous messages
713   elif numberOfVariableBeingValidated == 3: #addressLineOne
714     Label(patientDetails, text="                          ").grid(row = 9, column =
     3) #overwrites previous messages
715   elif numberOfVariableBeingValidated == 4: #addressLineTwo
716     Label(patientDetails, text="                          ").grid(row = 10, column
    = 3) #overwrites previous messages
```

```
717     elif numberOfVariableBeingValidated == 5: #Town/City name
718         Label(patientDetails, text="                          ").grid(row = 11, column
    = 3) #overwrites previous messages
719         return variableBeingValidated #returns the value of the validated variable as
    an element in the record
720
721 def dateOfBirthValidate(dateOfBirth): #validating date of birth input
722     try:
723         datetime.strptime(dateOfBirth, '%d/%m/%Y') #determines if the date of birth
    given is in the format '%d/%m/%Y'
724     except ValueError or UnboundLocalError: #if the input is not in the correct
    format
725         Label(patientDetails, text="Invalid Entry", fg="red", font=("calibri",
    10)).grid(row = 2, column = 3) #displays and positions error message
726         return False #the input is not valid
727     Label(patientDetails, text="                          ").grid(row = 2, column =
    3) #overwrites previous message
728     return dateOfBirth #returns the value of the validated variable as an field
    in the record
```

```
730 def genderValidate(gender): #validate gender input
731     if gender != "M" and gender != "F": #while the gender is not equal to male or
    female
732         Label(patientDetails, text="Invalid Entry", fg="red", font=("calibri",
    10)).grid(row = 3, column = 3) #displays and positions error message
733         return False  #returns the value of the gender variable as an element in
    the record
734     Label(patientDetails, text="                          ").grid(row = 3, column =
    3) #overwrites previous message
735     return gender #returns the value of the validated variable as a field in the
    record
736
737 def allergiesValidate(allergies): #validate allergy input
738     numbers = ["1","2","3","4","5","6","7","8","9","0"] #creating a list to store
    numbers
```

```
737 def allergiesValidate(allergies): #validate allergy input
738     numbers = ["1","2","3","4","5","6","7","8","9","0"] #creating a list to store
    numbers
739     number = False #initialising number in allergy input as false
740     for element in allergies: #reads each element in the input
741         if element in numbers: #checks if the element is a number
742             number = True #returns true if allergy input contains a number
743             break
744     if allergies == "0": #if the patient has no allergies
745         Label(patientDetails, text="                          ").grid(row = 6, column =
     3) #overwrites previus error message
746         return "N/A" #the value in the allergy filed should be set to not applicable
747     elif number == True or allergies == "" or len(allergies) < 2: #checks if
    entry is blank, contains a number or has a length of less than 2
748         Label(patientDetails, text="Invalid Entry", fg="red", font=("calibri",
    10)).grid(row = 6, column = 3) #displays error message
749         return False #indicates that the value is invalid
750     allergies = allergies.replace(" ,","|").replace(", ","|").replace("  ","")
    #replacing delimiter so the allergy input takes up one element in the list
751     Label(patientDetails, text="                          ").grid(row = 6, column =
    3) #overwrites previus error message
752     return allergies #returns the value of the validated variable as a field in
    the record
```

```
754 def determiningEthnicities():
755     global validEthnicity
756     validEthnicity = [] #sets the ethnicity as an array
757     with open("Ethnicities.txt") as f: #opens the ethncities file
758         for line in f: #reads through each line in the ethnicities file
759             validEthnicity.append(line.replace("\n","")) #adds each line to the array
    declared earlier, and replaces the new line (at the end of each line) with
    empty string to remove prevent "\n" from being added to the array
760     global lengthEthnicity
761     lengthEthnicity = len(validEthnicity) #finds the length of the array
762     return lengthEthnicity, validEthnicity #returns two values for use in
    'ethnicityValidate' function
763
764 def ethnicitiesList1(): #displays list of ethnicities
765     global ethnicitiesList
766     ethnicitiesList = Toplevel(mainScreen) #creates screen using the one defined
    in 'mainScreen'
767     ethnicitiesList.geometry("300x250") #sets timensions
768     ethnicitiesList.title("List of Ethnicities") #sets title
769     determiningEthnicities()
770     for i in range(lengthEthnicity): #continues to iterate for the length of the
    the ethnicity list
771         Label(ethnicitiesList, text=(str(i+1)+". "+validEthnicity[i])).pack()
    #prints each line in validEthnicity, with a number and fulvaluesop before it
```

