# **Combining Two Excel Sheets into One Using Python**

This document explains the process of combining two sheets from a single Excel file into one consolidated sheet using Python. The pandas library is used for reading, combining, and saving the data.

## **Steps Followed**

1. Import the pandas library for data manipulation.
2. Load the Excel file that contains two sheets.
3. Read both sheets separately using pandas.read\_excel().
4. Combine the data from both sheets using pandas.concat().
5. Export the combined data to a new Excel file using to\_excel().

## **Python Code**

import pandas as pd

# Load the Excel file

file\_path = "ConsolidatedIndianFinancialSystemCode.xlsx"

# Read both sheets

sheet1 = pd.read\_excel(file\_path, sheet\_name=0)

sheet2 = pd.read\_excel(file\_path, sheet\_name=1)

# Combine the two sheets

combined = pd.concat([sheet1, sheet2], ignore\_index=True)

# Save to new Excel

combined.to\_excel("combined\_file.xlsx", index=False

# **Cleaning the STATE Column in Dataset**

## **Introduction**

The dataset contains a **STATE** column that includes multiple issues such as:

* Abbreviations (e.g., *MP, TN, UP, MH*).
* Misspellings (e.g., *ODHISA, GUJRAT, CHATTISGARH*).
* Extra spaces (e.g., \*DELHI \*, \*CHHATTISGARH \*).
* Old names (e.g., *PONDICHERRY* instead of *PUDUCHERRY*).
* Invalid entries (e.g., *branch names or junk values*).

The goal of cleaning is to **standardize all state names** so that the dataset is consistent and reliable.

## **Steps Followed**

1. **Normalize text**
   * Converted all state names to **uppercase**.
   * Removed leading and trailing spaces.
2. **Replace variations and abbreviations**
   * Abbreviations such as *MP → MADHYA PRADESH*, *TN → TAMIL NADU*.
   * Variations like *TAMILNADU → TAMIL NADU*, *UTTA PRADES → UTTAR PRADESH*.
3. **Correct misspellings**
   * *ODHISA → ODISHA*
   * *GUJRAT → GUJARAT*
   * *CHHATISGARH/CHATTISGARH → CHHATTISGARH*
4. **Update old names**
   * *PONDICHERRY → PUDUCHERRY*
5. **Validate states**
   * Compared cleaned entries with a master list of **valid states/UTs of India**.
   * Non-matching entries were marked as **“UNKNOWN”**.

## **Python Code**

# Step 1: Normalize text

df["STATE"] = df["STATE"].astype(str).str.strip().str.upper()

# Step 2: Replace variations

replace\_map = {

"MP": "MADHYA PRADESH",

"MADHY PRADESH": "MADHYA PRADESH",

"MADHYAPRADESH": "MADHYA PRADESH",

"MH": "MAHARASHTRA",

"GUJRAT": "GUJARAT",

" GUJARAT ": "GUJARAT",

"TN": "TAMIL NADU",

"TAMILNADU": "TAMIL NADU",

"TAMILNADU ": "TAMIL NADU",

"UTTA PRADES": "UTTAR PRADESH",

"UTTARPRADESH": "UTTAR PRADESH",

"UP": "UTTAR PRADESH",

"CHHATISGARH": "CHHATTISGARH",

"CHATTISGARH": "CHHATTISGARH",

"CHHATISHGARH": "CHHATTISGARH",

"DELHI ": "DELHI",

"NCT OF DELHI": "DELHI",

"NEW DELHI": "DELHI",

"PONDICHERRY": "PUDUCHERRY",

"ANDHRAPRADESH": "ANDHRA PRADESH",

"AP": "ANDHRA PRADESH",

"KA": "KARNATAKA",

"KARANATAKA": "KARNATAKA",

"HARKHAND": "JHARKHAND",

"JH": "JHARKHAND",

"ORISSA": "ODISHA",

"ODHISA": "ODISHA",

"WESTBENGAL": "WEST BENGAL",

"HIMACHALPRADESH": "HIMACHAL PRADESH",

"HIMANCHAL PRADESH": "HIMACHAL PRADESH",

"UTTARKHAND": "UTTARAKHAND",

"UTTARANCHAL": "UTTARAKHAND",

}

df["STATE"] = df["STATE"].replace(replace\_map)

