OCR GCE A-Level

COMPUTER SCIENCE PROJECT

H446-03

Name : MacKenzie Marriott

Candidate Number: <INSERT CANDIDATE NUMBER>

<Institution Name>: Doncaster UTC

Title of Project: Bus Timetable application

H446-03 – Project CONTENTS

Table of Contents

[A. Analysis 4](#_Toc96612091)

[**Project definition:** 4](#_Toc96612092)

[**Why is the program needed?** 4](#_Toc96612093)

[**client interview:** 4](#_Toc96612094)

[Questions: 5](#_Toc96612095)

[responses from Miss bradley (the client): 5](#_Toc96612096)

[**USer survey:** 5](#_Toc96612097)

[Questions 5](#_Toc96612098)

[User responses: 6](#_Toc96612099)

[**Other solutions:** 7](#_Toc96612100)

[**The first bus website:** 7](#_Toc96612101)

[**Other options:** 8](#_Toc96612102)

[**List of features:** 9](#_Toc96612103)

[**Hardware and software requirements:** 9](#_Toc96612104)

[**Hardware** 9](#_Toc96612105)

[**Software** 10](#_Toc96612106)

[**additional libraries or utilities** 10](#_Toc96612107)

[**Computational methods:** 10](#_Toc96612108)

[Features that make the problem solvable via computational methods 10](#_Toc96612109)

[Problem decomposition 11](#_Toc96612110)

[Use of divide and conquer 11](#_Toc96612111)

[use of abstraction 11](#_Toc96612112)

[**Success criteria:** 12](#_Toc96612113)

[B. Design 12](#_Toc96612114)

[Wireframe and prototype 12](#_Toc96612115)

[General design choices 12](#_Toc96612116)

[The navigation bar: 12](#_Toc96612117)

[Main screen 13](#_Toc96612118)

[Route information screen 13](#_Toc96612119)

[Route list / favourite routes 14](#_Toc96612120)

[Client feedback on designs 16](#_Toc96612121)

[pseudocode 18](#_Toc96612122)

[Example code – proof of concept 18](#_Toc96612123)

[Map in tkinter 19](#_Toc96612124)

[Systems diagram 21](#_Toc96612125)

[C. Developing the coded solution (“The development story”) 22](#_Toc96612126)

[D. Evaluation 22](#_Toc96612127)

[Project Appendixes 22](#_Toc96612128)

# A. Analysis

## **Project definition:**

The problem that my project aims to solve is complications with local transport systems. Despite the fact that information is so readily available in the modern day, the transport infrastructure in the UK is lacking compared to other countries like Japan. Busses are often late or if they are simply not running, this is not always reported or reflected on timetables. I aim to create a sophisticated timetabling system which gives you as much information as possible in a well laid out and easy to read manner. This information will come directly from the First Bus API and be displayed in the form of a python-based desktop application.

My stakeholders are teaching staff from the Doncaster UTC, they want to verify the routes that students are coming to school on through the use of bus timetables. However, they currently don’t have the infrastructure to do so. As well as this, one of the stakeholders (Mrs Bradley) has an office in an area of poor WIFI. Whilst other staff may be able to use existing web-based solutions, she cannot load the pages quickly or reliably, meaning she requires an advanced and bespoke solution. They will use this data for when students are late to school or require help planning bus routes, this will show the staff the local bus timetables in order to verify why a student has been late, based on the bus routes and the timetables of these routes.

Current software is slow, unreliable, and is often difficult to navigate to find what information you need in order to plan your journey correctly. As well as this, they require high bandwidth internet to load, due to animations, images, and videos being present on the page. As the stakeholder has poor internet connection, this makes the existing solutions unsuitable for her current needs.

### **Why is the program needed?**

The product in question is a python-based application which monitors the location and status of current, running busses. This includes information about the capacity of any given bus, how many people are on board, if it is on time and whether or not the given bus has broken down. Information about the stops of any route will also be available on the application – this includes what time busses on a given route should arrive and the status of these busses. This application is aimed to provide more accurate and relevant information about local busses using given information for the client, helping them identify why students are turning up late (despite not being able to access the previously required websites). The data I am going to be using is going to be scraped and parsed once per day from various websites with the needed information. The scraping will only occur once per day in order to cut down on WIFI usage given the clients issues. This will draw data between two bounds in order to simulate late busses, amount of people on the bus etc. This information will be drawn from a continuously updated database in order to provide relevant and accurate information about all running busses at a given point in time.

My program aims to allow the staff to be able to access the data and information quickly, reliably, and easily about various bus routes that they need in order to verify students being late due to bus timetables.

## **client interview:**

### Questions:

1. How often would you need to use the product?
2. How often would students need to access the product?
3. Will this product be deployed on multiple terminals?
4. Are there any specific, currently unmentioned features that you wish to include?
5. What current infrastructure or programs do the school use?

#### Question justification:

1. This would show how often we need to update the data (roughly) so that less bandwidth is used by the machines as updates are called less.
2. This is a generic client needs question, finding out how often our product would be accessed.

### responses from Miss bradley (the client):

1. “We would use the product whenever a student either turns up late for school or needs help planning a journey without knowledge of bus routes or timetables.”
2. “Our students may access the product whenever they need to plan a journey using the terminals at the school, these terminals will run the program constantly and help students find their way to their destination using the local bus routes.”
3. “There will be around 2 or 3 terminals placed around the school running the software so that multiple students can use it at the same time. We don’t believe we will need more than this as the likelihood of more than 3 students needing to use it at the same time will be quite low.”
4. “We would like the product to have school branding on it if possible.”
5. “Currently, we do not have much of a supported infrastructure, students however currently try and use the first bus website, with varying success. This website is reasonably hard for students to use and doesn’t contain information about all of the relevant busses.”

## **USer survey:**

### Questions

1. What do you currently use to plan your journeys?
2. How often do you take the bus?
3. What companies’ busses do you use?
4. Are you often late to school?
5. What problems do you have with your current solution?

#### Question justification:

1. This question aims to find out what infrastructure students currently use, in order to identify my competitors and alternative solutions for my problem.
2. This will let us know how often people will (roughly) use the application.
3. This will help us collect a comprehensive list of bus companies to track, making our app both relevant and robust in the information and services we provide.
4. This will help show why we need to fix the problem overall and why our product is helpful.
5. This will help us identify areas to focus on us during the development of our project.

### User responses:

#### user 1:

1. Currently, I use the first bus website and application in order to see bus routes and bus timetables.
2. I take the bus roughly twice per week after school on Tuesdays and Thursdays.
3. I only use first busses, stagecoach do not do busses on the routes that I have to use to get home from school, meaning they’re out of the question.
4. As I do not take the bus to school, I am not often late, instead I am often early or on time as I get a lift from my parents to school every day.
5. Currently, the main problem I have with the application is accuracy, it isn’t updated often enough meaning it could say a bus is running late, when actually it is on time, meaning I miss it altogether, this is very inconvenient and has caused me to miss busses in the past.

#### User 2:

1. I use the UK bus checker application on my phone in order to check the bus times, routes, and the location of stops.
2. I take the bus around 3 times per week after school to go to work on Mondays, Wednesdays, and Fridays.
3. I use either a first bus or a stagecoach bus to get to work, whichever one turns up first (this is generally first busses).
4. As I do not take the bus to school, I am not often late, instead I am often early or on time as I get a lift from my parents to school every day. However, sometimes I am late for work as a result of bus delays or cancellations.
5. My main problem with the app is that it doesn’t alert me to delays, this can cause me to be late to school sometimes as a result of said delays. If I was aware of this, then I would be able to notify my manager at work beforehand.

#### User 3:

1. Currently, I use the stagecoach website on my phone in order to see bus routes and bus timetables.
2. I take the bus to and from school every day as well as sometimes over the weekend if I need to get anywhere.
3. I mainly use stagecoach busses; however, I use first if stagecoach do not cover where I need to go occasionally.
4. I’m often late as a result of bus delays, or sometimes them just not showing up at all. This causes many problems with school and my attendance.
5. Currently, the main problem I have with the application is that it doesn’t tell me if a bus is cancelled or delayed.

## **Other solutions:**

The most prominent and obvious alternative solution to my problem is the use of paper timetables or leaflets. These have been used for a very long time already as they are cheap to make, easy to distribute and reliable as there is no required internet connection or location information. There is however a big problem with this – the information cannot be routinely and easily updated. Even if it could be updated regularly, people may not necessarily be aware that a new timetable is coming out, or that changes are being made. Another apparent problem is the clients’ problems with WIFI availability in her office, with it being effectively a dead-zone.

### **The first bus website:**

Another solution would be to use the existing first bus timetable website, this is able to give you the current timetables in real time and display. This website however is very complex in use yet basic in given information and requires knowledge of bus routes and stop names in order to be used successfully. However, this website does have the benefit that (as it is not an application) it does not need to be installed on your device, meaning it can run on your device without needing to install anything. This can

be handy if you need information quickly or have a poor, but not unusable internet connection.

Graphical user interface, website

Description automatically generated

#### **The first bus website homepage:**

Here you can plan journeys and navigate through the various menus on the website. The menus include the journey planner, timetables, the next bus, buying tickets, and a support section. Whilst the features are clearly accessible, the UI seems (to me) quite cramped and disorganised, this I hope to avoid in my app, making it easier for users to navigate the pages and find what they need. Whilst the colours here are consistent on this page, other pages (as seen next) do not follow this colour scheme, which makes the experience feel quite disjointed.