```
773 def ethnicityValidate(ethnicity): #validating ethnicity
774   if ethnicity == "":
775     Label(patientDetails, text="Invalid Entry", fg="red", font=("calibri",
    10)).grid(row = 4, column = 3) #displays error message if ethnicity is an empty
    string
776     return False
777   for character in ethnicity: #goes through each character in the ethnciity
    input
778     if character.isalpha() == True: #checks if character is an alphabetical
    character
779       Label(patientDetails, text="Invalid Entry", fg="red", font=("calibri",
    10)).grid(row = 4, column = 3) #displays error message
780       return False #indicates that the value is invalid
781   lastDigit = int(repr(float(ethnicity))[-1]) #determines the last digit of the
    float number
782   lengthEthnicity, validEthnicity = determiningEthnicities() #stores
    ethnicityLength and validated
783   if lastDigit != 0 or int(ethnicity) < 1 or int(ethnicity) > lengthEthnicity:
784     Label(patientDetails, text="Invalid Entry", fg="red", font=("calibri",
    10)).grid(row = 4, column = 3) #error message
785     return False #returns false if the input is not equal to one of the values
    specified in 'List of Ethnicities'
786   validatedEthnicity = validEthnicity[int(ethnicity)-1] #returns ethnicity value
787   Label(patientDetails, text="                          ").grid(row = 4, column =
    3) #overwrites previous error messages
788   return validatedEthnicity #returns the value of the validated variable as a
    field in the record
```

```
790 def postcodeValidate(postcode): #validate postcode
791   postcode = postcode.replace(" ","") #removes spaces in postcode
792   if len(postcode) < 2  or len(postcode) > 7: #continues the loop until the
    postcode has a standard length
793     Label(patientDetails, text="Invalid Length", fg="red", font=("calibri",
    10)).grid(row = 12, column = 3) #displays error message
794     return False #indicates that the value is invalid
795   for x in range (len(postcode)): #goes through each value in the postcode
796     if x == 0: #if the value is the first value
797       isAlpha = postcode[x].isalpha() #determines if the first value is in the
    alphabet
798       if isAlpha == False: #loops until the first value of the postcode is in
    the alpahbet
799         Label(patientDetails, text="Invalid Format", fg="red", font=("calibri",
    10)).grid(row = 12, column = 3) #displays error message
800         return False #indicates that the value is invalid
801   Label(patientDetails, text="                          ").grid(row = 12, column =
    3)
802   return postcode #returns the value of the validated variable as a field in
    the record
```

```
804 def numberValidate(variableBeingValidated,numberOfVariableBeingValidated):
    #validating the inputs: numberOfCovidVaccines and numberOfCovidCases
805   string = False #sets string occurences as false
806   if variableBeingValidated == "":
807     if numberOfVariableBeingValidated == 1: #numberOfCovidVaccines
808       Label(patientDetails, text="Invalid Entry", fg="red",font= ("calibri",
    10)).grid(row = 7, column = 3) #displays error message
809     else: #numberOfCovidVaccines
810       Label(patientDetails, text="Invalid Entry", fg="red", font=("calibri",
    10)).grid(row = 8, column = 3) #displays error message
811     return False #returns false if input is empty
812   for index in range (len(variableBeingValidated)):
813     if variableBeingValidated[index].isalpha() == True:
814       string = True #sets string equal to true if input contains an
    alphabetical character
815       if numberOfVariableBeingValidated == 1: #numberOfCovidVaccines
816         Label(patientDetails, text="Invalid Entry", fg="red",font= ("calibri",
    10)).grid(row = 7, column = 3) #displays error message
817       else: #numberOfCovidCases
818         Label(patientDetails, text="Invalid Entry", fg="red", font=("calibri",
    10)).grid(row = 8, column = 3) #displays error message
819       return False #returns false if the input contains a string
820   lastDigit = int(repr(float(variableBeingValidated))[-1]) #provides gives the
    last digit of the variable "validatedNumber"
821   if lastDigit != 0 or int(variableBeingValidated) < 0: #checks if input is a
    float or greater than 1
822     if numberOfVariableBeingValidated == 1: #numberOfCovidVaccines
823       Label(patientDetails, text="Invalid Entry", fg="red", font=("calibri",
    10)).grid(row = 7, column = 3) #displays error message
824     else: #numberOfCovidCases
825       Label(patientDetails, text="Invalid Entry", fg="red", font=("calibri",
    10)).grid(row = 8, column = 3) #displays error message
826     return False #indicates that the value is invalid
```

