SI 206 Final Project Report - Cookie Recipes: GitHub Repository Link

By: Joslyn Vince and Shakuntala Balusu

I. Initial Project Goals

Our initial plan when we began this project was to work with food databases, specifically Spoonacular API and Zestful. We recognized that it would be more efficient and simpler to focus on a specific type of food or recipe, so we narrowed our scope down to cookie recipes. The Spoonacular API allowed us to access thousands of recipes, and the Zestful API would have allowed us to parse through specific ingredients in order to determine the primary ingredients used across various cookie recipes. We wanted to find and store information about the average amounts of each standard cookie ingredient, the minimum and maximum amounts of sugar, and the average cookie ratings. We considered standard cookie ingredients as any ingredients that were fairly common across various recipes, such as flour, butter, sugar, etc. The minimum and maximum amounts of sugar would be indicative of which cookies would be the lowest and highest in sugar content. Additionally, we looked into utilizing a third API, specifically Open Food Facts, since it did not require an API key. We wanted to utilize this in order to determine nutritional information about the different cookie recipes and ingredients used. As far as the visualizations, we planned to utilize Altair to create three separate types of charts: a histogram depicting the average number of cups of flour in each recipe, bar charts comparing the average measurements of each standard ingredient used in the recipes, and a pie chart showing the standard cookie rating across all the cookie recipes provided by the Spoonacular API. We did not have any specific predictions or assumptions going into this project, but wanted to create a general analysis of cookie recipes and nutrition info.

II. Updated Project Goal/Goals Achieved

Our overall goal for the project stayed rather consistent throughout the process, as we chose to still analyze cookie recipes and collect data about the ingredients used. However, after beginning the project, we quickly realized that it was difficult to use Spoonacular and Zestful in tandem, and that Zestful wouldn't necessarily provide data that we couldn't generate through just using Spoonacular itself. Instead, we chose to utilize the Kroger API to determine the prices for the individual ingredients, which we then utilized to calculate the prices for each of the recipes. We also decided to calculate the prep time for the recipes, rather than the amount of ingredients, as not only was it simpler to calculate, but we felt that it would provide more necessary information. Furthermore, we realized that Spoonacular does not provide ratings, so we did not end up analyzing that aspect of the various recipes. Additionally, we did not incorporate the additional API (Open Food Facts), because we found that it was difficult to integrate,

especially since we had already finalized the other aspects of our project before we looked into it. Even without an API key, it was difficult to utilize it, especially since we didn't find specific ingredient amounts, which meant that the nutritional facts would've been based on a 100g serving, which would not have been as helpful. Without specific ingredient amounts, it would be difficult to find information such as the recipes with the highest caloric amounts.

III. Problems

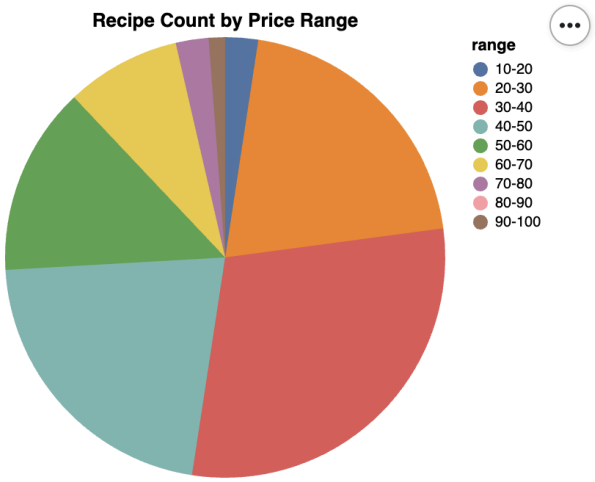
As mentioned earlier, we encountered difficulties with utilizing the Zestful API, as we realized that it would not work well in tandem with the Spoonacular API and wasn't necessarily relevant to our project. We struggled a bit in determining which API would be best suited to use with the Spoonacular and would provide relevant information about the cookie recipes. Furthermore, we only had a limited number of daily requests that we could make with the Spoonacular API which made it difficult to work on the code itself, especially at the beginning when we weren't aware of the fact that each time we changed our functions, we would be using a certain number of calls/requests. We also faced difficulty when we were initially setting up our recipes database and trying to ensure that each ingredient would be assigned a number/index after it was referenced once. This was one of the primary steps we needed to take, so we were unable to move on to the following steps without fixing this key issue.