Graphical user interface, website

Description automatically generated

#### **The timetables page:**

This is the main page that I will be focusing on for the website section of my project. This is a reasonably well-designed page from first, however, I believe that there is too much information presented before the actual timetable comes up. People click on this page to view the timetables, yet first opted to explain their timetables first, which I believe is inefficient and overall reduces the user experience. Along with this (as mentioned earlier) this site does not match the colour scheme of the main home page, which makes the experience of navigating between menus feel disjointed and almost foreign compared to other websites with consistent design.

One part of the app that I do like is how they make new and important information stand out. This is seen through the graphic in clear colours at the top that lets users know that there are currently issues with suspended service. This information in itself is clear without obstructing any other information meaning that it can be scrolled past quicky in order to get to the main focus of the page, whilst also getting the point across.

### **Other options:**

Other apps could also be used – such as UK bus checker. This app allows users to track buses from anywhere around the UK. However, many users have left reviews saying that lately the times stated on the app have become unreliable and haven’t been updated frequently enough. This is a problem which my app aims to solve.

#### Justification of my approaches based on the above solutions:

Among the above reasons, the main reason that these solutions are all unsuitable can all be drawn to one common point. This is that the stakeholder is unable to access the internet constantly or reliably. As a result of this, the client is unable to use the above websites or applications. My application aims to solve this by only requiring a weak WIFI connection around once per day in order to scrape the necessary information off of the relevant web pages.

## **List of features:**

|  |  |
| --- | --- |
| **Feature:** | **Justification of the feature:** |
| Able to give a list of times for busses over each weekday, for each individual bus route. | This feature is important as the entire project for the client is supposed to be a working, relevant, and updated timetable system for Doncaster bus systems. |
| Able to tell you if/when busses are running late, and if they are running late, where they are in real time. | This feature is important to the client, as the software will be used to decide whether or not the student is an appropriate amount late or whether they are lying about why they are late. |
| Able to tell you when to leave based on how close you are to a selected bus and time. | This important for journey planning, as it gives you an indication of when you have to be ready in order to set off for the bus stop at an appropriate time. |

## **Hardware and software requirements:**

### **Hardware**

The following are the hardware requirements for my application:

* Any dual core or better processor to run the application.
* WIFI connectivity for the application once per week in order for it to scrape the data off of the relevant website(s).

#### Justification

The program will be very low powered yet will need a CPU that can “multitask” to an extent, as multiple things may need to be processed at any one given time. In addition to this, dual core processors are very cheap and are realistically below the standard in the modern day. In addition to dual cores being able to handle the program, the processor shouldn’t need to have a particularly high clock speed, as this is a very simplistic program which doesn’t and shouldn’t need much power to run successfully.

WIFI connectivity is a must as part of the project is a web scraper, which requires the internet in order to be able to access and scrape the relevant and neccesarry websites. Without WIFI connections, the information is liable to become out of date reasonably quickly, which would be against the entire point of the application in general.

### **Software**

The following are the operating system and software requirements for my application:

* The application will need to be run on any computer which can run python 3 based applications, such as Windows 7 PCs or MacOS X or later devices.

#### Justification

The only reason that I have the specifications listed above is that the operating system needs to be modern in order to be capable of running python 3-based solutions. If the software could not do this, then the program would not be able to run, as the entire application is going to be built in python 3. Windows 7 (if properly updated) can run python 3 applications, the same for MacOS X or above mac devices, these devices however come with Python 3 pre-installed, whereas Windows 7 needs python 3 to be manually installed onto the device. Linux has not been chosen as a listed operating system. This is as I can provide limited support for it and can’t run proper tests on it reliably through the development process.

### **additional libraries or utilities**

I will need to import the following modules / libraries:

* TKinter and multiple sub modules such as TK and font in order to create the UI and application body.
* Requests in order to access the API and draw information from it.
* Beautiful soup to parse data
* I may also need to import modules in order to access databases or to create web scrapers to gain additional information.
* Webbrowser module to interact with the default system browser.
* The OS library to work with various directories whilst maintaining the transferability of my code.
* Json is needed in order to interact with my json file which stores the location information needed for route planning.
* Folium will be used in order to plot a chart on a map to show the user how to get from point A to point B via road
* Geopy will be used for my geocoding purposes.

## **Computational methods:**

### Features that make the problem solvable via computational methods

The timetables need to be quickly and comprehensively checked and input into the application on a regular basis. This can’t feasibly be done manually by people, making it very suited to computational methods, specifically scraping, and parsing the information to then validate and input the information. As the program will also check the user’s locational data in comparison to local bus stops and the busses that go through them. Doing this manually would (and does) take lots more time, especially when compared to how quickly a computer could do this. This makes our problem solvable through computational means. In addition to these problems, the timetables are stored here, meaning the information remains constant. However, if this was not solved by a computer, the information may become warped over time as people may forget slight details or alter them in their mind. Even if written down, this information may become warped as people may not be able to read each other’s handwriting. Inputting this data to a computer and application means that it is consistent and easy to read and navigate.

### Problem decomposition

The problem can be broken down into (x) easily solvable, smaller issues, which can be tackled individually.

The first issue is finding the users location, as well as the locations of various bus stops. To solve this, we will need a list of bus stops in the surrounding area and what busses go through them. We also need the user to either manually input their location, or have it be found automatically.

After solving the first issue, we then need to consider a related issue – calculating the time it should take to get to said bus stop. We can solve this by calculating the walking distance to the bus stop and then dividing it by the average person’s walking pace (3.5 miles per hour). There may need to be conversion here between metres and miles, depending on user preference.

Another problem will be updating the information on the application. To do this, I will need to use as little internet as possible, to make the clients low bandwidth not as big of a problem. In order to solve this issue, I will download the necessary webpages once per week, and then scrape the downloaded webpages using python script daily (to avoid chance of error).

After I have gained the information and it has been parsed and scraped correctly, I need to put this into a format that is easily readable and accessible by users. I will use tables in tkinter to do this. To access these tables, the users will have a searchable list of the various bus routes and be able to click on these in order to access the relevant timetable, with the current information.

### Use of divide and conquer

I will use the divide and conquer design paradigm along with my decomposition in order to divide the already decomposed problem into smaller pairs of problems, in order for me to tackle them one by one. Divide and conquer splits large, difficult to solve problems into smaller, easy to solve problems. This will help me divide the problem up, allowing me to work on individual aspects. This will also allow me to compare progress much closer to the success criteria than if I didn’t divide these problems up.

### use of abstraction

Using abstraction, I will make the data more easily readable and understandable, whilst still being informative. I will remove any unnecessary information in order to make the process simpler and allow the users to find what they need faster than they are currently able to through existing software solutions.

The main way I will be using abstraction is for addresses. The addresses stored on the websites which I will be scraping from are quite long and complex, due to the incorporation of too much information. In order to combat this, I will simplify the data by only including the street name and postcode, the only relevant information that the user needs. By doing this, I am making the user experience for my application more pleasant, also improving the client and user workflow by making information more easily accessible to those who need it.

## **Success criteria:**

* Using a given data set – the app should be able to show the expected location of busses and whether or not they will arrive on time.
* Should be able to show who the driver of a bus is
* Should show how many passengers are on a bus
* Should be able to show when to leave to get to the bus on time
* Should be able to update bus times if the timetable in the database changes
* The app should be able to interact with a database in order to get reliable and relevant data.
* The app needs to be able to scrape the required data in order to provide the most recent and most relevant information.

# B. Design

<See H446-03 Project Advice Booklet for help and guidance of what must go here.>

## Wireframe and prototype

### General design choices

#### Colour scheme

For the applications colour scheme, I have chosen purple and yellow for the background and buttons, with the information being displayed on a white background, this is in order to mimic the currently existing software that is in place at the UTC, which also uses purple, yellow, and white. This choice means that software at the UTC is more consistent than it would have been if I used a generic colour scheme, consistency is helpful for students as well as making the school seem more professional.

### The navigation bar:

**The navigation bar:**

The top of the screen will contain a navigation bar which will enable users to navigate the app and find the information that they need as quickly and easily as possible more easily. This navigation bar will have three main areas. The first section will contain the application name and branding (brighter futures bus application). The second element is a button which takes users to a list of favourited routes, routes that are selected by users to be marked as favourites so that they are marked by a star and show up in the favourite routes section. These routes are also more likely to show up on the main screen of the application. The third and final element of the navigation bar is the route list button. This button will take you to a full list of bus routes, from both stagecoach and first bus companies. This list will be interactive and allow people to click on them in order to expand the route information and view a full route timetable. On screens other than the main/home screen, this will also contain a fourth element, a home icon which takes the user back to the main screen when needed.

### Main screen

This is the main/home screen of my application; it is where an overview of all the available information is given to the user in a concise manner. As seen below, there are three main components to the screen, the left portion of the screen, the right portion, and the navigation bar.

**The left portion:**

The left side of the screen contains a map of the local area. This map will be displayed using the google street view API, the bus stops will then be marked on the map to enable people to see where the local bus stops are in relation to their current location. This map (if possible) will be interactive, meaning you will be able to zoom in and out, move the camera around, and click on bus stops to find out more information.

**Right portion:**

The right portion of the screen contains a basic list of commonly used routes from around the UTC, these routes will be able to be clicked on, which will then take you to the later mentioned route information screen. This condensed list will contain routes that have been most recently (or frequently) viewed by users on the school terminal. This list will be displayed in the form of a table, with the bus number in the left column, and basic route information on the right column. This information includes the route name, as well as the company that runs this bus line. For example, “**Stagecoach busses –** Doncaster to Worksop”

Graphical user interface, text, application

Description automatically generated

### Route information screen

Below is the route information screen – this screen will be very commonly used as it contains the timetable for the route, along with any other necessary information as and when it is required (delays, cancellations, etc.). If no other information is needed, it will simply display the route number, name, and a timetable below. As with most screens, this will have a navigation bar which contains buttons for the home screen, favourite routes, and the overall route list. There will also be a star icon, which can be clicked to favourite/unfavourite a bus route, which will add or remove it from the favourite routes list and screen.

