Data Cleaning Code

```
import pandas as pd
import numpy as np
from datetime import datetime
import re

"""
Retail_Live_Project_Dataset – Cleaning Utility
----------------------------------------------
• Reads either **Retail_Live_Project_Dataset.xlsx** (default) or a
user-supplied path.
• Applies column-name normalisation, type coercion, category harmonisation and
  data-quality flagging tailored to the schema that ships with the live
project
  workbook (≈ 16 columns).
• Writes a side-by-side file with **_cleaned** suffix, preserving the original
  file format (CSV ⇆ Excel).

Run from shell:
    python clean_retail_data.py  # uses default Excel file in cwd
    python clean_retail_data.py my_raw_file.csv
"""


CATEGORY_MAPPINGS: dict[str, str] = {
    "laptop": "Electronics",
    "sofa": "Home",
    "shoes": "Fashion",
    "shampoo": "Beauty",
    "novel": "Books",
}

TEXT_COLUMNS = [
    "City",
    "Product_Category",
    "Product_Name",
    "Payment_Mode",
    "Delivery_Status",
    "Customer_Name",
    "Email",
    "Gender",
    "Country",
]

NUMERIC_COLUMNS = [
    "Purchase_Amount",
    "Discount_Offered",
    "Customer_Satisfaction",
    "Age",
]
```

```
DATE_COLUMNS = ["Purchase_Date"]

PAYMENT_MODE_MAP = {
    "upi": "UPI",
    "debit card": "Debit Card",
    "credit card": "Credit Card",
    "cash": "Cash",
    "net banking": "Net Banking",
}

DELIVERY_STATUS_MAP = {
    "delivered": "Delivered",
    "pending": "Pending",
    "cancelled": "Cancelled",
    "returned": "Returned",
}

GENDER_MAP = {"male": "Male", "female": "Female", "m": "Male", "f": "Female"}

EMAIL_REGEX = re.compile(r"^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$")


def _read_file(path: str) -> pd.DataFrame:
    """Auto-detect loader based on extension."""
    if path.lower().endswith(".csv"):
        return pd.read_csv(path)
    if path.lower().endswith((".xls", ".xlsx")):
        return pd.read_excel(path)
    raise ValueError("Unsupported file format → use CSV or Excel.")


def _write_file(df: pd.DataFrame, original_path: str) -> str:
    """Save cleaned data next to original and return the new path."""
    cleaned_path = re.sub(r"\.(csv|xlsx?|xls)$", r"_cleaned.\1",
original_path, flags=re.I)
    if cleaned_path.lower().endswith(".csv"):
        df.to_csv(cleaned_path, index=False)
    else:
        df.to_excel(cleaned_path, index=False)
    return cleaned_path


def clean_dataset(df: pd.DataFrame) -> pd.DataFrame:
    """Primary cleaning pipeline aligned with Retail_Live_Project_Dataset
schema."""

    df = df.copy()
```

```
    # 1 column name hygiene
    df.columns = df.columns.str.strip()

    # 2 text normalisation
    for col in TEXT_COLUMNS:
        if col in df.columns:
            df[col] = df[col].astype(str).str.strip()

    # 3 product category correction (vectorised for speed)
    mask_any = pd.Series(False, index=df.index)
    for key, cat in CATEGORY_MAPPINGS.items():
        key_mask = df["Product_Name"].str.lower().str.contains(key, na=False)
        df.loc[key_mask, "Product_Category"] = cat
        mask_any |= key_mask
    # the rest remain as-is

    # 4 numeric coercion
    for col in NUMERIC_COLUMNS:
        if col in df.columns:
            df[col] = pd.to_numeric(df[col], errors="coerce")

    # 5 date parsing
    for col in DATE_COLUMNS:
        if col in df.columns:
            df[col] = pd.to_datetime(df[col], errors="coerce", dayfirst=False)

    # 6 categorical harmonisation
    if "Payment_Mode" in df.columns:
        df["Payment_Mode"] = (
            df["Payment_Mode"].str.lower().map(PAYMENT_MODE_MAP).fillna(df["Pa
yment_Mode"])
        )
    if "Delivery_Status" in df.columns:
        df["Delivery_Status"] = (
            df["Delivery_Status"].str.lower().map(DELIVERY_STATUS_MAP).fillna(
df["Delivery_Status"])
        )
    if "Gender" in df.columns:
        df["Gender"] =
df["Gender"].str.lower().map(GENDER_MAP).fillna(df["Gender"])

    # 7 email validation & formatting
    if "Email" in df.columns:
        df["Email"] = df["Email"].str.lower().str.strip()
        df["Email_Valid"] = df["Email"].str.match(EMAIL_REGEX, na=False)

    # 8 satisfaction clipping (1-5)
    if "Customer_Satisfaction" in df.columns:
```

```
        df["Customer_Satisfaction"] = df["Customer_Satisfaction"].clip(1, 5)

    # 9️⃣ proper-case cities / countries
    for loc_col in ("City", "Country"):
        if loc_col in df.columns:
            df[loc_col] = df[loc_col].str.title()

    # 🔟 quality flags
    df["Data_Quality_Issues"] = ""
    if "Customer_Name" in df.columns:
        miss_name = df["Customer_Name"].eq("") | df["Customer_Name"].isna()
        df.loc[miss_name, "Data_Quality_Issues"] += "Missing Customer Name; "
    if "Email" in df.columns:
        df.loc[~df["Email_Valid"], "Data_Quality_Issues"] += "Invalid Email; "
    if "Purchase_Amount" in df.columns:
        bad_amt = df["Purchase_Amount"].le(0) | df["Purchase_Amount"].isna()
        df.loc[bad_amt, "Data_Quality_Issues"] += "Invalid Purchase Amount; "

    # optional summary to stdout (can be removed in production)
    print("Cleaning summary →", {
        "rows": len(df),
        "issues": (df["Data_Quality_Issues"] != "").sum(),
    })

    return df


def load_clean_validate(file_path: str) -> pd.DataFrame | None:
    """Convenience wrapper used by __main__."""
    try:
        df_raw = _read_file(file_path)
        df_clean = clean_dataset(df_raw)
        out_path = _write_file(df_clean, file_path)
        print(f"✔️ Saved cleaned data → {out_path}\n")

        # simple validation report
        print("Validation snapshot:\n",
              df_clean.isnull().sum().to_frame("nulls").T)
        return df_clean
    except Exception as exc:
        print("❌", exc)
        return None


if __name__ == "__main__":
    import sys, pathlib

    # default to the live-project workbook in current directory
    default_file = "Retail_Live_Project_Dataset.xlsx"
```

```
    target = pathlib.Path(sys.argv[1]) if len(sys.argv) > 1 else
pathlib.Path(default_file)

    if not target.exists():
        raise FileNotFoundError(f"File not found → {target.resolve()}")

    load_clean_validate(str(target))
```