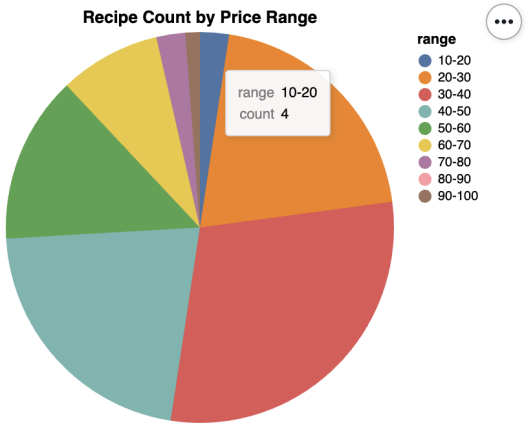
IV. Calculations from Data (final_calculations.txt)

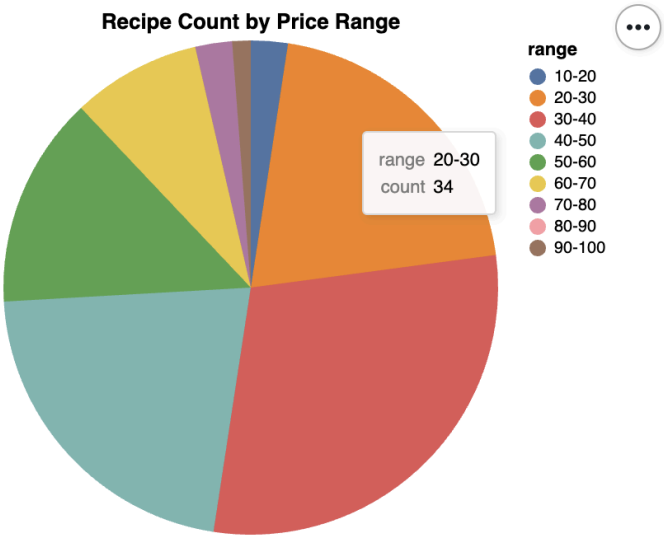
```
Getting Information about Recipe Average, Max Price, and Min Price
----------------------------------------------------------------------
Avergage price of recipes in database: $41.65283132530121
The name of the recipe with the **highest cost** is  - MINI OATMEAL COOKIE CHEESECAKES,
         with a price of: $91.96
The name of the recipe with the **lowest cost** is  - Reeses Peanut Butter Chocolate Chip Cookies,
         with a price of: $16.37

Getting Information about Price Per Serving Average, Max Price, and Min Price
--------------------------------------------------------------------------------
Avergage price per servings of recipes in database: $2.770240963855422
The name of the recipe with the **highest cost** is  - 90 Second Cookie in a Bowl,
        with a price per serving cost of: $26.82
The name of the recipe with the **lowest cost** is  - Eggless Chocolate Chip Crescent Cookies - Eggless Cookies,
         with a price per serving cost of: $0.36
```

