**Alex Mann**

Candidate number: \_\_\_\_\_

Centre name: \_\_\_\_\_

Centre number: \_\_\_\_\_

**Environmentally friendly food recommendation app**

Qualification code: \_\_\_\_\_

Date: \_\_\_\_\_

Contents

[Analysis 2](#_Toc98261650)

[Problem identification 2](#_Toc98261651)

[Why it is suited to a computational solution 2](#_Toc98261652)

[Computational methods that the solution lends itself to 2](#_Toc98261653)

[Stakeholders 3](#_Toc98261654)

[Algorithm research 3](#_Toc98261655)

[Existing product research 3](#_Toc98261656)

[Stakeholder research 5](#_Toc98261657)

[Survey 5](#_Toc98261658)

[Primary stakeholder research 7](#_Toc98261659)

[Requirements specification 8](#_Toc98261660)

[Essential features of solution 12](#_Toc98261661)

[Potential limitations of solution 13](#_Toc98261662)

[Hardware and software requirements 13](#_Toc98261663)

[Hardware 13](#_Toc98261664)

[Software 13](#_Toc98261665)

[Commentary 13](#_Toc98261666)

[Design 13](#_Toc98261667)

[Decomposition 13](#_Toc98261668)

[Product database 14](#_Toc98261669)

[Emissions algorithm 18](#_Toc98261670)

[Suggestion algorithm 24](#_Toc98261671)

[User interface 25](#_Toc98261672)

[Development 28](#_Toc98261673)

[Create sectors database 28](#_Toc98261674)

[Populate sectors database 28](#_Toc98261675)

[Create products database 37](#_Toc98261676)

[Populate products database 37](#_Toc98261677)

[Add emissions 59](#_Toc98261678)

[User interface 68](#_Toc98261679)

# Analysis

## Problem identification

In order to help combat climate change, many people are now trying to make more environmentally friendly purchasing choices in their everyday lives. However, it is often incredibly difficult to find suitable alternatives to food products and to know which products are better for the planet, as there are multiple factors which contribute to the emissions of an item. I would like to create a product to help a user find out what the best items are and offer alternatives which are similar and easy to switch to, but more sustainable.

This would require a program to scrape the internet to find relevant data and calculate the emissions of a product. This information would then be used in an app where a user could give a food item from a website, possibly in the form of a link, and the app would recommend alternatives. If I have time the interface could then be improved to either a web extension which would view a user’s online shopping basket and suggest alternatives on the page, or an item recognition software paired to a webcam or mobile camera which could take a photo of a physical item in a shop and show similar online products.

## Why it is suited to a computational solution

This problem lends itself well to a computational approach in many different ways. For example, the solution will need to collect a large amount of data on many products from various online sources, which would be incredibly time consuming to carry out manually, and the program can also continuously update its information. The software will also need to carry out a large number of calculations to work out the emissions of all the products, and then compare this data on many different products to inform the user of the best alternatives in terms of emissions. I may then integrate the software with online shopping websites to allow the user to alter their choices with minimal effort.

## Computational methods that the solution lends itself to

* Data retrieval with web scraping – the program would have to retrieve data from a variety of websites to find the environmental impact of a product and research suitable alternatives. This amount of data would be impractical for a person to do so the program will make the comparison process possible and enable users to make informed decisions on their food
* Database storage and analysis – the program will have to analyse a large quantity of data on production methods, production location, transport methods etc. on many different products to compare the emissions, making this perfect for a computer program
* Data presentation – the program will take a lot of data and represent it in a neat and easily comprehensible way to make the decisions as easy as possible for the user
* Problem decomposition – the problem can be broken down into a number of steps. These could be:
  + Scraping information about a product from the internet
  + Converting this data to an emissions statistic
  + Finding potential similar products
  + Calculating statistic for alternatives
  + Image recognition for products
  + Web scraping of online shopping basket
  + Data presentation of best changes
* Divide and conquer – the whole process would be unmanageable together, but by dividing into these subprograms, each step become challenging but doable, and then the processes can be combined with the divide and conquer method
* Abstraction – some of these processes rely on others to carry out their function, using abstraction the workings of the preceding functions can be ignored to simplify the problem

## Stakeholders

The potential clients for this software would be anyone who is interested in their environmental footprint which, in the modern day, is the majority of the population. I aim to make it as accessible and easy to use as possible to appeal to a wide range of people of all ages and technological abilities. The software could also be angled towards the vegetarian community who could use the software to find vegetarian alternatives to common meat-based foods.

The stakeholders for the software will be people with a particular interest in the environment who are keen to minimize their emissions from food as much as possible and may not know exactly which products have what effect on the environment.

During this project, I will consult a specific stakeholder, Kevin, a vegetarian and avid environmentalist who tries to buy environmentally friendly food wherever possible, but often finds it hard to find good alternatives and to know which items are better for the environment.

## Algorithm research

Before I begin, I need to know roughly how my emissions calculation algorithm will work and if it is possible to do accurately so I will research what the main emissions producing areas are in food production

My initial research showed 4 main sections: production, transportation, packaging, and cooking, and these can be calculated as follows:

* Production – this is the emissions it takes to make the food and is unique to each type of food. While it also depends on a few unknown factors such as the specific farming method, there exists accurate and well researched data online with the average emissions for each type of food, so I can use a database from the internet for this section
* Transportation – this is the emissions produced by transporting the food from the place of production to the distribution centre, and then to a user where it is delivered. I can write an algorithm to calculate this using the place of origin of the food and the location of the user, which can then be used to calculate the distance travelled, which can then be used with the method of transport, the emission rate of that mode of transport and the mass of the product to calculate transport emissions
* Packaging – my initial research showed the emissions of packaging are negligible compared to the other three sections, so I do not need to consider this section
* Cooking – this is the emissions produced to cook the item, which is affected by the cooking method and cooking time. However, most products will not have cooking information available on the website and the emissions will be heavily dependent on the specific appliances owned by the user, so I will only add this section if I have time at the end

## Existing product research

There are some companies aiming to achieve a similar purpose, but I do not think any of them have solved the specific problem I would like to solve. For example, there are many apps which work in a similar way but instead of giving information on environmental impact they give health data. One of these is EWG Food Scores (Fig 1), which is an easy-to-navigate database with ratings for thousands of products based on nutrition, ingredients, and processing. I would like to take a similar format of data presentation with visual graphs and helpful colours to make the program as intuitive as possible.

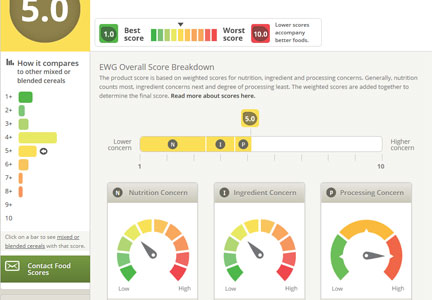


Figure - ‘EWG Food Scores’ example page

However, some projects have aimed to tackle the environmental impact of food, and one example of this is ‘Happy Cow’ (Fig 2), a simple and user-friendly app which tells you which local places offer vegan or vegetarian food. While this is useful, my aim is more to give environmentally better alternatives without drastic changes to someone’s diet, for example by changing from one brand of bread to another, or one cut of meat to another.

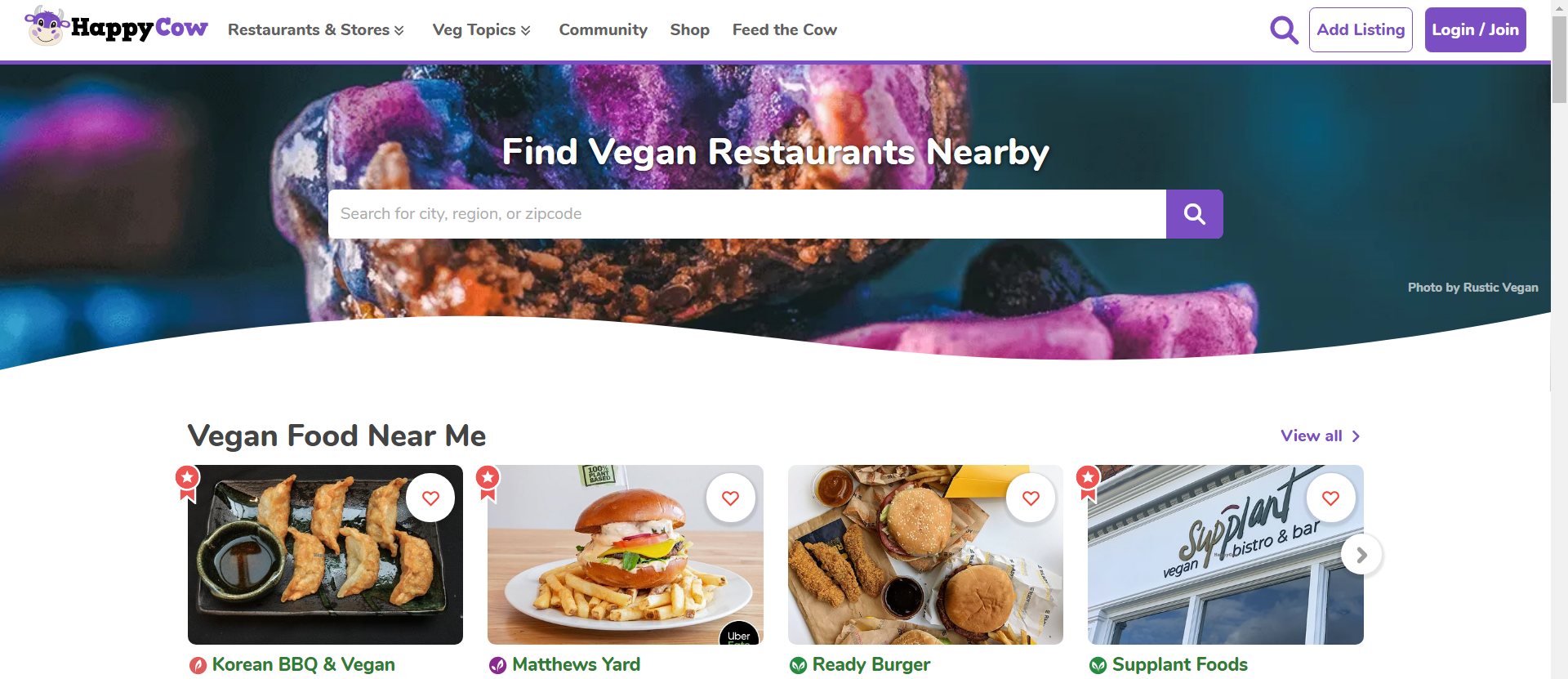


Figure - ‘Happy Cow’ home page

Another example is ‘How Good’ (Fig 3). This is the software which is most similar to my idea that I could find. It gives a detailed breakdown of the environmental impact of various foods and offers alternatives, but it is a largely commercial product aimed at big producers to make changes in their production line and stick a ‘How Good’ certificate on their products. I would like to aim for a smaller scale, personal product which enables individuals to make changes to their diets to benefit the environment. I would however like to achieve the detailed data analysis of ‘How Good’ on a smaller scale.

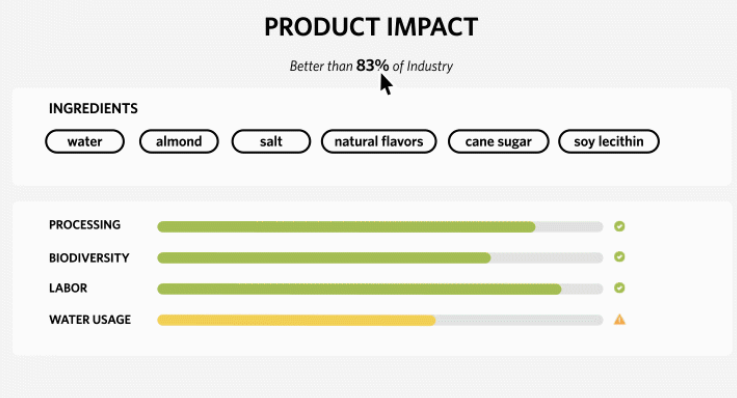


Figure – ‘How Good’ example page

## Stakeholder research

### Survey

My first step in stakeholder research was to carry out an online survey to find out if my idea would be useful, so I wrote questions to find out people’s current attitudes and opinions towards food shopping and the environment, and then distributed the survey amongst people I knew, covering as diverse of a demographic as possible. My survey had 30 respondents, and the results are analysed in the table below:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Question** | **Results** | **Why I chose this question** | **Results analysis** | **What this means for my project** |
| How often do you shop for groceries online? |  | I wanted to find out how frequently online grocery shopping is used to find out how much positive impact my product could have | 60% of users regularly shop for groceries online, with an average of 1.1 shops per week | Online grocery shopping is clearly widely used which means my product has the potential to have a significant positive impact on the environment |
| Which site do you use most often for grocery shopping? |  | My product will have to, at least originally, be specific to only one or two websites, so knowing which one more people use will help me decide which to focus on | Sainsbury’s was the most used website with 9 people, closely followed by Tesco with 7 | Dependent on later research, I will likely ensure my product is compatible with Sainsbury’s, and may also try to integrate with Tesco |
| How important is environmental impact to you when choosing food to buy? |  | I need to find out what people’s current attitudes are towards the impact of their food, if emissions are not something most people care about, my product will not have much impact. However, this also might be affected by how well informed they are about the food they buy, so I will clarify this with the proceeding questions | The majority of people said environmental impact was somewhat important, while no one said it was not at all important | All the respondents to my survey attached some importance to the environmental impact of their food, so it seems that with relevant and accurate information people would make some effort to reduce their emissions |
| How aware are you of the environmental impacts of different foods? |  | If most people are not particularly aware of the impacts of their food, but would be willing to change their habits if they had more information, then my product has the potential to greatly reduce emissions, while if everyone already knows all the relevant emissions statistics, my product is redundant | 56% of people said they were somewhat aware, and the data is clearly centred around the centre of the range | This suggests most people think they have a reasonable grasp of the environmental impacts of food, but most do not feel fully confident in their knowledge which could cause them not to make changes to their lifestyle based on unsure data. My product can therefore help this and encourage people to make changes to the food they consume |
| Rank the following items in order of emissions per kilogram you think they produce |  | This question quantitatively tests the perceptions people had in the previous question about their own knowledge and shows how accurate the data people have is. I tried to choose a wide variety of items to find out where, if any, are the most common misconceptions | The first graph shows how the average person ranked the items, while the order they are listed is the correct order. The scatter plot shows average rank on the x axis against emissions on the y axis, excluding beef. This has only a 60% correlation and it is clear from the graph the ranks are largely inaccurate | Most people correctly predicted the top product (beef) and the put the lowest 4 at the bottom (Tomatoes, pasta, frozen raspberries, potatoes), but people were not good at differentiating between the middle items. This shows the importance of my product to inform people about this data |
| Which of the following would you be willing to do to reduce the emissions of your food | |  |  | | --- | --- | | Choice | % Responses | | Switch to products produced more locally | 92 | | Switch to more expensive products | 47 | | Switch to slightly different products of the same type (e.g., different types of cheese or different sizes of fruit) | 72 | | Switch to products with a different production or transportation method (e.g., fresh to frozen) | 72 | | Switch to product alternatives (e.g., cow's milk to soy milk) | 39 | | Switch to similar foods (e.g., chicken to turkey) | 47 | | Switch to completely different foods | 17 | | I need to find out which types of sacrifice people would be willing to make to help the environment, so I know how best to suggest alternatives to a user. If people are only willing to make insignificant adjustments to their diet, then I need to only recommend products with minimal change, but if they are willing to change everything about what they eat then I can expand my range of suggestions | Most people are willing to make some sacrifices but there is a lot of variety in what sacrifices people would make | The fact most people ticked at least one box shows that my product can encourage positive change, while the variety suggests I need to allow the user to select the types of suggestions the app gives |

