Detailed Notes for our Code in Python BA Final Project App - Nutrition App

Travis Ke, NCCU

May 23, 2025

Contents

1 Functions

1.1 Disclaimer

1. This `.ipynb` file is not meant for the viewer to download directly and run the cells!
2. This is a seperate file to describe the lines of code building the Application. The purpose is to give the viewer or potential collaborator in the future a user manual of our code, and this very file is the first part of it.
3. We only import necessary packages and define functions that we are later going to use in the second seperate `.ipynb` file.

1.2 Importing Necessary Packages

This block imports all the external libraries required for the application's core functionality.

1. `streamlit` is used to build the **interactive web interface**. It allows for real-time updates, layout customization, and dynamic user interaction through input fields, buttons, and widgets.
2. `requests` is used to make **HTTP requests to external APIs**. In this project, it sends queries to the USDA FoodData Central API to retrieve nutritional information about branded food items.
3. `pandas` provides tools for **loading, organizing, and manipulating tabular data using DataFrames**. This is especially useful for storing nutrient values and computing scores across multiple food records.
4. `numpy` supports efficient **numerical computation**, such as calculating BMI or applying formulas to estimate TEE (Total Energy Expenditure).
5. `matplotlib.pyplot` is imported as `plt` and is used to generate **static visualizations**. The application uses it to draw radar charts comparing food nutrient values against reference targets.
6. `json` handles the encoding and decoding of data in JSON format. It is used for both **reading/writing the local user database (`user_db.json`) and parsing API responses**.
7. `os` is used for operating system-level tasks, such as checking if a file exists, building file paths, or saving data to disk.
8. `difflib` provides tools for computing string similarity, which can be useful for fuzzy matching—e.g., when a user enters a food keyword that doesn't exactly match database entries.

At the end of the block, `API_KEY` is defined as a string containing the USDA API key. This key is required to authenticate each request sent to the FoodData Central API and must be included in all API queries.

[1]:
```
import streamlit as st                  # Build web-based front-end interface
import requests                         # Send HTTP requests (e.g., API calls)
import pandas as pd                     # Handle tabular data (commonly with␣
 ↪DataFrame)
import numpy as np                      # Numerical computing with array and matrix␣
 ↪support
import matplotlib.pyplot as plt         # Plotting library for static visualizations
import json                             # Parse and store JSON data
import os                               # OS-level operations like file paths
import difflib                          # Compute string similarity (e.g., fuzzy␣
 ↪matching)

API_KEY = "nqj9Kh3QVKwI4AFfuwGddoSOQznWReylbYLFynzU" # This is the API key to␣
 ↪the USDA API
```

1.3 Function 1. `search_usda_foods(query, api_key, max_results=100)`

1.3.1 Code Explanation

This function `search_usda_foods(query, api_key, max_results=100)` sends a query to the USDA FoodData Central API and returns a list of matching food item IDs (`fdcId`s) based on a search keyword. It is a core utility for initiating branded food lookups based on user input.

1. The `url` variable defines the API endpoint for food search. This is a fixed URL from the USDA API that handles keyword-based food queries.
2. A dictionary named `params` is created to define the query parameters:
   - `api_key` is required for authentication and must be passed with each request.
   - `query` is the actual search keyword, which can be user-defined (e.g., "beef").
   - `pageSize` limits the number of returned results to `max_results`, defaulting to 100.
   - `dataType` is set to `"Branded"` to filter results to only branded food items (excluding generic or foundation foods).
3. A `GET` request is made using `requests.get()` with the URL and query parameters. The result is stored in the `response` object.
4. The function checks the HTTP response status code. If it is not `200` (indicating failure), an error message is shown in the Streamlit app using `st.error()`, and the function returns an empty list.
5. If the request is successful, the response is parsed as JSON, and a list comprehension extracts the `"fdcId"` value from each food item in the `"foods"` list. This list of IDs is returned to be used in downstream API calls for detailed nutrient information.

In summary, this function builds and sends a properly formatted query to the USDA food database, handles failure cases gracefully, and extracts only the minimal required output — the IDs of branded foods matching the user's search.

[15]:
```
def search_usda_foods(query, api_key, max_results=100):
    url = "https://api.nal.usda.gov/fdc/v1/foods/search"  # API endpoint for␣
↪USDA food search

    params = {                                  # Define query parameters
        "api_key": api_key,                     # API key for authentication
        "query": query,                         # Search keyword
        "pageSize": max_results,                # Max number of results to return
        "dataType": ["Branded"]                 # Restrict to branded food items
    }

    response = requests.get(url, params=params)          # Make HTTP GET request

    if response.status_code != 200:                      # Handle failed request
        st.error(f"Search error {response.status_code}") # Display error in␣
↪Streamlit UI
        return []                                        # Return empty list on␣
↪failure

    return [food["fdcId"] for food in                   # Extract list of food␣
↪IDs from JSON response
            response.json().get("foods", [])]
```

1.3.2 Note

The `max_result` is set to 100 to improve the searching speed, because we are going to do many manipulations of the data afterwards. Trimming down the results is necessary for better user experience.

1.3.3 Demo

The result below is the food IDs which can later be used to fetch nutritional facts. As you can see, this is the top 6 results that comes up when you search the keyword, butter, in the database.

[3]:
```
search_usda_foods("butter", API_KEY, max_results=6)
```

[3]:
```
[1920273, 2542726, 2103635, 1932883, 2094280, 2070614]
```

1.3.4 Further Explanation of the Lines of Code that could be Confusing

The last two lines of code can be very confusing to introductory learners of Python.

```
food["fdcId"] for food in response.json().get("foods", [])
# Extract list of food IDs from JSON response
```

In brief, the line of code above is the equivalent to the one below.

```
fdc_ids = []
for food in response.json().get("foods", []):
    fdc_ids.append(food["fdcId"])
```

1.4 Function 2. `fetch_multiple_foods(fdc_ids, api_key)`

1.4.1 Code Explanation

The `fetch_multiple_foods(fdc_ids, api_key)` function sends a batch request to the USDA FoodData Central API to retrieve detailed information about multiple food items, based on their `fdcId`s.