```
827     if numberOfVariableBeingValidated == 1:#numberOfCovidVaccines
828         Label(patientDetails, text="                          ").grid(row = 7, column =
    3) #overwrites previous error message
829     else: #numberOfCovidCases
830         Label(patientDetails, text="                          ").grid(row = 8, column
    = 3) #overwrites previous error message
831     return variableBeingValidated #returns the value of the validated variable as
    a field in the record
832
833 def submitDetails(detailsVerify): #enables user to submit the details of a new
    patient
834     record = detailsVerify #jumps to another subroutine in order to collect
    inputs which will form the record
835     patient = Patient(record) #provides the record to the Patient class, which
    will then combine the NHS ID with the other elements in the record
836     patient.saveRecord() #jumps to a subroutine in the Patient class where the
    record is written to file
837
838 mainScreen() #starts program
```

PATIENT CLASS CODE

The code which creates the record can be seen below:

Patient.py ×

```
1  #importing library which can be referenced later
2  import random
3
4  class Patient: #creates a class for each patient
5      def __init__(self, record): #creates a record where each field is derived from the
       record variable decleared in the main program
6          self.forename = record[0] #stores forename input
7          self.surname = record[1] #stores surename input
8          self.dateOfBirth = record[2] #stores date of birth input
9          self.gender = record[3] #stores gender input
10         self.ethnicity = record[4] #stores ethnicity input
11         self.allergies = record[5] #stores allergies input
12         self.numberOfCovidVaccines = record[6] #stores number of COVID-19 vaccines input
13         self.numberOfCovidCases = record[7] #stores number of COVID-19 cases input
14         self.addressLineOne = record[8] #stores address line one input
15         self.addressLineTwo = record[9] #stores address line two input
16         self.addressLineTown = record[10] #stores town/city name input
17         self.postcode = record[11] #stores postcode input
```

```
20    #generates unique 10 digit ID
21    def generateNHSId(self):
22      nhsId = [] #creates an empty array for the NHS ID
23      for x in range (0,10): #generates a 10 digit number
24        number_generator = random.randint(0,9) #generates a random number from 0 to 9
25        nhsId.append(number_generator) #adds the value to the array
26      newNhsId = "" #creates an empty string
27      self.delimiter = "," #sets a delimiter
28      for element in nhsId: #goes through each element in the nhs id
29       newNhsId += str(element) #converts each element into a string as the value does
    not need to be treated as an integer
30      with open("Patients.csv") as file: #opens patient file
31        for line in file: #reads each line (record)
32          line = line.split(self.delimiter) #splits the line upon an occurence of a
    delimiter so each element can be treated as a separate value
33          if line[0] == newNhsId: #if the first element is equal to the NHS ID generated
34            self.generateNHSId() #generates new nhs id if it has been taken
35          else:
36            continue
37      return newNhsId #returns the nhs id as a field value for the record
```

```
40    #save record
41    def saveRecord(self):
42      self.nhsID = self.generateNHSId() #collects NHS ID from a separate subroutine
43      self.delimiter = "," #sets a delimiter
44      self.record = self.nhsID + self.delimiter + self.forename + self.delimiter +
    self.surname + self.delimiter + self.dateOfBirth + self.delimiter + self.gender +
    self.delimiter + self.ethnicity + self.delimiter + self.allergies + self.delimiter +
    self.numberOfCovidVaccines + self.delimiter + self.numberOfCovidCases + self.delimiter
     + self.addressLineOne +  self.delimiter + self.addressLineTwo + self.delimiter +
    self.addressLineTown + self.delimiter + self.postcode #adds each field to create a
    record
45      with open ("Patients.csv", "a") as patientFile: #opens the patient file for
    appending
46        patientFile.write("\n") #writes a new line to the file
47        patientFile.write(self.record) #writes the record to the file
```