Data Processing Script (`data_preprocessing.py`)

This script processes bird monitoring data from two Excel files, performs data cleaning, transformations, and stores the cleaned data into a PostgreSQL database.

Step 1: Load and Clean Data

```
def load_and_clean_data():
    forest_sheets = pd.read_excel('Bird_Monitoring_Data_FOREST.xlsx',
sheet_name=None)
    grassland_sheets =
pd.read_excel('Bird_Monitoring_Data_GRASSLAND.xlsx', sheet_name=None)
```

- **Explanation:**
  - Reads two Excel files into using `pandas`. The `sheet_name=None` option reads all sheets into a dictionary.
  - `forest_sheets` contains all sheets from the 'Bird_Monitoring_Data_FOREST.xlsx' file, and similarly for `grassland_sheets`.

```
   forest_dfs = [df for df in forest_sheets.values() if not df.empty]
    grassland_dfs = [df for df in grassland_sheets.values() if not
df.empty]
```

- **Explanation:**
  - Filters out any empty sheets from both datasets, ensuring only non-empty sheets are kept.

```
   forest_df = pd.concat(forest_dfs, ignore_index=True) if forest_dfs
else pd.DataFrame()
    grassland_df = pd.concat(grassland_dfs, ignore_index=True) if
grassland_dfs else pd.DataFrame()
```

- **Explanation:**
  - Combines the filtered sheets for both the forest and grassland data into single `DataFrame` objects using `pd.concat()`.
  - If there are no non-empty sheets, it creates an empty `DataFrame`.

```
combined_df = pd.concat([forest_df, grassland_df], ignore_index=True)
```

- **Explanation:**
  - Merges the two data sets into one `DataFrame`, `combined_df`.

```
if combined_df.empty:
    raise ValueError("No valid data found in the Excel sheets.")
```

- **Explanation:**
  - Checks if the combined dataset is empty and raises an error if so.

```
combined_df = combined_df.dropna(subset=['Scientific_Name'])
```

- **Explanation:**
  - Removes any rows where the `Scientific_Name` column has missing (NaN) values.

```
combined_df['Temperature'] = combined_df['Temperature'].fillna(combined_df['Temperature'].mean())
combined_df['Humidity'] = combined_df['Humidity'].fillna(combined_df['Humidity'].mean())
```

- **Explanation:**
  - Fills any missing values in `Temperature` and `Humidity` columns with the mean value of those columns.

```
combined_df['Date'] = pd.to_datetime(combined_df['Date'])
combined_df['Year'] = combined_df['Date'].dt.year
combined_df['Month'] = combined_df['Date'].dt.month
```

- **Explanation:**
  - Converts the `Date` column to a `datetime` format, then extracts the `Year` and `Month` as separate columns.

```
combined_df['Interval_Length'] =
pd.to_numeric(combined_df['Interval_Length'], errors='coerce')
```

- **Explanation:**
    - Converts the `Interval_Length` column to numeric values. Non-convertible values are turned into NaN.

```
def categorize_interval(length):
    if pd.isnull(length):
        return 'Unknown'
    elif 0 <= length <= 2.5:
        return '0-2.5 min'
    elif 2.5 < length <= 5:
        return '2.5-5 min'
    elif 5 < length <= 7.5:
        return '5-7.5 min'
    elif 7.5 < length <= 10:
        return '7.5-10 min'
    else:
        return '10+ min'
combined_df['Interval_Length'] =
combined_df['Interval_Length'].apply(categorize_interval)
```

- **Explanation:**
    - Defines a function to categorize `Interval_Length` into ranges, then applies this function to the `Interval_Length` column to classify each entry.

```
combined_df['Sky'] = combined_df['Sky'].fillna('Unknown')
combined_df['Wind'] = combined_df['Wind'].fillna('Unknown')
```

- **Explanation:**
    - Fills missing values in the `Sky` and `Wind` columns with `'Unknown'`.

```
columns_to_keep = [
    'Admin_Unit_Code', 'Location_Type', 'Interval_Length',
'ID_Method', 'Year', 'Month', 'Date',
```

```
        'Scientific_Name', 'Common_Name', 'Temperature', 'Humidity',
'Distance', 'Flyover_Observed',
        'Sex', 'PIF_Watchlist_Status', 'Regional_Stewardship_Status',
'Disturbance', 'Plot_Name',
        'Sky', 'Wind', 'Observer', 'Visit'
    ]
    combined_df = combined_df[columns_to_keep]
```

- **Explanation:**
  - Filters out unnecessary columns, retaining only the relevant ones specified in the list `columns_to_keep`.