### Primary stakeholder research

My primary stakeholder is Kevin, he is an avid environmentalist and vegetarian so is very interested in this product. I initially chose to speak to him via email since there was no physical evidence I needed to show him, and so it was easier and more efficient to contact him remotely. Here is what he said:

*“Hi*

*I really like the idea for the project and think it has the potential to change a lot of people’s habits and awareness and save a lot of emissions. Here are some things I would like to see in the product:*

*Firstly, the data obviously needs to be accurate and reliable otherwise the whole process is useless, getting people to switch to items which we don’t really know if they’re better or not. But it’s also important that the data is given to a reasonably high level of precision to give the impression of accuracy, since I think people would be more likely to switch if they thought the data was accurate and reliable, I think 2 significant figures should be enough*

*Secondly, the data needs to be presented in a sensible way, I need to be able to easily compare the relevant information from various products without having to navigate around lots of pages. The data should also be given in sensible units, I don’t want to have to read 10 decimal places for no reason, and it would also be quite nice to be able to filter or sort the data, for example to only include vegetarian items, or to sort by emissions or by customer rating. Also, in order to choose a product, I need to obviously know its name, price and customer rating, and the mass and/or number of portions would be nice. In terms of the way emissions statistics should be given, I think emissions per portion would be the most useful metric, but it would also probably be nice to be able to switch between a few slightly varying metrics. However whichever metric is given I would definitely like to have either a traffic-light system to easily show how the products compare, and maybe the data could be compared to everyday events (this product is equivalent to a 5-minute drive).*

*In terms of the suggestions you’re giving, obviously these have to be relevant to my search, I don’t want to search for beef, and it suggests cereal, that’s not helpful and is not a change anyone would ever agree to. I also think it would be nice to be able to see why a product was swapped (“Asparagus from Peru -> Kale from Kent”) and to be able to change the strictness of the search since different people might have different opinions on how much or little they want to change*

*Finally, for the interface, it needs to be straightforward to use and it should be compatible with all common devices. Also, it needs to be compatible with either a lot of sites or one or two very popular sites otherwise no one will be able to use it. It would be great if the product looked nice too, but that’s not essential for a functional product like this. Some other things I would quite like would be for a search bar to always be present on any page, and to be able to search for a type of product or for the link to a product and it gives relevant suggestions, but either of these individually would be fine.*

*I hope that’s helpful and good luck*

*Kevin”*

## Requirements specification

Key:

* Essential features/stakeholder needs
* Nice-to-have features/stakeholder wants
* Requested by primary stakeholder

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **#** | | **User requirement** | **Solves** | **Evidence** | **Justification** | **Success Criteria** |
|  |  | **Emissions data** |  |  |  |  |
| 1 | 1.1 | Accurate data | Enable useful comparison between products | “The data obviously needs to be accurate and reliable” | Need data to be accurate enough to differentiate between various products’ emissions to recommend lower impact products | -Calculation algorithm considers all available factors using reliable and well-research data  -Checked by explaining algorithm to stakeholder and asking if acceptable |
|  | Precise data | Gives sufficient information to differentiate similar products | “It’s also important that the data is given to a reasonably high level of precision” | Shows a user that the data is accurate and enables differentiation between similar products, for example giving 1.3 and 1.1 shows a difference, rounding to 1 and 1 does not | -Data given to at least 2 significant figures |
| 1.2 | Production data | Significant part of important emissions data | My initial research showed this as a key are of emissions production | This is a major factor in the environmental impact of a food product, so I need to make sure I consider this in my calculations | -Product uses an algorithm to calculate production emissions  -Algorithm uses reliable and well-researched data  -Checked by explaining algorithm to stakeholder and asking if acceptable |
| 1.3 | Transport data | -Product uses an algorithm to calculate transport emissions  -Algorithm uses reliable and well-researched data  -Checked by explaining algorithm to stakeholder and asking if acceptable |
| 1.4 | Cooking data | -Product uses an algorithm to calculate cooking emissions  -Algorithm uses reliable and well-researched data  -Checked by explaining algorithm to stakeholder and asking if acceptable |
| 1.5 | Statistics given in sensible units | Enables people to understand the impact of their food more easily | “The data should also be given in sensible units” | If people are able to easily understand the information, they can make more informed decisions | -Statistics given in sensible units  -Number of non-significant figures never greater than 2  -Ask for stakeholder approval |
| 1.6 | Emissions data compared to everyday events | “Maybe the data could be compared to everyday events” | -Emissions data compared to suitable and simple everyday event  -Ask for primary stakeholder’s approval |
| 1.7 | Traffic light system for emissions data | “Definitely like to have … a traffic-light system” | -Colour system to indicate emission level  -Ask for primary stakeholder’s approval |
| 1.8 | User options for data format | Enables people to view emissions in different units, i.e., per kilogram, per serving, per portion or per calorie | “Would also probably be nice to be able to switch between a few slightly varying metrics” | Different metrics have different uses, e.g. people buy a product based on mass, emissions per kilogram is more useful, while if someone is aiming to buy a certain number of portions, emissions per portion is better | -Option for user to view data on, sort by and suggest based on multiple emissions metrics  -Options for units include CO2 emissions per kilogram, serving, portion and calorie |
|  |  | **Suggestion algorithm** |  |  |  |  |
| 2 | 2.1 | Relevant suggestions | Makes suggested alternatives more likely to be accepted by user | This is clearly advantageous  “Obviously [the suggestions] have to be relevant to my search” | I want to make people make as many environmentally-positive changes as possible, and relevant suggestions would clearly increase how many changes people make | -Trial should have at least a 20% suggestion accepting rate |
|  | 2.2 | Customisable suggestion criteria | Enables user to change which type of suggestions are given | My survey showed people have a variety of opinions on what changes they would make  “I also think it would be nice … to be able to change the strictness of the search” | It is clear this should be a user option to maximise positive changes | -User has option to change the strictness of the search |
|  |  | **Interface** |  |  |  |  |
| 3 | 3.1 | Easy to use | Enables user to use product easily | “Needs to be straightforward to use” | Increases likelihood of people using and continuing to use the product | -Primary stakeholder and majority of users in a trial are happy with ease of use |
|  | 3.2 | Compatibility with shopping sites | Product works for a reasonable number of potential users | “Needs to be compatible with … one or two very popular sites”  My survey showed the highest use for Sainsbury’s | Needs to be easily compatible with the most common website(s) to increase usership | -Compatible with at least Sainsbury’s, the most common website from my survey |
|  | 3.3 | Compatibility with devices | Product works on desktop, tablet and mobile | “Should be compatible with all common devices” | Increases usership and ease of use | -Product works and stakeholder is satisfied with ease of use on desktop, tablet and mobile |
|  | 3.4 | Aesthetically pleasing | Makes it nicer for users to use | “Would be great if the product looked nice too” | Increases likelihood of people using and continuing to use the product | -Primary stakeholder and majority of users in a trial are happy with layout |
|  | 3.5 | Constant search bar | Users always able to search for product | “a search bar to always be present on any page” | Enables users to quickly and easily be able to search for new products or comparisons wherever they are on the app | -Search bar always visible in app  -Search bar works |
|  | 3.6 | Search by product or URL | Gives users range of search options | “Be able to search for a type product or for the link to a product and it gives relevant suggestions” | Enables users to search for a product (e.g., bread) and be given comparisons of different bread products, or a link to a specific bread product on the Sainsbury’s website and similar options would be recommended | -Option to search by search term or URL |
|  | 3.7 | List of multiple options displayed | Gives users multiple choices on environmentally friendly options | “I need to be able to easily compare the relevant information from various products” | When a user searches for an item, it needs to give multiple alternatives with relevant data so the user can choose between them | -At least 3 alternatives displayed  -All alternatives relevant in some way |
|  | 3.8 | Name of product displayed for each item | Shows relevant data | “Also, in order to choose a product I need to obviously know its name, price and customer rating, and the mass and/or number of portions would be nice” | This is a detail which may influence which product a user chooses so it is vital to show this information for each product | -Product name displayed for each item |
|  | 3.9 | Price displayed for each item | -Accurate price displayed for each item |
|  | 3.10 | Customer rating displayed for each item | -Accurate rating displayed for each item |
|  | 3.11 | Mass displayed for each item | -Stated mass displayed for each item |
|  | 3.12 | Number of portions displayed for each item | -Stated number of portions displayed for each item |
|  | 3.12 | All data comparison given on one page | Easy comparison of products | “I need to be able to easily compare the relevant information from various products without having to navigate around lots of pages” | People need to be able to see all relevant in one place at the same time to easily be able to make decisions on which products to buy | -All data comparison given on one page |

## Essential features of solution

In summary, the main features which are required for my product are as follows:

* Easy-to-use system – must be minimal effort to encourage users to make sacrifices in their diets
* Accurate data – must give correct and useful information otherwise recommendations are meaningless
* Helpful alternatives – must have balance between environmental benefit and similarity to initial item. The user is unlikely to change from steak to artificial meat but change to a locally produced or organic meat.

## Potential limitations of solution

* May be difficult to get exact data – producers will never publish environmental impacts and not all factors will be discoverable with just the product name
* People may be more concerned with health, price, or taste rather than environmental impact
* Will only work with a limited number of online shopping websites, probably only Sainsburys, since each site is incredibly time consuming to cover
* Products with multiple ingredients (pies, ready meals, cakes) do not have listed quantities of ingredients so it will not be possible for me to calculate its emissions, so my product will only work for single ingredient products or products with known emissions (apples, steak, bread)

## Hardware and software requirements

### Hardware

* Smart mobile device capable of running the app, with touchscreen to interact with the program
* Computer capable of running the web extension
* Camera to take a photo of a product to search for, must be of sufficient resolution in order to recognise the product from an image

### Software

* Windows, Linux, or Mac operating system – operating systems supported by python
* Apple or Android mobile device to run app
* Python interpreter – the project will be written in python, so requires an interpreter
* Tkinter for Python – library used to make apps
* BeautifulSoup for Python – library used to scrape website data
* Webdriver for Python – library used to open pages on a web browser from my app
* Pandas for Python – library used to handle the large datasets
* Heroku – scheduler and web-based database storage, used to store, update, and access the database of products and relevant information about them used to calculate environmental data
* Psycopg2 module for Python – to access and update databases on Heroku from Python

## Commentary

Having researched all areas of my project from various sources, I now know the requirements for my product and the rough form it will take, so I can begin to design the product

# Design

## Decomposition

My project will be divided into 4 main sections:

* A database of all the products on one or more food shopping websites with key information such as price, size, rating etc. and also relevant data used to calculate emissions
* An algorithm which calculates the emissions of a product based on available data such as transport times, recycling material and ingredients
* An algorithm which takes an input product and recommends similar products which are more environmentally friendly, while also considering cost and similarity
* An interface for a user to use the rest of algorithms to change their online shopping choices, either in the form of a web extension or an app

### Product database

In order to create the product database, I will write python scripts to scrape the product data off supermarket websites.

