Data Preprocessing

Data preprocessing is the process of **cleaning and transforming raw data** into a format suitable for analysis.

1.**Loading Data:**Use Pandas to load data from various sources (CSV, Excel, databases).

```
import pandas as pd
data = pd.read_csv('data.csv')
```

2.Data Cleaning:

Handling Missing Values: You can fill missing values or drop rows/columns.

```
data.fillna(value=0, inplace=True)
```

Removing Duplicates:

```
data.drop_duplicates(inplace=True)
```

3. Data Transformation:

Encoding Categorical Variables: Convert categories to numerical values using methods like one-hot encoding.

```
data = pd.get_dummies(data, columns=['category_column'])
```

Feature Scaling: Standardize or normalize numerical features.

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
data[['num_column']] = scaler.fit_transform(data[['num_column']])
```

4.Data Integration: Merging multiple DataFrames.

```
merged_data = pd.merge(data1, data2, on='key_column')
```

5.Data Reduction: Techniques like PCA (Principal Component Analysis) for dimensionality reduction.

```
from sklearn.decomposition import PCA
pca = PCA(n_components=2)
```

```
reduced_data = pca.fit_transform(data)
```

6.Splitting Data: Divide data into training and testing sets.

```
from sklearn.model_selection import train_test_split
train, test = train_test_split(data, test_size=0.2)
```

These steps help prepare the data for analysis or machine learning modeling, ensuring better results and performance.

Data Loading

Loading data is the first step in data preprocessing and can vary based on the data format and source.

Here's a quick guide on how to load data in Python using different libraries and formats:

1. Loading CSV Files

The most common format is CSV. You can use Pandas for this:

```
import pandas as pd
# Load a CSV file
data = pd.read_csv('your_dataset.csv')
# Display the first few rows
print(data.head())
```

2. Loading Excel Files

If your data is in an Excel format, you can also use Pandas:

```
# Load an Excel file
data = pd.read_excel('your_dataset.xlsx', sheet_name='Sheet1')
# Display the first few rows
print(data.head())
```

3. Loading JSON Files

JSON (**JavaScript Object Notation**) is one of the most common data formats — especially for APIs and configuration data.
It stores data as key–value pairs and is both **human-readable** and **machine-friendly**.

For JSON data, Pandas can again handle the loading:

```
# Load a JSON file
data = pd.read_json('your_dataset.json')
# Display the first few rows
print(data.head())
```

4. Loading from SQL Databases

You can use SQLAlchemy along with Pandas to load data from a SQL database:

```
from sqlalchemy import create_engine
# Create a database connection
engine = create_engine('sqlite:///your_database.db')
# Load data from a SQL query
data = pd.read_sql_query('SELECT * FROM your_table', engine)
# Display the first few rows
print(data.head())
```

5. Loading Data from APIs

For data fetched from APIs, you can use the requests library along with Pandas:

(**API (Application Programming Interface)** is a way for two applications to communicate.
When you send a request to an API, it returns data — often in **JSON** or **XML** format.)

```
import requests
# Make a request to an API
response = requests.get('https://api.example.com/data')
```

```
data_json = response.json()
# Convert to DataFrame
data = pd.DataFrame(data_json)
# Display the first few rows
print(data.head())
```

6. Loading from Other Formats

- **Text Files:** If you have a text file with a specific delimiter, use pd.read_csv with the delimiter parameter.

```
data = pd.read_csv('your_data.txt', delimiter='\t')
```

- **HDF5 Files:** Use pd.read_hdf for HDF5 formats.

```
data = pd.read_hdf('your_data.h5')
```

HDF5 (Hierarchical Data Format) is a binary format that efficiently stores large amounts of numerical data.

The method you choose for loading data will depend on the format of your dataset. Once loaded, you can proceed with the preprocessing steps like handling missing values, encoding categorical variables, and scaling features.

Storage

Data storage is an essential aspect of data preprocessing, especially when dealing with large datasets or when you need to save intermediate results.

Here are some common methods for storing data in Python, along with examples using various formats:

1. Storing Data in CSV Format

CSV is a widely used format for storing tabular data. You can save a DataFrame to a CSV file using Pandas:

```
import pandas as pd
# Assume 'data' is your DataFrame
```

data.to_csv('processed_data.csv', index=False)

2. Storing Data in Excel Format

You can also save your data in Excel format:
data.to_excel('processed_data.xlsx', index=False, sheet_name='Sheet1')

3. Storing Data in JSON Format

To save your DataFrame as a JSON file:
data.to_json('processed_data.json', orient='records', lines=True)

4. Storing Data in SQL Databases

You can write your DataFrame to a SQL database using SQLAlchemy. This is especially useful for larger datasets:
from sqlalchemy import create_engine
# Create a database connection
engine = create_engine('sqlite:///your_database.db')
# Write the DataFrame to a SQL table
data.to_sql('your_table', con=engine, if_exists='replace', index=False)

5. Storing Data in HDF5 Format

HDF5 is a binary format that is efficient for storing large datasets:
data.to_hdf('processed_data.h5', key='df', mode='w')

6. Storing Data as Pickle Files

Pickle is a Python-specific format for serializing Python objects, which is useful for saving DataFrames:
data.to_pickle('processed_data.pkl')

7. Storing Data in Parquet Format

Parquet is a columnar storage format that is highly efficient for analytical queries:
data.to_parquet('processed_data.parquet', index=False)

8. Storing Data in Feather Format

Feather is another binary format that provides fast read and write times:
data.to_feather('processed_data.feather')

Choosing the right storage format depends on your specific needs, such as data size, compatibility, and performance. CSV and Excel are great for interoperability, while formats like HDF5, Parquet, and Feather are better suited for efficiency and speed with larger datasets.

File formats

Data preprocessing often involves working with various file formats, each having its own strengths and weaknesses.

Here's an overview of some common file formats used in data preprocessing:

1. CSV (Comma-Separated Values)

- **Usage:** Widely used for tabular data.
- **Pros:** Easy to read and write, human-readable, supported by many tools.
- **Cons:** Lacks support for complex data structures (e.g., nested data).

2. JSON (JavaScript Object Notation)

- **Usage:** Common for APIs and hierarchical data.
- **Pros:** Supports complex nested structures, human-readable.
- **Cons:** Larger file sizes compared to CSV, parsing can be slower.

3. Excel (XLS/XLSX)

- **Usage:** Common in business settings for spreadsheets.
- **Pros:** Supports multiple sheets, rich formatting options.
- **Cons:** Not easily readable by all programming languages, requires specific libraries for manipulation.

4. Parquet

- **Usage:** Columnar storage format often used with big data frameworks (e.g., Apache Spark).
- **Pros:** Efficient for querying large datasets, supports complex data types.
- **Cons:** Less human-readable, requires specific tools to read/write.

5. HDF5 (Hierarchical Data Format)

- **Usage:** Used for large datasets in scientific computing.
- **Pros:** Supports large amounts of data, hierarchical data organization.
- **Cons:** Complex structure, not as widely used for general data tasks.

6. SQL Databases

- **Usage:** Structured data storage and retrieval.
- **Pros:** Supports complex queries, ACID compliance.
- **Cons:** Requires a database setup, can be overkill for smaller datasets.

7. XML (eXtensible Markup Language)

- **Usage:** Used for structured data interchange.
- **Pros:** Supports hierarchical data, widely used in web services.
- **Cons:** Verbose compared to JSON, parsing can be complex.

8. Text Files

- **Usage:** General-purpose data storage.
- **Pros:** Simple and versatile.
- **Cons:** Lack of structure, can be difficult to parse consistently.

9. Pickle (Python-specific)

- **Usage:** Serializing Python objects.
- **Pros:** Easy to save and load Python data structures.
- **Cons:** Python-specific, not human-readable, security concerns when loading untrusted data.

10. Avro

- **Usage:** Binary serialization format often used in big data.
- **Pros:** Compact, supports schema evolution.
- **Cons:** Requires specific tools for reading/writing.

Best Practices for Choosing a Format:

- **Consider Data Size:** Larger datasets may benefit from binary formats like Parquet or Avro.
- **Data Complexity:** Use JSON or XML for hierarchical data.
- **Tooling and Ecosystem:** Choose formats based on compatibility with your existing tools and frameworks.
- **Readability Needs:**Opt for CSV or Excel if human readability is important.

Selecting the right file format for data preprocessing is crucial for efficiency, compatibility, and ease of use. Each format serves different needs, so it's essential to understand the specific requirements of your data and project.

Reading and Writing data in Text format

Reading and writing data in text format is a fundamental aspect of data preprocessing. Text files can be simple, unstructured, or structured (like CSV). Here's a concise guide on how to handle text files in data preprocessing using Python, which is a commonly used language for this purpose.

Reading Data from Text Files

1. Plain Text Files

You can read plain text files using Python's built-in functions.