Step 2: Connect to PostgreSQL Database

```
def connect_to_postgres():
    conn = psycopg2.connect(
        dbname="bird_db",
        user="postgres",
        password="Phani@1pk",
        host="localhost",
        port="5432"
    )
    return conn
```

- **Explanation:**
  - Establishes a connection to the PostgreSQL database using the `psycopg2` library with specified credentials.
  - Returns the connection object.

Step 3: Store Data in PostgreSQL

```
def store_data_in_postgres(df):
    conn = connect_to_postgres()
    cursor = conn.cursor()
```

- **Explanation:**
  - Connects to the PostgreSQL database and creates a cursor for executing queries.

```
drop_table_query = "DROP TABLE IF EXISTS bird_observations;"
 cursor.execute(drop_table_query)
 conn.commit()
```

- **Explanation:**
  - Drops the existing `bird_observations` table if it exists. This ensures that the table is recreated each time with the latest schema.

```
create_table_query = """
 CREATE TABLE bird_observations (
     Admin_Unit_Code VARCHAR(50),
     Location_Type VARCHAR(50),
     Interval_Length VARCHAR(50),
     ID_Method VARCHAR(50),
     Year INT,
     Month INT,
     Date DATE,
     Scientific_Name VARCHAR(100),
     Common_Name VARCHAR(100),
     Temperature FLOAT,
     Humidity FLOAT,
     Distance VARCHAR(50),
     Flyover_Observed BOOLEAN,
     Sex VARCHAR(50),
     PIF_Watchlist_Status BOOLEAN,
     Regional_Stewardship_Status BOOLEAN,
     Disturbance VARCHAR(100),
     Plot_Name VARCHAR(100),
     Sky VARCHAR(50),
     Wind VARCHAR(50),
     Observer VARCHAR(100),
     Visit INT
 );
 """
 cursor.execute(create_table_query)
 conn.commit()
```

- **Explanation:**
  - Creates a new `bird_observations` table in the PostgreSQL database with the specified columns.

```
    for _, row in df.iterrows():
        insert_query = sql.SQL("""INSERT INTO bird_observations (...)
VALUES (%s, %s, %s, ...);""")
        cursor.execute(insert_query, tuple(row))
    conn.commit()
```

- **Explanation:**
  - Iterates over each row in the cleaned DataFrame (`df`), inserting the data into the `bird_observations` table.
  - Commits the transaction to persist the changes.

Streamlit.py

1. Imports

```
import pandas as pd
import streamlit as st
import psycopg2
import plotly.express as px
```

Here, you're importing:

- `pandas` for data manipulation.
- `streamlit` for creating the web app and user interface.
- `psycopg2` for connecting to a PostgreSQL database.
- `plotly.express` for creating interactive plots and visualizations.

---

2. Database Connection

```
def connect_to_postgres():
    try:
        conn = psycopg2.connect(
            dbname="bird_db",
```

```
            user="postgres",  # Replace with actual username
            password="Phani@1pk",  # Replace with actual
password
            host="localhost",
            port="5432"
        )
        return conn
    except Exception as e:
        st.error(f"Database connection failed: {e}")
        return None
```

This function connects to a PostgreSQL database called `bird_db` using the provided credentials. If successful, it returns the connection object `conn`; if an error occurs, it displays an error message using Streamlit.

---

3. Query Data from PostgreSQL

```
def query_data_from_postgres(query):
    conn = connect_to_postgres()
    if conn is None:
        return pd.DataFrame()  # Return empty DataFrame if
connection fails

    try:
        df = pd.read_sql(query, conn)
        return df
    except Exception as e:
        st.error(f"Failed to execute query: {e}")
        return pd.DataFrame()
    finally:
        conn.close()
```

This function queries data from PostgreSQL:

- It first establishes a connection to the database using `connect_to_postgres()`.

- Then, it executes the SQL query and reads the result into a DataFrame using `pd.read_sql()`.
- In case of failure, it handles errors and closes the connection at the end.

---

4. Exploratory Data Analysis (EDA)

This is where all the actual data analysis and visualizations are done. Let's go over each part.

Temporal Analysis (Observations by Date)

```
if "date" in df.columns:
    df["date"] = pd.to_datetime(df["date"])  # Ensure date is in
datetime format
    date_counts = df["date"].value_counts().sort_index()
    fig = px.line(x=date_counts.index, y=date_counts.values,
labels={'x': 'Date', 'y': 'Number of Observations'})
    st.plotly_chart(fig)
    st.write("**Summary:** The temporal analysis shows the
number of bird observations over time. Peaks in the graph
indicate periods of higher bird activity, which may correlate
with migration or breeding seasons.")
```

- **Purpose**: Visualize how the number of bird observations changes over time.
- **Steps**:
  - Check if the `date` column exists.
  - Convert the `date` column into a datetime object.
  - Count the occurrences of each date (number of observations).
  - Plot the data as a line graph using Plotly Express (`px.line`).
  - Display the chart and a summary.