My first step is to choose which site(s) to use for the data, and I will choose this based on which has the most useful data for calculating emissions and how popular each website is. I have already gathered from my survey that Sainsbury’s is the most common site for my users, followed by Tesco, but I will also look at other factors. I compared the product pages on a variety of popular websites in the following table

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Brand** | **Product details** | | | | | | | | **Estimated UK users (millions)[[1]](#footnote-1)** |
| **Expiry duration** | **Vegan/ vegetarian** | **Detailed nutritional information** | **Package recycling** | **Cooking instructions** | **Environmental certificates** | **Religious information** | **Place of production** |
|  | ✔ | ✔ | ✖ | ✔✔ | ✔ | ✔ | ✔ | ✔ | **30.1** |
|  | ✔ | ✔ | ✖ | ✖✖ | ✔ | ✔ | ✖ | ✖ | **23.2** |
|  | ✔ | ✔ | ✔ | ✔✔ | ✔ | ✔ | ✖ | ✔ | **17.1** |
| Amazon.co.uk: Amazon Fresh Stores | Now Open in London | ✖ | ✔ | ✖ | ✖✖ | ✔ | ✖ | ✖ | ✖ | **15.7** |
|  | ✖ | ✔ | ✖ | ✔✖ | ✔ | ✖ | ✖ | ✖ | **15.7** |
|  | ✔ | ✔ | ✖ | ✔✔ | ✔ | ✖ | ✔ | ✖ | **15.0** |
| The Co-operative Group - Wikipedia | ✖ | ✖ | ✖ | ✔✔ | ✖ | ✖ | ✖ | ✔ | **6.8** |
|  | ✔ | ✔ | ✖ | ✔✔ | ✔ | ✔ | ✖ | ✔ | **5.5** |
|  | ✖ | ✔ | ✖ | (✔✔)[[2]](#footnote-2)\* | ✖ | ✖ | ✖ | ✔ | **5.5** |

From this table I think that the most suitable websites for my project are Sainsbury’s and Tesco due to their combination of a large amount of useful information and their high number of users. Having better data makes it easier for me to make accurate emissions estimations on products and the website having more customers makes my product useful to a wide range of people. I will begin by collecting data from Sainsbury’s since it has the easiest layout for web scraping and had the highest number of users in my survey, and then I will add products from Tesco later if I have time.

Then I will have to write a python script to scrape the data off these websites. I will use the Python library Beautiful Soup to parse the page and extract the data. Firstly, I will have to scrape the list of sectors within the websites into a ‘sectors’ database. These sectors make up the tree of categories in which the products are sorted on the websites (as below). Then for each of these sectors, I will scrape the home page of this sector for a list of all the products in the sector and visit each product page to gather the relevant information. The products and their data will be stored in a ‘products’ database.

Before I can collect the data, I need to design the databases to store the information:

**‘Sectors’ database:**

|  |  |  |  |
| --- | --- | --- | --- |
| Field name | Data type | Description | Example |
| uuid | uuid | Unique ID of sector | 36cc5ac9-86a4-471e-bba6-6e32505abde6 |
| name | string | Name of sector | “Bacon” |
| path | string | Full path of sector in tree | “Meat & fish>Bacon & sausages>Bacon” |
| num\_products | int | Number of products in the sector | 150 |
| date\_updated | date | Last date at which the products in this sector were scraped, None if never updated | 01/01/2021 |
| website | string | Which website the sector is on | “Sainsbury’s” |
| url | string | The link to the sector page | “https://www.sainsburys.co.uk/shop/gb/groceries/meat-fish/bacon” |

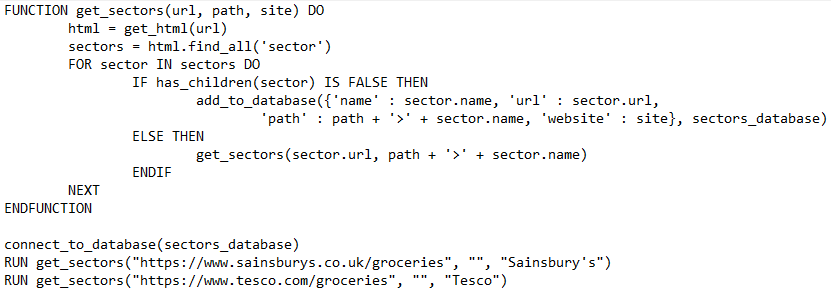
**‘Products’ database:**

|  |  |  |  |
| --- | --- | --- | --- |
| Field name | Data type | Description | Example |
| uuid | uuid | Unique ID of product | 5c87faef-cf83-4f9d-9964-3137aa9cd23a |
| name | string | Name of product | “Sainsbury’s Smoked Streaky British Bacon, Taste the Difference 220g” |
| date\_updated | date | Last date at which the product information was updated | 01/01/2021 |
| url | string | The link to the product page | “https://www.sainsburys.co.uk/gol-ui/product/bacon/sainsburys-ultimate-smoked-streaky-bacon--taste-the-difference-220g” |
| sector\_id | uuid | Uuid of sector which the product is in | 36cc5ac9-86a4-471e-bba6-6e32505abde6 |
| price | float | Price of product in £ | 2.00 |
| rating | float | Star rating out of 5 from customer reviews | 3.6 |
| num\_reviews | int | Number of customer reviews on the product | 61 |
| expiry\_duration | int | Typical life of product after purchase, in days | 7 |
| vegetarian | bool | Whether the product is vegetarian or not | FALSE |
| vegan | bool | Whether the product is vegan or not | FALSE |
| religious\_info | string | List of religious information e.g. Kosher, Halal | “[]” |
| description | string | Product description from website, for if a user wants to see more information about a potential replacement | “Oak smoked, dry cured, air dried, streaky bacon rashers made with Outdoor Bred British Pork.” |
| nutrition\_100 | string | Dictionary of nutritional information per 100g, all quantities measured in g except energy in kcal | “{‘Energy’: 337, ‘Fat’: 21.3, ‘Saturates’: 8.5, ‘Mono-unsaturates’: 14.3, ‘Polyunsaturates’: 2.5, ‘Carbohydrate’: 1.0, ‘Sugars’: 0.5, ‘Fibre’: 0.6, ‘Protein’: 31.0, ‘Salt’, 4.50}” |
| pack\_servings | int | Number of servings per pack | 7 |
| mass | int | Mass of product in grams | 220 |
| ingredients | string | List of ingredients | “[British Pork, Sea Salt, Sugar, Preservatives: Sodium Nitrite, Sodium Nitrate, Antioxidant: Sodium Ascorbate]” |
| allergens | string | List of allergens, empty list if none | “[]” |
| standards | string | List of any environmental/animal welfare certificates or standards associated with the product | “[RSPCA]” |
| info | string | List of any extra environmental/health information associated with the product | “[Outdoor bred]” |
| cook\_times | string | Dictionary of the average cooking time for various appliances, in minutes | “{‘Grill’: 5.5, ‘ShallowFry’: 3.5}” |
| origin\_country | string | Country of origin of product | “United Kingdom” |
| recycling | string | Dictionary of recycling instructions of the various parts of the packaging | “{‘base tray’: None, ‘top film’, None, ‘base Label’: ‘recycle’}” |

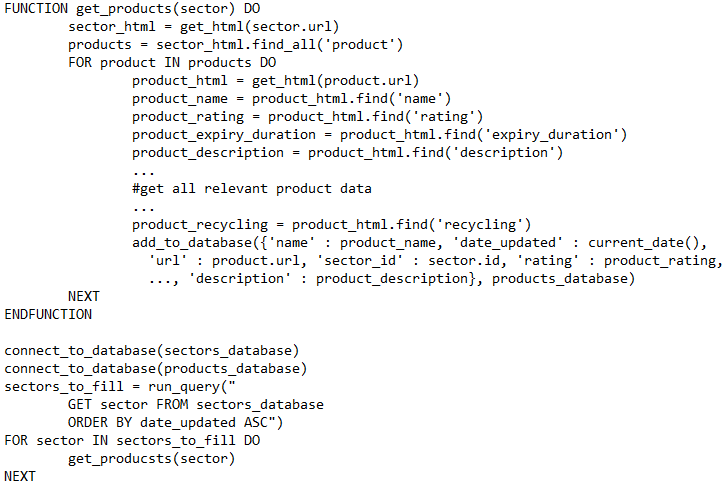
There is a one-to-many relationship between products and sectors between ‘sector\_id’ in the products database and ‘id’ in the sectors database. The databases are normalised to remove data redundancy, with all fields depending only on the ‘id’ column in each database

Then I will have to populate the database which I will do using 2 python scripts. The first will collect the data of all the sectors on the website and the second collect all the products for a given sector. Then by running the second program for each sector found by the first program I can populate the databases with all the products on the website. I created flow charts for each program to plan how they will work:

Program 1 – get sectors:



Program 2 – get products:



The final database is the ‘emissions\_stats’ database. This is initially the SuEatable food emissions database I downloaded from the internet, which was a result of extensive research done by the company. It initially contains the names of 325 products with the min, median and max emissions of production for that product in kg CO2 per kg food, but I will only use the median emissions for my calculations. I will have to do some modification to the database later to enable matching of each product in the products database with a row in this database. This database is constant, it will never be modified later

**‘Emissions\_stats’ database:**

|  |  |  |  |
| --- | --- | --- | --- |
| Field name | Data type | Description | Example |
| id | int | Unique ID of food type | 0 |
| name | string | Name of food type | “Beef” |
| emissions | float | kg CO2 per kg food type | 25.75 |

### Emissions algorithm

In order to calculate emissions of various food products, I had to research how various factors affect environmental impact. The three factors which produce emissions are production, which is determined by the type of food, transportation, and packaging

#### Production:

Firstly, I looked for existing research on the base impact of producing various food, and I found an article and database produced by scientists at SU-EATABLE[[3]](#footnote-3) with data on Carbon Footprints on 3350 products of 325 types. For each sector in my database, I can then link the sector to a row in the SU-EATABLE database to give the production emissions of the food type.

#### Transport:

Map

Description automatically generatedThe Sainsbury’s and Tesco’s websites have data on place of origin for most products, and I can use this to calculate the emissions of transport. I can create a list of all the Tesco’s and Sainsbury’s distribution centres and their coordinates using Google Maps (as below) and then use python to find the closest one to a user’s current location using ‘geocoder’ and ‘geopy’ python modules. From some internet research I found out that all products produced outside the UK are transported in by ship or by air, and I was able to obtain a list of products which are transported in each of these two methods. For transport by sea, I can assume the products are being transported to the nearest port from the place of origin and transported by road to the distribution centre and then to the user’s location, while for air-freighted goods I will assume they are flown directly to the distribution centre and then driven to the user location. I can calculate these distances, and then using the mass of the product and the emission coefficients (kilograms CO2 emitted per kilometre per tonne) for each transport type I can calculate the total emissions due to transportation for each product. The distances will not be completely accurate due to the specific shapes of roads and coasts, but I think they will be accurate enough to form a useful estimate.

Figure - map of UK Sainsbury's distribution centres

I can use known values for the coefficients for transport type in the calculations:[[4]](#footnote-4)

|  |  |  |
| --- | --- | --- |
| **Transport type** | **Emission coefficient** | |
| **Ambient** | **Temperature-controlled** |
| **Road** | 0.20 | 0.50 |
| **Sea** | 0.01 | 0.02 |
| **Air** | 1.13 | 1.13 |

The general formula for transport emissions is:

As in the table, the coefficients are different for temperature controlled or ambient transport, and this will depend on the food type. The specific algorithms for each food depending on whether it needs to be temperature controlled and whether it is transported by air or sea will therefore be as follow:

|  |  |  |
| --- | --- | --- |
| **Transport emissions** | **Ambient** | **Temperature-controlled** |
| **Sea** |  |  |
| **Air** |  |  |

#### Cooking:

In my initial data collection, I will gather data where it is available on the cooking instructions for each product. Then I can find estimates on the internet for emissions caused by each cooking method (microwave, oven, etc.) and how the cooking time affects these emissions. However I don’t think enough products will have this information to be useful, so I will first collect the data and then decide whether or not to implement an algorithm

#### Pseudocode:

These 3 components: food, transport, and cooking, make up the emissions of each product, and so I will write another program to combine these and calculate the total emissions for each record in my products database, although my initial design will not include cooking emissions as previously discussed. Emissions per kg, per portion and per calorie are also important metrics which people will want to compare the products by, so the program should also calculate these and put the information in the database. I also realised that the road emissions for each product will be effectively identical per kg, so I can ignore this part since the primary purpose of the program is to compare different items

The full pseudocode is shown below, broken down into sections:

Firstly, I set up the constants and data for the program:

Text

Description automatically generated

* **country\_list** is a list of all the countries in the world in lowercase and in the format used by the **geopy** python module
* **emissions** is a table of the emissions rates of various mode of transport in kg CO2 per kg of item per km travelled, stored as a dictionary of strings to lists
* **air\_food** is a list of all food items which are transported by air rather than sea
* **temp\_controlled\_food** is a list of all food items which need to be temperature controlled during transport
* **current\_location** is the country of residence of the user, for now this is always set to ‘united kingdom’ since the websites I am catering for only operate in the UK

Next, I define some functions for my program:

Text

Description automatically generated

* **get\_country\_coords** returns the geographical coordinates of the centre of a given **country** using the **geopy** python module. The country name must be in the format used by geopy
* **get\_countries\_distance** returns the geographical distance in km between two countries using the **geopy** module. The names are these two countries are passed in the parameters **to** and **from**
* **get\_travel\_emissions** returns the emissions caused by the transport of a food item of mass **mass**, travelling a distance **dist**, by a mode of transport **transport** (sea or air). The parameter **temp** is a boolean value indicating whether or not the product needs to be temperature controlled during transport

Then I get the full **products** data from Heroku:

Text

Description automatically generated

* This query selects all the products from the **products** database along with the ID of the name of the sector they are in
* A **LEFT JOIN** is used to get the sectors name by linking the **products** and **sectors** databases with the sector\_id
* This program as well as the program to update the **sectors** and **products** databases will be run on a regular e.g. hourly basis, so the query only returns rows where the emissions have not already been calculated (line 24)
* In line 26, **execute\_query** is a function which executes an SQL query (passed as a parameter) on Heroku and has access to my databases. The function returns the result of running the query as a pandas dataframe, and here this is stored in **result**

Then I get the **emissions\_stats** database:



* The query selects the whole **emissions\_stats** database
* **execute\_query** executes the query and stores the result in **links**

For each product, I then calculate the emissions. This first section begins to calculate transport emissions by finding the possible places of origin of the product:

