PROJECT 2

DATA AQUISITION AND DATA WRANGLING

- Merging datasets is a fundamental task in data analysis, essential for creating comprehensive and insightful views of information. This process involves combining data from multiple sources into a single, unified dataset, allowing for a more holistic analysis and better-informed decision-making.
- Python, with its rich ecosystem of data manipulation libraries, is an ideal choice for this task. Its flexibility, scalability, and automation capabilities make it a powerful tool for handling large and complex datasets. The pandas library, in particular, stands out as the primary tool for data manipulation in Python, offering efficient and intuitive methods for reading, merging, cleaning, and exporting data.

A
by Ankita Taneja

Setting Up Your Python Environment and Importing Libraries

To begin merging Excel datasets with Python, the first step is to set up your Python environment. If you don't already have Python installed, Anaconda is a recommended distribution that includes Python, essential packages, and a package manager. Alternatively, you can install Python directly from the official Python website.

Once Python is installed, you need to install the pandas library, which provides the necessary tools for data manipulation. You can install pandas using pip, the Python package installer. Open your command prompt or terminal and run the following command:

```
pip install pandas
```

After installing pandas, it's good practice to verify your installation by writing a short Python script to import the library. This also allows you to confirm which version of pandas you have installed. Create a python script like `check_pandas.py` and write the following lines:

```
import pandas as pd
print(pd.__version__)
```

Execute the script in the terminal with the command `python check_pandas.py`. This will print the version of pandas installed, for example 2.2.1. Ensure you have version 1.0+.

To import the pandas library into your Python script or notebook, use the following statement:

```
import pandas as pd
```

The `as pd` is a common convention that allows you to refer to pandas functions and objects using the `pd` prefix, making your code more concise and readable.

It's important to have a compatible version of Python installed. Pandas generally supports Python 3.7 and later. For working with `.xlsx` files, the `openpyxl` library is often required. You can install it using pip as well:

```
pip install openpyxl
```

Reading Data from Excel Files

Reading data from Excel files is a crucial step in the process of merging datasets. The pandas library provides the `read_excel()` function, which simplifies this task. To read data from an Excel file, use the following syntax:

```
import pandas as pd

df = pd.read_excel("data.xlsx")
```

Here, "data.xlsx" is the path to your Excel file. You can specify the file path in different ways:

- **Absolute Path:** The full path to the file, such as "C:/Users/Username/Documents/data.xlsx."
- **Relative Path:** The path relative to your current working directory, such as "data.xlsx" if the file is in the same directory as your Python script, or "data/data.xlsx" if the file is in a subdirectory named "data."

Excel files can contain multiple sheets. To read data from a specific sheet, use the `sheet_name` parameter:

```
df = pd.read_excel("data.xlsx", sheet_name="Sheet1")
```

In this example, the data from the sheet named "Sheet1" will be read into the DataFrame `df`. You can also pass the index of the sheet, if you prefer.

Excel files often have header rows that contain column names and index columns that serve as row labels. You can specify these using the `header` and `index_col` parameters:

```
df = pd.read_excel("data.xlsx", header=0, index_col=0)
```

Here, `header=0` indicates that the first row (index 0) contains the column names, and `index_col=0` indicates that the first column should be used as the index.

Excel files may contain missing values represented by different strings, such as "N/A," "NA," or empty strings. You can handle these by specifying the `na_values` parameter:

```
df = pd.read_excel("data.xlsx", na_values=['N/A', 'NA', ''])
```

This will treat any occurrences of "N/A," "NA," or empty strings as missing values (NaN) in the DataFrame. Similarly you can specify data types with the `dtype` parameter

To handle potential errors, such as non-existent files or incorrect file paths, use `try-except` blocks:

```
try:
    df = pd.read_excel("data.xlsx")
except FileNotFoundError:
    print("Error: The file 'data.xlsx' was not found.")
except Exception as e:
    print(f"An error occurred: {e}")
```

Merging Datasets Using Pandas

Pandas provides the `merge()` function for combining DataFrames based on common columns. This function supports various types of merges, each suited for different scenarios.