**The timetable:**

The timetable displayed will have a few main elements to it. In the leftmost column, there will be the stop name, this can be clicked to bring up the street on google maps. Next to this column, there will be the times of busses from the start until the end. Occasionally, when times are consistent for a given period of time, then this will display placeholder text to make the timetable smaller. For example, if busses run every 30 minutes, it will display “then every 30 minutes until” then the next column will be the time which it continues until. Three examples of this can be seen in the wireframe below.

**Table

Description automatically generated**

### Route list / favourite routes

The routes list and favourites screen are both extremely, looking identical except from the fact that the favourite routes screen will contain only favourited routes, whereas the route list contains both favourited and unfavourited routes.

**The table:**

The screen contains a table which contains 2 main elements for each item on the table. These two elements are the bus number and route information. The bus number is displayed in the leftmost column of the table and can be clicked in order to go to that routes route information screen. The other element of the table is basic route information – this is displayed next to the route number and contains the route name, the name of the company that runs the bus, and whether or not the route is favourited or not (as well as the ability to unfavourite said route on the favourite routes screen).

**The navigation bar:**

The navigation bars on these screens will have the standard two buttons that all iterations of the navigation bar have (favourite routes, as well as the route list button) in addition to a home screen button, which takes users back to the home screen. This is similar to all navigation bars which are not the home screen. The button to take the user back to this page will function as a way to get back to the top of the list without having to scroll through the entire list, it will also cause the page to reload which will ensure all information is as up to date as possible.

Graphical user interface, text, application

Description automatically generatedGraphical user interface, text, application

Description automatically generated

### Client feedback on designs

In order to ensure that they were up to the clients’ standard, I conducted a survey asking the client for feedback and potential changes to make to my designs in order to improve them from a user standpoint. I will then implement some of my changes to my designs and then show my progress and how this affects the overall product and completion of the various success criteria.

#### Main screen feedback and changes:

For the main screen, the client wished for some base designs to be changed, this was to do mainly with fonts and text styling. They want the titles for the tables to be both bold and underlined, rather than just bold. This is an easy implementation which we agree with. They would also like there to be an example map rather than just placeholder text, something else which we can easily implement and agree that it would help the stakeholders get a better feel for what the end product will actually look like.

##### The new main screen:

**Graphical user interface, application

Description automatically generated**

#### Route information screen feedback and changes:

The route information screen had one major complaint, and that was that (although in the example there was no additional information) the client felt that the section that would show additional information should still be included, then saying that none is required to the user. It will also notify the user that any additional information such as delays will be displayed there as soon as possible (this will likely require a page load to display).

##### The new route information screen:

Table

Description automatically generated

#### Route list / favourite routes screens feedback and changes:

Whilst the client was happy with the favourite routes screen, however, they had one main problem and piece of feedback that they felt would greatly improve user experience on the routes list screen. This is that the routes list screen does not show whether or not a route has been favourited or not. In order to change this and accommodate the clients’ needs, I will add a feature that looks similar to the stars that are on the favourite routes screen, enabling users to quickly see whether or not a route is favourited and then change the status if necessary. They would also like the table headings to be underlined in order to match the formatting of the home screen.

##### The new route list / favourite route screens:

Graphical user interface, application

Description automatically generated

## Pseudocode

### Main app pseudocode

The main application will draw multiple files into one main file in order to make the process of development more streamline. However, for my pseudocode, I have put very simplified versions of the code into one file as the more simplistic code can be easily fit into one main file and there is no need to have multiple pseudocode files – only one for the main app, and one for the web scraper.

##### Main application

Text

Description automatically generated

The above pseudocode is a rough interpretation of what my main screens code will look like – without the main computational elements to it. The pseudocode can be split up into 5 main sections; 3 functions, one section declaring the screen, and one section calling the functions (which does not need to be explained).

###### Section 1

Text

Description automatically generated

This above code is representative of the basic way to create my screen using TKinter in python, though the settings are not fully fledged out and some things need to be edited in the final draft. First, a variable called ‘screen’ is created, using the tkinter function “Tk” to give it the correct object type and the necessary functions required in order to be able to sufficiently configure the screen. Then an overall font is decided for the application – for now I am using Helvetica, which is stored as a string to be input into any and all text boxes at a later date. I decided to use a global variable here as it means if I decide to change the font at a later date then I only have to change one variable, not every mention of the used font in code.

## Example code – proof of concept

### Map in tkinter

For my project, I need to be able to display a map of local bus routes as well as having this map be interactable with features such as zooming in and moving the camera around. In order to do this, I have used a python library specifically built for use with Tkinter in order to create interactive maps. This library is called TkinterMapView and enables you to get maps from the internet from a variety of sources. The code below is proof of concept and shows how I am able to create and customise a map and a border for the map.

##### Code overview

My code contains 4 key parts: the declaration of the function, the border creation and placement, the map settings, and the map placement.

**Declaration of the function:**

The entirety of the map generation is wrapped in a function, this means that I am able to call the function from various files (which I will need to as my program progresses) meaning that I won’t have to rewrite code and will ultimately save time and maintain consistency. The function currently has one parameter passed into it called “Location” which (when the function is called) will be replaced with a variable created earlier in my code called “HightlightedLocation” which contains a string which contains the required location where the marker needs to be placed. As this is stored as a variable, it means that you can change the location and update it in real time as and when needed. This can be modified in order to place markers at each stop when clicked by the user in the route information section.

**Border creation and placement:**

The second key aspect of the below code is the border creation that controls the border for my map. This code sets colour, width, height, and position of the border. The code is contained within lines 2,3, and 4 in the below code and each line does the following:

1. Creates an empty label with a yellow background.
2. Sizes the label to be just larger than the map, meaning the edges of the label can be seen, even though the label itself will be behind the map.
3. Places the border in the same location as the map, but behind the map.

With this code, I can create a border for the map, I could also in theory later adapt the code in order to have a border around any item in the program, not just the map. In the implementation stage, this will likely be its own function, with the ability to resize and relocate the border when calling the function, this means that the code can be easily reused and adapted for other purposes.

**Map settings:**

The second portion of the function is the map settings. This section is where each aspect of the map is defined such as its source, height, width, and where it will be. This part of the code will define the look of the map and will be able to be modified to create maps of any size, with a variety of settings being able to change both the look and feel of the map. This is where the value of “location” in the function declaration is used, and is set to the “highlightedLocation” variable, which in this example code is Doncaster. As it is input through the function declaration, any location can be passed into the function, meaning that I could bring up a map from anywhere around the world. This means that (although the app will only contain Doncaster-based busses to begin with) the application will be expandable in order to create apps for other locations and schools. The code itself takes up 3 lines, each configuring various aspects of the map and the information which it displays:

1. This uses the TkinterMapView library to create a window on the “screen” screen, with a width of 400, height of 600, and a corner radius of 0 (meaning it does not have rounded corners) and saves this information in the variable map\_widget.
2. This takes the information from the map\_widget variable and sets the server for the map (meaning the place it will draw the map imagery from), in this case, I am using Google maps to draw my data from, as it is the most commonly used service, and will thus be recognisable. It also defines the max zoom for the map, I found that 22 was a suitable value in order to obtain all necessary information from the map.
3. This final line utilises the earlier mentioned passed-through variable “Location” and uses this in order to set the focal location for the map. The focal location will be the location that is (by default) displayed at the centre of the map. Then, I have enabled the marker, which will add a marker to the location I have specified. This feature will also likely be how I add the bus stops to the bus routes, in order to make clear to the user where each stop is.

**Map placement:**

The map placement portion of the function is only one line, yet is arguably one of the most important. This section places down the map onto the screen, in order for the user to be able to see and interact with the map. There are three variables that are used by the program, relx, rely, and anchor. Relx is a variable that decides the positioning of the map based on the relative x position of the window (specified as a decimal between 0 and 1). Rely is a similar variable to relx, yet instead of deciding where on the x axis it is placed, it determines the location on the y axis (yet again relative between 0 and 1). The final variable (anchor) is there to determine which part of the map is the point that is moved along the x and y axis, with CENTER being the direct centre of the image meaning that the centre of the image will be (in this case) on 0.2 and 0.53 on the x and y axis respectively.

Graphical user interface, text, website

Description automatically generated

Map

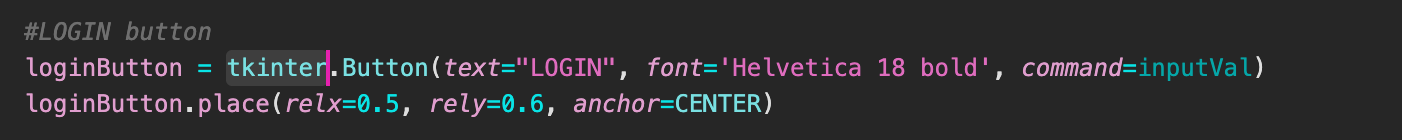
Description automatically generated

## Systems diagram

# C. Developing the coded solution (“The development story”)

## Phase 1 Work on the login screen

### User interface

Graphical user interface

Description automatically generated

Shown above is the UI for my login screen, with the code above showing the code needed for the button creation, functionality, and placement. This page was the simplest to make as it was simply 2 entry fields, 2 labels, and a button, all positioned in the center with equal spacing between. This was the first draft of my login page, which has since been altered to look as follows:

Graphical user interface, application

Description automatically generated

I found that this design is much better overall as it is smaller and takes up less of the screen, this means that there is less overall wasted space on the screen. However, to combat this, I then had to reposition the window to be in the center of the screen. This was done through the geometry configuration function in tkinter.

loginScreen.geometry("250x400+560+220")

above is the line of code used to make the window 250 by 400 pixels, then position it 560 pixels to the right and 220 pixels down.

