

**Contents**

|  |  |
| --- | --- |
| Analysis………………………………………………………………………………………………………… | **page 2** |
| Design …………………………………………………………………………….…………………………… | **page 23** |
| Technical Solution …………………………………………………………….……………………….… | **page 53** |
| Testing ..…………………………………………………………………………………………………….… | **page 124** |
| Failed Tests ……………………………………………………………………………………………….… | **page 161** |
| Evaluation …………………………………………………………………………………………………… | **page 172** |
|  |  |
| **Appendices** |  |
| Prototype code …………………………………………………………………………………..………. | **page 179** |
| Gantt Chart …………………………………………………………………………….…………………… | **page 190** |
| Unexpected Problems ….……………………………………………………………………………… | **page 191** |
| References ………………………………………………………………..………………………………… | **Page 201** |

**ANALYSIS**

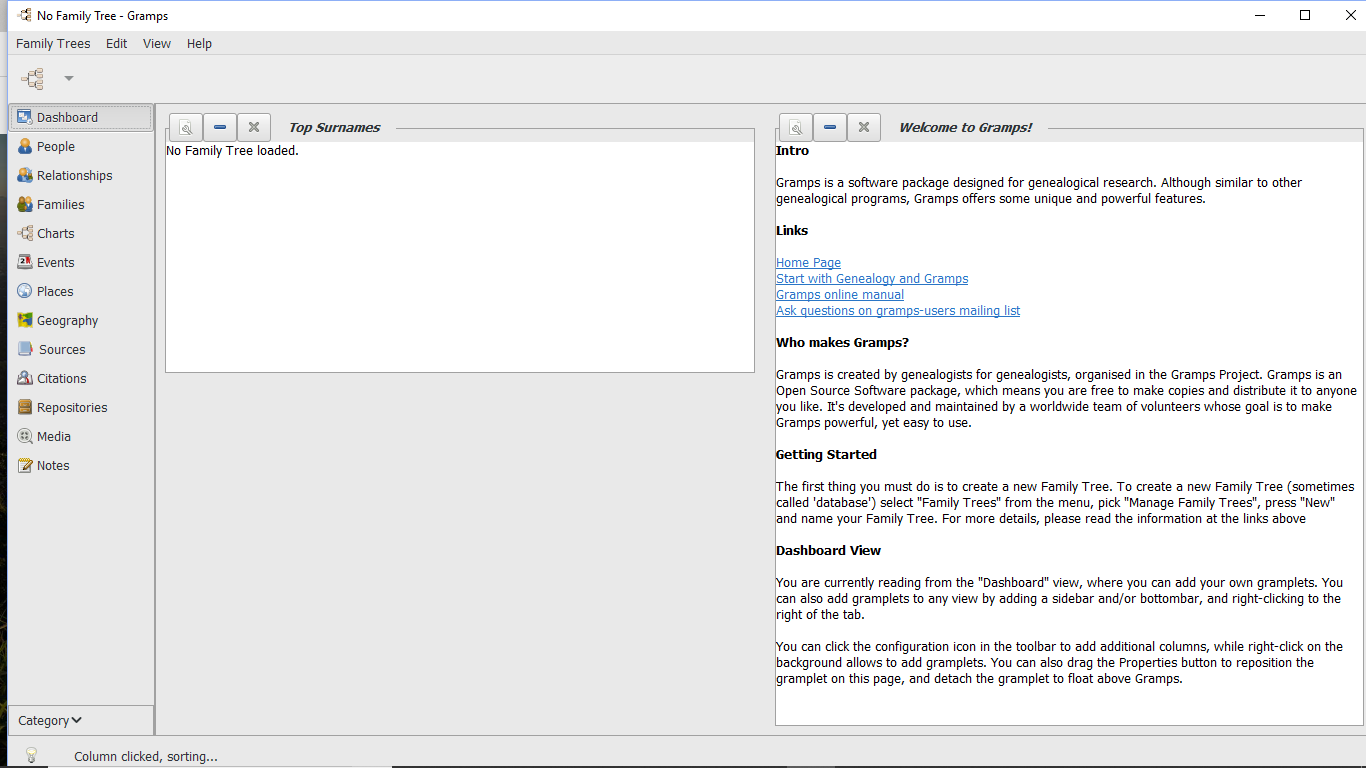
**Why I chose this project**

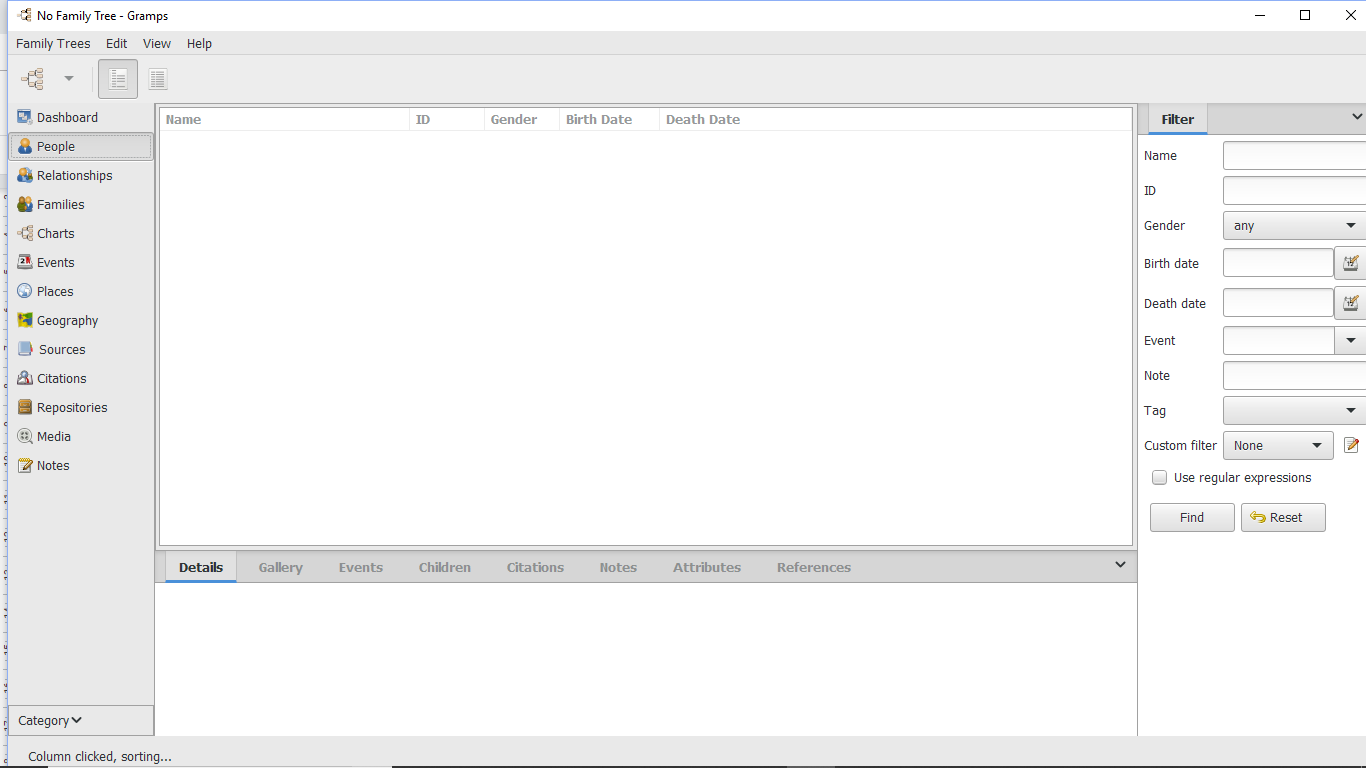
When it came to deciding what to do my project on, I was debating to do my project either on a family tree or on a simulation of the spread of an epidemic. I wanted to do them for different reasons. I wanted to do the family tree project, as I had always wanted to make a family tree and what had stopped me was the fact that using a program such as ‘Ancestry’ would cost around $19.99 a month: a huge commitment. Therefore, I thought that instead of using ‘Ancestry’ or another software, why not use my own? On the other hand, the idea of the epidemic was good as from research, it would involve a lot of calculus, especially differential equations. I got the idea to do this project from talks/lectures at universities, because at the time I was visiting different universities to decide where I would want to study mathematics and at most places there was a third year module called ‘mathematical biology’ that included looking at the spread of disease etc. and it looked quite interesting. But, I eventually decided on the family tree project because not only have I wanted to create a family tree for my family for a long time, but I felt that with this project I could really broaden my skillset, as I would learn to properly implement databases, interfaces and recursive algorithms to say the least, whereas in the other project on the outset it would seem that I would only be coding calculus. I also felt that the family tree project would have more end users.

**Overview**

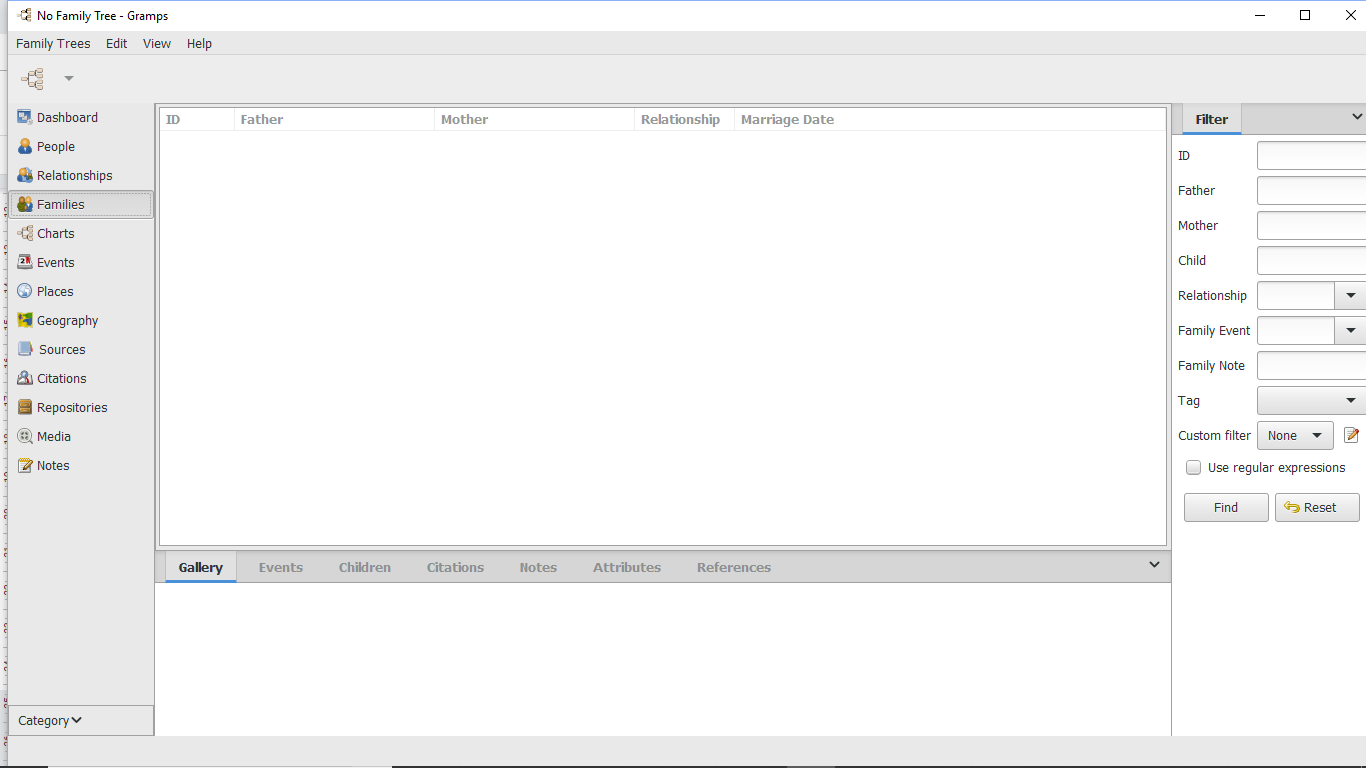
My project will allow any user to create a family tree, and I aim for it to be quite user-friendly. The first thing that will be displayed to the user is the main menu, where they can choose to add people into the database, edit a person already in the database, or delete a person from the database as well as an exit option where you can leave the program. The language I will use to code this will be Python. The information entered into the back-end SQL database will include, but not be limited to: First name, Surname, Date of birth (DOB), Date of death (DOD), Place of Birth (POB), Mother and Father. As an extension task, I could create another table that stores miscellaneous data such as photos, videos and a bio. After this, I will put the users into ‘families’ based on the information in the database. Each family consists of two parents and their children. Finally, before drawing the tree and displaying it to the user, I will need to create an algorithm that decides which families are on the same generation row, and within a generation row, which ones should be placed next to each other (i.e. sibling families which descend from a common parent family.) If someone clicks on a person in the tree, then more detailed information about that person is displayed. Other extensions include being able to enter any two people and have their relationship displayed or written out in words and, having other interesting information about the whole family displayed such as the mean age of the living family members.

**Looking at a program that creates family trees - Gramps**

Above is the dashboard. When you open Gramps, this where you are taken to first. Here, you can choose to open an existing file. On the right, descriptions about Gramps is given.

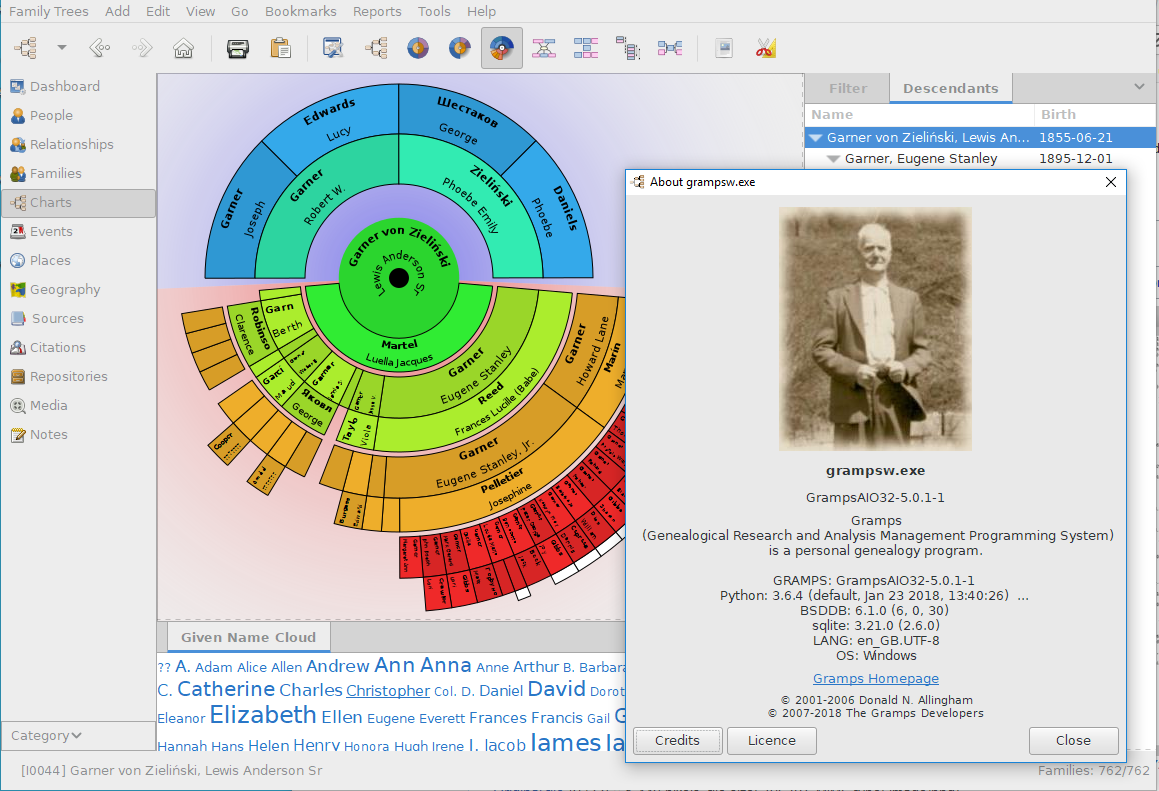


If you click on ‘People’ on the left, you can enter people into the database. Here, they only ask for name, ID, Gender, Birth Date and Death Date. These are entered on the side bar on the right.

If you click on Families, you are taken to this page, where you can enter a family, and the father and mother of that family.

The advantage of Gramps was the different options that it had. For example, the charts options allow you to display your family tree information in different charts including pedigree charts, as well as the standard family tree. You can also add important events, and can also display information on a world map by clicking the Geography option. Also, when entering dates of birth and death, you can choose from a range of different calendars including Gregorian, Islamic and Hebrew. The only disadvantage really is that I wish that the program would be more user-friendly, and explicitly say where to enter information.

But, looking at Gramps gave me the idea that adding a world map with dots representing where people were born or living, although may be too ambitious due to the constraints that have been placed, I have decided to add it as an extension objective.



The image above was one of Gramps’ options for displaying the family tree. This is known as a ‘fan chart’. The way it works, is that the person of interest is in the centre, and the next concentric circle is split into 2: one half for the mother and one half for the father. This process is then repeated, so that the person’s grandparents are above the parents. The reason I liked this representation was that not only was it colourful and unique, but I found it very intuitive and easy to see which people are in the same generation.

**Prototype**

Before creating a graphic user interface (GUI) for the section of the project where they are entering information into the database, I will create all the functions, but without an interface so that I can get an idea if I need to add any other options. For the section of the project where the tree is drawn, a prototype is not needed as I should just be able to test it as I code.

**Limitations**

* Because my project aims to be user-friendly and accessible by people with little coding experience, I will need to make a simple to use GUI.
* I will use simple colours with sufficient contrast to ensure that potential users with visual impairment can use the program more easily. However, if a family tree has a lot of members then it may be that a lot of information appears on screen and that text is smaller to accommodate. I will therefore implement a zoom function so that areas of the family tree can be enlarged.
* Since there is a lot of personal information entered into the program, the data will only be stored locally on a user’s own computer in this version. It could be that a later version would be cloud-based, however I would then need to make users aware of how their data was being used, according to GDPR laws and I would have to implement adequate security with a password.
* Programs using Python and Kivy can be packaged for various platforms such as Windows, iOS, Android etc, for example using pyinstaller. However, due to time constraints, I have decided that my family tree program will not be compiled or packaged in this version and end users will need to have Python 3.7.x and the Kivy module installed on their computer/device.
* Time constraints – Because my project relies on being able to query and insert information from the database, I need to make sure that the initial database section that I have described previously is completed soon, so that I can then move on code parts of project in parallel.

**Interviews and surveys**

**Interview #1 – Dad**

A couple of years ago, my dad attempted to make a family tree for our family.

Q1. What program did you use?  
**I used a program called Gramps**

Q2. Why did you use this program in particular?  
**It was open source, which meant it was free. It was simple to use and provided good charts for visualising the family tree such as a pedigree chart and a fan chart. In addition, it was easy to share reports as it supported various file formats such as PDF and OpenDocument.**

Q3. What information did you have to enter into the program when making the family tree?  
**First name, Surname, dates of birth, prefixes, gender and call name were the general ones. Additionally, I would enter the name given at birth (which is not necessarily the first name), date of death and relationships of each family member. I would also add photos and videos for some people.**

Q4. What do you think the program could have done better?  
**It could have been more user friendly. Sometimes it was a little confusing to navigate, but maybe this was because it had more features than I needed.**

**Interview #2 – Mr. Patterson**

Mr. Patterson is one of my computer science teachers, and he also recently made a family tree.

Q1. What software did you use to make your family tree or, did you do it by hand?

**I use RootsMagic 7 and I also have a copy on**[**Ancestry.co.uk**](https://www.ancestry.co.uk/)

2. What were the features that you used the most when using the software? Are there any that you wished that it had?

**I am happy with all the**[**features**](https://www.rootsmagic.com/RootsMagic/Features.aspx)**that the software has.  The features that I use most are adding core details including (birth, baptism, census, residence, occupation, marriages, children, death, burial and will data)**  
3. What were the pros and cons of the software?

**I did quite a bit of research before deciding on this software so there are few cons, however, with regards to producing trees it is far easier to produce an ancestry tree only showing direct ancestors whereas often I also want to show all the siblings as well.  Whilst this is possible, it is not the default setting.**

**The main pros are that I am able to customise most aspects, for example, add my own fact and source types.  I also like the built-in relationship calculator where you can put in two people and it tells you how they are related (e.g. 4 x great grand uncle).**

4. How important is it for you to have a program that is user-friendly and easy to navigate? Would you prefer to use a program with many features that is less user-friendly or a program with only the necessary information you are looking for and that is more user friendly.

**As I am a competent computer user, I like to have the flexibility to customise aspects so having many features is useful.  Although many features may not at first seem needed, they may be useful at a later date.  On the other hand, I can see why a beginner to family history or a less confident computer user would want to have only the basics in a user-friendly environment, although this may limit how they conduct their research.**

5. What extra information about your family would you find interesting to be displayed. E.g. mean age of people alive? Or would you not care much for the extra statistics?

**Extra statistics, whilst not essential would be interesting to read especially if they could be cross referenced against other data (for example mean age of those living in the 18th or 19th century or mean age of those living in a certain area of the country or doing a certain occupation).  Statistics would be interesting rather than essential.**

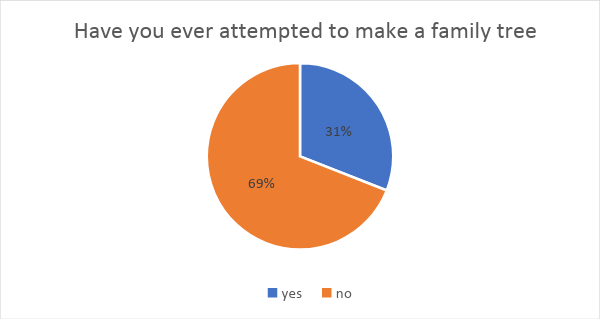
6. Instead of a typical family tree, I was thinking of making one where each 'family' is a unit, consisting of parents and their children. Each unit will then be connected to another unit, if both units contain the same person (connections most of the time if not all the time will be when the parent of one family is the child of another family). These units will be drawn instead of individual boxes for every person. Would you prefer this?

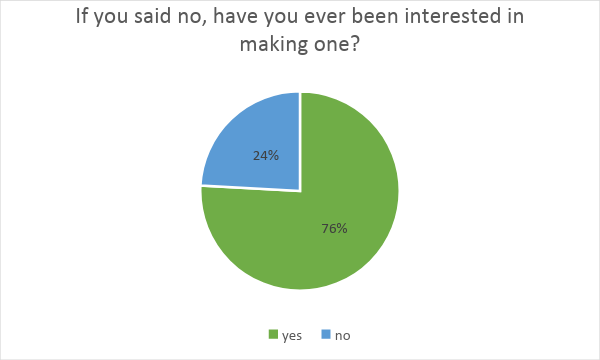
**It sounds interesting!  I would certainly like to see it!**

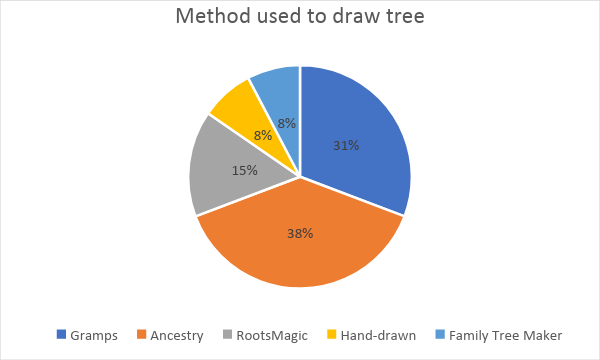
**Survey**

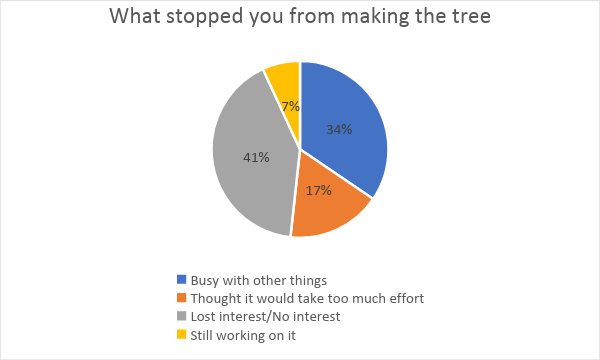
I also created a survey, consisting of a series of questions, that I asked my friends, people in my family, people in my martial arts club and teachers. In total, I got answers from 42 people.

The results are below:









These results indicated that the majority of the people had not created a family tree (69%) however, a lot of the people who said that they had not, were at some point interested in making a family tree (76% of the people who said no). This indicated to me that if I was to create my project, there would potentially be many end-users. Knowing that Ancestry and Gramps were the highest used software, suggested that I should inspect them, to see if there were any ideas I could maybe get. Finally, the fact that most people who did not make a family tree did not make their tree because they were busy with other things or though it would take too much effort, evidences that I should aim to make my program easy to use, and not populate it with too many unnecessary features.

**Input, Process, Storage, Output (IPSO) Diagram**

|  |  |  |  |
| --- | --- | --- | --- |
| **IPSO** | **Information** | **Necessary for project?** | **Evidence** |
| Input | Standard information about people in family:   * First name * Surname * Date of Birth * Date of Death * Place of Birth |  | Observation/Interview |
| Input | Miscellaneous data about each person:   * Bio * Work Life * Outstanding achievements * Photos * Videos |  | Observation/Interview |
| Process and store | Be able to add, edit and delete people from the database. |  | Observation |
| Process | Create a Mother ID and Father ID for people who enter their mother as ‘Unknown’ or father as ‘Unknown’ |  | Observation |
| Process | Add people to the database who have their personID of the form “Mother of…” or “Father of…” and add a parent of a person where one parent is known (e.g. someone may have entered their father, but not their mother) |  | Observation |
| Process | Once the user has entered people, arrange these people into families. |  | Observation |
| Store | Store families using at least 2 tables in the database. |  | Observation |
| Process | Find out which families should be on the same row |  | Observation |
| Output | Display the entire family tree with all entered families on it. |  | Observation |
| Output | Lines connecting relevant families |  | Observation |
| Output | Display information of person when they are clicked on in the tree |  | Observation |
| Process | Have a zoom/scroll function and a resizing window function |  | Observation |
| Input | Be able to say how two people who are in the database and have been chosen by the user, are related. |  | Observation |
| Process | Calculate statistics about the family such as mean age |  | Observation |
| Output | Output these statistics in a Microsoft Excel document |  | Observation |
| Output | Information about a person if they are clicked on |  | Observation |

Key:   
 Green = Definitely needed

Yellow = Not definitely needed, but should be included if possible

Red = Nice to have, but no negative impact if not included

**Data Dictionary (For Database)  
  
Table 1 – tblFamily**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Data Item | Data Type | Validation | Sample Data | How is it Created? |
| Person’s first name | TEXT | Only letters | Dariyan | Entered by user |
| Person’s surname | TEXT | Only Letters | Khan | Entered by user |
| PersonID | TEXT | 2 letters followed by 4 digits (unless in special circumstances where people who have not been entered by the user have been added so that the tree can be drawn) | DK1908 | Takes first letter of first name, followed by first letter of surname, followed by a randomly generated 4-digit number. |
| Date of Birth | DATE/TEXT | Only numbers and symbols (no letters)  =10 characters (including slashes to separate day/month/year) | 03/07/01 | Entered by user |
| Date of Death | DATE/TEXT | Only numbers and symbols (no letters)  =10 characters (including slashes to separate day/month/year) | 29/12/2431 | Entered by user |
| Place of Birth | TEXT | Only letters – no numbers | London | Entered by user |
| Gender | Male/Female | Only Male or Female | Male | Chosen via a drop-down menu |
| MotherID | TEXT | 2 letters followed by 4 digits (unless in special circumstances where people who have not been entered by the user have been added so that the tree can be drawn). Must be in database. | SK9054 | Chosen via drop-down menu |
| FatherID |  | 2 letters followed by 4 digits (unless in special circumstances where people who have not been entered by the user have been added so that the tree can be drawn). Must be in database. | TK8239 | Chosen via drop-down menu. |

**Table 2 – tblSingleFamily**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Data Item | Data Type | Validation | Sample Data | How is it created? |
| FamilyID | TEXT | Will always be of the form “FIDx” where x is a positive integer | FID1 | 1st family created will have FID1, 2nd will have FID2 and so on. |
| Mother | TEXT | 2 letters followed by 4 digits (unless in special circumstances where people who have not been entered by the user have been added so that the tree can be drawn). Must be in database. | PJ4391 | Will be the mother of a certain family after each family is created |
| Father | TEXT | 2 letters followed by 4 digits (unless in special circumstances where people who have not been entered by the user have been added so that the tree can be drawn). Must be in database. | HG1927 | Will be the father of a certain family after each family is created. |

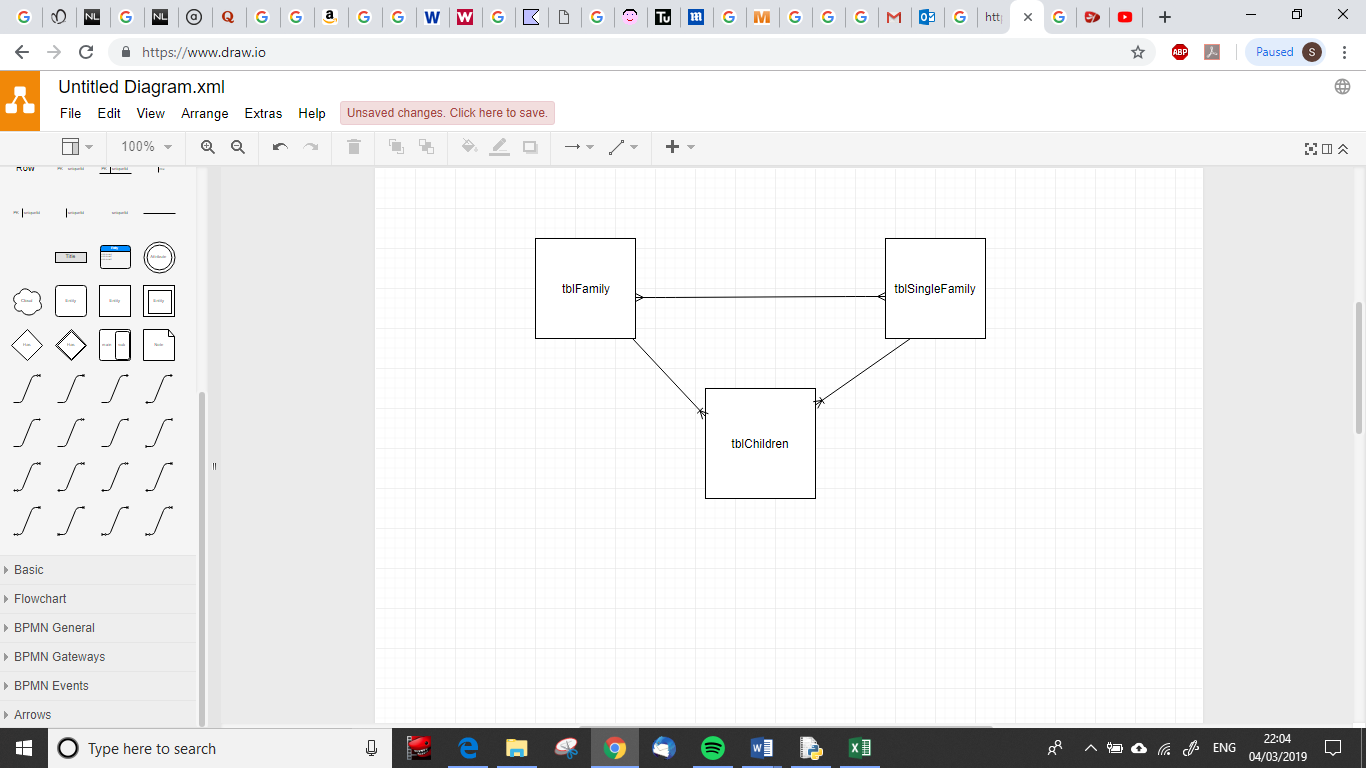
**Table 3 - tblChildren**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Data Item | Data Type | Validation | Sample Data | How is it created? |
| FamilyID | TEXT | Will always be of the form “FIDx” where x is a positive integer | FID1 | 1st family created will have FID1, 2nd will have FID2 and so on. |
| ChildID | TEXT | 2 letters followed by 4 digits (unless in special circumstances where people who have not been entered by the user have been added so that the tree can be drawn). Must be in database. | PJ4391 | Will be one of the children of a certain family after each family is created. Each child in each family will have its own separate record. |

**ER Diagrams**

tblFamily(PersonID,Firstname,Surname,DOB,DOD,Gender,MotherID,FatherID)

tblSingleFamily(FamilyID,Mother,Father)  
  
tblChildren(FamilyID,PersonID)



**Research About The Tree**

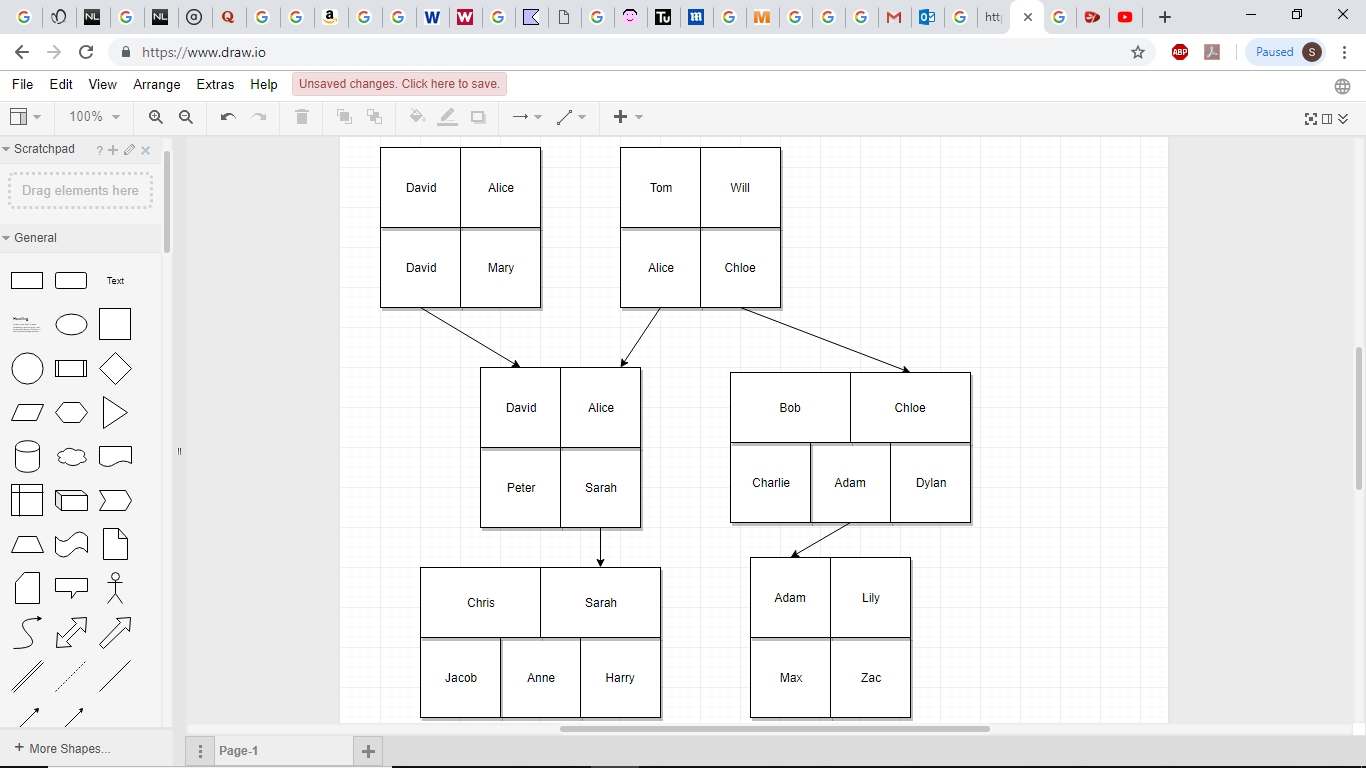
**Drawing the Tree**

I looked up different ways to represent families before deciding to group individuals into ‘nuclear family’ units. One of the ways was the ‘Ahnentafel’ method, where each person is assigned a positive integer, and then that person’s father is double that number, and the mother would be double + 1. However, one problem with this method was that it did not allow for siblings. This would have drastically reduced the complexity of my program, so I could not choose this method. But, if I was going to create an option where no siblings were to be added (just parents and children of a fixed person), this method would be perfect. Another system was the ‘Henry’ system. The system begins with 1. The oldest child becomes 11, the next child is 12, and so on. The oldest child of 11 is 111, the next 112, and so on. I felt that this system was more comprehensive than the Ahnentafel system but the problem about displaying the family tree would still not be solved. Also, previously in my ‘Analysis’, I looked at a fan chart. Although the fan chart looks aesthetically pleasing, it does not show siblings and children of the person we are interested in. Therefore, I decided to not make this my primary mode of displaying the information either.

**Storing the Tree**

At first, I thought that the graph used to represent the whole family would be a tree, as suggested by the name ‘family tree’. However, I soon realised that although my data structure used would involve no cycles, it would have to have more than one root node, as you could have grand-parents (for example) on both sides of the tree that have unknown parents. After research, I came across a tree-type called phylogenetic trees, that are used to represent the evolution of a species. However, I finally decided that my tree was actually not a tree but more generally a graph, that was acyclic, and that I could choose to make directed or undirected.

**Sample Tree**



This is a unit

Notes:

* In every unit, the top left box is the father and the top right box is the mother. The people under them in the same unit are their children.
* The arrows always point downwards
* People born in the same generation are on the same row.
* If there are more than 2 siblings, the length of the boxes for the parents will be wider to compensate.
* Each box, when I actually make it, is likely to be a button, so that when you click on it the information about the person you clicked on is displayed.

**Consideration of Possible Solutions to the Problem**

**Database Section**

First, I need to create a prototype of the code. The code should allow people to add new people to the database, edit people and delete people as well as an exit option. Once I have completed this, I will be able to see if any other options need to be added. Then, I will start creating a GUI, which connects to the back-end database when a user wants to do an action involving the database. The GUI will be made in a Python module called ‘Tkinter’. All the information entered so far by the user will be stored in one table in the database called tblFamily. I will write the database queries using ‘sqlite3’. After the user enters their information, their Person ID will be generated, using their first name, surname and a randomly generated 4-digit number. This Person ID will uniquely identify them (primary key). In order to view my database to check if the correct queries have been made etc., I will use a software called DBrowser.

**Displaying tree**  
Once the user has entered people into the database, they can choose to draw a tree. To do this, I will first create the ‘nuclear family units’ as seen in the sample tree. To tell if two people are “co-parents” (e.g. mum and dad) I will look at the intersection of the sets of their children. If there is an intersection, then the 2 people have a co-parent relationship and if the intersection is nothing, then they do not. When everyone has been put into families, tblSingleFamily and tblChildren can be made and data entered into them. Then, I will need to deal with people who have unknown parents and unknown spouses etc. Currently, the best way that I can see in tackling this problem would be to give unique Person IDs, that are of a different format to the other ones, to these people. After this, I need to see how different units are connected to each other and find out which units are in the same generation. To do this, I will create a matrix with all the connections using a depth first traversal. Finally, to display the actual tree, I will use a module called ‘Kivy’. Kivy is similar to Tkinter, as it allows a GUI to be created. But, Kivy has more features and is more interactive, which is more important for this part of the project as users should be able to click on boxes, and perhaps zoom in or zoom out, so that they can see more or less of the tree at once.

**Extension Tasks**

One of my extension tasks is to display statistics. These will include the mode, median, mean, variance and standard deviation of the ages. Calculating standard deviation and variance is in the A-Level Mathematics syllabus. If I also have time, I would like to display some information such as the proportion of females, a pie chart of where people were born in Excel. Another extension task was to have a feature where if you chose 2 people, the relationship between them would be given. This could be found by using the matrix that will be created while trying to draw the tree.

**Programs/important modules needed:**

* Python
* Tkinter
* DBrowser
* Sqlite3
* Kivy
* Possibly Microsoft Excel

**Objective and Performance Criteria**

|  |  |  |
| --- | --- | --- |
| Core Objectives | | |
| Number | Objective | Performance Criteria |
| 1 | Create a GUI interface that connects to a back-end database. | The GUI must be easy to use for all users, and be able to display different forms and windows, depending on which option a user chooses. |
| 2 | Allow users to add, delete and edit people in the back-end database. | These must be able to be done using buttons and forms displayed using the GUI. Adding people: A form should be displayed with the relevant information text boxes of information and drop-down menus which the user can choose from, so that they can eventually enter this information into the database.  Editing people: a form should be displayed with the information of the person they chose to edit, in the same form used for adding people. When submit button is pressed, the old record should be replaced with the new record.  Deleting people: A drop-down menu should be displayed allowing the user to pick someone in the database. Once they press the submit button, that person should be deleted. |
| 3 | Create tblFamily if it does not exist and if the user wants to edit a person when the database has not been created yet for example, they should be forced to enter a person first. | If the table does not exist when a user enters presses the submit button, then the table must be made. |
| 4 | To deal with people who enter ‘Unknown’ as their mother or father. | If someone (for example Bob) enters ‘Unknown’ as their mother, then their mother will be stored as “Mother of Bob…” If Bob had a sister called Alice, Alice could then choose her mother as “Mother of Bob…” |
| 5 | Create a Queue that contains family ID’s | This queue should be able to dequeue items in a FIFO order. |
| 6 | Put everyone into families and put these families into the database | Everyone that the **user** has entered should be a child in a certain family. Where the user has not entered parents for particular persons, new people will be added to the database but, their person ID will be of a different format. For example, if I (Dariyan Khan) had not entered my dad into the database, the program would say that my father is “Father of Dariyan Khan ….”. Once the families have been made, two new tables will be created, and the parents and children of each family will be entered into tblSingleFamily and tblFamily respectively. |
| 7 | Delete and create tblSingleFamily and tblChildren each time the ‘draw tree’ or ‘find relationships’ option is run | This is to ensure that if someone draws the tree, then adds a few people to the database and then decides to draw the tree again, recreating the tables means that the new person is accounted for. |
| 8 | Create a stack that can be used for the depth first traversal. | This stack should be able to have items pushed onto the stack and popped from the stack in a FILO order. |
| 9 | Create an algorithm that produces a family matrix | This algorithm should be recursive and carry out a depth first traversal, in order to find out which families are connected to which other ones. 1’s and -1’s should be stored in the nxn matrix (where n is the number of families) based on whether a family is directly below or above the other family in question (1 for below and -1 for above). |
| 10 | Decide which people are on the same row | This will be done by using the family matrix above, and counting the distance (will be 1 or -1 for each connection) between each family and a fixed family (usually FID1). Families with the same distance will be on the same row. |
| 11 | Draw the family tree | The family tree will be drawn in Kivy, and will have families in the same generation on the same row, and try to have families with a common parent-family close to each other, to reduce any overlaps when lines are drawn between two families that contain a common person |
| 12 | Draw lines between families that are different colours depending on whether the person linking the 2 families is male or female. | Lines should be drawn between families that are connected to each other, and could be different colours if the person linking the families is male as compared to female. |
| 13 | Display information when someone clicks on a person. | My program should display the information of a person, which includes at least their full name, dob and place of birth, when a person displaying the tree clicks on them. |
| 14 | Prompt the user to choose mothers/fathers for people who do not already have one entered | The user should be confronted by a series of drop-down menus for example, that allow the user to choose a mother/father for these people. |
| 15 | Have exception handling when entering data into the form. | The user should not be allowed to enter and submit invalid data into the database. Also, they must enter a firstname and a surname of the person they are entering. |
| Extension Objectives | | |
| 1 | To be able to choose 2 people, and have the program display the relationship between them. | For example, if I entered myself and Zoran (one of my uncles) it would display “Dariyan Khan’s Auntie’s Husband is Zoran” |
| 2 | To display some simple statistics such as mean age, standard deviation of ages etc. | All of these statistics will either be displayed in a python shell, tkinter window or excel. |
| 3 | Display pie charts etc. of the data | These should be read from the database and other data structures in the code, and be displayed in Excel |
| 4 | Be able to import data from a text file (csv) | A user should be able to import data from a text file in order to get the information into the database, instead of having to manually input everyone |
| 5 | Create a geographical map option. | This should be an option where the user will be displayed with a map of the world, with dots representing where people entered into the database are living/ were born. |
| 6 | Be able to click on people in the tree and have a zoom in/out function or perhaps a scroll function. | When a person clicks on a person in the tree, the information of that person will be displayed. A zoom in/out function or scroll will allow people to look at different parts of the tree more easily. |

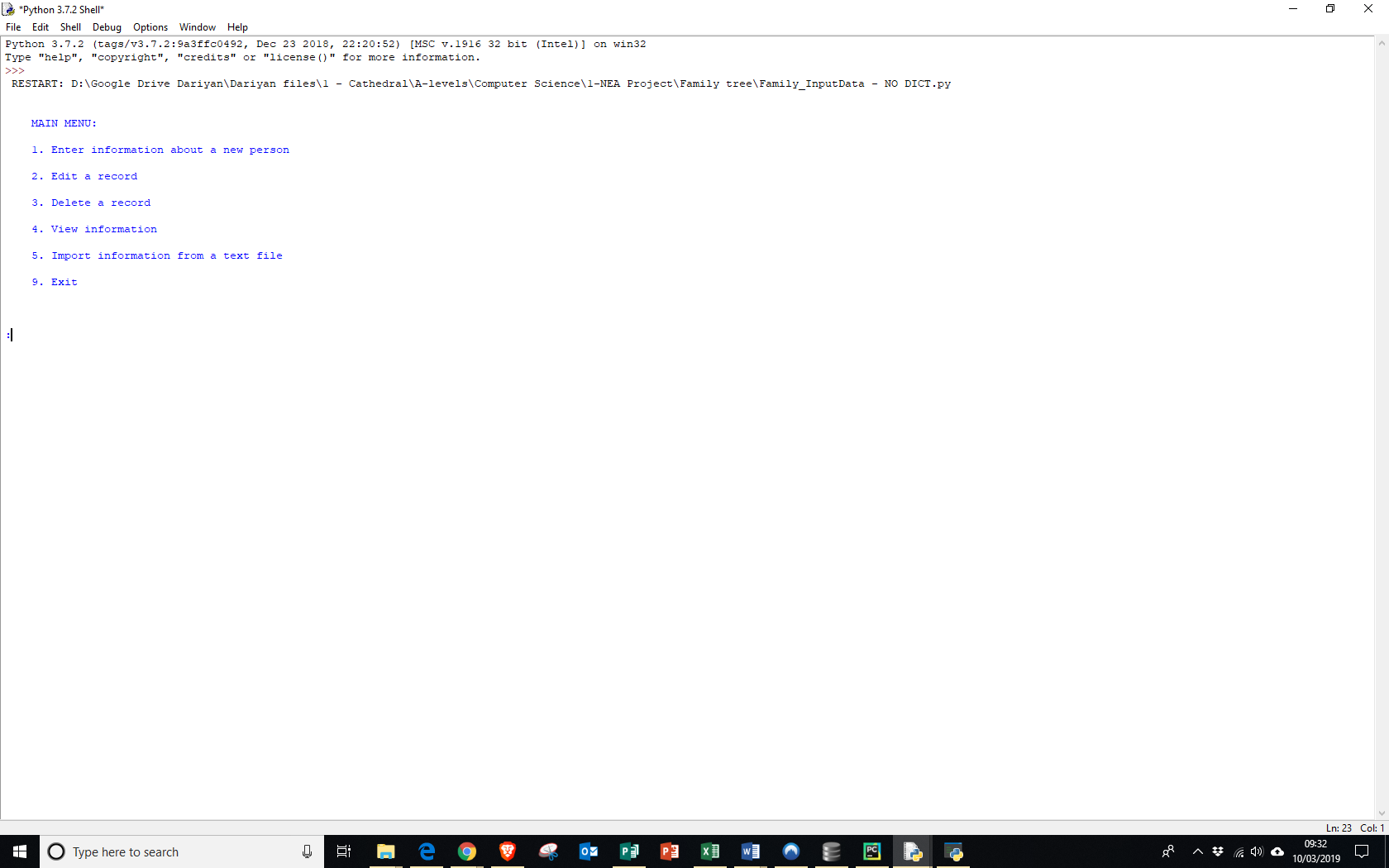
**Design**

**Prototypes**

**Database prototype\***

I first created a prototype for the database section of my project. This allowed me to assess and discuss with my end users what functionalities should be available to the user, and where and how should they be displayed. In my prototype, the whole program ran in the python shell, as denoted by the screen shots below. The options where add a person, edit a person, delete a person, view all entries and import data from a text file. This prototype was completed by mid – December.