Spatial Analysis (Species Diversity by Location Type)

```
if "location_type" in df.columns and "scientific_name" in
df.columns:
```

```
    location_diversity =
df.groupby('location_type')['scientific_name'].nunique().reset_i
ndex()
    location_diversity.columns = ['Location Type', 'Number of
Species']
    fig_species_diversity = px.bar(location_diversity,
x='Location Type', y='Number of Species', title='Species
Richness by Location Type', color='Location Type')
    st.plotly_chart(fig_species_diversity)
    st.write("**Summary:** This chart compares species richness
across different location types (e.g., forest, grassland). It
highlights which habitats support the highest biodiversity.")
```

- **Purpose**: Analyze how species richness varies by location type (forest, grassland, etc.).
- **Steps**:
  - Group data by `location_type` and count unique species (`scientific_name`).
  - Create a bar chart showing species richness by location type.
  - Display the plot and a summary.

---

5. Species Analysis

1. Section Header:

```
st.subheader("3. Species Analysis")
```

- This adds a subheader titled "3. Species Analysis" in the Streamlit app.
- It visually separates this section from others.

---

2. Activity Patterns Analysis

```
if "interval_length" in df.columns and "id_method" in
df.columns:
```

- This checks if the columns `"interval_length"` and `"id_method"` exist in the dataset (`df`).
- If both columns are present, it proceeds with the analysis.

```
activity_patterns = df.groupby(['interval_length',
'id_method']).size().reset_index(name='Observations')
```

- Groups the data by `interval_length` (how long an observation lasted) and `id_method` (the method used to identify the species).
- Uses `.size()` to count the number of observations for each group.
- `.reset_index(name='Observations')` converts the result into a DataFrame with a column named "Observations".

```
fig_activity = px.bar(activity_patterns, x='interval_length',
y='Observations', color='id_method', title="Activity Patterns by
Interval Length and Method")
```

- Creates a bar chart using Plotly Express (`px.bar`).
- `x='interval_length'`: The x-axis represents different observation time intervals.
- `y='Observations'`: The y-axis represents the number of observations recorded.
- `color='id_method'`: Uses different colors to differentiate identification methods.
- `title`: Sets the chart title.

```
st.plotly_chart(fig_activity)
```

- Displays the bar chart in the Streamlit app.

```
st.write("**Summary:** This chart shows the most common bird
activity patterns based on observation intervals and
identification methods. It helps identify preferred observation
durations and methods.")
```

- Displays an explanatory text below the chart.

Sex Ratio Analysis

```
if "sex" in df.columns:
```

- Checks if the `"sex"` column exists in the dataset.

```
sex_ratio = df.groupby(['scientific_name',
'sex']).size().reset_index(name='Count')
```

- Groups data by `scientific_name` (species name) and `sex` (Male/Female).
- Counts the number of observations for each sex in each species.
- Converts the result into a DataFrame with a column named "Count".

```
fig_sex_ratio = px.bar(sex_ratio, x='scientific_name',
y='Count', color='sex', title="Sex Ratio for Species")
```

- Creates a bar chart using Plotly Express (`px.bar`).
- `x='scientific_name'`: The x-axis represents different bird species.
- `y='Count'`: The y-axis represents the number of male and female observations.
- `color='sex'`: Different colors for male and female.
- `title`: Sets the chart title.

```
st.plotly_chart(fig_sex_ratio)
```

- Displays the bar chart in the Streamlit app.

```
st.write("**Summary:** The sex ratio analysis reveals the
male-to-female distribution across species. Some species may
show a skewed ratio, which could indicate gender-based
behavioral differences.")
```

- Displays an explanatory text below the chart.

Final Output

- First Chart: Shows how bird activity varies based on observation intervals and identification methods.
- Second Chart: Displays the male-to-female ratio for different bird species.

6. Environmental Conditions: Weather Correlation

1. Weather Correlation Analysis

```
if "temperature" in df.columns and "humidity" in df.columns and "sky" in
df.columns and "wind" in df.columns:
```

- This checks if the dataset (`df`) contains all four weather-related columns: `"temperature"`, `"humidity"`, `"sky"`, and `"wind"`.
- If all columns exist, the analysis proceeds.

Step 1: Add a Subheader

```
st.subheader("4. Environmental Conditions: Weather Correlation")
```

- Adds a subheader in Streamlit to introduce this section.

Step 2: Group Data by Weather Conditions

```
weather_conditions = df.groupby(['temperature', 'humidity', 'sky',
'wind']).size().reset_index(name='Observations')
```

- Groups the data by temperature, humidity, sky condition, and wind.
- `.size()` counts the number of bird observations under each weather condition.
- `.reset_index(name='Observations')` converts the result into a DataFrame with a column `"Observations"`.