Data Cleaning Code Breakdown

🧠 1. Read and Detect File Type

```
python
CopyEdit
def _read_file(path: str) -> pd.DataFrame:
```

- Automatically detects whether the input file is a **CSV** or **Excel** (`.xls/.xlsx`) and loads it into a DataFrame using pandas.

🧹 2. Column Name Cleaning

```
python
CopyEdit
df.columns = df.columns.str.strip()
```

- Removes extra whitespace around column names, so `" Product_Name "` becomes `"Product_Name"`.

📜 3. Text Column Cleanup

```
python
CopyEdit
TEXT_COLUMNS = [...]
df[col] = df[col].astype(str).str.strip()
```

- Converts text columns to string format and removes leading/trailing whitespace.
- Ensures consistent formatting across columns like `"City"`, `"Gender"`, `"Product_Name"`, etc.

🗂️ 4. Fix Product Category Using Product Name

```
python
CopyEdit
CATEGORY_MAPPINGS = {...}
```

- If `"Product_Name"` contains:
  - `"laptop"` → Category becomes `"Electronics"`
  - `"sofa"` → `"Home"`, etc.
- Uses `.str.contains()` to find keywords in product names and correct categories accordingly.

🔢 5. Convert Numeric Columns

```
python
CopyEdit
NUMERIC_COLUMNS = [...]
df[col] = pd.to_numeric(df[col], errors="coerce")
```

- Converts columns like `"Purchase_Amount"`, `"Age"`, `"Discount_Offered"` to numeric.
- Any invalid values (like text in a numeric column) are turned into `NaN`.

📅 6. Date Column Parsing

```
python
CopyEdit
DATE_COLUMNS = ["Purchase_Date"]
df[col] = pd.to_datetime(df[col], errors="coerce")
```

- Converts `"Purchase_Date"` to a proper date format.
- Invalid or unreadable dates become `NaT` (Not a Time).

🏷️ 7. Standardize Categorical Columns

```
python
CopyEdit
PAYMENT_MODE_MAP, DELIVERY_STATUS_MAP, GENDER_MAP
```

- Converts values like `"credit card"`, `"CREDIT CARD"` → `"Credit Card"`
- Also fixes `"F"` → `"Female"`, `"m"` → `"Male"` using `.str.lower().map(...)`

📧 8. Email Cleaning and Validation

```
python
CopyEdit
EMAIL_REGEX = ...
df["Email_Valid"] = df["Email"].str.match(...)
```

- Converts all emails to lowercase.
- Uses a regular expression to flag valid emails.
- Invalid emails get marked as `False` in `Email_Valid`.

📊 9. Clip Satisfaction Score

```
python
CopyEdit
df["Customer_Satisfaction"] = df["Customer_Satisfaction"].clip(1, 5)
```

- Ensures satisfaction ratings are between **1 and 5**.
- If someone gave `"0"` or `"10"`, it's adjusted within range.