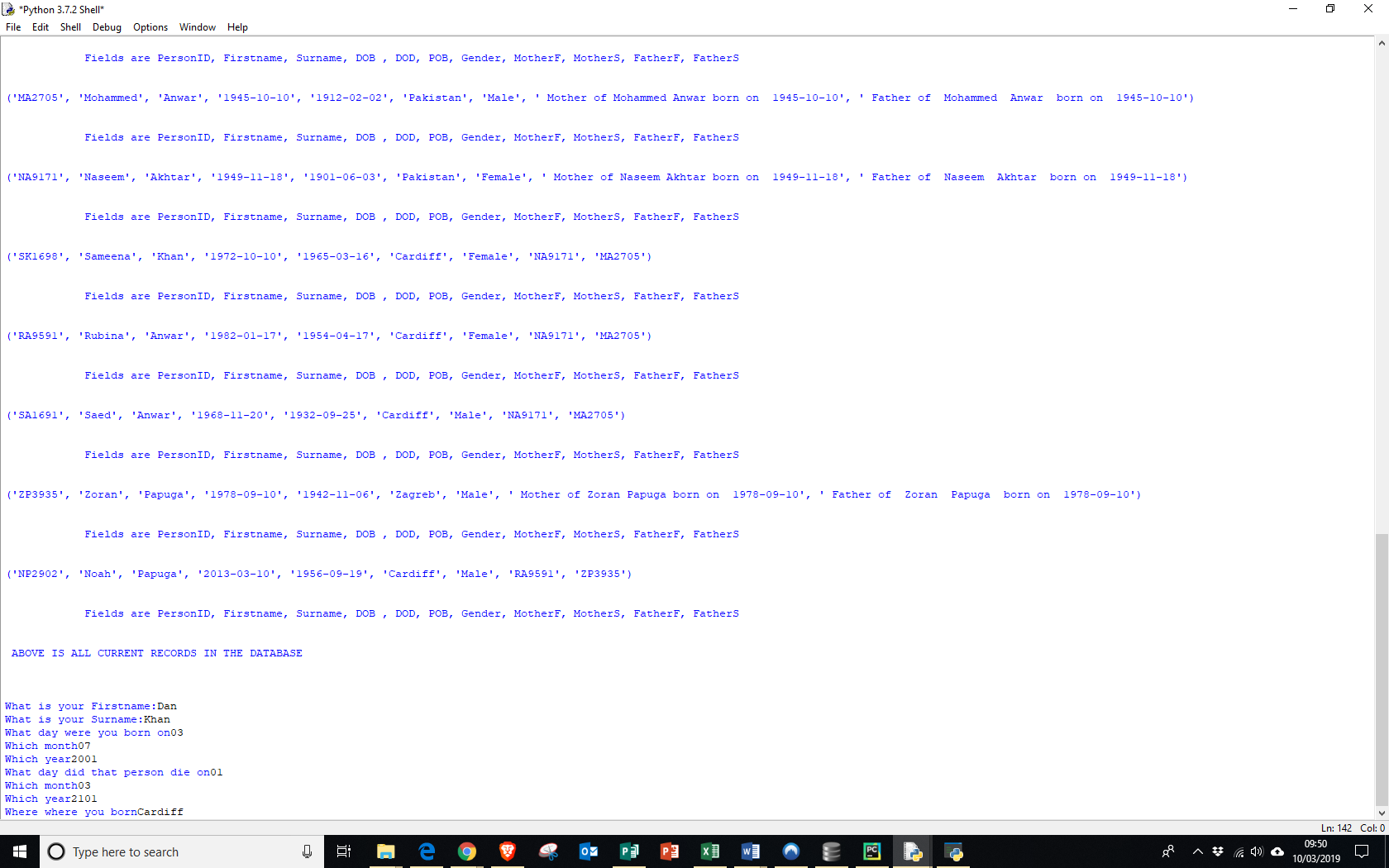
Main Menu



The main menu was quite simple, but was easy for the user to read and understand.

\*See Appendices for prototype code

Enter information about a new person

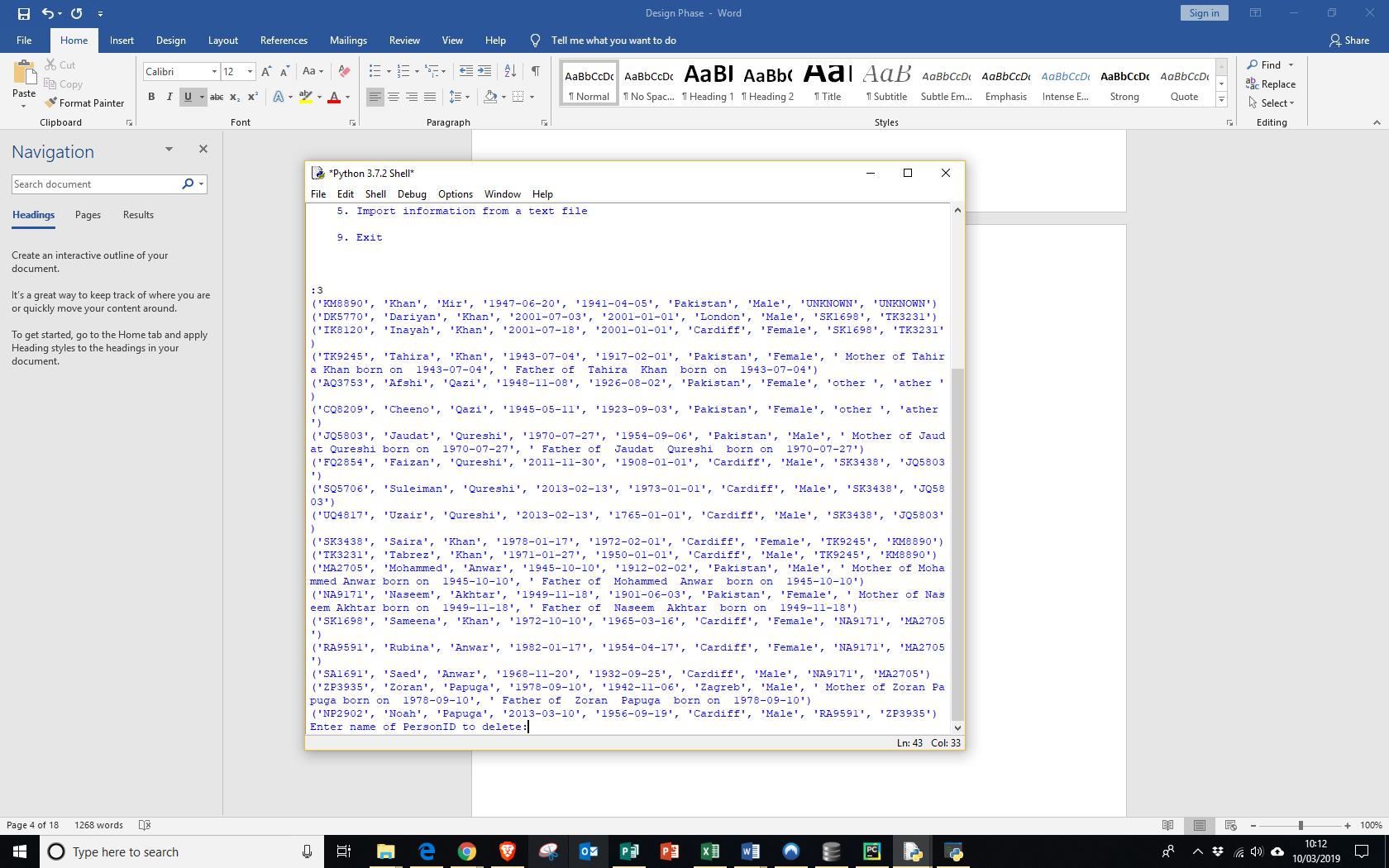


As you can see, when entering “1”, all the records in the database would be printed, before then sequentially asking the user to enter information about each person. Following discussion with a couple of my end users, we decided that displaying everyone in the database in this way would have been annoying to the user, as they would have to scroll through many people, if the user had entered lots of people, to see if someone was already entered. Not only that, but asking the user to enter information in this way would be prone to errors, and so would not only need a lot of error handling on my part, but if the user incorrectly typed in a piece of information such as their mother’s first name etc, this could lead to repercussions later on when the tree is being drawn.

Edit a person

Editing a person, just like entering a person, displays everyone in the database before choosing who to edit, what information to edit, and the new value. But, the problem is that if you want to edit a person, you have to first find them, and then remember information about them before typing it in. Also, what would happen if you wanted to change more than one field about a person? You would be sent to the main menu, and then have to enter all the information again.

Delete Person



The delete option again works in a similar way to editing a person, and so it again has similar problems.

Viewing all entries in the database and importing data from a text file

The ‘viewing all entries in the database’ option would just print out each person as done in the previous three options. Unfortunately, I did not get around to completely coding the ‘import data’ option, as instead of devoting time to it, I wanted to get feedback from some end users and decide whether I should include it, and if they are satisfied with the current layout.

Thoughts and feedback after prototype

This prototype confirmed that I should definitely have a GUI as, not only would this help with creating an increasingly more user-friendly experience but, using drop-down menus for selecting people for example, I could reduce the amount of error-handling I would have to do, while reducing the potential for errors on the users’ part. Also, my end users said that they would like the options to draw the tree etc. to be on the same menu. In addition, because my edit etc. menus would have drop-downs with everyone entered into the database in them, a “view all people” option may be superfluous. Also, they said that the import text file data option would be nice to have, it is not essential. Therefore, I made it as one of my extension objectives, which I can do if I have time at the end. Also, in my prototype I was using classes but after my prototype, I drifted away from using classes, as I felt that they were unnecessary.

**Database design**

tblFamily(PersonID, Firstname, Surname, DOB, DOD, POB, Gender, MotherID, FatherID)

DOB = date of birth  
DOD = date of death  
POB = place of birth

tblSingleFamily(FamilyID,Mother,Father)

tblChildren(FamilyID,PersonID)

|  |  |  |  |
| --- | --- | --- | --- |
| Table Name | tblFamily | | |
| Primary Key | PersonID | | |
| Foreign Key | N/A | | |
| Data Item | Data Type | Validation | Sample Data |
| PersonID | String | Not NULL | DK8413 |
| Firstname | String | - | Dariyan |
| Surname | String | - | Khan |
| DOB | Date | Is of form (YYYY/MM/DD), where the month is between 1 and 12 and the day is between 1 and 31 | 03/07/2001 |
| DOD | String | Is of form (YYYY/MM/DD), where the month is between 1 and 12 and the day is between 1 and 31 | 07/09/1985 |
| POB | String | - | Cardiff |
| Gender | String | Must be either male or female | Male |
| MotherID | String | - | SK6529 |
| FatherID | String | - | TK6780 |

|  |  |  |  |
| --- | --- | --- | --- |
| Table Name | tblSingleFamily | | |
| Primary Key | FamilyID | | |
| Foreign key | N/A | | |
| Data Item | Data type | Validation | Sample Data |
| FamilyID | String | Not NULL | FID1 |
| Mother | String | - | SK9812 |
| Father | String | - | TK7891 |

|  |  |  |  |
| --- | --- | --- | --- |
| Table Name | tblChildren | | |
| Primary Key | FamilyID, PersonID | | |
| Foreign key | N/A | | |
| Data Item | Data type | Validation | Sample Data |
| FamilyID | String | Not NULL | FID1 |
| PersonID | String | - | SK9812 |

**SQL Statements**

|  |  |
| --- | --- |
| What does it do | SQL statements |
| Creates tblFamily | Create TABLE tblFamily  (PersonID CHAR(6) NOT NULL PRIMARY KEY, Firstname VARCHAR ,  Surname VARCHAR, DOB DATE, DOD DATE,  POB VARCHAR, Gender VARCHAR(6), MotherID CHAR(6),  FatherID CHAR(6);) |
| Inserts values into tblFamily ( I have used the sample data in the table above) | INSERT INTO tblFamily  VALUES(“DK8413”, “Dariyan”, Khan, “03/07/2001”, “07/09/1985”, “Cardiff”, “Male”, “SK6529”, “TK6780”) |
| To delete value from tblFamily | DELETE FROM tblFamily WHERE PersonID = “DK8413” |
| To update a value in tblFamily (I have updated the place of birth for the sample data) | FROM tblFamily SET POB = “London” WHERE PersonID = “DK8413” |
| To display all records in mothers and fathers pop-up menu, in alphabetical order of first names. | SELECT \*  FROM tblFamily ORDER BY Firstname |
| To delete a table (e.g. tblSingleFamily) | DROP TABLE tblSingleFamily |
| A Query that involves 2 tables example | SELECT FamilyID FROM tblSingleFamily,tblFamily WHERE tblSingleFamily.Mother=tblFamily.PersonID OR tblSingleFamily.Father=tblFamily.PersonID |

Above are the main SQL statements that I will use. Of course, I will most likely use many more but, they will be variants of the ones in the table above.

**Critical Path**

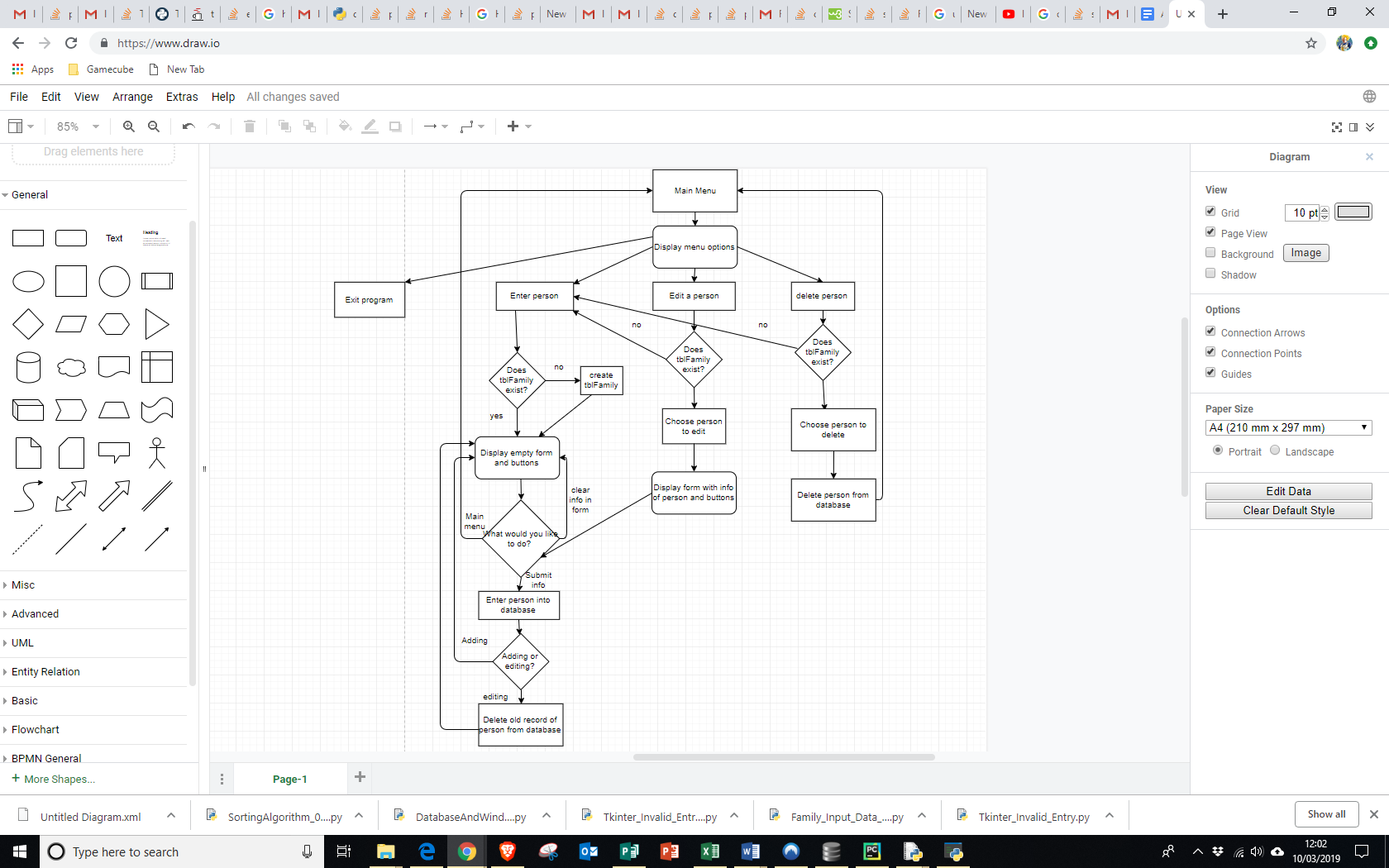
The critical path will be the order that I have written my objectives, and here is the order:

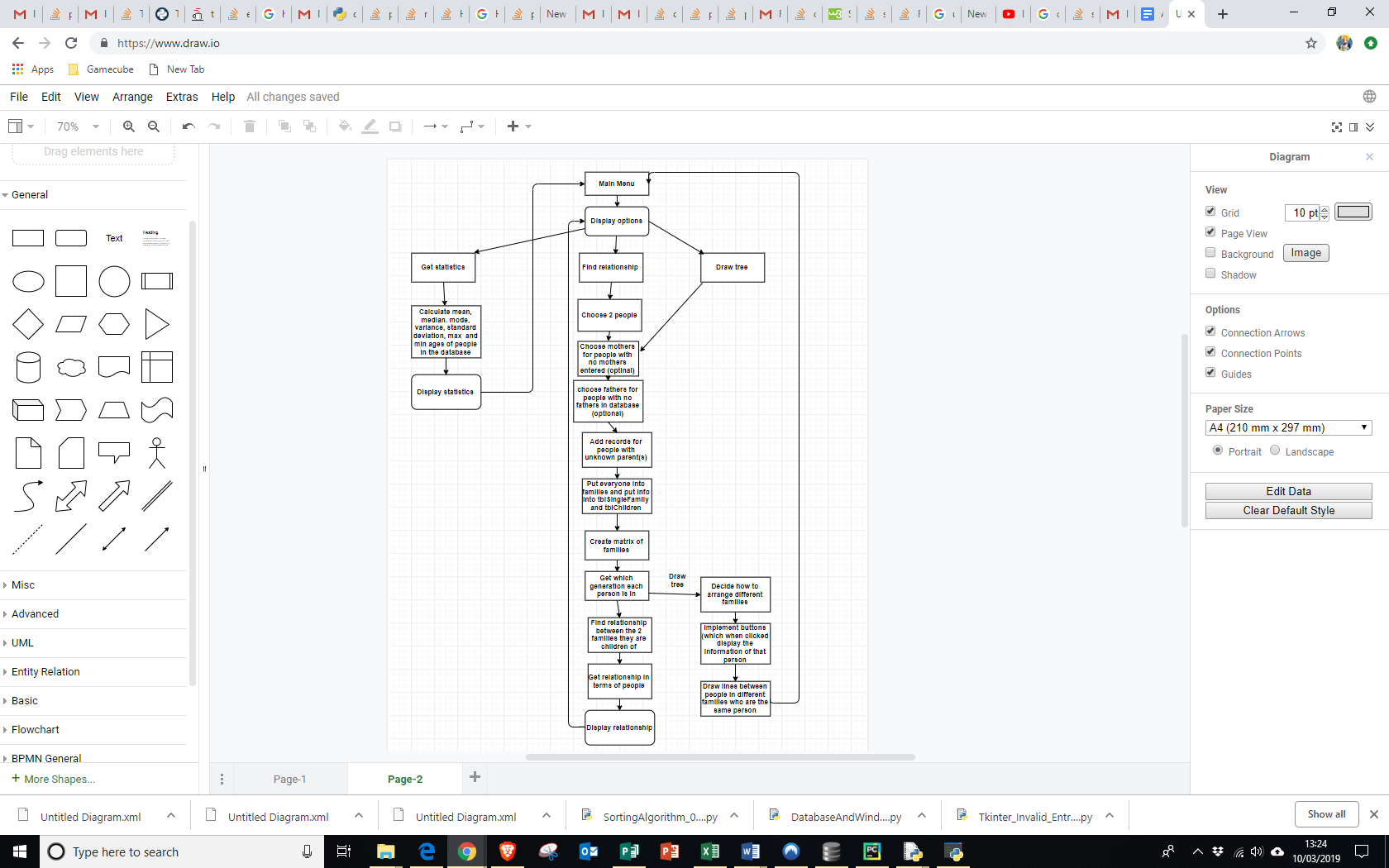
|  |  |
| --- | --- |
| 1 | The project must allow people to enter, edit and delete people into the database. |
| 2 | The user should be confronted with a GUI when they start the program, that has options to enter, edit and delete a person as well as exiting the program. |
| 3 | When they click on the ‘enter’ option, a form should be displayed which they can enter information into. |
| 4 | A random personID should be created using the first letter of the entered person’s first name, surname, and a randomly generated 4-digit number. |
| 5 | When they click on the ‘edit’ option, a form identical to the form in the ‘enter’ option should be displayed, but the text boxes etc. are filled with the information of the person. |
| 6 | The delete option should be created, that allows a user to delete a particular person. |
| 7 | After the user has entered everybody that they would like to into the database, people who have unknown an unknown parent(s) or spouse need to be dealt with separately (adding them into the database with different personID’s to the usual ones) |
| 8 | Sort everyone into families, based on the information given by the user |
| 9 | Create two new tables in the database, and enter the information about the families into the database. |
| 10 | Create a functioning queue (should be able to dequeue items) |
| 11 | Create a depth first algorithm, to find which families are related to which other ones. Information should be inputted into the empty ‘nxn’ matrix as the algorithm progresses. |
| 12 | Use the matrix to calculate which people should be on the same row when drawing the tree (people on the same row are in the same generation) |
| 13 | Put a list of all the families into one big list, in order to make the creation and testing of family matrix in kivy easier. |
| 14 | Create the family tree in Kivy and be able to display it |
| 15 | Draw lines, which may be different colours, between two nuclear families with a common person. |
| 16 | Link together all the different parts of my project, so that it is like a final product. |

**Overall Systems Design**



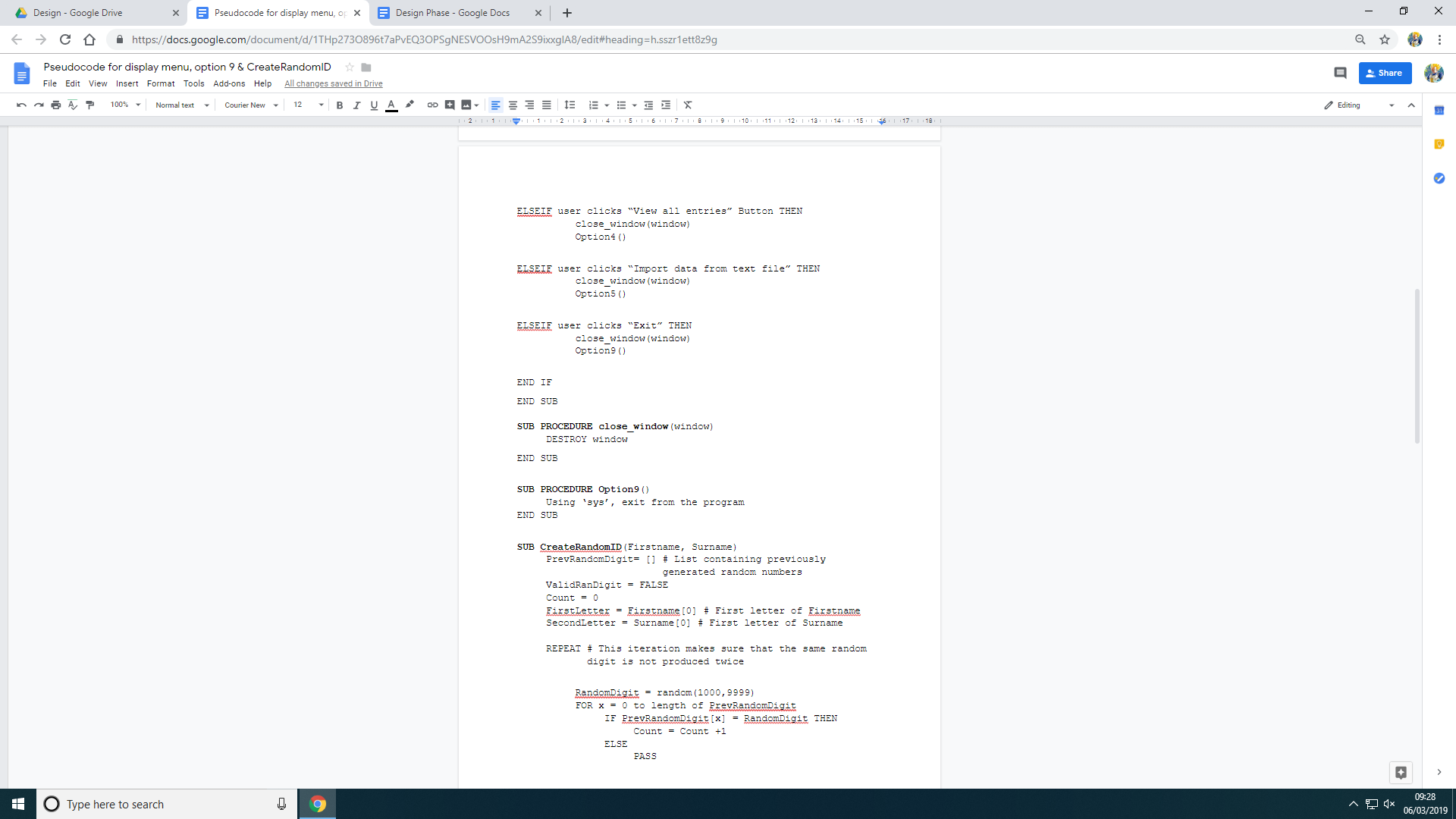
**Flow Diagram for adding, entering and deleting person**

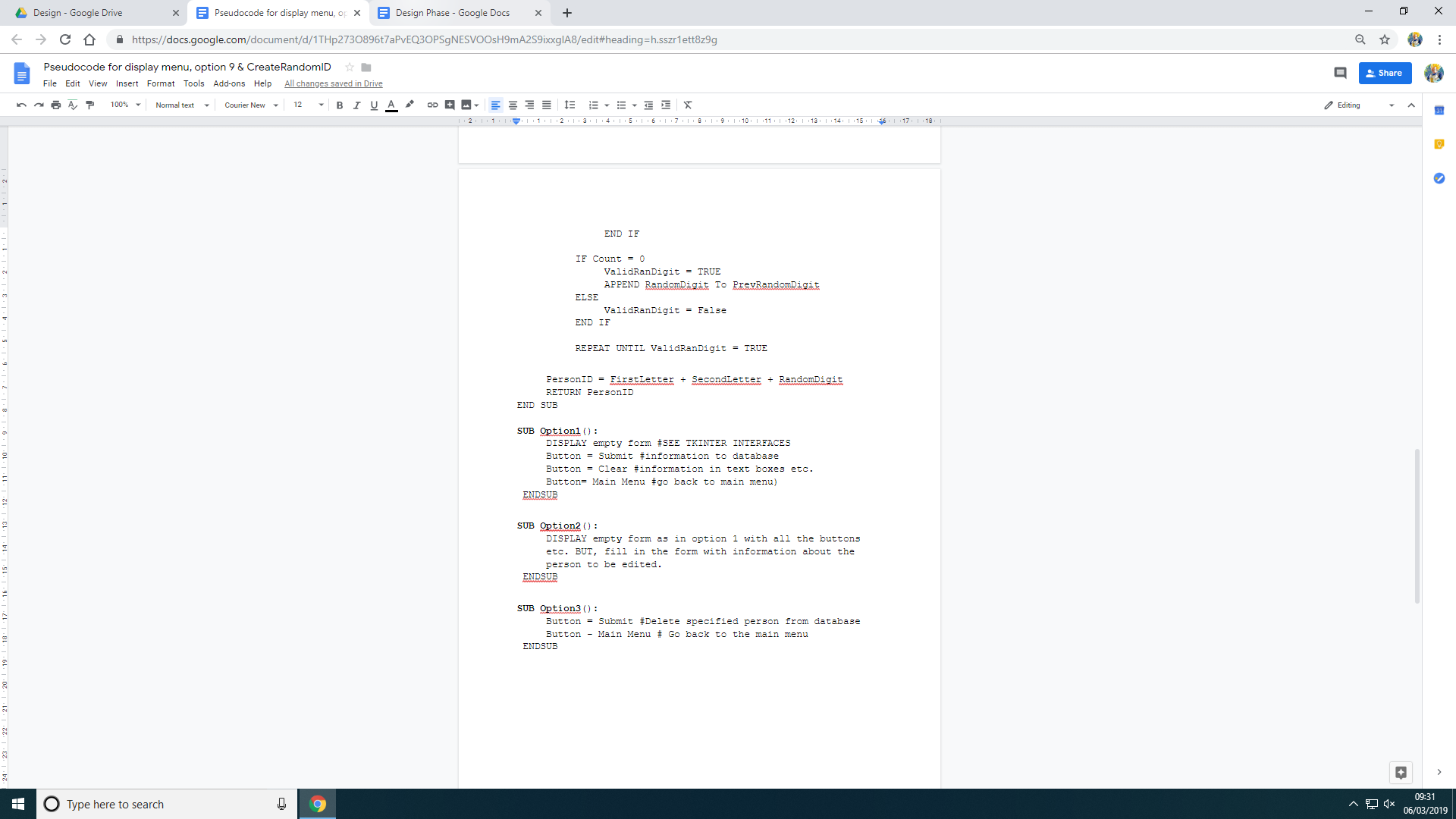


**Flow diagram for Drawing Tree, Getting Statistics and Finding Relationships**

**Pseudocode for adding, editing and deleting person**







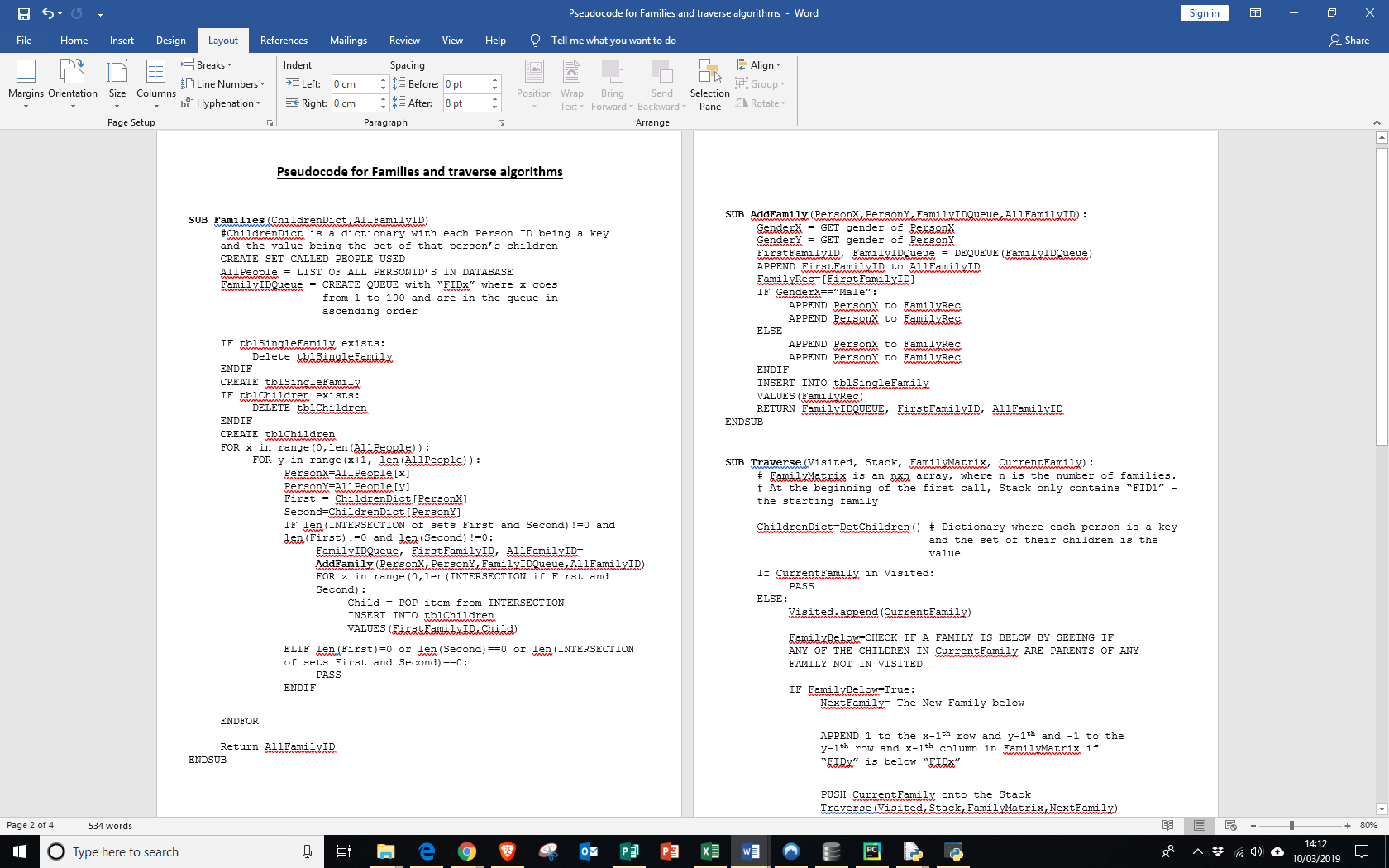
Notes:

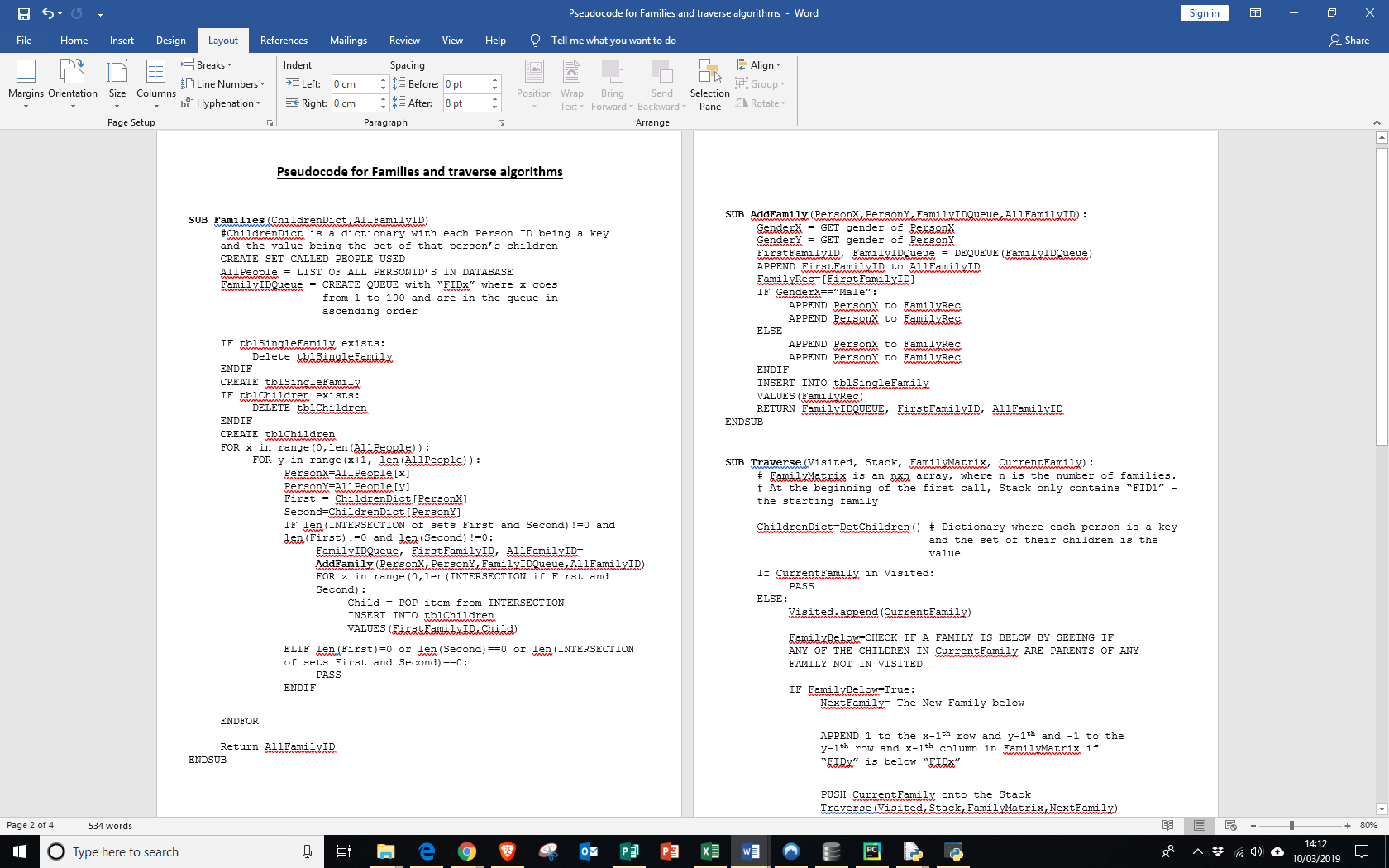
The pseudocode above is for the main menu. I have only written pseudocode for the above options, and not the options for buttons for draw tree etc, as they will be laid out in a similar way. All of these algorithms have been previously discussed and are self-explanatory, apart from the ‘CreateRandomID’ subroutine. This subroutine generates a random PersonID, that will be used as a primary key for each record. It works by choosing the first letter of the first name and surname, and pre-pending these to a randomly generated 4 digit number.

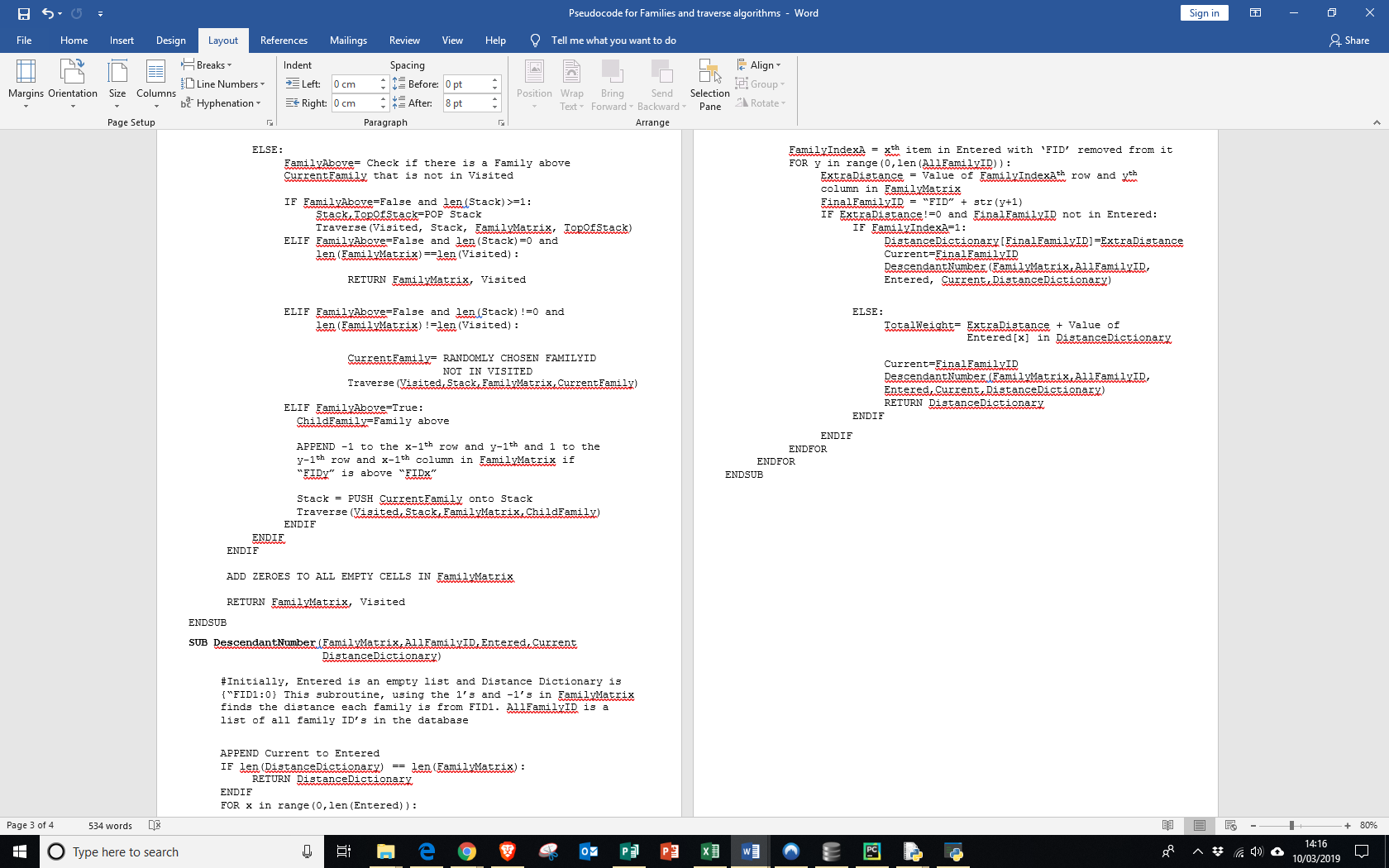
Also, near to the end of my project, I will add exception handling. Errors will occur mostly where the user is entering information into the form. If the user does any of the following things, then a box saying ‘Invalid Entry’ will be displayed and the record will not be entered into the database:

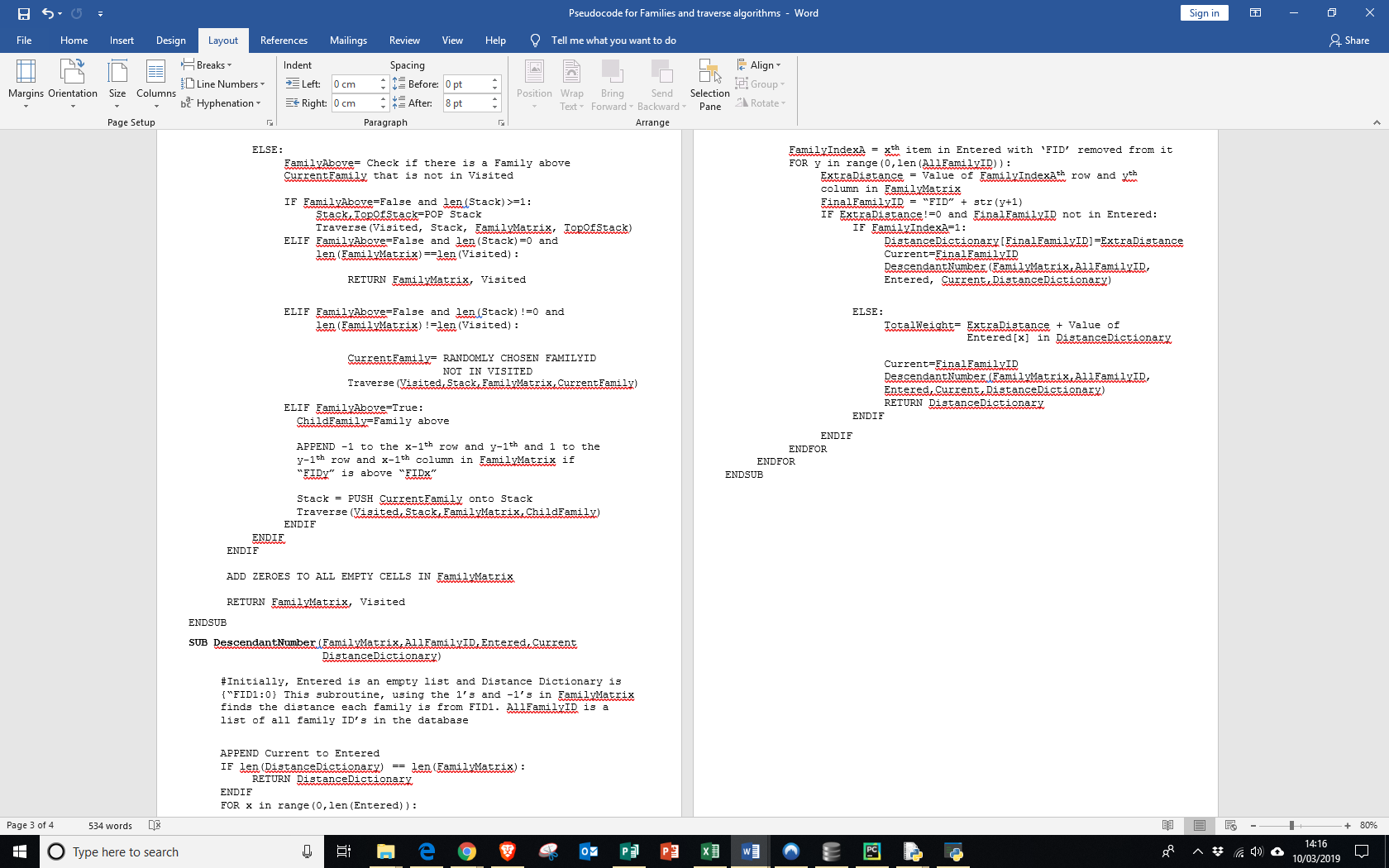
* Doesn’t enter a first name
* Doesn’t enter a surname
* Enters a date of birth greater than the date today
* Enters a date of death greater than the date today
* Enters a date of death greater than the date of birth
* Enters a date that cannot exist, such as 30/02/2019

Also, for the editing and deleting person options, I had to make sure that a user would keep the same Person ID after being edited. Otherwise, I would have to find out all the places where the old Person ID is used, and change them all to the new ID. Also, when deleting a person, I had to make sure that if someone was deleted, they would be deleted from everyone who had that person as their mother/father.

