*Detail documentation of the code*

**Introduction:**

This document provides an in-depth explanation of the Streamlit application code provided. This code primarily aims to fetch and visualize weather data, particularly focusing on sunshine duration and wind speed for different cities. It then calculates the total green energy generated based on solar and wind energy, comparing these values across different cities and dates to determine optimal data transfer locations.

**Imports and Setup:**

*A screenshot of a computer code

Description automatically generated*

1. Streamlit: Used to create interactive web applications for data science projects.
2. Pandas: Used for data manipulation and analysis, particularly with DataFrames.
3. Requests: Used to send HTTP requests to fetch data from web APIs.
4. Altair: Used for declarative statistical visualization in Python.
5. Datetime: Used to manipulate dates and times.
6. Matplotlib: Used for creating static, interactive, and animated visualizations in Python.
7. Numpy: Used for numerical computations and handling arrays.
8. Seaborn: Used for statistical data visualization, built on top of Matplotlib.
9. Plotly Express: Used for creating quick and easy interactive plots.
10. Plotly Graph Objects: Used for creating detailed and customizable interactive visualizations.
11. Geopy: Used to calculate the distance between two geographic locations.

**Variables for the Model:**

A screenshot of a computer

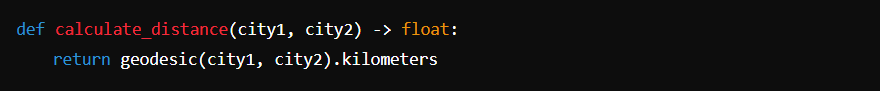
Description automatically generated

These variables are used in energy calculation formulas within the application. They can be changed to see the impact of some variable on the end results.

Those variables have been chosen after a bit of research, but it’s not the optimal way to calculate the green energy generated in a specific city. The optimal goal would be to change those integer variables into open-data API. Like that the code will fetch automatically the number of turbines or the number of solar panels in the decided city.

**Functions:**

1. Calculate the distance:



This function calculates the distance in kilometers between two cities using their geographical coordinates. It **should not** be touch, as it’s already up to date.

1. Fetch Weather Data

A computer screen with text on it

Description automatically generated

This function is designed to fetch weather data for a given city. It takes three parameters:

* **“city\_name”**: The name of the city (not used directly in the function but could be used for labeling or logging).
* **“latitude”**: The latitude coordinate of the city.
* **“longitude”**: The longitude coordinate of the city.

In the second line (URL=), we construct a URL to query the Open-Meteo API. The URL includes the following components:

* Base URL: [**https://api.open-meteo.com/v1/forecast**](https://api.open-meteo.com/v1/forecast)
* Query Parameters: **“latitude”**, **“longitude”**, “**hourly”**

This URL requests hourly weather data for sunshine duration and wind speed at a 10-meter height.

Next, the line (response =), makes an HTTP GET request to the constructed URL using the **requests** library (**req**). This sends a request to the Open-Meteo API to fetch the weather data.

The response from the API is in JSON format. The **response.json()** method parses the JSON response into a Python dictionary for easier manipulation.

The **data["hourly"]** part extracts the hourly weather data from the JSON response. This hourly data is then converted into a panda Data Frame, **df**. This Data Frame initially contains the following columns (based on the requested data): “time”, “Sunshine\_duration”, “wind\_speed\_10m”.



This line groups the data by the **date** column to aggregate daily values. We use the .agg**()** method to specify how to aggregate each column:

* **sunshine\_duration**: Summed to get the total sunshine duration per day.
* **wind\_speed\_10m**: Averaged to get the mean wind speed per day.

The **.reset\_index()** method is used to convert the grouped DataFrame back to a regular DataFrame, ensuring the **date** column remains a standard column rather than an index.

1. Weather Forecast:

A screen shot of a computer code

Description automatically generated

The function **weather\_forecastv2** takes one parameter:

**cities**: A dictionary where the keys are city names, and the values are tuples containing the latitude and longitude coordinates of the cities.