1. The `url` variable stores the endpoint for batch food lookup. This endpoint supports POST requests for fetching information on multiple food IDs at once.
2. A `headers` dictionary is defined to specify the content type as JSON. This tells the server that the request body is formatted as JSON data.
3. The `payload` dictionary contains the key `"fdcIds"` with a list of food item IDs as its value. This is the body of the POST request and indicates which specific items the app wants to retrieve.
4. The `params` dictionary includes the API key under the `"api_key"` field. This is required to authenticate the request and is passed in the URL.
5. A `POST` request is sent using `requests.post()`, which includes the URL, headers, JSON-formatted payload, and the authentication parameters.
6. The function then checks whether the response was successful (`status_code == 200`). If so, it returns the parsed JSON content. If not, it returns an empty list.

This function is essential for efficiently retrieving data about several food items in a single API call, rather than querying each one individually.

[5]:
```
def fetch_multiple_foods(fdc_ids, api_key):

    url = "https://api.nal.usda.gov/fdc/v1/foods"                # API endpoint␣
↪for batch food lookup

    headers = {"Content-Type": "application/json"}               # Specify JSON␣
↪content in POST header
    payload = {"fdcIds": fdc_ids}                                # Payload␣
↪includes list of food IDs
    params = {"api_key": api_key}                                # API key passed␣
↪as URL parameter

    response = requests.post(                                    # Send POST␣
↪request with payload & headers
        url, headers=headers, json=payload, params=params
    )

    return response.json() if response.status_code == 200 else []   # Return JSON␣
↪data or empty list
```

1.4.2 How this function work with other functions

The difference between function 1 and 2 is that function 1 returns a list of food IDs while function 2 returns multiple foods complete details based on a list of food IDs.

The demo for this code is omitted as it returns in a long json data, including food IDs, nutrient IDs, and so many more extra information about the food that we aren't going to used further down the line. The data is so long, I don't even want to put it in this file. However, I still copied and pasted it in a word document, here is the link to the PDF.

External Link: https://drive.google.com/file/d/1_d3IaOdPC57q5NA8358fQuqYYzQbiYIz/view?usp=sharing

We'll need function 3 to extract the necessary data (nutrients, brand name, etc.) and to make it into a Pandas DataFrame.

1.5 Function 3. `extract_nutrients_df(food_list)`

1.5.1 Code Explanation

The `extract_nutrients_df(food_list)` function processes a list of food item dictionaries (typically from the USDA API) and extracts selected nutrient values into a structured `pandas` DataFrame.

1. The dictionary `key_nutrients` maps the USDA nutrient names to cleaner display labels used as column headers. This helps standardize nutrient names despite the complexity of the original API naming.
2. The list `radar_labels` contains the target nutrient columns that should appear in every row. These labels match the values of `key_nutrients` and ensure consistency in the output DataFrame.
3. An empty list `records` is initialized to hold the nutrient data for each food item.
4. The outer `for` loop iterates through each food item in `food_list`. For each food, a new dictionary `row` is initialized containing basic information: the food's description, FDC ID, and brand owner (if available).
5. The inner `for` loop iterates over the list of nutrients in the `"foodNutrients"` field of each food item. For each nutrient, it tries to retrieve the nutrient's name. If the name is found in `key_nutrients`, its corresponding amount is added to the `row` dictionary using the mapped display label.
6. After processing the available nutrients, the loop over `radar_labels` ensures that all expected nutrient fields are present in the `row`. If a nutrient was not extracted, `setdefault()` fills in a value of `0.0`.
7. The fully populated `row` is appended to the `records` list, which accumulates the structured data for each food.
8. Finally, the list of dictionaries is converted into a `pandas` DataFrame using `pd.DataFrame(records)`. This structured table is returned and is ready for further scoring or visualization.

This function plays a key role in transforming raw USDA API responses into clean, uniform tabular data for analysis and charting.

[11]:
```
def extract_nutrients_df(food_list):

    key_nutrients = {                                          # Map USDA nutrient␣
↪names to display labels
        "Energy": "Calories",
        "Protein": "Protein (g)",
        "Total lipid (fat)": "Fat (g)",
        "Carbohydrate, by difference": "Carbs (g)",
        "Sugars, total including NLEA": "Sugar (g)",
        "Total Sugars": "Sugar (g)",
        "Fiber, total dietary": "Fiber (g)",
        "Sodium, Na": "Sodium (mg)"
    }

    radar_labels = [                                           # Nutrient labels to␣
↪ensure column consistency
        "Calories", "Protein (g)", "Fat (g)", "Carbs (g)",
        "Sugar (g)", "Fiber (g)", "Sodium (mg)"]
    records = []                                               # Initialize list to␣
↪hold each row of data

    for food in food_list:
        row = {
            "Food": food.get("description", ""),               # Extract food name
            "FDC ID": food.get("fdcId", ""),                   # Unique ID
            "Brand": food.get("brandOwner", "")                # Brand information
        }

        for item in food.get("foodNutrients", []):             # Loop through each␣
↪nutrient in food item
            name = item.get("nutrient", {}).get("name", "") # Get nutrient name
            if name in key_nutrients:
                row[key_nutrients[name]] = float(item.get("amount", 0))  # Save␣
↪amount if it's in key list

        for label in radar_labels:                             # Ensure all radar␣
↪labels are present
            row.setdefault(label, 0.0)                          # Default to 0 if␣
↪not extracted

        records.append(row)                                    # Append complete␣
↪row to list

    return pd.DataFrame(records)
```

1.6 Function 4. `calculate_tee(gender, age, height, weight, activity_level)`

1.6.1 Code Explanation

The `calculate_tee(gender, age, height, weight, activity_level)` function estimates Total Energy Expenditure (TEE) in kilocalories per day, based on a person's physical attributes and activity level. It applies different equations depending on age group, gender, and activity.

1. The function first checks the `gender`. If `gender == 'male'`, it applies formulas specifically for males; otherwise, it uses female-specific formulas.
2. For males:
   - If `age <= 2`, the function uses an infant-specific TEE formula for boys that incorporates age, height, and weight in a linear equation.
   - If `age < 19`, the function treats the individual as a boy aged 3 to 18. The equation used depends on the `activity_level`, which can be `'inactive'`, `'low active'`, or `'active'`. If the activity level does not match any of these, a fallback formula is applied (often interpreted as "very active" or unspecified).
   - If `age >= 19`, the individual is treated as an adult male. Again, different formulas are applied based on the declared `activity_level`. The constants and coefficients in each case are empirically derived and differ by activity level.
3. For females:
   - If `age <= 2`, a separate infant TEE formula for girls is used, with a different set of coefficients from the male version.
   - If `age < 19`, the function applies formulas for girls aged 3–18. Like the male counterpart, it distinguishes among `inactive`, `low active`, and `active` activity levels, with a fallback formula for other inputs.
   - If `age >= 19`, adult women are handled with yet another set of formulas, each tailored to a different activity level.
4. Each return statement calculates the TEE as a weighted linear combination of age, height, and weight, using different coefficients. These coefficients originate from nutritional science literature and are used to approximate metabolic requirements under various activity conditions.