**Pseudocode for generating families and family matrix**







Notes:

* The Traverse subroutine is a depth first search (DFS) and so it uses a stack.
* It works by starting at family with FID1, and then seeing if any other families are first ‘below’ it (below being if a child in FID1 is the parent in the other family). This family must also not be in the visited list, which is updated each time the algorithm changes the variable ‘current family’.
* If there is a family below, then this will be recorded in the matrix, the variable ‘current family’ is pushed onto the stack (originally containing only “FID1”, and then the new family becomes current family, and then the program calls itself.
* If there is no family below, the program checks if there is a family ‘above’.
* If there is, the same process is repeated.
* If there is no family above or below and the number of values in the stack is not zero, the first item is popped of, and the algorithm is called with this value.
* If the stack is empty and all the families have been visited, then the family matrix is returned.
* But if the stack is empty but all the families have not been visited, then that means that the graph of the families must be unconnected. Therefore, in this case a new family that has not been visited is randomly chosen, and the traverse algorithm is then called using that.
* The descendant number algorithm is also recursive.
* It first sees which families are connected to FID1, and assigns them a distance of 1 or -1, depending on the value assigned in the family matrix. These families are appended to a ‘connected’ list.
* Then, it repeats the process for each of the families in entered, but instead of just assigning 1 or -1, the distance between the Family ID in the entered list and FID1 is also added to the 1 or -1. This is repeated until all of the families are in the entered list.
* The ‘families’ algorithm first creates a dictionary, with the key being each PersonID in the database, and the value being the set of that ID’s children. Then, algorithm goes through each combination of people with a nested for loop.
* If the intersection of the sets of the values of two people is not zero, then they are deemed to be a couple/ in a relationship and so, a unit is created between those two people and their intersection.

**Drawing the Tree**

**Some more information about Kivy**

In my analysis, I explained a few simple reasons as to why I used Kivy. Here are some more facts about it:

* In Kivy, I assembled blocks of strings inline in my python code so that I could add variables but still describe the layout in Kivy language, which is easier to use than trying to create all the widgets in python. Once the required kv language string was put together, I used the Kivy ‘Builder’ module to parse the string and create the display of on-screen widgets. While doing it this way, I had to be careful with my use of single and triple quotes, as well as indentations, whilst creating the string, because the Builder module would throw an exception if even one space was incorrect.
* The benefit of using kv language was that Kivy then automatically works out all the dependencies between widgets and subsequently automatically detects and reacts to changes. For example, if you resize the window, Kivy handles resizing all the contained widgets. If I wrote the code in Python using something such as Tkinter, I would have to handle the resizing myself.

**Researching Kivy**

When coding the Kivy section, because it was new to me, I did thorough research before-hand, and made sure I used meaningful variable names. I tried to make my code look neat and add lots of comments, to help with debugging in case of errors. Since it took a lot of thinking and adjusting over time, the comments meant I would more easily understand and remember the reasons for my coding when I came back to it the next time.

**Imports**

* Kivy App – this import is necessary in order to open a kivy-app window. It is the base class from which you make your own app by creating a subclass. For my program the subclass was StartFamilyTreeApp and it is set up as follows:  
   class StartFamilyTreeApp(App):  
  To run the app, an instance of the subclass is created and then its run method is called:  
   StartFamilyTreeApp().run()   
  In order for the program to actually display something on-screen, there must be a method called ‘build’ in the class, that creates the ‘root widget’. Upon research, I saw that you could either write the Kivy definitions that make all the widgets:  
  1. in Python, which I read can be quite difficult  
  2. in a separate file (which would be static), or as I did it…  
  3. assemble it dynamically inline, in Python and pass it to a parser called ‘Builder’.
* Window – This allowed me to get a handle to the window object from which to get the actual height and width of the window in pixels.
* Relative layout – this allows relative positioning.

**Classes**

|  |
| --- |
| FamilyTreeCanvas() |
|  |
| +display\_info |

|  |
| --- |
| Family() |
| +id: string +gen: int  +mum: dictionary  +dad: dictionary +kids: list  +downlinks: list  +pos\_hint: list  +size\_hint: list |
|  |

**Layout**

When doing the layout, I created a variable called sib\_gap. This defined an onscreen gap between sibling-families (where a parent in each are siblings). This was to make them look ‘grouped’ by being slightly closer together than non-sibling families. This only applied for generation 0 however, because in higher generations, a family’s onscreen position was determined by the position of their children families lower down and sibling-families in the higher generations could therefore not necessarily be grouped.

Sib\_group\_gap was a variable that was a distance bigger than sib\_gap, and referred to the distances between groups of sibling-families.

An important subroutine in the making of the layout was the ‘build\_generation’ function.

This would return a string containing the Kivy language description for generation ‘g’. This consisted of nested widget definitions that construct a box layout for each family which is sub-divided into buttons for parents and children.

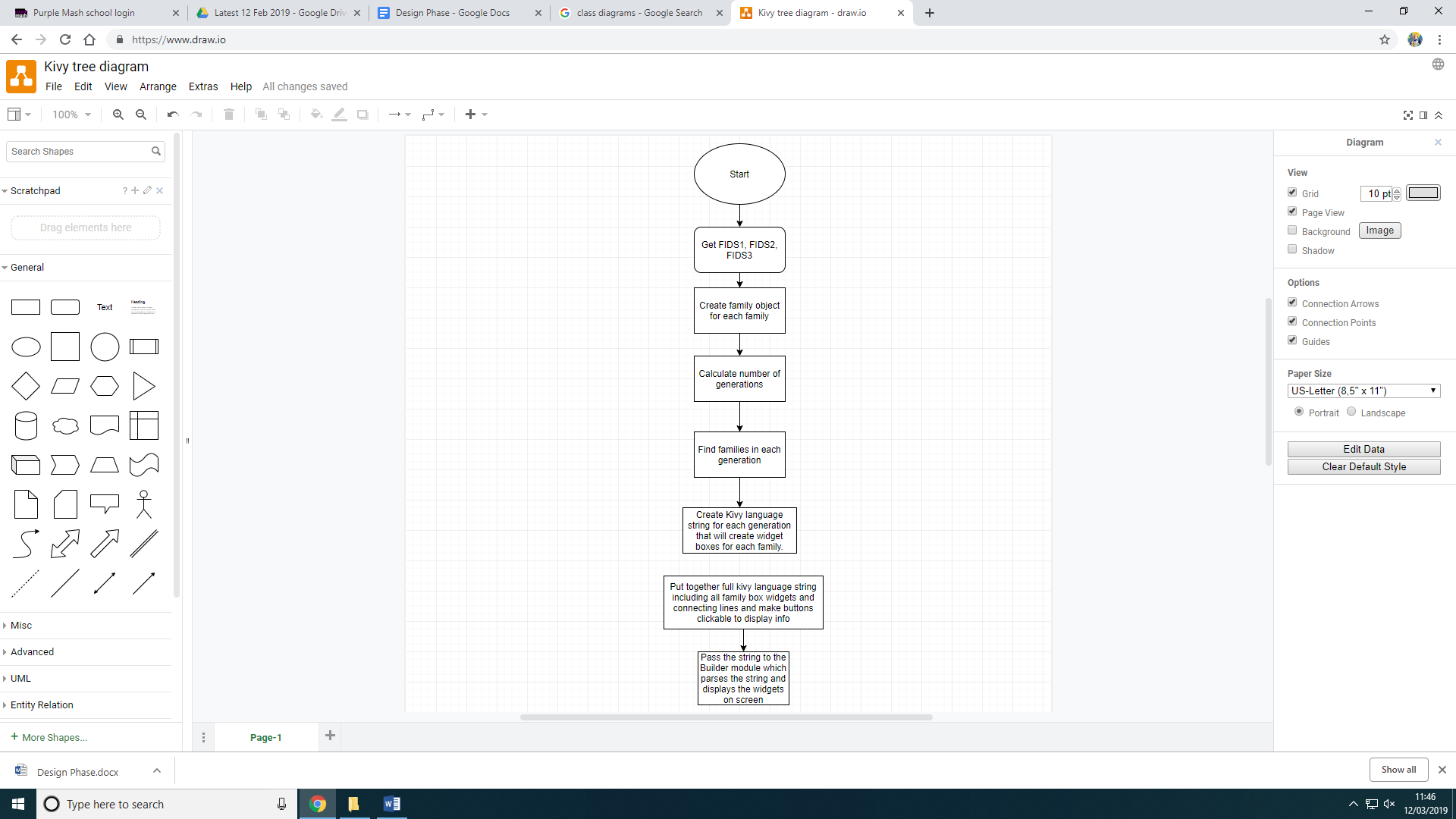
**Other information**

For the Kivy section of my project, I needed to pass into the program certain values These were:

* FIDS1: a list that contains, [FamilyID, Generation No, MotherID, FatherID, [Child ID, ChildID…]
* FIDS2: dictionary of personal info for every person, so that I do not have to keep on making many SQL queries.
* FIDS3: nested lists (from the family\_matrix array), [1 or -1, linking person, upper family, lower family]

Some of the global variables that I used were **families** (stores family objects once instantiated), **gens** (a list of generations where the generations that have been sorted, so that the lowest generation is 0), **sib\_generation** (a list of sibling families in the generation) and **fid\_done** (a list of families that have been added).

**Flow chart**



**Getting relationships**

In this option, the main subroutines are Traverse (slightly different to the previous Traverse algorithm), find\_route\_between\_people and people\_in\_route and route\_with\_relationships and path\_edit.

Traverse

This ‘Traverse’ algorithm in this option is very similar to the previous one. But the main difference is that it does not go through and find relationships between every family. The most important part of it is the stack. Here are the steps (I will use DK1485 and ZP1782 as examples of PersonID’s:

1. Find the families that DK1485 and ZP1782 are children in. For the purposes of this, let’s say the families are FID11 and FID3 respectively.
2. The Traverse algorithm starts at FID1, and finds other families from that starting family. Have variables foundP and foundR, which will be turned from False to True when FID11 and FID3 are about to be added onto the stack used for the traversal.
3. Carry out the traverse algorithm as before but, do not append anything to the family matrix, as the family matrix already exists.
4. When foundP and foundR, the value of the stack at each point where they become true is stored as person\_stack and family\_stack respectively. Once the family the algorithm will have reached its base case, so it will start to unwind and eventually return the person\_stack and relative\_stack variables.

The theory that this works on is the fact that if I get the route from each family to FID1, then the 2 routes must share some FID’s, whether that be all but one of the items in each stack or just FID1.

Find\_route\_between\_people()

This will have the 2 stacks passed in as a parameter. Let’s say person\_stack is [FID1,FID8,FID11] and relative\_stack be [FID1,FID8,FID9,FID3].

Because I find the route from the DK1485 to ZP1782 (from the person to the relative), I will create a new list called ‘route’, that will consist of person\_stack in reverse order followed by relative\_stack. For example, in this case route would be [FID11,FID8,FID1,FID1,FID8,FID9,FID3].

Then, I get rid of the repeats. I will start and the middle (where the person\_stack elements change to the relative\_stack elements) and start to remove the repeats. For example, FID1’s would be deleted but, both FID8’s would only be deleted if there is a connection between FID11 and FID9. If there is not, then only one of the FID8’s is deleted.

People\_in\_route

Given the route, I can then exchange the route for actual people, by looking at the connecting person between families in the family matrix. But, the PersonID’s themselves are not just added. Appended to each PersonID is “\_A/C” and “\_M/F”. A/C is determined

by whether the next family is above the previous family (\_A in this case and \_C if the next family is below the previous family. M/F is determined by whether the person is Male (M) or Female (F). This is done between each pair of consecutive families, and appended to a list. So, let’s say that the list is [DK1583, SK1782\_A\_F,RA9234\_C\_F, ZP1782\_A\_M]

Route\_with\_relationships

From the list above, relationships are then found, and attached to a string called ‘path’. For example, SK1782 is DK1583’s mother and so path would start off as ‘Dariyan Khan’s mother’s. Then, carrying on in this fashion, ‘sister’s would be appended to the list and then ‘husband’s’ would be appended. So, path would be ‘Dariyan Khan’s mother’s sister’s husband is Zoran’.

Path\_Edit

Then, some of the relationships are changed to more concise ones. For example, “Mother’s brother” would be changed to “uncle” etc. So, path would be changed to “Auntie’s husband” and then “Auntie’s Husband” would be changed to “Uncle’s”.

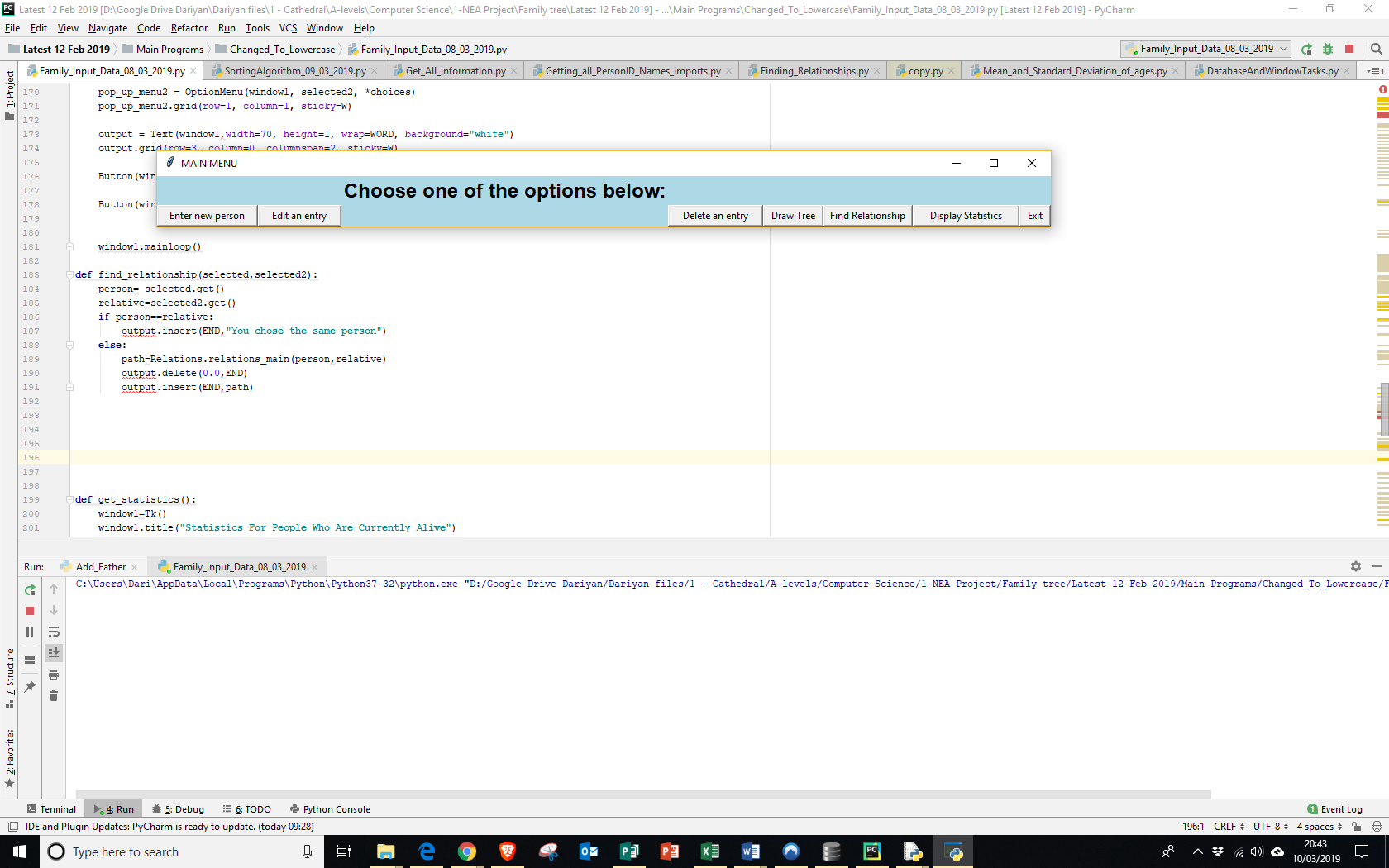
**Displaying Statistics**

After finding the ages of everyone alive, the following statistics will be calculated:

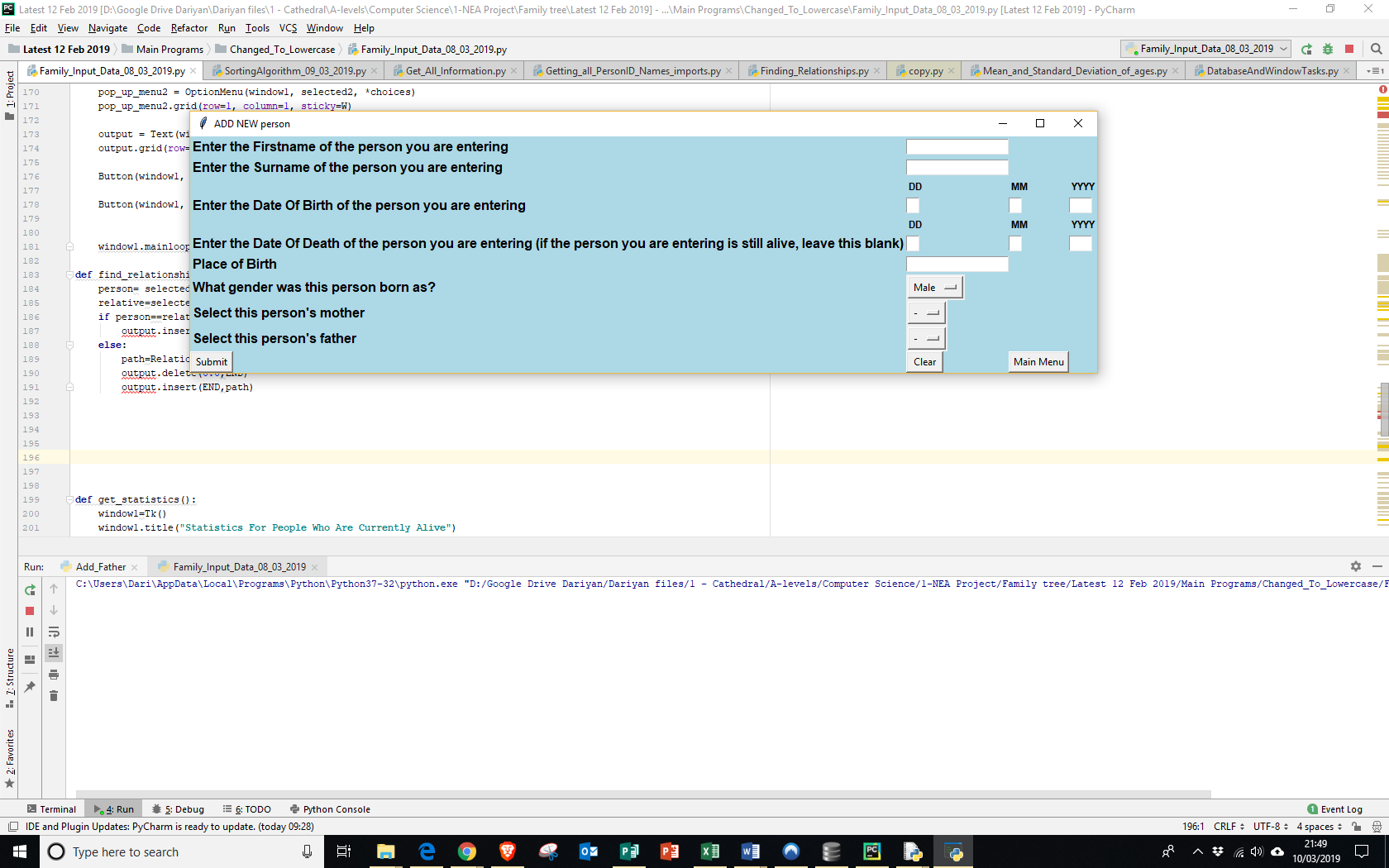
|  |  |
| --- | --- |
| Statistic | How it is worked out |
| Mean | Total age of everyone/number of people |
| Median | Put everyone’s age into ascending order using ‘sort’ and then find the middle value or the average of the 2 middle values |
| Mode | The most common age. |
| Max age | The last value in the list of ages sorted into ascending order. |
| Min age | The first value in the list of ages sorted into ascending order. |
| Variance (A-Level maths only) | Square each age, and add them together. Then divide this number by the number of ages. After this subtract the mean squared. |
| Standard deviation (A-Level maths only) | The square root of the variance |

**Interfaces**

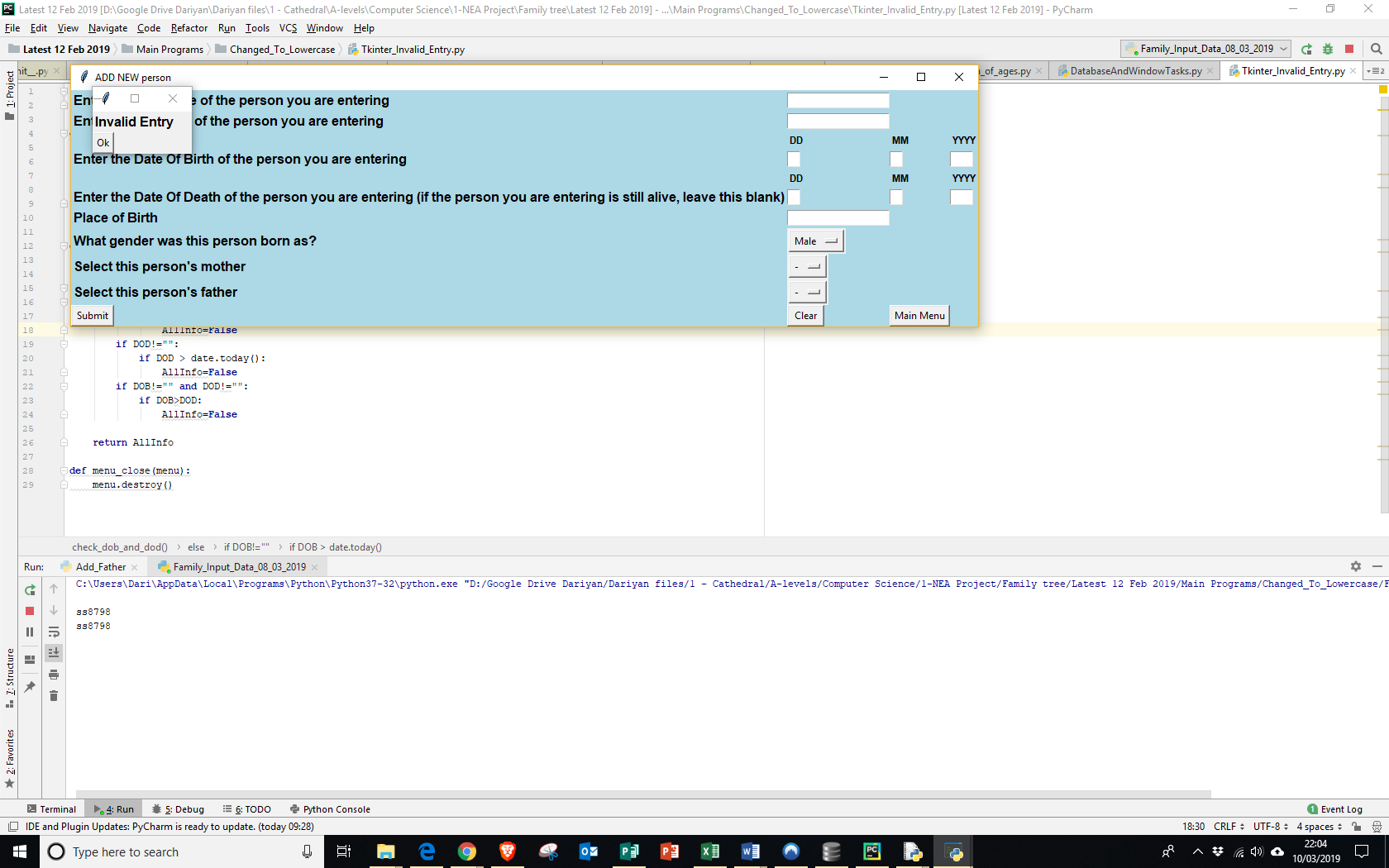
Main Menu



* Title = Main Menu
* Enter new person – displays empty form which you can then enter information into
* Edit an entry – Displays a drop-down menu where you can choose a person to edit
* Delete an entry – displays a drop-down menu where you can choose a person to delete.
* Draw tree – opens up a ‘Kivy’ window which will display the drawn tree
* Find relationship – presents 2 drop-down menus and an output box where the information is inputted
* Display statistics – Displays statistics about the ages of people in the family.
* Exit – exit the program

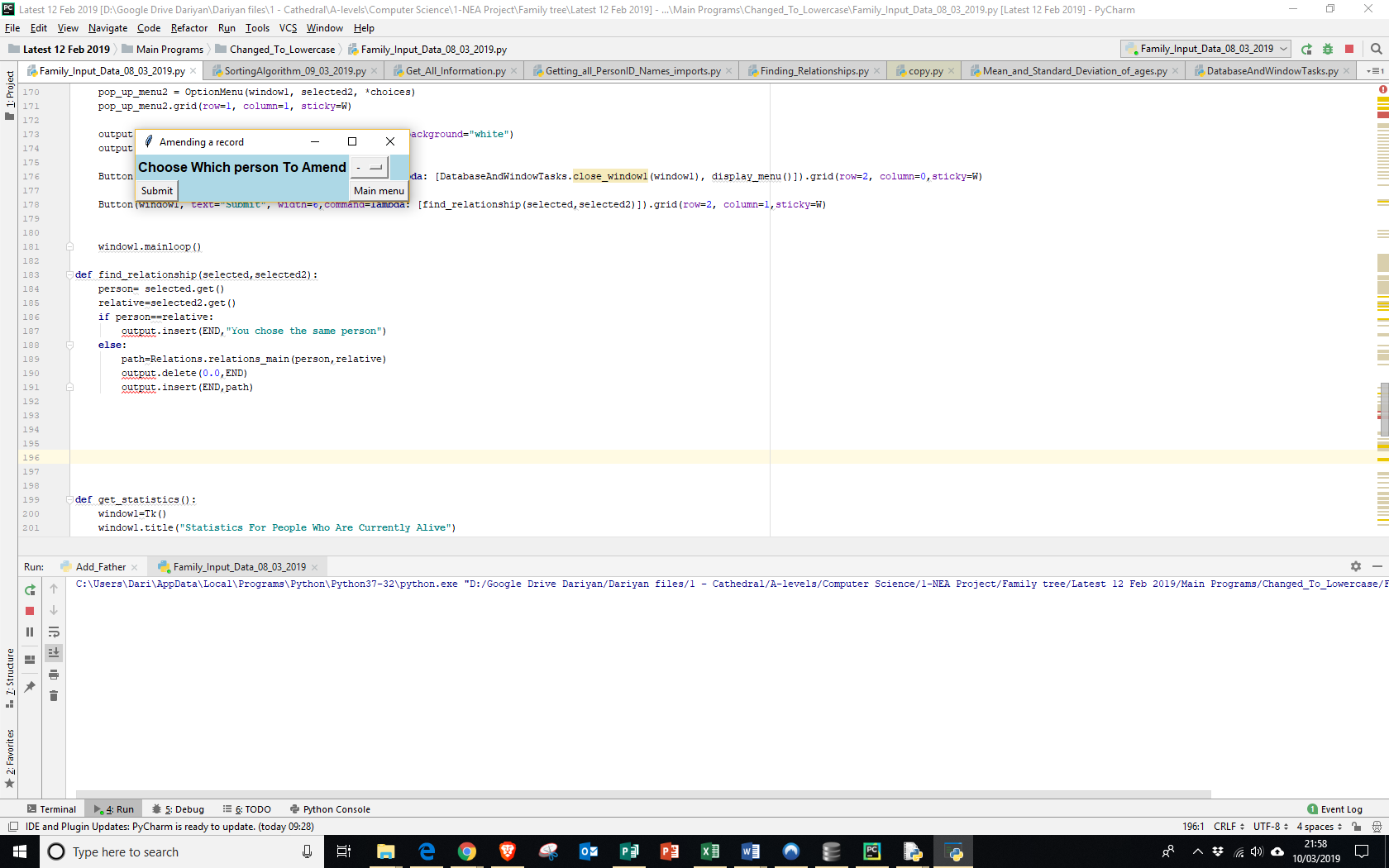
Enter new person

* Title = add new person
* The text boxes can take any value (apart from nothing for the surname and surname) and apart from the ones involving the dates of birth and death, where the days, moth and year must actually be a feasible date (so no float values, letters, special characters and dates such as 30/02/2019)
* Gender is a drop-down menu with either male or female, while select mother and father are also drop-down menus containing everyone female and male respectively.
* The submit button enters the data into the database.
* The clear button deletes everything in the text boxes and drop-down menus.
* Main menu takes you back to the main menu

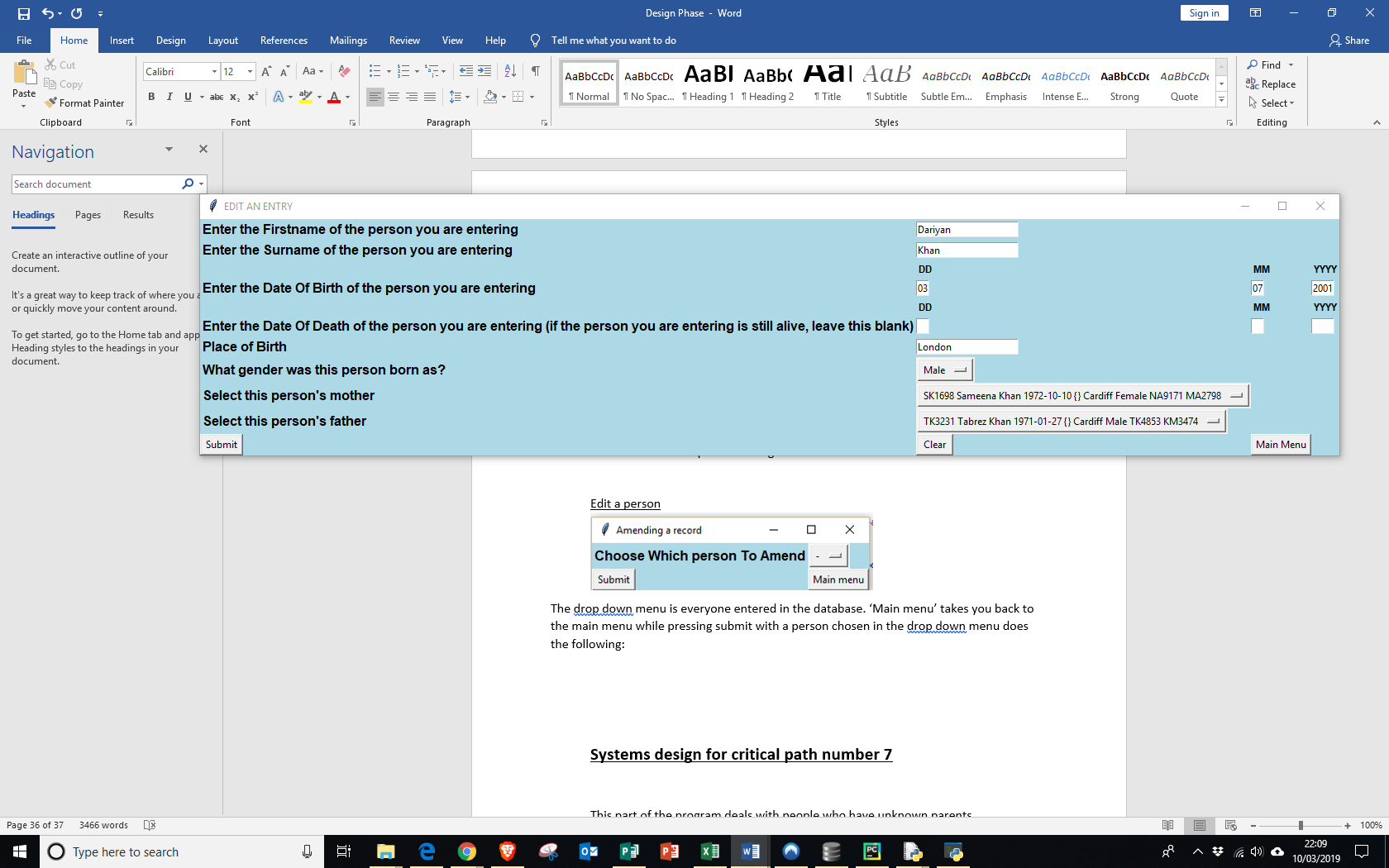
If a user enters invalid data and presses submit, all information in the text-boxes etc. is cleared (and not entered into the database) and the following is displayed:  


The user must then press ‘Ok’ to get rid of it.

Edit a person



The drop-down menu is everyone entered in the database. ‘Main menu’ takes you back to the main menu while pressing submit with a person chosen in the drop-down menu does the following:

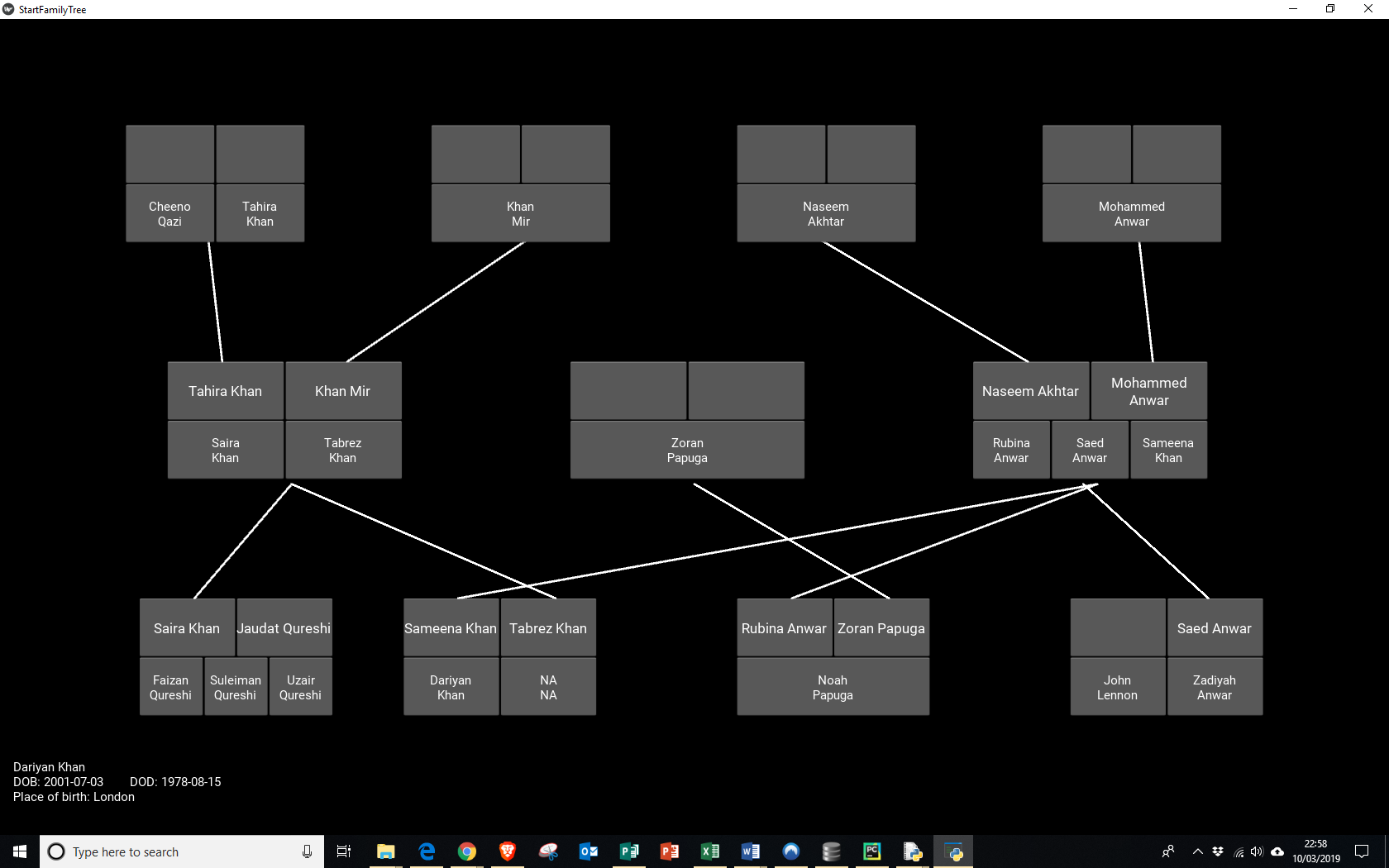


If I chose ‘Dariyan Khan’ on the previous screen, my information, in the style of the form in ‘Add new person’ would be displayed, so that the user could easily amend multiple information for one person.

Delete Person

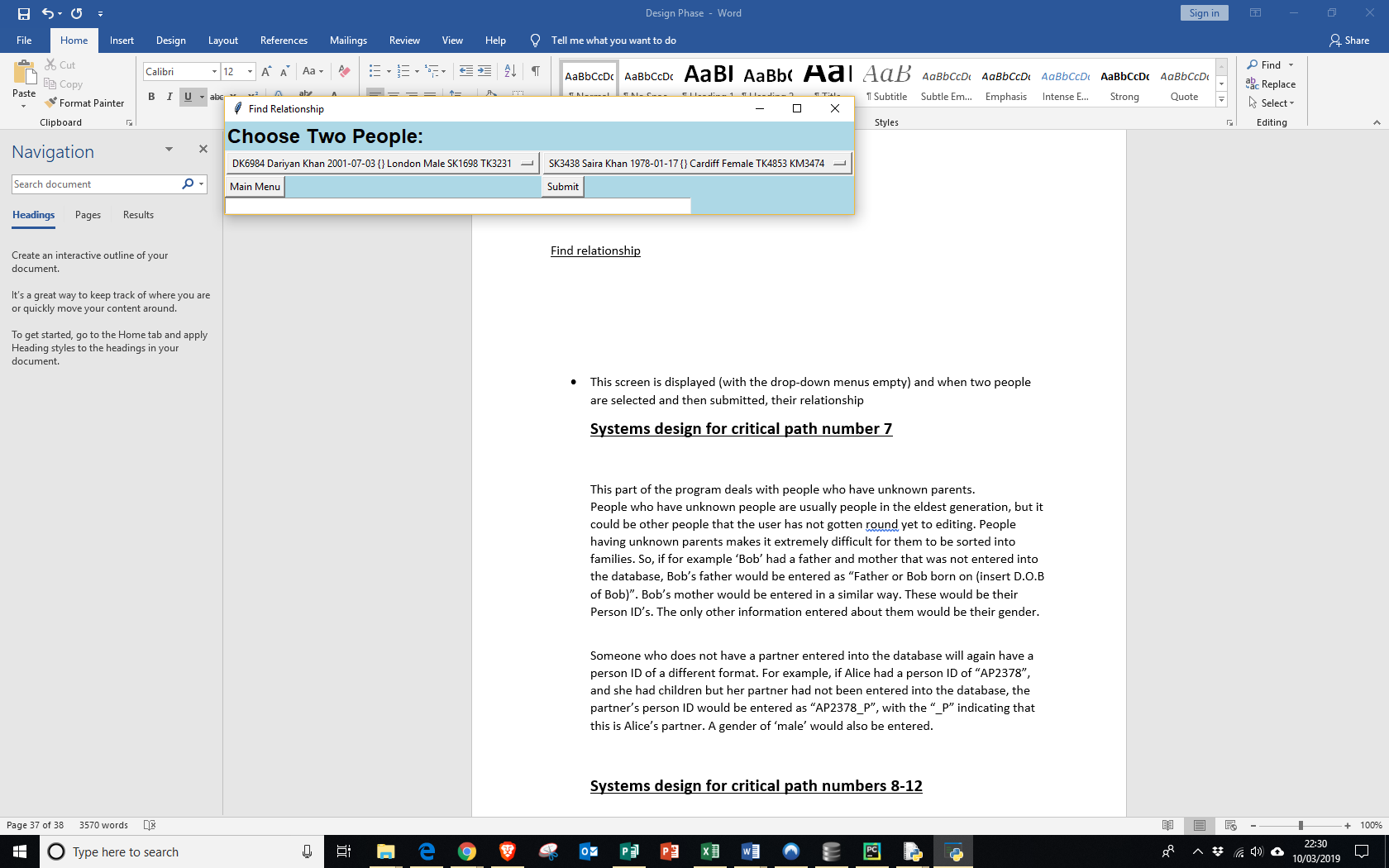
The same screen as the one that the user first encounters when ‘editing a person’ is displayed. But, instead of displaying another person when pressing submit, the user will be taken back to the main menu as the person is being deleted.

Draw tree



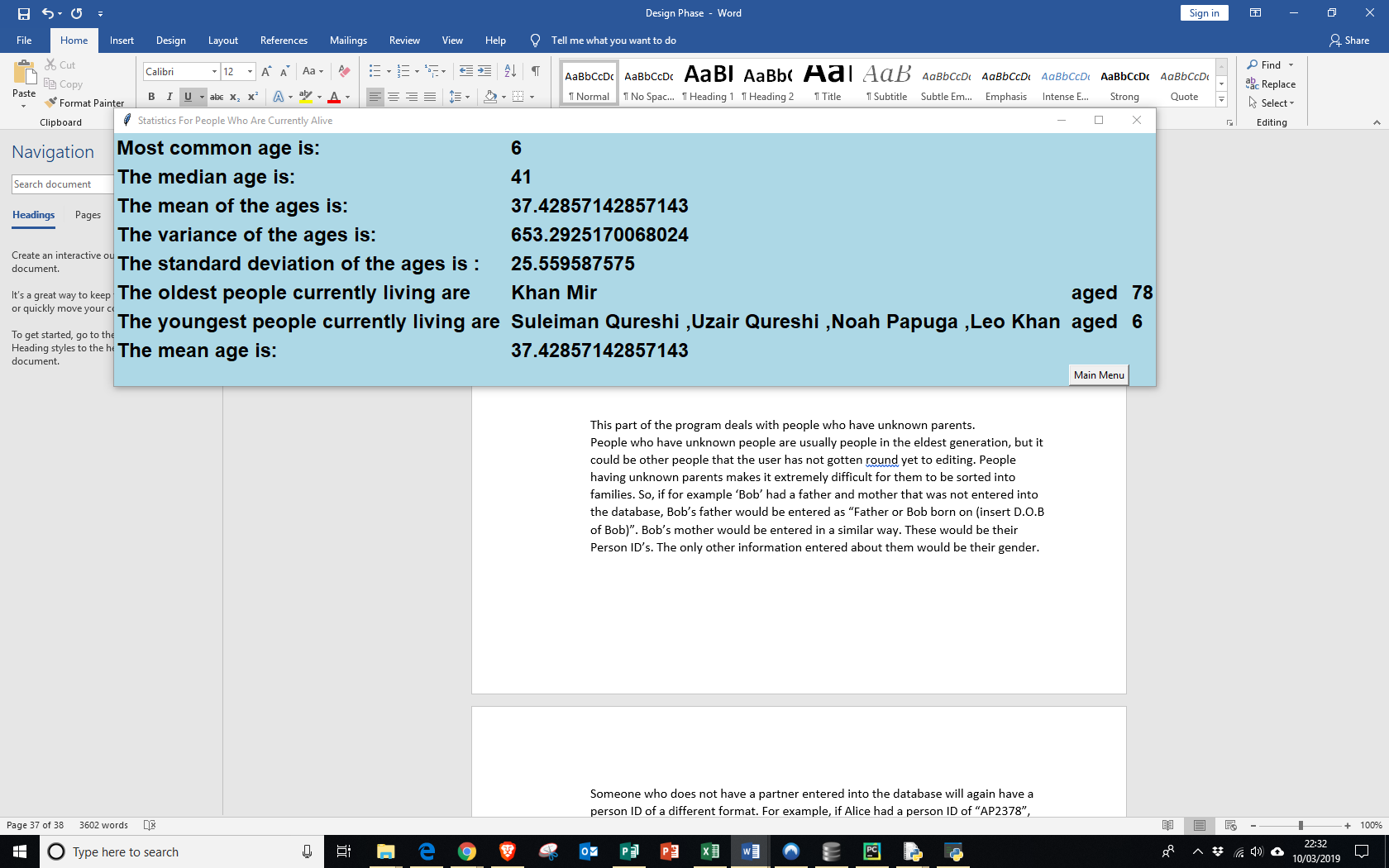
* When you click ‘draw tree’ a family tree such as the one above would be displayed (although the version above still has a few issues I have to work out, the final version will look very similar).
* Each line drawn connects a person to a family directly above them.
* Lines are drawn between connected families.
* When you click on a person, the information of that person is displayed at the bottom of the tree.