🏙️ 10. Proper Case for City and Country

```
python
CopyEdit
df[loc_col] = df[loc_col].str.title()
```

- Capitalizes first letter of each word in `"City"` and `"Country"`.
    - e.g., `"new york"` → `"New York"`

🚩 11. Add Data Quality Flags

```
python
CopyEdit
df["Data_Quality_Issues"] = ""
```

Flags records for:

- Missing `"Customer_Name"`
- Invalid `"Email"`
- Invalid or missing `"Purchase_Amount"`

Each issue is appended in a single `"Data_Quality_Issues"` column for transparency.

📁 12. Save Cleaned File

```
python
CopyEdit
def _write_file(...):
```

- Saves the cleaned dataset **next to the original**, renaming it like:
    - `Retail_Live_Project_Dataset_cleaned.xlsx`
    - or `your_file_cleaned.csv` (preserves format)

☑️ 13. Validation Snapshot

```
python
CopyEdit
df_clean.isnull().sum().to_frame("nulls")
```

- After cleaning, prints:

- Total number of nulls per column
- Count of rows with data quality issues

🧠 14. Main Runner

```
python
CopyEdit
if __name__ == "__main__":
```

- Allows running the script via:

```
bash
CopyEdit
python clean_retail_data.py
# or
python clean_retail_data.py your_file.csv
```

📜 Example Output Summary:

```
plaintext
CopyEdit
Cleaning summary → {'rows': 1000, 'issues': 48}
✔ Saved cleaned data → Retail_Live_Project_Dataset_cleaned.xlsx

Validation snapshot:
            nulls
City            2
Country         0
Purchase_Date   0
...
```