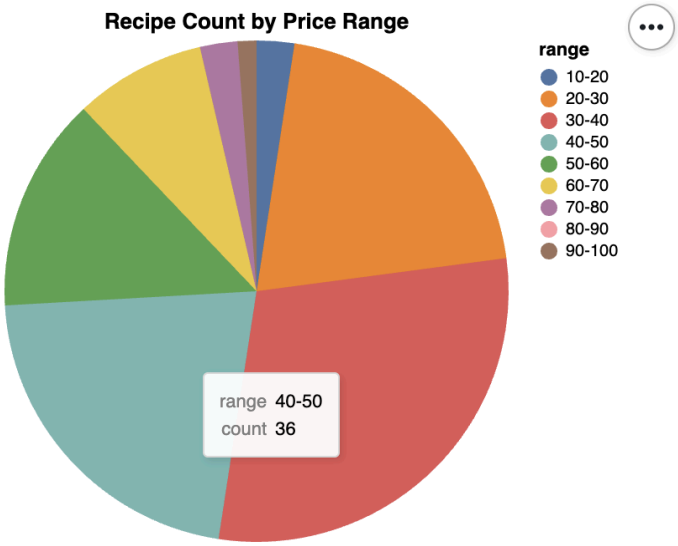
V. Visualizations

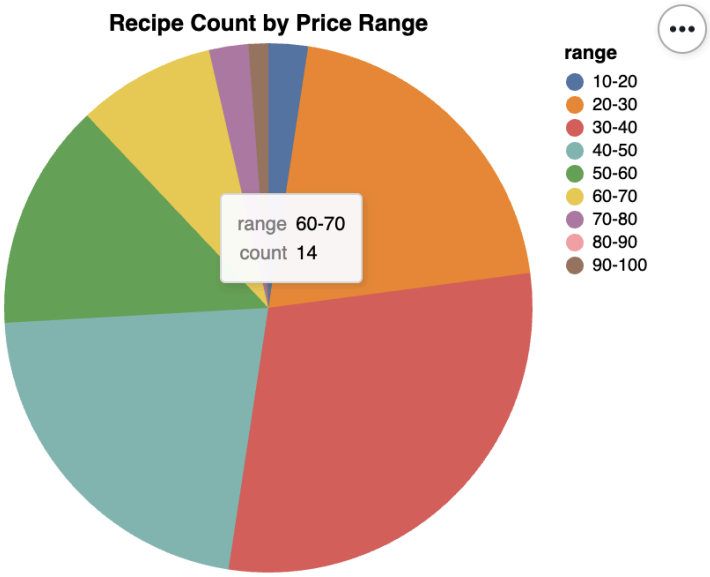
Visualization 1:

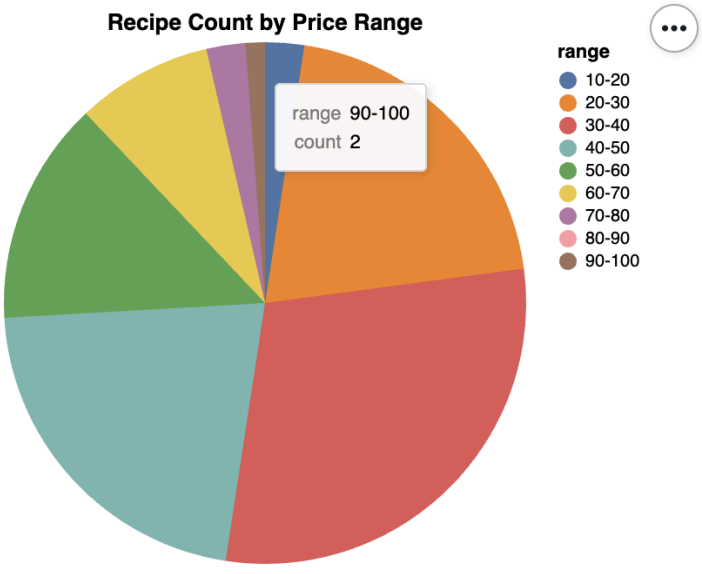

Recipe Count by Price Range
range
10-20
20-30
30-40
40-50
50-60
60-70
70-80
80-90
90-100


Recipe Count by Price Range
range 10-20
count 4
range
10-20
20-30
30-40
40-50
50-60
60-70
70-80
80-90
90-100