Find relationship



* This screen is displayed (with the drop-down menus empty) and when two people are selected and then submitted, their relationship will be displayed in the white strip underneath. So, for example in this case upon hitting submit, the white box would contain the text “Dariyan Khan’s auntie is Saira Khan”.

Display Statistics



* Upon hitting the display statistics option, the information of people currently alive will be displayed in the above format.

**Systems design for critical path number 7**

This part of the program deals with people who have unknown parents.  
People who have unknown people are usually people in the eldest generation, but it could be other people that the user has not gotten around yet to editing. People having unknown parents makes it extremely difficult for them to be sorted into families. So, if for example ‘Bob’ had a father and mother that was not entered into the database, Bob’s father would be entered as “Father or Bob born on (insert D.O.B of Bob)”. Bob’s mother would be entered in a similar way. These would be their Person ID’s. The only other information entered about them would be their gender.

Someone who does not have a partner entered into the database will again have a person ID of a different format. For example, if Alice had a person ID of “AP2378”, and she had children but her partner had not been entered into the database, the partner’s person ID would be entered as “AP2378\_P”, with the “\_P” indicating that this is Alice’s partner. A gender of ‘male’ would also be entered.

**Data Structures**

**Adjacency matrix or Adjacency List**

I had to decide how I was going to represent my graph. Should I use an adjacency list or an adjacency matrix? A list has the benefit that there would be less wasted space as an adjacency matrix representing a family is likely to be quite sparse. However, a matrix would allow new entries to be added easily, and relationships between people could be tested quicker. Therefore, in the end, I decided to go with an adjacency matrix as I felt that being able to efficiently search and add new nodes to the matrix would be more beneficial for further functionalities in the program.

The following data structures will be used:

|  |  |  |
| --- | --- | --- |
| Data Structure (notable subroutines used in) | Implementation and sample data | Functions |
| Array  Traverse (to create matrix)  DescendantNumber()  Traverse (to find relations) | This will be a nxn array, where ‘n’ is the number of families in the database (i.e. rows in tblSingleFamily). The matrix will consist of one list big list, which contains ‘n’ smaller lists, and then each of these lists would contain ‘n’ lists. Sample data (which would be held in the smallest sub-division of lists) would be similar to -1, DK1983, FID11, FID5 . This would be in the 4th small list in the 10th bigger list (4 and 10 instead of 11 and 5 because indexes start at 0)  -1= FID11 is below FID5  DK1983 = person common to both families.  FID11, FID5 shows that we are dealing with the 10th row and 5th column of the array. | The only functionality that the array will have will be to append information to a list, or retrieve information from a list. |
| Queue  Families() | The queue is implemented as a list and will contain ‘FID1’ at index[0], ‘FID2’ at index[1], ‘FID3’ at index[2], and so on up to about ‘FID50’ or so | You will only be able to dequeue items |
| Stack | The stack is also implemented as a list, and will dynamically change during the execution of a subroutine. It may look something like this:  Index[0] = ‘FID1’ Index[1] = ‘FID8’  Index[2] = ‘FID5’  It only really be used in depth first searches and similar processes. It will contain the family Id’s visited in order to get to a certain family | Push = pushes an item onto the stack  Pop = removes and returns the topmost item from the stack  Peek = returns topmost item of the stack |

**TECHNICAL SOLUTION**

**Main Program**

Imports:  
sqlite3: Allows database operations  
tkinter: allows GUI to be made  
DatabaseAndWindowTasks: contains database and interface creation operations  
Add\_Mother & Add\_Father: Prompts user to add mother/father for people before drawing tree  
Finding\_Relations:Allows relationships to be found  
SortingAlgorithm\_09\_03\_2019: Creates Family\_matrix, descendant number etc.  
KivyFamilyTree: Code to create information for Kivy (this import causes Kivy window to open as soon as program is run  
Get\_All\_information: Gets more information for Kivy section

**import** sqlite3  
**from** tkinter **import** \*  
**import** DatabaseAndWindowTasks **as** DatabaseAndWindowTasks  
**import** Mean\_and\_Standard\_Deviation\_of\_ages **as** Stats  
**import** Add\_Mother **as** Add\_Mother  
**import** Add\_Father **as** Add\_Father  
**import** Finding\_Relationships **as** Relations  
**import** SortingAlgorithm\_09\_03\_2019 **as** Sort  
**import** KivyFamilyTree **as** kv  
**import** Get\_All\_Information **as** GetAllInfo  
  
**def** display\_menu():  
 window = Tk()  
 window.title(**"MAIN MENU"**)  
 window.configure(background=**"Light blue"**)  
  
 *# Label for text saying what to do* Label (window, text=**"Choose one of the options below:"**, bg=**"Light blue"**, fg=**"black"**, font = **"none 18 bold"**) .grid(row=0, column=3, sticky=W)  
  
 *#Button for Enter New Person* Button(window, text=**"Enter new person"**, width=16, command=**lambda**:[close\_window(window),enter\_new\_person()]) .grid(row=50, column=0, sticky=W)  
  
 *# Button for Edit entry* Button(window, text=**"Edit an entry"**, width = 13, command=**lambda**:[close\_window(window), edit\_person()]) .grid(row=50, column=2, sticky=W)  
  
 *#Button for Delete entry* Button(window, text=**"Delete an entry"**, width = 15, command=**lambda**:[close\_window(window), delete\_person()]) .grid(row=50, column=4, sticky=W)  
  
 *#Button for Exit* Button(window, text=**"Exit"**, width=4, command=**lambda**:[close\_window(window),option9()]) .grid(row=50, column=14, sticky=W)  
  
 *#Button for draw tree* Button(window, text=**"Draw Tree"**, width=9, command=**lambda**:[close\_window(window), draw\_tree()]) .grid(row=50, column=8, sticky=W)  
  
 *#Button for get relationship* Button(window, text=**"Find Relationship"**, width=14, command=**lambda**:[close\_window(window), get\_relationship()]).grid(row=50, column=10, sticky=W)  
  
 *#Button for statistics* Button(window, text=**"Display Statistics"**, width=17, command=**lambda**:[close\_window(window), get\_statistics()]).grid(row=50, column=12, sticky=W)  
  
 window.mainloop()  
  
**def** close\_window(window):   
 window.destroy()  
   
   
 *#return window***def** option9():  
 exit()  
*#=================Enters person into database===================================***def** add\_new\_person():  
 file\_exists=DatabaseAndWindowTasks.check\_file\_exists()  
 **if** file\_exists==**False**:  
 DatabaseAndWindowTasks.create\_database() *### Checks to see if database exists, before trying to enter info into it* **else**:  
 DatabaseAndWindowTasks.create\_and\_enter\_form(**"ADD NEW person"**,**"1"**, **"NONE"**)  
 display\_menu()  
  
*#=================Edit and delete person options=====================================================================================***def** edit\_person():  
 file\_exists = DatabaseAndWindowTasks.check\_file\_exists()  
 **if** file\_exists == **False**:  
 add\_new\_person()  
 **else**:  
 choose\_person(**"Choose Which person To Amend"**, **"2"**)  
  
  
**def** delete\_person():  
 choose\_person(**"Choose Which person To Delete"**, **"3"**)  
  
**def** close\_window3(window3):  
 window3.destroy()  
  
**def** change\_info(person):  
 DatabaseAndWindowTasks.create\_and\_enter\_form(**"EDIT AN ENTRY"**,**"2"**,person)  
  
**def** delete\_person(person):  
 conn = sqlite3.connect(**"Family\_1.db"**)  
 print(person)  
 **with** conn:  
 cursor = conn.cursor()  
 print(person)  
 keyfield = **"'"** + person + **"'"** cursor.execute(**"DELETE FROM tblFamily WHERE personID ="** + keyfield)  
 conn.commit()  
 update\_after\_deletion(person)  
 conn.commit()  
  
  
  
  
**def** update\_after\_deletion(person): *# Account for where person was a mother or a father* **with** sqlite3.connect(**"Family\_1.db"**) **as** conn:  
 cursor=conn.cursor()  
 keyfield = **'"'** + person + **'"'** field=**'"'** + **""** + **'"'** cursor.execute(**"UPDATE tblFamily SET MotherID ="** + field + **"WHERE MotherID ="** + keyfield)  
 cursor.execute(**"UPDATE tblFamily SET FatherID="** + field + **"WHERE FatherID="** + keyfield)  
 conn.commit()  
  
  
  
  
  
  
**def** choose\_person(message,option\_num):  
 **with** sqlite3.connect(**"Family\_1.db"**) **as** conn:  
 cursor = conn.cursor()  
  
 window3=Tk()  
 window3.title(**"Amending a record"**)  
 window3.configure(background=**"Light blue"**)  
 *# label for choosing row to edit* Label(window3, text=message, bg=**"Light blue"**, fg=**"black"**, font =**"none 12 bold"**) .grid(row=0,column=0, sticky=W)  
 tkvar\_edit= StringVar(window3)  
 choices\_edit=[]  
 **for** row **in** cursor.execute(**"""SELECT \* FROM tblFamily WHERE PersonID NOT LIKE '%Mother%'  
 AND PersonID NOT LIKE '%Father%'  
 AND PersonID NOT LIKE '%\_P%' """**): *### Gets everyone in the database apart from a select few for the choices* choices\_edit.append(row)  
  
 tkvar\_edit.set(**'-'**)  
 popup\_menu\_edit=OptionMenu(window3, tkvar\_edit, \*choices\_edit)  
 popup\_menu\_edit.grid(row=0, column=1, sticky=W)  
  
 **if** option\_num==**"2"**: *#### Edit person* Button(window3, text=**"Submit"**, width=6, command=**lambda**:[close\_window3(window3), change\_info(tkvar\_edit.get()[2:8])]) .grid(row=3, column=0, sticky=W)  
  
 **elif** option\_num==**"3"**: *# Delete person* Button(window3, text=**"Submit"**, width=6, command=**lambda**:[close\_window3(window3), delete\_person(tkvar\_edit.get()[2:8]),display\_menu()]) .grid(row=3, column=0, sticky=W)  
 Button(window3, text=**"Main menu"**, width=9, command=**lambda**:[close\_window3(window3), display\_menu()]) .grid(row=3, column=1, sticky=W)  
  
**def** close\_window3(window3):  
 window3.destroy()  
*#==========================================================================================================================================================================================***def** draw\_tree():  
 Add\_Mother.add\_mother\_main()  
 Add\_Father.add\_father\_main()entire\_families, family\_matrix, children\_dict, kivy\_matrix=Sort.sorting\_main()  
 everything\_connected\_dict=GetAllInfo.get\_everything\_connected()  
 ft = kv.StartFamilyTreeApp()  
 ft.fids1 = entire\_families  
 ft.fids2 = everything\_connected\_dict  
 ft.fids3 = kivy\_matrix  
 ft.run()  
display\_menu()  
  
  
  
  
  
**def** get\_relationship():  
 conn = sqlite3.connect(**"Family\_1.db"**)  
 cursor = conn.cursor()  
 Add\_Mother.add\_mother\_main()  
 Add\_Father.add\_father\_main()  
 window1=Tk()  
 window1.title(**"Find Relationship"**)  
 window1.configure(background=**"Light blue"**)  
 Label(window1, text=**"Choose Two People:"**, bg=**"Light blue"**, fg=**"black"**, font=**"none 18 bold"**).grid(row=0, column=0,sticky=W)  
  
 choices = []  
 selected = StringVar(window1)  
 selected2 = StringVar(window1)  
 selected.set(**'-'**)  
 **for** row **in** cursor.execute(**"""SELECT PersonID, Firstname, Surname, DOB, DOD, Gender FROM tblFamily WHERE PersonID NOT LIKE '%Mother%' and**

Lambda here for example, means that the window is closed first, before option1 is run.

Destroys window widget

Exits from the program

Checks to see if Family\_1.db exists. If it does not, it is created along with tblFamily

Creates and displays form for entering new person

Checks if database exists again but, if it does not exist, the “Enter Person into database subroutine happens

Calls subroutine that allows user to choose a person to edit

Calls subroutine that allows user to choose a person to delete

Calls same function as “enter\_new\_person”

Deletes specified ‘person’ from the database

If the person chosen to be deleted was also a parent, the Mother/Father IDs of these people are set to “” (i.e. empty)

Creates window where user can choose a person to edit or delete.

Gets everyone in the database that effectively have 6-character long PersonID’s

Creates drop-down menu

If statement, that either calls change\_info or delete\_person, depending on whether user wants to edit or delete person

Creates a series of drop-down menus and asks user if they want to add Mother or Father IDs for some people, in order to make the family tree more comprehensive when drawn out.

Entire\_families, families\_matrix and everything\_connected\_dict which are needed to draw tree

Calls Kivy family tree program, that draws the tree. Once the ‘x’ button is clicked on the window which the tree is displayed in, user is taken back to the main menu.

**PersonID NOT LIKE '%Father%' AND PersonID NOT LIKE '%\_P%' """**): *# Gets choices for drop-down menu* choices.append(row)  
 pop\_up\_menu = OptionMenu(window1, selected, \*choices)  
 pop\_up\_menu.grid(row=1, column=0, sticky=W)  
  
 pop\_up\_menu2 = OptionMenu(window1, selected2, \*choices)  
 pop\_up\_menu2.grid(row=1, column=1, sticky=W)  
  
 output = Text(window1,width=140, height=1, wrap=WORD, background=**"white"**) *# Gets output box* output.grid(row=3, column=0, columnspan=2, sticky=W)  
  
 Button(window1, text=**"Main Menu"**, width=9,command=**lambda**: [DatabaseAndWindowTasks.close\_window1(window1), display\_menu()]).grid(row=2, column=0,sticky=W)  
  
 Button(window1, text=**"Submit"**, width=6,command=**lambda**: [find\_relationship(selected,selected2,output)]).grid(row=2, column=1,sticky=W)  
  
  
 window1.mainloop()  
  
**def** find\_relationship(selected,selected2,output):  
  
 entire\_families, family\_matrix,children\_dict, kivy\_matrix=Sort.sorting\_main()  
 person= selected.get()  
 relative=selected2.get()  
 all\_person\_id=DatabaseAndWindowTasks.get\_person\_id()  
  
 **for** x **in** range(0,len(all\_person\_id)):  
 **if** all\_person\_id[x][0] **in** person: *# Gets personID of person from drop-down menu selection* person = all\_person\_id[x][0]  
  
 **if** all\_person\_id[x][0] **in** relative: *#Gets personID for relative from drop-down menu selection* relative = all\_person\_id[x][0]  
  
 **if** person==relative:  
 output.delete(0.0, END)  
 output.insert(END,**"You chose the same person"**)

Output box which relationship is displayed in

If person and relative are the children of the same family, program returns “The two people are siblings”

Gets personIDs from the strings of the chosen people in the drop-down menu

If the person chose the same person in both menus, “You chose the same person” is returned.

Gets a dictionary of every personID as the key, and a set containing their children as their value. The next two line of code get the person and relative from the drop-down menus, as well as producing the family matrix and a list of all the families.

Creates two drop down menus, which the user can then choose 2 people in, and then press submit to find the relationship between two people.

**try**:  
 person\_family\_id= DatabaseAndWindowTasks.get\_family\_id(person)  
 relative\_family\_id=DatabaseAndWindowTasks.get\_family\_id(relative) *# Checks to see if user chose the same person*

**except Exception**:

If you cannot get the familyID for the person or the relative, then it is assumed that there is no connection between them.

output.delete(0.0,END)  
 output.insert(END **"There is no connection between the 2 entered people"**)  
 **if** person\_family\_id == relative\_family\_id **and** person!=relative:  
 output.delete(0.0, END)  
 output.insert(END,**"The two people are siblings"**) *# Checks if the 2 people are in the same family i.e. siblings.  
  
 ## Checks if one is the parent of the other* **elif** relative **in** children\_dict[person]:  
 person\_firstname=DatabaseAndWindowTasks.get\_firstname(person)  
 person\_surname=DatabaseAndWindowTasks.get\_surname(person)  
  
 relative\_firstname=DatabaseAndWindowTasks.get\_firstname(relative)  
 relative\_surname=DatabaseAndWindowTasks.get\_surname(relative)  
 relative\_gender=DatabaseAndWindowTasks.get\_gender(relative)  
 **if** relative\_gender==**"Male"**:  
 child=**"Son"  
 else**:  
 child=**"Daughter"** output.delete(0.0, END)  
 output.insert(END,relative\_firstname + **" "** + relative\_surname + **" "** + **" is the"** + **" "** + child + **" "** + **"of"** + **" "** + person\_firstname + **" "** + person\_surname)  
  
 **elif** person **in** children\_dict[relative]:  
 person\_firstname=DatabaseAndWindowTasks.get\_firstname(person)  
 person\_surname=DatabaseAndWindowTasks.get\_surname(person)  
  
 relative\_firstname=DatabaseAndWindowTasks.get\_firstname(relative)  
 relative\_surname=DatabaseAndWindowTasks.get\_surname(relative)  
 person\_gender=DatabaseAndWindowTasks.get\_gender(person)  
 **if** person\_gender==**"Male"**:  
 child=**"Son"  
 else**:  
 child=**"Daughter"** output.delete(0.0, END)  
 output.insert(END,person\_firstname + **" "** + person\_surname + **" "** + **"is the"** + **" "** + child + **" "** + **"of"** + **" "** + relative\_firstname + **" "** + relative\_surname)  
  
  
 **else**:

If the 2 people fit into none of the categories above, the relationship is determined by calling relations\_main

Checks to see if either person or relative is the child of the other, by checking if ‘person’ is part of the set values that the person/relative key is in the database. It then determines if it is a son/daughter relationship, by working out the gender of the child.

**try**:  
 path=Relations.relations\_main(person,relative)  
 output.delete(0.0,END)  
 output.insert(END,path)

**except Exception**:  
 output.delete(0.0,END)  
 output.insert(END **"There is no connection between the 2 entered people"**)  
  
  
  
  
  
  
  
**def** get\_statistics():  
  
 window1=Tk()  
 window1.title(**"Statistics For People Who Are Currently Alive"**)  
 window1.configure(background=**"Light blue"**)  
 mode,median, mean\_age,variance,standard\_deviation, max\_age\_people,max\_age\_people\_list,min\_age\_people,min\_age\_people\_list=Stats.statistics\_main()  
  
 max\_age\_string=**""  
 for** age in max\_age\_people\_list:  
 personF=DatabaseAndWindowTasks.get\_firstname(age)  
 personS=DatabaseAndWindowTasks.get\_surname(age)  
 max\_age\_string=max\_age\_string + personF + **" "** + personS + **" "  
 if** x!= int(len(max\_age\_people\_list)-1):  
 max\_age\_string = max\_age\_string + **","** *# Gets sentence for max\_age* min\_age\_string = **""  
 for** age **in** min\_age\_people\_list:  
 personF = DatabaseAndWindowTasks.get\_firstname(age)  
 personS = DatabaseAndWindowTasks.get\_surname(age)  
 min\_age\_string = min\_age\_string + personF + **" "** + personS + **" "  
 if** x != int(len(min\_age\_people\_list) - 1):  
 min\_age\_string = min\_age\_string + **","** *### gets sentence for min\_age  
  
  
 #mode* Label(window1, text=**"Most common age is:"**, bg=**"Light blue"**, fg=**"black"**, font=**"none 18 bold"**).grid(row=0,column=0,sticky=W)  
 Label(window1, text=mode, bg=**"Light blue"**, fg=**"black"**, font=**"none 18 bold"**).grid(row=0, column=1,sticky=W)  
  
 *#median* Label(window1, text=**"The median age is:"**, bg=**"Light blue"**, fg=**"black"**, font=**"none 18 bold"**).grid(row=1, column=0,sticky=W)  
 Label(window1, text=median, bg=**"Light blue"**, fg=**"black"**, font=**"none 18 bold"**).grid(row=1, column=1, sticky=W)  
  
 *#mean* Label(window1, text=**"The mean of the ages is:"**, bg=**"Light blue"**, fg=**"black"**, font=**"none 18 bold"**).grid(row=2, column=0,sticky=W)  
 Label(window1, text=mean\_age, bg=**"Light blue"**, fg=**"black"**, font=**"none 18 bold"**).grid(row=2, column=1, sticky=W)  
  
 *#variance* Label(window1, text=**"The variance of the ages is:"**, bg=**"Light blue"**, fg=**"black"**, font=**"none 18 bold"**).grid(row=3, column=0,sticky=W)  
 Label(window1, text=variance, bg=**"Light blue"**, fg=**"black"**, font=**"none 18 bold"**).grid(row=3, column=1, sticky=W)  
  
 *#standard deviation* Label(window1, text=**"The standard deviation of the ages is :"**, bg=**"Light blue"**, fg=**"black"**, font=**"none 18 bold"**).grid(row=4, column=0,sticky=W)  
 Label(window1, text=standard\_deviation, bg=**"Light blue"**, fg=**"black"**, font=**"none 18 bold"**).grid(row=4, column=1, sticky=W)  
  
 *#max\_age* Label(window1, text=**"The oldest people currently living are "**, bg=**"Light blue"**, fg=**"black"**, font=**"none 18 bold"**).grid(row=5, column=0,sticky=W)  
 Label(window1, text=max\_age\_string, bg=**"Light blue"**, fg=**"black"**, font=**"none 18 bold"**).grid(row=5, column=1, sticky=W)  
 Label(window1, text=**"aged"**, bg=**"Light blue"**, fg=**"black"**, font=**"none 18 bold"**).grid(row=5, column=2,sticky=W)  
 Label(window1, text=max\_age\_people, bg=**"Light blue"**, fg=**"black"**, font=**"none 18 bold"**).grid(row=5, column=3, sticky=W)  
  
 *#min age* Label(window1, text=**"The youngest people currently living are "**, bg=**"Light blue"**, fg=**"black"**,font=**"none 18 bold"**).grid(row=6, column=0, sticky=W)  
 Label(window1, text=min\_age\_string, bg=**"Light blue"**, fg=**"black"**, font=**"none 18 bold"**).grid(row=6, column=1,sticky=W)  
 Label(window1, text=**"aged"**, bg=**"Light blue"**, fg=**"black"**, font=**"none 18 bold"**).grid(row=6, column=2, sticky=W)  
 Label(window1, text=min\_age\_people, bg=**"Light blue"**, fg=**"black"**, font=**"none 18 bold"**).grid(row=6, column=3,sticky=W)  
  
  
 Button(window1, text=**"Main Menu"**, width=9, command=**lambda**:[DatabaseAndWindowTasks.close\_window1(window1), display\_menu()]) .grid(row=10, column=2, sticky=W)  
  
 window1.mainloop()  
  
*# displays menu*display\_menu()

If an error occurs when trying to get the relationship between 2 pepl then it must be because the 2 people are not related.

Same process repeated for people who are the youngest

Gets mean, median, mode, variance, standard deviation, max age and min age of people currently living in the database, including list of people who are the youngest or oldest.

For people who are the oldest, gets their full name

Puts names and their common age into a sentence to be displayed

Calls main menu

**Database And Window Tasks**

**import** random  
**import** datetime  
**import** sqlite3  
**import** os.path  
**from** tkinter **import** \*  
**import** re  
**import** Tkinter\_Invalid\_Entry **as** CheckInvalid  
  
  
   
*#========================Database and file operations=============================================================================================================***def** check\_file\_exists():  
 file\_exists = os.path.exists(**'Family\_1.db'**)  
 **return** file\_exists  
  
**def** check\_parents\_add(): *#Gets all mothers and fathers in the databaseand determines whether there any parents not added* add\_mother=**False** add\_father=**False**  
 conn=sqlite3.connect(**"Family\_1.db"**)  
 cursor=conn.cursor()  
 cursor.execute(**"SELECT MotherID, FatherID FROM tblFamily"**)  
 all\_parents=cursor.fetchall()  
 **while** (add\_mother==**False** **or** add\_Father==**False**) **and** x<len(all\_parents):   
 **if** all\_parents[x][0]==**""**:  
 add\_mother=**True**  
 **if** all\_parents[x][1]==**""**:  
 add\_father=**True**

If the number of pairs of parents is not equal to the number of mothers, some people must not have mothers entered. Same process is used for fathers. Add\_mother/add\_father is returned as True if some mothers/fathers need to be added.

Gets all fathers in database

Gets all mothers in the database

Gets all pairs of parents in database

Imports:  
random: Allows random integers to be generated  
datetime: Allows digits to be put into a date form  
sqlite3: Allows database use  
tkinter: Allows the use of widgets etc. to make a GUI  
re: Allows the use of regular expressions  
Tkinter\_Invalid\_Entry: Checks If information entered into form is correct

Gets all mothers and fathers from tblFamily

If one person has their mother as “” (i.e. empty), then add\_mother is ‘True’. This means that there is a mother that needs to be added to the database/edited. Same for fathers as well.

x=x+1 **return** add\_mother, add\_father  
  
  
  
**def** create\_database():  
 file\_exists = check\_file\_exists()  
 **if** file\_exists == **True**:  
 **pass  
 else**:  
 connection = sqlite3.connect(**"Family\_1.db"**)  
 cursor = connection.cursor()  
 sql\_command = **"""  
  
 CREATE TABLE `tblFamily`  
 (  
 `PersonID` TEXT,  
 `Firstname` TEXT,  
 `Surname` TEXT,  
 `DOB` DATE,  
 `DOD` DATE,  
 `POB` TEXT,  
 'Gender' TEXT,  
 'MotherID' TEXT,  
 'FatherID' TEXT,  
 PRIMARY KEY(`PersonID`)  
 )"""** cursor.execute(sql\_command)  
 connection.commit()  
 connection.close()  
  
*#=========================Get information about people==========================================================================================================================***def** get\_firstname(person):  
 conn = sqlite3.connect(**"Family\_1.db"**)  
 cursor = conn.cursor()  
 sql = **"""  
 SELECT Firstname  
 FROM tblFamily  
 Where PersonID = ?  
 """  
 for** row **in** cursor.execute(sql, (person,)):  
 pre\_firstname = (row[0])  
  
 conn.commit()  
 conn.close()  
 **return** pre\_firstname  
  
**def** get\_surname(person):  
 conn = sqlite3.connect(**"Family\_1.db"**)  
 cursor = conn.cursor()  
 sql = **"""  
 SELECT Surname  
 FROM tblFamily  
 Where PersonID = ?  
 """  
 for** row **in** cursor.execute(sql, (person,)):  
 pre\_surname = (row[0])  
  
 conn.commit()  
 conn.close()  
 **return** pre\_surname  
  
**def** get\_dob(person):  
 conn = sqlite3.connect(**"Family\_1.db"**)  
 cursor = conn.cursor()  
 sql = **"""  
 SELECT DOB  
 FROM tblFamily  
 Where PersonID = ?  
 """  
 for** row **in** cursor.execute(sql, (person,)):  
 pre\_dob = (row[0])  
 pre\_dob\_dd = pre\_dob[8:10]  
 pre\_dob\_mm = pre\_dob[5:7]  
 pre\_dob\_yyyy = pre\_dob[0:4]  
  
 conn.commit()  
 conn.close()  
 **return** pre\_dob\_dd, pre\_dob\_mm, pre\_dob\_yyyy  
  
**def** get\_dod(person):  
 conn = sqlite3.connect(**"Family\_1.db"**)  
 cursor = conn.cursor()  
 sql = **"""  
 SELECT DOD  
 FROM tblFamily  
 Where PersonID = ?  
 """  
 for** row **in** cursor.execute(sql, (person,)):  
 pre\_dod = (row[0])  
 pre\_dod\_dd = pre\_dod[8:10]  
 pre\_dod\_mm = pre\_dod[5:7]  
 pre\_dod\_yyyy = pre\_dod[0:4]  
  
 conn.commit()  
 conn.close()  
 **return** pre\_dod\_dd, pre\_dod\_mm, pre\_dod\_yyyy  
  
**def** get\_pob(person):  
 conn = sqlite3.connect(**"Family\_1.db"**)  
 cursor = conn.cursor()  
 sql = **"""  
 SELECT POB  
 FROM tblFamily  
 Where PersonID = ?  
 """  
 for** row **in** cursor.execute(sql, (person,)):  
 pre\_pob = (row[0])  
  
 conn.commit()  
 conn.close()  
 **return** pre\_pob  
  
**def** get\_gender(person):  
 conn = sqlite3.connect(**"Family\_1.db"**)  
 cursor = conn.cursor()  
 sql = **"""  
 SELECT Gender  
 FROM tblFamily  
 Where PersonID = ?  
 """  
 for** row **in** cursor.execute(sql, (person,)):  
 pre\_gender = (row[0])  
  
 conn.commit()  
 conn.close()  
 **return** pre\_gender  
  
**def** get\_mother(person):  
 conn = sqlite3.connect(**"Family\_1.db"**)  
 cursor = conn.cursor()  
 sql = **"""  
 SELECT MotherID  
 FROM tblFamily  
 Where PersonID = ?  
 """  
 for** row **in** cursor.execute(sql, (person,)):  
 need\_change\_m = **False** pre\_mother = (row[0])  
 **if** pre\_mother == **""**:  
 is\_mother = **False  
 return** is\_mother, pre\_mother, need\_change\_m  
  
 **else**:  
 is\_mother = **True** sql = **"""  
 SELECT \*  
 FROM tblFamily  
 WHERE PersonID = ?  
 """  
 for** row **in** cursor.execute(sql, (pre\_mother,)):  
 pre\_mother = row  
   
 ValidM = re.match(**"[M]{1}[o]{1}[t]{1}[h]{1}[e]{1}[r]"**, pre\_mother[0])  
 **if** ValidM:  
 need\_change\_m = **True  
 return** is\_mother, pre\_mother, need\_change\_m  
 **else**:  
 **return** is\_mother, pre\_mother, need\_change\_m

SQL statements used to create database, wit tblFamily if it does not exist already (file\_exists is false)

SQL statements to create database.

Gets the first name of ‘person’

Gets the surname of ‘person’

Gets the day, month and year that ‘person’ was born in, but day, month and year are not connected in one string

Gets the day, month and year that ‘person’ died in, but day, month and year are not connected in one string.

Gets the place of birth of ‘person’

Gets gender of ‘person’

Gets mother of ‘person’

If pre\_mother is “”, then ‘person’ did not have a mother entered previously and so if they chose to edit a person, the drop-down menu where they can choose their mother will be set as the standard ‘-‘

If it is not “” but is of the normal format (i.e. 6-8 characters long depending on if some have \_P in addition to the 2 letters and 4 numbers), then the drop-down menu will contain the current stored mother.

Regular expressions used to check if PersonID contains the word ‘Mother’. If it does, then other people with the same motherID will need to be changed simultaneously – reflected by variable ‘need\_change\_m’

**def** get\_father(person):  
 conn = sqlite3.connect(**"Family\_1.db"**)  
 cursor = conn.cursor()  
 sql = **"""  
 SELECT FatherID  
 FROM tblFamily  
 Where PersonID = ?  
 """  
 for** row **in** cursor.execute(sql, (person,)):  
 pre\_father1 = (row[0])  
 print(pre\_father1)  
 need\_change\_f = **False  
 if** pre\_father1 == **""**:  
 is\_father = **False** pre\_father = **""  
 return** is\_father, pre\_father, need\_change\_f  
  
 **else**:  
 is\_father = **True** sql = **"""  
 SELECT \*  
 FROM tblFamily  
 WHERE PersonID = ?  
 """  
 for** row **in** cursor.execute(sql, (pre\_father1,)):  
 pre\_father = row  
 valid\_f = re.match(**"[F]{1}[a]{1}[t]{1}[h]{1}[e]{1}[r]"**, pre\_father[0])  
 **if** valid\_f:  
 need\_change\_f = **True  
 return** is\_father, pre\_father, need\_change\_f  
 **else**:  
  
 **return** is\_father, pre\_father, need\_change\_f  
  
**def** get\_family\_id(person):  
 conn=sqlite3.connect(**"Family\_1.db"**)  
 cursor = conn.cursor()  
 sql = **"""  
 SELECT FamilyID  
 FROM tblChildren  
 WHERE PersonID = ?  
 """  
 for** row **in** cursor.execute(sql, (person,)):  
 family\_id= row  
 **return** family\_id  
  
**def** get\_person\_id():  
 conn=sqlite3.connect(**"Family\_1.db"**)  
 cursor=conn.cursor()  
 cursor.execute(**"SELECT PersonID from tblFamily"**)  
 all\_person\_id=cursor.fetchall()  
 **return** all\_person\_id  
  
**def** create\_random\_id(Firstname, Surname):  
 prev\_random\_digit = [] *# NEED TO MAKE SURE ANOTHER RANDOM DIGIT THAT IS SAME IS CREATED* valid\_ran\_digit = **False** count = 0  
 first\_letter = Firstname[0]  
 second\_letter = Surname[0]  
 **while** valid\_ran\_digit == **False**:  
 random\_digit = str(random.randint(1000, 9999))  
 **for** number **in** prev\_random\_digit:  
 **if** int(number)== int(random\_digit):  
 count = count + 1  
 **else**:  
 **pass  
 if** count == 0:  
 valid\_ran\_digit = **True** prev\_random\_digit.append(random\_digit)  
 **else**:  
 valid\_ran\_digit = **False** PersonID = str(first\_letter + second\_letter + random\_digit)  
 **return** PersonID  
*#===============================================================================================***def** check\_unknown\_mother(Mother, Firstname, Surname, DOB): *#Checks if mother is 'Unknown and if its, reutrns the respective ID.* **if** Mother == **"UNKNOWN":**  
 MotherID = **"Mother of "** + Firstname + **" "** + Surname + **" "** + **"born on"** + **" "** + **" "** + str(DOB)  
 **else**:  
 ValidM = re.match(**"[M]{1}[o]{1}[t]{1}[h]{1}[e]{1}[r]{1}"**, Mother[3:10]) *# Regular Expressions (RE) check if Mother in ID. If it is, then formatting is needed* **if** ValidM: *# To generate MotherID* MotherID = Mother[3:len(Mother) - 3]  
 **else**:  
 MotherID = str((Mother[2:8]))  
 **return** MotherID  
  
**def** check\_unknown\_father(Father, Firstname, Surname, DOB): *# Same process as above but with Fathers* **if** Father == **"UNKNOWN"**  
 FatherID = **"Father of "** + Firstname + **" "** + Surname + **" "** + **"born on"** + **" "** + **" "** + str(DOB)  
 **else**:  
 valid\_f = re.match(**"[F]{1}[a]{1}[t]{1}[h]{1}[e]{1}[r]{1}"**, Father[3:10])  
 **if** valid\_f:  
 FatherID = Father[3:len(Father) - 3]  
 **else**:  
 FatherID = str((Father[2:8]))  
 **return** FatherID  
*#=======================================================Tkinter GUI Code======================================================================================================***def** close\_window(window):  
 window.destroy()  
  
**def** close\_window1(window1):  
 window1.destroy()  
  
**def** clear\_info(textentry\_fi,textentry\_su,textentry\_birth\_dd,textentry\_birth\_mm,textentry\_birth\_yyyy,textentry\_death\_dd,textentry\_death\_mm,textentry\_death\_yyyy,textentry\_pob,tkvar,tkvar\_m,tkvar\_f): *#CLears information from text entries* textentry\_fi.delete(0, END)  
 textentry\_su.delete(0, END)  
 textentry\_birth\_dd.delete(0, END)  
 textentry\_birth\_mm.delete(0, END)  
 textentry\_birth\_yyyy.delete(0, END)  
 textentry\_death\_dd.delete(0, END)  
 textentry\_death\_mm.delete(0, END)  
 textentry\_death\_yyyy.delete(0, END)  
 textentry\_pob.delete(0, END)  
 tkvar.set(**"Male"**)  
 tkvar\_m.set(**"-"**)  
 tkvar\_f.set(**"-"**)  
  
**def** enter\_info(person,window1,option\_num,textentry\_fi,textentry\_su,textentry\_birth\_dd,textentry\_birth\_mm,textentry\_birth\_yyyy,textentry\_death\_dd,textentry\_death\_mm,textentry\_death\_yyyy,textentry\_pob,tkvar,tkvar\_m,tkvar\_f):  
  
 **with** sqlite3.connect(**"Family\_1.db"**) **as** conn:  
 cursor = conn.cursor()  
 person\_rec = []  
 all\_info=**True** *#try:* Firstname = textentry\_fi.get()  
 **if** Firstname==(**None or ""**):  
 all\_info=**False** Surname=textentry\_su.get()  
 **if** Surname==(**None or ""**): *########Exception handling, to make sure that information entered is of correect format* all\_info=**False  
  
 if** all\_info==**True**:  
 PersonID = create\_random\_id(Firstname, Surname)  
  
 **if** option\_num == **"2"**:  
 PersonID = person  
 **try**:  
 DOB = datetime.date(int(textentry\_birth\_yyyy.get()), int(textentry\_birth\_mm.get()), int(textentry\_birth\_dd.get()))  
 **except** Exception:  
 DOB=**""  
  
 try**:  
 DOD = datetime.date(int(textentry\_death\_yyyy.get()), int(textentry\_death\_mm.get()), int(textentry\_death\_dd.get()))  
 **except** Exception:  
 DOD=**""** all\_info=CheckInvalid.check\_dob\_and\_dod(DOB,DOD,all\_info)  
 POB=textentry\_pob.get()  
  
 **if** POB==**None**:  
 POB=**""** Gender = tkvar.get()  
 Mother = tkvar\_m.get()  
 MotherID = check\_unknown\_mother(Mother, Firstname, Surname, DOB)  
 **if** option\_num == **"2"**:  
 need\_change\_m = get\_mother(person)[2]  
  
 **if** need\_change\_m == **True**:  
 **if** type(pre\_mother) **is** tuple:  
 pre\_mother = pre\_mother[0]  
  
 fieldM = **"'"** + MotherID + **"'"** keyfieldM = **"'"** + pre\_mother + **"'"** cursor.execute(**"UPDATE tblFamily SET MotherID ="** + fieldM + **"WHERE MotherID ="** + keyfieldM)  
  
 Father = tkvar\_f.get()  
 FatherID = check\_unknown\_father(Father, Firstname, Surname, DOB)  
 **if** option\_num == **"2"**:  
 need\_change\_f = get\_father(person)[2]  
 **if** need\_change\_f == **True**:  
 **if** type(pre\_father) **is** tuple:  
 pre\_father = pre\_father[0]  
 field\_f = **"'"** + FatherID + **"'"** keyfield\_f = **"'"** + pre\_father + **"'"** cursor.execute(**"UPDATE tblFamily SET FatherID ="** + field\_f + **"WHERE FatherID ="** + keyfield\_f)  
  
 clear\_info(textentry\_fi,textentry\_su,textentry\_birth\_dd,textentry\_birth\_mm,textentry\_birth\_yyyy,textentry\_death\_dd,textentry\_death\_mm,textentry\_death\_yyyy,textentry\_pob,tkvar,tkvar\_m,tkvar\_f)  
  
 **if** all\_info==**False**:  
 CheckInvalid.pop\_up\_menu()  
 **else**:  
 person\_rec.append(PersonID)  
 person\_rec.append(Firstname)  
 person\_rec.append(Surname)  
 person\_rec.append(DOB)  
 person\_rec.append(DOD)  
 person\_rec.append(POB)  
 person\_rec.append(Gender)  
 person\_rec.append(MotherID)  
 person\_rec.append(FatherID)  
 **if** option\_num == **"2"**:  
 keyfield = **"'"** + PersonID + **"'"** cursor.execute(**"Delete FROM tblFamily WHERE PersonID ="** + keyfield)  
 cursor.execute(**"INSERT INTO tblFamily VALUES (?,?,?,?,?,?,?,?,?)"**, person\_rec) *###Adds records to database* conn.commit()  
 person\_rec = []  
 Label2 = Label(window1, text=**"Record has been added"**, bg=**"Light blue"**, fg=**"black"**,  
 font=**"none 12 bold"**).grid(row=100, column=1)  
 window1.update()  
 conn.commit()  
 conn.close()  
   
**def** create\_and\_enter\_form(title, option\_num, person):  
 window1 = Tk()  
 window1.title(title)  
 window1.configure(background=**"Light blue"**)  
 *# label & text box for First name* Label(window1, text=**"Enter the Firstname of the person you are entering"**, bg=**"Light blue"**, fg=**"black"**,  
 font=**"none 12 bold"**).grid(row=1, column=0, sticky=W)  
 textentry\_fi = Entry(window1, width=20, bg=**"white"**)  
 textentry\_fi.grid(row=1, column=1, sticky=W)  
 **if** option\_num == **"2"**:  
 pre\_firstname = get\_firstname(person)  
 textentry\_fi.insert(INSERT, pre\_firstname)  
  
 *# label & text box for Surname* Label(window1, text=**"Enter the Surname of the person you are entering"**, bg=**"Light blue"**, fg=**"black"**,  
 font=**"none 12 bold"**).grid(row=2, column=0, sticky=W)  
 textentry\_su = Entry(window1, width=20, bg=**"white"**)  
 textentry\_su.grid(row=2, column=1, sticky=W)  
 **if** option\_num == **"2"**:  
 pre\_surname = get\_surname(person)  
 textentry\_su.insert(INSERT, pre\_surname)  
  
 *# label & text box for DOB* Label(window1, text=**"DD"**, bg=**"Light blue"**, fg=**"black"**, font=**"none 9 bold"**).grid(row=4, column=1, sticky=W)  
 textentry\_birth\_dd = Entry(window1, width=2, bg=**"white"**)  
 textentry\_birth\_dd.grid(row=5, column=1, sticky=W)  
  
 Label(window1, text=**"MM"**, bg=**"Light blue"**, fg=**"black"**, font=**"none 9 bold"**).grid(row=4, column=2, sticky=W)  
 textentry\_birth\_mm = Entry(window1, width=2, bg=**"white"**)  
 textentry\_birth\_mm.grid(row=5, column=2, sticky=W)  
  
 Label(window1, text=**"YYYY"**, bg=**"Light blue"**, fg=**"black"**, font=**"none 9 bold"**).grid(row=4, column=3, sticky=W)  
 textentry\_birth\_yyyy = Entry(window1, width=4, bg=**"white"**)  
 textentry\_birth\_yyyy.grid(row=5, column=3, sticky=W)  
  
 Label(window1, text=**"Enter the Date Of Birth of the person you are entering"**, bg=**"Light blue"**, fg=**"black"**,  
 font=**"none 12 bold"**).grid(row=5, column=0, sticky=W)  
  
 **if** option\_num == **"2"**:  
 pre\_dob\_dd = get\_dob(person)[0]  
 pre\_dob\_mm = get\_dob(person)[1]  
 pre\_dob\_yyyy = get\_dob(person)[2]  
 textentry\_birth\_dd.insert(INSERT, pre\_dob\_dd)  
 textentry\_birth\_mm.insert(INSERT, pre\_dob\_mm)  
 textentry\_birth\_yyyy.insert(INSERT, pre\_dob\_yyyy)  
  
 *# label & text box for date of death* Label(window1, text=**"DD"**, bg=**"Light blue"**, fg=**"black"**, font=**"none 9 bold"**).grid(row=6, column=1, sticky=W)  
 textentry\_death\_dd = Entry(window1, width=2, bg=**"white"**)  
 textentry\_death\_dd.grid(row=7, column=1, sticky=W)  
  
 Label(window1, text=**"MM"**, bg=**"Light blue"**, fg=**"black"**, font=**"none 9 bold"**).grid(row=6, column=2, sticky=W)  
 textentry\_death\_mm = Entry(window1, width=2, bg=**"white"**)  
 textentry\_death\_mm.grid(row=7, column=2, sticky=W)  
  
 Label(window1, text=**"YYYY"**, bg=**"Light blue"**, fg=**"black"**, font=**"none 9 bold"**).grid(row=6, column=3, sticky=W)  
 textentry\_death\_yyyy = Entry(window1, width=4, bg=**"white"**)  
 textentry\_death\_yyyy.grid(row=7, column=3, sticky=W)  
  
 Label(window1,  
 text=**"Enter the Date Of Death of the person you are entering (if the person you are entering is still alive, leave this blank)"**,  
 bg=**"Light blue"**, fg=**"black"**, font=**"none 12 bold"**).grid(row=7, column=0, sticky=W)  
 **if** option\_num == **"2"**:  
 pre\_dod\_dd = get\_dod(person)[0]  
 pre\_dod\_mm = get\_dod(person)[1]  
 pre\_dod\_yyyy = get\_dod(person)[2]  
 textentry\_death\_dd.insert(INSERT, pre\_dod\_dd)  
 textentry\_death\_mm.insert(INSERT, pre\_dod\_mm)  
 textentry\_death\_yyyy.insert(INSERT, pre\_dod\_yyyy)  
  
 *# label for place of birth* Label(window1, text=**"Place of Birth"**, bg=**"Light blue"**, fg=**"black"**, font=**"none 12 bold"**).grid(row=8, column=0,  
 sticky=W)  
 textentry\_pob = Entry(window1, width=20, bg=**"white"**)  
 textentry\_pob.grid(row=8, column=1, sticky=W)  
 **if** option\_num == **"2"**:  
 pre\_pob = get\_pob(person)  
 textentry\_pob.insert(INSERT, pre\_pob)  
  
 *# label for gender* Label(window1, text=**"What gender was this person born as?"**, bg=**"Light blue"**, fg=**"black"**,  
 font=**"none 12 bold"**).grid(row=9, column=0, sticky=W)  
  
 tkvar = StringVar(window1)  
 choices = [**'Male'**, **'Female'**]  
 tkvar.set(**'Male'**)  
  
 **if** option\_num == **"2"**:  
 pre\_gender = get\_gender(person)  
 tkvar.set(pre\_gender)  
 pop\_up\_menu = OptionMenu(window1, tkvar, \*choices) *#####Drop down menu* pop\_up\_menu.grid(row=9, column=1, sticky=W)  
  
 *# label for Mother* Label(window1, text=**"Select this person's mother"**, bg=**"Light blue"**, fg=**"black"**, font=**"none 12 bold"**).grid(  
 row=10, column=0, sticky=W)  
  
 tkvar\_m = StringVar(window1)  
 choices\_m = [**'UNKNOWN'**]  
 tkvar\_m.set(**'-'**)  
 **with** sqlite3.connect(**"Family\_1.db"**) **as** conn:  
 cursor = conn.cursor()  
 **for** row **in** cursor.execute(**"SELECT \* FROM tblFamily WHERE tblFamily.Gender='Female' Order BY Firstname"**):  
 choices\_m.append(row)  
  
 **for** row **in** cursor.execute(**"SELECT MotherID FROM tblFamily WHERE tblFamily.MotherID LIKE '%Mother%'"**):  
 choices\_m.append(row)  
  
 pop\_up\_menu\_m = OptionMenu(window1, tkvar\_m, **"-"**, \*choices\_m)  
 pop\_up\_menu\_m.grid(row=10, column=1, sticky=W)  
  
 **if** option\_num == **"2"**:  
 is\_mother = get\_mother(person)[0]  
 **if** is\_mother == **False**:  
 tkvar\_m.set(**'-'**)  
  
 **elif** is\_mother == **True**:  
 pre\_mother = get\_mother(person)[1]  
 tkvar\_m.set(pre\_mother)  
  
 need\_change\_m = get\_mother(person)[2]  
  
 *# label for father* Label(window1, text=**"Select this person's father"**, bg=**"Light blue"**, fg=**"black"**, font=**"none 12 bold"**).grid(  
 row=11, column=0, sticky=W)  
  
 tkvar\_f = StringVar(window1)  
 choices\_f = [**'UNKNOWN'**]  
 tkvar\_f.set(**'-'**)  
  
 **with** sqlite3.connect(**"Family\_1.db"**) **as** conn:  
 cursor = conn.cursor()  
  
 **for** row **in** cursor.execute(**"SELECT \* FROM tblFamily WHERE tblFamily.Gender='Male' Order BY Firstname"**):  
 choices\_f.append(row)  
 **for** row **in** cursor.execute(**"SELECT FatherID FROM tblFamily WHERE tblFamily.FatherID LIKE '%Father%'"**):  
 choices\_f.append(row)  
  
 popup\_menu\_f = OptionMenu(window1, tkvar\_f, **"-"**, \*choices\_f)  
 popup\_menu\_f.grid(row=11, column=1, sticky=W) *#######Drop down menu* **if** option\_num == **"2"**:  
 is\_father = get\_father(person)[0]  
 **if** is\_father == **False**:  
 tkvar\_f.set(**'-'**)  
 **elif** is\_father == **True**:  
 pre\_father = get\_father(person)[1]  
 tkvar\_f.set(pre\_father)  
 need\_change\_f = get\_father(person)[2]  
  
 *# button for submit, clear data and go back to main menu* Button(window1, text=**"Submit"**, width=6, command=**lambda**: [enter\_info(person,window1,option\_num,textentry\_fi,textentry\_su,textentry\_birth\_dd,textentry\_birth\_mm,textentry\_birth\_yyyy,textentry\_death\_dd,textentry\_death\_mm,textentry\_death\_yyyy,textentry\_pob,tkvar,tkvar\_m,tkvar\_f)]).grid(row=22, column=0, sticky=W)

Gets the FamilyID that person is the child of (as everyone if the child of a family, but not everyone is the parent of a family

Sae process as above is done for father of person

Gets all PersonID’s in the database

Gets first letter of first name and surname

Generates random 4 digit number

Makes sure that the same 4 digit number is not generated twice

Puts together first letter of firstname, surname and random number to form personID

Generates ID if “UNKNOWN” chosen for mother

Checks if “Mother” is in drop-down menu selection for mother using regular expressions. If it is, then PersonID has to be extracted from the string used to represent the mother in the drop-down menu in a different way than it would be if it did not contain “Mother”

Same process as above repeated for fathers

Clears text/choices in drop down menu entered by user

If all\_info is False during the process of entering information into the database, the record will not be added, and a message will be displayed to the user.