Python Code for Columns to add in Cleaned Dataset

```
import pandas as pd
import numpy as np
from datetime import datetime
import re

"""
Retail_Live_Project_Dataset – Enhanced Cleaning Utility with PowerBI Columns
---------------------------------------------------------------------------
• Reads either **Retail_Live_Project_Dataset.xlsx** (default) or a user-supplied path.
• Applies column-name normalisation, type coercion, category harmonisation and
  data-quality flagging tailored to the schema that ships with the live project
  workbook (≈ 16 columns).
• ENHANCED: Adds calculated columns for PowerBI dashboard creation
• Writes a side-by-side file with **_cleaned** suffix, preserving the original
  file format (CSV ⇆ Excel).

Run from shell:
    python clean_retail_data.py  # uses default Excel file in cwd
    python clean_retail_data.py my_raw_file.csv
"""



CATEGORY_MAPPINGS: dict[str, str] = {
    "laptop": "Electronics",
    "sofa": "Home",
    "shoes": "Fashion",
    "shampoo": "Beauty",
    "novel": "Books",
}

TEXT_COLUMNS = [
```

```
    "City",
    "Product_Category",
    "Product_Name",
    "Payment_Mode",
    "Delivery_Status",
    "Customer_Name",
    "Email",
    "Gender",
    "Country",
]

NUMERIC_COLUMNS = [
    "Purchase_Amount",
    "Discount_Offered",
    "Customer_Satisfaction",
    "Age",
]

DATE_COLUMNS = ["Purchase_Date"]

PAYMENT_MODE_MAP = {
    "upi": "UPI",
    "debit card": "Debit Card",
    "credit card": "Credit Card",
    "cash": "Cash",
    "net banking": "Net Banking",
}

DELIVERY_STATUS_MAP = {
    "delivered": "Delivered",
    "pending": "Pending",
```

```
    "cancelled": "Cancelled",
    "returned": "Returned",
}

GENDER_MAP = {"male": "Male", "female": "Female", "m": "Male", "f": "Female"}

EMAIL_REGEX = re.compile(r"^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$")


def _read_file(path: str) -> pd.DataFrame:
    """Auto-detect loader based on extension."""
    if path.lower().endswith(".csv"):
        return pd.read_csv(path)
    if path.lower().endswith((".xls", ".xlsx")):
        return pd.read_excel(path)
    raise ValueError("Unsupported file format → use CSV or Excel.")


def _write_file(df: pd.DataFrame, original_path: str) -> str:
    """Save cleaned data next to original and return the new path."""
    cleaned_path = re.sub(r"\.(csv|xlsx?|xls)$", r"_cleaned.\1", original_path, flags=re.I)
    if cleaned_path.lower().endswith(".csv"):
        df.to_csv(cleaned_path, index=False)
    else:
        df.to_excel(cleaned_path, index=False)
    return cleaned_path


def add_calculated_columns(df: pd.DataFrame) -> pd.DataFrame:
    """Add calculated columns for PowerBI dashboard creation."""
```

```
# 1 REVENUE CALCULATIONS
if "Purchase_Amount" in df.columns and "Discount_Offered" in df.columns:
    # Net Revenue (Purchase Amount - Discount)
    df["Net_Revenue"] = df["Purchase_Amount"] - df["Discount_Offered"]

    # Discount Percentage
    df["Discount_Percentage"] = (df["Discount_Offered"] / df["Purchase_Amount"]) * 100
    df["Discount_Percentage"] = df["Discount_Percentage"].round(2)

    # Revenue Categories
    df["Revenue_Category"] = pd.cut(
        df["Net_Revenue"],
        bins=[0, 100, 300, 500, 1000, float('inf')],
        labels=["Low (0-100)", "Medium (100-300)", "High (300-500)", "Premium (500-1000)", "Luxury
(1000+)"],
        right=False
    )

# 2 DATE-BASED CALCULATIONS
if "Purchase_Date" in df.columns:
    # Extract date components
    df["Year"] = df["Purchase_Date"].dt.year
    df["Month"] = df["Purchase_Date"].dt.month
    df["Month_Name"] = df["Purchase_Date"].dt.strftime("%B")
    df["Quarter"] = df["Purchase_Date"].dt.quarter
    df["Day_of_Week"] = df["Purchase_Date"].dt.dayofweek
    df["Day_Name"] = df["Purchase_Date"].dt.strftime("%A")
    df["Week_of_Year"] = df["Purchase_Date"].dt.isocalendar().week

    # Season classification
    df["Season"] = df["Month"].map({
```

```
        12: "Winter", 1: "Winter", 2: "Winter",
        3: "Spring", 4: "Spring", 5: "Spring",
        6: "Summer", 7: "Summer", 8: "Summer",
        9: "Fall", 10: "Fall", 11: "Fall"
    })

    # Days since purchase (for recency analysis)
    df["Days_Since_Purchase"] = (datetime.now() - df["Purchase_Date"]).dt.days

    # Purchase recency categories
    df["Purchase_Recency"] = pd.cut(
        df["Days_Since_Purchase"],
        bins=[0, 30, 90, 180, 365, float('inf')],
        labels=["Recent (0-30 days)", "Medium (30-90 days)", "Old (90-180 days)",
                "Very Old (180-365 days)", "Stale (365+ days)"],
        right=False
    )

# 3 CUSTOMER SEGMENTATION
if "Age" in df.columns:
    # Age groups
    df["Age_Group"] = pd.cut(
        df["Age"],
        bins=[0, 25, 35, 45, 55, 65, 100],
        labels=["18-25", "26-35", "36-45", "46-55", "56-65", "65+"],
        right=False
    )

# 4 BUSINESS METRICS
if "Customer_Satisfaction" in df.columns:
    # Satisfaction categories
```

```
    df["Satisfaction_Category"] = df["Customer_Satisfaction"].map({
        1: "Very Dissatisfied",
        2: "Dissatisfied",
        3: "Neutral",
        4: "Satisfied",
        5: "Very Satisfied"
    })

    # NPS-like scoring
    df["NPS_Category"] = df["Customer_Satisfaction"].map({
        1: "Detractor", 2: "Detractor",
        3: "Passive",
        4: "Promoter", 5: "Promoter"
    })

# 5 DELIVERY & PAYMENT ANALYSIS
if "Delivery_Status" in df.columns:
    # Success flag
    df["Delivery_Success"] = df["Delivery_Status"].map({
        "Delivered": 1,
        "Pending": 0,
        "Cancelled": 0,
        "Returned": 0
    })

    # Order completion flag
    df["Order_Completed"] = df["Delivery_Status"].isin(["Delivered"]).astype(int)

if "Payment_Mode" in df.columns:
    # Digital vs Traditional payment
    df["Payment_Type"] = df["Payment_Mode"].map({
```

```
        "UPI": "Digital",
        "Net Banking": "Digital",
        "Debit Card": "Digital",
        "Credit Card": "Digital",
        "Cash": "Traditional"
    })

# 6 PRODUCT ANALYSIS
if "Product_Category" in df.columns:
    # Category priority (for business focus)
    category_priority = {
        "Electronics": "High",
        "Fashion": "High",
        "Home": "Medium",
        "Beauty": "Medium",
        "Books": "Low"
    }
    df["Category_Priority"] = df["Product_Category"].map(category_priority)

# 7 GEOGRAPHICAL ANALYSIS
if "Country" in df.columns:
    # Market classification (customize based on your business)
    market_classification = {
        "India": "Domestic",
        "United States": "International",
        "Canada": "International",
