For loops

Managing and analyzing e-commerce transaction data often involves several stages, from cleaning inconsistencies to extracting meaningful insights. MATLAB provides powerful tools for these tasks, and I applied its capabilities to simulate a comprehensive workflow. Through this process, I explored various operations, from dataset cleaning to iterative computations, gaining a deeper understanding of handling complex data scenarios.**Generating and Preparing the Dataset**

I began by generating a simulated e-commerce dataset, comprising 1,000 rows and 8 columns, to mimic transaction data. Each column represented key features like customer details, product information, and sales metrics. To simulate real-world challenges, I introduced intentional data issues: missing values, duplicate rows, inconsistent formatting, and outliers. This deliberate step allowed me to test and refine my data cleaning techniques.

Cleaning and Normalizing Data

The first cleaning step involved removing duplicate rows to ensure that the dataset only contained unique transactions. I then addressed missing values in the `Total` column by replacing them with the median. This choice preserved the dataset's integrity without distorting the distribution, a common strategy in handling missing numerical data.

Next, I normalized the `Price` column to make values comparable across transactions. By standardizing the data, I ensured that variations in price were represented on a consistent scale, which is particularly useful for downstream analyses involving comparisons or clustering.**Outlier Removal and Inconsistent Formatting**

Outliers in the `Quantity` column were identified using a threshold of three standard deviations from the mean. These extreme values were removed to prevent skewed analyses and ensure reliable insights. Additionally, I addressed inconsistencies in the `Region` column by standardizing all entries to uppercase, making the data uniform and easier to group or filter.

Handling Invalid Data and Aggregation

Invalid entries, such as a negative `ProductID`, were filtered out to ensure the dataset represented realistic scenarios. To gain further insights, I aggregated total sales by region, summarizing performance across geographical areas. This step provided a macro-level view of the data, highlighting high-performing regions and potential areas for improvement.

Iterative Operations for Deeper Analysis

With a clean dataset in place, I explored various iterative operations to analyze and manipulate the data. For instance, I used loops to calculate daily sales totals and product-level revenues. These iterations broke down the dataset into manageable components, allowing me to examine trends and performance metrics in detail.

I also experimented with nested loops to analyze relationships between products and days, generating detailed revenue reports. This granular approach offered valuable insights, such as identifying peak sales periods for specific products.**Creative Applications of Loops**

In addition to dataset analysis, I used MATLAB's loop structures for other practical applications. For instance, I leveraged a Fibonacci-like sequence to forecast stock levels over multiple periods, demonstrating how mathematical models can support inventory planning. I also iterated over index values to modify elements within vectors, a fundamental operation in MATLAB that highlights its flexibility in handling data structures.

Exploring Combinations and Patterns

Another interesting exercise involved combining characters and numbers using nested loops, which generated all possible combinations within defined limits. This task reinforced the utility of MATLAB for exploring permutations and patterns, a skill applicable in tasks like generating unique identifiers or modeling test cases.

Observations and Reflections

This project provided a comprehensive overview of MATLAB's capabilities for data cleaning and iterative analysis. Key observations included:

Importance of Cleaning: Addressing missing values, inconsistencies, and outliers is critical for ensuring data quality and reliability.

Iterative Operations: Loops offer immense flexibility for breaking down and analyzing datasets at different levels of granularity.

Creative Applications: From forecasting stock levels to generating combinations, loops extend MATLAB's utility beyond basic analysis.