This function ensures a flexible and detailed estimation of daily caloric needs by accommodating a broad range of demographics and lifestyles. Its output feeds directly into downstream processes, such as macronutrient target calculations and personalized food scoring.

[ ]:
```python
def calculate_tee(gender, age, height, weight, activity_level):
    if gender == 'male':
        if age <= 2:
            # Infant male TEE formula
            return -716.45 - (1.00 * age) + (17.82 * height) + (15.06 * weight)

        elif age < 19:
            # Boys aged 3-18, equations by activity level
            if activity_level == 'inactive':
                return -447.51 - 3.68 * age + 13.01 * height + 13.15 * weight
```

```
            elif activity_level == 'low active':
                return 19.12 + 3.68 * age + 8.62 * height + 20.28 * weight
            elif activity_level == 'active':
                return -388.19 + 3.68 * age + 12.66 * height + 20.46 * weight
            else:  # very active or unknown
                return -671.75 + 3.68 * age + 15.38 * height + 23.25 * weight

        else:
            # Adult male (19 years old), equations by activity level
            if activity_level == 'inactive':
                return 753.07 - 10.83 * age + 6.50 * height + 14.10 * weight
            elif activity_level == 'low active':
                return 581.47 - 10.83 * age + 8.30 * height + 14.94 * weight
            elif activity_level == 'active':
                return 1004.82 - 10.83 * age + 6.52 * height + 15.91 * weight
            else:
                return -517.88 - 10.83 * age + 15.61 * height + 19.11 * weight

    else:  # female
        if age <= 2:
            # Infant female TEE formula
            return -69.15 + 80.0 * age + 2.65 * height + 54.15 * weight

        elif age < 19:
            # Girls aged 3-18
            if activity_level == 'inactive':
                return 55.59 - 22.25 * age + 8.43 * height + 17.07 * weight
            elif activity_level == 'low active':
                return -297.54 - 22.25 * age + 12.77 * height + 14.73 * weight
            elif activity_level == 'active':
                return -189.55 - 22.25 * age + 11.74 * height + 18.34 * weight
            else:
                return -709.59 - 22.25 * age + 18.22 * height + 14.25 * weight

        else:
            # Adult female (19 years old)
            if activity_level == 'inactive':
                return 584.90 - 7.01 * age + 5.72 * height + 11.71 * weight
            elif activity_level == 'low active':
                return 575.77 - 7.01 * age + 6.60 * height + 12.14 * weight
            elif activity_level == 'active':
                return 710.25 - 7.01 * age + 6.54 * height + 12.34 * weight
            else:
                return 511.83 - 7.01 * age + 9.07 * height + 12.56 * weight
```

1.6.2 Reference

Our source for the TEE calculation method can be found in this site: https://nap.nationalacademies.org/read/26818/chapter/7#83

Reference: National Academies of Sciences, Engineering, and Medicine. (2023). Applications of the Dietary Reference Intakes for Energy. *In Dietary reference intakes for energy* (Chapter 5, pp. 84–85). The National Academies Press. https://doi.org/10.17226/26818

1.7 Function 5. `compute_target_macros_per_meal(tee)`

The `compute_target_macros_per_meal(tee)` function calculates the target intake (in grams) of protein, fat, and carbohydrates per meal, based on a person's Total Energy Expenditure (TEE). It uses fixed macronutrient ratios and standard energy conversion factors.

1. The function assumes a macronutrient distribution of:
   - 40% of total calories from protein,
   - 30% from fat,
   - 30% from carbohydrates.
2. For protein:
   - Calories from protein = `TEE * 0.4`
   - Since each gram of protein provides 4 kcal, divide by 4 to convert to grams.
   - Assuming 3 meals per day, divide again by 3 to get per-meal target.
   - Formula: `tee * 0.4 / 4 / 3`
3. For fat:
   - Calories from fat = `TEE * 0.3`
   - Fat has 9 kcal per gram, so divide by 9 to get grams.
   - Divide by 3 to get the per-meal target.
   - Formula: `tee * 0.3 / 9 / 3`
4. For carbohydrates:
   - Calories from carbs = `TEE * 0.3`
   - Carbs also provide 4 kcal per gram, so divide by 4.
   - Divide by 3 for per-meal distribution.
   - Formula: `tee * 0.3 / 4 / 3`

The function returns a dictionary with keys `"Protein (g)"`, `"Fat (g)"`, and `"Carbs (g)"`, each containing the computed gram-based per-meal recommendation. This output can then be used to score food items against personalized nutritional targets.

[ ]:
```
def compute_target_macros_per_meal(tee):
    return {
        "Protein (g)": tee * 0.4 / 4 / 3,   # 40% of calories → divide by 4 kcal/
↪g → 3 meals
        "Fat (g)":     tee * 0.3 / 9 / 3,   # 30% of calories → divide by 9 kcal/
↪g → 3 meals
        "Carbs (g)":   tee * 0.3 / 4 / 3    # 30% of calories → divide by 4 kcal/
↪g → 3 meals
    }
```

1.7.1 Reference

The 433-rule is set by our teammate, Jian-Hao Lin. He got dietary advices from multiple fitness coaches and discovered that their advices for daily nutrients intake can be funneled down to this rule.

1.8 Function 6. `score_menu(df, targets, tee, goal)`

The `score_menu(df, targets, tee, goal)` function scores each food item in a DataFrame against personalized nutritional targets. It uses different scoring logic and goal-specific weights to compute a final ranking for food selection.