        "United Kingdom": "International",
        "Australia": "International"
    }
    df["Market_Type"] = df["Country"].map(market_classification).fillna("International")
```

```
# 8️⃣ CUSTOMER VALUE SCORING
if "Net_Revenue" in df.columns and "Customer_Satisfaction" in df.columns:
    # Customer value score (combination of revenue and satisfaction)
    revenue_normalized = (df["Net_Revenue"] - df["Net_Revenue"].min()) /
(df["Net_Revenue"].max() - df["Net_Revenue"].min())
    satisfaction_normalized = (df["Customer_Satisfaction"] - 1) / 4  # Scale 1-5 to 0-1

    df["Customer_Value_Score"] = (revenue_normalized * 0.7 + satisfaction_normalized * 0.3) *
100
    df["Customer_Value_Score"] = df["Customer_Value_Score"].round(2)

    # Customer tier based on value score
    df["Customer_Tier"] = pd.cut(
        df["Customer_Value_Score"],
        bins=[0, 25, 50, 75, 100],
        labels=["Bronze", "Silver", "Gold", "Platinum"],
        right=False
    )

# 9️⃣ BUSINESS KPIs
if "Discount_Percentage" in df.columns:
    # Discount effectiveness
    df["Discount_Tier"] = pd.cut(
        df["Discount_Percentage"],
        bins=[0, 5, 15, 25, 100],
        labels=["No Discount", "Low Discount", "Medium Discount", "High Discount"],
        right=False
    )

# 🔟 PROFITABILITY INDICATORS
if "Net_Revenue" in df.columns:
    # Assuming a simple profit margin model (customize based on your business)
```

```
        df["Estimated_Profit"] = df["Net_Revenue"] * 0.2  # 20% margin assumption

        # Profit categories
        df["Profit_Category"] = pd.cut(
            df["Estimated_Profit"],
            bins=[0, 20, 60, 100, 200, float('inf')],
            labels=["Low Profit", "Medium Profit", "High Profit", "Very High Profit", "Exceptional Profit"],
            right=False
        )

    return df


def clean_dataset(df: pd.DataFrame) -> pd.DataFrame:
    """Primary cleaning pipeline aligned with Retail_Live_Project_Dataset schema."""

    df = df.copy()

    # 1 column name hygiene
    df.columns = df.columns.str.strip()

    # 2 text normalisation
    for col in TEXT_COLUMNS:
        if col in df.columns:
            df[col] = df[col].astype(str).str.strip()

    # 3 product category correction (vectorised for speed)
    mask_any = pd.Series(False, index=df.index)
    for key, cat in CATEGORY_MAPPINGS.items():
        key_mask = df["Product_Name"].str.lower().str.contains(key, na=False)
        df.loc[key_mask, "Product_Category"] = cat
```

```
    mask_any |= key_mask

# 4 numeric coercion
for col in NUMERIC_COLUMNS:
    if col in df.columns:
        df[col] = pd.to_numeric(df[col], errors="coerce")

# 5 date parsing
for col in DATE_COLUMNS:
    if col in df.columns:
        df[col] = pd.to_datetime(df[col], errors="coerce", dayfirst=False)

# 6 categorical harmonisation
if "Payment_Mode" in df.columns:
    df["Payment_Mode"] = (
        df["Payment_Mode"].str.lower().map(PAYMENT_MODE_MAP).fillna(df["Payment_Mode"])
    )
if "Delivery_Status" in df.columns:
    df["Delivery_Status"] = (
        df["Delivery_Status"].str.lower().map(DELIVERY_STATUS_MAP).fillna(df["Delivery_Status"])
    )
if "Gender" in df.columns:
    df["Gender"] = df["Gender"].str.lower().map(GENDER_MAP).fillna(df["Gender"])

# 7 email validation & formatting
if "Email" in df.columns:
    df["Email"] = df["Email"].str.lower().str.strip()
    df["Email_Valid"] = df["Email"].str.match(EMAIL_REGEX, na=False)

# 8 satisfaction clipping (1–5)
if "Customer_Satisfaction" in df.columns:
```

```
    df["Customer_Satisfaction"] = df["Customer_Satisfaction"].clip(1, 5)

    # 9️⃣ proper-case cities / countries
    for loc_col in ("City", "Country"):
        if loc_col in df.columns:
            df[loc_col] = df[loc_col].str.title()

    # 🔟 quality flags
    df["Data_Quality_Issues"] = ""
    if "Customer_Name" in df.columns:
        miss_name = df["Customer_Name"].eq("") | df["Customer_Name"].isna()
        df.loc[miss_name, "Data_Quality_Issues"] += "Missing Customer Name; "
    if "Email" in df.columns:
        df.loc[~df["Email_Valid"], "Data_Quality_Issues"] += "Invalid Email; "
    if "Purchase_Amount" in df.columns:
        bad_amt = df["Purchase_Amount"].le(0) | df["Purchase_Amount"].isna()
        df.loc[bad_amt, "Data_Quality_Issues"] += "Invalid Purchase Amount; "

    # 🆕 ADD CALCULATED COLUMNS FOR POWERBI
    df = add_calculated_columns(df)

    # Enhanced summary
    original_cols = len([col for col in df.columns if not col.startswith(('Net_', 'Discount_', 'Revenue_',
'Year', 'Month', 'Quarter', 'Season', 'Age_Group', 'Satisfaction_', 'NPS_', 'Delivery_', 'Order_',
'Payment_Type', 'Category_', 'Market_', 'Customer_', 'Profit_', 'Days_', 'Purchase_', 'Week_', 'Day_',
'Estimated_'))])
    new_cols = len(df.columns) - original_cols

    print(f"☑️ Cleaning & Enhancement Summary:")
    print(f"   • Total rows: {len(df)}")
    print(f"   • Original columns: {original_cols}")
    print(f"   • New calculated columns: {new_cols}")
```

```
    print(f"   • Records with data quality issues: {(df['Data_Quality_Issues'] != '').sum()}")
    print(f"   • PowerBI-ready columns added: {new_cols}")

    return df


def load_clean_validate(file_path: str) -> pd.DataFrame | None:
    """Convenience wrapper used by __main__."""
    try:
        df_raw = _read_file(file_path)
        df_clean = clean_dataset(df_raw)
        out_path = _write_file(df_clean, file_path)
        print(f"✔ Saved enhanced dataset → {out_path}")

        # Show new columns created
        new_columns = [col for col in df_clean.columns if col not in df_raw.columns]
        print(f"\n📊 New PowerBI columns created ({len(new_columns)}):")
        for col in sorted(new_columns):
            print(f"   • {col}")

        # Simple validation report
        print(f"\n📋 Dataset overview:")
        print(f"   • Shape: {df_clean.shape}")
        print(f"   • Date range: {df_clean['Purchase_Date'].min()} to {df_clean['Purchase_Date'].max()}")
        print(f"   • Revenue range: ${df_clean['Net_Revenue'].min():.2f} to
${df_clean['Net_Revenue'].max():.2f}")

        return df_clean
    except Exception as exc:
        print("❌", exc)
        return None
```

```
if __name__ == "__main__":
    import sys, pathlib

    # default to the live-project workbook in current directory
    default_file = "Retail_Live_Project_Dataset.xlsx"
    target = pathlib.Path(sys.argv[1]) if len(sys.argv) > 1 else pathlib.Path(default_file)

    if not target.exists():
        raise FileNotFoundError(f"File not found → {target.resolve()}")

    load_clean_validate(str(target))
```