*#LOGIN button*

loginButton = tkinter.Button(*text*="LOGIN", *background*="yellow", *activebackground*="#9b870c", *font*='Helvetica 18 bold', *command*=inputVal)

loginButton.place(*relx*=0.5, *rely*=0.7, *anchor*=CENTER)

This above code is setting the parameters for and placing the log in button. The command run is inputVal which will run through the inputs and compare them against what is required.

*#configures text boxes and their headings*

userLabel = tkinter.Label(*text*="USERNAME", *bg*="#8D16D8", *font*='Helvetica 18 bold')

passLabel = tkinter.Label(*text*="PASSWORD", *bg*="#8D16D8", *font*='Helvetica 18 bold')

userBox = tkinter.Entry()

passBox = tkinter.Entry(*show*="\*")

This code configures the 2 labels and the 2 entry boxes needed, specifying their backgrounds and specifying that the password text box will show asterisks instead of whatever text is in, for privacy reasons.

*#places text boxes and their headings*

userLabel.place(*relx*=0.5, *rely*=0.3, *anchor*=CENTER)

userBox.place(*relx*=0.5, *rely*=0.4, *anchor*=CENTER)

passLabel.place(*relx*=0.5, *rely*=0.5, *anchor*=CENTER)

passBox.place(*relx*=0.5, *rely*=0.6, *anchor*=CENTER)

Here is the code which places all of the boxes, the relative x is always 0.5, meaning that all of the elements will be centred, there is then a 10% gap between each item, meaning that it is all evenly spaced throughout the screen, with the central item being in the center of the screen.

### Functionality

To implement functionality, I decided that the username and password should be encrypted using SHA256 encryption. In order to do this, I had to research how to implement the encryption using the website below:

<https://datagy.io/python-sha256/>

This website shows the base of how to use the hashlib module for encryption. I then used this as a base in order to encrypt my data and ensure that the data is secure and not easily accessible.

The login screens functionality is wrapped up into one central function called inputVal (short for input validation), it takes no extra parameters. This function takes the hashed values of the inputs for each text field and compares them to the stored username and password in order to validate login and determine the users’ permissions before the application is loaded (i.e.: administrative permissions).

userPassDict = {

"8c6976e5b5410415bde908bd4dee15dfb167a9c873fc4bb8a81f6f2ab448a918":"5994471abb01112afcc18159f6cc74b4f511b99806da59b3caf5a9c173cacfc5",

"264c8c381bf16c982a4e59b0dd4c6f7808c51a05f64c35db42cc78a2a72875bb":"264c8c381bf16c982a4e59b0dd4c6f7808c51a05f64c35db42cc78a2a72875bb"

}

These are the stored, encrypted values of the 2 example usernames and passwords in the program – there is an account for students and an account for the admin.

password=hashlib.sha256((passBox.get()).encode()).hexdigest()

This code takes the input of the password input box, encodes the data, and then runs it through SHA256 encryption, which can later be applied in comparison to the stored passwords. Similarly, this has been used in order to encrypt and compare the username.

At one point, I wanted to make all buttons in the UI yellow as shown below: Graphical user interface

Description automatically generated

Despite the colouration working correctly, I later ran into implementation errors as a different module is needed for coloured buttons on macos. This meant that each coloured button could only have one command attached to it at any given time, meaning that some buttons needed to be white instead of yellow. Overall, I decided to simply remain using plain white buttons as it meant the UI was much more consistent throughout with the design language matching on each page whilst also still being functional.

*#checks for the username in the dictionary*

*if* username in userPassDict:

*if* userPassDict[username] == password:

*if* username=="8c6976e5b5410415bde908bd4dee15dfb167a9c873fc4bb8a81f6f2ab448a918":

mainApp.admin=True

print("Admin enabled")

loginScreen.destroy()

mainApp.\_\_main\_\_()

*else*:

*#removes 1 try, displays appropriate message*

loginTries-=1

invalidMessageText=("invalid username or password, you have", loginTries, "attempts remaining")

invalidMessage = tkinter.Label(*text*=invalidMessageText, *bg*='#8D16D8', *font*='Helvetica 18 bold', *fg*="red")

invalidMessage.place(*relx*=0.5, *rely*=0.35, *anchor*=CENTER)

*#if there are no tries remaining, the application closes*

*if* loginTries <=0:

loginScreen.quit()

*else*:

*#removes 1 try, displays appropriate message*

loginTries-=1

invalidMessageText=("invalid username or password, you have", str(loginTries), "attempts remaining")

*#refreshes and replaces invalid message*

invalidMessage = tkinter.Label(*text*=invalidMessageText, *bg*='#8D16D8', *font*='Helvetica 18 bold', *fg*="red")

invalidMessage.place(*relx*=0.5, *rely*=0.35, *anchor*=CENTER)

*if* loginTries <=0:

*#if there are no tries remaining, the application closes*

loginScreen.quit()

The above code compares the username and password and stored values to ensure that the correct usernames and passwords have been input. It checks the inputs 3 times and if it is incorrect each time, the application closes. Each time that the input is incorrect, it also displays an error message, along with the number of attempts remaining. On a successful input, the main screen is loaded. As well as this, if an admin account was used, the admin version of the page is loaded.

## Phase 2 Work on the main screen

### User interface

### back end

## Phase 3 Work on the Web scraper

### Beginning my research

For my web scraper, I first had to look at how to make it – which began with researching various python modules. The module that I had heard the most about from other students and friends was beautiful soup. My research began with reading some documentation and example code to start to understand the limitations and benefits of using beautiful soup. I began on the following website:

<https://www.crummy.com/software/BeautifulSoup/bs4/doc/>

By reading this, I decided that beautiful soup was going to fit well in my project and work for what I needed, the basic collection of timetable data from a given website to later store and display within my application.

After this, I looked through some example code in order to better understand how to implement the various elements of the module and what data different things gathered. Below is said example code:

*import* requests

*from* bs4 *import* BeautifulSoup

*import* csv

URL = "http://www.values.com/inspirational-quotes"

r = requests.get(URL)

soup = BeautifulSoup(r.content, 'html5lib')

quotes=[] *# a list to store quotes*

table = soup.find('div', *attrs* = {'id':'all\_quotes'})

*for* row *in* table.findAll('div', *attrs* = {'class':'col-6 col-lg-3 text-center margin-30px-bottom sm-margin-30px-top'}):

quote = {}

quote['theme'] = row.h5.text

quote['url'] = row.a['href']

quote['img'] = row.img['src']

quote['lines'] = row.img['alt'].split(" #")[0]

quote['author'] = row.img['alt'].split(" #")[1]

quotes.append(quote)

filename = 'inspirational\_quotes.csv'

*with* open(filename, 'w', *newline*='') *as* f:

w = csv.DictWriter(f,['theme','url','img','lines','author'])

w.writeheader()

*for* quote *in* quotes:

w.writerow(quote)

From the above code, I can understand that you need to first define a URL, parse the content on the site, and then you can begin searching through the data in order to retrieve and store various elements. After I believed I could understand the various sections of this code, I began working on my programming in order to implement what I had learned in a context that would help me.

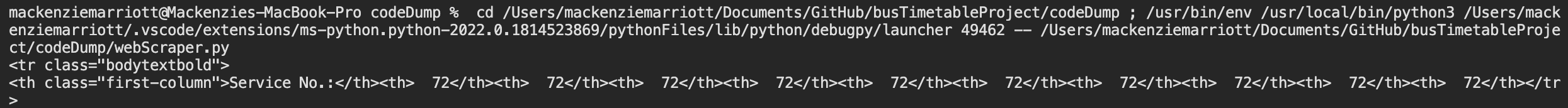
### Beginning my implementation

To begin with, I attempted to isolate an individual element from within my code.

Text

Description automatically generated

I began by writing the above code, which I believed would collect an individual element. However, instead The output was the entirety of the first row, not just the first element.



This is as I used both the wrong tag and the wrong class to collect data. Instead of searching in bodytextbold for the tr content, I needed to search for all td content, as this is where the times and stop locations are stored on the first website.

Text

Description automatically generated

The above is modified code which prints out every all text that is contained within a td tag. Below is the first result of this code:

Text

Description automatically generated with medium confidence

As seen above, this code will output all table data in the above format, which means that now I will be able to go through and collect the data where needed, storing it in a text file as it is collected. At first, I was going to store it as a csv file, but as I got more accustomed to implementing data into a tkinter treeview table, I believed that the information I collected would be better stored within a text file, as I could then simply go through line by line.

After implementing the storage of data, my code looked as follows:

Text

Description automatically generated

In this code, I had not yet determined a file naming system for the data, so I simply decided to let the user create a new text file to store the data in, later I would go to change this to a naming system that is created automatically based on some user input in the application that followed certain conventions. Once I got this code working and reasonably efficient, I began working on code that would find the dimensions of the given table in order to later format the table correctly.