1. Two helper functions are defined:
   - `bounded_score(x, t)` computes a linear score where the maximum score is `1` if the actual value `x` is at or above the target `t`. Otherwise, it returns a fraction `x/t`, giving partial credit for under-target values.
   - `penalized_score(x, t)` is designed to penalize overconsumption. It returns a decreasing score from `2 - x/t` when `x > t`, with a lower bound of 0. When under or on target, it behaves like `x/t`.
2. The function calculates individual nutrient scores for each row in the DataFrame:
   - `"Calories Score"` uses `penalized_score`, since excess calories are typically undesirable.
   - `"Protein Score"` uses `bounded_score`, rewarding high protein intake up to the target.
   - `"Fat Score"` and `"Carbs Score"` both use `penalized_score`, discouraging excessive intake relative to the target.
3. A `weights` dictionary is defined to assign different importance to each nutrient based on the user's `goal`:
   - `"muscle_gain"` gives highest weight to protein (0.4), while balancing calories, fat, and carbs equally at 0.2.
   - `"fat_loss"` emphasizes both calorie control and protein (0.3 and 0.4 respectively), with moderate attention to fat and carbs.
4. A new column `"Total Score"` is calculated as the weighted sum of the four nutrient scores:
   - Each nutrient score is multiplied by its corresponding weight, and the results are summed to yield the final score.
5. The DataFrame is sorted by `"Total Score"` in descending order using `df.sort_values()`, so that the top-ranked foods (those closest to the dietary goal) appear first.

This function operationalizes the trade-offs between nutrient intake and dietary objectives, converting raw nutrition data into an actionable ranking system tailored to the user's TEE and macro needs.

```
[ ]: def score_menu(df, targets, tee, goal):
         # Helper: score = x / t if under target, else penalize
         def bounded_score(x, t): return min(x / t, 1)

         # Penalize over-target macros with a decreasing function (2 - x/t), min 0
         def penalized_score(x, t): return max(0, 2 - x / t) if x > t else x / t

         # Individual nutrient scores
         df["Calories Score"] = df["Calories"].apply(lambda x: penalized_score(x,␣
     ↪tee))
         df["Protein Score"]  = df["Protein (g)"].apply(lambda x: bounded_score(x,␣
     ↪targets["Protein (g)"]))
         df["Fat Score"]      = df["Fat (g)"].apply(lambda x: penalized_score(x,␣
     ↪targets["Fat (g)"]))
         df["Carbs Score"]    = df["Carbs (g)"].apply(lambda x: penalized_score(x,␣
     ↪targets["Carbs (g)"]))

         # Different weights for different goals
         weights = {
             "muscle_gain": [0.2, 0.4, 0.2, 0.2],  # Emphasize protein for bulking
             "fat_loss":    [0.3, 0.4, 0.3, 0.2]   # Balance between calorie and␣
     ↪protein
         }[goal]

         # Weighted total score = sum of nutrient scores × weights
         df["Total Score"] = (
             df["Calories Score"] * weights[0] +
             df["Protein Score"]  * weights[1] +
             df["Fat Score"]      * weights[2] +
             df["Carbs Score"]    * weights[3]
         )

         return df.sort_values("Total Score", ascending=False)  # Highest score first
```

1.8.1 Reference

The scoring system is founded by our team-mates, Jian-Hao Lin, Ming-Chian Tsiang, and Bo-Yu Chuang. They just came up with this scoring system that gives a macro score to the food based on the consumer's fitness goal and the food's nutrients.

1.9 Function 7. `plot_radar_chart(row)`

The `plot_radar_chart(row)` function generates a radar chart (also known as a spider chart) to visually compare a food item's nutrient content with standard daily recommended values. This visualization helps assess how well a food meets nutritional goals across multiple dimensions.

1. A list called `labels` defines the nutrients to be visualized on the radar chart. These include `"Calories"`, `"Protein (g)"`, `"Fat (g)"`, `"Carbs (g)"`, `"Sugar (g)"`, `"Fiber (g)"`, and `"Sodium (mg)"`.
2. The dictionary `daily` specifies standard daily recommended values for each of these nutrients. These serve as the normalization baselines:
    - e.g., 2000 kcal for calories, 50g for protein and sugar, 70g for fat, etc.
3. The `values` list is constructed by dividing each nutrient value from `row` by its corresponding daily value. This normalizes each nutrient so that a value of `1.0` means the food meets 100% of the recommended daily amount. The first value is appended again to `values` to close the radar chart loop and form a complete shape.
4. The `angles` variable creates evenly spaced angle coordinates for the radar plot using `np.linspace`. It spans from `0` to `2π`, evenly dividing the circle by the number of nutrient categories. The starting point is repeated at the end to complete the shape.
5. A radar chart is initialized with `plt.subplots()` using the `polar=True` argument, which specifies the use of a polar coordinate system.
6. The data is plotted as a filled polygon:
    - `ax.plot(angles, values)` draws the outline.
    - `ax.fill(angles, values, alpha=0.25)` adds a translucent fill to visually emphasize the area.
7. Plot styling is handled as follows:
    - `ax.set_xticks()` and `ax.set_xticklabels()` set the axis ticks and nutrient labels around the circle.
    - `ax.set_ylim(0, 1)` scales all radial axes from 0 to 1 (i.e., from 0% to 100% of daily value).
    - `ax.set_title()` sets the food name as the chart title.
8. Finally, `st.pyplot(fig)` renders the chart in the Streamlit interface.

This function transforms raw nutrition data into an intuitive and comparable visual, allowing users to instantly see how a food item measures up against standard nutrient benchmarks.

[12]:
```
def plot_radar_chart(row):
    labels = [                                          # Nutrients to include in␣
↪the radar chart
        "Calories", "Protein (g)", "Fat (g)", "Carbs (g)",
        "Sugar (g)", "Fiber (g)", "Sodium (mg)"
    ]

    daily = {                                           # Daily recommended values␣
↪for each nutrient
        "Calories": 2000, "Protein (g)": 50, "Fat (g)": 78,
        "Carbs (g)": 300, "Sugar (g)": 50, "Fiber (g)": 28, "Sodium (mg)": 2300
    }

    values = [row[l] / daily[l] for l in labels]      # Normalize each nutrient␣
↪by daily value
    values += [values[0]]                               # Close the radar shape by␣
↪repeating the first value

    angles = np.linspace(0, 2 * np.pi, len(labels), endpoint=False).tolist() +␣
↪[0]  # Radar chart angles

    fig, ax = plt.subplots(figsize=(4, 4), subplot_kw=dict(polar=True))  #␣
↪Create polar (radar) plot
    ax.plot(angles, values)                             # Draw the outline
    ax.fill(angles, values, alpha=0.25)                 # Fill the area with␣
↪transparency

    ax.set_xticks(angles[:-1])                          # Set axis ticks
    ax.set_xticklabels(labels, fontsize=8)              # Set tick labels␣
↪(nutrients)
    ax.set_ylim(0, 1)                                   # Set radial axis from 0 to␣
↪100% of DV
    ax.set_title(row["Food"], y=1.1)                    # Title = food name

    st.pyplot(fig)                                      # Render plot in Streamlit
```

1.9.1 Reference

U.S. Food and Drug Administration. (2022, July 25). *Daily value on the nutrition and supplement facts labels.* FDA. https://www.fda.gov/food/nutrition-facts-label/daily-value-nutrition-and-supplement-facts-labels

1.10 Function 8. `estimate_speed_bmi_age(activity, bmi, age)`

The `estimate_speed_bmi_age(activity, bmi, age)` function estimates a person's average exercise speed in km/h based on their selected activity type, Body Mass Index (BMI), and age. The function returns a value rounded to two decimal places and incorporates realistic adjustments for weight and age.