Text

Description automatically generated

* Line 31 loops through each product in **result**
* **transport\_emissions** will store the emissions caused by the transport of this product. It is initially set to -1 so if errors are thrown later and this value is not updated, -1 signifies this error and the program will act accordingly
* In line 33, the ‘origin\_country’ field of the given product is made lower case and split up into a list of words, which is stored in the variable **origin\_country**. For example, the string ‘this product is grown in France Germany or the United States’ is converted to the list [‘this’, ‘product’, ‘is’, ‘grown’, ‘in’, ‘france’, ‘germany’, ‘or’, ‘the’, ‘united’, ‘states’]
* In line 34, **product\_countries** is set to an empty list. This will later store the list of countries the product can be produced in
* In lines 35-36, the program loops through each combination of up to 5 consecutive words in the **origin\_country**. It only checks up to length 5 since this is the number of words in the longest country name
* In line 37, the set of words selected is joined by spaces, for example some values of **new\_string** during the execution of this selection on the above data would be ‘this’, ‘product’, ‘is’, …., ‘this product’, ‘product is’, …, ‘this product is’,…
* In lines 38-39, if **new\_string** is an item in **country\_list**, then this string is a country and the country is added to **product\_countries**. On the above data, the if statement will be true when new\_string equals ‘france’, ‘germany’, or ‘united states’, and so after this section is run, **product\_countries** will have the value [‘france’, ‘germany’, ‘united states’]

Then I convert the list of countries or origin into an average total transport emissions figure:

Text

Description automatically generated

* Line 43 checks if **product\_countries** is empty. If it is, the following code shouldn’t run otherwise an error will be thrown. This means if no countries are found, no transport emissions will be calculated and **transport\_emissions** will keep the default value -1
* Line 44 uses a list comprehension to convert each country in **product\_countries** into a distance from the user’s location using the **get\_countries\_distance** function, and then takes the average value of this list, hence finding the average distance the food item will travel from production to delivery
* **transport\_type** and **temp\_controlled** store the type of transport which this food is transported by and whether or not it needs to be temperature controlled respectively. They are initially set to default values: ‘sea’ and False
* Lines 46-50 check if any of the items in **air\_food** are in the name of the product, and if so, change **transport\_type** to air. E.g. if the product is called ‘Sainsburys green beans 200g’, and ‘green beans’ is in air\_food, them transport\_type will be set to ‘air’. The lower function is used to make all the information lower case
* Lines 51-55 check if any of the items in **temp\_controlled­\_food** are in the name of the product, and if so, change **temp\_controlled** to True
* In lines 56-59, if mass data is available for the product, then the transport emissions are calculated using **get\_travel\_emissions** and stored in **transport\_emissions**

Next, I calculate the production emissions of the product: Text

Description automatically generated

* **food\_emissions** is initially set to -1 as a default value
* Again, the data is only calculated if mass data is available
* Lines 59-63 see if the product matches any of the records in the **links** table, and if so the corresponding emissions per kg data is retrieved
* This is multiplied by the mass of the product to calculate production emissions

The emissions data is then combined and uploaded to the database on Heroku

Text

Description automatically generated

* Line 71 checks if both the **transport\_emissions** and **food\_emissions** were calculated as expected
* **total\_emissions** is the sum of the **transport\_emissions** and **food\_emissions**
* **update\_query** is an SQL query to set the ‘emissions’ field of the relevant row in the **products** database to **total\_emissions**.
* Line 78 runs the query to modify the table on Heroku

Once the program has looped through all the products, other metrics in the database are updated

Text

Description automatically generated

* **update\_query** is an SQL query to calculate the **emissions\_per\_kg**, **emissions\_per\_portion** and **emissions\_per\_calorie** columns in the database
* The **nutrition\_100[‘calories’]** value is the calories per 100g, so it is multiplied by 10 to find calories per kg, and then divided from the emissions multiplied by the mass to find the emissions per calorie
* In practice, this part will be more complicated for 3 reasons:
  + There may be divide by zero errors if the **mass**, **pack\_servings**, or **nutrition\_100[‘calories’]** data was not available, so these need to be checked for each field
  + The **nutrition\_100** field is actually stored in the form of a dictionary inside string, so this needs to be unpacked
  + **nutrition\_100** will often incomplete, and may not have calorie data, so I need to make sure the correct data is being used

### Suggestion algorithm

I need to only show relevant items which the user wants to see. I will try to have 2 modes of input, where the user can either search by a URL or for a food type (e.g. ‘Chicken’).

In the case of searching by food type, I can simply show all products where each word in the search term is either in the name of the product or the name of the sector. This can be done via an SQL query. For example, searching for ‘pork sausage’ would return ‘Pork sausages’, ‘Sausages made of pork’, and a ‘Sausages’ product in the ‘pork’ sector. I also may want to give the option for a user to manually toggle whether certain sectors are shown after the search, as for example searching for ‘raspberries’ might show strawberries in the ‘raspberries and strawberries’ section, so the user should be able to manually filter the sectors.

For searching by a URL, the user copies in a link to a product and then the program should show all products which are similar. The easiest way to do this would simply be to show all products in the same sector, and I think this would an effective suggestion technique. The user could also have the option as to what level of sector the search is in, e.g. if the product searched is in the sector ‘Dairy, eggs & chilled > Dairy and eggs > Cheese > Soft and cottage cheese’, the user could select whether to show products in ‘Soft and cottage cheese’, ‘Cheese’, ‘Dairy and eggs’, or ‘Dairy, eggs & chilled’, depending on how much of a change the user is willing to make

### User interface

Next I need to design the user interface and in what format the data will be accessed and designed. I initially came up with two main designs, which are shown below, and then got feedback from my primary stakeholder to analyse the pros and cons

#### Design 1:

A picture containing text, whiteboard

Description automatically generated

* This is the home page, a simple search bar is in the centre of the screen
* The user has two options, to search for a key term (‘tomatoes’), for which the interface should then show all products with this term, or search for a URL, to which the interface should show all similar products
* This search is shown in the second page below
* There is a lot of empty space to make the program simple and easy to use, but this space could also later be used for extra features, for example an ‘about’ page or space for advertising

Table

Description automatically generated

* This is the search page, all the products in the search are shown in a table with the relevant information
* Initially, I think the crucial details are the name, customer rating (the number of reviews is in brackets), price, mass, and emissions
* There is also the possibility to add extra ‘information’ sections when the user hovers over a column, for example if they hover over the column heading ‘rating’, it might say ‘The customer rating out of 5 stars, the number of reviews is shown in brackets’
* The search should initially sort by emissions in ascending order to show the most environmentally-friendly items first, but the user clicks a column heading it should then sort by that column. If the column already being sorted by is clicked again it should reverse the direction of the sort

#### Design 2:

A piece of paper with writing on it

Description automatically generated with medium confidence

* In this setup, the user either puts URLs to two products in the two search bars, or searches for one product and the interface fills the second with the lowest-emissions similar product, and then the interface shows a visual comparison between the two products
* This design has space to store more information such as the product image and description, as well as a visual indicator of emissions
* Only 2 items can be shown at a time

I showed both my designs to my stakeholder, and he gave the following feedback:

*“I really like both of these designs, they’re very clear and easy to visualise the data, however there are big differences between the two and to me there are very clear pros and cons to each.*

*Firstly, the most obvious difference is that the first design shows a wide range of products while the second can only show two. I think this is the crucial factor, it would be really important to me to be offered a range of products with their emissions data rather than simply having the lowest emissions product suggested to me, as I might dislike this product, or it might be really expensive.*

*One of the things I really like about the second design is the extra space to store the image and the visuals at the bottom, but I don’t think this outweighs the advantage of having multiple products. However, I think some aspect of this could be incorporated into the table (e.g. with coloured text to show emissions?)*

*Another advantage is that I think the ‘Emissions per kg’ column is really useful, otherwise the emissions statistics are meaningless if the products are different sizes. If this could be incorporated into the first design that would be great”*

From this I decided to go with broadly the first design, but I will also try to add an ‘emissions per kg’ column and change the colour of the emissions text to clearly indicate the various levels. Using ‘emissions per kg’ as the main metric also probably makes the ‘equivalent to x minutes drive’ redundant since the ‘emissions per kg’ is not an actual amount of Carbon Dioxide being produced. One other thing I think I need to consider is units: I think it would be beneficial to have the unit in each cell in the ‘price’ and ‘mass’, since many products will change between ‘£’ and ‘p’, and ‘kg’ and ‘g’, but the emissions units should be in the column heading to save space in the table.

My interface will be created in the Tkinter Python module, which enables the creates of frames containing widgets positioned in various ways.

# Development

Now that the various sections of my product have been designed and planned, I can begin on the various programs I need to write. My first step is to write the program to populate the sectors database.

## Create sectors database

First, I had to create the sectors database. Since I am starting with only the Sainsbury’s website, this initial database will only store sectors from Sainsbury’s so I will call it **sainsburys\_sectors**. By creating a separate table for an individual website, I can also omit the ‘website’ field

I created it using an SQL query:

Text

Description automatically generated

* **uuid** stores the unique id of a sector, this is automatically filled using the DEFAULT and uuid\_generate\_v4 commands to produce a unique 128-bit-label for each record
* The ‘varchar’ type is used for strings

## Populate sectors database

Next, I wrote the program to populate **sainsburys\_sectors** using data from the Sainsbury’s website



* Lines 1-2 imports required modules
* **psycopg2** is an SQL database adapter for python, allowing me to send queries and receive data from the Heroku database
* **pandas** is a library for storing databases as Dataframes in Python. Used to store the database in the program before uploading to Heroku

Text

Description automatically generated

* **postgres\_connect** is a function to connect to the Heroku database and return any errors that are encountered
* lines 5-8 contain my Heroku account details so are blocked out. This information will not be visible for 2 reasons. Firstly, the data will be uploaded to the database privately, and only another script which accesses the database will be distributed. Furthermore, the source code will not be distributed so even that code will not be available
* line 12-15 try to connect to Heroku and returns the connection, if there is an error this is handled in lines 16-20

Text

Description automatically generated

* **postgres\_execute** runs an SQL query on the database
* Lines 27-29 create a connection with the database and run the query
* Lines 30-33 get the result of the query and format it as a pandas DataFrame
* In line 36, if there is no result the result is set to an empty DataFrame
* Lines 37-40 commit any updates to the database, close the connections and return the result

Text

Description automatically generated

* **heroku\_upload** uploads a DataFrame to the sectors database on Heroku into **sainsburys\_sectors**
* It runs an SQL insert query, using text formatting to insert the values in the DataFrame into the values in the query
* Lines 58-61 create a connection and commit the upload
* Once I had written these functions, I realised that I would need the same functions again later for updating the products database and running the GUI, so in order to minimise code duplication I moved these functions to a file called Heroku\_functions.py. Then I could import these functions from the file and begin to write the rest of the code in a new file:

Text

Description automatically generated

* Lines 1-6 import required modules
* **requests** and **bs4** are used to parse information from the Sainsbury’s website to collect the sectors
* **pandas** is used to store the database before uploading to Heroku
* **Heroku\_functions** is the file storing **postgres\_execute** and **postgres\_connect**. Although I later realised postgres\_execute was not actually used for this program since no data is retrieved from a database
* **datetime** and **time** are used to get the current date and time to fill the **date\_updated** column of the database

Text

Description automatically generated

* I then also realised that **heroku\_upload** had to be in the file itself since it contains the name of the table and fields to be inserted, so must be different for each file
* I also removed the parameter and replaced references to the parameter with **leaf\_sectors** (defined below) since it is a global variable

Text

Description automatically generated

* This section uses the BeautifulSoup library to get the HTML from the Sainsbury’s homepage and sets up the key data structures
* **bad\_urls** is initially an empty list and will be a list of any URLs which produce an error during execution of the code. I can then output these once the code has run so I can manually check the causes of these errors
* **soup** will store the soup object containing the HTML of the page
* Lines 35 and 36 use the **soup.find** function to get all top level sectors on the Sainsbury’s page and store a list of their name and link into **all\_sectors**
* **leaf\_sectors** is a pandas DataFrame that will store all the details about any leaf sectors (sectors in the tree which have no children). E.g. the ‘Meat & fish’ sector is not a leaf node since it has child sectors: ‘Meat & fish>Chicken’, ‘Meat & fish>Lamb’, ‘Meat & fish>Beef’ etc., while ‘Meat & fish>Chicken>Coooked chicken’ is a leaf sector since it has no sub-sectors)

Text

Description automatically generated

* The Sainsbury’s website’s tree is organised into 3 layers: departments, aisles, and shelves. So in line 39 I loop through each of these layers, each time finding all leaf nodes and adding any non-leaf nodes to all\_sectors. This will form a **breadth-first traversal** to find all leaf nodes in the tree. The various layers act as the recursion depth for the traversal algorithm
* Lines 40-41 print updates on the progress of the program so I can track how long it will take and spot any errors as the program is run
* **new\_sectors** stores the children of all the nodes in the current layer. For example when looking through departments, all the aisles within these departments will be stored in new\_sectors, and then once all departments have been looked through, all\_sectors will be replaced with new\_sectors to form the new queue for the next layer
* Line 43 loops through each sector in the queue, within the loop the sector will be checked to see if it is a leaf node, or if not its children will be found
* Line 44 prints the path of the sector to help track execution progress
* In line 45 **URL** is set to the url of the current sector being checked

Text

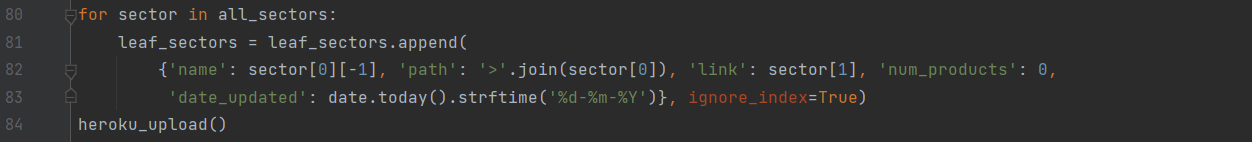
Description automatically generated

* Lines 53-55 try to access the given URL and store the page in **page**. Initially line 54 was the only line in this section but in my initial testing, the site occasionally shut down while I was accessing it, causing an error to be thrown
* Because of this, I added a try-except clause in a while loop to catch any error and reboot the connection up to 3 times for each URL before giving up and setting **bad\_url** to True

