Functional Data Processing Pipeline Documentation

Functional Data Processing Pipeline

1- Handle missing data (fill with defaults or remove).
   - Fill missing numeric values with the median.
   - Replace missing categorical values with the mode.

2- Standardize formats (dates, numerical precision).
   - (Price Per Unit, Total Spent) to all as float
   - Error OR Unknown
     i. replace with Mode or median

3- Data Transformation:
   - Filter rows based on conditions (eg. Item == Coffee).
   - **Compute New Columns:** Create a new column called Corrected Total by calculating Quantity * Price Per Unit to see if the original Total Spent matches (data validation).
   - **Aggregate Data:** Group by eg (Item = “Coffee") to calculate Total Spent for Coffee.

4- Data Analysis Perform operations such as:
   - Statistical summaries (mean, median, variance). ⬅️

5- Output Results:
   - Save processed data to files a clean CSV and display summaries to the console.

Data Visualization (Optional): Produce charts such as bar graphs or line charts

Cafe Sales - Dirty Data for Cleaning Training

https://www.kaggle.com/datasets/ahmedmohamed2003/cafe-sales-dirty-data-for-cleaning-training

Concepts we used in code

1) Higher Order Functions

1.1) Functional Example:

map, **filter**, and **reduce** are used extensively.

```python
# Aggregate Data (Data Transformation)
def get_aggregate_by_coloumn_for_item(data, item_name, column_name, operation):
    item_data_iter = filter(lambda row: filter_by_column_and_value(row, 'Item', item_name), data)
    final_item_list = list(item_data_iter)
    corected_totals_item = get_column(final_item_list, column_name, [])
    return reduce(operation, corected_totals_item, 0.0)

# Data Analysis & Statistical Summaries
def print_numeric_analysis(final_data, column_name, label):
    # Use map() to extract column efficiently (Faster than recursive get_column)
    values = list(map(lambda row: row[column_name], final_data))
    print(f"\n--- Analysis: {label} ---")
    print(f"Mean:     {statistics.mean(values):.2f}")
    print(f"Median:   {statistics.median(values):.2f}")
    print(f"Variance: {statistics.variance(values):.2f}")
```

1.2) Imperative Example:

Uses **explicit loops** instead of higher order functions.

```python
def clean_data_imperative(raw_data, defults):
    """
    Cleans and transforms data by modifying the list of dictionaries directly (in-place modification
    or building a new list with explicit loops), which is a characteristic of imperative style.
    """
    cleaned_data = [] # We build a new list to avoid modifying the input list in-place

    # Explicit loop replacing the map() function
    for row in raw_data:
        # Apply cleaning and parsing for each field
        quantity = parse_int(row['Quantity'], defults['defult_quantity_median'])
        price_per_unit = parse_float(row['Price Per Unit'], defults['defult_price_per_unit_mean'])

        # Compute New Columns
        corrected_total = quantity * price_per_unit

        # Build the new, cleaned row dictionary
        cleaned_row = {
            **row,
            'Item': parse_string(row['Item'], defults['defult_item_mode']),
            'Quantity': quantity,
            'Price Per Unit': price_per_unit,
            'Total Spent': parse_float(row['Total Spent'], 0.0), # keep 0.0 as it will be recomputed (0.0 is safe default)
            'Payment Method': parse_string(row['Payment Method'], defults['defult_payment_method_mode']),
            'Location': parse_string(row['Location'], defults['defult_location_mode']),
            'Transaction Date': parse_date(row['Transaction Date'], defults['defult_transaction_date_mode']),
            'Corrected Total': round(float(corrected_total), DECIMAL_PLACES)
        }
        cleaned_data.append(cleaned_row)

    return cleaned_data
```

1.3) Comparison:

1. - Functional code is more concise and expressive for data transformations.
2. - Imperative code is more explicit and easier to debug for beginners.