Columns Code detail Breakdown

code is a comprehensive script designed to clean and enhance a retail dataset, specifically for use with Power BI dashboards. It takes an input file (Excel or CSV), applies various data cleaning and harmonization steps, and then adds numerous calculated columns to enrich the data for analytical purposes.

Here's a detailed breakdown of the columns added and how they are created:

Columns Added for Power BI Analysis

The `add_calculated_columns` function is responsible for creating a total of **21 new columns** in the dataset. These columns are derived from existing ones and are categorized for clarity:

1 Revenue Calculations (3 Columns)

These columns are created if 'Purchase_Amount' and 'Discount_Offered' are present.

- **`Net_Revenue`**:
  - **How it's created**: Subtracts `Discount_Offered` from `Purchase_Amount`.
  - **Purpose**: Represents the actual revenue generated after discounts, crucial for profitability analysis.
  - **Formula**: `Purchase_Amount - Discount_Offered`
- **`Discount_Percentage`**:
  - **How it's created**: Calculates the percentage of the discount relative to the `Purchase_Amount`, rounded to two decimal places.
  - **Purpose**: Helps understand the impact of discounts on sales and customer behavior.
  - **Formula**: `(Discount_Offered / Purchase_Amount) * 100`
- **`Revenue_Category`**:
  - **How it's created**: Categorizes `Net_Revenue` into predefined bins: "Low (0-100)", "Medium (100-300)", "High (300-500)", "Premium (500-1000)", and "Luxury (1000+)".
  - **Purpose**: Facilitates revenue segmentation and analysis by value tiers.
  - **Method**: `pd.cut()` on `Net_Revenue`.

2 Date-Based Calculations (8 Columns)

These columns are created if 'Purchase_Date' is present and are derived from the date information.

- **`Year`**:
  - **How it's created**: Extracts the year from `Purchase_Date`.
  - **Purpose**: Enables yearly trend analysis.
  - **Method**: `.dt.year` on `Purchase_Date`.
- **`Month`**:
  - **How it's created**: Extracts the numerical month from `Purchase_Date`.
  - **Purpose**: Useful for monthly sales and performance tracking.