Text

Description automatically generated

* If the URL was accessed, then **bad\_url** will be False so the condition in line 66 will be satisfied and the rest of this section executed
* Line 67 gets the HTML of page of the current sector to be checked and stores into **soup**
* Line 68 gets all the children of the current sector and stores them into **new**
* If children are found, then a for loop is used to loop through all the children and append them to **new\_sectors**
* If no children are found, the sector is added to the **leaf\_sectors** DataFrame with the relevant information:
  + **sector[0]** contains the path of the current sector in list form e.g. [‘Meat & Fish’, ‘Chicken’, ‘Cooked chicken’]
  + **sector[0][-1]** therefore contains the name of the current sector e.g. ‘Cooked chicken’, so this is the value of the ‘name’ field
  + The value of the ‘path’ field is set to the list sector[0], joined by a ‘>’ character e.g. ‘Meat&Fish>Chicken>Cooked chicken’
  + **sector[1]** contains the URL of the current sector, so this is the value of the ‘link’ field
  + ‘num\_products’ is set to its default value of 0, this will contain the number of products in the sectors, but we do not know this information until we run the next program I will write
  + ‘date\_updated’ is set to the current date in dd-mm-yyyy form using the date module in python
  + The ignore\_index=True parameter stops pandas automatically adding a numerical id to the dataframe, since the path is a unique key within the program, and the database on Heroku contains an automatically generated uuid
* The program waits for 1 second to reduce the chance of the website closing the connection
* all\_sectors is set to new\_sectors as the new queue once the traversal of that layer has been completed



* The final layer (shelves) will never be traversed since they are all leaf nodes, so all of the shelves are added to leaf\_sectors in the same way as sectors were appended previously
* The data is uploaded to the Heroku database

Text

Description automatically generated

* Here you can see the start of the execution of the program as it goes through the departments and begins to look through aisles

However, a few URLs were causing issues, and I was able to track using the bad\_urls list that this was caused by some inconsistencies on the website where some links were formatted without the ‘sainsburys.co.uk’ part, so I added the following code to correct this error:



I then ran the code, and the data was uploaded to Heroku, as can be seen below:

Graphical user interface, text, application

Description automatically generated

This program clearly needs to be regularly run to update the sectors when they change, especially since there are frequently season-specific sectors on the website. However, if I ran this program again at the moment, all sectors found would simply be appended onto the end of the database, resulting in data duplication for any common sectors. In order to solve this, I added a section of code to run an SQL query to find out whether that sector was already in the database for each sector, so the following section:Text

Description automatically generated

was replaced by:

Text

Description automatically generated

* Lines 85-86 are an SQL query to get all the sectors with the same path as the current sector which are already in the database and stores them in **result**
* If the length of result is not 0, then this sector is already in the database and so the sector is appended to **leaf\_sectors**
* At the end of the program, leaf\_sectors is uploaded to Heroku as before, although now it only contains new sectors

This program worked and only updated new sectors, but I later realised there were 2 issues with this approach:

1. Any old sectors which are not found when the code is run but are in the database should be deleted, as they no longer exist. Later, once the products database is filled, it should also delete any products from that database which are in the old sector
2. This requires running an SQL query every time a leaf sector is encountered, roughly 1000 times each time the program is run, and this is very time consuming

In order to solve both of these issues, I instead ran a single query to find all the paths at the start, compared each sector to this initial result as it was encountered, and then deleted any sectors in the initial result not in the new database. This way I could run just one query at the start of the program and then three queries at the end for each old sector to be deleted (one to find the information about that sector, one to delete its products, and one to delete the sector), rather than one for every sector found. If there were 2 old sectors which had been removed since the last time the code was run (roughly average if the code was run weekly), then this would reduce the total number of queries from 1000 to 1 for checking new sectors while only adding 6 queries for deleting old sectors. The code added was as follows:

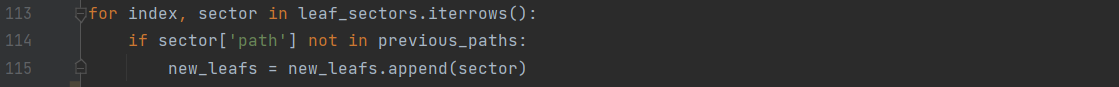


* Line 91 is an SQL query to get the paths of all the sectors already in the database
* In line 92, the query is run, converted to a list, and stored in **previous\_paths**

Text

Description automatically generated

* The purpose of this section is to deal with any old sectors which have been removed
* Line 100 loops through all the old sectors
* Line 101 checks if the old sector is not a new sector
* Lines 102-103 are an SQL query to get the uuid of the given sector, this is needed to identify the products which are in this sector for the query in 109-119, since the products database only contains the field **sector\_id**, and no other information about the sector
* Line 104 runs the query, converts it to a list and stores in **result**
* Lines 105-106 are an SQL query to delete the redundant sector. The query is run is the following line, with the ‘\_’ character used as a temporary variable to store the empty output
* Lines 109-110 are an SQL query to delete all the products from the products database which are in the redundant sector. The query is then run in line 111. Lines 108-111 were clearly added later since the products database is created and populated below
* The for loop in 108 should not usually be required and should be able to be replaced with ‘sector\_id = result[0]’ in any normal cases, since result should only have length 1. However, the reason I first added this section was because I ran the code for a second time and all the sectors were re-added to the database leading to duplication of data. This for loop ensures that if there are any duplicates, they are all dealt with appropriately



* The purpose of this section is to deal with any new sectors which were not in the old database
* Line 113 loops through all the sectors that were found, using the **.iterrows** function to loop through the rows in a pandas DataFrame
* Line 114 checks if the sector is already in the database
* If not, line 115 appends the sector to a new DataFrame called **new\_leafs**. This DataFrame also had to be defined earlier on in the same was as **leaf\_sectors** was. I also had to update the **heroku\_upload** function to make it upload data from ‘new\_leafs’ rather than ‘leaf\_sectors’

## Create products database

I then used another SQL query to create **sainsburys\_products**:

Text

Description automatically generated

* **uuid** is automatically generated as in the sectors database
* The ‘varchar’ type is used for strings
* Some data such as ‘religious\_info’ and ‘nutrition\_100’ are actually lists or dictionaries, but the easiest way for me to store them in my database is to store them in string format (e.g. ‘[1,2,3]’ or “{‘a’:1,’b’:2,’c’:3}”
* It would be preferable for normalisation to store this information in separate databases, but since I have a limit on the number of rows available in Heroku, this method of storing lists and dictionaries in strings saves space in the database

## Populate products database

Then I wrote the script to populate **sainsburys\_products**:

This script was a little more complicated in terms of the web scraping involved, for 2 reasons. Firstly, a lot of data needs to be collected from each page, and a lot of this is either inconsistent on the website, contains mistakes, or is formatted in ways meant for people rather than algorithms to understand, so one of the biggest obstacles in this project was collecting as much accurate data as possible. Secondly, I initially had some issues with the JavaScript on the page and with accepting cookies, so I had to supplement the ‘requests’ python package with **selenium**, a web driver which opens up chrome tabs from within python to enable you to scrape web data more effectively. The code was as follows:

**Text

Description automatically generated**

* First, I had to import the relevant libraries and define some useful constants
* **requests**, **bs4** and all the relevant **selenium** packages are used for web scraping
* **pandas** is used for creating and storing databases as DataFrames in Python
* **Heroku\_functions** is the file I created before containing functions I wrote to interact with the database on Heroku
* **datetime** and **time** are used to get the current date and time
* **re** is a module for carrying out searches with regular expressions, which I will use for finding relevant data in sections of text on the website
* **units** stores the English-language terms for the first 20 non-zero integers, while **tens** stores the terms for the tens digits in English. These will later be used for converting quantities written in words into number format

Text

Description automatically generated

* The **initialise\_chrome** function sets up a chrome browser from within the code, returning the driver object which will later be used to get data
* **chrome\_options.headless** can be set to False to show the browser to the user while the program is being run, or True to hide the window. This will eventually always be True but viewing the browser is useful for testing, debugging and progress tracking
* **ChromeDriverManager.install** makes sure the relevant chrome drivers are installed on the machine
* Line 31 creates a driver object and stores it in **new\_driver**
* Lines 32-33 maximise the window if the driver is not headless (browser is shown)
* The driver object is returned

Text

Description automatically generated

* **accept\_cookies** takes a driver object and attempt to accepts cookies on the page
* Line 38 finds all the elements on the page containing the word ‘Accept’
* Lines 40-45 go through each element found and attempt to click ‘enter’ while selecting that element. If this is the ‘Accept cookies’ button, the cookies will be accepted, and the program can continue. If not, an error will be thrown, and the program will move on to the next
* If none are found, this means the page has not asked for cookies to be accepted and so the rest of the program should still work as expected
* In line 46, the program waits for half a second to give time for the full page to load once cookies have been accepted

Text

Description automatically generated



* I then copied the **heroku\_upload** function from the code to populate sectors, but replacing the column names in **insert\_query** to be the fields in the **sainsburys\_sectors** database

Text

Description automatically generated

* **query\_text** stores an SQL query to get all the sectors in the sectors database
* The sectors are sorted by the ‘date\_updated’ column so that the ones which have been updated the longest ago get updated, so eventually the program cycles through all the sectors over a period of time
* It also then sorts by ‘num\_products’, so that once all the sectors have been cycled through, the biggest ones are updated again first
* **num\_sectors** is the number of sectors which the program will find the products for at a time. This program will be run regularly (e.g. hourly), so with the roughly 1000 sectors the full list of sectors would be updated around once every 8 days

Text

Description automatically generated

* I then define the DataFrame **new\_products** with the appropriate column names for the products database. This will store the products that are found for a given sector, which will then be uploaded to Heroku at the end of each sector

Text

Description automatically generated

* Line 117 loops through some of the sectors which were returned from the query. Normally the number of sectors which are looped through will be **num\_sectors**, but I use the min function so that if the total number of sectors returned from the query is less than num\_sectors, it will not cause an error and will instead simply loop through all the sectors in the result
* The rest of this section scrapes the homepage of a given sector and looks for the links of all the products in this sector
* For example, here is what a sector home page might look like:

Graphical user interface, application

Description automatically generated

The page continues down to show 60 items per page, and all the products in this sector are spread across 3 pages, as you can see in the box highlighted in green which shows the menu to select the page number. I need to get the links of all 161 products across all 3 pages

* Line 120 clears the **new\_products** DataFrame while keeping the column names, this is cleared for each sector and then uploaded to the database before moving on to the next sector
* My for loop actually only loops through the indexes of the sectors I want to examine, so line 121, stores the ith row in **result** into **sector**
* **URL** then stores the URL of the given sector by accessing the ‘link’ field from the DataFrame
* As with the program to get the sectors, I added lines 123-124 in case any links were formatted without the ‘https://ww.sainsburys.co.uk/’ part
* Lines 125-126 use the ‘requests’ and ‘bs4’ modules to get information from the page and store the HTML in **page\_soup**
* **pages\_left** stores whether or not there are any more pages of items in this sector that haven’t yet been scraped, this is initially set to True since the first page has not yet been scraped
* **all\_products** is a list to store the links to all the products in this sector. It is initially set to an empty list
* As long as there are pages which haven’t been scraped, line 130 uses the **soup.find\_all** function to find all elements in the HTML of type ‘div’ with class ‘productNameAndPromotions’. I had to inspect the HTML to find that this was the element that defined each item on the page. Then within this element the button to the link to the page of that product was inside another <h3> tag and an <a> tag, so I use a list comprehension to get the links from each of these items. This list is then added to **all\_products**
* In lines 131-134 I look for a ‘next page’ button on the page, and if there isn’t one then pages\_left is set to False. My initial inspection of a page showed that this button was inside an <a> tag inside an <li> tag with the class ‘next’, so I initially only had lines 133-134 in this part, but my first tests produced some errors. This turned out to be because once again there was some inconsistencies on the website, and sometimes this <a> tag did not exist. This meant I had to add lines 131-132 to first check if this tag existed, and if not, pages\_left is also set to False. If pages\_left is set to False, the while condition will no longer be satisfied so it will move onto the next section of code
* If the ‘next page’ button is found, then in lines 136 the link stored in the next page button is found and stored in **URL**, and then that page is checked, loaded and the HTML stored in **page\_soup** as before
* Then since pages\_left is still True, the program loops back to the start of the while loop with the new page in page\_soup, and continues to get new products until it runs out of pages

Graphical user interface, text

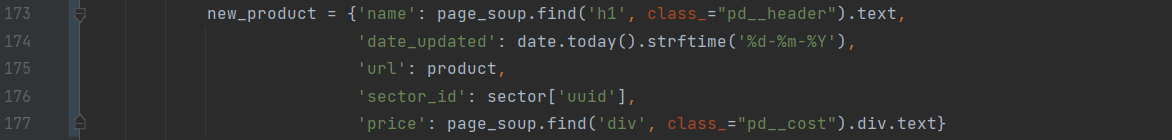
Description automatically generated

* I then check to see if any products were found
* If the length of all\_products is 0, then no products were found and the condition is satisfied, so a query is run to delete the current sector from the sectors database, since it is either empty or no longer exists

