Part 1: Data Preprocessing Using Core Python

In this step, we clean the insurance claim dataset using basic Python (no external libraries like pandas/numpy).

Key Steps:

- Read the CSV line-by-line using built-in `open()` and `readlines()`.
- Strip and split each row, and check for:
  - Correct number of columns
  - Presence of required fields
  - Valid numeric data (amounts)
- Normalize text data (city names, rejection remarks)
- Return a cleaned dataset as a list of lists

In [19]:
```python
def preprocess_csv(file_path):
    """
    Preprocess the CSV data using only core Python.
    Tasks:
    - Remove rows with missing/invalid data.
    - Convert numeric fields.
    - Standardize city names.
    - Validate mandatory fields.
    """
    cleaned_data = []
    with open(file_path, 'r', encoding='utf-8') as f:
        lines = f.readlines()

    header = lines[0].strip().split(',')
    cleaned_data.append(header)

    for line in lines[1:]:
        row = line.strip().split(',')

        if len(row) != len(header):
            continue  # skip rows with column mismatch

        claim_id, claim_date, customer_id, claim_amount, premium_collected, paid_amount, city, rejection_remarks = r

        # Validate required fields
        if not all([claim_id, claim_date, customer_id, claim_amount, premium_collected, paid_amount, city]):
            continue

        try:
            claim_amount = float(claim_amount)
            premium_collected = float(premium_collected)
            paid_amount = float(paid_amount)
        except ValueError:
            continue

        city = city.strip().title()  # normalize city name
        rejection_remarks = rejection_remarks.strip() if rejection_remarks else ""

        cleaned_data.append([
            claim_id.strip(), claim_date.strip(), customer_id.strip(),
            claim_amount, premium_collected, paid_amount, city, rejection_remarks
        ])

    return cleaned_data
```

Part 2: City Analysis for Shutdown Recommendation

XYZ TECH wants to shut down operations in one of the four cities: **Pune, Kolkata, Ranchi, or Guwahati**.

Objective:

Identify the city that performs worst in terms of:

- Least number of claims
- Least total premium collected
- Highest rejection rate (used as tiebreaker)

Steps:

- Loop through cleaned data and calculate:
  - Total claims per city
  - Total premium collected
  - Number of rejected claims (i.e., `PAID_AMOUNT == 0`)
- Print stats per city
- Return the city that is least beneficial to continue

In [20]:
```python
def analyze_city_for_shutdown(cleaned_data):
    """
    Recommend a city for shutdown based on:
    - Least number of claims
    - Least total premium collected
    - Highest rejection ratio
    """
    from collections import defaultdict

    city_stats = defaultdict(lambda: {'claims': 0, 'total_premium': 0.0, 'rejected': 0})

    header = cleaned_data[0]
    city_idx = header.index("CITY")
    premium_idx = header.index("PREMIUM_COLLECTED")
    paid_idx = header.index("PAID_AMOUNT")

    for row in cleaned_data[1:]:
        city = row[city_idx]
        premium = float(row[premium_idx])
        paid = float(row[paid_idx])

        city_stats[city]['claims'] += 1
        city_stats[city]['total_premium'] += premium
        if paid == 0.0:
            city_stats[city]['rejected'] += 1

    print("City-wise stats:\n")
    for city, stats in city_stats.items():
        rejection_rate = stats['rejected'] / stats['claims']
        print(f"{city}: Claims={stats['claims']}, Premium={stats['total_premium']}, Rejection Rate={rejection_rate:.

    # Heuristic: prioritize low claims, low premium, high rejection
    city_to_consider = min(
        city_stats.items(),
        key=lambda item: (
            item[1]['claims'],
            item[1]['total_premium'],
            -item[1]['rejected'] / item[1]['claims']
        )
    )[0]

    return city_to_consider
```

Part 3: Fix and Apply `complex_rejection_classifier`

We are provided a function `complex_rejection_classifier` that classifies `REJECTION_REMARKS` into broader categories.