    - **Method**: `.dt.month` on `Purchase_Date`.
- **`Month_Name`**:
    - **How it's created**: Extracts the full name of the month (e.g., "January") from `Purchase_Date`.
    - **Purpose**: Provides a more readable format for month-wise analysis in dashboards.
    - **Method**: `.dt.strftime("%B")` on `Purchase_Date`.
- **`Quarter`**:
    - **How it's created**: Extracts the quarter (1-4) from `Purchase_Date`.
    - **Purpose**: Facilitates quarterly reporting and performance review.
    - **Method**: `.dt.quarter` on `Purchase_Date`.
- **`Day_of_Week`**:
    - **How it's created**: Extracts the numerical day of the week (Monday=0, Sunday=6) from `Purchase_Date`.
    - **Purpose**: Helps identify daily sales patterns.
    - **Method**: `.dt.dayofweek` on `Purchase_Date`.
- **`Day_Name`**:
    - **How it's created**: Extracts the full name of the day of the week (e.g., "Monday") from `Purchase_Date`.
    - **Purpose**: Provides a readable format for day-of-week analysis.
    - **Method**: `.dt.strftime("%A")` on `Purchase_Date`.
- **`Week_of_Year`**:
    - **How it's created**: Extracts the ISO calendar week number from `Purchase_Date`.
    - **Purpose**: Allows for week-over-week performance tracking.
    - **Method**: `.dt.isocalendar().week` on `Purchase_Date`.
- **`Season`**:
    - **How it's created**: Maps the `Month` to a corresponding season (Winter, Spring, Summer, Fall).
    - **Purpose**: Enables seasonal sales analysis.
    - **Method**: Mapping `Month` using a dictionary.
- **`Days_Since_Purchase`**:
    - **How it's created**: Calculates the number of days between the `Purchase_Date` and the current date (when the script is run).
    - **Purpose**: Essential for recency analysis in RFM (Recency, Frequency, Monetary) modeling.
    - **Formula**: `(datetime.now() - Purchase_Date).dt.days`
- **`Purchase_Recency`**:
    - **How it's created**: Categorizes `Days_Since_Purchase` into recency bins: "Recent (0-30 days)", "Medium (30-90 days)", "Old (90-180 days)", "Very Old (180-365 days)", and "Stale (365+ days)".
    - **Purpose**: Groups customers based on how recently they made a purchase.
    - **Method**: `pd.cut()` on `Days_Since_Purchase`.

3 Customer Segmentation (1 Column)

This column is created if 'Age' is present.

- **`Age_Group`**:

    - **How it's created**: Segments `Age` into predefined groups: "18-25", "26-35", "36-45", "46-55", "56-65", and "65+".
    - **Purpose**: Allows for demographic analysis and targeted marketing.
    - **Method**: `pd.cut()` on `Age`.

4 Business Metrics (2 Columns)

These columns are created if 'Customer_Satisfaction' is present.

- **`Satisfaction_Category`**:
    - **How it's created**: Maps numerical `Customer_Satisfaction` ratings (1-5) to descriptive categories: "Very Dissatisfied", "Dissatisfied", "Neutral", "Satisfied", "Very Satisfied".
    - **Purpose**: Provides a more intuitive understanding of customer feedback.
    - **Method**: Mapping `Customer_Satisfaction` using a dictionary.
- **`NPS_Category`**:
    - **How it's created**: Classifies `Customer_Satisfaction` into NPS (Net Promoter Score) categories: "Detractor", "Passive", "Promoter".
    - **Purpose**: Enables calculation of NPS, a key customer loyalty metric.
    - **Method**: Mapping `Customer_Satisfaction` using a dictionary.

5 Delivery & Payment Analysis (3 Columns)

These columns are created if 'Delivery_Status' or 'Payment_Mode' are present.

- **`Delivery_Success`**:
    - **How it's created**: A binary flag (1 or 0) indicating if `Delivery_Status` is "Delivered".
    - **Purpose**: Helps track successful deliveries.
    - **Method**: Mapping `Delivery_Status` to 1 for "Delivered" and 0 for others.
- **`Order_Completed`**:
    - **How it's created**: A binary flag (1 or 0) indicating if `Delivery_Status` is "Delivered". (Similar to `Delivery_Success` but potentially for a broader "order completion" definition if other statuses were considered complete in different contexts).
    - **Purpose**: Tracks overall order fulfillment.
    - **Method**: `.isin(["Delivered"]).astype(int)` on `Delivery_Status`.
- **`Payment_Type`**:
    - **How it's created**: Categorizes `Payment_Mode` into "Digital" (UPI, Net Banking, Debit Card, Credit Card) or "Traditional" (Cash).
    - **Purpose**: Analyzes payment method trends and preferences.
    - **Method**: Mapping `Payment_Mode` using a dictionary.

6 Product Analysis (1 Column)

This column is created if 'Product_Category' is present.

- **`Category_Priority`**:

    - **How it's created**: Assigns a priority level (High, Medium, Low) to `Product_Category` based on predefined business importance.
    - **Purpose**: Helps in prioritizing product categories for inventory, marketing, or strategic focus.
    - **Method**: Mapping `Product_Category` using a dictionary.

7 Geographical Analysis (1 Column)

This column is created if 'Country' is present.

- **`Market_Type`**:
    - **How it's created**: Classifies `Country` as "Domestic" (India) or "International" (others), filling any missing values as "International".
    - **Purpose**: Facilitates analysis of domestic vs. international market performance.
    - **Method**: Mapping `Country` using a dictionary and `fillna`.

