The Makings of a Meal

By: Vaiva Raisys, Elizabeth Sheridan, Wiktoria Sobera

Github Repo Link: https://github.com/vaivaraisys/206FinalProject.git

I. Initial APIs/Websites

Initially for our project, we were planning on using three APIs revolving around music. These were Spotify, Apple Music, and Soundcloud. From these APIs, we were planning to collect the song information, such as title, artist, album, release date, duration, and rating. Also, the artist information, like the name, genre, followers, and rating. Then, we were going to store this information into a database and make tables for the songs and artists information. With the data, we were going to calculate the average song duration, most popular artist overtime, or the most popular types of music overtime. Then, we were going to use Matplotlib to make graphs that showed the popularity trends over time and a chart for genre distribution. However, we changed our entire project topic when we had a difficult time finding music APIs that did not require a membership or some sort of token for access.

II. Final APIs/Websites

Ultimately for our project, we decided to address the question: which letter has the highest food item and drink count? We used two APIs and one website. The APIs were TheMealDB and TheCocktailDB. These gave us a dictionary of meal information based on the first letter of the meal or drink, which we then scraped the names and ids into a list of tuples. The website was a Wikipedia page that had dessert names that we scraped into a list of desserts by first letter alphabetically. Then, we scraped the dessert names from the website into a list. In our code, we created a function that matches each letter of the alphabet to an integer key. Thus, the integer key represented the number corresponding to the starting letter of the food. For example, a = 1, b = 2, and so forth. For the meal API, we gathered the integer key, the starting letter, and

the meal name. In addition, we gathered the meal id which is an id associated with each different meal. We did the same for dessert, except collecting the dessert name, but there was not a dessert id provided with this website.

III. Problems

One of the largest errors that we ran into initially was finding APIs that had free and unauthorized access, as stated earlier in the report. We found several APIs that were asking us to create an account, retrieve a token, and were advised by GSIs to find different APIs that did not do this. To do this, we searched on GitHub which had a long list of free public APIs divided into several different categories. We looked at the food and drink category and found several that worked and seemed interesting so we decided to change our topic.

Another issue that we ran into when working on our final project was creating a function that limited the amount of data stored from our API to the database to 25 or fewer items. Each time we thought that we did it so it was going to only store 25 items or fewer, it did all the items. After going to office hours several times, we created a “max-items” variable in our argument that was defaulted to 25. This made it so that no more than 25 pieces of data were plotted at once. Further, we initialize a “nums_to_insert” variable that checks that we did not exceed the “max-items” variable and that we do not insert more than the items in the data from each API, hence the current_count variable. Pictures below show this.

```
def set_up_desserts_table(desserts_tuple, cur, conn, max_items=25):
```

```
cur.execute("SELECT COUNT(*) FROM desserts")
current_count = cur.fetchone()[0]
```

```
# limits the number of items inserted doesn't exceed the maximum allowed (max items), and
ensures that you don't try to insert more items than are desserts_tuple
num_to_insert = min(max_items, len(desserts_tuple) - current_count)
```

Another issue we ran into was invalid meal or dessert names. Some of the meal and dessert names we retrieved from our sources were not American desserts, so the starting letter was not in the english alphabet, so our function did not work in these instances. Therefore we had to implement an additional if statement to check that the starting letter is valid in the alphabet, and if not, then it prints “invalid letter”, and the letter that is not valid. For example, one of the invalid letters we ran into was Š, so when the dessert that started with this letter went through this if/else statement, it printed “Invalid starting letter: Š”.