Step 3: Create a Scatter Plot

```
fig_weather = px.scatter(weather_conditions, x='temperature',
y='humidity', color='sky', title="Weather Correlation with Observations")
```

- Uses Plotly Express (`px.scatter`) to create a scatter plot.
- `x='temperature'`: X-axis represents temperature.
- `y='humidity'`: Y-axis represents humidity.
- `color='sky'`: Colors the points based on sky conditions (e.g., clear, cloudy, rainy).
- `title`: Sets the chart title.

Step 4: Display the Chart

```
st.plotly_chart(fig_weather)
```

- Displays the scatter plot in the Streamlit app.

Step 5: Add an Explanation

```
st.write("**Summary:** This scatter plot explores the relationship between weather conditions (temperature, humidity, sky, wind) and bird observations. Certain weather conditions may correlate with higher bird activity.")
```

- Displays a summary to explain the chart.

---

2. Impact of Disturbance on Bird Sightings

```
if "disturbance" in df.columns:
```

- Checks if the `"disturbance"` column exists in the dataset.
- If present, the analysis proceeds.

Step 1: Add a Subheader

```
st.subheader("Impact of Disturbance on Bird Sightings")
```

- Adds a subheader to introduce this section.

Step 2: Group Data by Disturbance Type

```
disturbance_effect =
df.groupby('disturbance')['scientific_name'].count().reset_index()
disturbance_effect.columns = ['Disturbance', 'Sighting_Count']
```

- Groups the dataset by disturbance type and counts the number of bird sightings (`scientific_name`).
- Renames the columns to `Disturbance` and `Sighting_Count` for readability.

Step 3: Create a Bar Chart

```
fig = px.bar(disturbance_effect,
             x='Disturbance',
             y='Sighting_Count',
             title='Impact of Disturbance on Bird Sightings',
             labels={'Disturbance': 'Disturbance Type', 'Sighting_Count':
'Number of Bird Sightings'},
             color='Sighting_Count', color_continuous_scale='Viridis')
```

- Uses Plotly Express (`px.bar`) to create a bar chart.
- `x='Disturbance'`: X-axis represents disturbance types (e.g., human activity, noise, weather events).
- `y='Sighting_Count'`: Y-axis represents the number of bird sightings under each disturbance type.
- `color='Sighting_Count'`: Colors the bars based on the number of sightings using the Viridis color scale.

Step 4: Adjust Chart Layout

```
fig.update_layout(xaxis_title='Disturbance Type', yaxis_title='Number of
Bird Sightings')
fig.update_xaxes(tickangle=45)  # Rotate x-axis labels for better
readability
```

- Sets the axis labels.
- Rotates the x-axis labels for better readability.

Step 5: Display the Chart

```
st.plotly_chart(fig)
```

- Displays the bar chart in Streamlit.

Step 6: Add an Explanation

```
st.write("**Summary:** This chart shows how different types of
disturbances (e.g., human activity, weather events) impact bird sightings.
Some disturbances may reduce bird activity, while others may have no
significant effect.")
```

- Displays a summary to explain the chart.

Final Output

1. Scatter Plot:
   - Shows the correlation between temperature, humidity, sky conditions, and bird sightings.
   - Helps understand if birds are more active under specific weather conditions.
2. Bar Chart:
   - Shows the impact of disturbances on bird sightings.
   - Helps identify which disturbances (e.g., noise, human activity) reduce or increase bird sightings.

7.Distance and Behavior

1. Distance Analysis

```
st.subheader("5. Distance and Behavior")
```

- Adds a subheader to introduce the analysis section related to distance and behavior.

```
if "distance" in df.columns:
```

- Checks if the column "distance" exists in the dataset (df).
- If it exists, the code proceeds with distance analysis.

Step 1: Add a Subheader for Distance Analysis

```
st.subheader("Distance Analysis")
```

- Adds a subheader specifically for distance-related insights.

Step 2: Count Observations for Each Distance

```
distance_counts = df["distance"].value_counts().reset_index()

distance_counts.columns = ["Distance", "Count"]
```

- df["distance"].value_counts() counts how many times each distance value appears in the dataset.
- .reset_index() converts this count into a DataFrame.
- Columns are renamed to "Distance" and "Count" for clarity.

Step 3: Create a Bar Chart

```
fig_distance = px.bar(
    distance_counts,
    x="Distance",
    y="Count",
    title="Distribution of Observation Distances",
    labels={"Count": "Number of Observations"},
    color="Distance"
)
```

- Uses Plotly Express (px.bar) to create a bar chart.
- x="Distance": The X-axis represents different distances at which birds were observed.
- y="Count": The Y-axis represents how many times birds were observed at each distance.
- color="Distance": Colors bars differently based on distance for better visualization.