# Step 3: Validate against official states/UTs

valid\_states = [

'MAHARASHTRA','GUJARAT','KARNATAKA','TELANGANA','HARYANA','MADHYA PRADESH',

'ANDHRA PRADESH','RAJASTHAN','GOA','DELHI','PUNJAB','UTTAR PRADESH',

'CHHATTISGARH','HIMACHAL PRADESH','CHANDIGARH','JAMMU AND KASHMIR','ODISHA',

'WEST BENGAL','TAMIL NADU','ASSAM','UTTARAKHAND','BIHAR','JHARKHAND',

'KERALA','PUDUCHERRY','TRIPURA','NAGALAND','MIZORAM','DAMAN AND DIU',

'MANIPUR','DADRA AND NAGAR HAVELI','ANDAMAN AND NICOBAR ISLANDS','SIKKIM',

'MEGHALAYA','ARUNACHAL PRADESH','LADAKH','LAKSHADWEEP'

]

df["STATE"] = df["STATE"].where(df["STATE"].isin(valid\_states), other="")

## **STD Code Cleaning and Mapping Documentation**

## **Objective**

The purpose of this script is to clean and standardize the CITY1 and STD CODE columns in a dataset by mapping cities to their correct STD codes, while preserving existing values if they already contain valid information.

## **Steps Followed**

### **1. Normalize City Names**

df['CITY1'] = df['CITY1'].astype(str).str.strip().str.upper()

* Converts all values in CITY1 to string.
* Strips leading and trailing spaces.
* Converts names to uppercase for consistency.

### **2. Apply City to STD Code Mapping**

df['STD CODE'] = df.apply(

lambda r: city\_std\_map[r['CITY1']] if r['CITY1'] in city\_std\_map else r['STD CODE'],

axis=1

)

* If a city exists in the mapping dictionary, update its STD CODE.
* If not, retain the existing STD CODE.

### **3. Clean Formatting of STD CODE**

df['STD CODE'] = (

df['STD CODE']

.astype(str)

.str.extract(r'(\d+)')[0] # extract only digits

.fillna(df['STD CODE']) # if no match, keep original

)

* Keeps only digits from the STD CODE column.
* If extraction fails, preserves the original value.

### **4. Convert to Numeric (Optional)**

df['STD CODE'] = pd.to\_numeric(df['STD CODE'], errors='coerce').fillna(df['STD CODE'])

* Converts valid codes into integers (removes leading zeros automatically).
* If conversion fails, the original value is retained.

## **Example**

**Input Data:**

| **CITY1** | **STD CODE** |
| --- | --- |
| Vizianagaram | 08945 |
| Salur | 08941 |
| UnknownCity | 01234 |

**After Processing:**

| **CITY1** | **STD CODE** |
| --- | --- |
| VIZIANAGARAM | 8945 |
| SALUR | 8941 |
| UNKNOWNCITY | 1234 |

### **Step: Handle Duplicate IFSC Codes**

**Remove duplicate IFSC entries** Since IFSC codes must be unique, we drop duplicate rows based on the "IFSC" column, keeping only the first occurrence.  
  