The first loop iterates over each city in the **cities** dictionary:

1) It calls the **fetch\_weather\_data** function with the city name and its coordinates to get the weather data as a DataFrame **df**.

2) A new column **city** is added to **df** to store the name of the city.

3) The DataFrame **df** is concatenated to the **dfAll** DataFrame. The **pd.concat** function appends the data for each city to **dfAll**.

Three new columns are added to **dfAll** using the **assign** method:

**1) Ew**: Wind energy calculated using the formula E\_w = v \times a \times \text{wind\_speed\_10m}.

**2) Es**: Solar energy calculated using the formula E\_s = \frac{y \times \text{sunshine\_duration} \times q \times n}{1000}.

**3) Total\_green\_energy**: The sum of wind energy (**Ew**) and solar energy (**Es**).

A computer screen with text

Description automatically generated

These lines determine the city with the most energy generated by perform the following tasks:

1. **grouped**: Groups the DataFrame by **date**.
2. **max\_energy\_per\_date**: Finds the maximum **Total\_green\_energy** for each date.
3. **Most\_energy\_generated**: Adds a new column indicating the maximum energy generated for each date.
4. **mask**: Creates a mask to identify rows where **Total\_green\_energy** matches **Most\_energy\_generated**.
5. **max\_city\_per\_date**: Filters the rows using the mask to get the city names with the highest energy for each date.
6. **merge**: Merges the city names back into **dfAll** on the **date** column.
7. **rename**: Renames the merged column to **City\_most\_energy\_generated / label**.

A screen shot of a computer code

Description automatically generated

The sidebar provides filters for selecting:

**1) Stored Location**: The location where the data is currently stored.

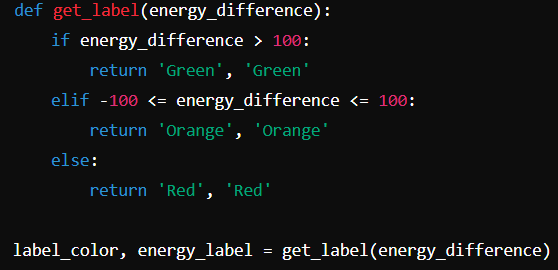
**2) Transfer Location**: The location to which you want to transfer the data.

**3) Date**: The date for which you want to compare the cities for transfer.

A screen shot of a computer code

Description automatically generated

These lines filter the DataFrame for the selected **stored\_location** and **location** on the **selected\_date**, then calculate the total green energy for both locations and the difference between them.



The **get\_label** function assigns a label based on the **energy\_difference**:

* Green: If the difference is greater than 100 Kwh.
* Orange: If the difference is between -100 and 100 Kwh.
* Red: If the difference is less than -100 Kwh.

A computer screen shot of a program code

Description automatically generated

These lines calculate the energy difference for each date and visualize it using a bar chart. The chart is color-coded based on the energy difference and includes a buffer for better visualization.

A computer screen with text on it

Description automatically generated

Finally, these lines create bar charts to visualize the total green energy, solar energy, and wind energy for each city on each date. The charts are displayed in the Streamlit dashboard.

Summary of the def Weather Forecast:

The weather\_forecastv2 function:

1. Fetches and processes weather data for multiple cities.
2. Calculates green energy metrics (wind and solar energy).
3. Identifies the city with the most energy generated each day.
4. Provides interactive filters for users to select locations and dates.
5. Displays energy differences and labels indicating the transfer benefit.
6. Visualizes wind speed, sunshine duration, and energy metrics using bar charts.

This function effectively combines data processing, calculation, and visualization to provide a comprehensive analysis of green energy potential for different cities.

**Conclusion:**

This Streamlit application provides a comprehensive tool for visualizing and comparing green energy generation across different cities and dates. It leverages multiple data visualization libraries and provides an interactive user experience through filters and dynamic updates based on user selections. The core calculations involve aggregating weather data and calculating energy outputs based on predefined formulas, with the results being visualized to aid decision-making for data transfer based on green energy availability.