Checks if user has entered firstname and surname (both needed to generate the PersonID)

Creates PersonID

If they are editing a person, we want the person’s unique ID to stay the same

If an invalid date of birth is entered (for example it contains letters or the date just cannot exist such as 30th Feb, then DOB is set as “”

If an invalid date of death is entered (for example it contains letters or the date just cannot exist such as 30th Feb, then DOB is set as “”

Checks for more cases of invalid DOB’s and DOD’s such as, a DOB/DOD that is in the future, and a DOB that supposedly happened after the DOD

If the user entered no place of birth (POB), POB is set to “”

Gets Gender of person

Gets MotherID, or creates it if ‘UNKNOWN’

If need\_change\_m is true ( i.e. PersonID contains “Mother”, other people with the same MotherID are updated too.

Gets FatherID, or creates it if it is ‘UNKNOWN’

If need\_change\_f is true ( i.e. PersonID contains “Father”, other people with the same FatherID are updated too.

Clears info entered before writing data to database, so that user can enter another person

Creates pop-up menu if there was an invalid entry – info is not written to database if this is displayed

Adds person to the database

Creates form for adding new person or editing existing person. If a person is being edited, then that existing persons information is displayed in each text-box and drop-down menu

Button(window1, text=**"Clear"**, width=5, command=**lambda**:[clear\_info(textentry\_fi,textentry\_su,textentry\_birth\_dd,textentry\_birth\_mm,textentry\_birth\_yyyy,textentry\_death\_dd,textentry\_death\_mm,textentry\_death\_yyyy,textentry\_pob,tkvar,tkvar\_m,tkvar\_f)]).grid(row=22, column=1, sticky=W)

Button(window1, text=**"Main Menu"**, width=9, command=**lambda**: [close\_window1(window1)]).grid(row=22,column=2,sticky=W)  
  
 window1.mainloop()

**Sorting Algorithm**

Imports:  
sqlite3: Allows databases to be manipulated  
Getting\_all\_PersonID\_Names\_imports: Gets a dictionary of people’s PersonID as a key and their first name and surname as the value. It also creates names for people who have no partner (e.g. if Bob had no partner, his partner would be called “Partner of Bob…”  
DatabaseChanges\_23\_02\_2019: Checks if tblFamily and tblChildren exist, as well as creating them if they do not and adding values to them.

**import** sqlite3  
**from** tkinter **import** \*  
**import** Getting\_all\_PersonID\_Names\_imports **as** AllPeopleInDatabase  
**import** DatabaseChanges\_23\_02\_2019 **as** DatabaseChanges  
**import** copy  
**import** Change\_Distance\_Dictionary **as** ChangeDistance  
  
  
  
  
*#============================== Determine children of people ===================================================================================***def** det\_children():  
 children\_dict={}  
 all\_people = AllPeopleInDatabase.list\_all()  
   
 **for** person **in** all\_people:  
 personX=person  
 person\_gender = get\_person\_gender(personX)  
 children\_set = get\_children(person\_gender,personX)  
 children\_dict[personX] = children\_set  
  
 **return** children\_dict  
   
   
*#============================================== Retrieving Information From Database =============================================================***def** get\_person\_gender(personX):  
 conn= sqlite3.connect(**"Family\_1.db"**)  
 cursor= conn.cursor()  
 sql = **"""  
 SELECT Gender  
 FROM tblFamily  
 Where PersonID = ?  
 """  
 for** row **in** cursor.execute(sql,(personX,)):  
 person\_gender=(row[0])  
 conn.commit()  
 conn.close()  
 **return** person\_gender  
  
  
**def** get\_children(person\_gender,personX):  
 children\_set=set()  
 conn = sqlite3.connect(**"Family\_1.db"**)  
 cursor = conn.cursor()  
 **if** person\_gender == **"Female"**:  
 sql = **"""  
 SELECT PersonID  
 FROM tblFamily  
 WHERE MotherID = ?  
 """  
 for** row **in** cursor.execute(sql,(personX,)):  
 child\_id = row[0]  
 child\_id=person\_id\_format(child\_id)  
 children\_set.add(child\_id)  
 **elif** person\_gender == **"Male"**:  
 sql = **"""  
 SELECT PersonID  
 FROM tblFamily  
 WHERE FatherID = ?  
 """  
 for** row **in** cursor.execute(sql,(personX,)):  
 child\_id = row[0]  
 child\_id=person\_id\_format(child\_id)  
 children\_set.add(child\_id)  
   
 children\_set.discard(**"set()"**)  
 **return** children\_set  
  
*#========================================================== Create families ============================================================================================*

Gets the gender of each person in turn, to determine whether Mother ID’s (if Female) or Father Id’s (if Male) need to be checked.

Uses tblFamily and the gender to determine all people who have a certain person as a mother or a father

These are then put into a dictionary called children\_dict

Gets all PersonID’s in the database

Gets the gender of ‘person’

If person\_gender is Female, Mother ID’s are checked

Adds each result of the SQL query to the set of children

Same process carried out if person\_gender is Male (FatherID’s are used instead however)

**def** families(children\_dict,all\_family\_id):  
 PeopleUsed=set()  
  
 all\_people=AllPeopleInDatabase.list\_all()  
 family\_id\_queue=initialise\_family\_id\_queue() *########Creates queue* exists = DatabaseChanges.check\_table\_exists(str(**"tblSingleFamily"**))  
 **if** exists == **True**:  
 DatabaseChanges.delete\_table(**"tblSingleFamily"**) *#####Checks if tblSingleFamily and tblChildren and either creates them,, or deletes them and then creates them* DatabaseChanges.create\_tblSingleFamily()  
 **else**:  
 DatabaseChanges.create\_tblSingleFamily()  
  
 exists=DatabaseChanges.check\_table\_exists(**"tblChildren"**)  
 **if** exists == **True**:  
 DatabaseChanges.delete\_table(**"tblChildren"**)  
 DatabaseChanges.create\_tblChildren()  
 **else**:  
 DatabaseChanges.create\_tblChildren()  
  
 **for** x **in** range(0,len(all\_people)):  
 **for** y **in** range(x+1,len(all\_people)):  
  
 personX=all\_people[x]  
 personY=all\_people[y]  
 first = children\_dict[personX]   
 second = children\_dict[personY]  
  
 **if** len(first.intersection(second))!=0 **and** len(first)!=0 **and** len(second)!=0: *##########Checks if the intersection of the sets is empty or not* family\_id\_queue, first\_family\_id, all\_family\_id= add\_family(personX, personY, family\_id\_queue,all\_family\_id)  
 DatabaseChanges.add\_children(first,second,first\_family\_id)  
 PeopleUsed.add(personX)  
 PeopleUsed.add(personY)  
  
 **elif** len(first.intersection(second))==0 **or** len(first)==0 **or** len(second)==0 :  
 **pass  
  
 return** all\_family\_id

If the intersection is not empty, they are deemed to be in a relationship and so add the 2 people and their intersection of children to tblSingleFamily and to tblChildren

Checks to see if the length of the intersection of the 2 sets of children is non-zero, and checks if either of the sets is empty.

If one of the conditions in the above if statement is not met, continue on to the next family

Gets the gender of the 2 people who are deemed to have a relationship

Creates a queue containing FID1,FID2,FID3….,FID99,FID100

If tblChildren exists, it is deleted and an empty tblChildren is created. If it does not exist, then tblChildren is just created.

Gets 2 people from the list of everyone’s person ID

Gets the set of children of PersonX and personY

Nested for loop checks between all pairs of people, but only once.

If tblSingleFamily exists, it is deleted and an empty tblSingleFamily is created. If it does not exist, then tblSingleFamily is just created.

**def** add\_family(personX, personY, family\_id\_queue,all\_family\_id):  
 GenderX = get\_person\_gender(personX)  
 GenderY = get\_person\_gender(personY)  
 first\_family\_id, family\_id\_queue = dequeue(family\_id\_queue)  
 all\_family\_id.append(first\_family\_id)  
 family\_rec=[first\_family\_id]  
 connection = sqlite3.connect(**"Family\_1.db"**)  
 cursor = connection.cursor()  
 **if** GenderX ==**"Male"**:  
 family\_rec.append(personY)  
 family\_rec.append(personX)  
 **else**:  
 family\_rec.append(personX)  
 family\_rec.append(personY)  
 cursor.execute(**"INSERT INTO tblSingleFamily VALUES (?,?,?)"**, family\_rec)  
 connection.commit()  
 **return** family\_id\_queue, first\_family\_id, all\_family\_id  
  
**def** get\_family(current\_family): *# Gets parents and children of a specific familyID* connection = sqlite3.connect(**"Family\_1.db"**)  
 persons=[]  
 parents=[]  
 children=[]  
 FamilyID = **"'"** + current\_family + **"'"** cursor=connection.cursor()  
 sql = (**"""  
 SELECT Mother, Father, PersonID  
 FROM tblSingleFamily, tblChildren  
 WHERE tblSingleFamily.FamilyID = %s  
 AND tblChildren.FamilyID = %s """** % (FamilyID, FamilyID,))  
 **for** row **in** cursor.execute(sql):  
 persons.append(row)  
 parents.append(persons[0][0])  
 parents.append(persons[0][1])  
 **for** x **in** range(0,len(persons)):  
 children.append(persons[x][2])  
  
 **return** parents, children   
  
  
  
*#======================================================== Dealing With People in tblFamily that are not of the typical format===================================================================***def** add\_two\_unknowns(children\_dict): *###Adds unknown people to database* person\_id\_dict=AllPeopleInDatabase.person\_id\_against\_name\_dict()  
 connection=sqlite3.connect(**"Family\_1.db"**)  
 cursor=connection.cursor()  
 sql=**"""  
 SELECT MotherID  
 FROM tblFamily  
 WHERE MotherID LIKE "%Mother%"  
 """** unknownm=[]  
 usedm=[]  
 **for** row **in** cursor.execute(sql):  
 unknownm=.append(row)  
 **for** z **in** range(len(unknownm)-1,-1,-1):  
 **if** unknownm[z] **in** person\_id\_dict:  
 unknownm.pop(z)  
 **for** x **in** range(0,len(unknownm)):  
 **if** (unknownm[x]**not in** usedm) and (unknownm[x][0] not in person\_id\_dict:  
 person\_rec=[unknownm[x][0],**""**,**""**,**""**,**""**,**""**,**"Female"**,**""**,**""**]  
 cursor.execute(**"INSERT INTO tblFamily VALUES (?,?,?,?,?,?,?,?,?)"**, person\_rec)  
 person\_rec=[]  
 usedm.append(unknownm[x])   
   
 connection.commit()  
   
  
 sql2=**"""  
 SELECT FatherID  
 FROM tblFamily  
 WHERE FatherID LIKE "%Father%"  
 """** unknownf=[]  
 usedf=[]  
 **for** row **in** cursor.execute(sql2):  
 unknownf.append(row)  
 **for** x **in** range(len(unknownf)-1,-1,-1): **if** unknownf[x] **in** person\_id\_dict:  
 unknownf.pop(x)  
 **for** y **in** range(0,len(unknownf)):  
 **if** (unknownf[y] **not in** usedf and (unknownf[y][0] not in person\_id\_dict):  
 person\_rec=[unknownf[y][0],**""**,**""**,**""**,**""**,**""**,**"Male"**,**""**,**""**]  
 cursor.execute(**"INSERT INTO tblFamily VALUES (?,?,?,?,?,?,?,?,?)"**, person\_rec)  
 person\_rec=[]  
 usedf.append(unknownf[y])  
 connection.commit()  
 **return** unknownm, unknownf  
  
**def** add\_2\_unknowns\_to\_children\_set(unknownm,unknownf,children\_dict): *###Adds unknown people to database* **for** x **in** range(0,len(unknownm)):  
 Mother=str(unknownm[x][0])  
 children\_set=get\_children(**"Female"**,Mother)  
 children\_dict[Mother]=children\_set  
 **for** y **in** range(0,len(unknownf)):  
 Father=str(unknownf[y][0])  
 children\_set=get\_children(**"Male"**,Father)  
 children\_dict[Father]=children\_set  
 **return** children\_dict  
   
**def** one\_unknown\_parent(children\_dict):  
 connection = sqlite3.connect(**"Family\_1.db"**)  
 cursor=connection.cursor()  
 sql= **"""  
 SELECT FatherID  
 FROM tblFamily  
 WHERE MotherID="" AND FatherID<>""  
 """  
 for** row **in** cursor.execute(sql):  
 **if** row!=**""**:  
 unknown\_mother = str(str(row[0] + **"\_P"**))  
 unknown\_mother = **"'"** + unknown\_mother + **"'"** Father = **"'"** + str(row[0]) + **"'"  
 for** x **in** range(0,len(children\_dict[row[0]])):  
 person = str(list(children\_dict[row[0]])[x])  
 person=person\_id\_format(person)  
 person= **"'"** + person + **"'"** cursor.execute(**"UPDATE tblFamily SET MotherID ="** + unknown\_mother + **"WHERE FatherID ="** + Father + **"AND PersonID ="** +

Gets the FamilyID that will be used to represent the family in the FIFO structure of a queue, and append it to family\_rec

Returns the parents and the children of a specific FamilyID

Returns the new family\_id\_queue, the familyID for that family (which is used for tblChildren and all\_famil\_id, which is a list of all FamilyID’s that have been given out.

Adds family\_rec to tblSingleFamily

Appends mother and then father to family\_rec, and it decides who is the mother and father of the family by looking at the genders

Adds the mother and father of the family to tblSingleFamily

Returns the queue, family\_id used, and a list of all the family Ids given out.

Gets the parents and children of a specific family ID

Gets every mother in the database who has “Mother” in their person ID

Makes sure person is not entered if they are already a record

Gets everyone’s mother, where their MotherID contains “Mother”. These people have their own record created.

Makes sure that the same person is not entered twice

Same process carried out for fathers

A list of all of the unknown mothers and a list of all the unknown father is returned

Adds the unknown people and their children to children\_dict (this is done before families() is run

Finds people who only have one parent entered, and gets the known partner’s ID, and appends “\_P” to it (for some reason the underscores are not showing)

Updates tblFamily, where the Mother is not known, with this newly generated ID

person )  
 connection.commit()  
   
   
   
   
 sql2 = **"""  
 SELECT MotherID  
 FROM tblFamily  
 WHERE MotherID<>"" AND FatherID = ""  
 """  
 for** row2 **in** cursor.execute(sql2):  
 **if** row2!=**""**:  
 unknown\_father = str(str(person\_id\_format(row2[0])) + **"\_P"**)  
 unknown\_father = **"'"** + unknown\_father + **"'"** Mother = **"'"** + str(person\_id\_format(row2[0])) + **"'"  
   
 for** y **in** range(0,len(children\_dict[person\_id\_format(row2)])):  
 cursor.execute(**"UPDATE tblFamily SET FatherID ="** + unknown\_father + **"WHERE MotherID ="** + Mother )  
 connection.commit()

Same process for people where only the Mother Is known

add\_unknown\_parents\_to\_children\_set(children\_dict)   
 connection.commit()  
 connection.close()  
   
   
  
**def** add\_unknown\_parents\_to\_children\_set(children\_dict): *# Adds unknown people to children set* person\_id\_dict = AllPeopleInDatabase.person\_id\_against\_name\_dict()  
 connection=sqlite3.connect(**"Family\_1.db"**)  
 cursor=connection.cursor()  
 sql = **"""  
 SELECT PersonID,MotherID  
 FROM tblFamily  
 WHERE MotherID LIKE "%\_P"  
 ORDER BY MotherID  
 """  
 for** row **in** cursor.execute(sql):  
 **if** (row[1] **not in** children\_dict) **and** (row[1] **not in** person\_id\_dict):  
 parent\_rec=[row[1],**""**,**""**,**""**,**""**,**""**,**"Female"**,**""**,**""**]  
 cursor.execute(**"INSERT INTO tblFamily VALUES (?,?,?,?,?,?,?,?,?)"**,parent\_rec)  
 connection.commit()  
 parent\_rec=[]  
 children\_dict[row[1]]=set()  
 children\_dict[row[1]].add(row[0])  
   
 connection=sqlite3.connect(**"Family\_1.db"**)  
 cursor=connection.cursor()  
 sql = **"""  
 SELECT PersonID,FatherID  
 FROM tblFamily  
 WHERE FatherID LIKE "%\_P"  
 ORDER BY FatherID  
 """  
 for** row **in** cursor.execute(sql):  
 **if** (row[1] **not in** children\_dict) **and** (row[1] **not in** person\_id\_dict):  
 parent\_rec=[row[1],**""**,**""**,**""**,**""**,**""**,**"Male"**,**""**,**""**]  
 cursor.execute(**"INSERT INTO tblFamily VALUES (?,?,?,?,?,?,?,?,?)"**,parent\_rec)  
 connection.commit()  
 parent\_rec=[]  
 children\_dict[row[1]]=set()  
 children\_dict[row[1]].add(row[0])  
 **return** children\_dict  
  
  
*#==============================================================ID and String Formatting (makes sure everything is in the same/similar format ============================================================================***def** person\_id\_format(variable):  
 **if** type(variable) **is** tuple:  
 variable=variable[0]  
  
 variable.replace(**"("**,**""**)  
 variable.replace(**")"**,**""**)  
 variable.replace(**","**,**""**)  
 **return** variable  
  
**def** check\_format\_family\_id(variable):  
 **if** type(variable) **is** list:  
 variable = str(variable[0])  
 **if** type(variable) **is** tuple:  
 variable=str(variable[0])  
 variable=variable.replace(**"'"**,**""**)  
 variable=variable.replace(**"("**,**""**)  
 variable=variable.replace(**")"**,**""**)  
 variable=variable.replace(**","**,**""**)  
 **return** variable  
  
**def** list\_format(list\_variable):  
 new\_list = []  
 **for** element **in** list\_variable:  
 entry = person\_id\_format(element)  
 new\_list.append(entry)  
 **return** new\_list  
   
*#========================================================= family Queue =========================================================================================  
#Creates Queues***def** initialise\_family\_id\_queue():  
 family\_id\_queue=[]  
 **for** x **in** range(1,101):  
 family\_id\_queue.append(str(**"FID"**+str(x)))  
 **return** family\_id\_queue  
  
**def** dequeue(family\_id\_queue):  
 first\_family\_id=family\_id\_queue.pop(0)  
 **return** first\_family\_id,family\_id\_queue  
  
*#========================================================== Subroutines for traverse Algorithm to work ===========================================================***def** list\_of\_family\_id():  
 connection=sqlite3.connect(**"Family\_1.db"**)  
 family\_id\_list=[]  
 cursor=connection.cursor()  
 sql = (**"""  
 SELECT FamilyID  
 FROM tblSingleFamily  
 """**)  
 **for** row **in** cursor.execute(sql):  
 family\_id\_list.append(row)  
 **return** family\_id\_list  
  
**def** create\_adjacency\_matrix():  
 family\_matrix=[]  
 family\_id\_list= list\_of\_family\_id()  
 family\_id\_length = len(family\_id\_list)  
  
 **for** x **in** range(0,family\_id\_length):  
 family\_matrix.append([])  
 **for** y **in** range(0,family\_id\_length):  
 **for** z **in** range(0, family\_id\_length):  
 family\_matrix[y].append([])  
  
 **return** family\_matrix  
  
**def** create\_visited\_family\_list():  
 visited=[]  
 **return** visited  
  
**def** add\_visited\_family\_list(family,visited):  
 visited.append(family)  
 **return** visited  
  
**def** check\_children(child\_id,det\_children):  
 **if** len(det\_children[child\_id])==0:  
 **pass  
 elif** len(det\_children[child\_id])!=0:  
   
 connection= sqlite3.connect(**"Family\_1.db"**)  
 cursor = connection.cursor()  
 parent\_family=[]  
 child\_id = **"'"** + child\_id + **"'"** sql = (**"""  
 SELECT FamilyID  
 FROM tblSingleFamily, tblFamily  
 WHERE (((tblSingleFamily.Mother= %s) AND (tblFamily.PersonID = %s))  
 OR ((tblSingleFamily.Father = %s) AND (tblFamily.PersonID = %s)))"""**% (child\_id, child\_id, child\_id, child\_id,))  
 **for** row **in** cursor.execute(sql):  
 parent\_family.append(row)  
 **return** parent\_family  
  
**def** check\_parents(parent\_id): *#Gets the familyID of a family above another one* valid\_person\_id = re.match(**"[A-Z]{1,2}[0-9]{4}"**, parent\_id)  
 **if not** valid\_person\_id:  
 **pass  
 else**:  
 parent\_in\_db=**False** ParentsList=[]  
 connection = sqlite3.connect(**"Family\_1.db"**)  
 cursor = connection.cursor()  
 parent\_id = **"'"** + parent\_id + **"'"** sql0 = (**"""  
 SELECT MotherID, FatherID  
 FROM tblFamily  
 WHERE tblFamily.PersonID = %s  
 """**% (parent\_id,))  
 **for** row **in** cursor.execute(sql0):  
 ParentsList.append(row)  
 **if** len(ParentsList)!=0:  
 parent\_in\_db=**True  
 else**:  
 **pass  
 if** parent\_in\_db == **True**:  
 child\_family=[]  
 sql = (**"""  
 SELECT FamilyID  
 FROM tblChildren  
 WHERE tblChildren.PersonID = %s """** % (parent\_id,))  
 **for** row **in** cursor.execute(sql):  
 child\_family.append(row)  
  
 **return** child\_family  
 **elif** parent\_in\_db==**False**:  
 **pass  
   
   
   
   
  
  
def** traverse(visited, stack, family\_matrix, current\_family,all\_family\_id): *# A depth first search* children\_dict=det\_children()  
 **if** current\_family **in** visited:  
 **pass  
 else**:  
 visited.append(current\_family) *# Appends to visited each time current\_family changes* current\_family=check\_format\_family\_id(current\_family)  
 parents, children = get\_family(str(current\_family))  
 family\_below=**False  
  
 for** x **in** range(0,len(children)):  
 child\_id = children[x]  
 parent\_family = check\_children(child\_id,children\_dict)  
 **if** parent\_family **is None**:  
 **pass  
 else**:  
 **for** y **in** range(0,len(parent\_family)):  
 parent\_familyY=parent\_family[y]  
 parent\_familyY=check\_format\_family\_id(parent\_family)  
 **if** parent\_familyY **in** visited:   
 **pass  
   
 else**:  
 family\_below=**True** next\_family = str(parent\_familyY)  
 next\_family = check\_format\_family\_id(next\_family)  
 childIDy=child\_id  
 **break  
   
  
 if** family\_below==**True**:  
  
 current\_family = check\_format\_family\_id(current\_family)  
 family\_index=int((str(current\_family)[3:int(len(current\_family))]))-1  
 next\_family\_index = int((next\_family[3:int(len(next\_family))]))-1 *#####Determine position of where info should be added in matrix* (family\_matrix[family\_index])[next\_family\_index].append(1)  
 (family\_matrix[next\_family\_index])[family\_index].append(-1)  
 (family\_matrix[family\_index])[next\_family\_index].append(childIDy)  
 (family\_matrix[next\_family\_index])[family\_index].append(childIDy)*##############Adds info to matrix* (family\_matrix[next\_family\_index])[family\_index].append(next\_family)  
 (family\_matrix[family\_index])[next\_family\_index].append(current\_family)  
 (family\_matrix[next\_family\_index])[family\_index].append(current\_family)  
 (family\_matrix[family\_index])[next\_family\_index].append(next\_family)  
  
 stack = push(stack, current\_family)  
 traverse(visited, stack, family\_matrix, next\_family,all\_family\_id) *##Updates stack after family matrix has been ammended* **elif** family\_below==**False**: *# If there is no family below, check if there is a family above* family\_above=**False**

Inserts these people into tblFamily

Adds these people to children\_dict, where the mothers are the key, and the personID’s of the people with the mother are values.

Same process carried out, but with fathers instead.

Gets all people and their mothers, where the mother contains “\_P”

Adds Unknown Partners as their own record in tblFamily, and adds them to children\_dict.

Makes sure all personID’s are of the same format (sometimes SQL searches return personID’s with brackets and apostrophes etc.)

Makes sure all FamilyID’s are of the same format (sometimes SQL searches return FamilyID’s with brackets and apostrophes etc.)

Makes sure all person IDs in a list are of the same format

Creates a queue, and loads it with “FID1”,”FID2”,”FID3”…”FID100”

Removes and returns the first element of the queue

Gets a list of all FamilyID’s that have been given out

Creates an nxn adjacency matrix (is programmed as nested lists), where n is the number of families (i.e. n=family\_id\_length)

Creates visited list, needed for traverse algorithm

Adds a family to visited

Checks if a person is a parent of a family if they are not, then SQL query is not needed

Gets Family ID where ‘child\_id’ is either the mother or the father.

Uses regular expressions to check whether parent\_id is of the form 2 letters followed by 4 digits. If it not, then there is no point continuing, as then the person with parent\_id will not have a parent entered into the database.

Checks if parent\_id has a mother/father entered into the database

If parent\_id does have a mother/father entered in the database, the program gets the family where parent\_id is a child and returns it.

Depth first search, that is recursive and uses a stack initially, visited, stack and family\_matrix are empty and current\_family is “FID1” and all\_family\_id is a list of all family IDs in the database.

Gets dictionary of every person in the database as a key, and each person’s children as a value.

If current\_family has already been travelled to (hence it is in the visited list), then it does not need to be appended again to visited

Makes sure that current\_family is of the correct format

Gets the parents and children of current\_family

Program goes through each child in current\_family and checks if they are a parent of a family (if they are, parent\_family will not be None)

Checks if the family that the child is a parent of has already been visited

If the family has not been visited, then the program will then visit this family. The family that it will visit is stored as next\_family and it means that there is a family below.

Gets ‘row’ and ‘column’ of where the info will be entered into. (if current\_family is FID3 and next\_family is FID9, info will be entered into row=2,column=8 and row=8,column=2

Appends data to correct lists in family\_matrix. If FID9 is below FID3, a ‘1’ will be appended to row=2,column=8 and a ‘-1’ will be appended to row=8,column=2

Data includes 1/-1, current\_family, next\_family (corresponds to row and column that data is being appended to) and the person linking the families.

Current\_family is pushed onto stack, and the algorithm calls itself

If there is no family below, the program checks if there is a family above

**for** z **in** range(0,len(parents)):  
 parent\_id = parents[z]  
 child\_family = check\_parents(parent\_id)  
  
 **if** child\_family==**None or** child\_family==**"None"**:  
 child\_family=[]  
  
 **if** len(child\_family)>0:  
 child\_family=check\_format\_family\_id(child\_family)  
 **else**:  
 **pass  
 if** len(child\_family) ==0:  
 **pass  
 elif** len(child\_family)>0 **and** child\_family **not in** visited:  
 family\_above = **True  
 break  
 else**:  
 **pass  
  
 if** family\_above==**False and** len(stack)>=1:  
 stack, top\_of\_stack = pop(stack)  
 traverse(visited, stack, family\_matrix, top\_of\_stack,all\_family\_id)  
  
 **elif** family\_above==**False and** len(stack)==0 **and** len(family\_matrix)==len(visited): *##########Base case for recursive algorithm* **return** family\_matrix, visited

Base case of recursive algorithm: if there is no family above or below, the length of the stack is zero, and all families have been visited, then all connections have been found and so family\_matrix and the list of visited families can be returned.

If there is no family above or below, but the length of the stack is greater than one, then the top FamilyID in the stack is removed and the algorithm is called with this top FamilyID, as there might be more than one sibling who is also a parent, or the other parent may also be a child in an entered family.

Program checks if the length of child\_family is greater than zero. It then checks if child\_family is not in visited. If child\_family not in visited as well, then there is a family above.

For each parent in current\_family, the program checks if they are a child of another family

If they are not, then child\_family is set to an empty list, otherwise child\_family is a FamilyID, and its format is checked.

If there is an element in ‘child\_family’ and it is not in the ‘visited’ list, then that must be a new family above that we have not visited yet.

If there is no family above or below bu the length of the stack is not zero, we take the top item from the stack, remove it, and use it as our ‘current family’ when we make th next recursive call.

Base case: If all the families have been visited (so the length of family\_martix = the length of visited) and there is no family above or below and the length of the stack is zero,all of the connections must have been found.

**elif** family\_above==**False and** len(stack)==0 **and** len(all\_family\_id)!=len(visited): *#This elif statement deals with families that are 'unconnected' trees* found=**False  
 while** found==**False**:  
 **for** x **in** range(0,len(all\_family\_id)):  
 **if** all\_family\_id[x] **not in** visited:  
 found=**True** current\_family=all\_family\_id[x]  
 traverse(visited,stack,family\_matrix,current\_family,all\_family\_id)  
  
   
   
 **elif** family\_above == **True**:  
  
 child\_family=check\_format\_family\_id(child\_family)  
 current\_family=check\_format\_family\_id(current\_family)  
 family\_index=int((current\_family[3:int(len(current\_family))]))-1  
 child\_family\_index=int((child\_family[3:int(len(child\_family))]))-1  
  
 family\_matrix[family\_index][child\_family\_index].append(-1)  
 family\_matrix[child\_family\_index][family\_index].append(1)  
 family\_matrix[family\_index][child\_family\_index].append(parent\_id)  
 family\_matrix[child\_family\_index][family\_index].append(parent\_id)  
 family\_matrix[child\_family\_index][family\_index].append(child\_family)  
 family\_matrix[family\_index][child\_family\_index].append(current\_family)  
 family\_matrix[child\_family\_index][family\_index].append(current\_family)  
 family\_matrix[family\_index][child\_family\_index].append(child\_family)  
   
 parent\_id=**None** stack = push(stack, current\_family)  
 traverse(visited, stack, family\_matrix, child\_family,all\_family\_id)  
   
  
   
 family\_id\_list= list\_of\_family\_id()  
 family\_id\_length = len(family\_id\_list)  
 **for** x **in** range(0,family\_id\_length):  
 **for** y **in** range(0,family\_id\_length):  
 **for** z **in** range(0, family\_id\_length):  
 **if** len(family\_matrix[y][z]) == 0:  
 family\_matrix[y][z].append(0)  
 family\_matrix[y][z].append(0)  
 family\_matrix[y][z].append(**"FID"** + str(y+1)) *#### Appends 0's afterwards to empty lists.* **else**:  
 **pass  
  
 return** family\_matrix, visited  
  
*#=============================================Create Stack for traverse algorithm==================================================================***def** create\_stack():  
 stack=[]  
 **return** stack  
  
**def** push(stack,Item):  
 stack.append(Item)  
 **return** stack  
  
**def** pop(stack):  
 top\_of\_stack= stack.pop()  
 **return** stack, top\_of\_stack  
  
**def** peek(stack):  
 **return** stack[len(stack)]  
*#==================================Subroutines for That return variables for Kivy =================================================================================*

Algorithm then calls itself with ‘child\_family’ as ‘current\_family’

The row and column that the nested list lies in is also appended

0’s added to empty nested lists

If there is no family above or below, the length of stack is zero and not all families have been visited, then graph representing the families must not be connected. So, if this happens, a familyID will be randomly selected, that has not been visited yet. The algorithm will then call itself with this family.

If there is a family above, in a similar way to as if there was a family below, info is added to the relevant nested lists in family\_matrix.

‘current\_family’ is pushed onto the stack, and child\_family ismade the new current family for the next recursive call.

Appends 0s and the familyID (row) to each nested list that is empty.

Creates stack

Adds item to the top of the stack

Removes and returns the top value of the stack

Just returns the top value in the stack

A recursive algorithm: family\_matrix is complete already, all\_family\_id is a list of all familyID’s allocated, entered is initially [], current is initially “FID1” and distance\_dictionary is initially “FID1:0”

**def** descendant\_number(family\_matrix, all\_family\_id, entered, current, distance\_dictionary):  
 entered.append(current)  
 **if** len(distance\_dictionary) == len(family\_matrix): *###Base case* **return** distance\_dictionary  
  
 **for** x **in** range(0, len(entered)):  
 family\_index\_a = int(entered[x][3:len(entered[x])])-1  
  
 **for** y **in** range(0, len(all\_family\_id)):  
 extra\_distance = int(((family\_matrix[family\_index\_a][y][0])))  
 final\_family\_id = str(**"FID"** + str(y+1))  
 **if** (extra\_distance!=0) **and** (final\_family\_id **not in** entered):  
  
 **if** family\_index\_a == 1:  
 distance\_dictionary[final\_family\_id] = extra\_distance *#Finds distance from previous familyID* current = final\_family\_id  
 descendant\_number(family\_matrix, all\_family\_id, entered, current, distance\_dictionary)  
 **return** distance\_dictionary  
  
 **else**:  
 total\_weight = distance\_dictionary.get(entered[x]) + extra\_distance *#Finds distance from FID1* distance\_dictionary[final\_family\_id] = total\_weight  
 current = final\_family\_id  
 descendant\_number(family\_matrix, all\_family\_id, entered, current, distance\_dictionary)  
 **return** distance\_dictionary

If the original family is not FID1, then the ‘distance’ from final\_family\_id and FID1 is the distance between the original family and FID1 plus the extra distance. A recursive call is then made

if the original chosen family is FID1, then the extra distance is the distance from FID1, so the extra\_distance is stored in the dictionary for the final\_family\_id. A recursive call is then made.

Gets a Family connected to the one already chosen,and finds the distance between them, to be either 1 or -1.

Base case is when the distance between every FamilyID and FID 1 (using the family \_matrix) has been found.

Gets a FamilyID from entered and gets the number row for which data for that familyID lies in.

**def** all\_families\_nested\_list(all\_family\_id, family\_matrix,distance\_dictionary): *####Gets all families in one list* entire\_families = []  
 conn = sqlite3.connect(**"Family\_1.db"**)  
 cursor = conn.cursor()  
 **for** family\_unit **in** all\_family\_id:  
 FamilyID = family\_unit  
 family = []  
 if distance\_dictionary==None:  
 generation\_number=None

Gets generation of family. If distance\_dictionary is none however, then an error would be thrown trying to get a distance from the dictionary. If this happens, generation\_number is set to None.

Goes through each familyID

Gets the children in the family

else:  
 generation\_number = distance\_dictionary[FamilyID]  
 family.append(FamilyID)  
 family.append(generation\_number)  
 FamilyID = **"'"** + FamilyID + **"'"** sql = (**"""  
 SELECT Mother,Father  
 FROM tblSingleFamily  
 WHERE tblSingleFamily.FamilyID = %s  
 """** % (FamilyID,))  
  
 **for** row **in** cursor.execute(sql):  
   
 row1=row[0]  
 row2=row[1]  
 row1 = person\_id\_format(row1)  
 row2=person\_id\_format(row2)  
 **if "Father" in** row1:  
 family.append(row2)  
 family.append(row1)  
 **else**:  
 family.append(row1)  
 family.append(row2)  
 family\_children = []  
  
 cursor.execute(**"""  
 SELECT PersonID  
 FROM tblChildren  
 WHERE tblChildren.FamilyID = %s  
 """** % (FamilyID,))  
 all\_children = cursor.fetchall()  
 all\_children = list\_format(all\_children)  
 family.append(all\_children)  
 entire\_families.append(family)  
 **return** entire\_families  
  
  
  
  
  
  
  
  
**def** create\_kivy\_matrix(family\_matrix1): *#### Gets rid of -1's and 0's for kivy, making processing easier later* kivy\_matrix = []  
 **for** x **in** range(0,len(family\_matrix1)):  
 **for** y **in** range(0,len(family\_matrix1)):  
 **if** family\_matrix1[x][y][0]==1 :  
 kivy\_matrix.append(family\_matrix1[x][y])  
 **return** kivy\_matrix  
  
*#===================================================================== Finds family Matrix, Entire families, kivy\_matrix and Distance Dictionary =========================================================***def** sorting\_main(): *#Gets all important information  
  
 #Gets families* all\_family\_id=[]   
 children\_dict=det\_children()   
 one\_unknown\_parent(children\_dict)  
 children\_dict = add\_unknown\_parents\_to\_children\_set(children\_dict)  
 unknownm, unknownf=add\_two\_unknowns(children\_dict)  
 children\_dict=add\_2\_unknowns\_to\_children\_set(unknownm,unknownf,children\_dict)  
 all\_family\_id=families(children\_dict,all\_family\_id)  
  
  
 *#Traverse algorithm* stack=create\_stack()  
 family\_matrix = create\_adjacency\_matrix()  
 visited = create\_visited\_family\_list()  
 family\_matrix, visited=traverse(visited, stack, family\_matrix, **"FID1"**)  
  
 *#Generations and Kivy info* distance\_dictionary={**"FID1"**:0}  
 entered=[]  
 distance\_dictionary=descendant\_number(family\_matrix,all\_family\_id,entered,**"FID1"**,distance\_dictionary)  
 distance\_dictionary=ChangeDistance.change\_distance\_dictionary(distance\_dictionary)  
 entire\_families=all\_families\_nested\_list(all\_family\_id,family\_matrix,distance\_dictionary)  
 family\_matrix1=copy.deepcopy(family\_matrix)  
 kivy\_matrix=create\_kivy\_matrix(family\_matrix1)  
 **return** entire\_families, family\_matrix,children\_dict, kivy\_matrix

Returns all families

Gets all of the children from a specific FamilyID

removes -1s, 0’s and empty lists from family\_matrix, to make the Kivy section easier to code

Gets the parents of the family

Changes distance\_dictionary so that the most recen generation is 0, and all the generations above it are positive integers, increasing by 1 each time you move up a generation

returns entire\_families, family\_matrix, children\_dict and kivy\_matrix

**Finding Relationships**

**import** sqlite3  
**import** copy  
**import** SortingAlgorithm\_09\_03\_2019 **as** Sort  
  
**def** get\_person\_id(): *#Gets all personID's in database* conn=sqlite3.connect(**"Family\_1.db"**)  
 cursor=conn.cursor()  
 cursor.execute(**"SELECT PersonID from tblFamily"**)  
 all\_person\_id=cursor.fetchall()  
 print(all\_person\_id)  
 **return** all\_person\_id  
  
**def** find\_families(entire\_families,person,relative): *# Finds families for chosen people* original\_person=copy.deepcopy(person)  
 original\_relative=copy.deepcopy(relative)  
 all\_person\_id=get\_person\_id()  
 **for** x **in** range(0,len(all\_person\_id)):  
 **if** all\_person\_id[x][0] **in** person:  
 person=all\_person\_id[x][0]  
 print(person)  
 **if** all\_person\_id[x][0] **in** relative:  
 relative=all\_person\_id[x][0]  
 **if "Mother" in** person:  
 adjustmentP=**"Child's"  
 elif "Father" in** person:  
 adjustmentP=**"Child's"  
 elif "\_P" in** person:  
 adjustmentP=**"Child's"  
 else**:  
 adjustmentP=**None  
   
 for** x **in** range(0,len(entire\_families)):  
 person\_family=entire\_families[x]  
 **if** adjustmentP!=**None**:  
 **if** (person\_family[2]==person) **or** (person\_family[3]==person):  
 person=person\_family[4][0]  
 **break  
 elif** adjustmentP==**None**:  
 **if** (person **in** person\_family[4]):  
 **break  
  
 if "Mother" in** relative:  
 adjustmentR=**"Mother"  
 elif "Father" in** relative:  
 adjustmentR=**"Father"  
 elif "\_P" in** relative:  
 adjustmentR=**"Parent"  
 else**:  
 adjustmentR=**None  
   
 for** y **in** range(0,len(entire\_families)):  
 relative\_family=entire\_families[y]  
 **if** adjustmentR!=**None**:  
 **if** (relative\_family[0]==relative) **or** (relative\_family[1]==relative):  
 relative=relative\_family[4][0]  
 **break  
 elif** adjustmentR==**None**:  
 **if** (relative **in** relative\_family[4]):  
 **break  
 return** adjustmentP,adjustmentR,person\_family,relative\_family, original\_person, original\_relative,person,relative  
  
   
   
**def** traverse(visited, stack, family\_matrix, current\_family,person\_family\_id,relative\_family\_id,foundP,foundR,person\_stack,relative\_stack):  
  
 **if** (foundP==**True and** foundR==**True**): *###Base case* **return** person\_stack, relative\_stack,foundR,foundP  
 **else**:  
 children\_dict=Sort.det\_children()  
 **if** current\_family **in** visited:  
 **pass  
 else**:  
 visited.append(current\_family)  
 current\_family=Sort.check\_format\_family\_id(current\_family)  
 parents, children = Sort.get\_family(str(current\_family))  
 family\_below=**False  
 for** x **in** range(0,len(children)):  
 child\_id = children[x]  
 parent\_family = Sort.check\_children(child\_id,children\_dict)  
 **if** parent\_family **is None**:  
 **pass  
 else**:  
 **for** y **in** range(0,len(parent\_family)):  
 parent\_familyY=parent\_family[y]  
 parent\_familyY=Sort.check\_format\_family\_id(parent\_family)  
 **if** parent\_familyY **in** visited:  
 **pass  
  
 else**:  
 family\_below=**True** next\_family = str(parent\_familyY)  
 next\_family = Sort.check\_format\_family\_id(next\_family)  
 child\_idY=child\_id  
 **break  
  
  
 if** family\_below==**True**:  
 current\_family = Sort.check\_format\_family\_id(current\_family)  
 stack = Sort.push(stack, current\_family)  
 **if** (next\_family== person\_family\_id) **and** (foundP==**False**):  
 person\_stack=copy.deepcopy(stack)  
 foundP=**True  
  
 if** (next\_family==relative\_family\_id) **and** (foundR==**False**):  
 relative\_stack=copy.deepcopy(stack)  
 foundR=**True** person\_stack,relative\_stack,foundR,foundP=traverse(visited, stack, family\_matrix, next\_family,person\_family\_id,relative\_family\_id,foundP,foundR,person\_stack,relative\_stack)  
 **return** person\_stack,relative\_stack,foundR,foundP  
  
 **elif** family\_below==**False**:  
  
 family\_above=**False  
 for** z **in** range(0,len(parents)):  
 ParentID = parents[z]  
 child\_family = Sort.check\_parents(ParentID)  
  
 **if** child\_family==**None or** child\_family==**"None"**:  
 child\_family=[]  
  
 **if** len(child\_family)>0:  
 child\_family=Sort.check\_format\_family\_id(child\_family)  
  
 **else**:  
 **pass  
 if** len(child\_family) ==0:  
 **pass  
 elif** len(child\_family)>0 **and** child\_family **not in** visited:  
 family\_above = **True  
 break  
 else**:  
 **pass  
  
 if** family\_above==**False and** len(stack)>=1:  
  
 stack, top\_of\_stack = Sort.pop(stack)  
 **if** (top\_of\_stack== person\_family\_id) **and** (foundP==**False**):  
 person\_stack=copy.deepcopy(stack)  
 foundP=**True  
  
 if** (top\_of\_stack==relative\_family\_id) **and** (foundR==**False**):  
 relative\_stack=copy.deepcopy(stack)  
 foundR=**True** person\_stack,relative\_stack,foundR,foundP=traverse(visited, stack, family\_matrix,

Imports:  
sqlite3 – allows databases to be manipulated  
copy – allows the value of variables to be copied to other variables  
SortingAlgorithm\_09\_03\_2019 – allows family\_matrix, and other variables at the bottom of the previous page to be created, as well as other functionalities.