 # Drop duplicate IFSC codes (keep the first occurrence)

df = df.drop\_duplicates(subset=["IFSC"], keep="first")

**Validate uniqueness** After removing duplicates, check how many unique IFSC codes are present in the dataset.  
  
 # Count unique IFSC codes

1. df["IFSC"].nunique()

**CLEANING STD CODE and PHONE**

### **Step 1️⃣ – Clean STD CODE**

df["STD CODE"] = (

df["STD CODE"]

.astype(str) # Convert all values to string (handles numeric values like 22.0 → "22.0")

.str.replace(".0", "", regex=False) # Remove trailing '.0' introduced by Excel/CSV

.replace({"nan": "", "NaN": ""}) # Replace literal strings 'nan'/'NaN' with empty string

.fillna("") # Replace actual NaN values with empty string

)

**Result:**

* 22.0 → 22
* 0.0 → 0
* NaN → `` (empty)

### **Step 2️⃣ – Validate STD CODE**

df["STD CODE"] = df["STD CODE"].apply(lambda x: "NA" if x.strip() in ["", "0"] else x)

**Logic:**

* If value is **empty** or "0" → mark as "NA" (invalid code).
* Else keep the valid STD code.

**Result:**

* 0 → NA
* `` (empty) → NA
* 22 → 22

### 

### 

### **Clean PHONE**

df["PHONE"] = (

df["PHONE"]

.astype(str) # Convert numbers to string

.str.replace(".0", "", regex=False) # Remove trailing '.0' (Excel formatting issue)

.replace({"nan": "", "NaN": ""}) # Replace text 'nan'/'NaN' with empty

.fillna("") # Replace actual NaN values with empty string

)

**Result:**

* 24702643.0 → 24702643
* 9653261383.0 → 9653261383
* NaN → ``

### **Step 4️⃣ – Validate PHONE**

df["PHONE"] = df["PHONE"].apply(lambda x: "NA" if x.strip() == "" else x)

**Logic:**

* If blank after cleaning → replace with "NA".
* Keep valid numbers (both mobile & landline).

**Result:**

* `` (empty) → NA
* 9653261383 → 9653261383
* 24702643 → 24702643

### **✅ Final Output Example**

| **STD CODE (raw)** | **PHONE (raw)** | **STD CODE (cleaned)** | **PHONE (cleaned)** |
| --- | --- | --- | --- |
| 22.0 | 24702643.0 | 22 | 24702643 |
| 0.0 | 9653261383 | NA | 9653261383 |
| 120.0 | 7217894931 | 120 | 7217894931 |
| NaN | NaN | NA | NA |

# 

# 

# 

# **Python Script for Loading Bank Data into PostgreSQL**

This script reads the **cleaned bank details CSV** and inserts the records into a PostgreSQL table.

**Step 1️⃣ – Import Libraries**

import psycopg2

import pandas as pd

* pandas → used to read and process the cleaned CSV file.
* psycopg2 → PostgreSQL adapter for Python, used to connect and insert data.

### **Step 2️⃣ – Load Cleaned CSV**

df = pd.read\_csv("Cleaned.csv")

* Reads the **Cleaned.csv** file containing validated bank data.

### **Step 3️⃣ – Standardize Column Names**

df.columns = ["bank", "ifsc", "branch", "address", "city1", "city2", "state", "stdCode", "phone"]

* Ensures column names in DataFrame **match the PostgreSQL table schema** (bank\_details).
* Example mapping:  
  + BANK → bank
  + IFSC → ifsc
  + STD CODE → stdCode

### **Step 4️⃣ – Connect to PostgreSQL**

conn = psycopg2.connect(

host="localhost",

database="Bank",

user="postgres",

password="Shan",

port="5432"

)

cur = conn.cursor()

* Establishes connection to the PostgreSQL database.
* Creates a cursor object (cur) for executing SQL queries.

### **Step 5️⃣ – Insert Data**

for \_, row in df.iterrows():

cur.execute("""

INSERT INTO m\_bank\_master(bank, ifsc, branch, address, city1, city2, state, stdCode, phone)

VALUES (%s, %s, %s, %s, %s, %s, %s, %s, %s)

ON CONFLICT (ifsc) DO NOTHING;

""", (

row["bank"],

row["ifsc"],

row["branch"],

row["address"],

row["city1"],

row["city2"],

row["state"],

row["stdCode"],

row["phone"]

))

**Details:**

* Iterates through each row of the DataFrame.
* Inserts values into the bank\_details table.
* Uses **parameterized query** (%s) to prevent SQL injection.
* ON CONFLICT (ifsc) DO NOTHING; → avoids duplicate records based on the **IFSC (unique key)**.