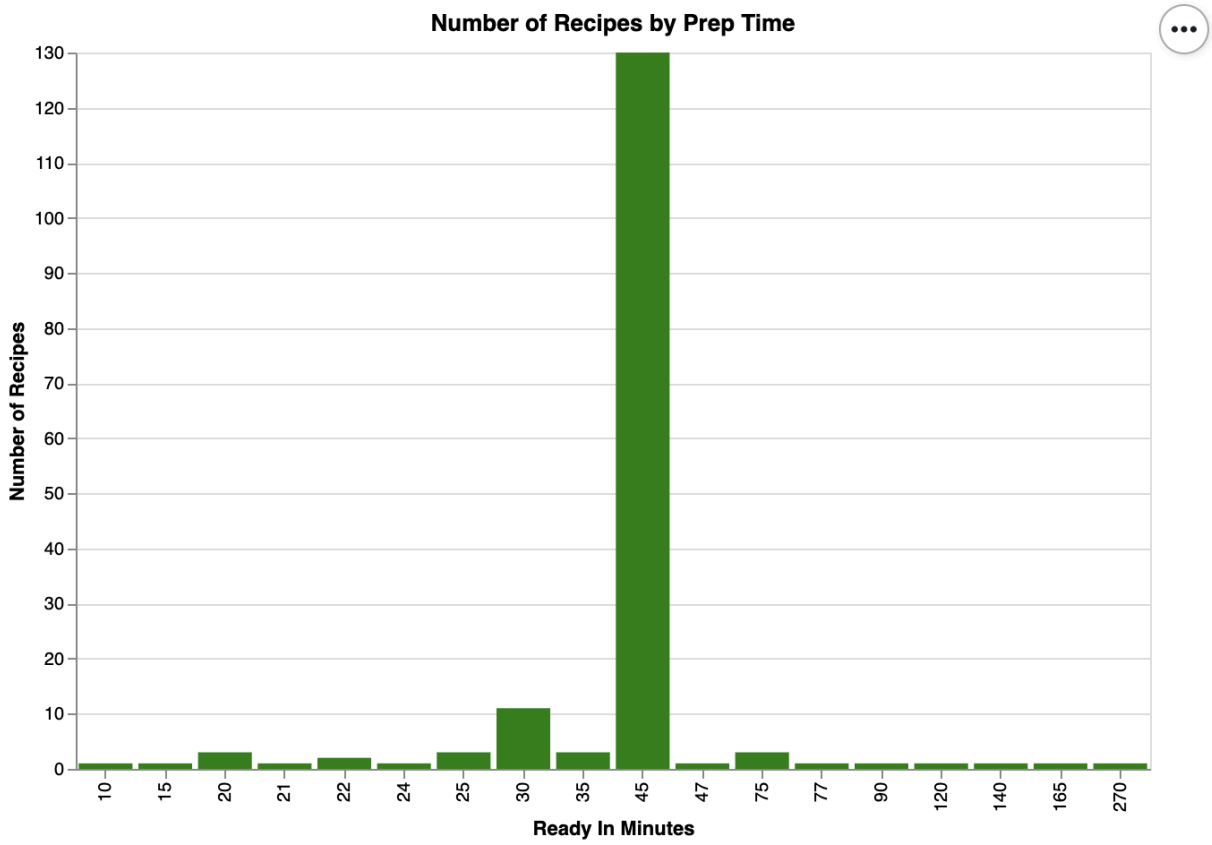

Recipe Count by Price Range
range 20-30
count 34
range
10-20
20-30
30-40
40-50
50-60
60-70
70-80
80-90
90-100


Recipe Count by Price Range
range 40-50
count 36
range
10-20
20-30
30-40
40-50
50-60
60-70
70-80
80-90
90-100

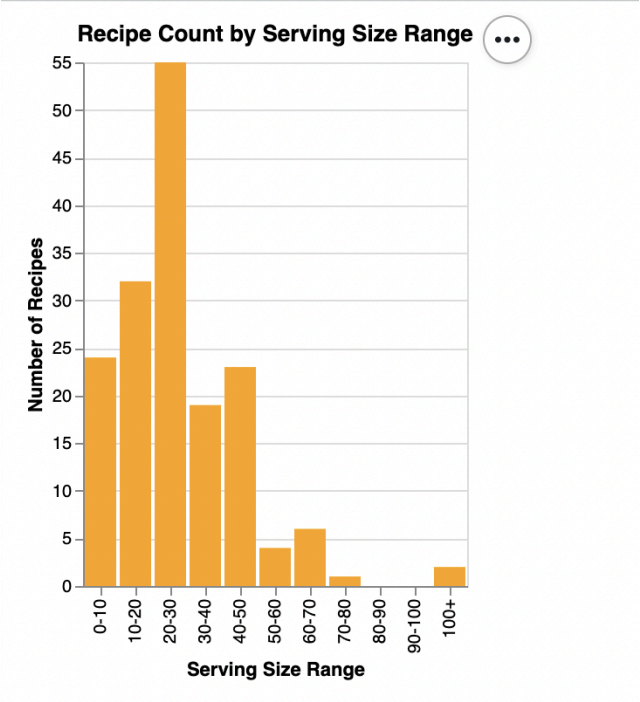

Recipe Count by Price Range
range 60-70
count 14
range
10-20
20-30
30-40
40-50
50-60
60-70
70-80
80-90
90-100