```
# Read a plain text file
with open('data.txt', 'r') as file:
    data = file.read()
print(data)
```

2. CSV Files

You can use the csv module or pandas for CSV files.

Using the csv module:

```
import csv
with open('data.csv', 'r') as file:
    reader = csv.reader(file)
    for row in reader:
        print(row)
```

Using pandas:

```
import pandas as pd
data = pd.read_csv('data.csv')
print(data.head())
```

Writing Data to Text Files

1. Plain Text Files

You can write to plain text files using Python's built-in functions.

```
data_to_write = "Hello, World!"
with open('output.txt', 'w') as file:
file.write(data_to_write)
```

2. CSV Files

You can also write to CSV files using the csv module or pandas.

Using the csv module:

```
import csv
```

```
data_to_write = [['Name', 'Age'], ['Alice', 30], ['Bob', 25]]
with open('output.csv', 'w', newline='') as file:
    writer = csv.writer(file)
writer.writerows(data_to_write)
```

Using pandas:

```
import pandas as pd
data_to_write = pd.DataFrame({
    'Name': ['Alice', 'Bob'],
    'Age': [30, 25]
})
data_to_write.to_csv('output.csv', index=False)
```

Tips for Text Data Preprocessing

1. **Handle Encodings:** Specify encodings (like utf-8) when reading/writing to avoid errors with special characters.

```
with open('data.txt', 'r', encoding='utf-8') as file:
    data = file.read()
```

2. **Data Cleaning:** Use string methods (like strip(), split(), replace()) to clean and preprocess text data.

```
cleaned_data = data.strip().splitlines()
```

3. **Error Handling:** Implement error handling (using try-except) to manage issues while reading/writing files.
4. **Batch Processing:** For large files, consider processing data in chunks to avoid memory issues.

```
with open('large_file.txt', 'r') as file:
    for line in file:
        # Process each line
        pass
```

5. **Use Libraries:** For advanced preprocessing (like tokenization, stemming, etc.), consider libraries like nltk or spaCy.

Reading and writing text data is straightforward with Python. By leveraging the right libraries and methods, you can effectively preprocess your data for analysis or modeling.

Binary Data formats

Binary data formats are efficient for storing and processing large datasets, especially in data preprocessing tasks. They are typically faster to read and write compared to text formats and can handle complex data structures.

Here's an overview of common binary formats and how to work with them in data preprocessing.

Common Binary Formats

1. **Parquet**
    - **Usage:** Columnar storage format optimized for big data processing.
    - **Pros:** Efficient compression, supports complex nested data.
    - **Libraries:** Apache Arrow, pyarrow, and pandas.
2. **HDF5 (Hierarchical Data Format)**
    - **Usage:** Suitable for large datasets and scientific computing.
    - **Pros:** Supports complex data types and efficient storage.
    - **Libraries:** h5py, pandas.
3. **Pickle**
    - **Usage:** Python-specific serialization format for Python objects.
    - **Pros:** Easy to use for storing Python data structures.
    - **Cons:** Not cross-language compatible.
    - **Libraries:** Built into Python.
4. **Avro**

- **Usage:** Row-oriented binary serialization format, often used in data pipelines.
- **Pros:** Compact storage, supports schema evolution.
- **Libraries:**fastavro, avro-python3.

Reading and Writing Binary Data

1. Parquet

```
import pandas as pd
# Writing data to a Parquet file
data = pd.DataFrame({'column1': [1, 2], 'column2': ['A', 'B']})
data.to_parquet('data.parquet')
# Reading data from a Parquet file
loaded_data = pd.read_parquet('data.parquet')
print(loaded_data)
```

2. HDF5

```
import pandas as pd
# Writing data to an HDF5 file
data = pd.DataFrame({'column1': [1, 2], 'column2': ['A', 'B']})
data.to_hdf('data.h5', key='df', mode='w')
# Reading data from an HDF5 file
loaded_data = pd.read_hdf('data.h5', key='df')
print(loaded_data)
```

3. Pickle

```
import pickle
# Writing data to a Pickle file
data = {'key': 'value', 'numbers': [1, 2, 3]}
with open('data.pkl', 'wb') as file:
```

```
pickle.dump(data, file)
# Reading data from a Pickle file
with open('data.pkl', 'rb') as file:
loaded_data = pickle.load(file)
print(loaded_data)
```

4. Avro

```
import fastavro
# Define schema for Avro
schema = {
   "type": "record",
   "name": "User",
   "fields": [
      {"name": "name", "type": "string"},
      {"name": "age", "type": "int"}
   ]
}
# Writing data to an Avro file
data = [{"name": "Alice", "age": 30}, {"name": "Bob", "age": 25}]
with open('data.avro', 'wb') as file:
fastavro.writer(file, schema, data)
# Reading data from an Avro file
with open('data.avro', 'rb') as file:
loaded_data = fastavro.reader(file)
   for record in loaded_data:
      print(record)
```

Tips for Binary Data Preprocessing

1. **Choose the Right Format:** Select a binary format based on your specific needs (e.g., Parquet for analytics, HDF5 for scientific data).

2. **Compression:** Many binary formats support compression options, which can significantly reduce file sizes.
3. **Chunking:** For very large datasets, consider processing data in chunks to manage memory efficiently.
4. **Schema Management:** Especially for formats like Avro and Parquet, define and manage schemas carefully to ensure compatibility across data versions.
5. **Error Handling:** Implement robust error handling to catch issues during read/write operations.

Binary formats are powerful tools for data preprocessing, offering efficiency and flexibility. By understanding how to read and write these formats, you can handle large and complex datasets effectively.

Data wrangling: Clean, Transform, Merge, Reshape

Data wrangling is a crucial part of data preprocessing, involving the steps of **cleaning, transforming, merging, and reshaping data to prepare it for analysis**.

Here's a comprehensive overview of each of these steps, primarily using Python and the pandas library.

1. Data Cleaning

Data cleaning involves identifying and correcting inaccuracies or inconsistencies in your dataset.

Common Cleaning Tasks:

- **Handling Missing Values:**

```
import pandas as pd
df = pd.read_csv('data.csv')
df.fillna(method='ffill', inplace=True)  # Forward fill
df.dropna(inplace=True)  # Drop rows with missing values
```

- **Removing Duplicates:**

```
df.drop_duplicates(inplace=True)
```

- **Correcting Data Types:**

```
df['date'] = pd.to_datetime(df['date'])  # Convert to datetime
df['age'] = df['age'].astype(int)  # Convert to integer
```

2. Data Transformation

Data transformation involves **modifying the data format or structure** to fit the requirements of your analysis.

Common Transformation Tasks:

- **Normalization/Standardization:**

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
df[['age', 'salary']] = scaler.fit_transform(df[['age', 'salary']])
```

- **Feature Engineering:**

```
df['age_group'] = pd.cut(df['age'], bins=[0, 18, 35, 60, 100],
labels=['child', 'young_adult', 'adult', 'senior'])
```

- **Date and Time Features:**

```
df['year'] = df['date'].dt.year
df['month'] = df['date'].dt.month
```

3. Data Merging

Merging data involves **combining multiple datasets into one** for comprehensive analysis.

Common Merging Techniques:

- **Concatenation:**

```
df1 = pd.read_csv('data1.csv')
df2 = pd.read_csv('data2.csv')
combined_df = pd.concat([df1, df2], ignore_index=True)
```

- **Joining:**

```
df_left = pd.DataFrame({'key': ['A', 'B'], 'value1': [1, 2]})
df_right = pd.DataFrame({'key': ['A', 'B'], 'value2': [3, 4]})
merged_df = pd.merge(df_left, df_right, on='key')
```

4. Data Reshaping

Reshaping data involves **changing its structure or format** to meet specific analysis needs.

