Test Cases for Data Aggregation and Validation

These test cases focus on verifying the correctness of data loading, schema validation, and data integrity:

Test Cases for Reading Excel, CSV, and JSON Files

1. Test Case: Read Excel File

Test Case ID: TC_Excel_001

Description: Verify that the Excel file is read correctly.

Precondition: The Excel file ('dbfs:/mnt/raw/ADM_TEST/proj/Customer.xls',) is available in the specified location.

Steps:

1. read the file.

2. Check the number of rows and columns.

3. Verify that the column names match the expected names.

Expected Result:

- The DataFrame should contain the correct number of rows and columns.

- The column names should match the expected names.

2. Test Case: Read CSV File

Test Case ID: TC_CSV_001

Description: Verify that the CSV file is read correctly.

Precondition: The CSV file ('dbfs:/mnt/raw/ADM_TEST/proj/Order.csv') is available in the specified location.

Steps:

1. Use `spark.read.csv("'dbfs:/mnt/raw/ADM_TEST/proj/Order.csv'", header=True, inferSchema=True)` to read the file.

2. Check the number of rows and columns.

3. Verify that the data types of each column are as expected.

Expected Result:

- The DataFrame should contain the correct number of rows and columns.

- The data types should match the expected types.

3. Test Case: Read JSON File

Test Case ID: TC_JSON_001

Description: Verify that the JSON file is read correctly.

Precondition: The JSON file ('dbfs:/mnt/raw/ADM_TEST/proj/Shipping.json') is available in the specified location.

Steps:

1. Use `spark.read.json("'dbfs:/mnt/raw/ADM_TEST/proj/Shipping.json'")` to read the file.

2. Check the number of rows and columns.

3. Verify that the structure of the JSON data is reflected correctly in the DataFrame.

Expected Result:

- The DataFrame should contain the correct number of rows and columns.

- The data structure should match the expected schema.

4. Test Case: Verify Schema for Excel, CSV, and JSON

Test Case ID: TC_Schema_001

Description: Ensure that the schema of the loaded DataFrame matches the expected schema.

Precondition: Data files are available and expected schema is defined.

Steps:

1. Read the data from Excel, CSV, and JSON as previously described.

2. Define the expected schema using `StructType`.

3. Compare the DataFrame schema with the expected schema.

Expected Result:

- The DataFrame schema should match the expected schema.

5. Test Case: Verify Data Integrity for CSV File

Test Case ID: TC_CSV_002

Description: Check for missing values and duplicates in the CSV file.

Precondition: The CSV file is available.

Steps:

1. Read the CSV file as described above.

2. Count the number of missing values in each column.

3. Check for duplicate rows in the DataFrame.

Expected Result:

- There should be no missing values in required columns.

- The DataFrame should not contain any duplicate rows.

6. Test Case: Verify Data Types for Excel File

Test Case ID: TC_Excel_002

Description: Validate the data types of columns in the Excel file.

Precondition: The Excel file is available.

Steps:

1. Read the Excel file.

2. Check the data types of each column against expected data types.

Expected Result:

- The data types of the columns should match the expected types.

7. Test Case: Valid Names (No Special Characters)

Test Case ID: TC_Name_001
Description: Verify that valid names do not raise an error.
Precondition: DataFrame (df1) with valid names.

Steps1 : Pass the dataframe from all three file formats .