Recipe Count by Price Range
range 90-100
count 2
range
10-20
20-30
30-40
40-50
50-60
60-70
70-80
80-90
90-100

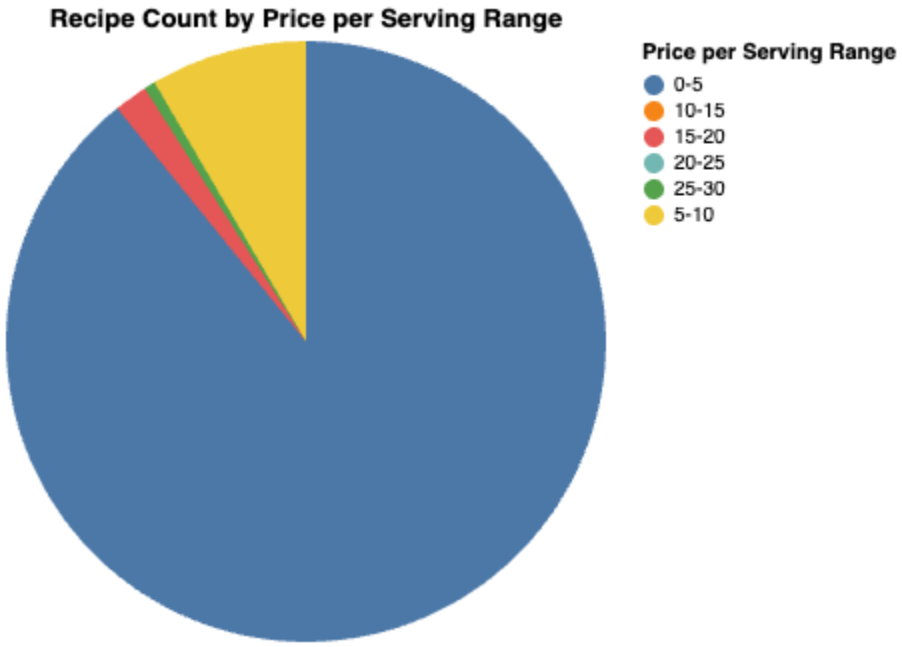
Visualization 2:



Visualization 3:



Visualization 4:



VI. Instructions for Running Code

A. Setup

1. When you initially open the project folder, there will be multiple Python files that work together to retrieve, process, and analyze data. These include scripts for collecting recipe information, accessing pricing data, organizing tables, generating calculations, and creating visualizations. All the files have to be stored in the same directory. Recipes.db, which is the database used in the project, will be created and populated automatically when the scripts are run.

B. Running the Code

First, run the script that gathers cookie recipe and ingredient information from the Spoonacular API, which is spoonacular.py. Then, run the script that connects to the Kroger API and collects prices for each ingredient (kroger.py). After that, run the script that calculates total prices and price per serving and stores this information in final tables (table_joins.py). Next, run the calculations script to print out statistics like the average recipe cost and the most and least expensive

recipes (calculations.py). Finally, run the four visualization scripts to generate interactive charts (vis1.py, vis2.py, vis3.py, vis4.py). Each visualization will save an .html file which can be opened in your browser.

C. **Expected Output**

Running the code will generate a database file called recipes.db. This database will contain multiple tables that store recipe information, ingredient names and prices, total recipe costs, and price per serving. After running the visualization scripts, four interactive charts will be saved in the project folder. All of these charts can be opened in a browser.