Gets the family which person is the child of

The whole ‘finding\_relationships’ option relies on getting the family where both ‘person’ and ‘relative are children. If person or relative is contains ‘Mother’,’Father’ or ‘\_P’, then person or relative will not be a child of any family. So, the process is carried out using one of their children, and the final relationship is then adjusted later to account for the fact that the parent of person/relative was used to find the relationship between person and relative.

Gets the PersonID of person and relative, from the drop-down menu choice

The variables ‘original\_person’ and ‘original\_relative will store the original values of person and relative before person and relative change. I cannot just say person=original\_person for example, as then when person changes, original\_person will automatically change as well

Gets all Person ID’s in the database

Gets the family which relative is the child of

Similar to previous traverse algorithm, but with some tweaks and a different base case. The purpose of this algorithm is to find the route from “FID1” to person\_family and from “FID1” to “relative\_family

Base case – returns person\_stack and relative\_stack when both person\_family and relative\_family have been visited.

Gets dictionary of children and appends current\_family to visited if it has not been visited yet.

If there is a family below and next\_family is person\_family or relative\_family and foundP/foundR is still False, the stack is deep-copied to person\_stack or relative\_stack, and foundP or foundR becomes true. A recursive call is then made.

If there is no family above or below but the top of the stack is person\_family or relative\_family and foundP/foundR is still False, the stack is deep-copied to person\_stack or relative\_stack, and foundP or foundR becomes true. A recursive call is then made.

deep-copied to person\_stack or relative\_stack, and foundP or foundR becomes true. A recursive call is then made.

top\_of\_stack,person\_family\_id,relative\_family\_id,foundP,foundR,person\_stack,relative\_stack)  
 **return** person\_stack,relative\_stack,foundP,foundR  
  
 **elif** family\_above==**False and** len(stack)==0 **and** len(family\_matrix)==len(visited):  
 **return** family\_matrix, visited  
 **elif** family\_above==**False and** len(stack)==0 **and** len(all\_family\_id)!=len(visited):  
 found=**False  
 while** found==**False**:  
 **for** x **in** range(0,len(all\_family\_id)):  
 **if** all\_family\_id[x] **not in** visited:  
 found=**True** current\_family=all\_family\_id[x]  
 person\_stack, relative\_stack, foundR, foundP =traverse (visited,stack,family\_matrix, current\_family,person\_family\_id,

If a new family has to be used a starting place because the tree is unconnected, do not check whether the new starting family is person\_family or relative\_family yet ( this can be done later on during the execution of the subroutine.

relative\_family\_id,foundP,foundR,person\_stack,relative\_stack)  
 **return** person\_stack,relative\_stack,foundR,foundP  
  
  
 **elif** family\_above == **True**:  
 child\_family=Sort.check\_format\_family\_id(child\_family)  
 ParentID=**None** stack = Sort.push(stack, current\_family)  
 **if** (child\_family== person\_family\_id) **and** (foundP==**False**):  
 person\_stack=copy.deepcopy(stack)  
 foundP=**True  
  
 if** (child\_family==relative\_family\_id) **and** (foundR==**False**):  
 relative\_stack=copy.deepcopy(stack)  
 foundR=**True** person\_stack,relative\_stack,foundR,foundP=traverse(visited, stack, family\_matrix,

If there is a family above and next\_family is person\_family or relative\_family and foundP/foundR is still False, the stack is deep-copied to person\_stack or relative\_stack, and foundP or foundR becomes true. A recursive call is then made.

child\_family,person\_family\_id,relative\_family\_id,foundP,foundR,person\_stack,relative\_stack)  
 **return** person\_stack,relative\_stack,foundR,foundP  
  
**def** find\_route\_between\_people(person\_stack,relative\_stack,person\_family\_id,relative\_family\_id,family\_matrix): *#Gets route between families* no\_duplicates=**False** count=0  
 route=[]  
 **if** person\_family\_id **not in** person\_stack:  
 person\_stack.append(person\_family\_id)  
 person\_stack\_length = len(person\_stack)  
 **if** relative\_family\_id **not in** relative\_stack:  
 relative\_stack.append(relative\_family\_id)  
 **for** x **in** range(len(person\_stack)-1,-1,-1):  
 route.append(person\_stack.pop(x))  
 **for** y **in** range(0,len(relative\_stack)):  
   
 route.append(relative\_stack.pop(0))  
  
 **while** no\_duplicates==**False**:  
 **try**: *#Used in case index number goes outside range, in which case, a familyID from the route may not need to be deleted* current\_fam\_num=int(route[person\_stack\_length-1-count][3:len(route[person\_stack\_length-count-1])])-1  
 next\_fam\_num=int(route[person\_stack\_length-count][3:len(route[person\_stack\_length-count])])-1  
  
 **except**:

Adds person\_family\_id and relative\_family\_id to route if they were not added to the route during the traverse algorithm. The route is also initially formed here. It is the person\_stack popped in reverse, and then the relative\_stack following after the person\_stack families.

If current\_fam\_num or next\_fam\_num cannot be found due to a ‘list out of range error’, then they are given an arbitrary number of ‘999’, which indicates that nothing else at all should be deleted from route.

Initially enters the route as person\_stack in reverse and then relative\_stack appended.

Current\_fam\_num initially represents the 2 Family ID’s at the point where the families popped from person\_stack changes to relative\_stack.

Arbitrary choices of 999 for current\_fam\_num and next\_fam\_num if an ‘out of range’ error occurs when trying to obtain them. A value of 999 indicates that there are definitely no more duplicates.

current\_fam\_num=999  
 next\_fam\_num=999  
  
 **try**: *#Used in case index number goes outside range, in which case, a familyID from the route may not need to be deleted* after\_current\_fam\_num= int(route[person\_stack\_length-2-count][3:len(route[person\_stack\_length-count-2])])-1  
 after\_next\_fam\_num=int(route[person\_stack\_length-count+1][3:len(route[person\_stack\_length-count+1])])-1  
  
 **except**:  
 after\_current\_fam\_num=1  
 after\_next\_fam\_num=1  
  
 **if** (int((family\_matrix[after\_current\_fam\_num][after\_next\_fam\_num][0])==0) **and** after\_current\_fam\_num!=after\_next\_fam\_num) **or**

If after\_current\_fam num and after\_next\_fam\_num cannot be found due to an out of range error, they are both assigined the same value of 1, as in this case we would want to delete both

If the route [FID5,FID8,FID1,FID1,FID8,FID9,FID6], the program would first see that FID1 has been repeated and that there are two FID8’s. So, both FID1’s would be deleted. But, if the first FID8 is changed to FID7, then both FID1’s would be deleted only if there is a connection  
between FID7 and FID8. Otherwise, only one of the FID1’s would be deleted.

(current\_fam\_num!=next\_fam\_num):  
 delete\_both=**False  
 else**:  
 delete\_both=**True  
 if** delete\_both==**True**: *#(route[person\_stack\_length-count]==route[person\_stack\_length-count+1]) and* route.pop(person\_stack\_length-count-1)  
 route.pop(person\_stack\_length-count-1)  
 **if** len(route)<3:  
 no\_duplicates=**True** count+=1  
 **if** current\_fam\_num!=999:  
  
 **if** delete\_both==**False and** (route[person\_stack\_length-1-count]==route[person\_stack\_length-count]):  
 route.pop(person\_stack\_length-count-1)  
 no\_duplicates=**True  
 else**:   
   
 no\_duplicates=**True  
 if** route[0]!=person\_family\_id:  
 route.insert(0,person\_family\_id)  
 **if** route[len(route)-1]!=relative\_family\_id:  
 route.append(relative\_family\_id)  
  
 **return** route  
  
  
  
**def** person\_id\_format(variable):  
 **if** type(variable) **is** tuple:  
 variable=variable[0]  
  
 variable.replace(**"("**,**""**)  
 variable.replace(**")"**,**""**)  
 variable.replace(**","**,**""**)  
 **return** variable  
  
  
**def** get\_firstname(person):  
 conn = sqlite3.connect(**"Family\_1.db"**)  
 cursor = conn.cursor()  
 sql = **"""  
 SELECT Firstname  
 FROM tblFamily  
 Where PersonID = ?  
 """  
 for** row **in** cursor.execute(sql, (person,)):  
 pre\_firstname = (row[0])  
  
 conn.commit()  
 conn.close()  
 **return** pre\_firstname  
  
  
**def** get\_surname(person):  
 conn = sqlite3.connect(**"Family\_1.db"**)  
 cursor = conn.cursor()  
 sql = **"""  
 SELECT Surname  
 FROM tblFamily  
 Where PersonID = ?  
 """  
 for** row **in** cursor.execute(sql, (person,)):  
 pre\_surname = (row[0])  
 conn.commit()  
 conn.close()  
 **return** pre\_surname  
  
  
**def** get\_family\_from\_person(person):  
 conn = sqlite3.connect(**"Family\_1.db"**)  
 cursor = conn.cursor()  
 sql = **"""  
 SELECT FamilyID  
 FROM tblChildren  
 WHERE PersonID=?  
 """  
 for** row **in** cursor.execute(sql, (person,)):  
 person\_family = row[0]  
  
 **return** person\_family  
  
**def** get\_person\_gender(personX): *# Gets gender of person* conn= sqlite3.connect(**"Family\_1.db"**)  
 cursor= conn.cursor()  
 sql = **"""  
 SELECT Gender  
 FROM tblFamily  
 Where PersonID = ?  
 """  
 for** row **in** cursor.execute(sql,(personX,)):  
 person\_gender=(row[0])  
 conn.commit()  
 conn.close()  
 **return** person\_gender  
  
**def** get\_children(person\_gender,personX):  
 children\_set=set()  
 conn = sqlite3.connect(**"Family\_1.db"**)  
 cursor = conn.cursor()  
 **if** person\_gender == **"Female"**:  
 sql = **"""  
 SELECT PersonID  
 FROM tblFamily  
 WHERE MotherID = ?  
 """  
 for** row **in** cursor.execute(sql,(personX,)):  
 child\_id = row[0]  
 child\_id=person\_id\_format(child\_id)  
 children\_set.add(child\_id)  
 **elif** person\_gender == **"Male"**:  
 sql = **"""  
 SELECT PersonID  
 FROM tblFamily  
 WHERE FatherID = ?  
 """  
 for** row **in** cursor.execute(sql,(personX,)):  
 child\_id = row[0]  
 child\_id=person\_id\_format(child\_id)  
 children\_set.add(child\_id)  
   
 children\_set.discard(**"set()"**)  
 **return** children\_set  
  
**def** people\_in\_route(route, person, relative,family\_matrix,adjustmentP,adjustmentR,original\_person,original\_relative): *#Changes familyID's to respective people* pen\_family\_num=int(int((route[len(route)-2][3:len(route[len(route)-2])])) - 1)  
 final\_family\_num = int(int((route[len(route)-1][3:len(route[len(route)-1])]))-1)  
 **if** family\_matrix[pen\_family\_num][final\_family\_num][0]==1:  
 child=**"Y"  
 else**:

If the length of the route is 1 or 2, then clearly no more deletions need to happen.

‘Count’ is adjusted every time 2 people are deleted, to make sure that the right families are being compared in the while loop when it restarts

Adds person\_family and relative\_family, in case they are not in the route anymore

Only deletes one of the repeating families

Deletes both ID’s in question from route.

Ensures a PersonID is of the correct format

Gets the first name of person

Gets the surname of person

Gets the familyID that a person is the child of

Gets the gender of personX

Gets the children of a person

This subroutine takes every consecutive family, and finds the linking person

Relative needs to be treated differently. Child determines whether child is above or below the penultimate family.

child=**"N"** people\_route=[]  
 **if** adjustmentP!=**None** :  
 people\_route.insert(0,original\_person)  
 **elif** adjustmentP==**None**:  
 people\_route.append(person)  
 **for** x **in** range(0,len(route)-1):  
 **if** x==0 **and** adjustmentP!=**None**:  
 person\_gender=get\_person\_gender(original\_person)  
 person\_gender=person\_gender[0]  
 people\_route.append(person+**"\_C"**+**"\_"**+str(person\_gender))  
 **else**:   
 family\_1\_num=int(int((route[x][3:len(route[x])]))-1)  
 family\_2\_num=int(int((route[x+1][3:len(route[x+1])]))-1)  
 next\_person=family\_matrix[family\_1\_num][family\_2\_num][1]  
 next\_person\_gender=get\_person\_gender(next\_person)  
 next\_person\_gender=next\_person\_gender[0]  
 **if** family\_matrix[family\_1\_num][family\_2\_num][0]==1:  
   
 people\_route.append(next\_person + **"\_C"** + **"\_"**+ str(next\_person\_gender))  
   
 **elif** family\_matrix[family\_1\_num][family\_2\_num][0]==-1:  
 people\_route.append(family\_matrix[family\_1\_num][family\_2\_num][1] +**"\_A"**+**"\_"**+ str(next\_person\_gender))  
 **if** adjustmentR!=**None**:  
 relative\_gender=get\_person\_gender(original\_relative)  
 people\_route.append(original\_relative+**"\_A"**+**"\_"**+str(relative\_gender))  
  
 **if** person **in** people\_route[1]:  
 people\_route.pop(1)  
 **return** people\_route,child  
  
  
  
**def** decide\_relationship(current\_person,next\_person,siblings,x,people\_route\_length,married): *# Finds preliminary relatinships* **if** (**"\_A" in** next\_person) **and** (**"\_M" in** next\_person) **and** siblings==**False and** married==**False**:  
 relationship=**"Father's"  
 elif** (**"\_A" in** next\_person) **and** (**"\_M" in** next\_person) **and** siblings==**True and** married==**False**:  
 relationship = **"Brother's"  
 elif** (**"\_A" in** next\_person) **and** (**"\_F" in** next\_person) **and** siblings==**False and** married==**False**:  
 relationship=**"Mother's"  
 elif** (**"\_A" in** next\_person) **and** (**"\_F" in** next\_person) **and** siblings==**True and** married==**False**:  
 relationship=**"Sister's"  
 elif** (**"\_C" in** next\_person) **and** (**"\_M" in** next\_person) **and** siblings==**False and** married==**False**:  
 relationship=**"Son's"  
 elif** (**"\_C" in** next\_person) **and** (**"\_M" in** next\_person) **and** siblings==**True and** married==**False**:  
 relationship= **"Brother's"  
 elif** (**"\_C" in** next\_person) **and** (**"\_F" in** next\_person) **and** siblings==**False and** married==**False**:  
 relationship=**"Daughter's"  
 elif** (**"\_C" in** next\_person) **and** (**"\_F" in** next\_person) **and** siblings==**True and** married==**False**:  
 relationship=**"Sister's"  
  
 elif** married==**True and** (**"\_F" in** next\_person):  
 relationship= **"Wife's"  
 elif** married==**True and** (**"\_M" in** next\_person):  
 relationship=**"Husband's"  
   
  
 return** relationship  
   
   
   
  
  
**def** route\_with\_relationships(people\_route,person,relative,adjustmentP,adjustmentR,original\_person,original\_relative,child): *# Changes ID's in path to relationships and names  
 # at either end* **if** adjustmentP!=**None**:  
 person\_firstname=get\_firstname(original\_person)  
 person\_surname=get\_surname(OringinalPerson)  
 person\_full\_name= person\_firstname + **" "** + person\_surname  
 **else**:  
 person\_firstname= get\_firstname(person)  
 person\_surname = get\_surname(person)  
 person\_full\_name= person\_firstname + **" "** + person\_surname  
  
 **if** adjustmentR!=**None**:  
 relative\_firstname=get\_firstname(original\_relative)  
 relative\_surname=get\_surname(original\_relative)  
 relative\_full\_name= relative\_firstname + **" "** + relative\_surname  
 **else**:  
 relative\_firstname= get\_firstname(relative)  
 relative\_surname = get\_surname(relative)  
 relative\_full\_name= relative\_firstname + **" "** + relative\_surname  
   
 path=person\_full\_name + **"'s"** people\_route\_length=len(people\_route)  
 **for** x **in** range(0,people\_route\_length-1):  
 current\_person=people\_route[x]  
 next\_person=people\_route[x+1]  
 current\_family=get\_family\_from\_person(current\_person[0:6])  
 next\_family=get\_family\_from\_person(next\_person[0:6])  
 **if** current\_family==next\_family:  
 siblings=**True  
 else**:  
 siblings=**False** current\_person\_gender=Sort.get\_person\_gender(current\_person[0:6])  
 next\_person\_gender=Sort.get\_person\_gender(next\_person[0:6])  
 current\_person\_children=Sort.get\_children(current\_person\_gender,current\_person[0:6])  
 next\_person\_children=Sort.get\_children(next\_person\_gender,next\_person[0:6])  
 **if** len(current\_person\_children.intersection(next\_person\_children))!=0 **and** len(current\_person\_children)!=0 **and**

Linking person found by consulting family\_matrix, which already has that appended to each nested list where there is a connection.

If there is an adjustment for relative, original\_relative is appended

“\_C” indicates that the next\_person is a child of the person before them in the route. “\_A indicates that next\_person is a parent of the previous person. “\_M” or “\_F” is also appended depending on whether the next person is male or female respectively.

If there is an adjustment, original\_person is appended first.

PersonID’s in the route are swapped out for relationships, depending on the state of some variables.

Each pair of consecutive people are dealt with in turn, and the relationship between them is found using the “\_C/A” and “\_M/F” suffixes in the find\_route\_between\_people subroutine, and by determining whether the two people are siblings or married. If they are siblings, they will both be children of the same family and if they are ‘married’, they will share children.

Gets the full name of ‘person’ and ‘relative’, and ‘person’s’ full name and adds it to the route, with “’s” attached to the end of it. Same process carried out if there are adjustments.

If the 2 people are children of the same family, they are siblings.

len(next\_person\_children)!=0:  
 married=**True  
 else**:  
 married=**False** relationship=decide\_relationship(current\_person,next\_person,siblings,x,people\_route\_length,married)  
 path= path + **" "** + relationship  
 **if** relative **not in** people\_route[len(people\_route)-1]:  
 gender\_relative = get\_person\_gender(relative)  
 **if** gender\_relative==**'Male' and** child==**"Y"**:  
 path=path + **" "** + **"Son's"**

If the 2 people share at least one child, they are deemed to be married/in a relationship

**elif** gender\_relative==**'Male' and** child==**"N" and** (get\_family\_from\_person(relative)!= get\_family\_from\_person(people\_route[0:6])):  
 path = path + **" "** + **"Father's"**

**elif** gender\_relative==**'Female' and** child==**"Y"**:  
 path = path + **" "** + **"Daughter's"**

**elif** gender\_relative==**'Female' and** child==**"N" and** (get\_family\_from\_person(relative)!=

get\_family\_from\_person(people\_route[len(people\_route)-1][0:6])):  
 path = path + **"Mother's"**

**elif** child==**"N" and** gender\_relative==**"Male" and** (get\_family\_from\_person(relative)==

get\_family\_from\_person(people\_route[len(people\_route)-1][0:6])):  
 path=path + **"Brother's"**

**elif** child==**"N" and** gender\_relative==**"Female" and** (get\_family\_from\_person(relative)==

get\_family\_from\_person(people\_route[len(people\_route)-1][0:6])):  
 path = path + **"Sister's"** path=path + **" "** + **"is"** + **" "** + relative\_full\_name  
 **return** path  
  
**def** path\_edit(path):  
 **for** x **in** range(0,6): *# Replaces prelimnary relationships with better ones.* path=path.replace(**"Father's Father's"**,**"GrandFather's"**)  
  
 path=path.replace(**"Father's Mother's"**,**"GrandMother's"** )  
  
 path=path.replace(**"Father's Son's"**,**"Brother's"**)  
  
 path=path.replace(**"Father's Daughter's"**,**"Sister's"**)  
  
 path=path.replace(**"Father's Sister's"**,**"Auntie's"**)  
  
 path=path.replace(**"Father's Brother's"**,**"Uncle's"**)  
  
 path=path.replace(**"Brother's Son's"**, **"Nephew"**)  
  
 path=path.replace(**"Uncle's Son's"**, **"Cousin's"**)  
  
 path=path.replace(**"Brother's Daughter's"**,**"Niece"**)  
  
 path=path.replace(**"Uncle's Daughter's"**,**"Cousin's"**)  
  
 path=path.replace(**"Wife's Brother's"**,**"Brother-in-Law's"**)  
  
 path=path.replace(**"Wife's Sister's"**, **"Sister-in-Law's"**)  
   
 path = path.replace(**"GrandFather's Father's"**,**"Great GrandFather's"**)  
  
 path= path.replace(**"GrandFather's Mother's"**, **"Great GrandMother's"**)  
  
 path= path.replace(**"GrandFather's Brother's"**,**"Great Uncle's"**)  
  
 path=path.replace(**"GrandFather's Sister's"**, **"Great Auntie's"**)  
   
  
 *#Mother* path=path.replace(**"Mother's Father's"**,**"GrandFather's"**)  
  
 path=path.replace(**"Mother's Mother's"**,**"GrandMother's"**)  
  
 path=path.replace(**"Mother's Son's"**,**"Brother's"**)  
  
 path=path.replace(**"Mother's Daughter's"**,**"Sister's"**)  
  
 path=path.replace(**"Mother's Sister's"**,**"Auntie's"**)  
  
 path=path.replace(**"Mother's Brother's"**,**"Uncle's"**)  
  
 path=path.replace(**"Mother's Sister's"**,**"Auntie's"**)  
  
 path=path.replace(**"Mother's Brother's"**,**"Uncle's"**)  
  
 path=path.replace(**"Sister's Son's"**, **"Nephew"**)  
  
 path=path.replace(**"Auntie's Son's"**, **"Cousin's"**)  
  
 path=path.replace(**"Sister's Daughter's"**,**"Niece"**)  
  
 path=path.replace(**"Auntie's Daughter's"**,**"Cousin's"**)  
  
 path=path.replace(**"Husband's Brother's"**,**"Brother-in-Law's"**)  
  
 path=path.replace(**"Husbands's Sister's"**, **"Sister-in-Law's"**)  
  
 path = path.replace(**"Sister's Husbands's"**, **"Sister-in-Law's"**)  
  
 path = path.replace(**"GrandMother's Father's"**,**"Great GrandFather's"**)  
  
 path= path.replace(**"GrandMother's Mother's"**, **"Great GrandMother's"**)  
  
 path= path.replace(**"GrandMother's Brother's"**,**"Great Uncle's"**)  
  
 path=path.replace(**"GrandMother's Sister's"**, **"Great Auntie's"**)  
  
 *#path.replace(" ", " ")  
 #path.replace(" ", " ")  
 #path.replace(" ", " ")* **return** path  
  
  
**def** path\_correction(path): *# removes last 's* **for** x **in** range(0,len(path)):  
 **if** path[x]==**"'" and** path[x+1]==**"s"**:  
 y=x  
  
 path=path[:y] + path[y+1:]  
 path=path[:y] + path[y+1:]  
 **return** path  
   
   
   
   
  
**def** relations\_main(person,relative,entire\_families,family\_matrix):  
 entire\_families, family\_matrix, children\_dict, kivy\_matrix= Sort.sorting\_main()  
 visited = Sort.create\_visited\_family\_list()  
 stack=Sort.create\_stack()  
 foundP=**False** foundR=**False** adjustmentP,adjustmentR,person\_family,relative\_family,original\_person,original\_relative,person,relative=find\_families(entire\_families,

Relative is then dealt separately here, by using ‘child’ variable from a previous algorithm, and by determining if relative and the penultimate person are children of the same family etc.

Preliminary relationships such as mother, brother etc. are combined to give more concise relationships such as ‘Uncle’ and ‘Nephew’

This subroutine exists to correct some of the grammar in the description of the relationship, by removing the last ‘s in the last ‘relationship’.

Pulls everything together and returns the path from the person to the relative.

person,relative)  
 person\_family\_id=person\_family[0]  
 relative\_family\_id=relative\_family[0]  
 person\_stack, relative\_stack,foundR,foundP=traverse(visited, stack, family\_matrix,

**“FID1”**,person\_family\_id,relative\_family\_id,foundP,foundR,[],[])  
 route=find\_route\_between\_people(person\_stack,relative\_stack,person\_family\_id,relative\_family\_id,family\_matrix)  
 people\_route,child=people\_in\_route(route,person,relative,family\_matrix,adjustmentP,adjustmentR,person,relative)  
 path=route\_with\_relationships(people\_route,person,relative,adjustmentP,adjustmentR,original\_person,original\_relative,child)  
 path=path\_edit(path)  
 path=path\_correction(path)  
 **return** path

**Get Statistics**

Imports:  
sqlite3 : allows databases to be manipulated  
datetime: allows string of the form DD-MM-YYYY or other permutations to be changed to a ‘date’ format.

**import** sqlite3  
**from** datetime **import** datetime,date

Gets everyone in the database who has a non-empty date of birth and an empty date of death (i.e. they are alive)

**def** list\_all\_alive():  
 all\_people=[]  
 conn = sqlite3.connect(**"Family\_1.db"**)  
 cursor = conn.cursor()  
 **for** row **in** cursor.execute(**'SELECT PersonID,DOB,DOD FROM tblFamily'**):  
 **if** row[1]!=**"" and** row[2]==**""**:  
 all\_people.append(row[0])  
  
 conn.commit()  
 conn.close()  
  
 **return** all\_people   
  
**def** calc\_current\_age(DOB):  
 date\_today=date.today()  
 DOB = DOB[0:4] + **" "** + DOB[5:7] + **" "** + DOB[8:10]  
 DOB = datetime.strptime(DOB, **"%Y %m %d"**)  
 **return** date\_today.year - DOB.year - ((date\_today.month, date\_today.day) < (DOB.month, DOB.day))   
  
**def** get\_ages():  
 ages\_dict={}  
 max\_person\_age=[**""**,0]  
 conn= sqlite3.connect(**"Family\_1.db"**)  
 cursor= conn.cursor()  
 sql = **"""  
 SELECT PersonID, DOB,DOD  
 FROM tblFamily  
 """** *#NEED TO EXCLUDE DEAD PEOPLE LATER* **for** row **in** cursor.execute(sql):  
 DOB = row[1]  
 **if** DOB!=**"" and** row[2]==**""**:  
 current\_age=calc\_current\_age(DOB)  
 ages\_dict[row[0]] = current\_age  
  
 **return** ages\_dict  
  
**def** calc\_mean():  
 ages\_dict = get\_ages()  
 all\_people=list\_all\_alive()  
 total\_age=0  
 **for** x **in** range(0,len(all\_people)):  
 total\_age= total\_age + int(ages\_dict[all\_people[x]])  
 mean\_age = total\_age/len(all\_people)  
  
 **return** mean\_age  
   
   
**def** calc\_square\_root(number):  
 interval=10  
 start =0  
 start=float(start)  
 **for** x **in** range(0,11):  
 found=**False** interval=float(interval/10)  
 **while** found==**False**:  
 **if** ((start\*\*2) <= float(number) **and** (start+interval)\*\*2 >= float(number)):  
 found = **True  
   
 else**:  
 start=start+interval  
 **return** round(start,10)  
   
**def** calc\_var():  
 ages\_dict=get\_ages()  
 all\_people=list\_all\_alive()  
 ages\_squared=0  
 mean\_age=calc\_mean()  
 **for** person **in** all\_people:  
 ages\_squared=ages\_squared + ((ages\_dict[person])\*\*2)  
 variance = (ages\_squared/len(all\_people)) - (mean\_age\*\*2)  
 **return** variance  
  
**def** calc\_sd():  
 variance=calc\_var()  
 standard\_deviation = calc\_square\_root(variance)  
 **return** standard\_deviation  
  
**def** just\_ages(ages\_dict,all\_people):  
 just\_ages\_list=[]  
 **for** person **in** all\_people:  
 **if** person **in** ages\_dict:  
 just\_ages\_list.append(ages\_dict[person])  
   
 **return** just\_ages\_list  
  
**def** median(just\_ages\_list):  
 just\_ages\_list=sorted(just\_ages\_list)  
 **if** len(just\_ages\_list)%2==1:  
 median\_age = just\_ages\_list[int((len(just\_ages\_list)+1)/2)-1]  
 **else**:  
 median\_age = (just\_ages\_list[int(len(just\_ages\_list)/2)-1] + just\_ages\_list[int((len(just\_ages\_list)/2))])/2  
  
 **return** median\_age  
  
**def** mode(just\_ages\_list):  
 **return** max(set(just\_ages\_list), key=just\_ages\_list.count)  
  
**def** max\_age(just\_ages\_list,ages\_dict,all\_people):  
 max\_age\_people=[]  
 max\_age=max(just\_ages\_list)  
 **for** person **in** all\_people:  
 **if** ages\_dict[person]==max\_age:  
 max\_age\_people.append(person)  
 **return** max\_age,max\_age\_people  
  
**def** min\_age(just\_ages\_list,ages\_dict,all\_people):  
 min\_age\_people=[]  
 min\_age = min(just\_ages\_list)  
 **for** person **in** all\_people:  
 **if** ages\_dict[person] == min\_age:  
 min\_age\_people.append(person)  
 **return** min\_age, min\_age\_people  
  
  
  
  
**def** statistics\_main():  
 all\_people=list\_all\_alive()  
 ages\_dict=get\_ages()  
 just\_ages\_list=just\_ages(ages\_dict,all\_people)  
 mode\_age =mode(just\_ages\_list)  
 median\_age= median(just\_ages\_list)  
 mean\_age=calc\_mean()  
 variance=calc\_var()  
 standard\_deviation=calc\_sd()  
 max\_age\_people,max\_age\_people\_list=max\_age(just\_ages\_list, ages\_dict,all\_people)  
 min\_age\_people, min\_age\_people\_list = min\_age(just\_ages\_list,ages\_dict,all\_people)  
 **return** mode\_age,median\_age, mean\_age,variance,standard\_deviation, max\_age\_people,max\_age\_people\_list,min\_age\_people,min\_age\_people\_list

Gets the ages of everyone in the database who is alive, and puts this into a dictionary where each person ID is a key and their value is their age.

Calculates the current age of a person, by putting their date of birth into the form YYYY-MM-DD using datetime and then subtracting this away from the date today.

Adds the ages of everyone alive and then divides by the total number of people alive to find the mean.

Calculates the square root of a number correct to 10 decimal places (could change this to any degree of accuracy by changing the range to another number instead of 11. It works by starting from zero, and finding 2 numbers where the squaring each of them, means that one is less than ‘number’ and one is greater than ‘number’. Interval is then divided by 10, and the process is repeated, but with the start being the lower number instead of ‘0’.

Calculates the variance of the ages by adding up the square of all the ages, dividing it by the number of people alive, and then taking away the mean squared (A-Level maths)

The square root of the variance

Gets a list of all the ages (no dictionary)

Sorts just\_ages\_list into ascending order.

If there are an even number of ages, the median is the mean of the middle two terms. If there are an odd number of ages, then the median is the middle term

Gets the most common age in just\_ages\_list.

Finds the max age in just\_ages\_list, and finds the people who are this age.

Finds the min age in just\_ages\_list , and returns the people who are this age.

Ties everything together and calculates all of the above statistics

**Draw Tree**

Imports:  
App: Allows Kivy to run (is necessary)  
Window: Creates a handle on the window to enable window height and width to be accessed.  
Relative layout: Objects on-screen stay in proportion in size and distance when window is re-sized  
Builder: Is needed for Kivy to parse a string passed to it and create the on-screen widgets from it.  
Counter: Used here with a filter to return the unique elements in a list which contains duplicates.

**from** kivy.app **import** App  
**from** kivy.core.window **import** Window  
**from** kivy.uix.relativelayout **import** RelativeLayout  
**from** kivy.lang **import** Builder  
**from** collections **import** Counter

**class** FamilyTreeCanvas(RelativeLayout):  
  
 **def** \_\_init\_\_(self, \*\*kwargs):  
 super(FamilyTreeCanvas, self).\_\_init\_\_(\*\*kwargs)  
  
 **def** display\_info(self, person\_id):  
 print (person\_id)  
 person = get\_person\_info(person\_id, self.FIDS2)  
 disp = person[**'name'**] + **'\n'** disp+= **'DOB: '** + person[**'dob'**]  
 disp+= (**' DOD: '** + person[**'dod'**]) **if** (person[**'dod'**] != **''**) **else ''** disp+= **'\n'** disp+= (**'Place of birth: '** + person[**'pob'**]) **if** (person[**'pob'**] != **''**) **else ''** self.ids.info\_disp.text = disp  
  
  
  
**class** Family:  
 **def** \_\_init\_\_(self, id, gen, mum, dad, kids):  
 self.id = id  
 self.gen = gen  
 self.mum = mum  
 self.dad = dad  
 self.kids = kids  
 self.downlinks = []  
 self.pos\_hint = []  
 self.size\_hint = []  
  
  
  
**def** get\_person\_info (person\_id, FIDS2):  
 **if** person\_id **in** FIDS2:  
 vals = FIDS2[person\_id]  
 v = {**"pid"**: vals[0],**"firstname"**: vals[1],**"surname"**: vals[2],**"name"**: vals[1] + **' '** + vals[2],**"dob"**: vals[3],**"dod"**: vals[4],**"pob"**:

Instantiates a family, with passed parameters such generation, mum, dad and kids.  
Also initiates the downlinks, pos\_hint and size\_hint attributes as blank lists that will be updated later.

This class matches the name of the top-level widget that is created on-screen. It is required by Kivy in order to connect the code to the widget and bind behaviour and navigate the widget tree.

display\_info function: Displays information such as name, date of birth, place of birth at the bottom-left of the screen when a person button is clicked.

Gets all information for a specific person from the FIDS2 dictionary

vals[5], **"gender"**: vals[6],  
 **"mother"**: vals[7],**"father"**: vals[8]}  
 **return** v  
 **else**:  
 print (**"PersonID not found in FIDS2"**)  
 v = {**"pid"**: person\_id,**"firstname"**: **'NA'**,**"surname"**: **'NA'**,**"name"**: **'Not found'**,**"dob"**: **'0'**,**"dod"**: **'0'**,**"pob"**: **'NA'**,**"gender"**: **'NA'**,**"mother"**: **'NA'**,**"father"**: **'NA'**}  
 **return** v

In case a person is not found, ‘NA’ and ‘not found’ are set for their information.

**def** families\_dict (FIDS1, FIDS2):  
 *# families{} - THIS IS THE DICTIONARY FOR STORING INFO ON EACH FAMILY IN THE FAMILY TREE* families = {} *# a dictionary of Family objects  
 # run through FIDS1 and then grab all the extra info from FIDS2 using get\_person\_info()  
 # example of FIDS1: ['FID11', 2, 'SK1698', 'TK3231', ['DK6984', 'IK8120']]  
 # Also, run through FIDS3 and get the connections to other families  
 # example of FIDS3: [1, 'NA9171', 'FID1', 'FID8']* **for** item **in** FIDS1:  
 fam\_id = item[0] *# the family id* fam\_gen = item[1] *# the generation number of the family* fam\_mum = get\_person\_info(item[2], FIDS2) *# returns a dictionary for the mother of the family* fam\_dad = get\_person\_info(item[3], FIDS2) *# returns a dictionary for the father of the family* fam\_kids = [] *##returns a LIST of dictionaries, one for each child of the family* **for** kid\_id **in** item[4]:  
 fam\_kids.append(get\_person\_info(kid\_id, FIDS2)) *# returns a dictionary for each child in the family  
  
 # make a Family object with all this info and add it to the families dictionary* newfam = Family(fam\_id, fam\_gen, fam\_mum, fam\_dad, fam\_kids)  
 families[fam\_id] = newfam  
  
 **return** families

Make the Family object.  
Add it to the **families** dictionary.

Using FIDS1 and FIDS2, builds up all the information needed to make a Family object, including info-dictionaries for each family member, made by calling the get\_person\_info subroutine.

Builds a dictionary called **families** using family id as a key and returning a ‘Family’ object for each family in the family tree.

**def** get\_unique\_gens (FIDS1):  
 *# Get some info on the generations of the family tree* gens = []  
 **for** item **in** FIDS1:  
 gens.append(item[1]) *# item 1 is the generation number* gens.sort() *# sort lowest to highest, eg [-1, 0, 0, 0, 1, 1, 1, 2, 2, 2, 2]*

Returns a sorted, unique list of generation numbers from the FIDS1 list

*# adjust all the generation nummbers if needed so that the lowest descendants are at generation 0* gens\_adjust = 0 - gens[0]  
 **if** gens\_adjust != 0:  
 gens = [element + gens\_adjust **for** element **in** gens]  
 **for** item **in** FIDS1: *# adjust them in the FIDS1 list too* item[1] += gens\_adjust  
  
 *# unique\_gens = Counter(gens).keys() #eg [0,1,2,3] - not working. giving a dict\_keys object in debugger instead of a list* unique\_gens = [\*Counter(gens)] *# googled it and found this way, using "unpacking" with \** **return** unique\_gens

**def** calculate\_sizings(sib\_gap\_units, sibgroup\_gap\_units, sibgeneration):  
 n = (len(sibgeneration) + 1) \* sibgroup\_gap\_units *# start with no of units between sibling family groups (inc. either side)* **for** sibgroup **in** sibgeneration:  
 n = n + (len(sibgroup) + ((len(sibgroup) - 1) \* sib\_gap\_units)) *# add a unit per sibling family and gap units between them* box\_w = round(1 / n, 2) *# calculate ACTUAL screen width for a family-box unit (percentage value where 1 = 100%). Round to 2dp* sib\_gap = round(box\_w \* sib\_gap\_units, 2) *# ACTUAL* sibgroup\_gap = round(box\_w \* sibgroup\_gap\_units, 2) *# ACTUAL* **return** box\_w, sib\_gap, sibgroup\_gap

This function calculates on-screen size of family box widgets and the gaps in between them. The spacings/sizings are different for each generation, based upon how many families are in the generation. The more families, the smaller the sizings, to allow them to fit on screen. The returned values are not in pixels but in relative proportion to the window (0.00 to 1.00) so that widgets automatically resize if the window is resized.

**def** buildgeneration (g, sibgeneration, families, box\_w, sibgroup\_gap, sib\_gap, y\_axis\_factor, info\_disp\_units):  
 poscursor = sibgroup\_gap  
 gen\_string = **''** y\_pos = y\_axis\_factor \* (g + info\_disp\_units)  
 **for** sibgp **in** sibgeneration:  
 **for** sib **in** sibgp:  
 fam = families.get(sib) *# a Family object* mum = fam.mum *# a dictionary* dad = fam.dad *# a dictionary* kids = fam.kids *# a list of dictionaries* fam.size\_hint = [box\_w, y\_axis\_factor \* 0.5]  
 fam.pos\_hint = [poscursor, y\_pos]  
 size\_hint = str(box\_w) + **", "** + str(y\_axis\_factor \* 0.5)  
 pos\_hint = **"{'x':"** + str(poscursor) + **", 'y':"** + str(y\_pos) + **"}"** childrenstring = **''** famboxstring = **'''  
 BoxLayout:  
 id: '''**+ sib +**'''  
 size\_hint: '''** + size\_hint + **'''**

The buildgeneration function is passed the generation number **g**, the list of families in that generation (**sibgeneration**), the dictionary of all families (**families**) and various sizing variables to dictate on-screen display parameters.

Using this info, it returns a string (**genstring**) that comprises Kivy markup language, describing how the on-screen widgets will be laid out for this generation of families.

**pos\_hint: '''** + pos\_hint + **'''  
 orientation: 'vertical\''''** parentstring = **'''  
 BoxLayout:  
 Button:  
 id: '''**+ mum[**'pid'**] +**'''  
 text: \''''**+ mum[**'name'**] +**''''  
 halign: 'center'  
 text\_size: self.width, None  
 font\_size: self.height \* .24  
 on\_press: root.display\_info(\''''** + mum[**'pid'**] + **'''')  
 Button:  
 id: '''**+ dad[**'pid'**] +**'''  
 text: \''''**+ dad[**'name'**] +**''''  
 halign: 'center'  
 text\_size: self.width, None  
 font\_size: self.height \* .24  
 on\_press: root.display\_info(\''''** + dad[**'pid'**] + **'''')  
 BoxLayout:'''  
  
 for** k **in** kids:  
 childrenstring += **'''  
 Button:  
 id: '''**+ k[**'pid'**] +**'''  
 text: \''''**+ k[**'firstname'**] + **'\\n'** + k[**'surname'**] + **''''  
 halign: 'center'  
 text\_size: self.width, None  
 font\_size: self.height \* .21  
 on\_press: root.display\_info(\''''** + k[**'pid'**] + **'\')'** *#font\_size: (self.height \*.25) if (len(self.text) < 12) else (self.height \*.15)* gen\_string += famboxstring + parentstring + childrenstring  
 poscursor += (box\_w + sib\_gap)  
 poscursor -= sib\_gap *#deduct the sib\_gap following final sib in a group, because we add a sibgroup\_gap instead (slightly larger)* poscursor += sibgroup\_gap  
 **return** gen\_string

**def** gather\_generation(g, FIDS1, FIDS3, families, sibgeneration, fid\_done, upfams):  
 **if** g == 0 :  
 *# process generation 0 this way...* **for** fid1 **in** FIDS1: *# loop through FIDS1* **if** fid1[1] == g: *# only act where Generation = 0* id = fid1[0] *# use the family id to look through FIDS3* **for** fid3\_a **in** FIDS3: *# loop through FIDS3* downfam = fid3\_a[3]  
  
 **if** (downfam == id):  
 upfam = fid3\_a[2] *# gets the upfam id* upfam\_obj = families.get(upfam) *# gets the Family object of upfam* sibs = [] *# initialise list to hold sibs ids (all downfams of upfam)* **for** fid3\_b **in** FIDS3: *# loop through FIDS3 again to collect the sibs ids* **if** fid3\_b[2] == upfam:  
 downfam = fid3\_b[3]  
 downfam\_obj = families.get(downfam) *# a Family object* linker = fid3\_b[1]  
 **if** downfam\_obj.mum.get(**'pid'**)==linker:  
 link\_side=**'mum'  
 else**:  
 link\_side=**'dad'** upfam\_obj.downlinks.append([downfam, link\_side])  
  
 **if** downfam **not in** fid\_done:  
 sibs.append(downfam)  
 fid\_done.append(downfam) *#add this FID to done list so we don't display this family more than once* **if** len(sibs) > 0 :  
 sibgeneration.append(sibs) *# add the sibs group (if not empty) to list of sibs groups for this generation (gen 0)* upfams.append(upfam) *# store the upfam for this sibs group in the upfams List* **else**:  
 *# process all other generations this way...* sibgeneration = [] *# reset this list* **for** fid1 **in** FIDS1: *# loop through FIDS1* **if** (fid1[1] == g) **and** (fid1[0] **not in** upfams): *# Add any FID not already in upfams List (because it had no child fams)* upfams.append(fid1[0])  
 thisgenfams = upfams  
 upfams = [] *# reset this list now to reuse for adding next generation's upfams* **for** this\_id **in** thisgenfams:

The job of this function is to return a list of families (**sibgeneration**) belonging to a particular generation (**g**).It also returns a list of ‘parent’ families (**upfams**), i.e. families who are the direct ancestor families for the ones in this generation. The upfams list is gathered at this point just to optimise on-screen-placement, i.e. more likely to be above the child family, but not possible in all cases. Also, specifically for the bottom-most generation (generation 0) who are the youngest families, it orders them into sibling-family groups, i.e. keeping families that share a parent family next to each other. This is again just to allow slightly better on-screen placement of the parent family in the generation above.

For every **upfam** we find, we append entries to the list called **downlinks** which is an attribute in the Family object of this upfam family. This stores the family ids of the descendent families which upfam links to, as well as if the child in upfam linking to each descendent family becomes the mother or the father in the descendant family.

This later dictates how the line connecting the two families is drawn. The line always originates at the bottom-centre of the upfam and terminates above either the father or the mother in the downfam, depending on who the linker person is.

**for** fid3 **in** FIDS3: *# loop through FIDS3 to see if there is an entry linking it to an upfam* **if** fid3[3] == this\_id:  
 upfam = fid3[2] *# gets the upfam id* upfam\_obj = families.get(upfam) *# gets the Family object of upfam* downfam\_obj = families.get(this\_id) *# a Family object* linker = fid3[1]  
 **if** downfam\_obj.mum.get(**'pid'**) == linker:  
 link\_side = **'mum'  
 else**:  
 link\_side = **'dad'** upfam\_obj.downlinks.append([this\_id, link\_side])  
 upfams.append(upfam) *# store the upfam for this fam id in the upfams List* **if** this\_id **not in** fid\_done: *# FIDS3 entry found for this\_id* lone\_sib = []  
 lone\_sib.append(this\_id)  
 sibgeneration.append(  
 lone\_sib) *# add the sibs group (only 1 fam) to the list of sibs groups for this generation (g)* fid\_done.append(this\_id) *# add this\_id to done list so we don't display it more than once* **return** sibgeneration, fid\_done, upfams

**def** assemble\_kivy\_string(y\_axis\_factor, info\_disp\_units, info\_disp\_height, buildgens, families):  
 rootstring = **'<FamilyTreeCanvas>:'** topboxstring = **'''  
 RelativeLayout:  
 id: top  
 Label:  
 id: info\_disp  
 pos\_hint: {'x':0.01, 'y':0.01}  
 size\_hint: 1, '''** + str(y\_axis\_factor \* info\_disp\_units) + **'''  
 text\_size: root.width, '''** + str(info\_disp\_height) + **'''  
 text:'click on a person to see more information'  
 color: 1,1,1,1 '''** generationstring = **''  
 for** b **in** buildgens:  
 generationstring += b

This does the final assembling of the string that will be passes to the Kivy language parser called **Builder**.

The top level widget (also called a Rule) is called **FamilyTreeCanvas** and this has a matching Class as mentioned above, so that there is a handle between what is displayed and python code.

The final string will be markup language describing the layout of all the widgets on-screen. It has to be constructed very carefully, with exact indentations that are multiples of 4 spaces. Therefore, triple quotes are used to retain the space formatting. There are breaks in the string where python code is used to dynamically insert parameters.