Step 4: Display the Chart

```
st.plotly_chart(fig_distance)
```

- Displays the bar chart in the Streamlit app.

Step 5: Add an Explanation

```
st.write("**Summary:** This bar chart shows the distribution of observation distances. It helps identify whether birds are typically observed closer or farther from the observer.")
```

- Explains the insight from the chart.

---

2. Flyover Frequency Analysis

```
if "flyover_observed" in df.columns:
```

- Checks if the column "flyover_observed" exists in the dataset.
- If present, the code proceeds with flyover frequency analysis.

Step 1: Add a Subheader for Flyover Analysis

```
st.subheader("Flyover Frequency Analysis")
```

- Adds a subheader to introduce the flyover frequency analysis.

Step 2: Count Flyover Observations

```
flyover_counts = df["flyover_observed"].value_counts().reset_index()

flyover_counts.columns = ["Flyover Observed", "Count"]
```

- df["flyover_observed"].value_counts() counts how many times flyovers were observed.
- .reset_index() converts this into a DataFrame.
- Renames columns to "Flyover Observed" and "Count" for better readability.

Step 3: Create a Bar Chart

```
fig_flyover = px.bar(

    flyover_counts,

    x="Flyover Observed",

    y="Count",

    title="Flyover Frequency",

    labels={"Count": "Number of Observations"},

    color="Flyover Observed"

)
```

- Uses Plotly Express (px.bar) to create a bar chart.
- x="Flyover Observed": X-axis represents whether a flyover was observed (Yes/No).
- y="Count": Y-axis represents the number of observations for each category.
- color="Flyover Observed": Colors bars based on flyover observation status.

Step 4: Display the Chart

```
st.plotly_chart(fig_flyover)
```

- Displays the flyover frequency bar chart.

Step 5: Add an Explanation

```
st.write("**Summary:** This chart shows how often flyovers (birds flying overhead) are observed. Frequent flyovers may indicate migration patterns or preferred flight paths.")
```

- Explains the insight from the chart.

Final Output

1. Distance Analysis (Bar Chart)
    - Shows the distribution of bird observations at different distances.
    - Helps identify if birds are typically observed closer or farther from the observer.
2. Flyover Frequency Analysis (Bar Chart)
    - Shows how frequently birds are observed flying overhead.
    - Helps detect migration patterns or common flight paths.

8.Observer Trends

1. Observer Trends Analysis

```
st.subheader("6. Observer Trends")
```

- Adds a **subheader** to introduce the section on observer trends.

Step 1: Check if "observer" Column Exists

```
if "observer" in df.columns:
```

- **Checks** if the dataset (df) contains the "observer" column.
- If it does, the analysis continues.

Step 2: Count Unique Species Recorded by Each Observer

```
observer_counts = df.groupby('observer')['scientific_name'].nunique().reset_index()
```

- Groups data by "observer", counting the number of **unique species (scientific_name)** recorded by each observer.
- .nunique() ensures that only **distinct species** observed by each observer are counted.
- .reset_index() converts the grouped data back into a **DataFrame**.

Step 3: Rename Columns for Clarity

```
observer_counts.columns = ['Observer', 'Unique Species Count']
```

- Renames the columns for better readability.

Step 4: Create a Bar Chart

```
fig_observer_bias = px.bar(
    observer_counts,
    x='Observer',
    y='Unique Species Count',
    title='Observer Trends and Bias',
    color='Observer'
)
```

- Uses **Plotly Express (px.bar)** to create a **bar chart**.
- x='Observer': X-axis represents different observers.
- y='Unique Species Count': Y-axis represents the number of unique species observed.
- color='Observer': Colors each observer differently for better visualization.

Step 5: Display the Chart

```
st.plotly_chart(fig_observer_bias)
```

- Displays the observer trends chart in the Streamlit app.

Step 6: Add an Explanation

```
st.write("**Summary:** This chart highlights observer trends, showing how many unique species each observer has recorded. It helps identify potential observer bias or expertise.")
```

- **Explains the insight** from the chart.
- Observers who record significantly more species might be more experienced or biased towards recording certain types.

---

2. Visit Patterns Analysis

Step 1: Check if "visit" Column Exists

```
if "visit" in df.columns:
```

- **Checks** if the "visit" column exists in the dataset.
- If present, the code proceeds with the visit pattern analysis.

Step 2: Count Unique Species Observed Per Visit

```
visit_counts = df.groupby('visit')['scientific_name'].nunique().reset_index()
```

- Groups data by "visit" and counts the **number of unique species** observed per visit.
- .nunique() ensures that only **distinct species** are counted.
- .reset_index() converts the grouped data back into a **DataFrame**.