### **Step 6️⃣ – Commit and Close Connection**

conn.commit()

cur.close()

conn.close()

* Commits all insertions to the database.
* Closes the cursor and database connection safely.

### **Step 7️⃣ – Confirmation**

print("✅ Data inserted successfully,")

* Prints a confirmation message after successful execution.

# **SQL Script for bank\_details Table**

### **Step 1️⃣ – Create Table**

CREATE TABLE bank\_details (

bank\_id SERIAL PRIMARY KEY, -- Auto-incremented unique ID for each record

bank VARCHAR(255), -- Bank name

ifsc VARCHAR(20) UNIQUE, -- IFSC code (unique for each branch)

branch VARCHAR(255), -- Branch name

address TEXT, -- Full branch address

city1 VARCHAR(255), -- City (primary)

city2 VARCHAR(255), -- City (secondary, if any)

state VARCHAR(100), -- State

stdCode VARCHAR(20), -- STD code (landline code or NA if not available)

phone VARCHAR(50) -- Phone number (mobile/landline or NA if not available)

);

### **Step 2️⃣ – Column Details**

| **Column** | **Data Type** | **Constraints** | **Description** |
| --- | --- | --- | --- |
| bank\_id | SERIAL | PRIMARY KEY | Auto-generated unique identifier for each row. |
| bank | VARCHAR(255) | – | Bank name (e.g., “State Bank of India”). |
| ifsc | VARCHAR(20) | UNIQUE | Unique **IFSC code** for each branch. Duplicate entries are not allowed. |
| branch | VARCHAR(255) | – | Branch name (e.g., “Mumbai Main”). |
| address | TEXT | – | Full postal address of the branch. |
| city1 | VARCHAR(255) | – | Primary city name. |
| city2 | VARCHAR(255) | – | Alternate/secondary city name (if applicable). |
| state | VARCHAR(100) | – | State where the branch is located. |
| stdCode | VARCHAR(20) | – | STD (landline code), cleaned (no .0 or 0, replaced with NA). |
| phone | VARCHAR(50) | – | Phone number (landline or mobile). Cleaned so no blanks, .0, or NULL. |

### **Step 3️⃣ – Key Constraints**

**Primary Key** bank\_id SERIAL PRIMARY KEY

* + Automatically generates unique IDs (1, 2, 3...) for each inserted row.