1. A dictionary called `base_speeds` is defined to associate each activity with a default average speed in kilometers per hour:
   - `"Running"`: 9.0 km/h
   - `"Swimming"`: 3.0 km/h
   - `"Cycling"`: 15.0 km/h
   - `"Walking"`: 5.0 km/h
2. The function retrieves the base speed for the given `activity` using `base_speeds.get(activity, 5.0)`. If the input activity is not found in the dictionary, a default value of 5.0 km/h is used.
3. A conditional adjustment is applied if the user is overweight. If `bmi > 25`, the base speed is reduced by 10% by multiplying the speed by `0.9`.
4. A second conditional adjustment is made if the user is above the age of 40. If `age > 40`, the speed is further reduced by 5% by multiplying by `0.95`.
5. The function concludes by rounding the final result to two decimal places using `round(speed, 2)` and returns the adjusted estimate.

This function is particularly useful for estimating exercise duration or distance in other parts of the app where calorie burn is converted into real-world physical activity.

[ ]:
```
def estimate_speed_bmi_age(activity, bmi, age):
    base_speeds = {                          # Define default speeds by activity␣
↪type
        "Running": 9.0,
        "Swimming": 3.0,
        "Cycling": 15.0,
        "Walking": 5.0
    }

    speed = base_speeds.get(activity, 5.0) # Use default of 5.0 if activity is␣
↪unknown

    if bmi > 25:                             # Reduce speed by 10% if overweight
        speed *= 0.9

    if age > 40:                             # Reduce speed by 5% if older
        speed *= 0.95

    return round(speed, 2)                  # Round final result to 2 decimal␣
↪places
```

1.10.1 Reference

The estimated speeds for different exercises and for people with different BMI is just a rough gauge of how the speed varies.

1.11 Function 9. `calories_to_exercise_with_distance(calories, bmi, age)`

The `calories_to_exercise_with_distance(calories, bmi, age)` function estimates the amount of time and distance required to burn a given number of calories for different physical activities, taking into account the user's BMI and age to adjust movement speed.

1. A dictionary named `activities` is defined, which maps four types of physical activity to their estimated energy expenditure rate (in kcal per minute):
   - `"Running"`: 10 kcal/min
   - `"Swimming"`: 14 kcal/min
   - `"Cycling"`: 8 kcal/min
   - `"Walking"`: 4 kcal/min
2. An empty dictionary called `result` is initialized to store output for each activity.
3. A `for` loop iterates through each `(activity, kcal_per_min)` pair in the `activities` dictionary.
4. For each activity:

- The number of minutes needed to burn the given `calories` is calculated using the formula:
  `minutes = calories / kcal_per_min`.
- The effective movement speed (in km/h) is estimated using the helper function `estimate_speed_bmi_age(activity, bmi, age)`, which adjusts for weight and age.
- The corresponding distance (in kilometers) is computed using:
  `distance = (minutes / 60) * speed`,
  which converts minutes to hours and then applies the speed.

5. The output for each activity is stored in the `result` dictionary, with three fields:
   - `"time_min"`: Rounded number of minutes required to burn the calories.
   - `"distance_km"`: Rounded distance (in kilometers), up to 2 decimal places.
   - `"speed_kmh"`: Raw adjusted speed in km/h.
6. The function finally returns the complete `result` dictionary, providing a summary of duration, distance, and speed per activity based on caloric burn.

[ ]:
```python
def calories_to_exercise_with_distance(calories, bmi, age):
    activities = {                              # Activity name and estimated kcal␣
↪burned per minute
        "Running": 10,
        "Swimming": 14,
        "Cycling": 8,
        "Walking": 4
    }

    result = {}

    for activity, kcal_per_min in activities.items():
        minutes = calories / kcal_per_min      # Time needed to burn target␣
↪calories
        speed = estimate_speed_bmi_age(activity, bmi, age)  # Adjusted speed (km/
↪h)
        distance = (minutes / 60) * speed      # Convert time to hours × speed =␣
↪distance

        result[activity] = {
            "time_min": round(minutes),        # Time in minutes
            "distance_km": round(distance, 2),# Distance in km, rounded to 2␣
↪decimals
            "speed_kmh": speed                 # Raw speed value
        }

    return result
```

1.11.1 Reference

The estimated calorie-burns for different exercises and for people with different BMI is just a rough gauge.

1.12 Function 10. `calculate_bmr(gender, age, height, weight)`

The `calculate_bmr(gender, age, height, weight)` function calculates the Basal Metabolic Rate (BMR) using the Mifflin-St Jeor Equation. BMR estimates the number of calories a person burns at rest, and it varies based on gender, age, height, and weight.

1. The function starts by evaluating the `gender` input using `gender.lower()` to make the comparison case-insensitive. This allows inputs like `"Male"` or `"MALE"` to be treated the same as `"male"`.
2. If the gender is `"male"`, the function uses the male-specific BMR formula:

   `BMR = 10 * weight + 6.25 * height - 5 * age + 5`

   This formula assumes weight in kilograms, height in centimeters, and age in years.
3. If the gender is anything other than `"male"` (implicitly assuming `"female"`), the function uses the female-specific formula:

   `BMR = 10 * weight + 6.25 * height - 5 * age - 161`

   The structure is the same, but the constant at the end changes from `+5` (for males) to `-161` (for females), reflecting the physiological difference in metabolic rates.
4. The use of `return` ensures that the function immediately exits and provides the computed BMR value for use in downstream calculations, such as TEE (Total Energy Expenditure).

This function encapsulates gender-based BMR logic in a compact and efficient manner, enabling it to plug seamlessly into energy and nutrition tracking workflows.