Step 3: Rename Columns for Clarity

```
visit_counts.columns = ['Visit', 'Number of Unique Species']
```

- Renames the columns for better readability.

Step 4: Create a Line Chart

```
fig_visit_patterns = px.line(
    visit_counts,
    x='Visit',
    y='Number of Unique Species',
    title='Visit Patterns and Species Diversity'
)
```

- Uses **Plotly Express (px.line)** to create a **line chart**.
- x='Visit': X-axis represents different visit instances.
- y='Number of Unique Species': Y-axis represents how many unique species were observed.
- A **line chart** is used because it shows trends over time.

Step 5: Display the Chart

st.plotly_chart(fig_visit_patterns)

- Displays the visit pattern chart.

Step 6: Add an Explanation

st.write("**Summary:** This line chart shows how species diversity changes with repeated visits to the same location. Increased diversity over time may indicate effective monitoring or seasonal changes.")

- **Explains the insight** from the chart.
- If species diversity **increases** over time, it might indicate:
  - Seasonal changes bringing in different species.
  - Effective monitoring and conservation efforts.

Final Output

1. **Observer Trends (Bar Chart)**
   - Shows how many **unique species** each observer recorded.
   - Helps detect observer **bias** or expertise.
2. **Visit Patterns (Line Chart)**
   - Tracks **species diversity** across repeated visits.
   - Helps analyze **seasonal changes** or monitoring effectiveness.

9.Conservation Insights

1. Section Header

```
st.subheader("7. Conservation Insights")
```

- Adds a **subheader** to introduce the section.

---

2. Check if Conservation-Related Columns Exist

```
if "pif_watchlist_status" in df.columns and
"regional_stewardship_status" in df.columns:
```

- **Ensures** that both `"pif_watchlist_status"` and `"regional_stewardship_status"` columns exist in the dataset.
- If **both** columns are present, the analysis continues.

---

3. PIF Watchlist Status Analysis

Step 1: Count Unique Species for Each Watchlist Status

```
watchlist_status_counts =
df.groupby('pif_watchlist_status')['scientific_name'].nunique().res
et_index()
```

- Groups the data by `"pif_watchlist_status"`, counting the **number of unique species** in each status category.
- `.nunique()` ensures that only **distinct species** are counted.
- `.reset_index()` converts the grouped data into a **DataFrame**.

Step 2: Rename Columns for Readability

```
watchlist_status_counts.columns = ['Watchlist Status', 'Species
Count']
```

- Renames the columns for better understanding.

Step 3: Create a Bar Chart

```
fig_watchlist = px.bar(

    watchlist_status_counts,

    x='Watchlist Status',

    y='Species Count',

    title='Species Count by PIF Watchlist Status',

    color='Watchlist Status'

)
```

- Uses **Plotly Express (`px.bar`)** to create a **bar chart**.
- `x='Watchlist Status'`: X-axis represents different watchlist statuses.

- `y='Species Count'`: Y-axis represents the number of species in each status category.
- `color='Watchlist Status'`: Each category is **color-coded**.

Step 4: Display the Chart

```
st.plotly_chart(fig_watchlist)
```

- Displays the **PIF Watchlist** status chart.

Step 5: Add an Explanation

```
st.write("**Summary:** This chart shows the number of species on the PIF Watchlist, highlighting those at risk and requiring conservation focus.")
```

- **Explains the insight** from the chart.
- The **PIF (Partners in Flight) Watchlist** identifies species at risk.
- Helps conservationists **prioritize species** needing protection.

---

4. Regional Stewardship Status Analysis

Step 1: Count Unique Species for Each Stewardship Status

```
stewardship_status_counts = df.groupby('regional_stewardship_status')['scientific_name'].nunique().reset_index()
```

- Groups the data by `"regional_stewardship_status"`, counting the **number of unique species** in each stewardship category.
- `.nunique()` ensures that only **distinct species** are counted.
- `.reset_index()` converts the grouped data into a **DataFrame**.

Step 2: Rename Columns for Readability

```
stewardship_status_counts.columns = ['Stewardship Status', 'Species Count']
```

- Renames the columns for clarity.

Step 3: Create a Bar Chart

```
fig_stewardship = px.bar(
```

```
    stewardship_status_counts,
    x='Stewardship Status',
    y='Species Count',
    title='Species Count by Regional Stewardship Status',
    color='Stewardship Status'
)
```

- Uses **Plotly Express (`px.bar`)** to create a **bar chart**.
- `x='Stewardship Status'`: X-axis represents different stewardship statuses.
- `y='Species Count'`: Y-axis represents the number of species under each category.
- `color='Stewardship Status'`: Each category is **color-coded**.

