AA - DB2 to Excel to OneDrive Data Gathering Template

Aim:

Page used to detail how MIS (IBM DB2) data should be gathered & joined to create assets for analysis for projects if direct access to DB2 cannot be established using a python library.

A Jupyter Notebook with the code contained in this wiki can be found here: Advisor_Analytics_Data_Gathering_Template.pynb

Contents:

1. Where to source data from.

1. Data for projects can be sourced from the MIS data lake through the IBM DB2 mainframe.
2. Microsoft Excel must be used to access data in DB2.
3. Data must be source from the 'DB2P' database alias,
    a. Credentials (username & password) must be provided by SGT or
    b. Shared credentials (username & password) must be used.
    c. This **video tutorial** can be used to learn how to access DB2P database within Excel.
4. This tutorial **video tutorial** explains how to export data from Microsoft Query into a Excel workbook.

2. Where to store files.

1. This project's shared work destination is a OneDrive folder owned by the Luyanda Dhlamini (E1005314).
2. This folder is being shared with edit access with members of the development team for the project.
3. Within the project's main folder **Advisor_Analytics**, there is a sub-folder called **Data**.
4. All files used within this project must be stored within the **Data** folder.

3. How to refer to files in Jupyter Notebooks.

1. The code block below demonstrates how to access the Data folder within the a OneDrive destination.
2. It assumes that this Notebook is being run within a **Code** sub-folder in the Project's main folder.
3. There is are 2 sample Excel Workbooks in the **Data (folder was created for Advisor Analysis project)** sub-folder that will be used for demonstration purposes.
    a. The workbooks are called sample1.xlsx & sample2.xlsx

Reading an example file

```python
import pandas as pd # import the pandas module

from datetime import datetime # import datetime sub-module

# Data files directory
data_files_path = "../Data/"

# Read the excel file(s)

# The files to be read are stored in a list object, this allows for multiple workbooks to be read at once
files_to_read = ["sample1.xlsx","sample2.xlsx","sample1.xlsx"]

# The sheet names to read from each workbook are also placed in a list.
# These must be placed in the order in which the files above must be read in.

sheets_to_read = ["Table1","Table3","Table2"]

# The list below specifies the columns to be read from each of the workbooks.
# This allows for consistency & flexibility when reading from multiple workbooks.

target_column_names_list = [
    ["A","B","C","D"],
    ["A","B","C","D"],
    ["A","B","C","D"],
]
```

4. Creating a python function for joining data.

```
def load_excel_files(data_files_path, files_to_read, sheets_to_read, target_column_names_list=None,
reset_indexes=False):
    """
    Load multiple Excel files and sheets into a single DataFrame.

    This function reads specified sheets from multiple Excel files, optionally selects specific columns,
    and concatenates them into a single DataFrame. If the target columns are not specified, all columns are
read.
    It also provides an option to reset the index of the resulting DataFrame.

    Parameters:
    - data_files_path (str): The directory path where the Excel files are located.
    - files_to_read (list of str): A list of Excel file names to be read.
    - sheets_to_read (list of str): A corresponding list of sheet names within each Excel file to be read.
    - target_column_names_list (list of list of str, optional): A list where each element is a list of column
names to be read from the respective Excel sheet. If None, all columns are read. Default is None.
    - reset_indexes (bool): If True, the index of the concatenated DataFrame will be reset. Default is False.

    Returns:
    - pandas.DataFrame: A DataFrame containing the concatenated data from the specified Excel files and sheets.
    """

    master_df = pd.DataFrame()

    if target_column_names_list is None:
        target_column_names_list = [None] * len(files_to_read)

    for file_name, sheet_name, column_list in zip(files_to_read, sheets_to_read, target_column_names_list):
        temp_df = pd.read_excel(
                    data_files_path + file_name,
                    sheet_name=sheet_name,
                    usecols=column_list)

        master_df = pd.concat([master_df, temp_df])

    if reset_indexes:
        master_df.reset_index(drop=True, inplace=True)

    return master_df


# Demonstration:

result_df = load_excel_files(data_files_path, files_to_read, sheets_to_read, target_column_names_list,
reset_indexes=False)
result_df
```

5. Creating a python function for deduplicating data.

Once the excel workbooks have bean joined to form one dataframe, it is possible that there are duplicated entries/rows in the dataframe.

1. The function below removes duplicates by using a subset of columns to identify which entries have been duplicated.
2. Caution must be practiced when choosing which columns to use to identify duplicates.

```
def remove_duplicates(dataframe, key_columns_list=None):
    """
    Remove duplicates from a DataFrame based on a subset of columns.

    Parameters:
    - dataframe: The DataFrame to deduplicate.
    - key_columns_list: List of column names to consider for finding duplicates.
                        If None, all columns are considered.

    Returns:
    - A DataFrame with duplicates removed, keeping only the second entry in case of duplication.
    """

    # Check if key_columns_list is provided, otherwise use all columns
    if key_columns_list is None:
        key_columns_list = dataframe.columns.tolist()

    # Remove duplicates, keeping the second occurrence
    deduped_df = dataframe.drop_duplicates(subset=key_columns_list, keep='last')

    return deduped_df


# 1. An example where all columns are used to look for duplicates:

remove_duplicates(dataframe=result_df, key_columns_list=None)

# Since each row is unique (when all columns are considered), the dataframe is returned as is.

# 2. An example where columns A & C are used to look for duplicates:

remove_duplicates(dataframe=result_df, key_columns_list=["A","C"])

# Since each row is not unique (when onlt considering columns A & C), only a subset of the dataframe is
returned, with only the last entry/version of each duplicate being kept.
```

6. Saving the resulting file in a shared location.

1. Once files have been read & deduped, they should be saved in the project's Data folder.
2. In order to be able to distinguish between different version of files, the format below should be used to store files:
    a. File_name_yyyy-mm-dd-hh-ss.CSV

```
def save_data_to_folder(file_name, dataframe, data_files_path="", sep=",", index=False):
    """
    Saves a given DataFrame as a CSV file in the specified file path with a timestamp.
    The file will be saved with the format: file_name_yyyy-mm-dd-hh-ss.CSV

    Parameters:
    - file_name: The base name for the file.
    - dataframe: The DataFrame to be saved as a CSV file.
    - data_files_path: The folder path where the CSV file will be saved.
    - sep: Separator to be used in the CSV file. Default is comma (',').
    - index: Whether to write the DataFrame's index. Default is False.
    """

    # Format the current timestamp
    timestamp = datetime.now().strftime("%Y-%m-%d_%H-%M")

    # Construct the full file path
    full_file_path = "{}{}_{}.csv".format( data_files_path, file_name, timestamp)

    try:
        # Save DataFrame to CSV file
        dataframe.to_csv(full_file_path, sep=sep, index=index)
        print(f"Data successfully saved to {full_file_path}")
    except Exception as e:
        print(f"An error occurred while saving the data: {e}")


# Demonstration

save_data_to_folder(
    file_name="example_file", # File name to be used as prefix
    dataframe=result_df, # Dataframe to save
    data_files_path=data_files_path, # Path to save to (e.g. ../Data/)
    sep=",", # Separator to use in CSV file
    index=False # Whether to keep index or not
    )
```

Ends