The different types of merges are:

```
merged_df = pd.merge(df1, df2, on='CustomerID', how='inner')
```

```
merged_df = pd.merge(df1, df2, on='CustomerID', how='outer')
```

```
merged_df = pd.merge(df1, df2, on='CustomerID', how='left')
```

```
merged_df = pd.merge(df1, df2, on='CustomerID', how='right')
```

- **Inner Join:** Includes only rows with matching keys in both DataFrames. Rows with keys that exist in only one DataFrame are excluded. For example, to perform an inner join on the 'CustomerID' column:
- **Outer Join:** Includes all rows from both DataFrames. If there are no matching keys, missing values (NaN) are filled in. For example, to perform an outer join on the 'CustomerID' column:
- **Left Join:** Includes all rows from the left DataFrame and matching rows from the right DataFrame. If there are no matching keys in the right DataFrame, missing values (NaN) are filled in for the columns from the right DataFrame. For example, to perform a left join on the 'CustomerID' column:
- **Right Join:** Includes all rows from the right DataFrame and matching rows from the left DataFrame. If there are no matching keys in the left DataFrame, missing values (NaN) are filled in for the columns from the left DataFrame. For example, to perform a right join on the 'CustomerID' column:

To specify the merge key, use the `on` parameter. If the keys have different names in the two DataFrames, use `left_on` and `right_on`:

```
merged_df = pd.merge(df1, df2, left_on='CustID', right_on='CustomerID', how='inner')
```

In this example, the DataFrames `df1` and `df2` are merged based on the 'CustID' column in `df1` and the 'CustomerID' column in `df2`.

When merging DataFrames, duplicate column names can arise. To handle this, use the `suffixes` parameter to add suffixes to the duplicate column names:

```
merged_df = pd.merge(df1, df2, on='CustomerID', how='inner', suffixes=('_left', '_right'))
```

Data Cleaning and Transformation After Merging

After merging datasets, it's essential to clean and transform the data to ensure its quality and usability. Common data cleaning tasks include handling missing values, removing duplicate rows, converting data types, and renaming columns.

```
merged_df.fillna(0)  # Fill missing values with 0
merged_df.dropna()  # Remove rows with any missing values
```

```
merged_df.drop_duplicates()
```

```
merged_df['Date'] = merged_df['Date'].astype('datetime64[ns]')
```

```
merged_df.rename(columns={'old_name': 'new_name'})
```

- **Handling Missing Values:** Use the `fillna()` function to fill missing values with a specific value or the `dropna()` function to remove rows with missing values:
- **Removing Duplicate Rows:** Use the `drop_duplicates()` function to remove duplicate rows from the DataFrame:
- **Converting Data Types:** Use the `astype()` function to convert the data type of a column:
- **Renaming Columns:** Use the `rename()` function to rename columns for clarity:

In addition to cleaning, data transformation involves creating new columns based on existing ones or applying functions to columns.

```
merged_df['Total'] = merged_df['Quantity'] * merged_df['Price']
```

```
merged_df['Category'] = merged_df['Category'].apply(lambda x: x.upper())
```

- **Creating New Columns:** Create a new column by performing calculations on existing columns:
- **Applying Functions to Columns:** Use the `apply()` function to apply a function to each element in a column:

This example converts all category names to uppercase.

Exporting the Merged Dataset to Excel

After merging and cleaning the datasets, the final step is to export the resulting DataFrame back to an Excel file. The pandas library provides the `to_excel()` function for this purpose.

To export the merged DataFrame to an Excel file, use the following syntax:

```
merged_df.to_excel("merged_data.xlsx", sheet_name="MergedSheet", index=False)
```

Here, "merged_data.xlsx" is the name of the output file, `sheet_name` specifies the name of the sheet in the Excel file, and `index=False` prevents the DataFrame index from being written to the Excel file.

You can control whether to include the index in the output file using the `index` parameter. By default, `index=True`, which includes the index in the output. To exclude the index, set `index=False`.

When working with large DataFrames, consider performance implications. Writing large amounts of data to Excel can be time-consuming. For very large datasets, consider alternative formats like CSV or database storage for better performance.

To ensure data integrity throughout the entire process (reading, merging, cleaning, and exporting), follow these best practices:

- Validate data types and ranges to catch errors early.
- Document all data cleaning and transformation steps.
- Test the merging process with sample datasets to ensure correctness.
- Verify the exported data against the original data sources.