Formatting the table would rely on 3 key aspects, the table width, height, and total number of values that need to be output. Getting the total amount of values was simple, as I would only need to rely on finding the length of the text file with all the elements in. however, getting the other dimensions for the timetable would prove to be more difficult.

Text

Description automatically generated

The above screenshot shows how I would go through the list, printing out a value from 0 to the width of the table, for the number of rows in the table, effectively creating a repeating grid in a 2-dimensional array. This took another earlier function into account, which found the total number of tables on the website – though in my later implementation I would find this was not needed. After this, I tried to add the relevant data to a list formatted like a timetable to be able to print out this information in an easy-to-read manner. The output looked like this:

Chart

Description automatically generated with medium confidence

As you can see, though the correct amount of data is shown, the data displayed is almost completely wrong, with it simply repeating the same value throughout, which was not the intent. Eventually I fixed the code up and got the following as my end product:

### The end product

*from* bs4 *import* BeautifulSoup

*import* requests

*import* os

*import* webbrowser

*#table dimensions*

tableDict=[]

totalTableCount=0

widthPerTable=0

totalHeight = 0

heightPerTable = 0

*#array to store dimensions*

dimensionArray=[]

*#length of a given text file with stop time data*

fileLength = 0

*#declarations so values can be used later as globals in other functions*

completeFileName = ""

soup = None

*#main web scraping function - finds all needed data and writes to text file*

def webScrape(*routeURL*, *routeName*):

global completeFileName

global soup

*#gets and requests data from the url*

url = *routeURL*

urlRequest = requests.get(url)

soup = BeautifulSoup(urlRequest.content, 'html5lib')

*#configures file and path naming*

timeFileName = *routeName*+".txt"

dataFileName = *routeName*+"Data.txt"

pathName = (os.getcwd()+'/timetableFiles/txtFiles')

completeFileName = os.path.join(pathName, timeFileName)

*#finds all elements with tag 'td' and writes them into the text file*

tdValue = 0

tdContainer = soup.findAll('td')

openFile=open(completeFileName, 'w')

*# goes through each td in the list and writes each value to a new line on the text file*

*for* td *in* tdContainer:

openFile.write((tdContainer[tdValue].text+'\n'))

tdValue+=1

openFile.close()

totalTables()

tableHeight()

tableWidth()

lineCount()

tableData = [str(widthPerTable)+"\n", str(heightPerTable)+"\n", str(fileLength)]

tableDataFile = (os.path.join(pathName, dataFileName))

openFile = open(tableDataFile, 'w')

openFile.writelines(tableData)

openFile.close()

*#function to find the height of a first bus table - not including notes row*

def tableHeight():

global totalHeight

global heightPerTable

tableHeightContainer = soup.findAll('tr', *attrs*={"class": "table\_alt"})

*for* tr *in* tableHeightContainer:

totalHeight+=1

tableHeightContainer = soup.findAll('tr', *attrs*={"class" : "table\_alt"})

*for* tr *in* tableHeightContainer:

totalHeight+=1

*#calculates the average height of each table*

firstColumns = soup.findAll("td", *attrs*={"class":"first-column"})

*for* td *in* firstColumns:

heightPerTable+=1

heightPerTable=int(heightPerTable/totalTableCount)

*#counts the total number of columns in each table*

def tableWidth():

global widthPerTable

total=0

firstRow = soup.find("span", *attrs*={"class":"magenta\_on"})

*for* i *in* range (1, (len(firstRow)+1)):

total+=1

total+=(totalTableCount)

widthPerTable = total+3

*#finds total amount of tables on a page - not used in end product (not needed)*

def totalTables():

global totalTableCount

totalTableCount = 0

tableCount = soup.findAll("tr", *attrs*={"class":"bodytextbold"})

*for* tr *in* tableCount:

totalTableCount+=1

*#creates dictionary of dimensions for all tables on a page - not used in end product (couldnt get accurate values)*

def fileInput():

global tableDict

global totalTableCount

global heightPerTable

tableDict = []

lineTracker = 0

file = open(completeFileName)

fileContent = file.readlines()

*#makes array 3D for amount of tables*

*for* i *in* range(0,totalTableCount):

tableDict.append([])

*#makes the smaller arrays, one for each table line*

*for* i *in* range(0, totalTableCount):

*for* x *in* range(0,heightPerTable):

tableDict[i].append([])

*for* b *in* range(0,11):

tableDict[i][x].append(fileContent[lineTracker])

*while* lineTracker != fileLength-1:

lineTracker+=1

*break*

*#counts total lines of data in a given text file*

def lineCount():

global fileLength

*with* open(completeFileName, 'r') *as* openFile:

fileLength = len(openFile.readlines())

*#makes empty arrays to function as dimensions for each table - not used in end product*

def tableDimensions():

global dimensionArray

global heightPerTable

*for* i *in* range(0, totalTableCount):

dimensionArray.append([])

dimensionArray[i].append(widthPerTable)

dimensionArray[i].append(heightPerTable)

print(dimensionArray)

*#used instaed of scraping the web to ensure low WIFI usage - this would cut bandwidth as its only accessed if/when needed*

*#table on the website is also extremely difficult to read through - making issues during development and data hard to display*

def cancellations():

*#takes user to website in order to*

webbrowser.open\_new("https://www.firstbus.co.uk/doncaster/plan-journey/timetables/journey-cancellations")

'''

#finds out overall data

lineCount()

totalTables()

#calculates dimensions of timetables

tableHeight()

tableWidth()

tableDimensions()

#inputs timetable data to file and 3D array

fileInput()

'''

*#^^functions were called to test - no longer needed to be called, messes with the program when importing module*

'''

webScrape("https://www.firstbus.co.uk/doncaster/plan-journey/timetables/?day=1&source\_id=2&service=72%2F72x%2F73%2F73x&routeid=23936179&operator=34&source=sp", "73")

totalTables()

tableWidth()

tableHeight()

'''

*#webScrape("https://www.firstbus.co.uk/doncaster/plan-journey/timetables/?day=1&source\_id=2&service=87%2F87a%2F87b&routeid=23936181&operator=34&source=sp", "87")*

*#webScraper was running automatically with 73 route, slowing system down*

*#print(widthPerTable)*

*#print(heightPerTable)*

*#^^ test to show working web scraping*

The code I have above includes all tests commented out towards the end. This code is also implemented with the rest of my code, allowing it to be properly called when needed and storing the information. The main downside here is that not all of the code is needed or used, with some functions being completely inconsequential to the end product.

## phase 4 Distance calculator

### geocoding

In order to calculate the distance between 2 given points, I need to geocode the data. This is as code uses coordinates, rather than street names, in order to calculate distances by placing coordinates on a map and then following the quickest road from the start point to the destination. In order to geocode my data, I used the Nominatim portion of the geopy.geocoders module.

#### research

As I had never done any geocoding in any programming language, my first port of call was to research what exactly geocoding is, and then how to go on and do it in python. For a lot of this research, as well as related research, I used a website called medium. They have a great child website called towards data science, which effectively discusses various principles in data science and how to apply them in real world scenarios.

The specific page I used for my geocoding:

<https://towardsdatascience.com/geocode-with-python-161ec1e62b89>

This page discusses the geopy module and how to use and implement it in order to correctly obtain coordinates from a given position. This site uses the nominatim module as the example as it is built on top of open route services API, which is very commonly used as a free API alternative to similar Google services, whilst still maintaining almost all functionality. The site also goes through plotting various points onto a map or graph, which (while useful if I were to expand the program) I do not need for the base functionality of my program, though this may prove useful in future projects.

#### Implementation

The implementation for geocoding is reasonably simple and uses only 3 built in functions (for what I need).

* Nominatim() – a function which sets the user agent for the nominatim geocoder.
* Geocode() – a function which gets the latitude and longitude coordinates for a given location when passed as a string.
* .latitude and .longitude – functions which specify the latitude or longitude of a given set of coordinates.

As seen below, they are simple to implement and to get the coordinates as individual variables it only takes 3 simple lines of code. Errors only start to appear if the passed location is not found or formatted correctly. This error does come up at a later date as some of the stop names are not formatted in a way which is able to be geocoded by the nominatim geolocation service.

*#specifies which geolocation service to use (nominatim)*

geolocator = Nominatim(*user\_agent*="alevel-application")