Common Reshaping Techniques:

- **Pivoting:**

```
pivot_df = df.pivot(index='date', columns='category', values='sales')
```

- **Melting:**

```
melted_df = df.melt(id_vars=['id'], value_vars=['feature1', 'feature2'],
var_name='feature', value_name='value')
```

- **Stacking and Unstacking:**

```
stacked_df = df.set_index(['date', 'category']).stack()
unstacked_df = stacked_df.unstack()
```

pivoting: pivot() converts *long-form* data into a *wide-form* table.

example:

```
import pandas as pd

data = {
   'date': ['2024-01-01', '2024-01-01', '2024-01-02', '2024-01-02'],
   'category': ['Electronics', 'Clothing', 'Electronics', 'Clothing'],
   'sales': [2000, 1500, 3000, 1200]
}

df = pd.DataFrame(data)
print(df)
```

	date	category	sales
2024-01-01	Electronics	2000	
2024-01-01	Clothing	1500	
2024-01-02	Electronics	3000	
2024-01-02	Clothing	1200	

Pivot Operation

```
pivot_df = df.pivot(index='date', columns='category', values='sales')
```

print(pivot_df)

date	Clothing	Electronics
2024-01-01	1500	2000
2024-01-02	1200	3000

Melting: it converts data from *wide format* to *long format*.

Example:

```
data = {
   'id': [1, 2],
   'feature1': [10, 20],
   'feature2': [30, 40]
}
df = pd.DataFrame(data)
print(df)
```

id	feature1	feature2
1	10	30
2	20	40

```
melted_df = df.melt(id_vars=['id'],
                    value_vars=['feature1', 'feature2'],
                    var_name='feature',
                    value_name='value')
print(melted_df)
```

id	feature	value
1	feature1	10
2	feature1	20
1	feature2	30
2	feature2	40

Stacking and Unstacking: They *pivot* the inner level of the index or columns **into/out of the row axis.**

Example:

```
data = {
   'date': ['2024-01-01', '2024-01-01', '2024-01-02', '2024-01-02'],
   'category': ['Electronics', 'Clothing', 'Electronics', 'Clothing'],
   'sales': [2000, 1500, 3000, 1200]
}
```

df = pd.DataFrame(data)
print(df)

	date	category	sales
2024-01-01	Electronics	2000	
2024-01-01	Clothing	1500	
2024-01-02	Electronics	3000	
2024-01-02	Clothing	1200	

stacked_df = df.set_index(['date', 'category']).stack()
print(stacked_df)

	date	category		0
2024-01-01	Electronics	sales	2000	
2024-01-01	Clothing	sales	1500	
2024-01-02	Electronics	sales	3000	
2024-01-02	Clothing	sales	1200	

unstacked_df = stacked_df.unstack()
print(unstacked_df)

	date	category	sales
2024-01-01	Clothing	1500	
2024-01-01	Electronics	2000	
2024-01-02	Clothing	1200	
2024-01-02	Electronics	3000	

- **stack() → moves columns into rows.**
- **unstack() → moves inner row index back into columns**.

Example Workflow

Here's a complete example workflow that incorporates all these steps:

```
import pandas as pd
from sklearn.preprocessing import StandardScaler
# Step 1: Load Data
df = pd.read_csv('data.csv')
# Step 2: Clean Data
df.drop_duplicates(inplace=True)
df.fillna(df.mean(), inplace=True)
# Step 3: Transform Data
scaler = StandardScaler()
```

```
df[['age', 'salary']] = scaler.fit_transform(df[['age', 'salary']])
df['age_group'] = pd.cut(df['age'], bins=[0, 18, 35, 60, 100],
labels=['child', 'young_adult', 'adult', 'senior'])
# Step 4: Merge Data
df_other = pd.read_csv('other_data.csv')
merged_df = pd.merge(df, df_other, on='key_column')
# Step 5: Reshape Data
pivot_df = merged_df.pivot(index='date', columns='category',
values='sales')
# Display the final DataFrame
print(pivot_df)
```

Data wrangling is essential for preparing datasets for analysis. By following these steps—cleaning, transforming, merging, and reshaping—you can ensure that your data is in the best possible format for your analytical needs. The pandas library in Python provides powerful tools to accomplish these tasks efficiently.

Reshaping and pivoting

Reshaping and pivoting data are important techniques in data preprocessing, allowing you to **restructure** your data for better analysis. In Python, the pandas library provides powerful functions to perform these tasks.

Here's a detailed guide on how to reshape and pivot data effectively.

1. Reshaping Data

Reshaping refers to changing the structure of a DataFrame. Common reshaping techniques include stacking, unstacking, and melting.

a. Stacking and Unstacking

- **Stacking** converts columns into rows, resulting in a Series.
- **Unstacking** converts rows into columns, resulting in a DataFrame.

Example:

```
import pandas as pd
# Sample DataFrame
data = {
    'date': ['2024-01-01', '2024-01-02'],
    'A': [1, 2],
    'B': [3, 4]
}
df = pd.DataFrame(data).set_index('date')
# Stacking
stacked = df.stack()
print("Stacked DataFrame:")
print(stacked)
# Unstacking
unstacked = stacked.unstack()
print("\nUnstackedDataFrame:")
print(unstacked)
```

output:

date	A	B
2024-01-01	1	3
2024-01-02	2	4

After `stack()`

`stack()` turns **columns (A, B)** into an **inner row index level**.

Stacked DataFrame:

date

2024-01-01 A 1

B 3

2024-01-02 A 2

B 4

dtype: int64

After `unstack()`

`unstack()` reverses `stack()` —
it moves the **inner index (A, B)** back to columns.

Unstacked DataFrame:

A B

date

2024-01-01 1 3

2024-01-02 2 4

b. Melting

Melting is used to transform a wide-format DataFrame into a long-format DataFrame, making it easier to work with.

Example:

```
# Wide-format DataFrame
wide_df = pd.DataFrame({
   'id': [1, 2],
   'feature1': [10, 20],
   'feature2': [30, 40]
})
# Melting
melted_df = pd.melt(
wide_df,
 id_vars=['id'],
value_vars=['feature1', 'feature2'],
var_name='feature',
value_name='value')
print("Melted DataFrame:")
print(melted_df)
```

id	feature1	feature2
1	10	30
2	20	40

After `pd.melt()`

```
Melted DataFrame:
   id   feature  value
0   1  feature1     10
1   2  feature1     20
2   1  feature2     30
3   2  feature2     40
```

2. Pivoting Data

Pivoting allows you to reshape data by specifying which columns to use as new index and columns, and which values to fill in.

a. Basic Pivoting

You can use the pivot() function to reshape data.

Example:

```
# Sample DataFrame
data = {
    'date': ['2024-01-01', '2024-01-01', '2024-01-02', '2024-01-02'],
    'category': ['A', 'B', 'A', 'B'],
    'value': [10, 20, 30, 40]
}
df = pd.DataFrame(data)
# Pivoting
pivoted_df = df.pivot(index='date', columns='category', values='value')
print("Pivoted DataFrame:")
print(pivoted_df)
```

b. Pivot Table

The pivot_table() function provides more flexibility, allowing you to aggregate data. It is useful when there are multiple values for the same index/column combination.

Example:

```
# Sample DataFrame with duplicate entries
data = {
```

```
    'date': ['2024-01-01', '2024-01-01', '2024-01-02', '2024-01-02'],
    'category': ['A', 'B', 'A', 'B'],
    'value': [10, 20, 30, 40]
}
df = pd.DataFrame(data)
# Pivot table with aggregation (mean)
pivot_table_df = df.pivot_table(index='date', columns='category',
values='value', aggfunc='mean')
print("Pivot Table DataFrame:")
print(pivot_table_df)
```

output:

date	category	value
2024-01-01	A	10
2024-01-01	B	20
2024-01-02	A	30
2024-01-02	B	40

Pivot Table DataFrame:

category	A	B
date		
2024-01-01	10	20
2024-01-02	30	40

3. Handling Missing Values in Pivoting

When pivoting, if there are no values for a certain combination, the result will contain NaN. You can handle missing values using methods like fillna().

```
# Filling missing values
pivot_table_df.fillna(0, inplace=True)
print("Pivot Table with NaN replaced:")
print(pivot_table_df)
```

Reshaping and pivoting data are essential preprocessing techniques that facilitate better data analysis.

By using functions like stack(), unstack(), melt(), pivot(), and pivot_table(), you can transform your data into the desired structure for insights and modeling.

The pandas library makes these tasks efficient and straightforward, enabling you to work with complex datasets effectively.

String Manipulation

String manipulation is an important aspect of data pre-processing, especially when dealing with textual data.

It allows you to clean, modify, and extract meaningful information from strings.