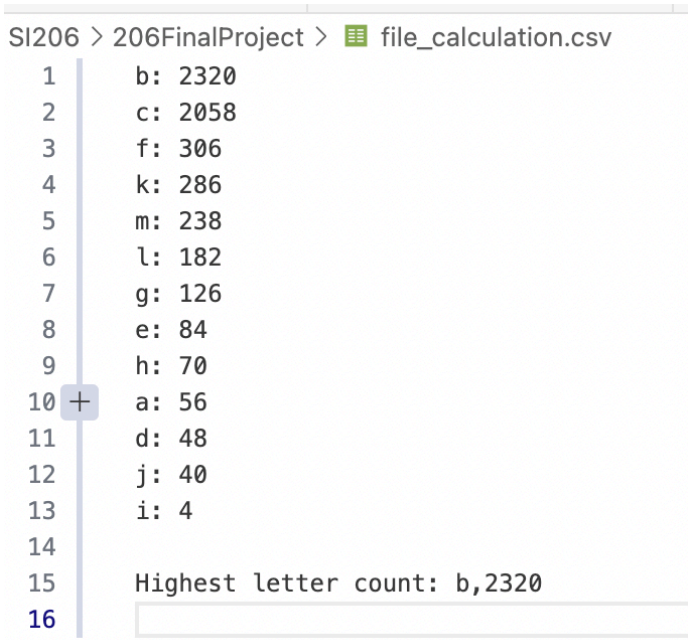
```
if dessert[0].lower() in string.ascii_lowercase:
        starting_letter = dessert[0].lower()
```

IV. Calculations

For our calculation, we calculated the group of meals and desserts combined that started with the same letter that had the highest count. Then, we sorted it by the letter count in

descending order. We did this by selecting the dessert and meal names and doing an inner join of meal names on desserts. After, we went through that data and found the highest count for both the meal and dessert data. We wrote that to a file and displayed the highest count along with the letter that was associated with this count. The result was the letter b and 2320. A picture of the calculation, as well as the calculation file are below.

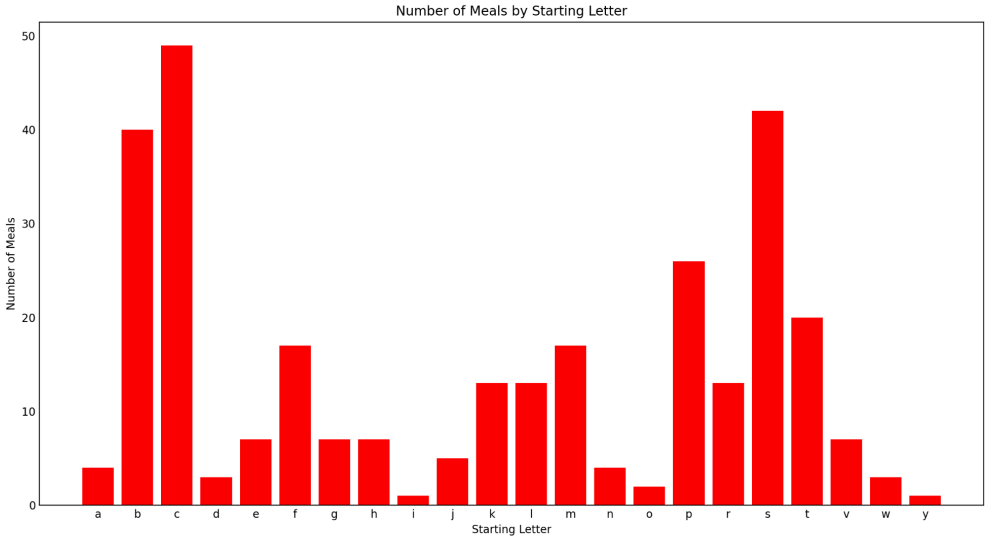
```python
import requests
import json
import unittest
import os
import sqlite3
import random
import string
from bs4 import BeautifulSoup
# from mealfunctions import set_up_meal_table
# from dessertfunctions import set_up_desserts_table
from mealfunctions import set_up_database


def meal_dessert_data(cur, conn):
    cur.execute(
        "SELECT desserts.Dessert_name, meal_names.Meal_name FROM desserts INNER JOIN meal_names ON desserts.Integer_Key = meal_names.Integer_Key AND desserts.Starting_Letter = meal_names.Starting_Letter"
    )
    meal_dessert_data = cur.fetchall()
    # print(meal_dessert_data)
    return meal_dessert_data

def calculate_most_letter_meals(meal_dessert_tuple):
    # cur.execute("SELECT Starting_Letter, COUNT(*) AS Meal_Count FROM meal_names GROUP BY Starting_Letter ORDER BY Meal_Count DESC LIMIT 1")
    # most_common_letter = cur.fetchone()
    # print(most_common_letter)
    letter_count = {}
    # print(meal_dessert_tuple)
    for dessert, meal in meal_dessert_tuple:
        dessert_letter = dessert[0].lower()
        meal_letter = meal[0].lower()

        if dessert_letter in letter_count:
            letter_count[dessert_letter] += 1
        else:
            letter_count[dessert_letter] = 1

        if meal_letter in letter_count:
            letter_count[meal_letter] += 1
        else:
            letter_count[meal_letter] = 1
    sorted_letter_dict = dict(sorted(letter_count.items(), key=lambda item:item[1], reverse=True))
    most_common_letter = max(letter_count, key=letter_count.get)
    # print(most_common_letter)
    print(sorted_letter_dict)
    return sorted_letter_dict

def write_csv_file(file, sorted_meal_dict):
    with open(file, "w") as fh:
        for key, value in sorted_meal_dict.items():
            fh.write(f"{key}: {value}\n")

        highest_letter, highest_count = next(iter(sorted_meal_dict.items()))
        fh.write(f"\nHighest letter count: {highest_letter}, {highest_count}\n")


def main():
    cur, conn = set_up_database("food_data.db")
    meal_dessert_tuple = meal_dessert_data(cur, conn)
    sorted_meal_dict = calculate_most_letter_meals(meal_dessert_tuple)
    write_file = write_csv_file("file_calculation.csv", sorted_meal_dict)
main()
```