*#configures starting coordinates*

coords = geolocator.geocode("College Road Doncaster")

coordLat = coords.latitude

coordLong = coords.longitude

*#configures ending coordinates*

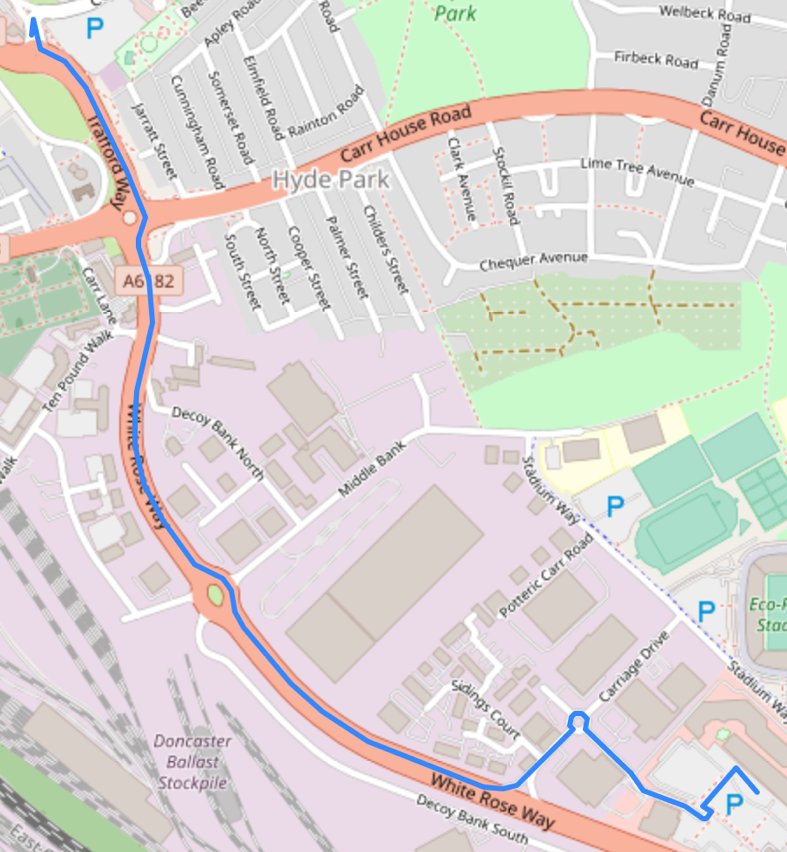
endCoords = geolocator.geocode(*endLocation*)

endCoordLat = endCoords.latitude

endCoordLong = endCoords.longitude

The above code specifies the service that I will use (nominatim) in order to geocode. In addition to this, I have to specify the “user agent” – which is effectively a key – that I am using. Then I geocode the coordinates from the school (Doncaster UTC on college road Doncaster) and store the latitude and longitude coordinates in order to be used later in comparison. After this, I geocode a given end location, which is yet again stored for the distance and route to be calculated and displayed.

All of my geocoding code is wrapped in a single function named “coordGet” with a variable passed in that I have referred to as “endLocation” which will be a string passed in representing the location name of the destination (in most cases this will be the target bus stop specified by the user.



This is what the geocoding is used for and the product it creates when it has run, the coordinates at each end of the blue line are found through inputting data, although the line is drawn and calculated separately using other modules such as open route service and the folium module.

### open route service

Open route service is a free alternative to services such as the google maps suite of APIs – allowing for very similar functionality to what Google offer, whilst also keeping development costs down – though it does have some limitations such as a lack of street view and a less user-familiar map UI, though I believe the functionality of the module and the trade-offs it comes with are worth the benefit in terms of development costs.

#### Research

For my research I used a different page on the “medium” website that I used for the geopy module. Though I used open route service in order to display my map on other screens in my application, here I needed to be able to interact more concisely with the map and save it as some form of file. This site helped massively, showing me how to properly utilise the open route service, as well as being able to save the product of my code in a html file – which I am then able to open in a web browser using the webbrowser module whilst also not needing an internet connection.

### folium

# D. Evaluation

<See H446-03 Project Advice Booklet for help and guidance of what must go here.>

# Project Appendixes

## Complete code listing

### MainApp.py

*import* tkinter

*from* tkinter *import* \*

*import* tkinter.font *as* font

*import* webbrowser

*import* mapGenerationCode

*import* configurePage

*import* tables

*import* os

*import* distanceCalculator

*import* routeScreen

*import* time

myFont='Helvetica 18 bold'

admin = False

def \_\_main\_\_():

global admin

*#creates an empty deafault tkinter screen*

mainScreen = tkinter.Tk()

def configMap():

distanceCalculator.coordGet("Bawtry Road")

distanceCalculator.routing()

fileName = 'file:///'+os.getcwd()+'/'+'map.html'

webbrowser.open\_new\_tab(fileName)

*#calls functions to configure the main page layout*

configurePage.configure(mainScreen)

configurePage.configHeader(mainScreen, "Main Screen")

*#calls both necessary functions to place the map onto the main screen*

mapGenerationCode.mapSettings(400, 600, "doncaster")

mapGenerationCode.generate(mainScreen)

*#code to refresh the map*

def resetMap():

*#calls both necessary functions to place the map onto the main screen*

mapGenerationCode.mapSettings(400, 600, "doncaster")

mapGenerationCode.generate(mainScreen)

mapButton = tkinter.Button(mainScreen, *font*=myFont, *text*="RESET POSITION", *command*=lambda:[resetMap()])

mapButton.place(*relx*=0.2, *rely*=0.9, *anchor*=N)

*#calls functions to generate the correct table*

tables.tableSettings(700, 750, 0.65, 0.53)

tables.generateTable(mainScreen)

'''

testButton = tkinter.Button(text="test", command=routeScreen.\_\_main\_\_).place(relx=0.5, rely=0.5)

^ test button to test connection to route screen, and changing of displayed information

'''

*if* admin == True:

configurePage.configSettingsButton(mainScreen)

'''

#test button

test = tkinter.Button(text="test", command=configMap)

test.place(relx=0.5, rely=0.05, anchor=CENTER)

'''

mainScreen.mainloop()

### Loginsystem.py

*import* tkinter

*from* tkinter *import* \*

*import* tkinter.font

*import* hashlib

*import* mainApp

*import* configurePage

*import* tkmacosx

*#^^tkmacosx is unused - buttons do not work as expected*

loginScreen = Tk()

*#dictionary to store the hashed usernames and passwords*

userPassDict = {

"8c6976e5b5410415bde908bd4dee15dfb167a9c873fc4bb8a81f6f2ab448a918":"5994471abb01112afcc18159f6cc74b4f511b99806da59b3caf5a9c173cacfc5",

"264c8c381bf16c982a4e59b0dd4c6f7808c51a05f64c35db42cc78a2a72875bb":"264c8c381bf16c982a4e59b0dd4c6f7808c51a05f64c35db42cc78a2a72875bb"

}

*#variables for login validation*

loginTries = 3

invalidMessageText = ("invalid username or password, you have", loginTries, "attempts remaining")

invalidMessage = tkinter.Label(*text*=invalidMessageText, *bg*='#8D16D8', *font*='Helvetica 18 bold', *fg*="red")

*#all code to validate user input - including hashing values*

def inputVal():

*#imports necessary variables*

global userPassDict

global loginTries

global invalidMessageText

*#gets user input and stores as variables*

*#converts username input to hashed value*

username=hashlib.sha256((userBox.get()).encode()).hexdigest()

*#converts password input to hashed value*

password=hashlib.sha256((passBox.get()).encode()).hexdigest()

*#checks for the username in the dictionary*

*if* username in userPassDict:

*if* userPassDict[username] == password:

*if* username=="8c6976e5b5410415bde908bd4dee15dfb167a9c873fc4bb8a81f6f2ab448a918":

mainApp.admin=True

print("Admin enabled")

loginScreen.destroy()

mainApp.\_\_main\_\_()

*else*:

*#removes 1 try, displays appropriate message*

loginTries-=1

invalidMessageText=("invalid username or password, you have", loginTries, "attempts remaining")

invalidMessage = tkinter.Label(*text*=invalidMessageText, *bg*='#8D16D8', *font*='Helvetica 18 bold', *fg*="red")

invalidMessage.place(*relx*=0.5, *rely*=0.35, *anchor*=CENTER)

*#if there are no tries remaining, the application closes*

*if* loginTries <=0:

loginScreen.quit()

*else*:

*#removes 1 try, displays appropriate message*

loginTries-=1

invalidMessageText=("invalid username or password, you have", str(loginTries), "attempts remaining")

*#refreshes and replaces invalid message*

invalidMessage = tkinter.Label(*text*=invalidMessageText, *bg*='#8D16D8', *font*='Helvetica 18 bold', *fg*="red")

invalidMessage.place(*relx*=0.5, *rely*=0.35, *anchor*=CENTER)

*if* loginTries <=0:

*#if there are no tries remaining, the application closes*

loginScreen.quit()

*#configures text boxes and their headings*

userLabel = tkinter.Label(*text*="USERNAME", *bg*="#8D16D8", *font*='Helvetica 18 bold')

passLabel = tkinter.Label(*text*="PASSWORD", *bg*="#8D16D8", *font*='Helvetica 18 bold')

userBox = tkinter.Entry()

passBox = tkinter.Entry(*show*="\*")

*#places text boxes and their headings*

userLabel.place(*relx*=0.5, *rely*=0.3, *anchor*=CENTER)

userBox.place(*relx*=0.5, *rely*=0.4, *anchor*=CENTER)

passLabel.place(*relx*=0.5, *rely*=0.5, *anchor*=CENTER)

passBox.place(*relx*=0.5, *rely*=0.6, *anchor*=CENTER)

*#LOGIN button*

loginButton = tkinter.Button(*text*="LOGIN", *background*="yellow", *activebackground*="#9b870c", *font*='Helvetica 18 bold', *command*=inputVal)

loginButton.place(*relx*=0.5, *rely*=0.7, *anchor*=CENTER)

'''

loginButton = Button(text="LOGIN", background="yellow", activebackground="#9b870c", font='Helvetica 18 bold', command=inputVal)

loginButton.place(relx=0.5, rely=0.7, anchor=CENTER)

'''

configurePage.configure(loginScreen)

loginScreen.geometry("250x400+560+220")

*#^^places in centre of*

'''

loginScreen.geometry("250x400")

^^ places in top left, not centre

'''

loginScreen.mainloop()

### distanceCalculator.py

*#routing modules*

*import* openrouteservice

*from* openrouteservice *import* convert

*import* json

*import* folium

*#geocode modules*

*from* geopy.geocoders *import* Nominatim

*#variables to store locational data*

*#starting long/lat*

coordLong = ""

coordLat = ""

*#ending long/lat*

endCoordLong = ""

endCoordLat = ""

*#takes address and converts to latitude and longitude to use in distance calculation*

def coordGet(*endLocation*):