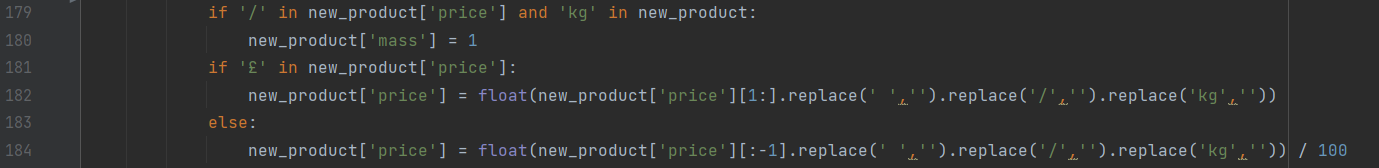
Text

Description automatically generated

* If some products were found, then the next section runs
* In 148 the driver is initialised using the previously written **initialise\_chrome** function
* Then all the products found are looped through in line 150
* The purpose of this section is to load the page of the given product in the webdriver and ensure the page has loaded correctly
* **counter** stores the number of times the driver has attempted to load the page, **found** stores whether or not any elements have been found on the page, i.e. whether the page has loaded yet
* In lines 155-157, if the driver has already tried to load the page twice, the program reboots the driver
* In lines 159-163 the driver attempts to load up the given page, and waits for either 10 seconds or until a ‘header’ object has loaded, whichever comes first
* If it takes more than 10 seconds then an error is raised, the counter is incremented, and it loops again
* Line 169 then tries to accept cookies using the previously written function
* Lines 170-171 get the HTML of the page from the driver and stores it in **page\_soup**



* **new\_product** is a dictionary to store all the information about the given product, to later be stored in the products database
* Here I initially set some key data which, and the rest of the program will fill in the remaining information
* Line 173 finds the header with class ‘pd\_\_header’, which represents the name of the product, and uses .text to get the name in a string format to store as the ‘name’ field
* Line 174 sets the ‘date\_updated’ field to the current date in dd/mm/yyyy format
* Line 175 sets the ‘url’ field to ‘product’, which currently stores the URL of the page
* Line 176 sets the ‘sector\_id’ field to the uuid of the current sector
* Line 177 finds the ‘div’ element with the class ‘pd\_\_cost’, which is the element containing the price of the item, and stores the price into the ‘price’ field



* In the last section I set the ‘price’ field, but it is currently in a string format (‘£1.50’ or ‘99p’) and I want to convert it to a number in pounds. To do this, line 181 looks to see if there is a ‘£’ symbol in the price, and if so, removes the ‘£’ at the start (by using the slice [1:]), and removes any spaces, slashes, or the text ‘kg’ (this is for if the price is stated as £2/kg for example)
* If ‘£’ is not in the price, then that means the price is in ‘99p’ format, so line 184 removes the ‘p’ at the end (by using the slice [:-1]), and removes spaces, slashes, and the text ‘kg’ as before. This time since the price will be in pennies, I also divide by 100 to get the price in pounds
* Sometimes for objects with changeable masses such as loose fruits, the price can be listed as a price per kg, if this is the case, I set the mass to be 1kg in line 180 and then the price calculations are carried out as before



* This section gets the customer rating of the product
* The rating is shown on the website as a number of stars, as shown in Figure 5 below:

Text

Description automatically generated with medium confidence

Figure - example title and reviews display

* To get the rating for the product, I use the find\_all function to search for all ‘path’ elements with the class ‘star-rating-icon—full’, which indicate a filled in star. The number of filled in stars which are found is the rating out of 5, so the length of this list of found stars is stored into the ‘rating’ field

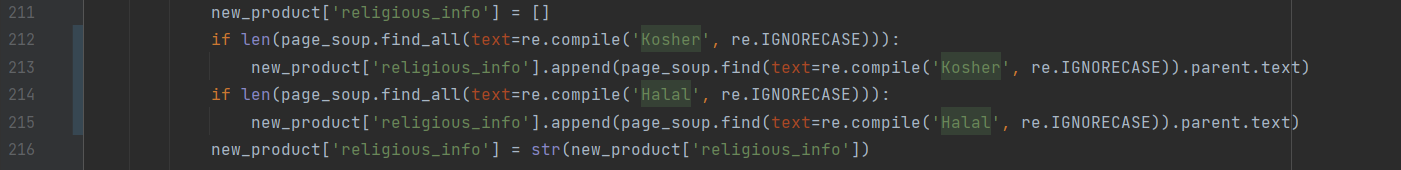


* This section gets the number of reviews on the given product, which can be seen as the number in brackets in Figure 5, (20). Line 190 tries to find the relevant text: ‘Read reviews (20)’ by searching for a <span> tag with class ‘pd\_\_reviews\_\_read’, then splitting the text by the ‘(‘ character to give ’20)’, before using the slice [:-1] to remove the final close bracket
* If there are no reviews on the product this element does not exist and so an error will be thrown when trying to access the text, this is caught in the except clause and in this case the ‘num\_reviews’ field is set to a default value of 0

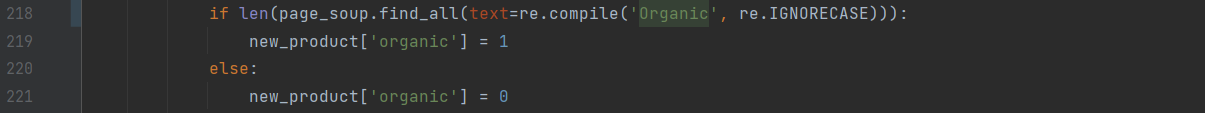
Graphical user interface, text, chat or text message

Description automatically generated

* This section tries to determine if the product is specifically suitable for vegetarians or vegans
* Lines 202-204 use a regular expression search to look for any occurrences of the terms ‘suitable for vegans’, or ‘vegan’, with the re.IGNORECASE parameter to ignore the case of the text. If this is found, then the ‘vegan’ and ‘vegetarian’ fields are both set to 1
* If this is not the case, then lines 205-207 carry out the same process but looking for terms for vegetarian instead of vegan. If a valid term is found, then the ‘vegetarian’ field is set to 1, but the ‘vegan’ field to 0
* However, when I first tested this section of code, I realised that the term ‘suitable for vegetarians’ was found when the term ‘not suitable for vegetarians’ was in the text, and likewise for vegan, so I had to add lines 196-201 to first check for this case and then have the previous section in an ‘elif’ statement
* If none of the terms are found, it is assumed that the product is neither vegetarian nor vegan



* This section gets some religious information for the product
* **new\_product[‘religious\_info’]** is a list to store the religious information on the product
* This section works in a similar way to the previous section, using regular expressions to search for the ‘Kosher’ and ‘Halal’ key terms in lines 212 and 214 respectively
* If the terms are found, I want to get more specific information. For example, this information might be written as ‘Kosher – KLBD, not for Passover use’, and I want to store all this information
* To do this, lines 213 and 215 find the keyword, and then use .parent.text to get the full text of that line, which is then appended to new\_product[‘religious\_info’]
* new\_product[‘religious\_info’] is then converted to a string, as discussed before, so the list [‘Kosher’, ‘Halal’] becomes the string “[‘Kosher’, ‘Halal’]”



* A similar process is then carried out to see if the process is organic, and information stored in the ‘organic’ field, with 1 for True and 0 for False

Text

Description automatically generated

* This section gets the description of the given products, an example is shown below:

Graphical user interface, application, Teams

Description automatically generated

* The description often is stored in the website in multiple <div> tags, so I need to find all of these tags and join them together
* **product\_description** is a list to store all the sections of the description to be found
* Line 224 finds the header tag which contains the text ‘Description’, using regex to ignore case, and stores it in **description\_heading**
* **flag** indicates whether all the description has been found
* Line 228-235 tries to set **description\_heading** to its next sibling in the tree, i.e. the next tag on the page. If this next sibling is a <div>, then it is still part of the description and so is appended to product\_description. If the next sibling is not a <div>, then flag is set to False
* If no next sibling is found at any point, this will cause an error which is caught in the except clause, which then sets the flag to true
* The code loops until flag is True, so keeps adding next tags to product\_description until flag is True
* Line 236 joins the lines in product\_description with a ‘\n’ character, which indicates a new line, and stores it in the ‘description’ field of new\_product

My next challenge was to get the nutritional information of the given product. This proved to be an issue in my initial research and experimentation due to the inconsistincies in the formatting of the website. For example, below are 3 different nutrition tables for various products (Fig 6-8), which are formatted in very different ways. One way in which they differ is the name of the nutrition per 100g column, which is ‘100g contains’, ‘Per 100g (grilled)’, and ‘Per 100ml’ in the 3 tables. Not only is the unit different in some cases, but also the words in the title (contains, grilled, per) vary. In order to cope with this I will have to use regular expressions to find this column. Furthermore, in the Figures 6 and 7 the energy in kJ and in kcal are stored in separate rows while they are in the same row in Fig 8, which could lead to some issues. In other products, I also found a lot of missing data and other formats for the column name in other products, such as ‘/100g’.

Table

Description automatically generated

Figure - nutrition table example 1

A picture containing table

Description automatically generated

Figure - nutrition table example 2

Table

Description automatically generated

Figure - nutrition table example 3

I then wrote some code to try to get the nutrition information from a product

Text

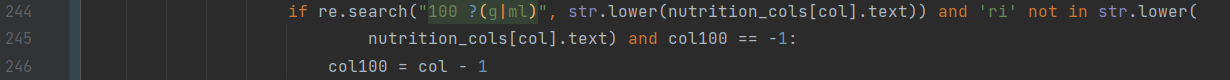
Description automatically generated

* In line 238, the ‘nutrition\_100’ field is initially set to an empty dictionary, this will store the nutrition information per 100g, for example it could be {‘Energy’:49,’Fat’:1.7,’Of which saturates’:1.0,…}
* **col100** stores the index of the ‘per 100g’ column in the nutrition table, for example, in the 3rd picture above, this should eventually be 0 since the ‘Per 100ml’ column is the column with index 1. In line 230 this is set to a default -1
* Lines 241-245 try to find the column which contains the nutritional values per 100g
* Line 242 finds the nutrition table by searching for a <table> tag with class ‘nutritionTable’, and then finds all <th> tags in this table. The <th> tags represent the column headers, and are stored in the list **nutrition\_cols**
* Line 243 loops through the columns in the tables, ignoring the first column since this contains the row names
* Line 244 uses a regular expression to see if the given column is the nutrition per 100g column. It searches for the expression “100 ?(g|ml)”, which means that the string should have the number 100, followed by zero or one spaces, and then either ‘g’ or ‘ml’, somewhere in the column title. For example, it should accept ‘100g’, ‘Per 100g’ or ‘/100ml’, but not ‘/10g’ or ‘Per 200g serving’
* Line 245 sets col100 to the current index subtract 1, the subtract 1 is to account for the row title column

I tested this section of code on some products, and I noticed some erroneous data. This turned out to be due to the fact that some columns representing % of Reference Intake would satisfy my regular expression, for example a column called ‘%RI per 100g’, so I added a condition that the term ‘RI’ was not in the column name. However, it’s possible that a product I haven’t looked at would have another name for this column (e.g., ‘intake per 100g’, ‘% daily guide’), so to try to avoid this issue from happening again, I also changed the code so that the program uses the first column which matches the expressions rather than the last, so whereas before if there were 2 columns called ‘Per 100g’ and ‘%intake per 100g’, since both match the expression the second column would be selected, now it should select the first column. To do this, this section:



was replaced with:



Now that I have the index of the relevant column, I need to get all the data from this column

Text

Description automatically generated

* Line 249 checks to see if **col100** has been changed from its default value i.e. if the correct column was found
* Line 250 finds the table by searching for a <table> tag with class ‘nutritionTable’, and then gets all the rows in the table by finding all <tr> tags in the table
* Line 251 loops through all the rows in the table, using the slice [1:] to ignore the first row since this is the column names
* Lines 252-255 try to add some nutrition information to the dictionary **new\_product[‘nutrition\_100’]**. For example, for the row shown below, it would try to add {‘Fat’:1.7}



* In line 253, ‘row.th.text’ finds the row name (e.g. ‘Fat’) to use as the key, and the value is found by finding the cell in that row with column index ‘col100’, removing the ‘g’ at the end, and converting to a float
* This part is in a try-except clause, so if any data the error is caught, and the program can continue
* This whole section is also in another try-except clause, with the except part in line 257, where the error is printed for debugging purposes
* Line 258 converts the dictionary to string format, for example the dictionary {‘Energy’:49, ‘Fat’:1.7,…} to the string “{‘Energy’:49, ‘Fat’:1.7,…}”

However, when I tried to test this section by running the code on a sample of products, a lot of my data was either missing or incorrectly formatted, and this was due to the inconsistencies in the formatting of the ‘energy’ rows in different products. Some products had the energy in kcal and kJ stored in separate rows, some had separate rows which were merged into one title, and some were stored in one row, for example with the value ‘100kcal/418kJ’. I also am only interested in the energy in calories. To solve this, I tried to handle the energy rows separately by searching for relevant terms, and then get the rest of the information with the previous method. The new code was as follows:

Text

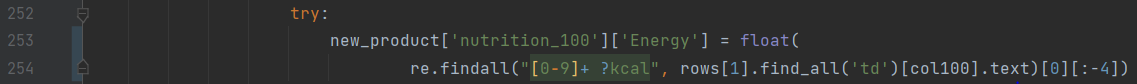
Description automatically generated

* As before, line 251 gets all the rows in the table
* Line 253 gets the data in the 1st row and tries to split it by the text ‘kcal’. If the value contains the text ‘kcal’ (e.g. ‘100kcal’ or ‘100kcal/418kJ’), then the text will be split and the section before ‘kcal’ kept (both of the above would become ‘100’). This is then converted to a float and stored in the ‘Energy’ field
* If ‘kcal’ is not in the text, and error is thrown and the except clause is run, this carries out the same function but on the 0th row
* Again, if ‘kcal’ is not in this text, an error is thrown and the next except clause is run. This simply gets all the data in the first row and removes any spaces, new lines, or slashes. This would get the correct data if the energy information was stored in the table without units
* If this also produces an error, this means the cell in the table is empty, and so I simply use the pass command to move on, leaving the ‘nutrition\_100’ empty for now
* As before, lines 263-267 get the data for the rest of the rows, but now use the slice [2:] to ignore the energy rows