1. vis1.html: a pie chart showing the number of recipes grouped by total price
2. vis2.html: a bar chart showing the number of recipes by prep time
3. vis3.html: a bar chart showing how many recipes fall into each serving size range
4. vis4.html: a pie chart showing recipe counts by price per serving

D. **Troubleshooting Tips**

If any errors occur while using the Kroger API, double-check the client ID and secret. If you notice missing or incomplete data in the database, it may help to delete the recipes.db file and rerun the main script. Make sure that all files are in the same directory and that the visualization scripts are only run after the database is fully built.

VII. Documentation of Functions

A. Spoonacular File (spoonacular.py)

1. get_cookie_recipes
    a) This function essentially sends a request to the Spoonacular API to retrieve any recipes that are related to cookies. It creates a URL with parameters like number and offset, and then returns a list of recipe dictionaries.
    b) Inputs
        (1) Number (int) → the number of recipes to retrieve
        (2) Offset (int, default = 0) → tells the API where to start getting recipes from (offset = 25 would skip the first 25)
    c) Outputs

(1) A list of recipe dictionaries (each includes id, title, image, etc.)

2. connecting_with_recipes_database
    a) This function creates the Recipes table in the SQLite database and fills it with the different cookie recipes from the API. It only fills the table if the current number of recipes is less than the specified target number. After filling the table, it deletes any recipes that aren't cookie-related.
    b) Inputs
        (1) cur: SQLite cursor object
        (2) conn: SQLite connection object
        (3) target (int): target number of cookie recipes
        (4) offset (int): offset for the number of pages when pulling new data
    c) Outputs
        (1) None → inserts data directly into the Recipes table
3. connecting_with_ingredients_table
    a) This function creates the Ingredients table and fills it with the number of servings, the time it takes to prepare the recipe, and a string of ingredients (separated by commas). It receives this information from the Spoonacular API using each recipe ID that is stored in the Recipes table.
    b) Inputs
        (1) cur: SQLite cursor object
        (2) conn: SQLite connection object
    c) Outputs
        (1) None → modifies the database by inserting into the Ingredients table
4. connecting_with_integer_key_table
    a) This function creates a table called IngredientNames to ensure that each ingredient has a unique ID (integer). It splits the string of ingredients into individual components and inserts them using INSERT or IGNORE in order to ensure there are no duplicates.
    b) Inputs:
        (1) cur: SQLite cursor object
        (2) conn: SQLite connection object
    c) Outputs:
        (1) None → fills the IngredientNames table with unique ingredient names
5. ingredients_table_with_integers

a) This function creates a new table called IngredientsWithIDs which copies the data from the Ingredients table but replaces the names with their respective numeric IDs.
b) Inputs
    (1) cur: SQLite cursor object
    (2) conn: SQLite connection object
c) Outputs
    (1) None → just fills the IngredientsWithIDs table with the ID's

B. Kroger File (krogerAPI.py)

1. get_kroger_access_token
    a) This function takes the client ID and secret to request an access token from the Kroger API. This token is necessary to make product/pricing requests.
    b) Inputs:
        (1) client_id (str): Kroger API client ID
        (2) client_secret (str): Kroger API client secret
    c) Outputs:
        (1) token (str): A string access token used for requests
2. get_price_for_ingredient
    a) This function asks the Kroger API for the price of a specific ingredient and uses the access token and ingredient name to search and return the regular price of the first product that matches.
    b) Inputs
        (1) token (str): Access token from the Kroger API
        (2) query (str): Name of the ingredient to look up
    c) Outputs
        (1) price (float or None): The regular price of the item (or None if there's no result)
3. store_all_prices
    a) This function checks each of the ingredients in the IngredientNames table and tries to find its price using the Kroger API. If a price is found or it already exists in the database, it skips it. If not, it retrieves the price and stores it in the KrogerPrices table.
    b) Inputs
        (1) cur: SQLite cursor object
        (2) conn: SQLite connection object
        (3) token (str): Access token from the Kroger API

c) Outputs
    (1) None → stores all the prices directly in the database

4. fill_missing_kroger_prices
    a) This function fills in missing prices in the KrogerPrices table, calculates the average of all known prices, and uses that value for any ingredient that does not already have a price.
    b) Inputs
        (1) cur → cursor
        (2) conn → connection
    c) Outputs
        (1) None → updates the database directly
5. round_all_prices
    a) This function rounds all the prices in the KrogerPrices table to two decimal places.
    b) Inputs
        (1) cur → cursor
        (2) conn → connection
    c) Outputs
        (1) None → updates database directly