Here's a detailed overview of common string manipulation techniques using Python, primarily with the pandas library.

1. Basic String Operations

a. Accessing String Attributes

You can access string attributes directly through the str accessor in a pandas DataFrame.

```
import pandas as pd
# Sample DataFrame
df = pd.DataFrame({'Names': ['Alice Smith', 'Bob Johnson', 'Charlie Brown']})
# Accessing string attributes
df['Length'] = df['Names'].str.len()  # Length of each name
print("Length of Names:")
print(df)
```

output:

```
Length of Names:
           Names  Length
0    Alice Smith      11
1    Bob Johnson      11
2  Charlie Brown      13
```

b. Changing Case

You can convert strings to different cases using methods like .lower(), .upper().

example:

```
import pandas as pd

import numpy as np

data = {'Names': ['Gulshan', 'Shashank', 'Bablu', 'Abhishek', 'Anand', np.nan, 'Pratap'],

        'City': ['Delhi', 'Mumbai', 'Kolkata', 'Delhi', 'Chennai', 'Bangalore', 'Hyderabad']}

df = pd.DataFrame(data)

print(df)
```

```
      Names       City
0   Gulshan      Delhi
1  Shashank     Mumbai
2     Bablu    Kolkata
3  Abhishek      Delhi
4     Anand    Chennai
5       NaN  Bangalore
6    Pratap  Hyderabad
```

Lower():

Converts all uppercase characters in strings in the DataFrame to lower case and returns the lowercase strings in the result.

```
print(df['Names'].str.lower())
```

```
0     gulshan
1    shashank
2       bablu
3    abhishek
4       anand
5         NaN
6      pratap
Name: Names, dtype: object
```

Upper():

Converts all lowercase characters in strings in the DataFrame to upper case and returns the uppercase strings in result.

print(df['Names'].str.upper())

output:

```
0     GULSHAN
1    SHASHANK
2       BABLU
3    ABHISHEK
4       ANAND
5         NaN
6      PRATAP
Name: Names, dtype: object
```

2. String Cleaning

a. Removing Whitespace

Use .strip(), .lstrip(), and .rstrip() to remove whitespace.

- **.str.lstrip()** → removes **leading (left) whitespace** from each string.
- **.str.rstrip()** → removes **trailing (right) whitespace** from each string.
- Both are similar to .str.strip(), which removes **both sides**.

Strip():

If there are spaces at the beginning or end of a string, we should trim the strings to eliminate spaces using strip() or remove the extra spaces contained by a string in DataFrame.

Example:

```
import pandas as pd
df = pd.DataFrame({
    'Names': ['  Alice Smith  ', '  Bob Johnson', 'Charlie Brown   ']
})
print("Original DataFrame:")
print(df)
df['Cleaned_Names'] = df['Names'].str.strip()
print("After stripping whitespace:")
print(df)
```

output:

```
            Names    Cleaned_Names
0     '  Alice Smith  '    'Alice Smith'
1     '  Bob Johnson'      'Bob Johnson'
2     'Charlie Brown   '   'Charlie Brown'
```

b. Replacing Substrings

You can replace parts of strings using .replace().

It replaces the value a with the value b like below in example 'Smith' is being replaced by 'junnu'.

```
0       Alice junnu
1         Bob Johnson
2    Charlie Brown
Name: Names, dtype: object
```

3. Extracting Information
a. Substring Extraction

🎬 Substring extraction means **extracting a part of a string** from a column.

🎬 In Pandas, this is typically done using the **.str accessor** methods:

Slicing:series.str[start:stop] → extracts by index.

You can extract substrings using .str.slice() or .str[<start>:<end>].
Example:

```
# Extract first 5 characters
df['First_5'] = df['Names'].str[:5]
print(df)
```

```
              Names First_5
0       Alice Smith   Alice
1       Bob Johnson   Bob J
2  Charlie Brown      Charl
```

```
# Extract last name
df['Last_Name'] = df['Names'].str.split().str[1]
print(df)
```

output:

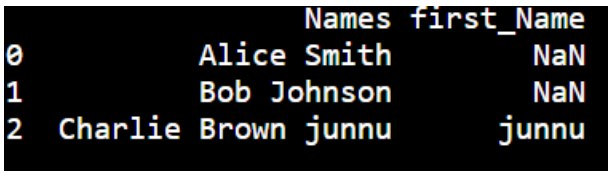
```
              Names Last_Name
0       Alice Smith     Smith
1       Bob Johnson   Johnson
2  Charlie Brown        Brown
```

df['first_Name'] = df['Names'].str.split().str[0]

```
              Names first_Name
0       Alice Smith      Alice
1       Bob Johnson        Bob
2  Charlie Brown       Charlie
```

df = pd.DataFrame({
'Names': ['Alice Smith', 'Bob Johnson', 'Charlie Brown junnu']
})

df['first_Name'] = df['Names'].str.split().str[2]



```
                 Names first_Name
0          Alice Smith        NaN
1          Bob Johnson        NaN
2  Charlie Brown junnu      junnu
```

b. Regular Expressions

For more complex extractions, you can use regular expressions with .str.extract()

Extracting first and last names using regex
df[['First_Name', 'Last_Name']] =
df['Names'].str.extract(r'(\w+)\s+(\w+)')
print("\nExtracted First and Last Names:")
print(df)

output:

Names First_NameLast_Name
0 Alice Smith Alice Smith
1 Bob Johnson Bob Johnson
2 Charlie Brown Charlie Brown

4. String Concatenation

You can concatenate strings using the + operator or .str.cat().

```
# Concatenating strings
df['Full_Name'] = df['First_Name'] + ' ' + df['Last_Name']
print("\nConcatenated Full Names:")
print(df[['First_Name', 'Last_Name', 'Full_Name']])
```

example:

```
import pandas as pd

df = pd.DataFrame({
   'First_Name': ['Alice', 'Bob', 'Charlie'],
   'Last_Name': ['Smith', 'Johnson', 'Brown']
})

print(df)
df['Full_Name'] = df['First_Name'] + ' ' + df['Last_Name']
```

output:

```
First_NameLast_NameFull_Name
0     Alice     Smith    Alice Smith
1       Bob   Johnson    Bob Johnson
2    Charlie     Brown  Charlie Brown
```

df['Full_Name'] = df['First_Name'].str.cat(df['Last_Name'], sep=' ')

output:

First_Name	Last_Name	Full_Name
Alice	Smith	Alice Smith
Bob	Johnson	Bob Johnson
Charlie	Brown	Charlie Brown

5. Handling Missing Values

When working with string data, you may encounter missing values. Use .fillna() to handle them.

```
# Fill missing names with a default value
df['Names'].fillna('Unknown', inplace=True)
print("\nMissing Names Handled:")
print(df)
```

6. Case Transformation and Normalization

You might want to standardize the format of your string data, such as making them all lowercase.

```
# Normalize to lowercase
df['Normalized'] = df['Names'].str.lower()
print("\nNormalized Names:")
print(df)
```

By using various methods available in the pandas library, you can efficiently perform operations such as cleaning, extracting, and transforming strings. Mastering these techniques will enhance your ability to work with data that includes textual information.

Data Aggregation and Group Operations

Data aggregation and group operations are essential techniques in data pre-processing, allowing you to summarize and analyse your data effectively.

In Python, the pandas library provides powerful functions to perform these operations.

Here's a detailed guide on how to use aggregation and grouping in your data pre-processing tasks.

1. Aggregation

Aggregation involves summarizing data by applying a function to a set of values. Common aggregation functions include sum, mean, median, count, and more.

a. Using groupby()

The groupby() function is used to group data based on one or more columns.

Example:

```
import pandas as pd
# Sample DataFrame
data = {
   'Category': ['A', 'B', 'A', 'B', 'A'],
   'Value': [10, 20, 30, 40, 50]
}
df = pd.DataFrame(data)
# Aggregating data by category
grouped = df.groupby('Category')['Value'].sum()
print("Aggregated Sum by Category:")
print(grouped)
```

output:

```
Aggregated Sum by Category:
Category
A   90
B   60
Name: Value, dtype: int64
```

2. Multiple Aggregations

You can apply multiple aggregation functions at once using the .agg() method.