This section now works on a sample of products I selected, getting all the relevant nutritional information, but I realised that I could make the search more consistent for any future products with slightly different formatting by using regular expressions, so I replaced the lines:



With:



* Line 254 searches for the regular expression ‘[0-9]+ ?kcal’, meaning any number followed by any number of spaces followed by ‘kcal’. This means that ‘100 kcal’ or ‘contains 100kcal’ would be found and the correct value produced where they wouldn’t have been before
* The slice [:-4] is then used on any text which matches the expression to remove the ‘kcal’ at the end, and then converted to a float and stored in the ‘Energy’ field as before
* The same swap is done for all other similar lines

My next step was getting the number of servings in a product. The challenge here was that this information can be stored in a few different places and formats on the page, for example in Figure 9, the servings information is in the nutrition table in its own row, while in Figure 10 it is below the table in a separate section of text. The number of servings is also often stated in words as well as in numerals

Graphical user interface

Description automatically generated with low confidence

Figure - servings example 1

Table

Description automatically generated

Figure - servings example 2

To cope with these inconsistencies, I will use a regular expression to search the entire page for relevant information, and deal with a few special cases separately:

Text

Description automatically generated

* Line 285 sets the ‘pack\_servings’ field to a default value of 0
* Line 286 searches for the regular expression ‘Contains .+ servings’. This means looking for any section of text with ‘Contains ’ at the start and ‘ servings’ at the end, so it would find ‘Contains 10 servings’, ‘Contains three servings’, ‘Contains 4 300g servings’ and ‘Contains approx. 5 servings’
* If one such expression is found, line 288 splits the text by the text ‘contains ’ and ‘servings’ to get the information in the middle, then splits the text by spaces and takes the first word. So, for the first three examples above it would give ‘10’, ‘three’ and ‘4’
* However, for the 4th example it would give ‘approx.’, and this is quite a regular formatting, so I also removed the words ‘approx.’ and ‘approximately’ before splitting the text into words. Now the 4th example should give ‘5’
* Line 291 attempts to convert the data into an integer, so the first, third and fourth examples above would give 10, 4 and 5 respectively, while the third would give an error
* The error caused by the number of servings being in words is caught in line 292, then lines 294-298 try to convert the word into a number:
* Line 294 checks if there is a dash in the number (e.g., ‘thirty-two’), and if so, splits the number into two parts (‘thirty’ and ‘two’)
* Line 296 uses the lists ‘tens’ and ‘units’ (defined earlier to contain the English language words for tens and units numbers, tens = [‘ten’, ‘twenty’, ‘thirty’,…], units = [‘one’, ‘two’,…]) and Python’s ‘.index’ function to convert each part into a number, so ‘thirty-two’ became ‘thirty’ and ‘two’, and now becomes 3 and 2. The tens digit is then multiplied by 10 and the numbers added together
* If there is no dash in the number, then it is a single digit number, so it simply uses the ‘units’ list to convert it to a number
* If an error is caused during this process, most likely due to missing data, this is caught in line 299 so the program can continue. This happened quite a lot since for some reason a lot of products had ‘Contains servings’ written on the page, where the number of servings were unknown

When I ran the program, all the data produced was correct, but quite a lot was missing, and I noticed that a lot of these missing data cases were products where the product was divided up into distinct portions (for example a 6-pack of crisps), and the number of servings was implied by a having ‘x6’ in the product name so was not explicitly stated elsewhere. To add these cases I added the following code:

Text

Description automatically generated

* Line 301 checks if the ‘pack\_servings’ field has already been filled, so this section only runs if there is not already serving information
* Lines 303-304 search in the product title for the regular expression ‘ x[0-9]+’, meaning a space followed by an ‘x’ and then a number. If this is found, the slice ‘[2:]’ is used to remove the space and the ‘x’ at the start, and then it is converted to an integer
* Lines 307-308 carrying out the same process, but just looking for a number in the title, so that if the title was ‘Packet of 6 Kit-Kats’, the servings would be 6

I then used the same process to find the mass of the product, by searching for the regular expression " [0-9.]+(kg|l)". This finds a space followed by any number (the dot is included to allow decimals) followed by either ‘kg’ or ‘l’. If this is not found, I also try the regular expression " [0-9.]+(g|ml)" and divide the result by 1000

Next I had to find the ingredients of the product. From looking on some product pages I found that this information is stored in its own section either in an <h3> tag with the text ‘Ingredients’ or in a <strong> tag with text ‘INGREDIENTS:’ (Fig 11), so I will search for both of these in the page and if found, get all the text in the section, split it by commas and store the resulting list in the ‘ingredients’ field:

Text

Description automatically generated with medium confidence

Figure - example ingredients information

Text

Description automatically generated

I then used the exact same method to find the allergens, but only getting the ingredients in bold, which can be determined by the filter “style=’font-weight: bold;’”

I then had to find the health information, cooking instructions, information on country of origin, and packaging information, and store these in the ‘product\_info’, ‘cooking\_info’, ‘origin\_country’ and ‘recycling’ fields, and these were all done using the exact same method as for the description, with the only change being initially searching for the headings ‘Health’, ‘Preparation’, ‘Country of Origin’ and ‘Packaging’ respectively rather than ‘Description’. An example for health is shows below:

Text

Description automatically generated

Finally, all the information in ‘new\_product’ is appended to the DataFrame ‘new\_products’:



Then once all the data is in new\_products, the chrome drive is closed and the data uploaded to Heroku:



Finally, there are a few other queries which need to be run to update relevant information:

Text

Description automatically generated

* This query deletes any old products in the database from the sector which has just been searched, using the condition “date\_updated <> ‘\*\*date\*\*’” in order to not delete the products which were just uploaded

Text

Description automatically generated

* This query updates the relevant information in the ‘sainsburys\_sectors’ database
* The ‘num\_products’ field, which stores the number of products in the sector, is set to the length of ‘new\_products’
* The ‘date\_updated’ field is set to the current date

I ran the code on 5 sectors, and ‘sainsburys\_products’ began to be populated, with almost all of the data being filled in. The only missing information is where that information is not available on the website. The populated database can be seen below:

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, application

Description automatically generated

Graphical user interface, application

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

All the data uploaded as expected, but there were a few issues and improvements that I discovered:

1. Many of the sectors in the database do not contain food products and so are not applicable to my program. For example, one of the sectors which was updated when I ran the program was ‘Homeware and outdoor’. To fix this, I added a new column to ‘sainsburys\_sectors’ called ‘required’, which is set to 1 if that sector is relevant and 0 if not. I gave this column a default value of 1 and then manually set all irrelevant sectors to 0 using SQL queries like the following for various categories:

A screenshot of a computer

Description automatically generated with medium confidence

I then modified the initial query in the ‘get products’ program so that only sectors with required=0 were searched:

Text

Description automatically generated

1. I also realised that if, for example, one sector was updated and all its products retrieved on one day, and then on the next day a new sector was found and added to the sectors database, then this current query would prioritise the full sector over the empty one since the date updated was earlier. To fix this, I added a new column – ‘products\_scraped’, which takes the value 1 if the products have been retrieved from this sector or 0 if not. This is set to a default value of 0. I then modified the query at the end of the ‘get products’ program to update this value to 1 for any sectors which have been updated:

Text

Description automatically generated

* The modification is at the start of line 481

I then also modified the query at the start of the ‘get products’ program to order the sectors by products\_scraped:

Text

Description automatically generated

* The modification is at the start of line 91

1. Since my dates were stored as strings in the ‘dd/mm/yyyy’ format, when the query tried to sort the sectors by ‘date\_updated’, the order in which they were outputted was incorrect. For example, ‘10/06/2000’ would come before ‘20/01/2000’ even though it is later, since it is first alphabetically. To fix this, I changed all the date formats in the program to ‘yyyy/mm/dd’ format so that alphabetical order would be the same as date order
2. Often a product would be missing one of either the ‘pack\_servings’ or ‘mass’ data, but I realised it would usually be possible to calculate one from the other using the nutritional information. For example, in the table below, I can use the ‘100g’ column and the ‘serving’ column to calculate the serving size, in this case 100g since the data is the same in both columns.

Table

Description automatically generated with medium confidence

I can then use the formula “mass = serving size \* number of servings” to calculate one of the mass or number of servings from the other. To do this:

* First I copied the code to find the ‘per 100g’ in order to column to find the ‘serving’ column, swapping the search for the term ‘100g’ with a search for ‘serving’
* Then I got the ‘Energy’ data for both columns and divided the ‘serving’ data by the ‘100g’ data, then multiplied by 0.1kg, to get the serving size in kg, which I stored in **serving\_size**. I chose to use ‘Energy’ since this is the highest value and so produces the most accurate data
* Then once all the information for a product has been retrieved, I use the following code to try to update the information:

A screenshot of a computer

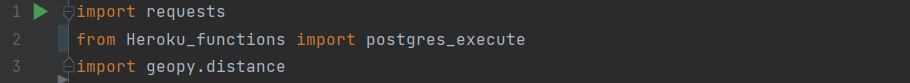
Description automatically generated with medium confidence

I ran the code again and this time more of the information was filled in and the dates were being ordered correctly. If I were to release the project, I would upload this program, along with the ‘get sectors’ program, to Heroku, and use its in-built functionality to schedule the programs to run hourly for ‘get products’ and weekly for ‘get\_sectors’.

My next step is to write a program to calculate the emissions for each product and add this information to the database

## Add emissions

This program will go through each item in the ‘sainsburys\_products’ database, calculating the emissions using the algorithm described in the design section. However, before I write the program, I need somewhere to store the emissions data, so I added a column to ‘sainsburys\_products’ called ‘emissions’, which stores the kg CO2 produced by that product



* **requests** is used to search the **OpenStreetMap** website for a countries coordinates
* **Heroku\_functions** is the file containing the relevant functions for running SQL queries on Heroku
* **geopy.distance** is used to get the geographical distance between two coordinates

A screenshot of a computer

Description automatically generated with medium confidence

* **countries** is a list of all the countries in the world in the format used by OpenStreetMap, it is too long to show in full but the first looks like this:



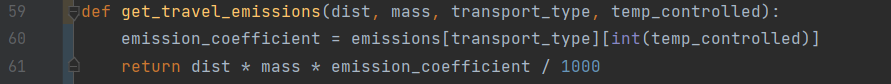
* Since the countries in this list are in very specific formats which will not always be used by the Sainsbury’s website, I created a dictionary called **country\_swaps** of some common alternative names linked to the correct format of that country
* **emissions** is a dictionary containing the emission of various transport types in kg of CO2 produced per kg of food per km travelled. The keys are the transport type and the values are a list of length 2 containing the emissions with and without temperature control
* **air\_food** is a list of foods which are transported by air rather than by sea[[5]](#footnote-5)
* **controlled\_food** is a list of foods which have to be temperature controlled during transport[[6]](#footnote-6)



* **get\_country\_coords** gets the coordinates of a given country using the OpenStreetMap website
* Line 50 creates an OpenStreetMap search URL for the given country, formatted in JSON
* Line 51 gets the JSON output and stores it in **response**
* Line 52 gets the latitude and longitude data, converts them to floats and returns them in a list



* **get\_countries\_distance** gets the geographical distance in km between two coordinates, ‘**a’** and ‘**b’**
* Line 56 uses the **get\_country­\_coords** function on ‘a’ and ‘b’ to get their coordinates, then uses the ‘distance.distance’ function in the geopy module to get the geographical distance between the two sets of coordinates
* This distance is currently a Geopy distance object, so I use convert it to a string
* This string is in the format ’10.000 km’, so I use the slice [:-3] to remove the ‘ km’ at the end, before converting the remaining number to a float



* **get\_travel\_emissions** calculates the emissions caused by transporting a food item
* It takes the following parameters:
  + **dist** is a float of the distance in km the item is travelling
  + **mass** is a float of the mass of the object in kg
  + **transport\_type** is the mode of transport being used (‘sea’, ‘air’)
  + **temp\_controlled** is a Boolean of whether or not the food needs to be temperature controlled in transport
* Line 60 gets the **emission\_coefficient** (the kg CO2 emitted per km travelled per tonne of food) from the **emissions** dictionary defined in line 43. For example, if transport type was ‘sea’, then emissions[transport\_type] would get [0.01, 0.02] from the dictionary
* This denoted the emissions\_coefficient without and with temperature controlling, so then ‘temp\_controlled’ is converted to an integer (True/False to 1/0), and this is used to index the list [0.01, 0.02] to find the relevant emission\_coefficient
* Line 61 calculated the emissions by multiplying the distance, the mass, and the emission coefficient. Since the emission\_coefficient is per tonne of food while the mass is in kg, so I divide by 1000 to get the correct emissions and return this from the function



* The user’s current location is set to ‘united kingdom’ since my product currently only works for UK websites

Text

Description automatically generated

* This query gets all the products in the products database along with the name of their sectors
* Lines 68-69 join the products database with the sectors database on the relevant columns: ‘sector\_id’ in ‘sainsburys\_products’ to ‘uuid’ in ‘sainsburys\_sectors’
* Line 70 makes the query only return products where the emissions have not already been calculated, so the program can update new products without having to calculate old products which have already been calculated
* Line 67 is such that all the information from the products database is returned, as well as just the sector name from the sectors database
* The result is stored in **result**

The next stage is getting the emissions from the production of the item. I downloaded a database from a research project containing the production emissions per kg (Fig 12), but I then had the issue of how to get my program to know which row in the database an item corresponded to. For example, the program needs to know that a product called ‘Heineken beer x6 bottles’ corresponds to ‘BEER IN CAN’.

