

# 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.

#### 1.2)

```
110
111 # Aggregate Data (Data Transformation)
112 def get_aggregate_by_column_for_item(data, item_name, column_name, operation):
113     item_data_iter = filter(lambda row: filter_by_column_and_value(row, 'Item', item_name), data)
114     final_item_list = list(item_data_iter)
115     corrected_totals_item = get_column(final_item_list, column_name, [])
116     return reduce(operation, corrected_totals_item, 0.0)
117
118 # Data Analysis & Statistical Summaries
119 def print_numeric_analysis(final_data, column_name, label):
120     # Use map() to extract column efficiently (Faster than recursive get_column)
121     values = list(map(lambda row: row[column_name], final_data))
122     print(f"\n--- Analysis: {label} ---")
123     print(f"Mean: {statistics.mean(values):.2f}")
124     print(f"Median: {statistics.median(values):.2f}")
125     print(f"Variance: {statistics.variance(values):.2f}")
126
```

#### Imperative Example:

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

```
142 def clean_data_imperative(raw_data, defaults):
143     """
144     Cleans and transforms data by modifying the list of dictionaries directly (in-place modification
145     or building a new list with explicit loops), which is a characteristic of imperative style.
146     """
147     cleaned_data = [] # We build a new list to avoid modifying the input list in-place
148
149     # Explicit loop replacing the map() function
150     for row in raw_data:
151         # Apply cleaning and parsing for each field
152         quantity = parse_int(row['Quantity'], defaults['default_quantity_median'])
153         price_per_unit = parse_float(row['Price Per Unit'], defaults['default_price_per_unit_mean'])
154
155         # Compute New Columns
156         corrected_total = quantity * price_per_unit
157
158         # Build the new, cleaned row dictionary
159         cleaned_row = {
160             **row,
161             'Item': parse_string(row['Item'], defaults['default_item_mode']),
162             'Quantity': quantity,
163             'Price Per Unit': price_per_unit,
164             'Total Spent': parse_float(row['Total Spent'], 0.0), # keep 0.0 as it will be recomputed (0.0 is safe default)
165             'Payment Method': parse_string(row['Payment Method'], defaults['default_payment_method_mode']),
166             'Location': parse_string(row['Location'], defaults['default_location_mode']),
167             'Transaction Date': parse_date(row['Transaction Date'], defaults['default_transaction_date_mode']),
168             'Corrected Total': round(float(corrected_total), DECIMAL_PLACES)
169         }
170         cleaned_data.append(cleaned_row)
171
172     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:

```
78
79 def get_column(data, column_name, accumulator):      Anas Alamir, 5 days ago • first commit
80     match data:
81         case []:
82             return accumulator
83         case [head, *tail]:
84             match head[column_name]:
85                 case "ERROR" | "UNKNOWN" | "":
86                     return get_column(tail, column_name, accumulator)
87                 case _:
88                     return get_column(tail, column_name, accumulator + [head[column_name]])
89
```

### 2.2) Imperative Example:

Uses **loops** instead Tail Recursion.

```
149     # Explicit loop replacing the map() function
150     for row in raw_data:
151         # Apply cleaning and parsing for each field
152         quantity = parse_int(row['Quantity'], defaults['default_quantity_median'])
153         price_per_unit = parse_float(row['Price Per Unit'], defaults['default_price_per_unit_mean'])
154
155         # Compute New Columns
156         corrected_total = quantity * price_per_unit
157
```

### 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

```
90 def clean_row(row, defaults):
91     quantity = parse_int(row['Quantity'], defaults['default_quantity_median'])
92     price_per_unit = parse_float(row['Price Per Unit'], defaults['default_price_per_unit_mean'])
93     # Compute New Columns
94     corrected_total = quantity * price_per_unit
95     return {
96         **row,
97         'Item': parse_string(row['Item'], defaults['default_item_mode']),
98         'Quantity': quantity,
99         'Price Per Unit': price_per_unit,
100         'Total Spent': parse_float(row['Total Spent'], 0.0), # keep 0.0 as it will be recomputed
101         'Payment Method': parse_string(row['Payment Method'], defaults['default_payment_method_mode']),
102         'Location': parse_string(row['Location'], defaults['default_location_mode']),
103         'Transaction Date': parse_date(row['Transaction Date'], defaults['default_transaction_date_mode']),
104         'Corrected Total': round(float(corrected_total), DECIMAL_PLACES)
105     }
106
```

#### 3.2) Imperative Example:

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