[14]:
```python
def calculate_bmr(gender, age, height, weight):
    if gender.lower() == "male":
        return 88.362 + (13.397 * weight) + (4.799 * height) - (5.677 * age)
    else:
        return 10 * weight + 6.25 * height - 5 * age - 161
        return 447.593 + (9.247 * weight) + (3.098 * height) - (4.330 * age)
```

1.12.1 Reference

Garnet Health. (2016, July 1). *Basal metabolic rate calculator.* https://www.garnethealth.org/news/basal-metabolic-rate-calculator

2 App Interface Design with Streamlit

2.1 Disclaimer

1. This `.ipynb` file is not meant for the viewer to download directly and run the cells!
2. This is a seperate file to describe the lines of code building the Application. The purpose is to give the viewer or potential collaborator in the future a user manual of our code, and this very file is the second part of it.
3. If you run this file directly, it will first generate errors beacuse the packages aren't imported and the functions that we defined is not written in this very file. All of which are written in the seperate `.ipynb` file named `functions_lecnote.ipynb`, which can be seen as the previous chapter of the user manual.

2.2 Part 1 Logo & Setting up / Initializing User Data Base

2.2.1 Code Explanation

The `USER_DB` initialization block handles loading and preparing user data for the application.

1. A variable `logo_url` stores the URL of an image icon used in the app's header, which will be displayed in the Streamlit interface.
2. A variable `USER_DB_PATH` is assigned the value `"user_db.json"` to specify the local file used for saving and loading user data.
3. The script checks if the file at `USER_DB_PATH` exists using `os.path.exists()`:
   - If it exists, the file is opened in read mode and parsed using `json.load(f)` to populate the `USER_DB` dictionary.
   - If it does not exist, a default dictionary `USER_DB` is created manually with sample users.
4. Each user in `USER_DB` (e.g., `"alice"` and `"bob"`) contains the following fields:
   - `"password"`: a plain text password for login
   - `"gender"`: either `"male"` or `"female"`
   - `"age"`: numeric value in years
   - `"height"`: height in centimeters
   - `"weight"`: weight in kilograms
   - `"activity_level"`: one of `"inactive"`, `"low active"`, `"active"`, or `"very active"`
   - `"goal"`: either `"fat_loss"` or `"muscle_gain"` to indicate their objective

[2]:
```
# --- Logo Link ---
logo_url = "https://cdn-icons-png.flaticon.com/512/590/590685.png"  # Logo URL␣
 ↪used in Streamlit header

# --- File path for user DB persistence ---
USER_DB_PATH = "user_db.json"  # Local file to persist user data

# --- Load user DB from file if exists ---
if os.path.exists(USER_DB_PATH):
    with open(USER_DB_PATH, "r") as f:
```

```
        USER_DB = json.load(f)                  # Load user data from JSON file
else:
    USER_DB = {                                 # If file doesn't exist, initialize␣
↪in-memory user DB
        "alice": {
            "password": "1234",                 # Simple password (not secure for␣
↪real apps)
            "gender": "female",
            "age": 28,
            "height": 160,                      # in cm
            "weight": 55,                       # in kg
            "activity_level": "active",         # User-reported activity level
            "goal": "fat_loss"                  # Goal: either "fat_loss" or␣
↪"muscle_gain"
        },
        "bob": {
            "password": "5678",
            "gender": "male",
            "age": 30,
            "height": 175,
            "weight": 70,
            "activity_level": "inactive",
            "goal": "muscle_gain"
        }
    }
```

2.2.2 Further Explanation for Lines of Code that could be Confusing

Line 9

1. `open(USER_DB_PATH, "r")`

   `open()` is a built-in Python function used to open a file.

   `"r"` is the mode, which stands for read. It means you want to read the file, not write to or modify it.

   `USER_DB_PATH` is the file path — in this case, `"user_db.json"`, which is a JSON file used to store user data.

2. `with ... as f:`

   This is Python's `with` statement, which ensures the file is properly closed after being opened — even if an error occurs.

   `f` is the name given to the file object. You can use it to read the contents of the file.

2.3 Part 2 Laying Out the Page and Setting What Goes to the Header

2.3.1 Code Explanation

The Streamlit setup and header display logic ensures a polished layout with branding.

1. The `st.set_page_config(layout="centered")` function configures the Streamlit page layout so that all content is centered. This enhances visual balance and makes the app look more professional on all screen sizes.
2. The line `col_logo, col_title = st.columns([2, 6])` creates a layout with two columns using Streamlit's layout API. The first column (1/4 width) is for the logo, and the second column (3/4 width) is for the title and subtitle text.
3. Inside the `with col_logo:` block, the logo image stored at `logo_url` is rendered using `st.image()`, scaled to a width of 150 pixels to fit cleanly into the layout.
4. The `with col_title:` block renders the following elements:
   - `st.markdown("## Nutrition Scoring App 2.5.3")`: A markdown-based header for the app title.
   - `st.caption("Your personalized guide to smarter food choices!")`: A short descriptive subtitle under the title.
   - `st.caption("A Python Project Created by Group 02 with Python and Streamlit")`: A final caption crediting the development team and tools used.

[ ]:
```python
# --- Streamlit page setup ---
st.set_page_config(layout="centered")  # Set layout to centered (better visual␣
↪balance)

# --- Header with logo aligned to title ---
col_logo, col_title = st.columns([2, 6])  # Two columns: logo (1/4 width), title␣
↪(3/4 width)

with col_logo:
    st.image(logo_url, width=150)  # Display logo image at defined width

with col_title:
    st.markdown("## Nutrition Scoring App 2.5.3")  # Main title (Markdown style)
    st.caption("Your personalized guide to smarter food choices!")  # Subtitle␣
↪or tagline
    st.caption("A Python Project Created by Group 02 with Python and Streamlit")␣
↪ # Credit line
```

2.4 Part 3 Logging in

2.4.1 Code Explanation

The `login/register` interface uses Streamlit session state to manage user authentication and persist login data across interactions.

1. The code first checks whether the key `"logged_in"` exists in `st.session_state`. If it doesn't, it initializes the login state by setting `st.session_state.logged_in = False`.
2. If the user is not logged in, the app displays a login/registration form. A radio button `st.radio()` lets the user choose between `"Login"` and `"Register"` modes.
3. In `"Login"` mode:

- Two text input fields collect the `username` and `password`, with password input masked.
- When the `"Login"` button is clicked:
  - It verifies the entered credentials by checking if the `username` exists in `USER_DB` and the stored password matches.
  - If valid, it sets three session state variables: `logged_in`, `user_profile`, and `username`, and triggers a rerun using `st.rerun()`.
  - If the login fails, it shows an error message using `st.error()`.