Table

Description automatically generated

Figure – SuEatable food emissions database

Firstly, I removed all the columns except ‘Food commodity ITEM’ and ‘Carbon Footprint ITEM’ (columns B and C) as these were the only necessary columns. I then had to manually add conditions for each relevant product, for which if a product met this condition it would be classified as that type of food. To do this I added 3 columns to the database:

* ‘Product contains’ contains any strings which must be in the name of the product, for example apple juice must contain the words ‘apple’ and ‘juice’
* ‘Sector contains’ contains any strings which must be in the name of the sector, for example pasta must be in a sector containing the words ‘food cupboard’. This cannot be done via a condition in ‘Product contains’ since if I just had ‘pasta’ in Product contains, it would accept any cooked pasta dishes, while these emissions statistics are only for raw pasta
* ‘Product not contains’ contains any words which cannot be in the product name. For example, without this, ‘Apple juice’ would satisfy the condition for ‘Apple’, while they are clearly different things, so ‘juice’ is in the ‘Product not contains’ column for ‘Apple’
* Often more than one word or phrase needs to be present, and these are separated by commas, while if either of two conditions can be present, these are separated by ‘ or ‘
* A sample of the database is shown below:

Graphical user interface, table

Description automatically generated with medium confidence

I then saved this as a .csv file in the project directory used the following query and code to upload it to a new database in Heroku called ‘**emissions\_stats’**:

Text

Description automatically generated



* First I import pandas for handling Databases and csv files as DataFrames
* I also import the ‘postgres\_connect’ function from my ‘Heroku\_functions’ file to connect to Heroku

Text

Description automatically generated

* I then define ‘heroku\_upload’. This is copied from previous files but with the column names changed to the column names of ‘emissions\_stats’



* I then read the csv file using pandas and store it in **data**. A slice is used to remove blank rows and columns

The database as expected can be seen below:

Graphical user interface, application

Description automatically generated

Then I could continue to write ‘get emissions’ code:

Text

Description automatically generated with medium confidence

* This section gets the links database and stores it in **links**

Text

Description automatically generated

* Line 80 loops through all the products in ‘result’
* The rest of this section converts the string in the ‘origin\_country’ field into a list of countries, as shown in the comment in line 83
* In line 82, ‘transport\_emissions’ is set to a default value of -1
* **product\_countries** will contain the list of countries
* Line 85 gets the data from the ‘origin\_country’ field, removes any commas or newlines, converts to lower case, and then splits it into words
* Line 86 removes any full stops from each word, and uses the strip() function to ensure plain words are left
* As described in the pseudocode in the design section, lines 87-95 loop through the words and use the ‘countries’ lists to convert this list of words into a list of countries

Text

Description automatically generated

* This section calculates the average transport emissions using the ‘product\_countries’ list which was just created
* Line 96 checks if product\_countries contains any countries, if not, the else statement in lines 112-113 is run instead, which assumes the product originated from the United Kingdom, and so the emissions to get to the United Kingdom will be 0
* In line 98, a list comprehension is used on ‘product\_countries’ to run the ‘get\_countries\_distance’ function on each of these countries. This produces a list of the distances between the potential countries of origin and the UK
* This list is then summed and divided by the length of ‘product\_countries’ to get an average distance the product has to travel which is stored in **average\_dist**
* Lines 102-104 use the ‘air\_food’ list to check if the product is transported by air, while lines 105-107 use the ‘controlled\_food’ list to check if the product is temperature controlled during transport
* Line 108 tries to use the ‘get\_travel\_emissions’ function to calculate the travel emissions based on the ‘average\_dist’, the mass of the product, the transport type, and whether or not the item is temperature controlled
* If the mass data is not available, an error will be thrown, which is caught in line 110

Text

Description automatically generated

* This section tries to find a corresponding row in ‘emissions\_stats’ for a given product
* **food\_emissions** will store the total production emissions, and is set to 0 by default
* **match** stores whether a matching row in ‘emissions\_stats’ has been found
* **valid** is initially set to True, but if two rows in ‘emissions\_stats’ produce a match, then the program does not know which row to use for the data and so valid will be set to False
* Line 118 loops through the information in ‘links’, which is the data from ‘emissions\_stats’
* The row is only checked if it is not empty (line 119)
* **match\_row** is whether the product meets the conditions for this row in ‘emissions\_stats’. This is initially True but is set to False if it fails to meet the ‘product\_contains’, ‘sector\_contains’, or ‘product\_not\_contains’ conditions
* Lines 122-128 check the product against the ‘product\_contains’ condition
* Line 124 splits ‘product contains’ by ‘or’, and so the product can meet any of these conditions
* Line 125 uses a list comprehension to check if all the terms, split by a comma, are in the product name, if so the condition is met, if not, match\_row is set to False
* Lines 122-128 are then copied and slightly modified for ‘sector\_contains’ and ‘product\_not\_contains’

Text

Description automatically generated

* Once all three have been checked, if the row matches the product, then if there has already been a match valid is set to False, otherwise the index of the matching row is stored in match

Text

Description automatically generated

* Once all the rows have been looked through, if valid is True and there is a match, then ‘food\_emissions’ is set to the data in ‘emissions\_stats’ (the kg CO2 produced per kg food) multiplied by the mass of the product in kg
* If the mass data is not available this would cause an error so check for this first in line 151
* Line 153 checks to see if both food\_emissions and transport\_emissions were filled with data as expected, and if so, the two are added together, rounded, and set as the value of the ‘emissions’ column in the DataFrame
* If one of them didn’t work, I print some relevant information to help debugging and check that this was not due to a fault in my code
* Line 160 sets the emissions to 0 if the data was not calculated correctly, so that next time the code is run it will not try to calculate the same product again (at the start of the program the query only selects products where emissions = NULL, so 0 would not be selected)

A few queries are then run to update the databases:

Text

Description automatically generated

* Within the loop, this query updated the ‘emissions’ field of the current product

I then wanted to calculate some extra metrics: emission per kg, per portion and per calorie. For the first two, this is very simple:

Text

Description automatically generated

* The queries calculate all of these values, only on rows where the relevant data is not 0 in order to not produce divide by zero errors

For emissions per calorie, since the calorie information is currently stored in a dictionary of all nutritional information (e.g. “{‘Energy’:100.0, ‘Fat’:1.0, …}”), it is a little more complicated to get the information:

Text

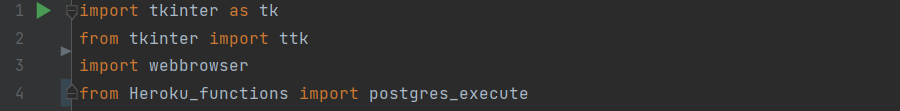
Description automatically generated

* This query searches for information between the string ‘Energy:’, and the next digit after the next full stop (so “‘Energy’:100.0” would be converted to “100.0”)
* This is then converted to a float, to give the calories per 100g
* Since the energy is the calories per 100g, I then multiply by 10 to give calories per kg, and then multiply by mass to give total calories
* This is then all divided from the emissions to give kg CO2/kcal
* This data will be very small, so to make the numbers more reasonable, I multiply by 1000 to give the g of CO2 per kcal, which is stored in ‘emissions\_per\_calorie’

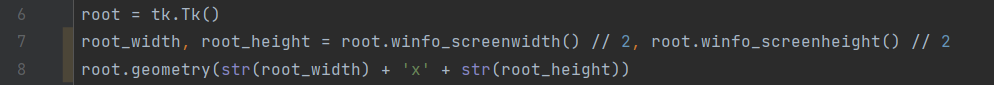
## User interface

My final step in the programming of the project was to create a Graphical User Interface to present relevant data to a user in an easily understandable way, in order to inform them about the emissions of various products so that they can make changes which benefit the environment. The final design, as shown below, was created from my initial designs with a few changes from my stakeholder feedback.

In order to create the main page, I will use Tkinter’s **grid layout** to position the widgets on the page, while the table in the search page will be created using the **treeview layout**.



* The relevant functions are imported
* **webbrowser** is used for opening chrome from within Python, so that the user can click on an item, and the page for that item on the Sainsbury’s website by opened in a browser



* Line 6 sets up the Tkinter window, stored in **root**
* Lines 7-8 set the window size to be half the screen height and width

First I set up the home page:

Text

Description automatically generated

* **clear\_window** clears the Tkinter window by creating a list of all the elements in the window, and then destroying them individually in line 24

Text

Description automatically generated

* **config\_grid** is used to configure the layout of a Tkinter grid, given lists of the weights and minimum sizes of the rows and columns
* For the grid layout, the sizes of rows and columns are set by assigning ‘weights’ to the rows and columns, they also can have minimum sizes, so that when the window is resized they cannot go below a certain width
* Lines 28-30 set the weight and min size for each row, while lines 31-33 do the same for each column

Text

Description automatically generated

* **search\_products** gets all the products from the products database which match a search term. For example, if the search term is ‘beef steak’, it should return a DataFrame of all products in the products database with ‘beef’ either in the product name or sector name, as well as ‘steak’ either in the product name or sector name
* To do this, the SQL query in lines 37-46 gets all the information from ‘sainsburys\_products’, joined with ‘sainsburys\_sectors’, where there exists emissions data and a condition is met
* This condition is created in lines 44-45, using the AND, OR and LIKE keywords to make sure all words in the search term are in either the product name or sector name
* The result of the query is returned



* **data\_columns** and **tree\_columns** are lists of the column headings for the table
* data\_columns gives the names of the relevant columns in the products database, while tree\_columns gives the headings for the columns on the interface

Text

Description automatically generated

* **home\_page** described the steps to create the home page
* Line 52 clears the window, and line 53 configures the grid with set weights and minimum sizes
* Line 55 creates the home button as a tk.Button called ‘home\_button’, setting the command on clicking the button to ‘home\_page’, so that when this button is clicked the program goes to the home page
* Line 56 adds ‘home\_button’ to the grid in the top left corner
* Line 57 creates the search bar as a tk.Entry called ‘search\_bar’
* Line 58 adds ‘search\_bar’ to the grid in its correct position
* Line 59 creates the ‘Go’ button as a tk.Button called ‘search\_button’, setting the command on clicking the button to ‘search\_page’. The parameters of this function are the current contents of the search bar, which will be the search term in the search page, and ‘emissions’, which is the column that the data will initially be sorted on
* Line 60 adds ‘search\_button’ to the grid
* Line 65 updates the page

Text

Description automatically generated

* **search\_page** describes the steps to create search page, taking the parameters:
  + **search\_term** is the search term entered by the user
  + **sort\_by** is the field to be sorted by
  + **desc** is whether the table should be sorted in descending or ascending order of the field in ‘sort\_by’. True means descending and False means ascending
* Lines 69-72 act the same as in ‘home\_page’, clearing the window, configuring the grid, and creating the home button
* Lines 73-76 create the search bar and ‘go’ button as before, but this time they are positioned at the top of the page
* Line 77 creates a Tkinter Treeview object, this is essentially a table which can be used a widget and placed in the grid
  + The column name are set to ‘data\_columns’
  + In line 78, the height of the table is calculated, this is given as the number of rows to be shown. To calculate the number of rows which can be shown, I first get the height of the cell in the grid which the table will be in in pixels, subtract the height of the column headings in pixels (25) and divide by the row height in pixels (30)
* Lines 79-80 set the column headings for the table
* Line 81 runs ‘search\_products’ to get the database of relevant products and stores it in **data**
* Lines 82-83 add the data from the DataFrame into the table, using a list comprehension to only get the information from the columns listed in ‘data\_columns’

Next, I define the function, ‘item\_selected’, that is called when a cell in the table is clicked:

Text

Description automatically generated

* Lines 70-71 use in-built functions to determine the column and row that were clicked
* For the condition in line 72, “col == ‘#1’” happens when the ‘Product name’ column is clicked, and ‘row’ is True when a row in the table is clicked (rather than empty space or the column heading, so this condition is True when the name of a product is clicked
* Line 73 opens the URL to that product in a Chrome browser when the name of the product is clicked in the table. ‘row’ is in hexadecimal format with a ‘#’ at the start so must be converted using int(row[1:],16)
* Line 74 is a condition to check if the place being clicked is a column heading
* Line 75 converts ‘col’ in the format (‘#1’ or ‘#2’) into an integer column index (0 or 1)
* **col\_name** stores the name of this column
* At this stage, a column heading has been clicked. If the column that was clicked is not the column that the data is currently being sorted on, then ‘search\_page’ is run with ‘col\_name’ as the new ‘sort\_by’ parameter. If the column clicked is the column already being sorted by, then the direction of sorting is reversed (desc=not desc)

This function is then bound to the table:



Text

Description automatically generated

* Lines 99-101 sets the widths and minimum widths of the columns in the tree. The ‘name’ column is dealt with separately since it should be larger than the rest, while all the others are dealt with in a loop in lines 100-101
* Line 102 places the table in the grid
* Lines 103-105 create a scrollbar for the table
* Line 109 updates the window

Finally, to begin the program, ‘home\_page’ is run:



Rounding

Metrics

Order by

New query for finding sectors for products

Success criteria

Colours

Filters

Evaluation – multiple ingredient products

Testing

References

1. Estimated from data at [https://www.statista.com/forecasts/997903/online-grocery-shopping-by-store-brand-in-the-uk](about:blank) and https://www.worldometers.info/world-population/uk-population/ [↑](#footnote-ref-1)
2. \* Only on own-brand products [↑](#footnote-ref-2)
3. https://www.nature.com/articles/s41597-021-00909-8 [↑](#footnote-ref-3)
4. https://ourworldindata.org/food-transport-by-mode#:~:text=Transporting%20food%20by%20air%20emits,CO2eq%20by%20air. [↑](#footnote-ref-4)
5. https://ourworldindata.org/food-transport-by-mode#:~:text=Asparagus%2C%20green%20beans%20and%20berries,makes%20them%20hard%20to%20avoid. [↑](#footnote-ref-5)
6. https://www.health.state.mn.us/communities/environment/food/docs/fs/tcsfoodfs.pdf [↑](#footnote-ref-6)