*#starting coord lat/long*

global coordLat

global coordLong

*#ending coord lat/long*

global endCoordLat

global endCoordLong

*#specifies which geolocation service to use (nominatim)*

geolocator = Nominatim(*user\_agent*="alevel-application")

*#configures starting coordinates*

coords = geolocator.geocode("College Road Doncaster")

coordLat = coords.latitude

coordLong = coords.longitude

*#configures ending coordinates*

endCoords = geolocator.geocode(*endLocation*)

endCoordLat = endCoords.latitude

endCoordLong = endCoords.longitude

*#configures routing so that distance/time data can be taken*

def routing():

*#starting point coords*

global coordLong

global coordLat

*#end point coords*

global endCoordLat

global endCoordLong

*#sets a list to the corresponding values for coordinates*

coords = ((coordLong, coordLat), (endCoordLong,endCoordLat))

client = openrouteservice.Client(*key*='5b3ce3597851110001cf62488cb4c056c9294a4ba1a5a37393b69413')

*#calls from the API*

res = client.directions(coords)

*#tested json response*

*with*(open('test.json','+w')) *as* f:

f.write(json.dumps(res,*indent*=4, *sort\_keys*=True))

*#takes geometry from json, decodes it for use in map*

geometry = client.directions(coords)['routes'][0]['geometry']

decoded = convert.decode\_polyline(geometry)

*#configures map settings*

m = folium.Map(*location*=[coordLat, coordLong],*zoom\_start*=15, *control\_scale*=True,*tiles*="openstreetmap")

*#creates and saves map*

folium.GeoJson(decoded).add\_to(m)

m.save('map.html')

'''

coordGet("Thorne Road Doncaster")

routing()

^^order and example of how to run functions

'''

### configurepage.py

*import* tkinter

*from* tkinter *import* \*

*from* tkinter *import* font

*import* routes

*import* mainApp

*import* webScraper

myFont = "Helvetica 18 bold"

*#configures the base settings for every page on the app*

def configure(*pageName*):

*pageName*.config(*bg*='#8D16D8')

*pageName*.title("Brighter Futures Bus Application")

*pageName*.geometry("1920x1080")

*#configuers the screens header and places it*

def configHeader(*pageName*, *pageTitle*):

global myFont

*#configures the size and shape of the header, customisable for any page*

headerLabel = tkinter.Label(*pageName*, *text*=*pageTitle*)

headerLabel.config(*font*=myFont,*width*=150, *height*=2)

headerLabel.place(*relx*=0.5, *rely*=0, *anchor*=N)

*#configures a button to take the user back to the home screen*

homeButton=tkinter.Button(*pageName*, *text*="HOME", *background*="yellow", *activebackground*="#9b870c", *font*=myFont, *command*=lambda:[*pageName*.destroy(), mainApp.\_\_main\_\_()])

homeButton.place(*relx*=0.75, *rely*=0.025, *anchor*=CENTER)

*#configuration of the routes button - a button that takes the user to a list of all routes*

routesButton=tkinter.Button(*pageName*, *text*="ROUTES", *background*="yellow", *activebackground*="#9b870c", *font*=myFont, *command*=lambda:[*pageName*.destroy(), routes.routesPage()])

routesButton.place(*relx*=0.65, *rely*=0.025, *anchor*=CENTER)

*#configuration of the cancellations button - does not unclude .destroy as it opens in the browser, not tkinter*

cancellationsButton = tkinter.Button(*pageName*, *text*="CANCELLATION", *background*="yellow", *activebackground*="#9b870c", *font*=myFont, *command*=lambda:[webScraper.cancellations()])

cancellationsButton.place(*relx*=0.85, *rely*=0.025, *anchor*=CENTER)

def configSettingsButton(*pageName*):

settingsButton = tkinter.Button(*pageName*, *text*="SETTINGS", *background*="yellow", *activebackground*="#9b870c", *font*=myFont, *command*=lambda:[settingsTopLevel()])

settingsButton.place(*relx*=0.95, *rely*=0.025, *anchor*=CENTER)

def settingsTopLevel():

settingsPage = tkinter.Toplevel()

settingsPage.title("Brighter Futures Bus Application")

settingsPage.config(*bg*='#8D16D8')

settingsPage.geometry("960x540+250+150")

headerLabel = tkinter.Label(settingsPage, *text*="SETTINGS")

headerLabel.config(*font*=myFont,*width*=150, *height*=2)

headerLabel.place(*relx*=0.5, *rely*=0, *anchor*=N)

settings(settingsPage)

def settings(*pageName*):

logoutButton = tkinter.Button(*pageName*, *font*=myFont, *text*="LOGOUT", *command*=lambda:[logout(*pageName*)])

logoutButton.place(*relx*=0.5, *rely*=0.5, *anchor*=CENTER)

newRouteButton = tkinter.Button(*pageName*, *font*=myFont, *text*="NEW ROUTE", *command*=lambda:[newRoute()])

newRouteButton.place(*relx*=0.5, *rely*=0.6, *anchor*=CENTER)

def logout(*pageName*):

*pageName*.quit()

def newRoute():

routeList = open("routeList.txt", "a")

newRouteInfo = {

"Number":"",

"Name":"",

"URL":""

}

*#config page*

newRoutePage = tkinter.Toplevel()

newRoutePage.title("Brighter Futures Bus Application")

newRoutePage.config(*bg*='#8D16D8')

newRoutePage.geometry("960x540+250+150")

*#^^ places in direct center of page*

*#config header*

headerLabel = tkinter.Label(newRoutePage, *text*="NEW ROUTE")

headerLabel.config(*font*=myFont,*width*=150, *height*=2)

headerLabel.place(*relx*=0.5, *rely*=0, *anchor*=N)

*#config entry fields*

routeNumber = tkinter.Entry(newRoutePage, *font*=myFont)

routeName = tkinter.Entry(newRoutePage, *font*=myFont)

routeURL = tkinter.Entry(newRoutePage, *font*=myFont)

*#place entry fields*

routeNumber.place(*relx*=0.5, *rely*=0.4, *anchor*=CENTER)

routeName.place(*relx*=0.5, *rely*=0.5, *anchor*=CENTER)

routeURL.place(*relx*=0.5, *rely*=0.6, *anchor*=CENTER)

*#config entry labels*

tkinter.Label(newRoutePage, *font*=myFont, *text*="route number", *bg*='#8D16D8').place(*relx*=0.2, *rely*=0.4, *anchor*=W)

tkinter.Label(newRoutePage, *font*=myFont, *text*="route name", *bg*='#8D16D8').place(*relx*=0.2, *rely*=0.5, *anchor*=W)

tkinter.Label(newRoutePage, *font*=myFont, *text*="route URL", *bg*='#8D16D8').place(*relx*=0.2, *rely*=0.6, *anchor*=W)

def scrape():

*#scrapes web using given url and saves in new file with corresponding route number as file title*

webScraper.webScrape(newRouteInfo["URL"], newRouteInfo["Number"])

def setInformation():

*#validity check*

*if* routeName.get()!="" and routeNumber.get()!="" and routeURL.get()!="":

*#set dictionary values*

newRouteInfo["Name"]=routeName.get()

newRouteInfo["Number"]=routeNumber.get()

newRouteInfo["URL"]=routeURL.get()

routeList.write('\n')

routeList.write(newRouteInfo["Number"])

routeList.write('\n')

routeList.write(newRouteInfo["Name"])

print(newRouteInfo["Name"])

routeList.close()

scrape()

newRoutePage.destroy()

*else*:

displayError(newRoutePage, 0.5, 0.3, "1 or more text field(s) are empty, please correct and try again")

def displayError(*pageName*, *posX*, *posY*, *errorMessage*):

errorLabel = tkinter.Label(*pageName*, *bg*="#8D16D8", *fg*="red", *font*=myFont, *text*=*errorMessage*).place(*relx*=*posX*, *rely*=*posY*, *anchor*=CENTER)

*#confirmation button*

confirmButton = tkinter.Button(newRoutePage, *font*=myFont, *text*="CONFIRM", *command* = lambda:[setInformation()]).place(*relx*=0.5, *rely*=0.7, *anchor*=CENTER)

### routes.py

*import* tkinter

*from* tkinter *import* \*

*from* tkinter *import* font

*import* configurePage

*import* tables

myFont = "'Helvetica 18 bold'"

def routesPage():

routePage = tkinter.Tk()

global myFont

*#calls page configuration functions*

configurePage.configure(routePage)

configurePage.configHeader(routePage, "Routes")

'''

#tests label placement

testLabel = tkinter.Label(routePage, text="test")

testLabel.place(relx=0.5, rely=0.5, anchor=CENTER)

'''

*#generates sample table*

tables.tableSettings(700, 750, 0.5, 0.53)

tables.generateTable(routePage)

### routeScreen.py

*import* tkinter

*from* tkinter *import* \*

*from* tkinter *import* ttk

*import* mainApp

*import* routes

*import* os

*import* distanceCalculator

*import* re

*import* webbrowser

myFont = 'Helvetica 18 bold'

routeNumber = "73"

def \_\_main\_\_():

*#timetable config info*

tableHeight = 400

tableWidth = 300

tableX = 0.5

tableY = 0.52

*#0.52 to account for the headingof width 2, anchor N (means it takes rel 0.02 off the top)*