Efficiency and Scalability: MATLAB's tools and functions are well-suited for handling large datasets efficiently, making it a valuable resource for data-intensive tasks.

```
%% Step 1: Load the Dataset
% I started by creating a dataset that mimics an e-commerce transaction dataset.
rows = 1000; % I chose 1000 rows to simulate a moderately large dataset for analysis.
columns = 8; % I decided on 8 columns to capture various transaction-related features.
data = array2table(randi([1, 500], rows, columns), ... % Generating random numbers to populate the dataset.
    'VariableNames', {'CustomerID', 'ProductID', 'Quantity', 'Price', 'Total', 'PurchaseDate', 'Region', 'Category'}); % Assigning meaning

% Fixed the data issue by ensuring 'Region' is a cell array to store strings.
data.Region = repmat({'Unknown'}, rows, 1); % Initialized the 'Region' column as a cell array with default values.

% I introduced a few data issues intentionally to simulate real-world cleaning challenges.
data.Total(10:20) = NaN; % Adding NaN values to the 'Total' column to mimic missing entries.
data.CustomerID(1:5) = 1; % Adding duplicate CustomerIDs to simulate duplicate rows.
data.Region(1:15) = {'na'}; % Introducing inconsistent case formatting in the 'Region' column.
data.ProductID(25) = -999; % Adding an invalid ProductID to test handling incorrect data.
data.Quantity(100) = 99999; % Creating an extreme outlier in 'Quantity' to test outlier detection.

%% Step 2: Remove Duplicate Records
% I used the `unique` function to remove duplicate rows. This step ensures only unique rows are kept.
data = unique(data, 'rows'); % Filtering out duplicate rows based on all columns.

%% Step 3: Handle Missing Values
% Since there were missing values in the 'Total' column, I decided to replace them with the median.
medianTotal = median(data.Total, 'omitnan'); % Calculating the median, ignoring NaN values.
data.Total(isnan(data.Total)) = medianTotal; % Filling missing values with the median.

%% Step 4: Normalize the Data
% I normalized the 'Price' column to make it easier to compare values across transactions.
meanPrice = mean(data.Price); % Calculating the mean of the 'Price' column.
stdPrice = std(data.Price); % Calculating the standard deviation of the 'Price' column.
data.Price = (data.Price - meanPrice) / stdPrice; % Applying normalization.

%% Step 5: Remove Outliers
% To identify and remove outliers in 'Quantity', I used a threshold of three standard deviations.
meanQuantity = mean(data.Quantity); % Finding the mean of 'Quantity'.
stdQuantity = std(data.Quantity); % Finding the standard deviation of 'Quantity'.
outliers = abs(data.Quantity - meanQuantity) > 3 * stdQuantity; % Identifying rows with extreme values.
data(outliers, :) = []; % Removing rows with extreme 'Quantity' values.

%% Step 6: Convert Data Types
% I converted the 'PurchaseDate' column from numeric to datetime for better time-based analysis.
data.PurchaseDate = datetime(data.PurchaseDate, 'ConvertFrom', 'posixtime'); % Converting to datetime format.

%% Step 7: Address Inconsistent Data Entry
% I standardized the 'Region' column by converting all text to uppercase.
data.Region = cellfun(@upper, data.Region, 'UniformOutput', false); % Ensuring consistent case for region values.

%% Step 8: Remove Invalid or Incorrect Data
% I removed rows with an invalid 'ProductID' value of -999.
data(data.ProductID == -999, :) = []; % Filtering out rows with invalid 'ProductID'.

%% Step 9: Aggregate Data
% I summarized total sales by region to analyze regional performance.
aggregatedData = varfun(@sum, data, 'InputVariables', 'Total', 'GroupingVariables', 'Region'); % Aggregating sales data.

% Displaying a snapshot of the cleaned dataset.
disp('Cleaned Dataset Snapshot:');
```

```
Cleaned Dataset Snapshot:
```

```
disp(head(data, 10)); % Showing the first 10 rows.
```

```
    CustomerID    ProductID    Quantity     Price      Total       PurchaseDate          Region       Category
    __________    _________    ________    ________    _____    ____________________    ___________    ________

        1             30           65       0.93012      10     01-Jan-1970 00:06:26    {'NA'     }      206
        1            173          384      0.050446     251     01-Jan-1970 00:07:55    {'NA'     }      153
        1            262          243       -1.0796      82     01-Jan-1970 00:07:00    {'NA'     }      141
        1            407          337       -1.0728       1     01-Jan-1970 00:03:14    {'NA'     }      427
        1            415           62      -0.67359     459     01-Jan-1970 00:04:31    {'NA'     }      380
        2            441          366      -0.99163     237     01-Jan-1970 00:03:17    {'UNKNOWN'}       47
        2            446          278        1.5459      14     01-Jan-1970 00:02:05    {'UNKNOWN'}      159
        3            244           16       -1.1405     270     01-Jan-1970 00:00:57    {'UNKNOWN'}       27
        4             37          145       -1.5262     159     01-Jan-1970 00:07:20    {'UNKNOWN'}      367
        4            341          142       0.86922     415     01-Jan-1970 00:03:11    {'UNKNOWN'}      347
```

```
%% Step 10: For Loops

% Step 10.1: Iterate Over Columns of a Matrix
% I created a simple 2x3 matrix and used a loop to iterate over its columns.
some_matrix = [1, 2, 3; 4, 5, 6]; % Defining a small matrix.
```

```
for some_column = some_matrix
    disp(some_column); % Displaying each column separately.
end
```

```
     1
     4

     2
     5

     3
     6
```

```
% Step 10.2: Same Counter Nested Loops
% I tested using the same counter variable in nested loops to understand how MATLAB handles it.
for x = 1:3 % Outer loop runs three times.
    for x = 1:3 % Inner loop also uses 'x' and runs three times per outer loop iteration.
        fprintf('%d,', x); % Printing the value of the inner loop variable.
    end
    fprintf('\n'); % Adding a newline after each outer loop iteration.
end
```

```
1,2,3,1,2,3,1,2,3,
```

```
% Step 10.3: Iterate Over Elements of a Vector
% I used a loop to iterate over elements of a vector to display them one by one.
my_vector = [4, 3, 5, 1, 2]; % A simple row vector.
for element = my_vector
    disp(element); % Displaying each element during iteration.
end
```

```
     4

     3

     5

     1

     2
```

```
% Step 10.4: Nested Loops
% I combined characters and numbers using nested loops to generate all combinations.
chars = 'abc'; % A string of characters.
n = 3; % A numeric limit for combinations.
for char = chars
    for k = 1:n
        disp([char, num2str(k)]); % Concatenating a character with a number.
    end
end
```

```
a1
a2
a3
b1
b2
b3
c1
c2
c3
```

```
% Step 10.5: Fibonacci Series with Nested Loops
% I calculated and displayed every nth Fibonacci number using nested loops.
N = 10; % Number of Fibonacci numbers to calculate.
n = 3; % Interval for displaying numbers.
a1 = 0; % Starting value for the Fibonacci sequence.
a2 = 1; % Second value in the sequence.
for j = 1:N
    for k = 1:n
        an = a1 + a2; % Computing the next number in the sequence.
        a1 = a2; % Updating the first number.
        a2 = an; % Updating the second number.
    end
    disp(an); % Displaying the nth Fibonacci number.
end
```

```
     3

    13

    55
```

```
   233

   987

        4181

       17711

       75025

      317811
```

```
% Step 10.6: Loop Over Indexes
% I incremented each element in a vector by 1 using a loop.
my_vector = [0, 2, 1, 3, 9]; % Original vector.
for i = 1:numel(my_vector)
    my_vector(i) = my_vector(i) + 1; % Adding 1 to each element.
end
disp('Modified Vector:'); % Displaying the final vector.
```

```
Modified Vector:
```

```
disp(my_vector);
```

```
     1     3     2     4    10
```