Example:

```
# Multiple aggregations
agg_result = df.groupby('Category')['Value'].agg(['mean', 'sum', 'count'])
print("\nMultiple Aggregations:")
print(agg_result)
```

output:
Multiple Aggregations:

Category	mean	sum	count
A	30.0	90	3
B	30.0	60	2

3. Custom Aggregations

You can define your own aggregation functions using a custom function or lambda function.a

Example:

```
# Custom aggregation function
import pandas as pd

data = {
   'Category': ['A', 'B', 'A', 'B', 'A'],
   'Value': [10, 20, 30, 40, 50]
}
df = pd.DataFrame(data)
print(df)
defrange_func(x):
   return x.max() - x.min()
custom_agg_result =
df.groupby('Category')['Value'].agg(range=range_func)
print("\nCustom Aggregation (Range):")
print(custom_agg_result)
```

explanation:

Range = Maximum − Minimum

Category	Values	Max	Min	Range
A	[10, 30, 50]	50	10	50 − 10 = **40**
B	[20, 40]	40	20	40 − 20 = **20**

output:

Custom Aggregation (Range):
Category
A 40
B 20
Name: range, dtype: int64

4. Grouping by Multiple Columns

You can group by multiple columns to create more granular summaries.

Example:

```
# Sample DataFrame with multiple grouping columns
data = {
   'Category': ['A', 'B', 'A', 'B', 'A'],
   'Subcategory': ['X', 'Y', 'Y', 'X', 'X'],
   'Value': [10, 20, 30, 40, 50]
}
df = pd.DataFrame(data)
# Grouping by multiple columns
grouped_multi = df.groupby(['Category',
'Subcategory'])['Value'].sum()
print("\nAggregated Sum by Category and Subcategory:")
print(grouped_multi)
```

output:

Index	Category	Subcategory	Value
0	A	X	10
1	B	Y	20
2	A	Y	30
3	B	X	40
4	A	X	50

Internal calculation

Category	Subcategory	Values in Group	Sum
A	X	[10, 50]	10 + 50 = **60**

		Index	Category Subcategory Value
A	Y	[30]	**30**
B	X	[40]	**40**
B	Y	[20]	**20**

Final output:
Aggregated Sum by Category and Subcategory:
Category Subcategory
A X 60
Y 30
B X 40
Y 20
Name: Value, dtype: int64

5. Transformations

Sometimes, you might want to perform operations that return a DataFrame with the same shape as the original. Use transform() for such cases.

Example:

```
# Transforming to get the mean per category in the original DataFrame
df['Value_Mean'] = df.groupby('Category')['Value'].transform('mean')
print("\nOriginal DataFrame with Mean Values:")
print(df)
```

explanation:

Category	Values in Group	Mean of Group
A	[10, 30, 50]	(10 + 30 + 50) / 3 = **30.0**
B	[20, 40]	(20 + 40) / 2 = **30.0**

Output:
Category
A 30.0
B 30.0
Name: Value, dtype: float64

6. Filtering Groups

You can filter groups based on certain conditions using filter().

Example:

```
# Filtering groups where the sum is greater than a threshold
filtered_groups = df.groupby('Category').filter(lambda x:
x['Value'].sum() > 50)
print("\nFiltered Groups (Sum > 50):")
print(filtered_groups)
```

explanation:

Category	Values	Sum	Condition (> 50)?	Keep Group?
A	[10, 30, 50]	90	✅ 90 > 50 → True	✅ Keep
B	[20, 40]	60	✅ 60 > 50 → True	✅ Keep

Output:

```
Filtered Groups (Sum > 50):
  Category Subcategory  Value
0        A           X     10
1        B           Y     20
2        A           Y     30
3        B           X     40
4        A           X     50
```

7. Pivot Tables

Pivot tables are a powerful way to aggregate and summarize data in a more structured format.

Example:

```
# Creating a pivot table
pivot_table = df.pivot_table(values='Value', index='Category',
columns='Subcategory', aggfunc='sum', fill_value=0)
print("\nPivot Table:")
print(pivot_table)
```

explanation:

Category	Subcategory	Values	Sum
A	X	[10, 50]	10 + 50 = **60**
A	Y	[30]	**30**
B	X	[40]	**40**
B	Y	[20]	**20**

Category ↓ / Subcategory →	X	Y
A	60	30
B	40	20

output:

Pivot Table:

Subcategory	X	Y
Category		
A	60	30
B	40	20

Conclusion

Data aggregation and group operations are crucial for summarizing and analyzing data effectively. By leveraging functions like groupby(), agg(), transform(), and pivot_table(), you can extract valuable insights from your data.

Group by Mechanics

Understanding the mechanics of the groupby operation in data preprocessing is essential for effectively summarizing and analyzing datasets. The groupby function in the pandas library allows you to group data based on one or more keys (columns) and perform various operations on those groups. Here's a comprehensive overview of how groupby works and how to use it effectively.

1. Basic Mechanics of groupby

The groupby function splits the data into groups based on unique values in the specified columns. After splitting, you can perform operations like aggregation, transformation, or filtering.

Syntax:

```
import pandas as pd
# Sample DataFrame
data = {
   'Category': ['A', 'B', 'A', 'B', 'A'],
   'Value': [10, 20, 30, 40, 50]
}
df = pd.DataFrame(data)
# Group by 'Category'
grouped = df.groupby('Category')
```

explanation:

- groupby('Category')**splits the DataFrame** into groups based on unique values in the Category column.
- Here, two groups are created: A and B.

Index	Category	Value
0	A	10
1	B	20
2	A	30
3	B	40
4	A	50

Output:

{'A': [0, 2, 4], 'B': [1, 3]}

2. Creating Groups

You can group by one or multiple columns.

a. Grouping by One Column

```
import pandas as pd
# Sample DataFrame
data = {
   'Category': ['A', 'B', 'A', 'B', 'A'],
   'Value': [10, 20, 30, 40, 50]
}
df = pd.DataFrame(data)
# Group by 'Category'
grouped = df.groupby('Category')
```

b. Grouping by Multiple Columns

```
# Sample DataFrame with multiple grouping columns
data = {
   'Category': ['A', 'B', 'A', 'B', 'A'],
   'Subcategory': ['X', 'Y', 'Y', 'X', 'X'],
   'Value': [10, 20, 30, 40, 50]
}
df = pd.DataFrame(data)
# Group by both 'Category' and 'Subcategory'
grouped_multi = df.groupby(['Category', 'Subcategory'])
```

Index	Category	Subcategory	Value
0	A	X	10
1	B	Y	20
2	A	Y	30
3	B	X	40
4	A	X	50

creates groups based on **all unique combinations** of `Category` and `Subcategory`.

Output:

{('A', 'X'): [0, 4], ('A', 'Y'): [2], ('B', 'X'): [3], ('B', 'Y'): [1]}

3. Aggregation

After grouping, you can perform various aggregation operations using the .agg() method.

a. Built-in Aggregations

You can use common aggregation functions like sum, mean, count, etc.

```
# Aggregating with built-in functions
agg_result = grouped['Value'].agg(['sum', 'mean', 'count'])
print(agg_result)
```

output:

```
sum  mean  count
Category
A        90  30.0     3
B        60  30.0     2
```

b. Multiple Aggregations

You can specify multiple aggregation functions at once.

```
# Multiple aggregations
agg_result_multi = grouped['Value'].agg(['mean', 'sum', lambda x: x.max() - x.min()])
print(agg_result_multi)
```

output:

```
mean  sum  range
Category
A       30.0   90    40
B       30.0   60    20
```

4. Transformations

Transformations allow you to apply functions to each group and return a DataFrame that maintains the original shape.

```
# Applying transformation to get the mean value per category
df['Value_Mean'] = grouped['Value'].transform('mean')
```

print(df)
output:
Category
A 30.0
B 30.0
Name: Value, dtype: float64

5. Filtering Groups

You can filter out groups based on a condition.
# Filtering groups where the sum of values is greater than 50
filtered_groups = df.groupby('Category').filter(lambda x: x['Value'].sum() > 50)
print(filtered_groups)
output:

Category	Value
A	10
B	20
A	30
B	40
A	50

6. Applying Custom Functions

You can apply custom functions using .apply().
Custom function to apply
defcustom_function(x):
  return pd.Series({'Sum': x.sum(), 'Count': x.count()})
custom_agg_result = grouped['Value'].apply(custom_function)
print(custom_agg_result)
explanation:
`.apply()` is especially useful for returning multiple values per group in a structured way.

- Input `x` is the **group's `Value` column**.
- `x.sum()` → sum of values in the group
- `x.count()` → number of elements in the group
- Returns a **pandas Series** with named values: `'Sum'` and `'Count'`.