2) Tail Recursion

2.1) Functional Example:

```python

def get_column(data, column_name, accumulator):        Anas Alamir, 5 days ago • first commit
    match data:
        case []:
            return accumulator
        case [head, *tail]:
            match head[column_name]:
                case "ERROR" | "UNKNOWN" | "":
                    return get_column(tail, column_name, accumulator)
                case _:
                    return get_column(tail, column_name, accumulator + [head[column_name]])

```

2.2) Imperative Example:

Uses **loops** instead Tail Recursion.

```python
    # Explicit loop replacing the map() function
    for row in raw_data:
        # Apply cleaning and parsing for each field
        quantity = parse_int(row['Quantity'], defults['defult_quantity_median'])
        price_per_unit = parse_float(row['Price Per Unit'], defults['defult_price_per_unit_mean

        # Compute New Columns
        corrected_total = quantity * price_per_unit

```

2.3) Comparison:

1. Functional code can leverage recursion for list processing, but may hit recursion limits in Python.
2. Imperative code avoids recursion, using loops for better performance in Python.

3) Single Assignment (Pure Functions)

3.1) Functional Example:

Functions like **clean_row** do not modify input data, but return new data

```
def clean_row(row, defults):        Anas Alamir, 5 days ago • first commit
    quantity = parse_int(row['Quantity'], defults['defult_quantity_median'])
    price_per_unit = parse_float(row['Price Per Unit'], defults['defult_price_per_unit_mean'])
    # Compute New Columns
    corrected_total = quantity * price_per_unit
    return {
        **row,
        'Item': parse_string(row['Item'], defults['defult_item_mode']),
        'Quantity': quantity,
        'Price Per Unit': price_per_unit,
        'Total Spent': parse_float(row['Total Spent'], 0.0), # keep 0.0 as it will be recomputed
        'Payment Method': parse_string(row['Payment Method'], defults['defult_payment_method_mode']),
        'Location': parse_string(row['Location'], defults['defult_location_mode']),
        'Transaction Date': parse_date(row['Transaction Date'], defults['defult_transaction_date_mode']),
        'Corrected Total': round(float(corrected_total), DECIMAL_PLACES)
    }
```

3.2) Imperative Example:

May modify or build new lists, but often uses in-place updates.

```
def clean_data_imperative(raw_data, defults):
    """
    Cleans and transforms data by modifying the list of dictionaries directly (in-place modification
    or building a new list with explicit loops), which is a characteristic of imperative style.
    """
    cleaned_data = [] # We build a new list to avoid modifying the input list in-place

    # Explicit loop replacing the map() function
    for row in raw_data:
        # Apply cleaning and parsing for each field
        quantity = parse_int(row['Quantity'], defults['defult_quantity_median'])
        price_per_unit = parse_float(row['Price Per Unit'], defults['defult_price_per_unit_mean'])

        # Compute New Columns
        corrected_total = quantity * price_per_unit

        # Build the new, cleaned row dictionary
        cleaned_row = {
            **row,
            'Item': parse_string(row['Item'], defults['defult_item_mode']),
            'Quantity': quantity,
            'Price Per Unit': price_per_unit,
            'Total Spent': parse_float(row['Total Spent'], 0.0), # keep 0.0 as it will be recomputed (0.0 is safe default)
            'Payment Method': parse_string(row['Payment Method'], defults['defult_payment_method_mode']),
            'Location': parse_string(row['Location'], defults['defult_location_mode']),
            'Transaction Date': parse_date(row['Transaction Date'], defults['defult_transaction_date_mode']),
            'Corrected Total': round(float(corrected_total), DECIMAL_PLACES)
        }
        cleaned_data.append(cleaned_row)

    return cleaned_data
```

3.3) Comparison:

1. Functional code encourages immutability and pure functions.
2. Imperative code may use mutable data structures and side effects.