Categories:

- `Policy Issue`
- `Fraudulent Claim`
- `Documentation Issue`
- `Coverage Issue`
- `Pre-existing Condition`
- `Late Filing`
- `Other`

Steps:

- Fix any syntax or logic issues in the classifier
- Normalize remarks (lowercase, strip)
- Apply classification to each row
- Add a new column `REJECTION_CLASS` to the cleaned dataset

In [21]:
```python
def complex_rejection_classifier(remark):
    remark = remark.lower().strip()
    if 'policy lapsed' in remark:
        return 'Policy Issue'
    # elif 'fraud' in remark or 'fake' in remark:
    #     return 'Fraudulent Claim'
    # elif 'document' in remark or 'missing' in remark:
    #     return 'Documentation Issue'
    # elif 'not covered' in remark or 'exclusion' in remark:
    #     return 'Coverage Issue'
    # elif 'pre-existing' in remark:
    #     return 'Pre-existing Condition'
    # elif 'late' in remark or 'delay' in remark:
    #     return 'Late Filing'
    elif "fake_doc" in remark:
        return 'Fake_Document'
    elif "not_covered" in remark:
        return 'Not_Covered'
    elif "policy_expired" in remark:
        return 'Policy_expired'
    else:
        return 'Other'
```

Applying the `complex_rejection_classifier` Function

Now that we have cleaned the data, we proceed to classify the reasons for rejected claims using a custom classification function.

Goal:

Create a new column `REJECTION_CLASS` based on keywords in the `REJECTION_REMARKS` field.

Steps:

1. Define the corrected `complex_rejection_classifier` function.
2. For each row in the cleaned data:
   - If `REJECTION_REMARKS` is non-empty and `PAID_AMOUNT == 0`, classify the reason.
   - Otherwise, assign `'No Remark'` or `'Not Rejected'` accordingly.
3. Append the new classification column to the data.

In [22]:
```python
def classify_rejection_remarks(cleaned_data):
    header = cleaned_data[0] + ['REJECTION_CLASS']
    classified_data = [header]

    remark_idx = cleaned_data[0].index("REJECTION_REMARKS")

    for row in cleaned_data[1:]:
        remark = row[remark_idx]
        if remark:
            rejection_class = complex_rejection_classifier(remark)
        else:
            rejection_class = 'No Remark'
        classified_data.append(row + [rejection_class])

    return classified_data
```

Output:

Cleaned data with an additional column `REJECTION_CLASS`, ready for further analysis or export.

In [23]:
```python
# Step 1: Preprocess
file_path = "Insurance_auto_data.csv"
cleaned = preprocess_csv(file_path)

# Step 2: City Recommendation
recommended_city = analyze_city_for_shutdown(cleaned)
print(f"\n Recommended City for Shutdown: {recommended_city}")

# Step 3: Classification
final_data = classify_rejection_remarks(cleaned)
```

```
City-wise stats:

Pune: Claims=30, Premium=295452.03, Rejection Rate=0.10
Guwahati: Claims=22, Premium=254727.02000000005, Rejection Rate=0.09
Ranchi: Claims=13, Premium=114105.14, Rejection Rate=0.15
Kolkata: Claims=11, Premium=108026.83999999998, Rejection Rate=0.00

 Recommended City for Shutdown: Kolkata
```

Step 4: Save Cleaned and Classified Data to CSV

In [24]:
```python
# Step 4: Save Cleaned and Classified Data to CSV

def save_to_csv(data, output_file):
    with open(output_file, 'w', encoding='utf-8') as f:
        for row in data:
            # Ensure all values are converted to strings and properly escaped
            escaped_row = [str(value).replace(",", " ") for value in row]
            line = ",".join(escaped_row)
            f.write(line + "\n")

# Save to CSV file
output_file = "cleaned_insurance_data_with_classes.csv"
save_to_csv(final_data, output_file)

print(f"\nCleaned and classified data saved to: {output_file}")
```

```
Cleaned and classified data saved to: cleaned_insurance_data_with_classes.csv
```