Output:

Sum Count

Category

A 90 3

B 60 2

7. Grouping by Index

You can also group by the DataFrame's index.
# Set index and group by index
df.set_index('Category', inplace=True)
grouped_by_index = df.groupby(level=0) # Group by the first level of the index
print(grouped_by_index['Value'].sum())
output:
Category
A 90
B 60
Name: Value, dtype: int64

8. Using as_index Parameter

By default, the grouped keys are used as the index of the resulting DataFrame. You can set as_index=False to keep the original DataFrame structure.

```
# Grouping without setting the index
result = df.groupby('Category', as_index=False)['Value'].sum()
print(result)
```

explanation:

- `groupby('Category')` → splits the DataFrame into groups based on `Category`.
- `'Value'.sum()` → sums the `Value` column within each group.
- `as_index=False` → keeps `'Category'` as a **regular column**, **not as the index**.

Output:

	Category	Value
0	A	90
1	B	60

9. Handling Missing Values

When working with groups, be mindful of missing values, which can affect aggregation results.

```
# Fill missing values before aggregation
df['Value'].fillna(0, inplace=True)
agg_result = df.groupby('Category')['Value'].sum()
print(agg_result)
```

The groupby operation in pandas is a powerful tool for data analysis and preprocessing.

By understanding its mechanics—how to group data, apply aggregations, transformations, and filters—you can extract meaningful insights from your datasets.

Data aggregation

Data aggregation is a crucial step in data preprocessing that involves summarizing and combining data to derive meaningful insights.

This process allows you to condense large datasets into simpler forms, making it easier to analyze trends, patterns, and key metrics.

In Python, the pandas library provides powerful tools for performing data aggregation. Here's a comprehensive guide to data aggregation in pandas.

1. Basics of Data Aggregation

Aggregation typically involves applying statistical functions to groups of data. Common aggregation functions include:

- **Sum:** Total of values.
- **Mean:** Average of values.
- **Count:** Number of occurrences.
- **Min/Max:** Minimum and maximum values.
- **Median:** Middle value of sorted data.

2. Using groupby()

The groupby() function is the foundation of data aggregation in pandas. It splits the data into groups based on one or more keys (columns), allowing you to apply aggregation functions to these groups.

Example:

```
import pandas as pd
# Sample DataFrame
data = {
   'Category': ['A', 'B', 'A', 'B', 'A'],
   'Value': [10, 20, 30, 40, 50]
```

```
}
df = pd.DataFrame(data)
# Group by 'Category'
grouped = df.groupby('Category')
```

3. Basic Aggregation

Once the data is grouped, you can easily apply aggregation functions.

a. Single Aggregation

```
# Aggregating with sum
sum_result = grouped['Value'].sum()
print("Sum by Category:")
print(sum_result)
```

output:

```
Sum by Category:
Category
A    90
B    60
Name: Value, dtype: int64
```

b. Multiple Aggregations

You can apply multiple aggregation functions using the .agg() method.

```
# Multiple aggregations
agg_result = grouped['Value'].agg(['mean', 'sum', 'count'])
print("\nMultiple Aggregations:")
print(agg_result)
```

4. Custom Aggregations

You can also define custom aggregation functions to perform more complex calculations.

```
# Custom aggregation function
defrange_func(x):
    return x.max() - x.min()
```

```
custom_agg_result = grouped['Value'].agg(range=range_func)
print("\nCustom Aggregation (Range):")
print(custom_agg_result)
```

5. Aggregation with Multiple Columns

You can group by multiple columns for more detailed analysis.

Example:

```
# Sample DataFrame with multiple grouping columns
data = {
   'Category': ['A', 'B', 'A', 'B', 'A'],
   'Subcategory': ['X', 'Y', 'Y', 'X', 'X'],
   'Value': [10, 20, 30, 40, 50]
}
df = pd.DataFrame(data)
# Grouping by both 'Category' and 'Subcategory'
grouped_multi = df.groupby(['Category',
'Subcategory']).agg({'Value': ['mean', 'sum']})
print("\nAggregated by Category and Subcategory:")
print(grouped_multi)
```

Explanation:

- `.groupby(['Category', 'Subcategory'])` → splits the DataFrame into **groups based on both `Category` and `Subcategory`**.
- `.agg({'Value': ['mean', 'sum']})` → applies multiple aggregation functions (`mean` and `sum`) to the `Value` column.

output:

		Value	
		mean	sum
Category	Subcategory		
A	X	30	60
	Y	30	30

B X 40 40
Y 20 20

6. Transformations

Transformations allow you to apply a function to each group and return a DataFrame with the same shape as the original. This is useful for creating new columns based on group statistics.

```
# Adding a column with the mean value per category
df['Value_Mean'] = grouped['Value'].transform('mean')
print("\nDataFrame with Mean Values Added:")
print(df)
```

output:

Category	Value	Value_Mean
A	10	30
B	20	30
A	30	30
B	40	30
A	50	30

7. Filtering Groups

You can filter out groups based on aggregation results using the filter() method.

```
# Filter groups where the sum is greater than a threshold
filtered_groups = df.groupby('Category').filter(lambda x:
x['Value'].sum() > 50)
print("\nFiltered Groups (Sum > 50):")
print(filtered_groups)
```

8. Pivot Tables

Pivot tables provide a flexible way to aggregate data, allowing you to summarize and reorganize it in a table format.

```
# Creating a pivot table
pivot_table = df.pivot_table(values='Value', index='Category',
columns='Subcategory', aggfunc='sum', fill_value=0)
```

```
print("\nPivot Table:")
print(pivot_table)
```

9. Handling Missing Values

Before performing aggregation, ensure you handle missing values appropriately, as they can skew results.

```
# Fill missing values before aggregation
df['Value'].fillna(0, inplace=True)
agg_result = df.groupby('Category')['Value'].sum()
print("\nSum after Filling Missing Values:")
print(agg_result)
```

Data aggregation is a powerful technique in data preprocessing that enables you to summarize and analyze data efficiently.

By using the groupby() function along with aggregation methods like .agg(), you can extract valuable insights from your datasets.

Pivot Tables

Pivot tables are a powerful tool in data preprocessing that allow you to summarize and reorganize data in a structured format.

1. What is a Pivot Table?

A pivot table is a data processing tool that allows you to transform data from a long format into a wide format. It aggregates data based on one or more keys (columns) and provides a summary for different combinations of these keys.

2. Creating a Pivot Table with pandas

To create a pivot table in pandas, you can use the pivot_table() function.

Basic Syntax

```
pd.pivot_table(data, values, index, columns, aggfunc, fill_value)
```

- **data:** The DataFrame you want to pivot.

- **values:** The column(s) to aggregate.
- **index:** The column(s) to use to make new frame's index.
- **columns:** The column(s) to use to make new frame's columns.
- **aggfunc:** The aggregation function(s) to use (e.g., sum, mean).
- **fill_value:** Value to replace missing values.

3. Example of Creating a Pivot Table

Here's an example to demonstrate how to create and use pivot tables:

```
import pandas as pd
# Sample DataFrame
data = {
   'Category': ['A', 'B', 'A', 'B', 'A'],
   'Subcategory': ['X', 'Y', 'Y', 'X', 'X'],
'Value': [10, 20, 30, 40, 50]
}
df = pd.DataFrame(data)
# Creating a pivot table
pivot_table = pd.pivot_table(df, values='Value', index='Category', columns='Subcategory', aggfunc='sum', fill_value=0)
print("Pivot Table:")
print(pivot_table)
```

```
Pivot Table:
Subcategory   X   Y
Category
A            60  30
B            40  20
```

4. Aggregation Functions

You can use various aggregation functions like sum, mean, count, etc., to summarize the data in your pivot table.