Step 4: Display the Chart

```
st.plotly_chart(fig_stewardship)
```

- Displays the **Regional Stewardship Status** chart.

Step 5: Add an Explanation

```
st.write("**Summary:** This chart highlights species under regional stewardship, indicating areas where conservation efforts are most needed.")
```

- **Explains the insight** from the chart.
- Helps conservationists understand which species need **localized conservation**.

Final Output

1. **PIF Watchlist Status (Bar Chart)**
   - Shows how many species are on the **watchlist**.
   - Highlights species at **risk** and needing **conservation focus**.
2. **Regional Stewardship Status (Bar Chart)**
   - Shows species under **regional conservation programs**.
   - Helps focus **efforts where they are most needed**.

10.Distance vs Species HeatMap

1. Checking if Required Columns Exist

```
if "distance" in df.columns and "scientific_name" in df.columns:
```

- **Ensures** that both `"distance"` and `"scientific_name"` columns exist in the dataset.
- If these columns are present, the analysis continues.

---

2. Grouping Data by Distance and Species

```
distance_impact = df.groupby(["distance",
"scientific_name"]).size().reset_index(name="count")
```

- Groups the dataset based on **distance** and **species name** (`scientific_name`).
- `.size()` **counts** the number of observations for each species at a given distance.
- `.reset_index(name="count")` converts the grouped data into a new **DataFrame** with a column `"count"` representing the number of observations.

---

3. Adding a Section Header

```
st.subheader("8. Distance vs. Species Heatmap")
```

- Displays a **subheader** in the Streamlit app to introduce the heatmap.

---

4. Creating the Heatmap

```
fig_heatmap = px.density_heatmap(

    distance_impact,

    x="distance",

    y="scientific_name",

    z="count",

    title="Heatmap of Distance vs. Species Observations",
```

```
    labels={"count": "Observation Density", "distance": "Distance",
"scientific_name": "Species"},

    color_continuous_scale="Viridis"

)
```

- Uses **Plotly Express (`px.density_heatmap`)** to create a heatmap.
- **X-axis (`x="distance"`)** → Represents the observation distance.
- **Y-axis (`y="scientific_name"`)** → Represents the species.
- **Z-axis (`z="count"`)** → Represents the **density** of observations (how frequently a species is observed at a specific distance).
- `color_continuous_scale="Viridis"` → Uses the **Viridis color scale** for better visibility.

---

5. Displaying the Heatmap

```
st.plotly_chart(fig_heatmap)
```

- Displays the heatmap in **Streamlit**.

---

6. Adding an Explanation

```
st.write("**Summary:** This heatmap shows the relationship between
observation distance and species. It helps identify species that
are typically observed at specific distances.")
```

- **Explains the insight** from the heatmap.
- **Why is this important?**
  - Some species might be **more visible at short distances**, while others are **typically observed from far away**.
  - Helps researchers understand **bird behavior and habitat preferences**.

Final Output

- A **heatmap** showing the relationship between **observation distance** and **species**.
- Helps in understanding:
  1. Which species are **observed more frequently** at certain distances.
  2. Whether some species prefer **close vs. distant observations**.

11.Number of Bird Species Observed at Different Temperatures

1. Checking if Required Columns Exist

```
if "temperature" in df.columns and "scientific_name" in df.columns:
```

- Ensures that both `"temperature"` and `"scientific_name"` columns exist in the dataset.
- If these columns are present, the analysis proceeds.

---

2. Converting Temperature to Numeric Format

```
df['temperature'] = pd.to_numeric(df['temperature'], errors='coerce')
```

- Converts the `"temperature"` column to a **numeric type**.
- `errors='coerce'` ensures that **non-numeric values** are converted to `NaN` (missing values), preventing errors in analysis.

---

3. Removing Missing Values

```
df_cleaned = df.dropna(subset=['temperature', 'scientific_name'])
```

- Drops rows where `"temperature"` or `"scientific_name"` is missing (`NaN`).
- Ensures that only **valid** data is used in the analysis.

---

4. Grouping Data by Temperature

```
temp_bird_counts =
df_cleaned.groupby('temperature')['scientific_name'].nunique().reset_i
ndex()
```

- Groups the dataset by **temperature**.
- `.nunique()` counts the **unique number of species** observed at each temperature.
- `.reset_index()` converts the grouped data into a **DataFrame**.

---

5. Renaming Columns for Clarity

```
temp_bird_counts.columns = ['Temperature', 'Number of Species']
```

- Renames the columns to make them **more readable**.