4. In `"Register"` mode:

   - Three password fields are displayed to input a `username`, `new_pass`, and `confirm_pass`.
   - Additional profile information is collected using input fields and dropdowns:
     - `gender`, `age`, `height`, `weight`, `activity_level`, and `goal`.

5. Upon clicking `"Register"`:

   - The program performs a series of checks:
     - If the `username` already exists in `USER_DB`, show a username conflict error.
     - If `new_pass` and `confirm_pass` do not match, show a password mismatch error.
     - If the username is shorter than 3 characters or password shorter than 4, show a warning.
   - If all checks pass:
     - A new user record is added to `USER_DB` with all profile info.
     - It tries to save `USER_DB` to disk using `json.dump()` inside a `try-except` block.
     - On success, it updates the session state and displays a success message.

6. Finally, `st.stop()` is called to prevent the rest of the app from rendering unless the user has successfully logged in.

[ ]:
```python
# --- Track login state ---
if "logged_in" not in st.session_state:
    st.session_state.logged_in = False  # Initialize login state

# --- Show login/register form if not logged in ---
if not st.session_state.logged_in:
    st.markdown("##  Member Access")
    auth_mode = st.radio("Choose action", ["Login", "Register"])

    # --- Login form ---
    if auth_mode == "Login": # What shows if user choose the Login action
        username = st.text_input("Username")
        password = st.text_input("Password", type="password")

        if st.button("Login"): # What happens next if user hits the Login button
            if username in USER_DB and USER_DB[username]["password"] == password:
                st.session_state.logged_in = True # Change the session state␣
↪from not logged in to logged in
                st.session_state.user_profile = USER_DB[username]  # Load full␣
↪user data
                st.session_state.username = username
```

```
                st.rerun()
            else:
                st.error(" Invalid username or password")

    # --- Registration form ---
    elif auth_mode == "Register": # What shows if user choose the Register action
        new_user = st.text_input("Choose a username(at least 3 characters)")
        new_pass = st.text_input("Create a password(at least 4 characters)",␣
↪type="password")
        confirm_pass = st.text_input("Confirm password(at least 4 characters)",␣
↪type="password")

        # Profile information fields
        st.markdown("###  Profile Info")
        col1, col2 = st.columns(2)
        with col1:
            gender = st.selectbox("Biological Sex", ["male", "female"])
            age = st.number_input("Age", 1, 99, 25)
            height = st.number_input("Height (cm)", 100, 250, 165)
        with col2:
            weight = st.number_input("Weight (kg)", 30, 150, 60)
            activity_level = st.selectbox("Activity level", ["inactive", "low␣
↪active", "active", "very active"])
            goal = st.selectbox("Goal", ["muscle_gain", "fat_loss"])

        # Validation and account creation
        if st.button("Register"):
            if new_user in USER_DB: # Username has to be unique
                st.error(" Username already taken.")
            elif new_pass != confirm_pass: # Password must match to confirm the␣
↪user correctly typed in his/her desired password
                st.error(" Passwords do not match.")
            elif len(new_user) < 3 or len(new_pass) < 4:
                st.warning(" Username must be 3+ characters, password 4+.")
            else:
                # Save new user data by updating the previous user database
                USER_DB[new_user] = {
                    "password": new_pass,
                    "gender": gender,
                    "age": age,
                    "height": height,
                    "weight": weight,
                    "activity_level": activity_level,
                    "goal": goal
                }

                try:
```

```
                    with open(USER_DB_PATH, "w") as f:
                        json.dump(USER_DB, f, indent=4)
                        print(" Saved USER_DB")
                except Exception as e:
                    st.error(f" Error saving user: {e}")

                st.session_state.logged_in = True # Change the session state␣
↪from not logged in to logged in
                st.session_state.user_profile = USER_DB[new_user] # Log in with␣
↪the new user's profile
                st.session_state.username = new_user
                st.success(" Registration successful! Logging you in...")
                st.rerun()

    st.stop()  # Prevent rendering other UI before login
```

2.4.2 Further Explanation for Lines of Code that could be Confusing

Line 62 to 68 The purpose of try and except is to prevent your app from crashing if saving fails.

1. `try:`
   Starts a try block — this is where we put code that *might fail*.
2. `with open(USER_DB_PATH, "w") as f:`
   Tries to open the file defined by `USER_DB_PATH` in **write mode** (`"w"`).
   - If the file doesn't exist, Python will create it.
   - If it exists, Python will **overwrite** it.
3. `json.dump(USER_DB, f, indent=4)`

   The `json.dump()` is a method from Python's built-in json module. It writes a Python object (like a dictionary or list) to a file in JSON format.

   In this case, it takes the `USER_DB` Python dictionary and **saves it in JSON format** to the file `f`.
   - `indent=4` makes it human-readable (pretty printed).
4. `print(" Saved USER_DB")`
   Shows a message in the terminal (not Streamlit) that saving succeeded.
5. `except Exception as e:`
   If anything goes wrong (like the path doesn't exist), Python jumps here.
6. `st.error(f" Error saving user: {e}")`
   Shows an error message in the **Streamlit app UI**, using the exception `e`.
7. `return`
   Stops further execution (especially inside a function).

2.5 Part 4 After Logged in - Logging Out

2.5.1 Code Explanation

The `sidebar logout` block provides a user-friendly way to display the current login status and allow the user to log out via a sidebar interface in Streamlit.

1. It checks if both `"logged_in"` and `"username"` exist in `st.session_state`. These conditions confirm the user is logged in.
2. If the user is authenticated:
    - A welcome message is shown in the sidebar using `st.sidebar.success()`, dynamically displaying the current `username`.
3. Below the greeting, a logout button labeled `" Logout"` is rendered using `st.sidebar.button()`.
4. When the logout button is clicked, the `on_click` callback triggers `st.session_state.clear()`, which removes all session variables, effectively logging the user out and resetting the app state.

[ ]:
```
# --- Sidebar logout ---
if st.session_state.get("logged_in") and st.session_state.get("username"):
    st.sidebar.success(f" Logged in as {st.session_state.username}")  # Welcome␣
 ↪message
    st.sidebar.button(
        " Logout",
        on_click=lambda: st.session_state.clear()  # Clear session on logout
    )
```

2.6 Part 5 After Logged in - The Searching Engine

2.6.1 Code Explanation

This section defines the main logic flow of the app, combining user profile input, energy calculations, search functionality, and result visualization.