Example with Multiple Aggregation Functions:

```
# Creating a pivot table with multiple aggregation functions
```

```
pivot_table_multi = pd.pivot_table(df, values='Value',
index='Category', columns='Subcategory', aggfunc=[sum, 'mean'],
fill_value=0)
print("\nPivot Table with Multiple Aggregation Functions:")
print(pivot_table_multi)
```

3. Calculations per group

Category A:

- Subcategory X → values `[10, 50]` → sum = 60, mean = 30
- Subcategory Y → values `[30]` → sum = 30, mean = 30

Category B:

- Subcategory X → values `[40]` → sum = 40, mean = 40
- Subcategory Y → values `[20]` → sum = 20, mean = 20

```
Pivot Table with Multiple Aggregation Functions:
            sum      mean
Subcategory   X   Y     X     Y
Category
A            60  30  30.0  30.0
B            40  20  40.0  20.0
```

5. Flattening the Pivot Table

Sometimes, the resulting pivot table may have a multi-level column index, especially when using multiple aggregation functions. You can flatten it for easier access:

```
# Flattening the column index
pivot_table_flat = pivot_table_multi.reset_index()
print("\nFlattened Pivot Table:")
print(pivot_table_flat)
```

OUTPUT:

```
Category  sum      mean
          X   Y    X    Y
0         A  60  30   30.0  30.0
1         B  40  20   40.0  20.0
```

6. Using pivot()

While pivot_table() is used for aggregation, you can also use the pivot() function if you do not need to aggregate data.

```
# Sample DataFrame
data = {
   'Date': ['2024-01-01', '2024-01-01', '2024-01-02', '2024-01-02'],
   'Category': ['A', 'B', 'A', 'B'],
   'Value': [10, 20, 30, 40]
}
df = pd.DataFrame(data)
# Using pivot
pivot_simple = df.pivot(index='Date', columns='Category',
values='Value')
print("\nSimple Pivot Table:")
print(pivot_simple)
```

OUTPUT:

```
Category      A   B
Date
2024-01-01    10  20
2024-01-02    30  40
```

7. Handling Missing Values

When creating pivot tables, it's common to encounter missing values. You can handle these using the fill_value parameter, which replaces NaN with a specified value.

```
# Creating a pivot table with fill_value
pivot_table_fillna = pd.pivot_table(df, values='Value',
index='Category', columns='Subcategory', aggfunc='sum',
fillna_value=0)
print("\nPivot Table with Fill Value for Missing Data:")
print(pivot_table_fill)
```

Pivot tables are an essential tool for data preprocessing, allowing you to summarize and analyze data effectively.

By leveraging the pivot_table() function in pandas, you can easily aggregate data, explore relationships, and prepare your data for further analysis or modeling.

Understanding how to create and manipulate pivot tables will greatly enhance your data analysis capabilities in Python.

Cross Tabulation

Cross tabulation is a powerful technique in data preprocessing used to summarize and analyze the relationship between two or more categorical variables.

In Python, the pandas library provides a straightforward way to perform cross tabulation using the crosstab() function. Here's a detailed guide on how to use cross tabulation in your data preprocessing tasks.

1. What is Cross Tabulation?

Cross tabulation (or crosstab) creates a table that displays the frequency distribution of one variable against another. This can help you identify patterns, relationships, and trends between categorical variables.

Example:
It helps in understanding how one category varies with another — like gender vs. purchase decision, education vs. income group, etc.

2. Using pd.crosstab()

The basic syntax for creating a cross tabulation in pandas is:

pd.crosstab(index, columns, values=None, aggfunc=None, margins=False, margins_name='All')

- **index:** The rows of the table.
- **columns:** The columns of the table.
- **values:** An optional array of values to aggregate.
- **aggfunc:** The aggregation function to apply (e.g., sum, mean).
- **margins:** If True, adds row/column totals.
- **margins_name:** Name for the totals row/column.

3. Example of Cross Tabulation

Here's a simple example to illustrate how to perform cross tabulation:

```
import pandas as pd
data = {
   'Gender': ['Male', 'Female', 'Male', 'Female', 'Male'],
   'Purchase': ['Yes', 'No', 'Yes', 'Yes', 'No']
}
df = pd.DataFrame(data)
result = pd.crosstab(df['Gender'], df['Purchase'])
print(result)
```

output:

Purchase No Yes
Gender
Female 1 1
Male 1 2

#With Row & Column Totals

result = pd.crosstab(df['Gender'], df['Purchase'], margins=True)
print(result)
output:
Purchase No Yes All
Gender
Female 1 1 2
Male 1 2 3
All 2 3 5

#what percentage of each gender purchased or not purchased.

ROW WISE:

pd.crosstab(df['Gender'], df['Purchase'], normalize='index')
output:
Purchase No Yes
Gender
Female 0.50 0.50
Male 0.33 0.67
Explanation:

- **Female** customers: 2 total → 1 No & 1 Yes
  - 1/2 = **0.50**
- **Male** customers: 3 total → 1 No & 2 Yes
  - 1/3 = **0.33**
  - 2/3 = **0.67**

COLUMN WISE:

pd.crosstab(df['Gender'], df['Purchase'], normalize='columns')

output:

Purchase No Yes

Gender

Female 0.50 0.33

Male 0.50 0.67

Explanation:

`normalize='columns'` gives **column-wise proportions**:

- **"No" column:**
  - Total "No" = 2 (1 Female, 1 Male)
  - Female = 1/2 = **0.50**
  - Male = 1/2 = **0.50**
- **"Yes" column:**
  - Total "Yes" = 3 (1 Female, 2 Male)
  - Female = 1/3 ≈ **0.33**
  - Male = 2/3 ≈ **0.67**

4. Adding Aggregated Values

You can also include aggregated values in your cross tabulation.

Example:

```
# Sample DataFrame with values
data_with_values = {
   'Category': ['A', 'B', 'A', 'B', 'A', 'B'],
   'Subcategory': ['X', 'Y', 'X', 'X', 'Y', 'Y'],
   'Value': [10, 20, 30, 40, 50, 60]
}
df_values = pd.DataFrame(data_with_values)
# Creating a cross tabulation with values
crosstab_values = pd.crosstab(
df_values['Category'],
 df_values['Subcategory'],
```

```
values=df_values['Value'],
 aggfunc='sum',
margins=True)
print("\nCross Tabulation with Aggregated Values:")
print(crosstab_values)
```

output:

```
Subcategory   X    Y   All
Category
A            40   50   90
B            40   80   120
All          80  130   210
```

Explanation:

- **Rows** → `Category` (A, B)
- **Columns** → `Subcategory` (X, Y)
- **Values** → `Value` column
- Aggregation → `sum` (added all values belonging to each group)
- `margins=True` adds **row and column totals** ("All")

Category A:

Subcategory	Values
X	10 + 30 = **40**
Y	50 = **50**

Row total for A = 40 + 50 = **90**

CATEGORY B:

Subcategory	Values

Subcategory	Values
X	40 = **40**
Y	20 + 60 = **80**

Row total for B = 40 + 80 = **120**

Column Totals

Subcategory	Sum
X	40 + 40 = **80**
Y	50 + 80 = **130**

5. Adding Margins

You can add row and column totals to your cross tabulation by setting the margins parameter to True.

```
# Cross tabulation with margins
crosstab_margins = pd.crosstab(df['Category'], df['Subcategory'], margins=True)
print("\nCross Tabulation with Margins:")
print(crosstab_margins)
```

6. Normalizing the Cross Tabulation

You can normalize the cross tabulation to show proportions instead of raw counts.

```
# Normalizing the cross tabulation
crosstab_normalized = pd.crosstab(df['Category'], df['Subcategory'], normalize='index')
print("\nNormalized Cross Tabulation:")
```

```
print(crosstab_normalized)
```
OUTPUT:

Subcategory	X	Y
Category		
A	0.667	0.333
B	0.333	0.667

Explanation:

For Category A:

- X: 2/3 ≈ **0.667**
- Y: 1/3 ≈ **0.333**

For Category B:

- X: 1/3 ≈ **0.333**
- Y: 2/3 ≈ **0.667**

7. Visualizing Cross Tabulation

Visualizing cross tabulation can provide additional insights. You can use libraries like matplotlib or seaborn to create heatmaps.

```
import seaborn as sns
import matplotlib.pyplot as plt
# Visualizing the cross tabulation
plt.figure(figsize=(8, 6))
sns.heatmap(crosstab_result, annot=True, cmap='Blues', fmt='g')
plt.title('Cross Tabulation Heatmap')
plt.show()
```

matplotlib:

Matplotlib is the **fundamental plotting library** in Python.

It provides functions to create charts and graphs.

Seaborn:

Seaborn is a **higher-level visualization library** built on top of Matplotlib.

It provides **prettier, statistical, and more complex visualizations** with less code.