C. Calculations File (calculations.py)

1. get_average_recipe_price
    a) This function calculates the average total price of all recipes in the FullRecipeInfo table.
    b) Inputs
        (1) cur: SQLite cursor object
        (2) file: File object to write results to
    c) Outputs
        (1) Writes a string showing the average total price of all recipes to final_calculations.txt
2. get_min_price
    a) This function finds the recipe with the lowest total price and prints its name and price.
    b) Inputs
        (1) cur: SQLite cursor project
        (2) file: File object to write results to
    c) Outputs:
        (1) Writes the title and total price of the cheapest recipe to final_calculations.txt
3. get_max_price

a) This function finds the recipe with the highest total price and prints its name and price.
b) Inputs
    (1) cur: SQLite cursor object
    (2) file: File object to write results to
c) Outputs
    (1) Writes the title and total price of the most expensive recipe to final_caluclations.txt

4. get_average_serving_price
    a) This function calculates the average price per serving across all recipes in the FullRecipeInfo table.
    b) Inputs
        (1) cur: SQLite cursor object
        (2) file: File object to write results to
    c) Outputs
        (1) Writes the average price per serving to final_calculations.txt
5. get_min_price_per_serving
    a) This function finds the recipe with the lowest price per serving and prints its name and value.
    b) Inputs
        (1) cur: SQLite cursor object
        (2) file: File object to write results to
    c) Outputs
        (1) Writes the title and price per serving for the cheapest recipe per portion final_calculations.txt
6. get_max_price_per_serving
    a) This function finds the recipe with the highest price per serving and prints its name and value.
    b) Inputs
        (1) cur: SQLite cursor object
        (2) file: File object to write results to
    c) Outputs
        (1) Writes the title and price per serving for the most expensive recipe per portion to final_calculations.txt

D. Joint Table File (table_joins.py)

1. create_recipe_prices_table
    a) This function creates a new table called RecipePrices which stores the total cost of each recipe by summing the prices of all its

ingredients. It retrieves each recipe's ingredient IDs from IngredientsWithIDs and matches them to prices in KrogerPrices, and then calculates the total cost. looks up prices using the IDs from IngredientsWithIDs and KrogerPrices.

b) Inputs
   (1) cur: SQLite cursor object
   (2) conn: SQlite connection object

c) Outputs
   (1) RecipePrices table with recipe_id (int) and total_price (float)

2. create_price_per_serving_table
   a) This function creates PricePerServing which is a table that calculates the price per serving for each recipe using the total price from RecipePrices divided by the number of servings from Ingredients.
   b) Inputs
      (1) cur: SQLite cursor object
      (2) conn: SQLite connection object
   c) Outputs
      (1) A table named PricePerServing w/ recipe_id (int) and price_per_serving (float)
3. all_recipe_info
   a) This function creates FullRecipeInfo which brings together all relevant information about each recipe, including title, servings, prep time, total price, and price per serving.
   b) Inputs
      (1) cur: SQLite cursor object
      (2) conn: SQLite connection object
   c) Outputs
      (1) A table named FullRecipeInfo w/ recipe_id (int), title (str), servings (int), readyInMinutes (int), total_price (float), and price_per_serving (float)

E. Visualization 1 File (vis1.py)

1. fetch_prices
   a) This function connects to recipes.db and retrieves all recipe total_price values from the FullRecipeInfo table.
   b) Inputs
      (1) None
   c) Outputs

(1) A list of float values representing total prices of all recipes

2. bucket_prices
    a) This function groups the list of prices into predefined price ranges/buckets using pandas. It also creates a DataFrame that shows how many recipes fall into each price range.
    b) Inputs
        (1) prices (list of floats): A list of recipe prices
    c) Outputs
        (1) A pandas DataFrame with 2 columns (range and count)
3. plot_pie
    a) This function generates a pie chart using Altair that visualizes the number of recipes in each price range, and saves the chart as an HTML file.
    b) Inputs
        (1) df (pandas DataFrame): The DataFrame containing price ranges and their counts
    c) Outputs
        (1) Saves a pie chart as vis1.html
        (2) Prints a confirmation message with the chart's file path