---

6. Creating a Bar Chart

```
fig = px.bar(
    temp_bird_counts,
    x='Temperature',
    y='Number of Species',
    title='9. Number of Bird Species Observed at Different
Temperatures',
    labels={'Temperature': 'Temperature (°C)', 'Number of Species':
'Unique Species Count'}
)
```

- Uses **Plotly Express (`px.bar`)** to create a **bar chart**.
- **X-axis (`x='Temperature'`)** → Represents temperature in degrees Celsius (°C).
- **Y-axis (`y='Number of Species'`)** → Represents the **number of unique bird species** observed at each temperature.
- `title` → Sets a **descriptive title**.
- `labels` → Adds **clear axis labels**.

---

7. Displaying the Chart in Streamlit

```
st.plotly_chart(fig)
```

- Displays the bar chart in the **Streamlit** app.

---

8. Adding an Explanation

```
st.write("**Summary:** This chart shows how bird species diversity
varies with temperature. Certain temperature ranges may support higher
biodiversity.")
```

- **Explains the insight** from the chart.
- **Why is this important?**
    - Helps **identify temperature ranges** where bird species diversity is highest.

- Certain birds might be **more active in specific temperatures** due to climate preferences.

Final Output

- A **bar chart** showing the number of **unique bird species** observed at different temperatures.
- Helps in understanding:
  1. **Which temperature ranges** support the highest bird diversity.
  2. How **weather conditions** influence bird activity.

12. Creating a Dashboard

```
def create_dashboard(df):
    st.title("Bird Species Observation Analysis")
    st.write("This dashboard provides insights into bird species
distribution and diversity across forests and grasslands.")
```

- **Purpose**: Set up the title and description for the Streamlit app.
- **Steps**:
  - Display the app title using `st.title()`.
  - Show a brief description using `st.write()`.

---

13. Filters for Sidebar

```
st.sidebar.header("Filters")
```

This creates a sidebar header that will hold the filters.

Location Type Filter

```
if "location_type" in df.columns:
    location_type = st.sidebar.selectbox("Select Location Type",
["All", "Forest", "Grassland"])
else:
    st.sidebar.write("Location Type data not available.")
    location_type = "All"
```

- If location_type exists in the DataFrame, a select box will allow the user to choose between "All", "Forest", or "Grassland".
- If it doesn't exist, the user is informed, and the filter is set to "All".

Admin Unit Code Filter

```
if "admin_unit_code" in df.columns:
    admin_units = df["admin_unit_code"].unique()
    selected_admin_unit = st.sidebar.selectbox("Select Admin
Unit Code", ["All"] + list(admin_units))
else:
    selected_admin_unit = "All"
```

- If admin_unit_code exists, a select box allows the user to filter by different admin units. "All" is the default.
- If it doesn't exist, the filter is set to "All".

Date Range Filter

```
if "date" in df.columns:
    df["date"] = pd.to_datetime(df["date"])
    min_date, max_date = df["date"].min(), df["date"].max()
    date_range = st.sidebar.date_input("Select Date Range",
[min_date, max_date], min_value=min_date, max_value=max_date)
else:
    st.sidebar.write("Date data not available.")
    date_range = None
```

- If date exists, a date input allows the user to filter the data based on a selected date range.

---

14. Applying Filters to the Data

```
filtered_df = df
if selected_admin_unit != "All":
    filtered_df = filtered_df[filtered_df['admin_unit_code'] ==
selected_admin_unit]
```

```
if location_type and location_type != "All":
    filtered_df = filtered_df[filtered_df['location_type'] ==
location_type]
if date_range and len(date_range) == 2:
    start_date, end_date = pd.to_datetime(date_range[0]),
pd.to_datetime(date_range[1])
    filtered_df = filtered_df[(filtered_df['date'] >=
start_date) & (filtered_df['date'] <= end_date)]
```

- Filters are applied to the DataFrame based on the selected options in the sidebar.

---

15. Displaying Filtered Data

```
st.subheader("Filtered Data")
st.write(filtered_df)
```

- Displays the filtered dataset after the sidebar filters are applied.

---

16. Perform EDA on Filtered Data

```
perform_eda(filtered_df)
```

- This function performs the EDA on the filtered data and generates visualizations.

---

17. Main Function

```
if __name__ == "__main__":
    st.sidebar.title("Data Source")

    # Query data from PostgreSQL
    query = "SELECT * FROM bird_observations;"
```

```
    df_from_postgres = query_data_from_postgres(query)

    # Create Streamlit dashboard
    create_dashboard(df_from_postgres)
```

- This section is the entry point of the application.
- It queries the data from PostgreSQL and creates the Streamlit dashboard with the retrieved data.

Summary

This code sets up a Streamlit web app to display bird observation data. It connects to a PostgreSQL database, fetches the data, and performs various analyses like temporal, spatial, and species analysis. Users can filter data by location, date, and other criteria using a sidebar. The app provides an interactive dashboard with visualizations for insights into bird species distribution and biodiversity.