4) Lists

4.1) Functional Example:

Uses list comprehensions, map, and recursion

```
# Compute Defults
list_quantity_defaults = get_column(raw_data, 'Quantity', [])
quantity_values_defaults = list(map(lambda x: parse_int(x, 0), list_quantity_defaults))
quantity_median_defaults = statistics.median(quantity_values_defaults)

list_price_per_unit_defaults = get_column(raw_data, 'Price Per Unit', [])
price_per_unit_values_defaults = list(map(lambda x: parse_float(x, 0.0), list_price_per_unit_defaults))
price_per_unit_mean_defaults = statistics.mean(price_per_unit_values_defaults)
```

4.2) Imperative Example:

Uses explicit loops to build lists:

```
def print_numeric_analysis(data, column_name, label):
    """Calculates and prints numeric stats using an imperative loop for data extraction."""
    values = []      Anas Alamir, 4 days ago • Mahmoud Elhefnawy imparative code
    # Explicit loop replacing the map() function
    for row in data:
        # We assume the data is cleaned and the column value is a number
        values.append(row[column_name])

```

4.3) Comparison:

1. Both paradigms use lists, but functional code prefers declarative transformations, while imperative code uses explicit iteration.

5) Mutability Vs Immutability

5.1) Functional Example:

immutability ideas accumulator, returning new data

```python

def get_column(data, column_name, accumulator):
    match data:
        case []:
            return accumulator
        case [head, *tail]:
            match head[column_name]:
                case "ERROR" | "UNKNOWN" | "":
                    return get_column(tail, column_name, accumulator)
                case _:
                    return get_column(tail, column_name, accumulator + [head[column_name]])

def clean_row(row, defults):
    quantity = parse_int(row['Quantity'], defults['defult_quantity_median'])
    price_per_unit = parse_float(row['Price Per Unit'], defults['defult_price_per_unit_mean'])
    # Compute New Columns
    corrected_total = quantity * price_per_unit
    return {
        **row,
        'Item': parse_string(row['Item'], defults['defult_item_mode']),
        'Quantity': quantity,
        'Price Per Unit': price_per_unit,     Anas Alamir, 5 days ago • first commit
        'Total Spent': parse_float(row['Total Spent'], 0.0), # keep 0.0 as it will be recomputed
        'Payment Method': parse_string(row['Payment Method'], defults['defult_payment_method_mode']),
        'Location': parse_string(row['Location'], defults['defult_location_mode']),
        'Transaction Date': parse_date(row['Transaction Date'], defults['defult_transaction_date_mode']),
        'Corrected Total': round(float(corrected_total), DECIMAL_PLACES)
    }
```

5.2) Imperative Example:

Mutability: quantities, prices_per_unit are mutated in loops

```python
def compute_column_stats(data):
    """
    Computes statistical defaults (median, mean, mode) for cleaning in an imperative style.
    This replaces the recursive get_column and subsequent map/statistics calls in the main.
    """
    # 1. Collect all non-dirty values for required columns in lists
    quantities = []
    prices_per_unit = []
    items = []
    payment_methods = []
    locations = []
    transaction_dates = []

    for row in data:
        # Quantity (Need to parse to int first)
        if row['Quantity'] not in ["ERROR", "UNKNOWN", ""]:
            try:
                quantities.append(int(row['Quantity']))
            except ValueError:
                pass # Skip unparseable values

        # Price Per Unit (Need to parse to float first)
        if row['Price Per Unit'] not in ["ERROR", "UNKNOWN", ""]:
            try:
                prices_per_unit.append(float(row['Price Per Unit']))
            except ValueError:
                pass # Skip unparseable values

```

5.3) Comparison:

1. **Mutable objects** can be changed in place; **immutable objects** cannot be changed after creation (a new object is created on “change”).

6) Conclusion

- **Functional programming** offers concise, expressive code with a focus on immutability and pure functions, but may be less familiar to Python programmers and can hit recursion limits.

- **Imperative programming** is more explicit, easier to debug, and better suited for Python's performance characteristics, but can be more verbose and prone to side effects.

Choosing a paradigm depends on the problem, team familiarity, and language features. Both styles are valuable and can be mixed for practical software development.