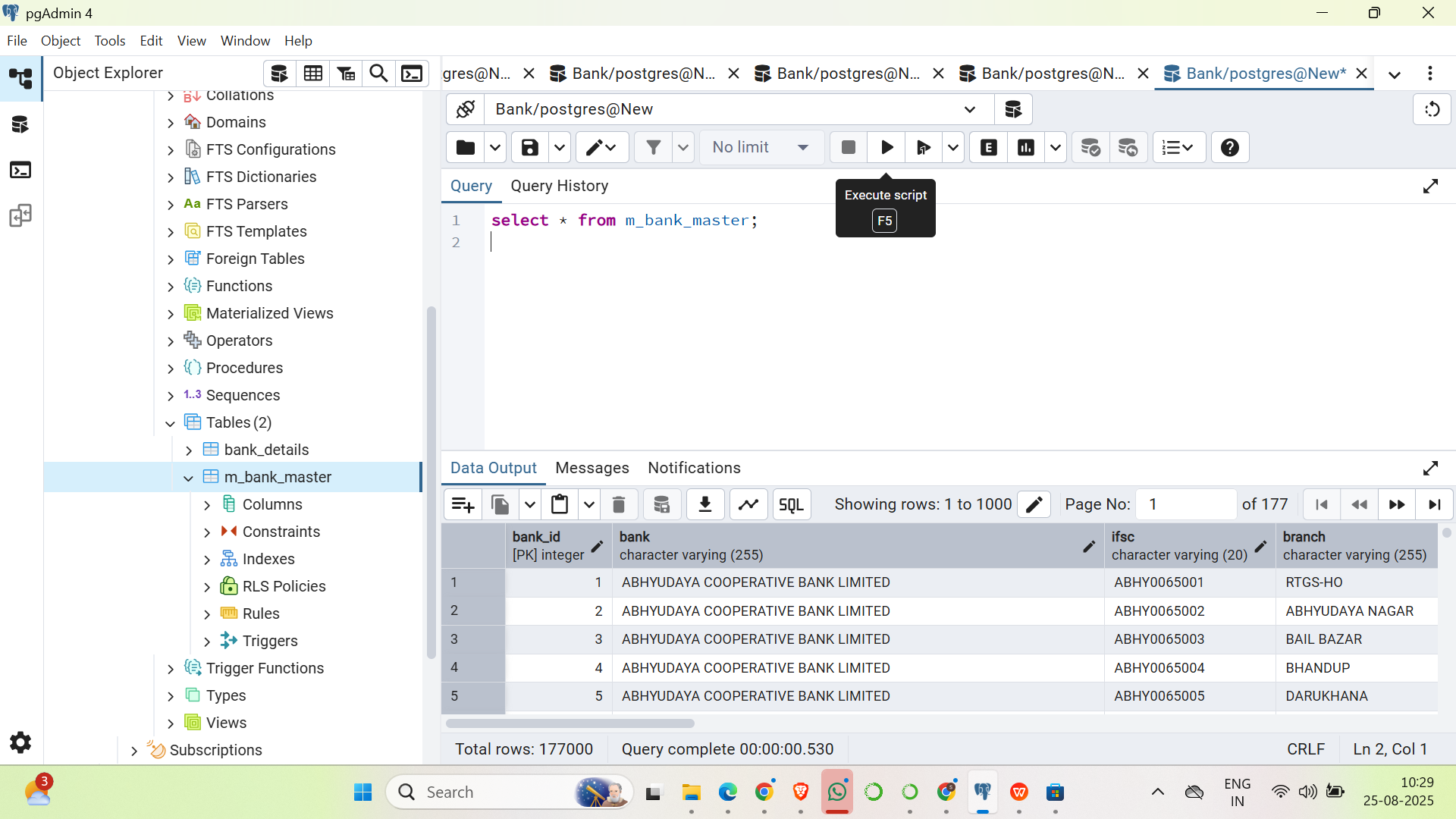
**Unique Constraint** ifsc VARCHAR(20) UNIQUE

* + Ensures no two branches have the same **IFSC code**.
  + Used in ON CONFLICT (ifsc) DO NOTHING; in the Python script to avoid duplicates.

### 

### **Step 4️⃣ – Usage Notes**

* Always **clean data before insertion**:  
  + IFSC → must match standard regex (^[A-Z]{4}0[A-Z0-9]{6}$).
  + STD code & Phone → replace .0, 0, NULL with "NA"



# **Duplicate IFSC Check**

### **Step 1️⃣ – Purpose**

* To ensure **data integrity** in the m\_bank\_master (or bank\_details) table.
* IFSC must be **unique per branch** (already enforced by UNIQUE constraint in table schema).
* This query helps **identify duplicates** before insertion or after data load (in case raw data was inserted without constraints).

### **Step 2️⃣ – SQL Script**

-- Duplicate IFSC Validation

SELECT ifsc, COUNT(\*)

FROM m\_bank\_master

GROUP BY ifsc

HAVING COUNT(\*) > 1;

### **Step 3️⃣ – Explanation**

1. **GROUP BY ifsc**
   * Groups all records by IFSC code.
2. **COUNT(\*)**
   * Counts how many times each IFSC appears.
3. **HAVING COUNT(\*) > 1**
   * Filters results to only show IFSC codes appearing more than once.
   * This identifies **duplicate IFSC entries**.