1. If the `user_profile` exists in `st.session_state`, it extracts the user's data fields including `gender`, `age`, `height`, `weight`, `activity_level`, and `goal`.
2. The sidebar displays the extracted profile using `st.sidebar.write()` with formatted text.
3. Energy needs are calculated using two custom functions:
    - `calculate_tee()` for Total Energy Expenditure (TEE).
    - `calculate_bmr()` for Basal Metabolic Rate (BMR). The BMI is also computed using the standard formula.
4. The estimated time and distance required to burn 1/3 of TEE (one meal's worth) are computed via `calories_to_exercise_with_distance()` and shown in the sidebar per activity.
5. In the main panel, users can enter a food keyword (default is `"beef"`) to search USDA's API.
6. Once the "Find Foods" button is clicked:

- Step 1: The `search_usda_foods()` function retrieves a list of `fdcIds` using the keyword.
- Step 2: These IDs are passed to `fetch_multiple_foods()` to retrieve full nutritional data.
- Step 3: The raw API data is processed into a structured `DataFrame` using `extract_nutrients_df()`.

7. Step 4: If the dataset contains calorie data, the app estimates how much activity is needed to burn the average calories.
8. Step 5: Using the user profile, the TEE and target macronutrients per meal are calculated. Then `score_menu()` is called to generate a weighted score for each food based on how closely it matches the user's nutritional targets.
9. The scored results are displayed in a table using `st.dataframe()` and introduced with a formatted subheader.
10. Finally, a radar chart is rendered for each food using `plot_radar_chart()`, iterating through each row and displaying the nutrient breakdown visually in two-column layout using `st.columns(2)`.

[ ]:
```python
# --- Input Section (for logged-in users only) ---
if "user_profile" in st.session_state:
    profile = st.session_state.user_profile

    # Extract personal info from session state
    gender = profile["gender"]
    age = profile["age"]
    height = profile["height"]
    weight = profile["weight"]
    activity_level = profile["activity_level"]
    goal = profile["goal"]

    # --- Sidebar: Display user profile and metrics ---
    st.sidebar.markdown("###  Your Profile")
    st.sidebar.write(f"** Gender:** {gender}")
    st.sidebar.write(f"** Age:** {age} years")
    st.sidebar.write(f"** Height:** {height} cm")
    st.sidebar.write(f"** Weight:** {weight} kg")
    st.sidebar.write(f"** Activity Level:** {activity_level}")
    st.sidebar.write(f"** Goal:** {goal.replace('_', ' ').title()}")

    # This is later added just for the demo, it shows the usernames registered.
    st.sidebar.markdown("###  Show Registered Users (Dev Only)")
    st.sidebar.write("Registered usernames:")
    st.sidebar.write(list(USER_DB.keys()))

    # Calculate energy needs and activity equivalents
    tee = calculate_tee(gender, age, height, weight, activity_level)      #␣
↪Total Energy Expenditure
```

```
    bmi = weight / ((height / 100) ** 2)                                    #␣
↪Body Mass Index
    burn_data = calories_to_exercise_with_distance(tee / 3, bmi, age)      # Burn␣
↪1 meal worth of kcal

    bmr = calculate_bmr(gender, age, height, weight)                        #␣
↪Basal Metabolic Rate

    # --- Sidebar: Display BMR and TEE results ---
    st.sidebar.markdown("###  Daily Energy Estimates")
    st.sidebar.write(f"** BMR:** **{round(bmr)}** kcal/day")
    st.sidebar.write(f"** TEE:** **{round(tee)}** kcal/day")

    # --- Sidebar: Display burn estimates for 1/3 TEE ---
    st.sidebar.markdown("###  Burn 1 Meal (~ TEE):")
    for activity, stats in burn_data.items():
        st.sidebar.write(f"**{activity}**: {stats['time_min']} min ␣
↪{stats['distance_km']} km")

    # --- Main panel: search bar ---
    st.markdown("###  Search Food by Keyword")
    keyword = st.text_input("Search food keyword", value="beef")  # default =␣
↪"beef"
    submitted = st.button(" Find Foods")

    # --- Search logic begins ---
    if submitted:
        # Step 1: Search fdcIds by keyword
        fdc_ids = search_usda_foods(keyword, API_KEY)

        # Step 2: Fetch nutrient data by fdcIds
        foods = fetch_multiple_foods(fdc_ids, API_KEY)

        # Step 3: Convert to structured dataframe
        df = extract_nutrients_df(foods)

        # Step 4: (Optional) Estimate how much effort needed to burn average␣
↪food calories
        if "Calories" in df.columns:
            avg_calories = df["Calories"].mean()
            bmi = weight / ((height / 100) ** 2)
            exercise_data = calories_to_exercise_with_distance(avg_calories,␣
↪bmi, age)

        # Step 5: Score food based on user profile
        tee = calculate_tee(gender, age, height, weight, activity_level)
```

```
        targets = compute_target_macros_per_meal(tee)
        scored = score_menu(df, targets, tee, goal)

        # --- Output ranked results ---
        st.subheader(
            f" Top Foods for '{keyword}' (Goal: {goal.replace('_', ' ').
↪title()})"
        )
        st.dataframe(scored.head(10))  # Show table with ranking

        # --- Show radar charts for each food item ---
        cols = st.columns(2)  # Two-column layout for radar charts
        for i, (_, row) in enumerate(scored.head(10).iterrows()):
            with cols[i % 2]:
                st.markdown(f"####  {row['Food']} - {row['Brand']}")
                plot_radar_chart(row)
```

2.6.2 Note

We only show the top 10 results based on our scoring system because we are also going to draw a radar chart for each item, we don't want to overwhelm the user. Also, it's unnecessary to show all 100 results after we scored the items, the whole point of scoring is to seek the top results.

2.6.3 Further Explanation for Lines of Code that could be Confusing

Line 74 to 75

1. `scored.iterrows()` returns each row in the DataFrame as a `(index, row)` tuple.
2. `enumerate(...)` adds a counter `i` (0, 1, 2, ...).
3. `_` is used to ignore the original index since it's not needed. The reason it's not needed is because it has no meaning after we sort the data based on nutrient score.
4. `row` contains the actual data from the DataFrame.
5. `cols[i % 2]` alternates between `cols[0]` and `cols[1]` using the modulo operator (`% 2`), placing content in the left and right columns in turn.
6. The `with` block scopes layout content inside the chosen column.