(2 families where one of the parents from each are siblings will be closer together than 2 families where none of the parents are siblings

*# Add lines* linestring = **'''  
 RelativeLayout:  
 id: lines'''  
  
 for** fam **in** families.values():  
 **if** len(fam.downlinks) > 0:  
 up\_pos = fam.pos\_hint  
 up\_size = fam.size\_hint  
 reltop\_x = up\_pos[0] + (up\_size[0] \* 0.5)  
 reltop\_y = up\_pos[1]  
  
 **for** df **in** fam.downlinks:  
 down\_id = df[0]  
 down\_side = df[1]  
 downfam\_obj = families.get(down\_id)  
 down\_pos = downfam\_obj.pos\_hint  
 down\_size = downfam\_obj.size\_hint  
 relpos\_x = (down\_pos[0] + (down\_size[0] \* 0.25)) **if** down\_side == **'mum' else** (  
 down\_pos[0] + (down\_size[0] \* 0.75))  
 relpos\_y = down\_pos[1] + down\_size[1]  
 relpos = **"{'x':"** + str(**"{0:.2f}"**.format(relpos\_x)) + **", 'y':"** + str(**"{0:.2f}"**.format(relpos\_y)) + **"}"** relsize = str(**"{0:.2f}"**.format(reltop\_x - relpos\_x)) + **", "** + str(**"{0:.2f}"**.format(reltop\_y - relpos\_y))  
 line\_col = **'1, 0.75, 0.79, 1' if** down\_side == **'mum' else '.52, 0.8, 0.98, 1'** linestring += **'''  
 BoxLayout:  
 id: line\_'''** + fam.id + **'''  
 pos\_hint: '''** + relpos + **'''  
 size\_hint: '''** + relsize + **'''  
 canvas:  
 Color:  
 rgba: '''** + line\_col + **'''  
 Line:  
 points: self.x, self.y, self.right, self.top  
 width: 2'''** fullstring = rootstring + linestring + topboxstring + generationstring  
 **return** fullstring

*# ============== MAIN SUBROUTINE. CALLED WITH PARAMETERS FIDS1, FIDS2, FIDS3 ===================================================***def** draw\_family\_tree(FIDS1, FIDS2, FIDS3):  
 *# FIDS1 example : ['FID11', 0, 'SK1698', 'TK3231', ['DK6984', 'IK8120']] [familyid, generation, motherid, fatherid, [kids ids]]  
 # FIDS2 example : 'FQ2854': ('FQ2854', 'Faizan', 'Qureshi', '2011-11-30', '1908-01-01', 'Cardiff', 'Male', 'SK3438', 'JQ5803')*

*# PersonID: PersonID, Firstnm, Surnmm, dob, dod, pob, gender, mother, father   
 # FIDS3 example : [1, 'NA9171', 'FID1', 'FID8'] [ignore, linking person, upfam id, downfam id]  
  
 # Using the FIDS1 List make a family object for each fam and put them all in the 'families' dictionary* families = families\_dict(FIDS1, FIDS2)  
 *# Get a list of generation numbers (levels) for this family tree* unique\_gens = get\_unique\_gens(FIDS1)  
  
 *# For each generation number (0 to n) make a List of families belonging to it* sibgeneration = [] *# The list for the generation* fid\_done = [] *# keep track of families in the family tree that have been processed.* upfams = [] *# List of parent families in the next generation up* buildgens = [] *# This will hold the Kivy language strings describing each generatiom, later parsed by Kivy Builder function.  
  
 # Set some sizes for on-screen spaces between family boxes and the information display area at screen bottom* sib\_gap\_units = 0.35  
 sibgroup\_gap\_units = 0.7  
 info\_disp\_units = 0.5 *# as a proportion (0 to 1) of height of generation rows. Governs height of area at screen bottom* y\_axis\_units = len(  
 unique\_gens) + info\_disp\_units *# how many units to divide the root (RelativeLayout) height by* y\_axis\_factor = round(1 / y\_axis\_units, 2) *# as a proportion (0 to 1) of the root (RelativeLayout) height* info\_disp\_height = Window.height \* (y\_axis\_factor \* info\_disp\_units) *# ACTUAL  
  
 # Now process each generation and construct the Kivy code string* **for** g **in** unique\_gens:  
 *# 1. Find all fams in generation 'g' (sibgeneration) and also their immediate ancestor fams (upfams)* sibgeneration, fid\_done, upfams = gather\_generation(g, FIDS1, FIDS3, families, sibgeneration,  
 fid\_done, upfams)  
 *# 2. Calculate how to fit them across the screen based on num of fams in the generation* box\_w, sib\_gap, sibgroup\_gap = calculate\_sizings(sib\_gap\_units, sibgroup\_gap\_units, sibgeneration)  
 *# 3. Construct a 'kivy code' string for this generation using the above information* buildgen = buildgeneration(g, sibgeneration, families, box\_w, sibgroup\_gap, sib\_gap, y\_axis\_factor,  
 info\_disp\_units)  
 *# 4. Add the string for each generation into a list (buildgens)* buildgens.append(buildgen)  
  
 *# Assemble the full string including connecting lines and information display area* fullstring = assemble\_kivy\_string(y\_axis\_factor, info\_disp\_units, info\_disp\_height, buildgens, families)  
  
 *# PASS TO THE KIVY BUILDER FUNCTION* Builder.load\_string(fullstring)  
  
 FamilyTreeCanvas.FIDS2 = FIDS2  
  
*# =================================== END ===================================================*

Call **Builder** to create on-screen widgets by parsing **fullstring**.

This is the main subroutine that is called to create the Family Tree display. It sets up certain variables, then calls various subroutines to build up the Kivy code string.

Had to store the FIDS2 dictionary into an attribute on the root widget class, so that screen button presses could get access to person info.

*# ================== KIVY APP IS KICKED OFF BY CREATING A STARTER CLASS WHICH INHERITS FROM KIVY.APP ===================***class** StartFamilyTreeApp(App):  
 **def** build(self):  
 draw\_family\_tree(self.fids1, self.fids2, self.fids3)  
 **return** FamilyTreeCanvas()  
  
*# =================================== END ===================================================***if** \_\_name\_\_ == **"\_\_main\_\_"**:  
 **pass** *# do nothing if running this module directly  
 #StartFamilyTreeApp().run()*

Not used in my implementation, but useful for testing and is needed for security if the Kivy app was a standalone app, as it would then prevent the app being called and run from an external program. In such a case I would remove ‘pass’ and uncomment the last line.

All Kivy apps need a class definition that inherits from kivy.app (**App**).  
App defines a build function, which is overwritten by our instance and must return the top level widget class (**FamilyTreeCanvas**).

**Database Changes**

Sqlite3: allows tables and database to be manipulated

**import** sqlite3  
  
**def** check\_table\_exists(tablename):  
 connection = sqlite3.connect(**'Family\_1.db'**)  
 cursor = connection.cursor()  
 cursor.execute(**"SELECT name FROM sqlite\_master WHERE type='table';"**)  
 list\_of\_tables=(cursor.fetchall())  
 **if** (tablename,) **in** list\_of\_tables:  
 exists=**True  
 return** exists  
 **else**:  
 exists=**False  
 return** exists, list\_of\_tables  
  
**def** delete\_table(tablename):  
 connection= sqlite3.connect(**'Family\_1.db'**,)  
 cursor=connection.cursor()  
 sql= (**"""DROP TABLE %s """** % tablename)  
 cursor.execute(sql)  
 connection.commit()  
 connection.close()  
   
   
  
   
**def** create\_tblSingleFamily():  
 *#first\_family\_id= FamilyIDQueue.Dequeue(FamilyID)* connection = sqlite3.connect(**"Family\_1.db"**)  
 cursor = connection.cursor()  
 sql\_command = **"""  
 CREATE TABLE tblSingleFamily  
 (  
 FamilyID TEXT,  
 Mother TEXT,  
 Father TEXT,  
 primary key (FamilyID)  
 )"""** cursor.execute(sql\_command)  
 connection.commit()  
 connection.close()

Creates tblSingleFamily

Deletes table called ‘tablename’ from the database.

Checks if ‘tablename’ is a table in the list of tables

Gets a list of all tables in the database

**def** create\_tblChildren():  
 *#first\_family\_id= FamilyIDQueue.Dequeue(FamilyID)* connection = sqlite3.connect(**"Family\_1.db"**)  
 cursor = connection.cursor()  
 sql\_command = **"""  
 CREATE TABLE tblChildren  
 (  
 FamilyID TEXT,  
 PersonID TEXT,  
 primary key (FamilyID, PersonID)  
 )"""** cursor.execute(sql\_command)  
 print(**"tblChildren"**)  
 connection.commit()  
 connection.close()  
  
**def** add\_children(first, second, first\_family\_id):  
 connection=sqlite3.connect(**'Family\_1.db'**)  
 cursor=connection.cursor()  
 children\_of\_people= first.intersection(second)  
 children\_rec=[first\_family\_id]  
 **for** z **in** range(0,len(children\_of\_people)):  
 children\_rec.append(children\_of\_people.pop())  
 cursor.execute(**"INSERT INTO tblChildren VALUES (?,?)"**, children\_rec)  
 connection.commit()  
 children\_rec=[first\_family\_id]  
  
 connection.close()

Adds each child of ‘first’ and ‘second’ to tblChildren, where the FamilyID is ‘first\_family\_id’.

Creates the table tblChildren

**Get\_All\_Info**

**import** sqlite3  
  
**def** get\_everything():  
 everything\_dict={}  
 conn=sqlite3.connect(**"Family\_1.db"**)  
 cursor=conn.cursor()  
 **for** row **in** cursor.execute (**'SELECT \* FROM tblFamily'**):  
 all=cursor.fetchall()  
 **for** y **in** range(0,len(all)):  
 everything\_dict[all[y][0]]=all[y]  
 **return** everything\_dict  
  
  
  
**def** get\_everything\_connected():  
 all=[]  
 everything\_connected\_dict = {}  
 conn = sqlite3.connect(**"Family\_1.db"**)  
 cursor = conn.cursor()  
 **for** row **in** cursor.execute(**'SELECT \* FROM tblFamily'**):  
 all.append(row)  
 **for** y **in** range(0, len(all)):  
 all[y]  
 everything\_connected\_dict[all[y][0]] = all[y]  
 **return** everything\_connected\_dict

Gets all information about everybody from the database, so that the tree can be drawn without having to make a lot of sql searches

**Tkinter Invalid Entry**

Imports:  
tkinter: Allows widgets to be created  
datetime: Allows dates to be manipulated

**from** tkinter **import** \*  
**from** datetime **import** datetime, date  
  
**def** pop\_up\_menu():  
 menu = Tk()  
 menu.wm\_title(**"Invalid Entry"**)  
 label =Label(menu, text=**"Invalid Entry"**, font=**"none 12 bold"**).grid(row=1,column=0)  
 confirm =Button(menu, text=**"Ok"**, command=**lambda**:[menu\_close(menu)]).grid(row=2,column=0,sticky=W)  
 menu.mainloop()  
  
  
**def** check\_dob\_and\_dod(DOB,DOD,AllInfo):  
 **if** DOB==**"" and** DOD==**""**:  
 **return** AllInfo  
 **else**:  
 **if** DOB!=**""**:  
 **if** DOB > date.today():  
 AllInfo=**False  
 if** DOD!=**""**:  
 **if** DOD > date.today():  
 AllInfo=**False  
 if** DOB!=**"" and** DOD!=**""**:  
 **if** DOB>DOD:  
 AllInfo=**False  
  
 return** AllInfo  
  
**def** menu\_close(menu):  
 menu.destroy()

If DOB is greater than DOD, then All Info is also false.

Checks if DOB and DOD is greater than today’s date. If it is, then All Info is false.

If DOB or DOD are both empty, none of the checks below need to take place (NOTE that AllInfo being false means that there has been an invalid entry)

Creates window that pops up when an invalid entry is made.

**Change Distance Dictionary**

values = A list of all generation numbers  
highest\_value = last item in the list once the list has been sorted in ascending order.   
  
the highest generation number will be the most recent generation. To make this the lowest generation number of 0 and all the other generations be positive integers, we take each generation number and subtract it from the highest value, and assign these to each familyID in the distance dictionary.

**def** change\_distance\_dictionary(distance\_dictionary):  
 if distance\_dictionary==None:  
 **return** distance\_dictionary

else:

values=[]  
 **for** x **in** range(0,len(distance\_dictionary)):  
 values.append(distance\_dictionary[**"FID"**+str(x+1)])  
 sorted\_values=sorted(values)  
 highest\_value=int(sorted\_values.pop(len(sorted\_values)-1))  
 **for** y **in** range(0,len(distance\_dictionary)):  
 item=int(distance\_dictionary[**"FID"**+str(y+1)])  
 new\_value = highest\_value - item  
 distance\_dictionary[**"FID"**+str(y+1)]=new\_value  
 **return** distance\_dictionary

If the graph is completely unconnected, then distance\_dictionary will be None.

**Add Mother**

Imports:

Tkinter: Allows the creation of widgets for a GUI  
sqlite3: Allows databases and tables within databases to be manipulated

**from** tkinter **import** \*  
**import** sqlite3  
  
  
**def** close\_window1(window1):  
 window1.destroy()  
  
**def** extract\_id(person):  
 person=person.strip()  
 **if "\_P" in** person:  
 new\_parent\_id=person[2:10]  
 **elif "Mother" in** person:  
 **return** person  
 **else**:  
 new\_parent\_id=person[2:8]  
 **return** new\_parent\_id  
  
**def** get\_mother\_id(new\_mother):  
 count=0  
 x=0  
 **while** count!=2:  
 letter=new\_mother[x]  
 **if** letter==**"-"**:  
 count+=1  
 x=x+1  
 MotherID=new\_mother[2:x+2]  
 **return** MotherID  
  
  
**def** submit\_mother(complete\_no\_mother):  
 conn = sqlite3.connect(**"Family\_1.db"**)  
 cursor = conn.cursor()  
 print(complete\_no\_mother)  
 **for** x **in** range(0, len(complete\_no\_mother)):  
 new\_mother = complete\_no\_mother[x][2].get()  
 **if** new\_mother==**"-"**:  
 MotherID=**""  
 elif "Father" in** new\_mother **or "Mother" in** new\_mother:  
 **if** new\_mother[2]==**""**:  
 new\_mother=new\_mother[0:1] + new\_mother[3:len(new\_mother)]  
 MotherID=get\_mother\_id(new\_mother)  
  
  
 **else**:  
 MotherID = extract\_id(new\_mother)  
  
 current\_person\_id = complete\_no\_mother[x][0]  
 MotherID= **"'"** + MotherID + **"'"** current\_person\_id= **"'"** + current\_person\_id + **"'"** cursor.execute(**"UPDATE tblFamily SET MotherID ="** + MotherID + **"WHERE PersonID="** + current\_person\_id)  
 conn.commit()  
 conn.close()  
  
  
  
**def** get\_no\_mother():  
 complete\_no\_mother=[]  
 no\_mother\_list=[]  
 conn=sqlite3.connect(**"Family\_1.db"**)  
 cursor=conn.cursor()  
 **for** row **in** cursor.execute(**"SELECT PersonID,Firstname,Surname FROM tblFamily WHERE (PersonID NOT LIKE '%Mother%' AND PersonID NOT LIKE '%Father%' AND PersonID NOT LIKE '%\_P%') AND MotherID = '""' "**):  
 no\_mother\_list.append(row)  
 **if** len(no\_mother\_list)==0:  
 **return** complete\_no\_mother  
 **else**:  
 window1 = Tk()  
 window1.configure(background=**"Light Blue"**)  
 **for** x **in** range(0,len(no\_mother\_list)):  
 selected=create\_no\_mother\_dropdown(window1,x)  
 no\_mother=[no\_mother\_list[x][0], no\_mother\_list[x][1] + **" "** + no\_mother\_list[x][2], selected]  
 complete\_no\_mother.append(no\_mother)  
 Label(window1, text=**"Choose a mother from the database for:"**, bg=**"Light blue"**, fg=**"black"**, font=**"none 18 bold"**).grid(row=x, column=0,sticky=W)  
 Label(window1, text=no\_mother\_list[x][1] + **" "** + no\_mother\_list[x][2], bg=**"Light blue"**,fg=**"black"**, font=**"none 18 bold"**).grid(row=x, column=1,sticky=W)  
 Button(window1, text=**"Submit"**, width=6, command=**lambda**:[submit\_mother(complete\_no\_mother),close\_window1(window1)]) .grid(row=x+1, column=2, sticky=W)  
 window1.mainloop()  
 **return** complete\_no\_mother  
  
  
  
**def** create\_no\_mother\_dropdown(window1,x):  
 conn=sqlite3.connect(**"Family\_1.db"**)  
 cursor=conn.cursor()  
 choices=[]

Adds mother to the database

Gets the PersonID from the drop-down menu choice

Gets a list of people with no mothers entered into the database.

Gets a separate drop-down menu for each person who does not have a moother entered.

selected = StringVar(window1)  
 selected.set(**'-'**)  
 **for** row **in** cursor.execute(**"SELECT \* FROM tblFamily WHERE Gender= 'Female' AND Firstname <> '""' "**):  
 choices.append(row)  
 **for** row2 **in** cursor.execute(**"SELECT MotherID FROM tblFamily WHERE MotherID LIKE '%Mother%' "**):  
 choices.append(row2)  
 pop\_up\_menu = OptionMenu(window1, selected, \*choices)  
 pop\_up\_menu.grid(row=x, column=2, sticky=W)  
 conn.commit()  
 conn.close()  
 **return** selected  
  
**def** add\_mother\_main():  
 complete\_no\_mother=get\_no\_mother()

**Add\_Father**

**from** tkinter **import** \*  
**import** sqlite3  
  
  
**def** close\_window1(window1):  
 window1.destroy()  
  
**def** extract\_id(person):  
 person=person.strip()  
 **if "\_P" in** person:  
 new\_parent\_id=person[2:10]  
 **else**:  
 new\_parent\_id=person[2:8]  
 **return** new\_parent\_id  
  
**def** get\_father\_id(new\_father):  
 print(new\_father[0])  
 count=0  
 x=0  
 **while** count!=2:  
 letter=new\_father[x]  
 **if** letter==**"-"**:  
 count+=1  
 x=x+1  
 FatherID=new\_father[2:x+2]  
 **return** FatherID  
  
  
  
  
**def** submit\_father(complete\_no\_father):  
 conn = sqlite3.connect(**"Family\_1.db"**)  
 cursor = conn.cursor()  
 print(complete\_no\_father)  
 **for** x **in** range(0, len(complete\_no\_father)):  
 new\_father = complete\_no\_father[x][2].get()  
 **if** new\_father==**"-"**:  
 FatherID=**""  
 elif "Father" in** new\_father **or "Mother" in** new\_father:  
 **if** new\_father[2]==**""**:  
 new\_father=new\_father[0:1] + new\_father[3:len(new\_father)]  
 FatherID=get\_father\_id(new\_father)  
  
  
  
 **else**:  
 FatherID = extract\_id(new\_father)  
  
 current\_person\_id = complete\_no\_father[x][0]  
 FatherID= **"'"** + FatherID + **"'"** current\_person\_id= **"'"** + current\_person\_id + **"'"** cursor.execute(**"UPDATE tblFamily SET FatherID ="** + FatherID + **"WHERE PersonID="** + current\_person\_id)  
 conn.commit()  
 conn.close()  
  
  
  
**def** get\_no\_father():  
 no\_father\_list=[]  
 complete\_no\_father=[]  
 conn=sqlite3.connect(**"Family\_1.db"**)  
 cursor=conn.cursor()  
 **for** row **in** cursor.execute(**"SELECT PersonID,Firstname,Surname FROM tblFamily WHERE (PersonID NOT LIKE '%Father%' AND PersonID NOT LIKE '%Mother%' AND PersonID NOT LIKE '%\_P%') AND FatherID= '""' "**):  
 no\_father\_list.append(row)  
 **if** len(no\_father\_list)==0:  
 **return** complete\_no\_father

Almost Identical to Add\_Mother

**else**:  
 window1 = Tk()  
 window1.configure(background=**"Light Blue"**)  
 **for** x **in** range(0,len(no\_father\_list)):

selected=create\_no\_father\_dropdown(window1,x)  
 no\_father=[no\_father\_list[x][0], no\_father\_list[x][1] + **" "** + no\_father\_list[x][2], selected]  
 complete\_no\_father.append(no\_father)  
 Label(window1, text=**"Choose a father from the database for:"**, bg=**"Light blue"**, fg=**"black"**, font=**"none 18 bold"**).grid(row=x, column=0,sticky=W)  
 Label(window1, text=no\_father\_list[x][1] + **" "** + no\_father\_list[x][2], bg=**"Light blue"**,fg=**"black"**, font=**"none 18 bold"**).grid(row=x, column=1,sticky=W)  
 Button(window1, text=**"Submit"**, width=6, command=**lambda**:[submit\_father(complete\_no\_father),close\_window1(window1)]) .grid(row=x+1, column=2, sticky=W)  
 window1.mainloop()  
 **return** complete\_no\_father  
  
  
  
**def** create\_no\_father\_dropdown(window1,x):  
 conn=sqlite3.connect(**"Family\_1.db"**)  
 cursor=conn.cursor()  
 choices=[]  
 selected = StringVar(window1)  
 selected.set(**'-'**)  
 **for** row **in** cursor.execute(**"""SELECT \* FROM tblFamily WHERE Gender="Male" """**):  
 choices.append(row)  
 pop\_up\_menu = OptionMenu(window1, selected, \*choices)  
 pop\_up\_menu.grid(row=x, column=2, sticky=W)  
 conn.commit()  
 conn.close()  
 **return** selected  
  
**def** add\_father\_main():  
 complete\_no\_father=get\_no\_father()

**Some small changes to my above code were made after testing. See my failed tests to see these small change**

**TESTING**

|  |  |
| --- | --- |
| Test No | T1 |
| Type of test | Black-box testing |
| Objective No | 1,2,3 |
| Purpose of test | To make sure that my program does not throw an error when the program is first being run and the “add new person” option is not chosen. |
| Description of test | I will create a new database, and see if the program works if I do not initially choose the ‘enter new person’ option. |
| Test data | First name = Dariyan Surname=Khan DOB = 03-03-2001 DOD= POB = London Mother= - Father= - |
| Expected result | Program does not crash, and first person I enter is added to the database. |
| Actual Result | |  | | --- | |  | |  | |  | |
| Description | As you can see, I deleted my previous data and when I clicked on ‘Edit an Entry’ when there was no-one in the database, I was taken to the person where you can add a new person. After I entered my information and pressed submit, you can see that this time I was allowed to edit a record, and when I checked the drop-down menu for who I could edit, only I was in the menu. |

|  |  |
| --- | --- |
| Test No | T2 |
| Type of test | Black-box testing |
| Objective No | 1,2 |
| Purpose of test | To see whether the ‘edit a person’, option works. |
| Description of test | I should be able to change my first-name, without an error. |
| Test data | Try changing ‘Dariyan’ to ‘Bob’ |
| Expected result | When I view my record, it should be exactly the same, apart from my first name |
| Actual Result | |  | | --- | |  | |  | |  | |  | |
| Description | My program successfully changed my first name to ‘Bob’, without changing any other properties (escpeically my PersonID) |

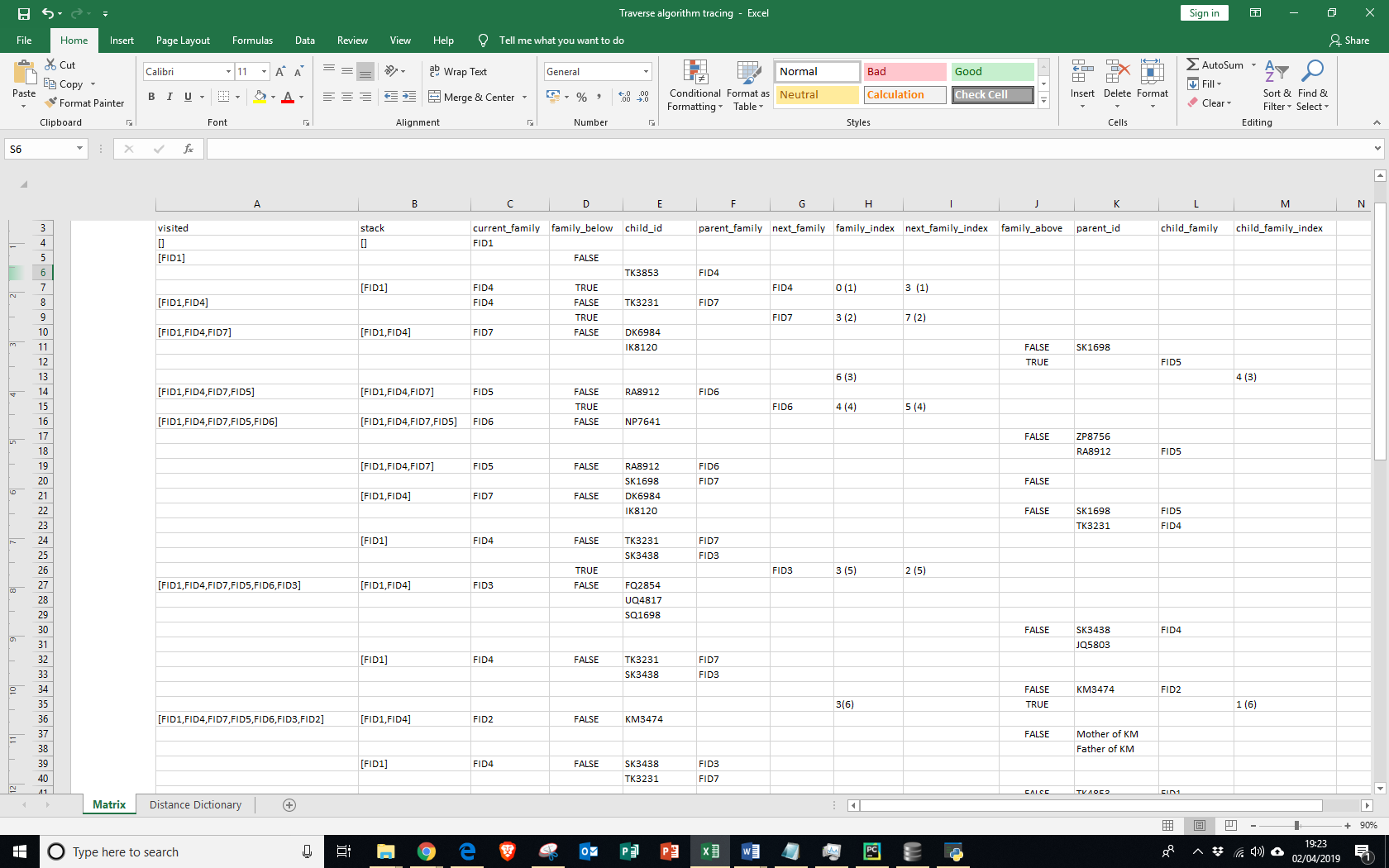
|  |  |
| --- | --- |
| Test No | T3 |
| Type of test | Black box testing |
| Objective No | 1,2 |
| Purpose of test | To see if I can delete people using my GUI |
| Description of test | I will try to delete ‘Bob Khan’ from the database |
| Test data | ‘Bob Khan’ |
| Expected result | The drop-down menu afterwards should be empty |
| Actual Result | |  | | --- | |  | |  | |  | |
| Description | As you can see, if Bob was not deleted, then he would appear as a possible father in the ‘Select this person’s father’ drop down menu. But because he has not, Bob must have been deleted. |

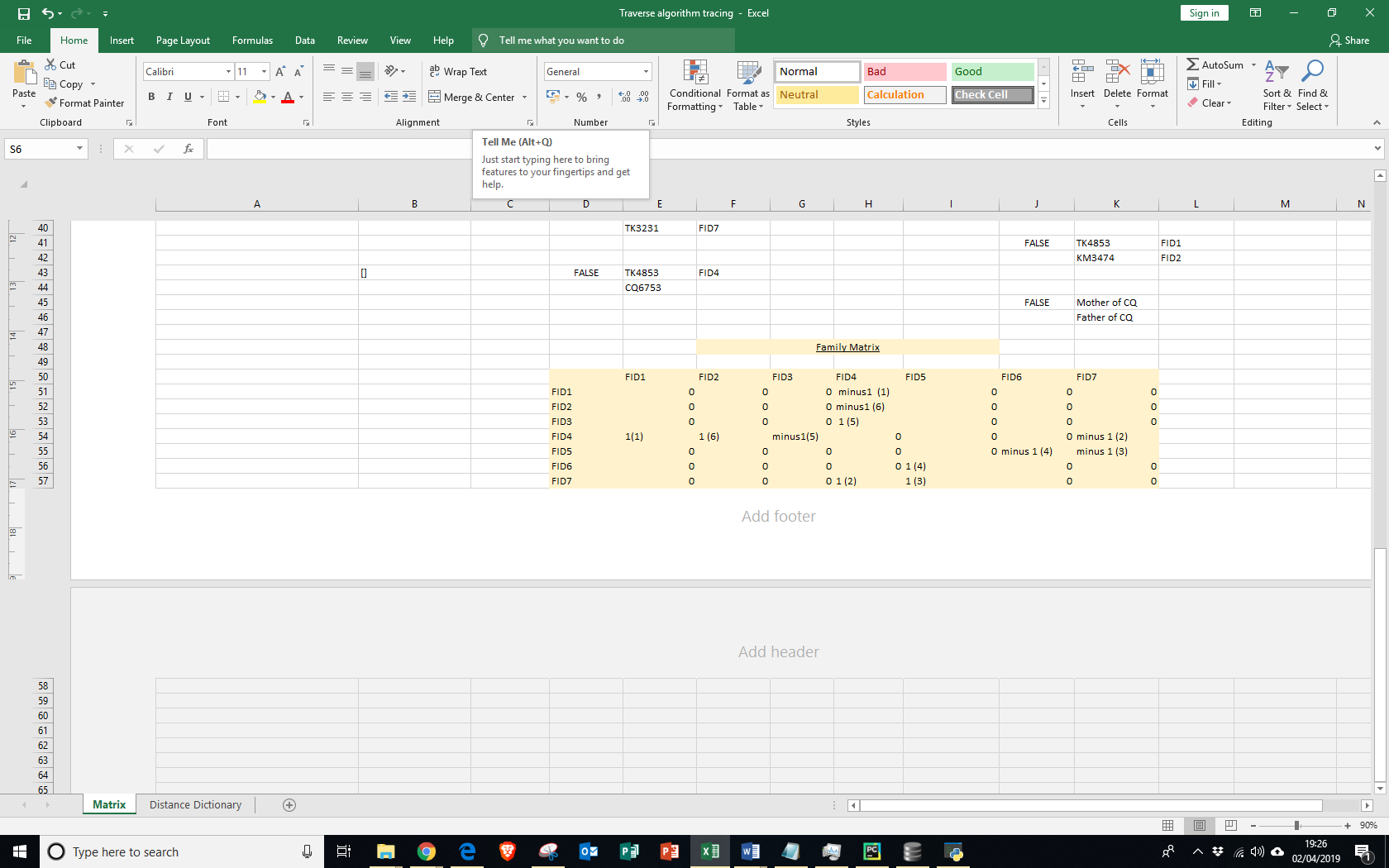
|  |  |
| --- | --- |
| Test No | T4 |
| Type of test | Black-box testing |
| Objective No | 15 |
| Purpose of test | To see if the ‘Invalid’ entry box is displayed when certain data is entered. |
| Description of test | I will enter a selection of typical, erroneous and extreme data for the form, and see how the program responds. (F=first name, S=surname, DOB = Date of birth and DOD= date of death. I tested this on 18/03/2019. I have also temporarily commented out the line of code that clears the data after pressing submit, so that you can see what I typed into the form before pressing submit. |
| Test data | 1. F=Alpha, S=Beta, DOB=01/02/2003, DOD=03/04/2011 2. F= (empty), S=Gamma, DOB= 09/07/1972, DOD=07/08/2013 3. F=Leo, S=Khan, DOB=17/03/2019 DOD=(empty) 4. F=Jim, S=Owens, DOB=18/03/2019, DOD=18/03/2019 5. F=Bruce, S=Lee, DOB=hi/every/body   DOD=goodbye/every/body   1. F=John, S=Smith, DOB=21/04/2082   DOD= (empty) |
| Expected result | 1. Is normal data, so should return a pop-up window. 2. Is erroneous data (no Surname) 3. Is boundary data (empty DOD) 4. Is Boundary data (DOB and DOD are the same date, and are the current date. 5. Is erroneous data (DOB and DOD contain characters other than numbers) 6. Is erroneous data (DOB is greater than current date). |
| Actual Result | |  |  | | --- | --- | | a) |  | | b) |  | | c) |  | | d) |  | | e) | SEE FAILED TESTS | | f) |  | |
| Description |  |

|  |  |
| --- | --- |
| Test No | T5 |
| Type of test | Module Testing |
| Objective No | 4 |
| Purpose of test | To check whether the program correctly adds a mother or father or both for a person who chose Unknown for either or both of their parents, and to check whether someone else could then choose those 2 parents as their parents. |
| Description of test | The program should create a Mother and Father ID for the first person I enter, and then I should be able to choose those parents for another person. |
| Test data | I will first enter a person called ‘Tom Wood’, and set his parents to unknown. I will then create another person called ‘Ellie Wood’, who is Tom’s sister. I will select her parents to be the same as Tom’s, and see if the program stores it correctly. |
| Expected result | Tom’s parents should be stored as ‘Mother of Tom Wood born on 12/01/2001’ and ‘father of Tom Wood born on 12/01/2001’  Ellie’s parents, after being entered into the database, should be stored as ‘Mother of Tom Wood born on 12/01/2001’ and ‘father of Tom Wood born on 12/01/2001’. |
| Actual Result | |  | | --- | |  | |  | |  | |  | | SEE FAILED TEST NUMBER 4 | |
| Description | Note that people who have ‘Father’, ‘Mother’ or ‘\_P’ as part of their PersonID are no longer displayed. Below is code to display the drop-down menu, and what it looks like now.   |  | | --- | | **def** choose\_person(message,option\_num):  **with** sqlite3.connect(**"Family\_1.db"**) **as** conn:  cursor = conn.cursor()  choices\_edit=[]  **for** row **in** cursor.execute(**"""SELECT \* FROM tblFamily WHERE PersonID NOT LIKE '%Mother%'  AND PersonID NOT LIKE '%Father%'  AND PersonID NOT LIKE '%\_P%' """**): *### Gets everyone in the database apart from a select few for the choices* choices\_edit.append(row) | |  | |

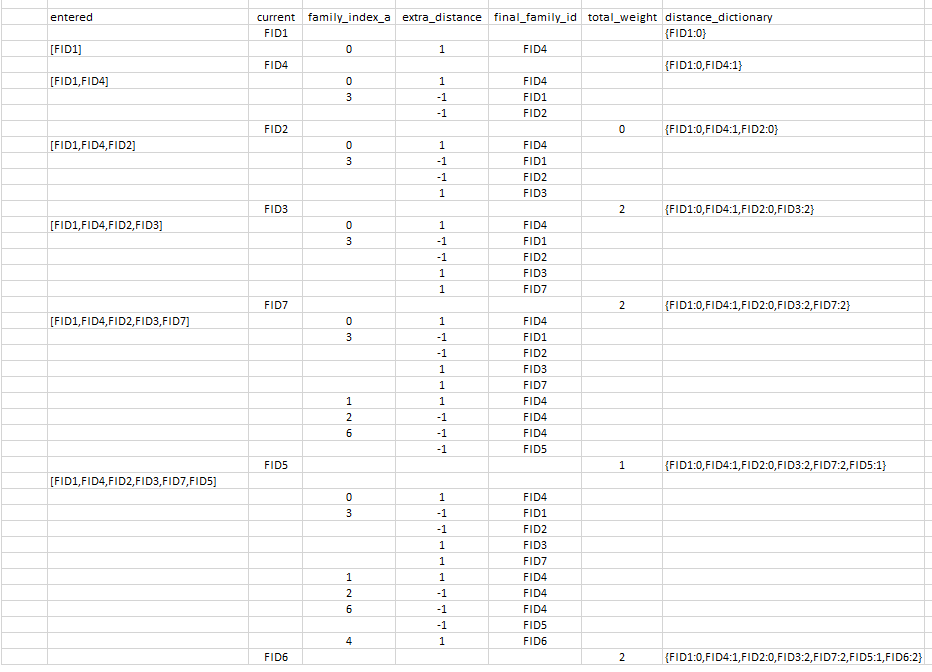
|  |  |
| --- | --- |
| Test No | T6 |
| Type of test | Module Testing |
| Objective No | 5,6,7 |
| Purpose of test | To see whether my program puts people in the database into correct families. |
| Description of test | I will enter some people into the database. I will then, run the “Find relationships” option and choose a trivial relationship. I am not interested in if the relationship outputted is correct but what I am interested in is if the program created tblSingleFamily and tblChildren while finding the relationships, put people into families, and accounted for people with unknown parents. |
| Test data | |  | | --- | |  | |  | |
| Expected result | Parents should be created for Alpha and Beta, and he should be the only child in that family.  A partner should be created for Jim Owens, and that partner, Jim Owens and Bruce Lee should constitute another family.  Tom Wood and Ellie Wood should be another family. Peter Jones, Max Jones and Emma Thomas should also be created as a family. |
| Actual Result | |  |  | | --- | --- | | tblFamily |  | | tblSingleFamily |  | | tblChildren |  | |
| Description | The data above shows that the correct information was added to each table, tblSingleFamily and tblChildren were created, and that my queue works, as FID1-3 only were distributed. |

|  |  |
| --- | --- |
| Test No | T7 |
| Type of test | Module Testing |
| Objective No | 8,9 |
| Purpose of test | To see whether the family matrix produced is correct |
| Description of test | I will let the program create the family matrix and print it out, before tracing the algorithm myself and seeing whether I get the same result. If I do, then I know that my algorithm is correct. (for the purposes of this trace I will only record the 1’s and -1’s and the 0’s. Also, the numbers in brackets in the trace table are there just to improve clarity (they are not appended when the program really runs). |
| Test data | Below are the three tables in my database, and the variable ‘children\_dict’ (in Python, ‘Set()’ stands for an empty set’).   |  |  | | --- | --- | | tblFamily |  | | tblSingleFamily |  | | tblChildren |  | | Children\_dict | |  |  | | --- | --- | | PersonID | Children | | "IK8120" | Set() | | "JQ5803" | FQ2854, SQ5706, UQ4817 | | "FQ2854" | Set() | | "SQ5706" | Set() | | "UQ4817" | Set() | | "CQ6753" | Set() | | "DK6984" | Set() | | "TK4853" | TK3231, SK3438 | | "SK1698" | DK6984, IK8120 | | "KM3474" | SK3438, TK3231 | | "SK3438" | FQ2854, SQ5706, UQ4817 | | "TK3231" | DK6984, IK8120 | | "NA9171" | SK1698, RA8912 | | "MA2798" | SK1698, RA8912 | | "RA8912" | NP7641 | | "ZP8756" | NP7641 | | "NP7641" | Set() | | "Mother of Cheeno Qazi born on 1943-07-03" | CQ6753, TK4853 | | "Mother of Khan Mir born on 1941-02-02" | KM3474 | | "Father of Cheeno Qazi born on 1943-07-03" | CQ6753, TK4853 | | "Father of Khan Mir born on 1941-02-02" | KM3474 | | |
| Expected result | \*\*See page 141 and 142\*\* |
| Actual Result | [[[0, 0, 'FID1'], [0, 0, 'FID1'], [0, 0, 'FID1'], [1, 'TK4853', 'FID1', 'FID4'], [0, 0, 'FID1'], [0, 0, 'FID1'], [0, 0, 'FID1']], [[0, 0, 'FID2'], [0, 0, 'FID2'], [0, 0, 'FID2'], [1, 'KM3474', 'FID2', 'FID4'], [0, 0, 'FID2'], [0, 0, 'FID2'], [0, 0, 'FID2']], [[0, 0, 'FID3'], [0, 0, 'FID3'], [0, 0, 'FID3'], [-1, 'SK3438', 'FID3', 'FID4'], [0, 0, 'FID3'], [0, 0, 'FID3'], [0, 0, 'FID3']], [[-1, 'TK4853', 'FID4', 'FID1'], [-1, 'KM3474', 'FID4', 'FID2'], [1, 'SK3438', 'FID4', 'FID3'], [0, 0, 'FID4'], [0, 0, 'FID4'], [0, 0, 'FID4'], [1, 'TK3231', 'FID4', 'FID7']], [[0, 0, 'FID5'], [0, 0, 'FID5'], [0, 0, 'FID5'], [0, 0, 'FID5'], [0, 0, 'FID5'], [1, 'RA8912', 'FID5', 'FID6'], [1, 'SK1698', 'FID5', 'FID7']], [[0, 0, 'FID6'], [0, 0, 'FID6'], [0, 0, 'FID6'], [0, 0, 'FID6'], [-1, 'RA8912', 'FID6', 'FID5'], [0, 0, 'FID6'], [0, 0, 'FID6']], [[0, 0, 'FID7'], [0, 0, 'FID7'], [0, 0, 'FID7'], [-1, 'TK3231', 'FID7', 'FID4'], [-1, 'SK1698', 'FID7', 'FID5'], [0, 0, 'FID7'], [0, 0, 'FID7']]]   |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | |  | FID1 | FID2 | FID3 | FID4 | FID5 | FID6 | FID7 | | FID1 | 0 | 0 | 0 | -1 | 0 | 0 | 0 | | FID2 | 0 | 0 | 0 | -1 | 0 | 0 | 0 | | FID3 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | | FID4 | 1 | 1 | -1 | 0 | 0 | 0 | -1 | | FID5 | 0 | 0 | 0 | 0 | 0 | -1 | -1 | | FID6 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | | FID7 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | |
| Description |  |

**Family Matrix Algorithm Trace Table**



|  |  |
| --- | --- |
| Test No | T8 |
| Type of test | Module Testing |
| Objective no | Extension Objective 2 |
| Purpose of test | To see whether the correct distances relative to FID1 are created. |
| Description of test | I will use a few families, and get the program to print distance dictionary. I will then trace the program using a trace table and see whether the output produced is correct. In my tracing, I will not write down values where extra distance is zero, to make the trace table shorter and more readable. |
| Test data | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | |  | FID1 | FID2 | FID3 | FID4 | FID5 | FID6 | FID7 | | FID1 | 0 | 0 | 0 | -1 | 0 | 0 | 0 | | FID2 | 0 | 0 | 0 | -1 | 0 | 0 | 0 | | FID3 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | | FID4 | 1 | 1 | -1 | 0 | 0 | 0 | -1 | | FID5 | 0 | 0 | 0 | 0 | 0 | -1 | -1 | | FID6 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | | FID7 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |   I will use the same data as in test 7 |
| Expected result | \*\*See Page 147\*\* |
| Actual result | {'FID1': 0, 'FID4': 1, 'FID2': 0, 'FID3': 2, 'FID7': 2, 'FID5': 1, 'FID6': 2} |
| Description | My trace is the same as the expected result so my algorithm most likely works. |

**Distance Dictionary Trace**

|  |  |
| --- | --- |
| Test No | T9 |
| Type of test | Blackbox |
| Objective no | Extension Objective 2 |
| Purpose of test | To see whether the statistics produced are correct. |
| Description of test | I will calculate the statistics on my calculator, and see whether the statistics that my calculator generators are the same as the ones that my program generates. |
| Test data | List of all ages=[18,17,48,65,2,10,16,43,22,18,44] = in ascending order, [2,10,16,17,18,18,22,43,44,48,65] |
| Expected result | mean= (sum of ages)/11 = 27.54545454… mode= 18 (most common value) median = 6th item in ascending order list = 18  variance=340.79338842975, standard deviation = 18.46 (square root of variance), max age=65, min age = 2 |
| Actual result |  |
| Description | As you can see, the expected statistics match up with the actual statistics calculated. |

|  |  |
| --- | --- |
| Test No | T10 |
| Type of test | Module Testing |
| Objective no | Extension Objective 1 |
| Purpose of test | To see whether my program produces the correct relationship between 2 entered people. |
| Description of test | I will calculate the statistics on my calculator, and see whether the statistics that my calculator generators are the same as the ones that my program generates. |
| Test data | a) Boundary/Erroneous data - choose the same person in both drop-down menus. – Dariyan Khan, Dariyan Khan  b) Normal data - choose two people who are siblings (so are children of the same family) – Dariyan Khan, Inayah Khan  c) Normal data - choose 2 people where 1 person is the parent of the other. Dariyan Khan, Sameena Khan  d) Normal data - choose 2 people who are very distantly related (not even blood related), Faizan Qureshi and Naseem Akhtar  e) Boundary data – choose 2 people that have no connection between them – Tomos Boyles (not in the database yet) and Dariyan Khan |
| Expected result | 1. You chose the same person 2. The 2 people you chose are siblings 3. Dariyan Khan is the son of Sameena Khan 4. Faizan Qureshi’s uncle’s wife’s mother is Naseem Akhtar 5. There is no connection between the 2 entered people. |
| Actual result | |  | | --- | |  | |  | |  | |  | |  | |
| Description | As you can see, my program returns all of the correct results |

|  |  |
| --- | --- |
| Test No | T11 |
| Type of test | Module Testing |
| Objective no | 11,12,13 |
| Purpose of test | To see whether my program draws a tree, with lines joining people, and be able to display the information of a person when you click on them. |
| Description of test | I will use a database of people, and see whether the tree drawn is correct. |
| Test data | |  | | --- | |  | |  | |
| Expected result | A family tree with all of the correct families generated, and correct families connected to each other. |
| Actual result |  |
| Description | The tree displayed is 100% accurate to the data that I inputted. |

|  |  |
| --- | --- |
| Test number | T12 |
| Type of test | Module testing |
| Objective No. | 14 |
| Purpose of test | To check whether my program prompts a user to enter a person’s mother or father into the database if they have not been added yet before drawing the tree |
| Description of test | 1. Program should be able to display many people and drop-down menus below each other. 2. A window should not be displayed if there are no mothers/fathers   For the purposes of this test, I will just test the mothers, as the code for the mothers and fathers is almost identical. |
| Test data | |  | | --- | |  | |  | |
| Expected result | 1. Alpha, Bruce, Max and Emma only should be displayed. 2. No mother options should be displayed, but for the fathers ‘Emma Thomas’ should be displayed. |
| Actual result | |  | | --- | |  | |  | |
| Description | Both of my tests worked perfectly. |