F. Visualization 2 File (vis2.py)

1. load_ready_times
    a) This function connects to recipes.db and loads the readyInMinutes values from the FullRecipeInfo table into a pandas DataFrame.
    b) Inputs
        (1) None
    c) Outputs
        (1) A pandas DataFrame with one column: readyInMinutes (int) – representing how long each recipe takes to prepare
2. plot_ready_minutes
    a) This function takes the prep time data and generates a bar chart using Altair. The chart then shows how many recipes take a given amount of time to prepare, and is then saved as an HTML file.
    b) Inputs
        (1) df (pandas DataFrame): A DataFrame containing readyInMinutes values
    c) Outputs
        (1) Saves a bar chart as vis2.html
        (2) The chart shows prep time on the x-axis and the recipe count on the y-axis

G. Visualization 3 File (vis3.py) (changed from presentation)

1. fetch_servings
    a) This function essentially connects to the recipes.db database, selects the "servings" column from the FullRecipeInfo table, and returns all the values as a list.
    b) Inputs
        (1) None
    c) Outputs
        (1) A list of integers which represent the number of servings for each recipe
2. bucket_servings
    a) This function groups the list of servings into size ranges using predefined bins. It uses pandas to separate the servings into ranges, and counts how many recipes fall into each range.
    b) Inputs
        (1) Servings (list of integers) → number of servings per recipe
    c) Outputs
        (1) Pandas
3. plot_histogram
    a) This function uses Altair to generate a bar chart that visualizes the number of recipes in each serving range, which is then saved as an HTML file.
    b) Inputs
        (1) df → grouped data with serving size ranges and recipe counts
    c) Outputs
        (1) HTML file named vis3.html with the chart

H. Visualization 4 File (vis4.py) (changed from presentation)

1. fetch_price_per_serving
    a) This function connects to the recipes.db database and receives all the values from the "price_per_serving" column in the FullRecipeInfo table, and then returns the values as a list.
    b) Inputs
        (1) None
    c) Outputs
        (1) A list of floats representing price per serving for each recipe
2. bucket_price_per_serving

a) This function takes the list of price per serving values and separates them into predefined ranges using pandas, and then returns a df with counts for each price range.
b) Inputs
    (1) Prices (list of floats) → price per serving for each recipe
c) Outputs
    (1) A pandas df with two columns (range and count)

3. plot_price_per_serving_pie
    a) This function uses Altair to create a pie chart that shows how many recipes fall into each price per serving range, which is then saved as an HTML file.
    b) Inputs
        (1) df → grouped price per serving data
    c) Outputs
        (1) HTML file containing the pie chart

VIII. Documentation of Resources

Date	Issue Description	Location of Resource	Result
4/9/25	Creating the integer keys and separating the strings/keys into two tables	ChatGPT	Helped us create a new function in order to organize our tables
4/14/2025	When calling our Walmart API the calls needed to be separated by 1 second and I did not know how to implement that	AI Overview when searched on google	We added “import time” to help separate API calls by 1 second
4/14/2025	To use the Kroger API, we had to use the base64 encoding to use their API and I did not know how to format this	I had ChatGPT help me to understand all of the parts of base64 encoding system so I could put it into my program	I got my API working after this because the encoding was done correctly.
4/14/2025	To use the Kroger API, I did not understand all of the parameters that were necessary to use this code	I had ChatGPT help me to understand why the params were necessary and why they are needed	I understood the issues of why my code wasn’t working and fixed my issues.
4/14/2025	I couldn’t figure out how to incorporate an API that didn’t require a key, because the other two had used keys.	I used ChatGPT to figure out how I should use the API and how I could best incorporate Open Food Facts into the existing code.	*We ended up deleting the Open Food Facts file and not using the API*, but this did help me initially determine how to incorporate it or include it without an API key.