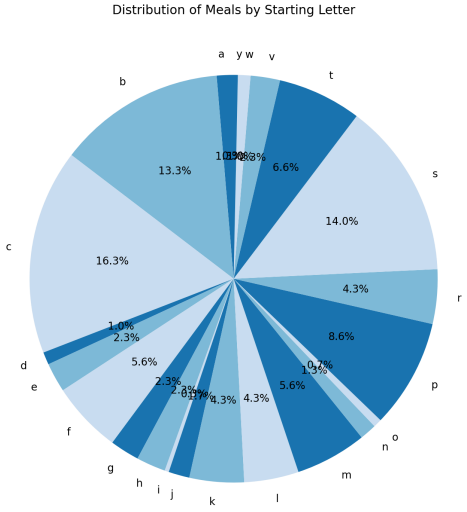


V. Visualizations

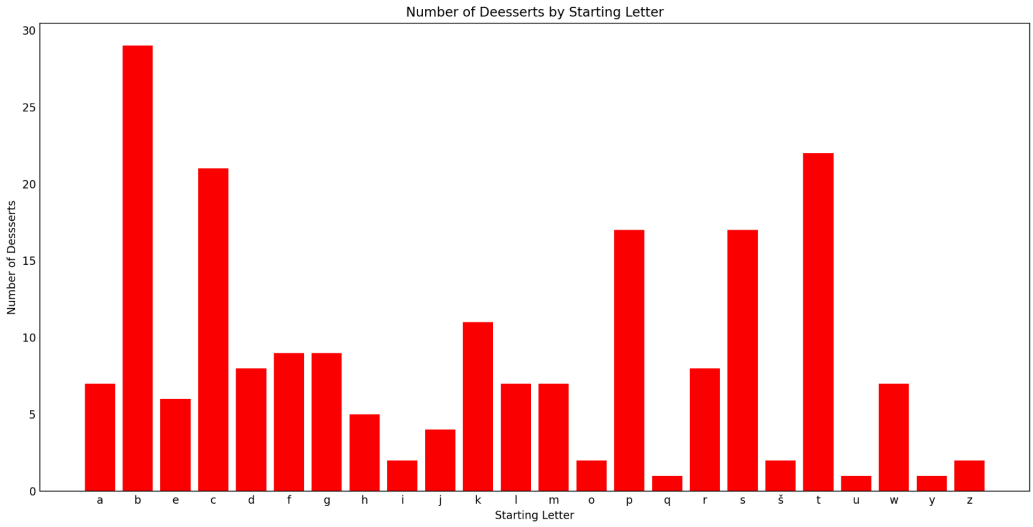
In total, we made four visualizations with two corresponding to meals and two corresponding to desserts. For both meals and desserts, we created a pie chart that showed the distribution of meals or desserts by starting letter. In addition, we created two bar graphs that showed the relationship between the starting letter and the number of meals/desserts.

