Data Cleaning / Grouping - Task 6

By Menna Jaheen

Data Cleaning in Pandas :

1. ### Remove Duplicates

   - **Purpose:** To ensure that each row in the dataset is unique.
   - **Example:**

     ```
     df.drop_duplicates(inplace=True)
     ```

   - **Explanation:** The `drop_duplicates()` function removes any duplicate rows in the DataFrame, ensuring that each entry is unique. The `inplace=True` argument modifies the DataFrame in place.

---

2. ### Drop Unnecessary Columns

   - **Purpose:** To remove columns that are not needed for analysis.
   - **Example:**

     ```
     df.drop(columns=['Unnecessary_Column'], inplace=True)
     ```

   - **Explanation:** The `drop()` function with the `columns` argument allows you to remove specified columns. This helps to streamline the dataset and focus on relevant data.

---

3. ### Handle Missing Values

   - **Purpose:** To deal with missing data points that could affect analysis.

- **Example:**

```
df.dropna(inplace=True)  # Remove rows with missing val
ues
df.fillna(value={'Column_Name': 'Default_Value'}, inpla
ce=True)  # Fill missing values
```

- **Explanation:** Missing values can be addressed by either removing rows with missing data (`dropna()`) or filling them with a specific value (`fillna()`).

---

4. ## Correct Syntax Errors in Data

- **Purpose:** To standardize the format of data entries.
- **Example:**

```
df['Name'] = df['Name'].str.replace(r'[^\w\s]', '', reg
ex=True).str.capitalize()
df['Phone'] = df['Phone'].str.replace(r'[^\d]', '', reg
ex=True)
```

- **Explanation:** The `str.replace()` method can be used to remove unwanted characters or standardize formats. The `str.capitalize()` method standardizes text data by capitalizing the first letter.

---

5. ## Normalize Data Formats

- **Purpose:** To ensure consistent formatting across the dataset.
- **Example:**

```
df['Date'] = pd.to_datetime(df['Date'], format='%Y-%m-%
d')
df['Name'] = df['Name'].str.capitalize()
```

- **Explanation:** Normalizing data formats ensures that data is consistent and can be easily compared or processed. The `pd.to_datetime()` function converts date strings to datetime objects, ensuring a consistent date format.

---

6. ### Set a Unique Identifier
   - **Purpose:** To ensure that each entry has a unique identifier.
   - **Example:**

     ```
     df['Unique_ID'] = pd.factorize(df['Identifier'])[0] + 1
     ```

   - **Explanation:** If a unique identifier is needed, the `factorize()` function can be used to create a new column with unique values.

---

7. ### Correct Data Types
   - **Purpose:** To ensure that each column has the correct data type for analysis.
   - **Example:**

     ```
     df['Age'] = df['Age'].astype(int)
     df['Price'] = df['Price'].astype(float)
     ```

   - **Explanation:** The `astype()` function is used to convert columns to the appropriate data type, which is crucial for accurate calculations and analysis.

Data Grouping in Pandas:

- ### Grouping Data

- **Purpose:** To split the data into groups based on the values in one or more columns.
- **Example:**

```
grouped = df.groupby('column_name')
```

- **Explanation:** The `groupby()` function is used to create a GroupBy object, which can be further used to apply aggregation functions to each group. You can group by a single column or multiple columns.

- ## Aggregating Data
  - **Purpose:** To compute summary statistics for each group.
  - **Example:**

```
grouped = df.groupby('column_name').mean()
grouped = df.groupby(['column1', 'column2']).sum()
```

  - **Explanation:** After grouping the data, you can apply aggregation functions like `mean()`, `sum()`, `count()`, `min()`, `max()`, etc., to calculate summary statistics for each group. These functions return a DataFrame or Series depending on the operation.

- ## Applying Multiple Aggregations
  - **Purpose:** To compute different aggregations for different columns within each group.
  - **Example:**

```
grouped = df.groupby('column_name').agg({'column1': 'me
an', 'column2': 'sum'})
```

  - **Explanation:** The `agg()` function allows you to apply multiple aggregation functions to different columns within each group. You can specify a

dictionary where the keys are column names and the values are the aggregation functions.

- ### Filtering Groups
  - **Purpose:** To include only groups that meet a certain condition.
  - **Example:**

    ```
    filtered = df.groupby('column_name').filter(lambda x: x['column2'].mean() > 50)
    ```

  - **Explanation:** The `filter()` function allows you to exclude groups that don't meet a specified condition. This is useful when you only want to analyze a subset of groups that satisfy certain criteria.

- ### Transforming Groups
  - **Purpose:** To apply a function to each group and return a transformed version of the data.
  - **Example:**

    ```
    transformed = df.groupby('column_name').transform(lambda x: x - x.mean())
    ```

  - **Explanation:** The `transform()` function applies a function to each group and returns a DataFrame or Series with the same shape as the original data. This is useful for standardizing or normalizing data within each group.

- ### Iterating Over Groups
  - **Purpose:** To loop through each group and perform custom operations.
  - **Example:**

```
for name, group in df.groupby('column_name'):
    print(name)
    print(group)
```

- **Explanation:** Iterating over a GroupBy object allows you to access each group and perform custom operations. `name` represents the group name, and `group` is the DataFrame corresponding to that group.

- **Grouping with Hierarchical Indexing**
  - **Purpose:** To group data and return a DataFrame with a MultiIndex.
  - **Example:**

```
grouped = df.groupby(['column1', 'column2']).mean()
```

  - **Explanation:** When grouping by multiple columns, the result is a DataFrame with a MultiIndex. This allows for more complex data structures and operations on hierarchical data.