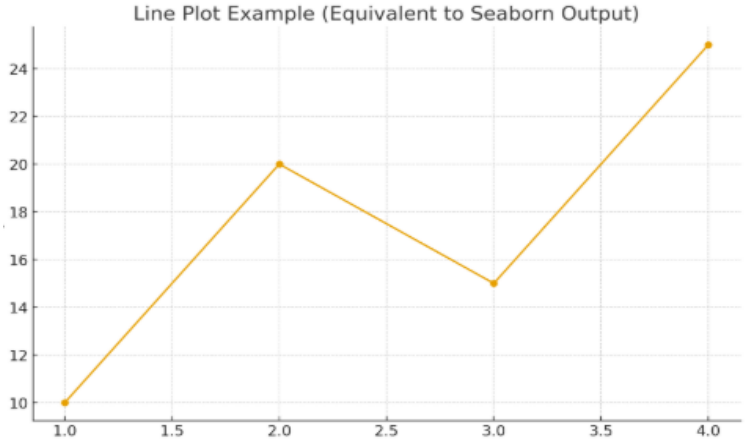
Part	Meaning
`sns.heatmap()`	Creates a heatmap plot
`crosstab_result`	The data (your cross-tab result) to visualize
`annot=True`	Shows the actual numbers inside each cell
`cmap='Blues'`	Uses different shades of blue to represent values
`fmt='g'`	Displays annotation in **general format** (no scientific notation or decimals)

Output:

Examples:

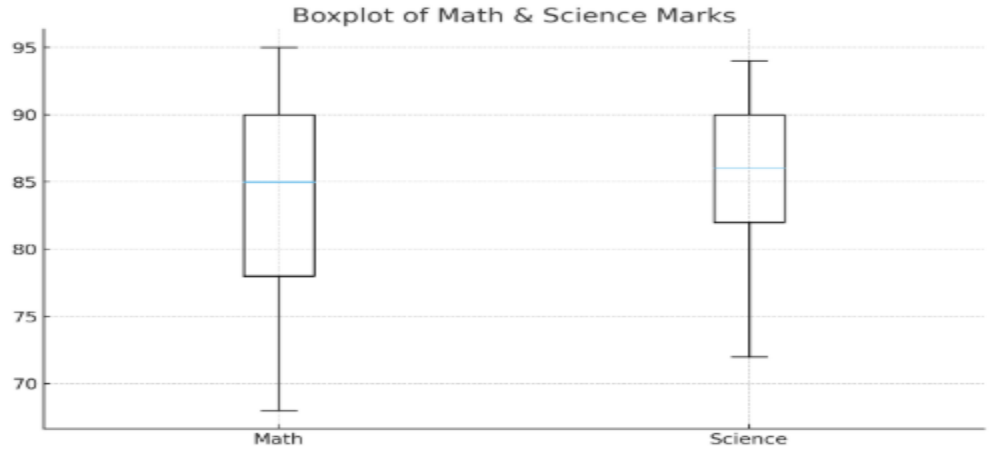
```
import seaborn as sns

sns.lineplot(x=[1,2,3,4], y=[10,20,15,25])
```

output:




```
import seaborn as sns
import pandas as pd
import matplotlib.pyplot as plt

data = {
```

```
    "Math": [78, 85, 90, 92, 70, 68, 88, 95, 80],
    "Science": [82, 88, 85, 91, 75, 72, 90, 94, 86]
}

df = pd.DataFrame(data)

sns.boxplot(data=df)
plt.title("Boxplot of Math & Science Marks")
plt.show()
```



Cross tabulation is a powerful method for summarizing and analyzing the relationship between categorical variables.

By using the crosstab() function in pandas, you can easily create contingency tables, aggregate values, and visualize relationships in your data.

Combining and Merging data sets

Combining and merging datasets is a key aspect of data preprocessing, especially when you need to analyze information from multiple sources.

Here’s a detailed guide on how to combine and merge datasets using Python, primarily with the pandas library.

Types of Data Combination

1. Concatenation: Stacking datasets on top of one another (or side by side).

2. Merging: Joining datasets based on common keys or columns.

1. Concatenation

Concatenation is used when you want to combine multiple datasets into one.

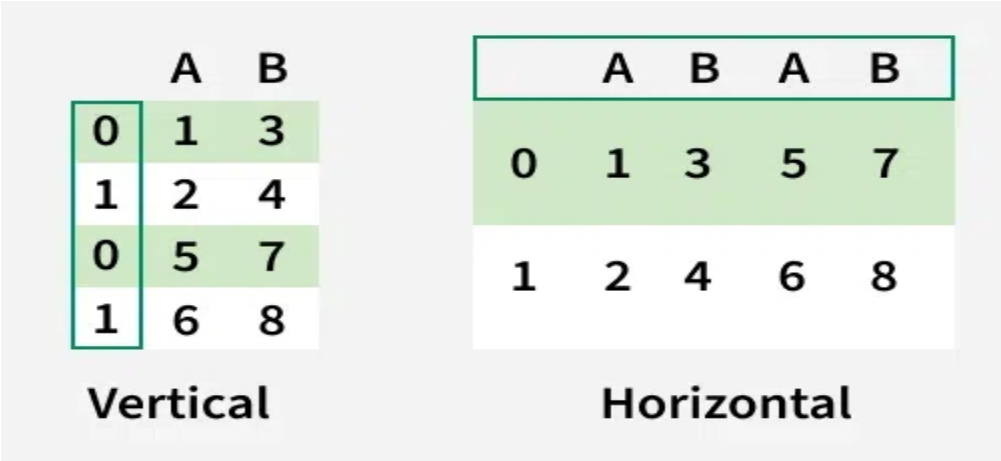
This can be done vertically (stacking rows) or horizontally (stacking columns).

Vertical Concatenation:

Use pd.concat() to combine datasets with the same columns.

```
import pandas as pd


df1 = pd.DataFrame({'A': [1, 2], 'B': [3, 4]})

df2 = pd.DataFrame({'A': [5, 6], 'B': [7, 8]})


vertical_concat = pd.concat([df1, df2], axis=0)

horizontal_concat = pd.concat([df1, df2], axis=1)

print("Vertical:")

display(vertical_concat)

print("Horizontal:")

display(horizontal_concat)
```

output:



2. Merging

Merging combines datasets based on common keys or columns. This is similar to SQL joins

(inner, outer, left, right).

Basic Merge

Use pd.merge() to merge DataFrames.

Sample DataFrames with a common key

import pandas as pd

df1 = pd.DataFrame({'id': [1, 2, 3], 'name': ['Alice', 'Bob', 'Charlie']})

df2 = pd.DataFrame({'id': [2, 3, 4], 'city': ['New York', 'London', 'Paris']})

Inner merge on 'id'

merged_df_inner = pd.merge(df1, df2, on='id', how='inner')

print("Inner Merge:\n", merged_df_inner)

output:

- Returns **only matching rows** from both DataFrames based on `id`
- Common `id` values = **2 and 3**

Result

id	name	city
2	Bob	New York
3	Charlie	London

Left merge on 'id'

merged_df_left = pd.merge(df1, df2, on='id', how='left')

print("\nLeft Merge:\n", merged_df_left)

- Returns **all rows from df1**
- Adds matching values from df2
- If no match → **NaN**

Result

id	name	city
1	Alice	NaN
2	Bob	New York
3	Charlie	London

pd.merge(df1, df2, on='id', how='right')

output:

- **Right join** keeps **all rows from df2** (right DataFrame)
- Matches rows from df1 (left DataFrame)
- If no match in df1 → `NaN`

	id	name	city
2		Bob	New York
3		Charlie	London
4		NaN	Paris

pd.merge(df1, df2, on='id', how='outer')

id	name	city
1	Alice	NaN
2	Bob	New York
3	Charlie	London
4	NaN	Paris

DataFrame.join():

The join() method is primarily used for combining DataFrames based on their indices, performing a left join by default.

```
import pandas as pd


df1 = pd.DataFrame({'A': [1, 2, 3]}, index=['x', 'y', 'z'])
df2 = pd.DataFrame({'B': [4, 5, 6]}, index=['y', 'z', 'w'])
# Default (left) join on index
joined_df = df1.join(df2)
print("Joined DataFrame:\n", joined_df)
```

Result DataFrame

index	A	B
x	1	NaN
y	2	4
z	3	5

Types of Joins

1. Inner Join: Only includes rows with keys present in both DataFrames.

2. Outer Join: Includes all rows from both DataFrames, filling in NaN for missing matches.

3. Left Join: Includes all rows from the left DataFrame and matched rows from the right.

4. Right Join: Includes all rows from the right DataFrame and matched rows from the left.