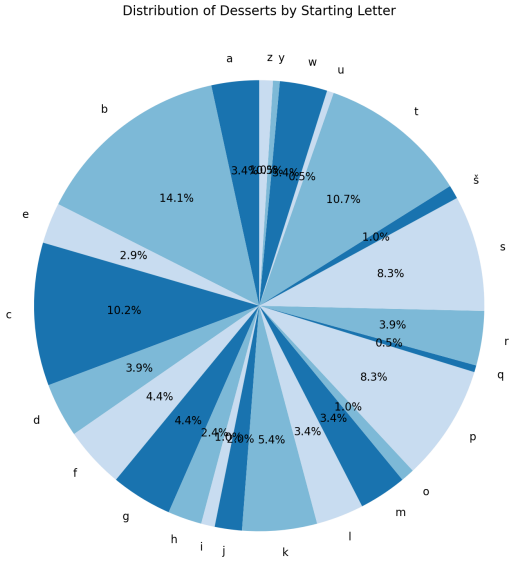
Meal Visualizations:





Dessert Visualizations:





VI. Instructions for Running the Code

In order to run our code, the first step is to delete the database, if it is already created. From there, you run the dessert functions file. After the first run, the database will be recreated, and you will see the desserts table created with 25 data points in it. You then continue to run the dessert functions file until there are 203 data points in the table. You then continue to the drink

function file, and run that file until there are 282 data points in the drink table. Lastly, you run the meal functions file until there are 301 data points in it. At this point, the database is fully loaded with all of the data and our database is ready to be processed. The next step is to run the process data file, which once it is run, it joins the dessert and meals on their starting letter, calculates the most common letter, and writes it a csv file. The final step is to run the visualizations file. We found that running the visuals individually made it run faster than all at the same time. Therefore, to run each one individually, you have to go to the main function of the visualizations file and comment back in whatever chart you want displayed.

VII. Documentation for each Function

A. Dessert Function File

Retrieve_desserts: This function does not take in any arguments. It uses Beautifulsoup to retrieve the dessert names into a list. The function then returns a list of strings, in alphabetical order, of dessert names.

Set_up_dessert_table: This function takes 4 arguments – the list of desserts from retrieve_desserts, a cursor to make execute statements, connector (to commit items to the database) and a max_items variable that is defaulted to 25 items to ensures no more than 25 items are placed into the table at once. This function does not return anything.

Main: The main function calls all of the functions from this file. It first calls the retrieve dessert function to get the list of dessert, it sets up the database, with the cur and connector from the meals functions file (which is imported at the top), and calls the function to set up the dessert table.

B. Drink Function File

Get_drink_data: This function takes no arguments. The function loops through the alphabet calling the API each time, putting each drink instructions and then drink ID into a tuple, and then appending that tuple into a list. This list of tuples is what is returned.

Set_up_drink_table: This function takes in 4 arguments – the list of tuples from get_drink_data, a cursor, connector and max_items that is defaulted to 25, so that no more than 25 items are inputted at a time. This function does not return anything.

Main: This main function calls 3 functions. It gets the drink data and stores it into a variable called “drink_dict”, sets up the database, which is imported from the meals function file, and calls the set up drink table function.

C. Meal Function File

Get_meal_data: This function does not take any arguments. The function loops through the alphabet calling the API each time, putting each meal name and then meal ID into a tuple, and then appending that tuple into a list. This list of tuples is what is returned.

Generate_number_letter_tuple: This function does not take any arguments. It iterates through the alphabet and assigns a number to each letter, and it returns a list of tuples that has the number with the associated letter in each tuple.

Set_up_database: This function takes in the name you want your database to be called. The function returns the cursor and the connector. The cursor is the object that interacts with the database through executing and fetching queries. The connector is the way of connecting to the

database itself and committing changes to it; it establishes the connection to the database. This function does not return anything.

Set_up_integer_table: This function takes 3 arguments – a cursor, a connector and number letter tuple. The cursor and connector are from the setup database function that establish the connection to the database and allow for the functions to interact with the database through execute statements and fetchall queries. The number letter tuple is the output from the generate_number_letter_tuple function and the output from this function is taken in. This function does not return anything.