*#cannot use below functions from configurePage.py - causes circular input*

def configHeader(*pageName*, *headerText*):

headerLabel = tkinter.Label(*pageName*, *text*=*headerText*, *font*=myFont, *width*=150, *height*=2).place(*relx*=0.5, *rely*=0, *anchor*=N)

homeButton=tkinter.Button(*pageName*, *text*="HOME", *background*="yellow", *activebackground*="#9b870c", *font*=myFont, *command*=lambda:[mainApp.\_\_main\_\_(), *pageName*.destroy()]).place(*relx*=0.85, *rely*=0.025, *anchor*=CENTER)

routesButton=tkinter.Button(*pageName*, *text*="ROUTES", *background*="yellow", *activebackground*="#9b870c", *font*=myFont, *command*=lambda:[routes.routesPage(), *pageName*.destroy()]).place(*relx*=0.75, *rely*=0.025, *anchor*=CENTER)

def configurePage(*pageName*):

*pageName*.title("Brighter Futures Bus Application")

*pageName*.config(*bg*="#8D16D8")

*pageName*.geometry("1920x1080")

*#importing tables.py would cause circular import - added modified function here instead*

def generateTimetable(*pageName*, *tableX*, *tableY*, *tableHeight*, *tableWidth*):

*#configuring style for table contents*

tableContentsStyle = ttk.Style()

tableContentsStyle.configure("myStyle.Treeview", *font*='Helvetica 15')

*#configuring style for headings on tables*

tableStyle = ttk.Style()

tableStyle.configure("Treeview.Heading", *font*='Helvetica 18 bold')

tableContentsFileName = routeNumber

tableDataFileName = routeNumber

tableContentsFileName = re.findall(r'\d+', tableContentsFileName)

tableContentsFileName = (tableContentsFileName[0]+'.txt')

pathName = (os.getcwd()+'/timetableFiles/txtFiles')

completeFileName = os.path.join(pathName, tableContentsFileName)

tableFile = open(completeFileName, 'r')

tableInput = tableFile.readlines()

tableDataFileName = re.findall(r'\d+', tableDataFileName)

tableDataFileName = tableDataFileName[0]+'Data.txt'

tableDataFile=(os.path.join(pathName, tableDataFileName))

tableData = open(tableDataFile, 'r')

tableDataContents = tableData.readlines()

'''

print(tableData.readlines())

^^testing file output

'''

tableFrame = Frame(*pageName*)

tableFrame.config(*height*=*tableHeight*, *width*=*tableWidth*)

tableFrame.place(*relx*=*tableX*, *rely*=*tableY*, *anchor*=CENTER)

columns = []

*for* i *in* range (1, int(tableDataContents[1])+1):

columns.append(str(i))

timetable = ttk.Treeview(tableFrame, *style*="myStyle.Treeview", *height*=30)

timetable['columns']=columns

timetable.column("#0", *width*=0, *stretch*=NO)

timetable.column("1", *width*=200, *anchor*=CENTER)

*for* i *in* range (2, int(tableDataContents[1])+1):

timetable.column(str(i), *width* = 100, *anchor*=CENTER)

timetable.heading("#0", *text*="")

timetable.heading("#1", *text*="Stop location")

*for* i *in* range (2, int(tableDataContents[1])+1):

timetable.heading("#"+str(i), *text*=routeNumber)

i=0

c=0

fileLength = tableDataContents

*#while loop in that inputs timetable data to the table*

*while* i < int(re.findall(r'\d+', fileLength[2])[0]):

b=0

tableDataList = []

'''

#prints the first row - 0 issues

if i == 0:

for b in range(i, int(re.findall(r'\d+', tableDataContents[1])[0])+i):

if b < int(re.findall(r'\d+', fileLength[2])[0]):

tableDataList.append(tableInput[b])

print(tableDataList)

# prints the second row (some issues so far)

else:

for b in range(i, int(re.findall(r'\d+', tableDataContents[1])[0])+i):

if b < int(re.findall(r'\d+', fileLength[2])[0]):

c+=1

for d in range (0, int(re.findall(r'\d+', tableDataContents[1])[0])):

tableDataList.append(tableInput[d+c])

print(tableDataList)

# code that didnt really work - just kept repeating the same line again and again

'''

*while* b < int(re.findall(r'\d+', tableDataContents[1])[0]):

*if* i==0:

tableDataList.append(tableInput[b+i])

*elif* b+i+int(re.findall(r'\d+', tableDataContents[1])[0]) <int(re.findall(r'\d+', tableDataContents[2])[0]):

tableDataList.append(tableInput[c+int(re.findall(r'\d+', tableDataContents[1])[0])])

c+=1

print(tableDataList)

b+=1

*#writes data to the file*

timetable.insert(*parent*='', *index*='end', *iid*=i, *text*="", *values*=tableDataList)

print(i)

print(re.findall(r'\d+', tableDataContents[1]))

i = i+int(re.findall(r'\d+', tableDataContents[1])[0])+1

'''

timetable.insert(parent='', index='end', iid=0, text="", values=("Doncaster Interchange", "1000"))

timetable.insert(parent='', index='end', iid=1, text="", values=("Lakeside Village", "1000"))

'''

def OnDoubleClick(*self*):

global routeNumber

selectedItem = timetable.focus()

selectedStop = timetable.item(selectedItem, "values")[0]

distanceCalculator.coordGet(selectedStop+" Doncaster")

distanceCalculator.routing()

fileName = 'file:///'+os.getcwd()+'/'+'map.html'

webbrowser.open\_new\_tab(fileName)

timetable.bind("<Double-1>", OnDoubleClick)

tableData.close()

tableFile.close()

timetable.pack()

routePage = tkinter.Tk()

*##\n needed as route data brings text files new line declaration, meaning text will be uncentered*

configHeader(routePage,'\n'+ "ROUTE "+str(routeNumber))

configurePage(routePage)

generateTimetable(routePage, tableX, tableY, tableWidth, tableHeight)

routePage.mainloop()

### tables.py

*from* tkinter *import* \*

*from* tkinter *import* ttk

*import* distanceCalculator

*import* os

*import* webbrowser

*import* routeScreen

tableHeight = 0

tableWidth = 0

tableX = 0

tableY = 0

selectedRoute = ""

def tableSettings(*height*, *width*, *x*, *y*):

global tableHeight

global tableWidth

global tableX

global tableY

tableHeight=*height*

tableWidth=*width*

tableX=*x*

tableY=*y*

def generateTable(*pageName*):

routeList = open('routeList.txt', 'r')

content=routeList.readlines()

tableFrame = Frame(*pageName*)

tableFrame.config(*height*=tableHeight, *width*=tableWidth)

tableFrame.place(*relx*=tableX, *rely*=tableY, *anchor*=CENTER)

timetable = ttk.Treeview(tableFrame)

timetable['columns']= ('1', '2')

timetable.column("#0", *width*=0, *stretch*=NO)

timetable.column("1", *width*=100, *anchor*=CENTER)

timetable.column("2", *width*=600, *anchor*=W)

timetable.heading("#0", *text*="")

timetable.heading("#1", *text*="route number")

timetable.heading("#2", *text*="route name")

def OnDoubleClick(*self*):

global selectedRoute

distanceCalculator.coordGet("Bawtry Road")

distanceCalculator.routing()

selectedItem = timetable.focus()

selectedRoute = timetable.item(selectedItem, "values")[0]

routeScreen.routeNumber=selectedRoute

*pageName*.destroy()

routeScreen.\_\_main\_\_()

timetable.bind("<Double-1>", OnDoubleClick)

i=0

*while* i<(len(content)-1) and i<20:

timetable.insert(*parent*='', *index*='end', *iid*=i+1, *text*="", *values*=(content[i], content[i+1]))

i+=2

timetable.pack()

'''

for i in range(0, len(content)):

timetable.insert(parent='', index='end', iid=i+1, text="", values=(content[i], content[i+1]))

'''

### webscraper.py

*from* tkinter *import* \*

*from* tkinter *import* ttk

*import* distanceCalculator

*import* os

*import* webbrowser

*import* routeScreen

tableHeight = 0

tableWidth = 0

tableX = 0

tableY = 0

selectedRoute = ""

def tableSettings(*height*, *width*, *x*, *y*):

global tableHeight

global tableWidth

global tableX

global tableY

tableHeight=*height*

tableWidth=*width*

tableX=*x*

tableY=*y*

def generateTable(*pageName*):

routeList = open('routeList.txt', 'r')

content=routeList.readlines()

tableFrame = Frame(*pageName*)

tableFrame.config(*height*=tableHeight, *width*=tableWidth)

tableFrame.place(*relx*=tableX, *rely*=tableY, *anchor*=CENTER)

timetable = ttk.Treeview(tableFrame)

timetable['columns']= ('1', '2')

timetable.column("#0", *width*=0, *stretch*=NO)

timetable.column("1", *width*=100, *anchor*=CENTER)

timetable.column("2", *width*=600, *anchor*=W)

timetable.heading("#0", *text*="")

timetable.heading("#1", *text*="route number")

timetable.heading("#2", *text*="route name")

def OnDoubleClick(*self*):

global selectedRoute

distanceCalculator.coordGet("Bawtry Road")

distanceCalculator.routing()

selectedItem = timetable.focus()

selectedRoute = timetable.item(selectedItem, "values")[0]

routeScreen.routeNumber=selectedRoute

*pageName*.destroy()

routeScreen.\_\_main\_\_()

timetable.bind("<Double-1>", OnDoubleClick)

i=0

*while* i<(len(content)-1) and i<20:

timetable.insert(*parent*='', *index*='end', *iid*=i+1, *text*="", *values*=(content[i], content[i+1]))

i+=2

timetable.pack()

'''

for i in range(0, len(content)):

timetable.insert(parent='', index='end', iid=i+1, text="", values=(content[i], content[i+1]))

'''