Step2 :Run the special character validation logic

```
def spec_check(df,column):
  if df.filter(regexp_extract(col(column),"[^a-zA-Z]",0)!='').count()!=0:
    print(f"Special characters present in the following columns: {column} Test Case Failed")
  else:
    print("Test Case Passed")
```

Expected Result:

-Output : ‘Test Case Passed’

8. Test Case: Empty Names

Test Case ID: TC_Name_001
Description: Verify that empty names do not raise an error, but can be handled as a separate validation case if needed.
Precondition: DataFrame (df1) .

```
def spec_check(df,column):

  if df.filter(col(column) == '').count() > 0:

    print(f"Null values present in the following columns: {column} Test Case Failed")

  else:

    print("Test Case Passed")
```

9. Total Amount Spent for Pending Delivery Status by Country

Test Case: Verify Total Amount for Pending Delivery Status by Country

- **Test Case Name**: TC_001_Pending_Delivery_Total_Amount
- **Description**: Check if the total amount is correctly calculated for orders with the status "Pending" for each country.
- **Input**: DataFrame with Country, Status, Amount.
- **Expected Output**: The correct sum of amounts for "Pending" status by country

Pass Df to below Function:

```
def test_pending_delivery_total_amount(df):

    pending_df = df.filter(df.Status == "Pending").groupBy("Country").sum("Amount")

    assert not pending_df.rdd.isEmpty(), "Pending delivery amount calculation failed."
```

10. Total Transactions, Quantity Sold, and Amount Spent for Each Customer

Test Case: Verify Total Transactions, Quantity, and Amount for Each Customer

- **Test Case Name**: TC_002_Total_Transactions_And_Amount_Per_Customer
- **Description**: Check if the total number of transactions, quantity, and total amount are correctly aggregated per customer.
- **Input**: DataFrame with Customer_ID, Order_ID, Amount.
- **Expected Output**: Correct count of transactions and total amount per customer.

Pass Df to below Function:

```
def test_total_transactions_amount_per_customer(df):

    customer_metrics_df = df.groupBy("Customer_ID").agg({"Order_ID": "count", "Amount": "sum"})

    assert not customer_metrics_df.rdd.isEmpty(), "Customer transactions and amount aggregation failed."
```

11. Maximum Product Purchased per Country

Test Case: Verify Maximum Product Purchased per Country

- **Test Case Name**: TC_003_Max_Product_Per_Country
- **Description**: Check if the most purchased product is correctly identified for each country.
- **Input**: DataFrame with Country, Item.
- **Expected Output**: The most purchased product for each country is identified.

Pass Df to below Function:

```
def test_max_product_per_country(df):

    max_product_df = df.groupBy("Country", "Item").count().groupBy("Country").max("count")

                    assert not max_product_df.rdd.isEmpty(), "Max product per country calculation failed."
```

12. Most Purchased Product Based on Age Category (<30 and >=30)

Test Case: Verify Most Purchased Product for Age Categories

- **Test Case Name**: TC_004_Most_Purchased_Product_By_Age
- **Description**: Verify that the most purchased product is identified for age groups less than 30 and 30 or above.
- **Input**: DataFrame with Age, Item.
- **Expected Output**: Most purchased product for both age categories.

Pass Df to below Function:

```
def test_most_purchased_product_by_age(df):
    age_lt_30_df = df.filter(df.Age < 30).groupBy("Item").count().orderBy("count",
            ascending=False)
    age_ge_30_df = df.filter(df.Age >= 30).groupBy("Item").count().orderBy("count",
            ascending=False)

    assert not age_lt_30_df.rdd.isEmpty(), "No products found for Age < 30."
    assert not age_ge_30_df.rdd.isEmpty(), "No products found for Age >= 30."
```

13. Country with Minimum Transactions and Sales Amount

Test Case: Verify Country with Minimum Transactions and Sales

- **Test Case Name**: TC_005_Min_Transactions_And_Sales_Country
- **Description**: Check if the country with the minimum number of transactions and sales amount is identified.
- **Input**: DataFrame with Country, Order_ID, Amount.

- **Expected Output**: The country with the minimum transactions and sales amount is identified.

Pass Df to below Function:

```
def test_country_with_min_transactions_and_sales(df):

    min_country_df = df.groupBy("Country").agg({"Order_ID": "count", "Amount": "sum"}).orderBy("count(Order_ID)", ascending=True)

    assert not min_country_df.rdd.isEmpty(), "Failed to identify country with minimum transactions and sales."
```