Set_up_meal_table: this function takes 4 arguments – the meal data, a cursor, a connector, and max_items variable. The meal data argument comes from the get meal data function. The cursor and connector are from the setup database function that establish the connection to the database and allow for the functions to interact with the database through execute statements and fetchall queries. The max_items variable is defaulted to be at 25, and is used so that no more than 25 data points are put onto the table at a time. This function does not return anything.

Main: This main function calls 5 functions. It calls the generate number letter tuple function and stores it into a variable to be used as an argument in the set up integer table. It calls get meal data and also stores it into a variable to be used as an argument in the set up meal table function. It sets up the database itself through the cursor and connector. Finally, it calls the functions set up meal table and set up integer table to set up these individual tables.

D. Process Data File

Meal_dessert_data: This function takes in two arguments — a cursor and a connector. The function returns a list of tuples that have dessert names and meal names joined on the starting letter.

Calculate_most_letters_meal: this function takes in one argument — a meal/dessert tuple. This meal/dessert tuple is coming from the output of the meal_dessert_data function. This function returns a sorted dictionary in descending order of the count of each letter based on the dessert meal data.

Write_csv_file: this function takes two arguments – a file name and the sorted meal dictionary. The filename is what you want the file to be named once it is written. The sorted meal dictionary is the output of the calculated most letter meals function. This function does not return anything.

Main: The main function for this file calls 4 functions. It sets up the database itself through the cursor and connector, which is imported from the meal file. It calls the meal_dessert_data function and stores it into a variable to use as an argument for another function. It also calls the calculate_most_letter_meals and write_csv_file to calculate the highest count and write that count to a csv file.

E. Visualizations File

Meal_pie_chart: This function does not take in any arguments. Inside the function it calls get meal data and plots that data onto a pie chart. This function does return anything either.

Meal_bar_chart: This function does not take in any arguments. Inside the function it calls get meal data and plots that data onto a bar chart. This function does return anything either.

Dessert_pie_chart: This function does not take in any arguments. Inside the function it calls get dessert data and plots that data onto a pie chart. This function does return anything either.

Dessert_bar_chart: This function does not take in any arguments. Inside the function it calls get meal data and plots that data onto a pie chart. This function does return anything either.

Main: this function calls 4 functions – each of the visualization functions. In order for it to run properly, we only have 1 written out at a time and the other 3 commented out.

VIII. Resources

Date	**Issue Description**	**Location of Resource**	**Result (did it solve the issue?)**
4/11	Attended lecture as we were having trouble starting the project. Thus, asked questions on how to get us in the right direction to start the project.	CCCB Lecture Room	Yes, we were able to get started on our project and get our initial questions answered.
4/15	Talked to GSI about our issues with getting access to certain API keys we were trying to use.	Zoom Office Hours	She advised us to switch our topic. Once she did, our issues were resolved.
4/16	Attended lecture to get help with implementing our chosen APIs and website.	CCCB Lecture Room	Yes, we got all of our questions answered.
4/17	Got help with various questions that we had about the project: how to store things to the database, how to	Zoom Office Hours	Yes, we got all of our questions answered.

	create the tables, and the visualizations.		
4/18	Got help on errors that we were having as a result of functions we wrote in office hours.	CCCB Lecture Room	Yes, we got all of our questions answered.
4/21	Got an idea of how to implement a function that would store 25 items to the database at a time. Also, she gave good guidance on what specific site to make the visualizations on and a potential calculation that we could incorporate.	UMSI Peer Tutoring Zoom	Yes, we got all of our questions answered.
4/21	We could not figure out how to implement a function that only drops 25 data points at once. In office hours we implemented an index variable that was updated based on the count of how many rows there were at the beginning of the function. However, we could not figure out how to loop through our data using the index variable but stopping it after 25, and then picking up where we left off the next time we ran the function.	Chat GPT	Chat GPT gave us the idea to implement a “max_items” and “nums_to_insert” variable that took the current count and the total count and made sure we only placed 25 points at a time.
4/22	We were having	Zoom Office Hours	No, the first IA that

	problems with getting 25 items at a time. Also, we clarified what we needed to present on Tuesday 4/23 in lecture.		we talked to was not able to answer our question. When we were getting transferred to the other GSI's room, the zoom ended.