8 Customer Value Scoring (2 Columns)

These columns are created if 'Net_Revenue' and 'Customer_Satisfaction' are present.

- **`Customer_Value_Score`**:
    - **How it's created**: A weighted score combining normalized `Net_Revenue` (70%) and normalized `Customer_Satisfaction` (30%), scaled to 0-100 and rounded to two decimal places.
    - **Purpose**: Provides a single metric to rank customer value, incorporating both spending and satisfaction.
    - **Formula**: `(normalized_revenue * 0.7 + normalized_satisfaction * 0.3) * 100`
- **`Customer_Tier`**:
    - **How it's created**: Categorizes `Customer_Value_Score` into "Bronze", "Silver", "Gold", and "Platinum" tiers.
    - **Purpose**: Enables customer segmentation for loyalty programs or differentiated service.
    - **Method**: `pd.cut()` on `Customer_Value_Score`.

9 Business KPIs (1 Column)

This column is created if 'Discount_Percentage' is present.

- **`Discount_Tier`**:
    - **How it's created**: Categorizes `Discount_Percentage` into "No Discount", "Low Discount", "Medium Discount", and "High Discount" bins.
    - **Purpose**: Helps analyze the effectiveness and impact of different discount levels.
    - **Method**: `pd.cut()` on `Discount_Percentage`.

🔟 Profitability Indicators (2 Columns)

These columns are created if 'Net_Revenue' is present.

- **`Estimated_Profit`**:
  - **How it's created**: Calculates an estimated profit assuming a 20% margin on `Net_Revenue`. This is a placeholder and should be customized with actual business logic.
  - **Purpose**: Provides an initial estimate of profitability for each transaction.
  - **Formula**: `Net_Revenue * 0.2`
- **`Profit_Category`**:
  - **How it's created**: Categorizes `Estimated_Profit` into "Low Profit", "Medium Profit", "High Profit", "Very High Profit", and "Exceptional Profit" bins.
  - **Purpose**: Facilitates quick understanding of profit levels per transaction.
  - **Method**: `pd.cut()` on `Estimated_Profit`.

Overall Code Flow

The script follows a logical progression:

1. **Imports**: Necessary libraries like pandas (for data manipulation), numpy (for numerical operations), datetime (for date handling), and re (for regular expressions) are imported.
2. **Constants**: Defines various mappings and lists for column names, categories, and regex patterns to ensure consistency and reusability.
3. **File I/O (`_read_file`, `_write_file`)**:
   - `_read_file`: Auto-detects file type (CSV or Excel) based on extension and reads the data into a pandas DataFrame.
   - `_write_file`: Saves the processed DataFrame to a new file with `_cleaned` suffix, preserving the original format.
4. **`add_calculated_columns` Function**: This is where all the **21 new columns** detailed above are created based on existing data. It's designed to add valuable metrics and dimensions for Power BI dashboards. Each set of new columns is conditionally added, ensuring the code only attempts to create them if the source columns exist.
5. **`clean_dataset` Function**: This is the core cleaning pipeline.
   - **Column Name Hygiene**: Strips whitespace from column names.
   - **Text Normalization**: Strips whitespace from specified text columns.
   - **Product Category Correction**: Uses regex to map product names to harmonized categories (e.g., "laptop" -> "Electronics").
   - **Numeric Coercion**: Converts specified columns to numeric types, coercing errors to `NaN`.
   - **Date Parsing**: Converts specified columns to datetime objects.
   - **Categorical Harmonization**: Standardizes values in 'Payment_Mode', 'Delivery_Status', and 'Gender' columns using predefined mappings.
   - **Email Validation & Formatting**: Converts emails to lowercase, strips whitespace, and adds an `Email_Valid` boolean flag based on a regex pattern.
   - **Satisfaction Clipping**: Ensures `Customer_Satisfaction` values are within the 1-5 range.

- **Proper-Case Cities/Countries**: Converts 'City' and 'Country' names to title case.
- **Quality Flags**: Adds a `Data_Quality_Issues` column to flag missing names, invalid emails, or invalid purchase amounts.
- **Integration of `add_calculated_columns`**: Calls the `add_calculated_columns` function to append the new Power BI columns to the cleaned dataset.
- **Summary Print**: Outputs a summary of the cleaning and enhancement process, including the number of new columns and data quality issues.

6. **`load_clean_validate` Function**: This is a wrapper function that orchestrates the entire process:
   - Reads the raw data.
   - Cleans and enhances the data using `clean_dataset`.
   - Writes the cleaned data to a new file.
   - Prints a list of the newly created Power BI columns.
   - Provides a simple overview of the transformed dataset.
   - Handles potential errors during the process.
7. **Main Execution Block (`if __name__ == "__main__":`)**:
   - Determines the input file path: either from command-line arguments or defaults to "Retail_Live_Project_Dataset.xlsx".
   - Checks if the specified file exists.
   - Calls `load_clean_validate` to execute the data processing