```
141
142 def clean_data_imperative(raw_data, defaults):
143     """
144     Cleans and transforms data by modifying the list of dictionaries directly (in-place modification
145     or building a new list with explicit loops), which is a characteristic of imperative style.
146     """
147     cleaned_data = [] # We build a new list to avoid modifying the input list in-place
148
149     # Explicit loop replacing the map() function
150     for row in raw_data:
151         # Apply cleaning and parsing for each field
152         quantity = parse_int(row['Quantity'], defaults['default_quantity_median'])
153         price_per_unit = parse_float(row['Price Per Unit'], defaults['default_price_per_unit_mean'])
154
155         # Compute New Columns
156         corrected_total = quantity * price_per_unit
157
158         # Build the new, cleaned row dictionary
159         cleaned_row = {
160             **row,
161             'Item': parse_string(row['Item'], defaults['default_item_mode']),
162             'Quantity': quantity,
163             'Price Per Unit': price_per_unit,
164             'Total Spent': parse_float(row['Total Spent'], 0.0), # keep 0.0 as it will be recomputed (0.0 is safe default)
165             'Payment Method': parse_string(row['Payment Method'], defaults['default_payment_method_mode']),
166             'Location': parse_string(row['Location'], defaults['default_location_mode']),
167             'Transaction Date': parse_date(row['Transaction Date'], defaults['default_transaction_date_mode']),
168             'Corrected Total': round(float(corrected_total), DECIMAL_PLACES)
169         }
170         cleaned_data.append(cleaned_row)
171
172     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:

```
78
79 def get_column(data, column_name, accumulator):    Anas Alamir, 5 days ago • first commit
80     match data:
81         case []:
82             return accumulator
83         case [head, *tail]:
84             match head[column_name]:
85                 case "ERROR" | "UNKNOWN" | "":
86                     return get_column(tail, column_name, accumulator)
87                 case _:
88                     return get_column(tail, column_name, accumulator + [head[column_name]])
89
```

```
182
183 # Compute Defaults
184 list_quantity_defaults = get_column(raw_data, 'Quantity', [])
185 quantity_values_defaults = list(map(lambda x: parse_int(x, 0), list_quantity_defaults))
186 quantity_median_defaults = statistics.median(quantity_values_defaults)
187
188 list_price_per_unit_defaults = get_column(raw_data, 'Price Per Unit', [])
189 price_per_unit_values_defaults = list(map(lambda x: parse_float(x, 0.0), list_price_per_unit_defaults))
190 price_per_unit_mean_defaults = statistics.mean(price_per_unit_values_defaults)
191
```

### 4.2) Imperative Example:

Uses explicit loops to build lists:

```
190 def print_numeric_analysis(data, column_name, label):
191     """Calculates and prints numeric stats using an imperative loop for data extraction."""
192     values = []    Anas Alamir, 4 days ago • Mahmoud Elhefnawy imperative code
193     # Explicit loop replacing the map() function
194     for row in data:
195         # We assume the data is cleaned and the column value is a number
196         values.append(row[column_name])
197
```

### 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

```
78
79 def get_column(data, column_name, accumulator):
80     match data:
81         case []:
82             return accumulator
83         case [head, *tail]:
84             match head[column_name]:
85                 case "ERROR" | "UNKNOWN" | "":
86                     return get_column(tail, column_name, accumulator)
87                 case _:
88                     return get_column(tail, column_name, accumulator + [head[column_name]])
89
90 def clean_row(row, defaults):
91     quantity = parse_int(row['Quantity'], defaults['default_quantity_median'])
92     price_per_unit = parse_float(row['Price Per Unit'], defaults['default_price_per_unit_mean'])
93     # Compute New Columns
94     corrected_total = quantity * price_per_unit
95     return [
96         **row,
97         'Item': parse_string(row['Item'], defaults['default_item_mode']),
98         'Quantity': quantity,
99         'Price Per Unit': price_per_unit,
100         'Total Spent': parse_float(row['Total Spent'], 0.0), # keep 0.0 as it will be recomputed
101         'Payment Method': parse_string(row['Payment Method'], defaults['default_payment_method_mode']),
102         'Location': parse_string(row['Location'], defaults['default_location_mode']),
103         'Transaction Date': parse_date(row['Transaction Date'], defaults['default_transaction_date_mode']),
104         'Corrected Total': round(float(corrected_total), DECIMAL_PLACES)
105     ]
```

### 5.2) Imperative Example:

Mutability: quantities, prices\_per\_unit are mutated in loops

```
92 def compute_column_stats(data):
93     """
94     Computes statistical defaults (median, mean, mode) for cleaning in an imperative style.
95     This replaces the recursive get_column and subsequent map/statistics calls in the main.
96     """
97     # 1. Collect all non-dirty values for required columns in lists
98     quantities = []
99     prices_per_unit = []
100     items = []
101     payment_methods = []
102     locations = []
103     transaction_dates = []
104
105     for row in data:
106         # Quantity (Need to parse to int first)
107         if row['Quantity'] not in ["ERROR", "UNKNOWN", ""]:
108             try:
109                 quantities.append(int(row['Quantity']))
110             except ValueError:
111                 pass # Skip unparseable values
112
113         # Price Per Unit (Need to parse to float first)
114         if row['Price Per Unit'] not in ["ERROR", "UNKNOWN", ""]:
115             try:
116                 prices_per_unit.append(float(row['Price Per Unit']))
117             except ValueError:
118                 pass # Skip unparseable values
119
```

### 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.