**FAILED TESTS**

|  |  |
| --- | --- |
| Fail No. | F1 |
| Relates to Test | T2 |
| Description | When I was on the ‘edit an entry option’ and click main menu, instead of going back to the main menu, the program would just close. First, I tried changing the code from (1) to (2), but this would cause a main menu option to appear as soon as you clicked the “edit an entry” option., as in (3). Therefore, I did not change my code to (2) but instead, I added the highlighted section in (4), and that made the code work. |
| Code before | **def** change\_info(person):  DatabaseAndWindowTasks.create\_and\_enter\_form(**"EDIT AN ENTRY"**,**"2"**,person) |
| Code after | **def** change\_info(person):  DatabaseAndWindowTasks.create\_and\_enter\_form(**"EDIT AN ENTRY"**,**"2"**,person)  display\_menu() |
| Any evidence | |  |  | | --- | --- | | 1. | **def** edit\_person():  file\_exists = DatabaseAndWindowTasks.check\_file\_exists()  **if** file\_exists == **False**:  enter\_new\_person()  **else**:  choose\_person(**"Choose Which person To Amend"**, **"2"**) | | 2. | **def** edit\_person():  file\_exists = DatabaseAndWindowTasks.check\_file\_exists()  **if** file\_exists == **False**:  enter\_new\_person()  **else**:  choose\_person(**"Choose Which person To Amend"**, **"2"**)  display\_menu() | | 3. |  | | 4. | **def** change\_info(person):  DatabaseAndWindowTasks.create\_and\_enter\_form(**"EDIT AN ENTRY"**,**"2"**,person)  display\_menu() | |

|  |  |
| --- | --- |
| Fail no. | F2 |
| Relates to Test | T3 |
| Description | While I was doing test number 3, I found out that my program would throw an error, as seen in (1). Because there were no people, a pop-up menu with all the people in the database could not be made, as there were not any people. To fix this, you can see the code that I have changes below, so that if the length of choices is 0, then you are forced to enter a person into the database first. |
| Code before | **with** sqlite3.connect(**"Family\_1.db"**) **as** conn:  cursor = conn.cursor()   window3=Tk()  window3.title(**"Amending a record"**)  window3.configure(background=**"Light blue"**)  *# label for choosing row to edit* Label(window3, text=message, bg=**"Light blue"**, fg=**"black"**, font =**"none 12 bold"**) .grid(row=0,column=0, sticky=W)  tkvar\_edit= StringVar(window3)  choices\_edit=[]  **for** row **in** cursor.execute(**"""SELECT \* FROM tblFamily WHERE PersonID NOT LIKE '%Mother%'  AND PersonID NOT LIKE '%Father%'  AND PersonID NOT LIKE '%\_P%' """**): *### Gets everyone in the database apart from a select few for the choices* choices\_edit.append(row)   tkvar\_edit.set(**'-'**)  popup\_menu\_edit=OptionMenu(window3, tkvar\_edit, \*choices\_edit)  popup\_menu\_edit.grid(row=0, column=1, sticky=W) |
| Code after | **with** sqlite3.connect(**"Family\_1.db"**) **as** conn:  cursor = conn.cursor()  choices\_edit=[]  **for** row **in** cursor.execute(**"""SELECT \* FROM tblFamily WHERE PersonID NOT LIKE '%Mother%'  AND PersonID NOT LIKE '%Father%'  AND PersonID NOT LIKE '%\_P%' """**): *### Gets everyone in the database apart from a select few for the choices* choices\_edit.append(row)  **if** len(choices\_edit)==0:  enter\_new\_person()  **else**:  window3 = Tk()  window3.title(message)  window3.configure(background=**"Light blue"**)  Label(window3, text=message, bg=**"Light blue"**, fg=**"black"**, font=**"none 12 bold"**).grid(row=0, column=0,sticky=W)  tkvar\_edit = StringVar(window3)  tkvar\_edit.set(**'-'**)  popup\_menu\_edit=OptionMenu(window3, tkvar\_edit, \*choices\_edit)  popup\_menu\_edit.grid(row=0, column=1, sticky=W) |
| Any evidence | |  | | --- | |  | |

|  |  |
| --- | --- |
| Fail no. | F3 |
| Relates to test | T4 |
| Description | This failed test was for test 4, part e). When I typed in a string as a DOB and DOD, the program would append the DOB and DOD as empty. But I did not want this to happen, as they would not be aware that they need to add a DOB and DOD (if they want to) to the record that they entered. This was because if an invalid DOB or DOD was entered, they would get changed to “” and then when it was passed into the function that determines whether to display “Invalid Entry”, they would both be deemed as valid dates and so would be kept as that and the record would be added. So, I changed my code so that if an exception is caught, the DOB and DOD are both set to -1, and the check\_dob\_and\_dod() algorithm then checks for the -1’s, and will turn All Info False if it detects any. |
| Code Before | **try**:  DOB = datetime.date(int(textentry\_birth\_yyyy.get()), int(textentry\_birth\_mm.get()), int(textentry\_birth\_dd.get())) **except** Exception:  DOB= “”  **try**:  DOD = datetime.date(int(textentry\_death\_yyyy.get()), int(textentry\_death\_mm.get()), int(textentry\_death\_dd.get())) **except** Exception:  DOD= “” |
| Code After | **if** len(textentry\_birth\_yyyy.get())==0 **and** len(textentry\_birth\_mm.get())==0 **and** len(textentry\_birth\_dd.get())==0:  DOB=**"" else**:  **try**:  DOB = datetime.date(int(textentry\_birth\_yyyy.get()), int(textentry\_birth\_mm.get()), int(textentry\_birth\_dd.get()))  **except** Exception:  DOB=-1  **if** len(textentry\_death\_yyyy.get())==0 **and** len(textentry\_death\_mm.get())==0 **and** len(textentry\_death\_dd.get())==0:  DOD=**"" else**:  **try**:  DOD = datetime.date(int(textentry\_death\_yyyy.get()), int(textentry\_death\_mm.get()), int(textentry\_death\_dd.get()))  **except** Exception:  DOD=-1  **def** check\_dob\_and\_dod(DOB,DOD,AllInfo):  **if** DOB==**"" and** DOD==**""**:  **return** AllInfo  **if** DOB==-1 **or** DOD==-1:  AllInfo=**False  return** AllInfo  **else**:  **if** DOB!=**""**:  **if** DOB > date.today():  AllInfo=**False  if** DOD!=**""**:  **if** DOD > date.today():  AllInfo=**False  if** DOB!=**"" and** DOD!=**""**:  **if** DOB>DOD:  AllInfo=**False   return** AllInfo |
| Any Evidence |  |

|  |  |
| --- | --- |
| Fail no. | F4 |
| Relates to test no. | T2 |
| Description | When entering (1), after submitting, the record that would be stored for Ellie would be stored as in (2). Note that ‘Father of Tom Wood…” was stored as Father. Also, even though ‘Mother of Tom Wood’ is correct, it was only after editing my code that I changed it from “Mother” to what it is now. Below is the code that causes ‘Father’ to be stored and displayed, and the code that causes ‘Mother of Tom Wood…” to be stored and displayed. Since the code for the mothers and fathers is identical (the only difference is that mother is used where father used in the respective subroutines), I have only pasted code for either the mother or the father and not both. Also, I decided to change the SQL statements (after discussion with one of my end users), so that the drop-down menus contained less information, improving clarity. You can see in (3) that after editing the subroutine that deals with Fathers too, the program now works. |
| Code before | **def** check\_unknown\_father(Father, Firstname, Surname, DOB): *# Same process as above but with Fathers* **if** Father == **"UNKNOWN"**:  FatherID = **"Father of "** + Firstname + **" "** + Surname + **" "** + **"born on"** + **" "** + **" "** + str(DOB)  **else**:  valid\_f = re.match(**"[F]{1}[a]{1}[t]{1}[h]{1}[e]{1}[r]{1}"**, Father[3:10])  **if** valid\_f:  FatherID = Father[3:len(Father) - 3]  **else**:  FatherID = str((Father[2:8]))  **return** FatherID  **for** row **in** cursor. execute (**"SELECT \* FROM tblFamily WHERE tblFamily.Gender='Female' Order BY Firstname"**): |
| Code after | **def** check\_unknown\_mother(Mother, Firstname, Surname, DOB): *#Checks if mother is 'Unknown and if its, returns the respective ID.* **if** Mother == **"UNKNOWN"**:  MotherID = **"Mother of "** + Firstname + **" "** + Surname + **" "** + **"born on"** + **" "** + **" "** + str(DOB)  **else**:  ValidM = re.match(**"[M]{1}[o]{1}[t]{1}[h]{1}[e]{1}[r]{1}"**, Mother[2:8]) *# Regular Expressions (RE) check if Mother in ID. If it is, then formatting is needed* **if** ValidM: *# To generate MotherID* MotherID=copy.deepcopy(Mother[2:len(Mother)-3])  **else**:  MotherID = str((Mother[2:8]))  **return** MotherID  **for** row **in** cursor.execute(**"SELECT PersonID,Firstname,Surname,DOB FROM tblFamily WHERE tblFamily.Gender='Female' Order BY Firstname"**): |
|  | |  |  | | --- | --- | | 1 |  | | 2 |  | | 3 |  | |

|  |  |
| --- | --- |
| Fail no. | F5 |
| Test no. | T6 |
| Description | When doing test number 6, my program would crash (normally due to sql statements) when it was trying to deal with people who had at least one un-entered parent, and there were not any such people. To fix this, I created a new program and imported into sorting algorithm, that first checked if there were such people. |
| Code before |  |
| Code after | **New Program**  **import** sqlite3  **def** check\_add\_1\_unknown():  people=[]  conn=sqlite3.connect(**"Family\_1.db"**)  cursor=conn.cursor()  **for** row **in** cursor.execute(**"SELECT \* FROM tblFamily WHERE MotherID = '""' OR FatherID = '""' AND (MotherID <> FatherID) "**):  people.append(row)  **if** people==**None or** len(people)==0:  need\_add\_1\_people=**False  else**:  need\_add\_1\_people=**True   return** need\_add\_1\_people  **def** check\_add\_2\_people():  people=[]  conn=sqlite3.connect(**"Family\_1.db"**)  cursor=conn.cursor()  **for** row **in** cursor.execute(**"SELECT \* FROM tblFamily WHERE MotherID = '""' AND FatherID= '""' "**):  people.append(row)  **if** people==**None or** len(people)==0:  need\_add\_2\_people=**False  else**:  need\_add\_2\_people=**True  return** need\_add\_2\_people  **Sorting Algorithm**  **import** Check\_Add\_Unknown\_People\_20\_03\_2019 **as** CheckAddUnknowns  need\_add\_1\_people=CheckAddUnknowns.check\_add\_1\_unknown() **if** need\_add\_1\_people==**True**:  one\_unknown\_parent(children\_dict)  children\_dict = add\_unknown\_parents\_to\_children\_set(children\_dict) need\_add\_2\_people=CheckAddUnknowns.check\_add\_1\_unknown() **if** need\_add\_2\_people== **True**:  unknownm, unknownf=add\_two\_unknowns(children\_dict)  children\_dict=add\_2\_unknowns\_to\_children\_set(unknownm,unknownf,children\_dict) |
| Any evidence |  |

|  |  |
| --- | --- |
| Fail no. | F6 |
| Test no. | T9 |
| Description | During a beta test for the statistics option, my friend entered himself into the database and then when he clicked on the statistics option when he was the only person in the database, an error was produced, as some of the variables could not be calculated. Therefore, I had to change my code to allow for this. |
| Code before | **def** get\_statistics():  file\_exists = DatabaseAndWindowTasks.check\_file\_exists()  **if** file\_exists == **False**:  enter\_new\_person()  **else**:   window1=Tk()  window1.title(**"Statistics For People Who Are Currently Alive"**)  window1.configure(background=**"Light blue"**)  mode,median, mean\_age,variance,standard\_deviation, max\_age\_people,max\_age\_people\_list,min\_age\_people,min\_age\_people\_list=Stats.statistics\_main() |
| Code after | **def** get\_statistics():  file\_exists = DatabaseAndWindowTasks.check\_file\_exists()  **if** file\_exists == **False**:  enter\_new\_person()  **else**:   window1=Tk()  window1.title(**"Statistics For People Who Are Currently Alive"**)  window1.configure(background=**"Light blue"**)  **try**:  mode\_age,median\_age, mean\_age,variance,standard\_deviation,  max\_age\_people,max\_age\_people\_list,min\_age\_people,min\_age\_people\_list=  Stats.statistics\_main()  **except** Exception:  mode\_age, median\_age, mean\_age, variance, standard\_deviation, max\_age\_people,  max\_age\_people\_list, min\_age\_people, min\_age\_people\_list =  Stats.statistics\_main\_1\_person()  **def** statistics\_main\_1\_person():  all\_people = list\_all\_alive()  ages\_dict = get\_ages()  just\_ages\_list = just\_ages(ages\_dict, all\_people)  mode\_age=just\_ages\_list[0]  median\_age=just\_ages\_list[0]  mean\_age=just\_ages\_list[0]  variance=0  standard\_deviation=0  max\_age\_people, max\_age\_people\_list = max\_age(just\_ages\_list, ages\_dict, all\_people)  min\_age\_people, min\_age\_people\_list = min\_age(just\_ages\_list, ages\_dict, all\_people)  **return** mode\_age, median\_age, mean\_age, variance, standard\_deviation, max\_age\_people,  max\_age\_people\_list, min\_age\_people, min\_age\_people\_list |
| Any evidence? | |  | | --- | |  | |  | |

**EVALUATION**

|  |  |  |
| --- | --- | --- |
| Core Objectives | | |
| Number | Objective | Evaluation |
| 1 | Create a GUI interface that connects to a back-end database. | -Achieved-  My project has a user interface that was quite user friendly for the purposes of his project. But, when talking to my end-users, a common criticism was that adding, editing and deleting a person should be on the same window, so that it is easier to carry out the functions as currently, a person has to go back to the main menu if they want to edit a person after entering or deleting a person. |
| 2 | Allow users to add, delete and edit people in the back-end database. | -Achieved-  My GUI has all of these functions, and they can be done easily by the end user. |
| 3 | Create tblFamily if it does not exist (even when edit option or delete option is chosen and no-one is in the database) | -Achieved-  The table is created every time without fail, when a user enters a person for the first time. Also, when the user presses the ‘edit person’ or ‘delete person’ button as soon as they enter the program, they are forced to enter a person. |
| 4 | To deal with people who enter ‘Unknown’ as their mother or father. | -Achieved-  When a person (for example Tom) chooses their father as ‘Unknown’ then, his father will be stored as “Father of Tom...2 and if the user was to enter any of Tom’s siblings, they would be entered as “Father of Tom…” |
| 5 | Create a Queue that contains family ID’s | -Achieved-  This queue contains family IDs starting with FID1. |
| 6 | Put everyone into families and put these families into the database | -Achieved-  My program carries out this task, and puts everyone into families, based on if they share children or not. One criticism that I have of my own code was that if for example Bob (with PersonID BG1834) had no father in the database, his father would be stored as ‘Father of Bob…”. But, looking back, this just made things more difficult and instead, should have stored it as BG1834\_F, so that it would be easier to manipulate, as then other people who have no parents entered would have their parents entered in the same format, and so all the ID’s would be of length 8. However, I do not think that my program would work if step-parents etc. were added and so if I had more time, I would allow my program to account for this. |
| 7 | Delete and create tblSingleFamily and tblChildren each time the ‘draw tree’ or ‘find relationships’ option is run | -Achieved-  My program does this every time the ‘draw tree’ or ‘find relationships’ option is chosen. Although this removes any inconsistencies in the data and makes sure that everyone is included, if I had more time I would make my program more efficient by not recreating the families every time, but adding or deleting or updating records each time. |
| 8 | Create a stack that can be used for the depth first traversal. | -Achieved-  My program fulfils this objective fully, as it is a LIFO data structure, and has functions to pop, peek and push items onto the stack. |
| 9 | Create an algorithm that produces a family matrix | -Achieved-  My program uses a recursive algorithm that uses a depth first search and traverses the whole graph, assigning 1 or -1 to each row or column (depending which one is beneath which one), the linking person, and the IDs of the 2 families. |
| 10 | Decide which people are on the same row | -Achieved-  My program uses a recursive algorithm to find the distances relative to FID1 for each family. Because ‘going up’ the tree has a weight of -1 and ‘going down’ the tree has a weight of 1, people who were on the same generation had the same distance from FID1 |
| 11 | Draw the family tree | -Achieved-  My program achieved this. It drew a tree, that was arranged into rows and had families with a common parent-family closer together. But, when talking to my end-users, they would prefer the ‘find relationships’ option to be integrated into the tree, so that for example if you right clicked on two people, the relationship between them would be displayed. |
| 12 | Draw lines between connected families that are different colours depending on whether the person linking the 2 families is male or female. | -Achieved-  My program achieves this, and draws lines from a person to the centre of the family above it. Having experimented with the colours, I have chosen pink and blue as default colours, but this could be made selectable in the future. |
| 13 | Display information when someone clicks on a person in the family tree | -Achieved-  When somebody clicks on a person, the information of that person (which are all of the attributes in tblFamily apart from the PersonID) are displayed at the bottom of the screen. |
| 14 | Prompt user to enter a mother/father for people who have not had one entered. | -Achieved-  When the user chooses to draw the tree for example, they are first shown popups with a list of drop-down menus, where they can optionally add mothers and fathers for individuals in order to make the family tree more comprehensive. |
| 15 | Have exception handling (data validation) when entering data into the form. | -Achieved-  When the user enters an invalid date of birth or does not enter a first name or surname for a person for example, a pop-up window appears saying invalid entry. When this happens, the data that the user entered is not added to the database. |
| Extension Objectives | | |
| 1 | To be able to choose 2 people, and have the program display the relationship between them. | -Partially achieved-  My program partially achieves. This is because although my program identifies most relationships, e.g. a person is the nephew of another or their auntie’s husband, it cannot differentiate between cousin levels. For example, it can tell you if 2 people are cousins, but not if they are first, second or third cousins, which is what some of my end users would have liked to have seen. |
| 2 | To display some simple statistics such as mean age, standard deviation of ages etc. | -Partially Achieved-  My program calculates all of the statistics specified in the cell to the left but, my end-users said that they would like to have some filters on the statistics, so for example instead of just displaying the statistics for everyone alive, be able to display statistics for everyone born in the 19th century for example or everyone who is female. This should not be too hard to code, but may take a bit too much time, which I unfortunately do not have. But if I was to continue this project, I would definitely add these filters. |
| 3 | Display pie charts etc. of the data in excel | -Not achieved-  Unfortunately, I did not get around to coding this section and while the end users stated that it would have been a nice addition to the project, they overall said that it was not that important and that they could easily just do it themselves, if they had access to the back-end database. |
| 4 | Be able to import data from a text file (csv) | -Not Achieved-  There was not enough time to do this, but the end users said that they would rather just use my GUI instead, as they have little to no experience in using csv/text files, and that the effort spent on typing the csv, would be almost equivalent to the effort needed to do it via the GUI. Also, it would have been an inconvenience for users to specify links between a person and their parents, and to code these links by generating Person IDs and then Mother/ Father IDs for each person. In fact, a different system altogether may have needed to have been implemented. |
| 5 | Create a geographical map | -Not Achieved-  I did not have any time at all to create this, as close to the end of my project, I decided that it would be better to refine some of my core objectives, instead of adding this feature, which I may not even finish. If I did this in the future I would probably use the Google Maps API. |
| 6 | Be able to click on people in the tree and have a zoom in/out function or perhaps a scroll function. | -Partially achieved-  Although I did not create a zoom in/out function, I did make the window resizable, and so if you made the window full screen, all the families would fit on the screen. But, at the moment you cannot zoom in or out of particular parts. |

**Interview with end-users**

**Interview #1 - Dad**

**What are your thoughts and feelings of my program? What did it do well and what could I improve?**

“I thought that the program was quite easy to use, apart from the fact that I had to click on buttons to take me to different windows to then edit/add/delete a person. I would much prefer it if I could easily do all of these functions on one screen. It also took quite a while to enter everyone I wanted into the program but, I do not know how you could change that. The display statistics option was quite good but, I would rather have statistics such as the male to female ratio, than to have the ‘variance’ and ‘standard deviation’ of everyone’s age - to a person like myself, these 2 statistics in particular mean nothing. I would have also liked more variety in the statistics; they all seemed to be focused on age. The ‘find relationships’ option worked perfectly apart from occasional minor grammatical mistakes in the output. However, this does not matter at all. The draw-tree option was very accurate but, for some people their name would not fit in the box. So, if you were going to make another version, that would have to be fixed.”

**Interview #2 - Computer Science Teacher**

**What are your thoughts and feelings of my program? What did do well and what could I improve?**

**“**The program was very well developed considering that it was an A-Level project. The functionalities of the GUI that connected to the back-end database all worked but, I found having to enter everyone into the database one by one via the GUI quite tedious - I would have really liked the option to import data from a CSV file. Also, the pop-up that appears when an invalid entry was made could have looked better, but this is just a minor fault. On the other hand, the ‘get statistics’ could have been better. I believe that an option where you can choose filters from a drop-down menu could have improved the program. The ‘find relationships’ option worked quite well but, I would like more specific relationships to sometimes be returned. Finally, the ‘draw tree’ option was interesting. I had never thought of representing the family tree using the ‘nuclear family’ as the unit. It makes sense because less lines have to be drawn so it’s far tidier and easier on the eye. Not only that, but I think that you made the right choice to draw the tree in Kivy, rather than Tkinter or Turtle, as not only is the family tree interactive (when you click on a person), but it also looks aesthetically pleasing.”

**Overall Improvements Summary**

1. Account for step parents, as the end users stated that this would make the project more comprehensive. If I did this, I may have to rethink the way that I put people into families and the way that I test which family people are children/parents of.
2. Create a better GUI, where you can add, enter and delete people on the same window. This should not be too difficult, but it was just not something that I thought of at the time.
3. Create a better zoom function and a scroll function, to create a better viewing experience for the user. Sometimes peoples’ names would extend outside of the box. So, this should be improved so that everyone’s names are contained inside of their box. If everyone does not fit on the screen as a consequence of this, a scroll function needs to be created.
4. Be able to import data from a csv file - although not everyone will have used it, it may have allowed me to speed up the testing process, as I would be able to add or delete multiple records into the database quicker.
5. Make the relationship be able to differentiate between different cousins for example. For this to happen, I would somehow need to retrieve how far apart ‘horizontally’ people are on the family tree instead of just by generation.
6. To create simple filters for the statistics such as, only females/males and also by century born in etc. This again should not be too hard to code, but I did not have enough time to create this option in the end.
7. Export data to an Excel document, that creates pie charts and bar charts for different demographics such as age and place of birth. This would require me to download another module such as ‘xlwt’.   
   Alternatively, charts could also be displayed using a module such as matplotlib.

|  |
| --- |
| APPENDICES |

**PROTOTYPE CODE**

*# Set-up person class***import** sys  
**import** random  
**import** datetime  
**import** sqlite3  
**import** os.path  
**from** tkinter **import** \*  
  
  
  
**def** DisplayMenu():  
   
 print(**"""  
  
 MAIN MENU:  
  
 1. Enter information about a new person  
  
 2. Edit a record  
  
 3. Delete a record  
  
 4. View information  
  
 5. Import information from a text file  
  
 9. Exit  
  
  
 """**)  
   
  
**def** MenuChoice():  
 DisplayMenu()  
 Option=int(input(**":"**))  
   
  
 **if** Option == 1:  
 Option1()  
  
 *#get all the info for a new person  
 #create new Person* **if** Option ==2:  
 print(Option2())  
 MenuChoice()  
  
 **if** Option ==3:  
 Option3(**"Family\_1.db"**)  
 MenuChoice()  
  
 **if** Option ==4:  
 Option4()  
 MenuChoice()  
  
 **if** Option ==5:  
 Option5()  
   
   
 **if** Option == 9:  
 sys.exit()  
  
  
   
   
   
  
**class** Person:  
  
 **def** \_\_init\_\_(self,Firstname,Surname,DOB,DOD,POB,Gender):  
 self.Firstname = Firstname  
 self.Surname = Surname  
 self.DOB = DOB  
 self.DOD = DOD  
 self.POB = POB  
 self.Gender = Gender  
  
  
 **def** EnterFirstname():  
 Firstname=str(input(**"What is your Firstname:"**))  
 **return** Firstname  
  
 **def** EnterSurname():  
 Surname=str(input(**"What is your Surname:"**))  
 **return** Surname  
  
 **def** CreateRandomID(Firstname,Surname):  
 PrevRandomDigit = [] *# NEED TO MAKE SURE ANOTHER RANDOM DIGIT THAT IS SAME IS CREATED* ValidRanDigit = **False** Count = 0  
 FirstLetter = Firstname[0]  
 SecondLetter=Surname[0]  
 **while** ValidRanDigit == **False**:  
 RandomDigit=str(random.randint(1000,9999))  
 **for** x **in** range(0,len(PrevRandomDigit)):  
 **if** int(PrevRandomDigit[x]) == int(RandomDigit):  
 Count = Count+1  
 **else**:  
 **pass  
 if** Count == 0 :  
 ValidRanDigit = **True** PrevRandomDigit.append(RandomDigit)  
 **else**:  
 ValidRanDigit = **False** PersonID = str(FirstLetter + SecondLetter + RandomDigit)  
 **return** PersonID  
  
 **def** EnterGender():  
 Gender = str(input(**"Are you biologically male or female"**))  
 **return** Gender  
 **def** EnterPOB():  
 POB=str(input(**"Where were you born"**))  
 **return** POB  
  
 **def** EnterDOB():  
 Day= int(input(**"What day were you born on"**))  
 Month = int(input(**"Which month"**))  
 Year = int(input(**"Which year"**))  
 DOB = datetime.date(Year,Month,Day)  
 **return** DOB  
  
 **def** EnterDOD():  
 Day= int(input(**"What day did that person die on"**))  
 Month = int(input(**"Which month"**))  
 Year = int(input(**"Which year"**))  
 DOD = datetime.date(Year,Month,Day)  
 **return** DOD  
  
 **def** EnterMotherFirstname():  
 MotherF=input(**"Enter your mother's Firstname (If unknown type 'UNKNOWN'):"**)  
 **return** MotherF  
  
 **def** EnterMotherSurname():  
 MotherS=input(**"Enter your mother's Surname (If unknown type 'UNKNOWN'):"**)  
 **return** MotherS  
  
 **def** EnterFatherFirstname():  
 FatherF=input(**"Enter your father's Firstname (If unknown type 'UNKNOWN'):"**)  
 **return** FatherF  
  
 **def** EnterFatherSurname():  
 FatherS=input(**"Enter your father's Surname (If unknown type 'UNKNOWN'):"**)  
 **return** FatherS  
  
 **def** EnterMotherID():  
 MotherID=input(**"Enter the unique ID of your mother (IF NOT KNOWN LEAVE IT BLANK):"**)  
 **return** MotherID  
  
 **def** EnterFatherID():  
 FatherID=input(**"Enter the unique ID of your mother (IF NOT KNOWN LEAVE IT BLANK):"**)  
 **return** FatherID  
  
 **def** IsMotherKnown(MotherF,MotherS):  
 MotherKnown = **False  
 if** (MotherF ==**'Unknown' and** MotherS == **'Unknown'**):  
 MotherKnown = **False  
 else**:  
 MotherKnown = **True  
  
 return** MotherKnown  
   
 **def** IsFatherKnown(FatherF,FatherS):  
 FatherKnown = **False  
 if** (FatherF ==**'Unknown' and** FatherS == **'Unknown'**):  
 FatherKnown = **False  
 else**:  
 FatherKnown = **True  
  
 return** FatherKnown  
   
 **def** IsMotherIDKnown(MotherID):  
 MotherIDKnown = **False  
 if** MotherID == **""**:  
 MotherIDKnown = **False  
 else**:  
 MotherIDKnown = **True  
 return** MotherIDKnown  
   
 **def** IsFatherIDKnown(FatherID):  
 FatherIDKnown = **False  
 if** FatherID == **""**:  
 FatherIDKnown = **False  
 else**:  
 FatherIDKnown = **True  
 return** FatherIDKnown  
   
 **def** DecideCreateMother(IsMotherKnown,IsMotherIDKnown):  
 **if** IsMotherKnown == **False**:  
 **pass  
   
 if** IsMotherKnown == **True and** IsMotherIDKnown == **True**:  
 **pass  
   
 if** IsMotherKnown == **True and** IsMotherIDKnown == **False**:  
 **pass  
   
   
   
 def** DecideCreateFather(IsFatherKnown,IsFatherIDKnown):  
 **if** IsFatherKnown == **False**:  
 **pass  
   
 if** IsFatherKnown == **True and** IsFatherIDKnown == **True**:  
 **pass  
   
 if** IsFatherKnown == **True and** IsFatherIDKnown == **False**:  
 **pass  
   
  
 def** GetAllInfo():  
 Firstname=Person.EnterFirstname()  
 Surname=Person.EnterSurname()  
 PersonID = Person.CreateRandomID(Firstname,Surname)  
 DOB = Person.EnterDOB()  
 DOD = Person.EnterDOD()  
 POB = Person.EnterPOB()  
 Gender = Person.EnterGender()  
 **return** Firstname  
 **return** Surname  
 **return** PersonID  
 **return** DOB  
 **return** DOD  
 **return** POB  
 **return** Gender  
  
   
   
   
   
**def** CreateDatabase():  
  
*#Creates database called Family\_1.db  
#save\_path = 'C:\Users\Xman\Google Drive\Dariyan files\1 - Cathedral\A-levels\Computer Science\1-NEA Project\Family tree\Database'  
#save\_path = os.path.join(save\_path, "Family\_1.db")* connection = sqlite3.connect(**"Family\_1.db"**)  
 cursor = connection.cursor()  
  
*#SQL commands to create table* sqlCommand = **"""  
  
 CREATE TABLE `tblFamily`  
 (  
 `PersonID` TEXT,  
 `Firstname` TEXT,  
 `Surname` TEXT,  
 `Gender` TEXT,  
 `DOB` DATE,  
 `DOD` DATE,  
 `POB` TEXT,  
 `MotherFirstname` TEXT,  
 `MotherSurname` TEXT,  
 `FatherFirstname` TEXT,  
 `FatherSurname` TEXT,  
 PRIMARY KEY(`PersonID`)  
 )"""** cursor.execute(sqlCommand)  
 print(**"Table is created"**)  
 connection.commit()  
 connection.close()  
   
   
*#def ConnectToDatabase():  
 # conn=sqlite3.connect("Family\_1.db")  
 # cursor = conn.cursor()  
 # return cursor  
  
#def CloseDatabase():  
 #conn.commit()  
 #conn.close()  
 #print("Changes have been successfully made")***def** CheckIfExists():  
 FileExists = os.path.exists(**'Family\_1.db'**)  
 **return** FileExists  
  
  
   
   
  
*#Firstname=Person.EnterFirstname()  
#Surname=Person.EnterSurname() ####CODE TO GENERATE RANDOM PERSON ID  
#print(Person.CreateRandomID(Firstname,Surname))  
  
  
  
###########################################################################################################################################################################################################  
  
############################################ IF OPTION IS 1 ###############################################################################################################################################***def** Option1():  
 EnterInfo = **True** FileExists = os.path.exists(**'Family\_1.db'**)  
 **if** FileExists==**True**:  
 conn=sqlite3.connect(**"Family\_1.db"**)  
 cursor = conn.cursor()  
   
 **if** FileExists ==**False**:  
 connection = sqlite3.connect(**"Family\_1.db"**)  
 cursor = connection.cursor()  
  
*#SQL commands to create table* sqlCommand = **"""  
  
 CREATE TABLE `tblFamily`  
 (  
 `PersonID` TEXT,  
 `Firstname` TEXT,  
 `Surname` TEXT,  
 `Gender` TEXT,  
 `DOB` DATE,  
 `DOD` DATE,  
 `POB` TEXT,  
 `MotherF` TEXT,  
 `MotherS` TEXT,  
 `FatherF` TEXT,  
 `FatherS` TEXT,  
 'MotherID' TEXT,  
 'FatherID' TEXT,  
 PRIMARY KEY(`PersonID`)  
 )"""** cursor.execute(sqlCommand)  
 print(**"Table is created"**)  
 connection.commit()  
 *#connection.close()* conn=sqlite3.connect(**"Family\_1.db"**)  
 cursor = conn.cursor()  
 PersonRec=[]  
 **while** EnterInfo == **True**:  
 Option4()  
 print(**""" ABOVE IS ALL CURRENT RECORDS IN THE DATABASE  
  
  
"""**)  
 Firstname=Person.EnterFirstname()  
 Surname=Person.EnterSurname()  
 PersonID = Person.CreateRandomID(Firstname,Surname)  
 DOB = Person.EnterDOB()  
 DOD = Person.EnterDOD()  
 POB = Person.EnterPOB()  
 Gender = Person.EnterGender()  
 MotherF = Person.EnterMotherFirstname()  
 MotherS = Person.EnterMotherSurname()  
 IsMotherKnown = Person.IsMotherKnown(MotherF,MotherS)  
 FatherF = Person.EnterFatherFirstname()  
 FatherS = Person.EnterFatherSurname()  
 IsFatherKnown = Person.IsFatherKnown(FatherF,FatherS)  
 MotherID = Person.EnterMotherID()  
 FatherID = Person.EnterFatherID()  
 IsMotherIDKnown = Person.IsMotherIDKnown(MotherID)  
 IsFatherIDKnown = Person.IsFatherIDKnown(FatherID)  
 Person.DecideCreateMother(IsMotherKnown,IsMotherIDKnown)  
 Person.DecideCreateFather(IsFatherKnown,IsFatherIDKnown)  
 PersonRec.append(PersonID)  
 PersonRec.append(Firstname)  
 PersonRec.append(Surname)  
 PersonRec.append(DOB)  
 PersonRec.append(DOD)  
 PersonRec.append(POB)  
 PersonRec.append(Gender)  
 PersonRec.append(MotherF)  
 PersonRec.append(MotherS)  
 PersonRec.append(FatherF)  
 PersonRec.append(FatherS)  
 PersonRec.append(MotherID)  
 PersonRec.append(FatherID)  
 cursor.execute(**"INSERT INTO tblFamily VALUES (?,?,?,?,?,?,?,?,?,?,?,?,?)"**, PersonRec)  
 conn.commit()  
 PersonRec=[]  
   
   
 Choice= input(**"Would you like to enter information about another person (Y or N)"**).upper()  
 **if** Choice ==**"Y"**:  
 **pass  
 if** Choice == **"N"**:  
 EnterInfo= **False** conn.close()  
 MenuChoice()  
  
**def** Option2():  
 **with** sqlite3.connect(**"Family\_1.db"**) **as** conn:  
 cursor = conn.cursor()  
 **for** row **in** cursor.execute(**"SELECT \* FROM tblFamily"**):  
 print(row)  
 print(**"""  
  
 Fields are PersonID, Firstname, Surname, DOB , DOD, POB, Gender, MotherF, MotherS, FatherF, FatherS  
  
 """**)  
  
 keyfield = input(**"Enter the PersonID of the record to amend:"**)  
  
 keyfield = **"'"** + keyfield + **"'"** field = input(**"Change which field?"**)  
 newvalue = input(**"Enter the new value for this field:"**)  
 newvalue = **"'"** + newvalue + **"'"  
  
 try**:  
 cursor.execute(**"Update tblFamily SET "** + field + **"="** + newvalue + **" Where PersonID= "** + keyfield)  
 print(**"\nRecord Updated"**)  
 **except**:  
 print(**"\nNo record updated - invalid data entered"**)  
  
 **for** row **in** cursor.execute(**"SELECT \* FROM tblFamily"**) :  
 print(row)  
  
**def** Option3(dbname):  
 conn = sqlite3.connect (dbname)  
 **with** conn:  
 cursor = conn.cursor()  
 **for** row **in** cursor.execute(**"SELECT \* FROM tblFamily"**):  
 print(row)  
 DelPersonID = input(**"Enter name of PersonID to delete:"**)  
 keyfield = **"'"** + DelPersonID + **"'"** cursor.execute(**"DELETE FROM tblFamily WHERE PersonID ="** + keyfield)  
 **for** row **in** cursor.execute(**"SELECT \* FROM tblFamily"**):  
 print(row)  
   
  
**def** Option4():  
 **with** sqlite3.connect(**"Family\_1.db"**) **as** conn:  
 cursor = conn.cursor()  
 **for** row **in** cursor.execute(**"SELECT \* FROM tblFamily"**):  
 print(row)  
 print(**"""  
  
 Fields are PersonID, Firstname, Surname, DOB , DOD, POB, Gender, MotherF, MotherS, FatherF, FatherS  
  
 """**)  
  
**def** Option5(FamilyFile):  
 connection = sqlite3.connect(**"Family\_1.db"**)  
 cursor = connection.cursor()  
  
 numRecs = 0  
 FamilyDBRec = []  
 FamilyTextRec = FamilyFile.readline()  
 **while** FamilyTextRec != **""**:  
 numRecs +=1  
 field = FamilyTextRec.split(**","**)  
 PersonID = field[0]  
 Firstname = field[1]  
 Surname = field[2]  
 Gender = field[3]  
 DOB = field[4]  
 DOD = field[5]  
 POB = field[6]  
 MotherFirstname = field[7]  
 MotherSurname = field[8]  
 FatherFirstname = field[9]  
 FatherSurname = field[10]  
   
   
   
  
MenuChoice()

**GANTT CHART**



**UNEXPECTED PROBLEMS**

|  |  |  |  |
| --- | --- | --- | --- |
| Problem | How I dealt with it | Date problem found | Date problem fixed |
| At first, I wrote much of my code in order to allow users to enter information into python, without a user interface. This meant that much of my code that I had written had to be deleted/ edited. | I used a tkinter interface and had to re-write much of my code, so that the data entered into the database was taken from text boxes and drop-down menus instead. But, the good thing about coding drop-down menus etc. in a tkinter interface, is that it stops invalid information from being entered. | 12/2018 | 1/2019 |
| I need to get the ID of the person’s mother and father. However, a lot of the time the user would not have entered their parents. | When this happens, at that moment in time they will have no parent(s) entered into the database. But, when the decide to draw the tree etc., they will be able to choose a mother/father for each person in the database with no mother or father. If for example Alice had not entered her mother even when prompted again, her mother would simply be ‘Mother of Alice….” | 23/12/2018 | 28/02/2019 |
| When a person enters information into the database and presses ‘submit’, ‘Recorded has been added’ appears in the window. But I wanted it to go after some time. | I could have perhaps used the time.sleep() method after importing time. However, this would mean that the entire program would stop for the length of time, and so the user would not be able to click on any buttons etc. This is not what I wanted. And when I researched how I would go about doing this on ‘Stack Overflow’, the suggestions given such as using the .after() method and forget() seemed to not work………… | 18/01/2019 | Not solved |
| When I needed to take major changes to my project, or try adding a new feature, I needed to create a new file but, it was difficult to arrange all of the files. | I would date and write a short description of how each file was different from the other files. | 18/01/2019 | 18/01/2019 |
| When I started coding my project, I did not use the conventions used to name variables. | I decided to change this by defining any new variables that I made with the convention, and then once the project was finished, change any remaining variables. | 18/01/2019 | 07/03/2019 |
| How to display the family tree | Originally, I was going to do a standard family tree, where everyone had an individual rectangle, and the boxes would be spread out. However, I felt that coding a family tree this way would become extremely unclear as many people are added to the database, due to the fact that there would be many lines connecting people, and having the program to tell the difference between married people and siblings would be hard. So, I would store each family (consisting of the parents and then the children) as a single unit. The advantage of this was that it would make the family tree have fewer lines, be tidier and be more intuitive.  Furthermore, I looked up different ways to represent families before coming up with this idea. One of them was the ‘Ahnentafel’ method, where each person is assigned a positive integer, and then that person’s father is double that number, and the mother would be double + 1. However, one problem with this method was that it did not allow for siblings. Another system was the Henry system. The system begins with 1. The oldest child becomes 11, the next child is 12, and so on. The oldest child of 11 is 111, the next 112, and so on. I felt that this system was more comprehensive than the Ahnentafel system but, the problem about displaying the family tree would still not be solved. | 27/01/19 | 28/01/19 |
| Iterating over a list | if I had a list called families for example, in order to deal with every element in the list I would write for x in range(0, len(families)): But, today I found out that you can just do for fam in families:  for example, and this makes the code much more readable. Therefore, I decided to go back and change quite a few (but not all of them as some were in complicated subroutines) to this new format. | 27/01/19 |  |
| Table design | Following this change in how I was going to display the family, I would have to store a FamilyID or each family. This required a new table to be created, with primary key FamilyID. This would be stored with the mother and father of the unit. The children would have to be stored too however, if we stored them in the table with primary key FamilyID, there would be a repeating group in the ‘children’ field. Therefore, A new table for the children had to be created, in with a composite primary key consisting of ‘Person ID’ and ‘FamilyID’ | 27/01/19 | 28/01/19 |
| Finding relatives | While finding each unit of family, I realised that my current program would not recognise that 2 people who have unknown parents but are related would not be recognised. To fix this, when a person with unknown parents was entered into the database, instead of storing their ID as ‘UNKNOWN’, I decided to store their ID as ‘Mother/Father of [Insert Person’s full name and DOB]’. | 30/01/2019 | 30/01/2019 |
| Deciding what abstract data type to use to represent the families | At first, I thought that a tree would be used as suggested by the name ‘family tree’ however, I soon realised that although my data structure used would involve no cycles, it would have to have more than one root node, as you could have grand-parents (for example) on both sides of the tree, that have unknown parents. After research, I came across a type of trees called phylogenetic trees, that are used to represent the evolution of a species. However, I finally decided that my tree was actually not a tree but more generally a graph, that was acyclic, and that I could choose to make directed or undirected. The final part that I had to decide was how I was going to represent my graph. Should I use an adjacency list or an adjacency matrix? A list has the benefit that there would be less wasted space as an adjacency matrix representing a family is likely to be quite sparse. However, a matrix would allow new entries to be added easily, and relationships between people could be tested quicker. Therefore, in the end, I decided to go with an adjacency matrix as I felt that being able to efficiently search and add new nodes to the matrix would be more beneficial for further functionalities in the program. In order to create the matrix, I decided to use a depth-first search. | 01/01/2019 | 02/02/2019 |
| SQL statements | When creating the adjacency matrix for my graph recursively, I ran into the problem that sql queries return an error, if no entity matches the conditions specified in the query. Therefore, particularly for checking if the parents of one family where children of another family in the tree, I had to use regular expressions, so that if any person was stored in the form (“Mother of [Insert name]”) etc., then python would not check them. I also needed to make sure that Python did not check for people who had not entered their mother/father into the database yet, even though they knew who they were. | 02/02/2019 |  |
| Traverse algorithm - FamilyIDs being of different formats | Each FamilyID should be of the format FIDX where X is a positive integer. But sometimes the Family ID’s where not kept in this form. Sometimes they had quotation marks and brackets around them, and other times they were lists. This made string Manipulation increasingly difficult, particularly when ‘splicing’ strings. Therefore, I had to create a new subroutine, that ensured that all of the ID’s were in the same format. | 02/02/2019 | 02/02/2019 |
| Fixing my recursive algorithm | When I implemented by recursive algorithm and was testing to see if it would work, I often got results that would carry on infinitely. This made it difficult to read what was being produced in the shell. In order to combat this, I had to use the time.sleep() method in python, that pauses the program for an inputted number of seconds, so that I could then read what was being produced. | 02/02/2019 | 05/02/2019 |
| My file containing my form | When I wrote this half of my project, I did not do much planning. However, after a while, my code started to look messy with the lots of commented out code, and much of my code was being repeated. Because of this, I decided to go back and ‘clean up’ my code, so that it looked much tidier and ran more efficiently, making future updating and editing much easier. | 8/02/2019 | 9/02/2019 |
| Deleting people from the table and updating parents | When a user chooses to delete someone from the table and they are a parent, the fact that that person’s children would still have their parents as an entry. Also, if someone wanted to update a person’s mother or father ID, I had to write code so that other people with the same mother/father would have their record changed | 12/02/2019 | 13/02/2019 |
| How to draw the tree | After researching, I decided that I would either draw the tree in Kivy or using Python’s turtle graphics. Although using Kivy would mean that I would have to learn how to code using the Kivy module, I decided to use it as it would allow my program to have greater capabilities such as being interactive | 15/02/19 | 18/02/2019 |
| Deciding how to arrange the families | When starting to draw the familyID boxes, I realised that I would need to find a way to decide in which row the boxes would be drawn, depending on the generation that each family is in. To do this, I decided to create a recursive algorithm, and for each family, find the distance (using the matrix I made previously) from FID1. Originally, I was going to use Djikstra’s algorithm however, I soon realised that this was not needed. | 18/02/19 | 20/02/2019 |
| Extra information | I decided to produce a Python file that would display other interesting statistics such as the mean age of everyone, the variance of the ages and the standard deviation of the ages (which are in A-Level maths).  However, I decided to only do it for people who are currently alive. | 19/02/19 | 19/02/2019 |
| Finding the relationship between 2 people | For this part of a project, I decided to re-use my earlier algorithm, that creates the family matrix, but edited slightly. The edits cause the route from each person to FID1 to be found (which is given by the stack). These 2 routes must intersect at some point, and so using this I could find the route from one family to another, then find this route in terms of people, and then find it in terms of relationships e.g. brother, daughter etc. | 24/02/19 | 02/03/2019 |
| Drawing the tree with relative distances. | In my Kivy code, I decided to give the co-ordinates of all my boxes etc. relative values instead of absolute values, so that when i resized the window, everything would stay in proportion. However, when drawing lines between families, the problem was that lines are not widgets in Kivy. They are drawing instructions so, they do not have properties that allow relative positioning. This meant that I would have to calculate and use absolute co-ordinates for each line. The trouble with this was that if the window was re-sized, the lines would not adapt automatically. Rather than re-drawing the lines in a function bound to the ‘resize’ event, I decided to try and find a way to solve the issue still using Kivy language. The solution I came up with was to create a ‘RelativeLayout’ corresponding to each line. This could be positioned using relative co-ordinates so that its bottom left corner and its top right corner corresponded to the end-points. Then I was able to draw a line inside the relative layout specifying the two corners as absolute co-ordinates for the line. Since the relative layout was a widget it would auto-resize and thus so would the line. | 04/03/2019 | 08/03/2019 |
| Indents in Kivy | In Kivy, indents had to be a multiple of 4 spaces. If they were not, an error would be thrown. Some of my Kivy code indents were 3 spaces, and so I had to tediously add spaces to a lot of my code. A common error that was often displayed was: ‘TypeError; ‘NoneType’ object is subscriptable. | 01/03/2019 | 02/03/2019 |

**REFERENCES**

PM Heathcote. (2017). *Learning to Program in Python*. Dorset: PG Online Limited

<https://www.youtube.com/watch?v=_lSNIrR1nZU> - Published on the 16th January 2017. Used in December 2018. ‘Learn Tkinter in 20 minutes’.

<https://kivy.org/doc/stable/> - Published in 2018. Used in February 2019. ‘Kivy Documentation’

|  |
| --- |
| --- END --- |