Week 3: Data Manipulation and Preparation with Pandas

Introduction

- Data manipulation and preparation is a crucial step in the data analysis process.
- It involves cleaning, transforming, and organizing raw data into a format that is suitable for analysis.

This process is significant for several reasons:

1. Data Quality Improvement: Data manipulation helps identify and address issues such as missing values, duplicates, and outliers, enhancing the overall quality of the dataset.

2. Enhanced Analysis Accuracy: Well-organized data allows for more accurate and reliable analysis. Clean and properly formatted data sets the foundation for generating meaningful insights.

Introduction

3. Feature Engineering: Data manipulation enables the creation of new features or variables, providing analysts with more relevant and insightful information for their analyses.

4. Data Exploration: Manipulating data allows analysts to explore and understand the characteristics of the dataset, making it easier to identify patterns, trends, or anomalies.

5. Preparation for Modeling: Before applying machine learning or statistical models, data manipulation is essential to ensure that the data is in a format suitable for training and evaluation.

Introduction

1. Sorting Data: the significance of sorting data for better analysis.
2. Handling Missing Data: common methods for handling missing data: dropna, fillna
3. Removing Duplicates: the importance of identifying and removing duplicate values.
4. Outliers: the methods to detect and handle outliers, such as using z-scores.
5. Merging and concatenating DataFrames: the concept of merging and joining DataFrames.
6. GroupBy Operations: the GroupBy operation for aggregating and analyzing data.
7. Apply and Map Functions: the apply and map functions for custom transformations.
8. Feature engineering: creating new features.

1. Sorting Data

Sorting

- Sorting is a fundamental operation in data analysis that involves arranging data in a specific order.

- In Pandas, the *sort_values()* function is used to sort DataFrame rows based on one or more columns.

- Sorting is crucial for better data organization, aiding in easier comprehension and analysis.

- It helps identify trends, patterns, and outliers within the dataset.

sort_values()

Sorting by one or multiple columns

- The by parameter specifies the column(s) to sort by, and ascending determines the sorting order.

```
df.sort_values(by='column_name', ascending=True)
```

- Sorting can be performed on multiple columns, prioritizing the sorting order for each.

```
df.sort_values(by=['column1', 'column2'], ascending=[True, False])
```

Example

Let’s load our dataset from the Excel file.

```
df = pd.read_excel('dataset_10.xlsx')

df
```

	Year	Age	Role	City	Education	Experience	Salary
0	2018	38	Product owner	Rotterdam	WO	17	3900
1	2018	29	Software Engineer	Eindhoven	WO	3	4400
2	2019	29	Software Engineer	Utrecht	HBO	6	6400
3	2019	26	Support Engineer	Zwolle	MBO	2	1800
4	2020	39	IT Manager	Eindhoven	WO	13	8565
5	2020	31	Software Engineer	Tilburg	HBO	4	2750
6	2020	30	Software Engineer	Utrecht	WO	5	5100
7	2021	39	Lead Developer	Randstad	MBO	15	7700
8	2021	31	Lead Developer	Groningen	WO	5	5100
9	2021	29	Microsoft Consultant	Breda	HBO	0	2300

Example

Sorting based on a single column

```
# Sorting by a single column in ascending order
df.sort_values(by='Age', ascending=True)
```

	Year	Age	Role	City	Education	Experience	Salary
3	2019	26	Support Engineer	Zwolle	MBO	2	1800
1	2018	29	Software Engineer	Eindhoven	WO	3	4400
2	2019	29	Software Engineer	Utrecht	HBO	6	6400
9	2021	29	Microsoft Consultant	Breda	HBO	0	2300
6	2020	30	Software Engineer	Utrecht	WO	5	5100
5	2020	31	Software Engineer	Tilburg	HBO	4	2750
8	2021	31	Lead Developer	Groningen	WO	5	5100
0	2018	38	Product owner	Rotterdam	WO	17	3900
4	2020	39	IT Manager	Eindhoven	WO	13	8565
7	2021	39	Lead Developer	Randstad	MBO	15	7700

Example

Sorting based on multiple columns

```
# Sorting by multiple columns with different orders
df.sort_values(by=['Year', 'Salary'], ascending=[True, False])
```

	Year	Age	Role	City	Education	Experience	Salary
1	2018	29	Software Engineer	Eindhoven	WO	3	4400
0	2018	38	Product owner	Rotterdam	WO	17	3900
2	2019	29	Software Engineer	Utrecht	HBO	6	6400
3	2019	26	Support Engineer	Zwolle	MBO	2	1800
4	2020	39	IT Manager	Eindhoven	WO	13	8565
6	2020	30	Software Engineer	Utrecht	WO	5	5100
5	2020	31	Software Engineer	Tilburg	HBO	4	2750
7	2021	39	Lead Developer	Randstad	MBO	15	7700
8	2021	31	Lead Developer	Groningen	WO	5	5100
9	2021	29	Microsoft Consultant	Breda	HBO	0	2300

2. Handling Missing Data

Handling missing data

- Identifying null values
- Filling missing values
- Techniques for imputing missing values (mean, median)
- Dropping rows/columns with missing values

Handling Missing Data

- Missing data is a common challenge in datasets that can impact the accuracy and reliability of analyses and models.

- Addressing missing data is essential for obtaining meaningful insights.

- Common techniques for handling missing data in Pandas such as:
  - Imputing with mean/median
  - Dropping missing values
  - Using advanced imputation methods

Dataframe with missing data

Dataframe can have missing value (null-values) in some columns

```
df = pd.read_excel('dataset_10_nan.xlsx')

df
```

	Year	Age	Role	City	Education	Experience	Salary
0	2018	26	IT consultant	Utrecht	WO	1	2150.0
1	2018	24	Business Analyst	Amsterdam	WO	0	2700.0
2	2019	32	Software Engineer	Utrecht	HBO	8	NaN
3	2019	31	System Engineering	Eindhoven	WO	6	4250.0
4	2020	38	Solution Architect	Randstad	NaN	12	NaN
5	2020	33	IT Manager	Randstad	WO	9	5300.0
6	2020	25	Front-end Developer	Eindhoven	HBO	2	2400.0
7	2021	39	Lead Developer	Randstad	MBO	15	7700.0
8	2021	24	Lead Developer	NaN	NaN	6	7679.0
9	2021	24	Front-end Developer	Rotterdam	HBO	1	1900.0

info()

- info() method Provides a concise summary of the DataFrame, including the count of non-null values per column.

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10 entries, 0 to 9
Data columns (total 7 columns):
 #   Column      Non-Null Count  Dtype  
---  ------      --------------  -----  
 0   Year        10 non-null     int64  
 1   Age         10 non-null     int64  
 2   Role        10 non-null     object 
 3   City        9 non-null      object 
 4   Education   8 non-null      object 
 5   Experience  10 non-null     int64  
 6   Salary      8 non-null      float64
dtypes: float64(1), int64(3), object(3)
memory usage: 692.0+ bytes
```

isnull() and notnull()

- These methods return a DataFrame of the same shape as the input, where each element is a Boolean value indicating whether the corresponding element is null.

```
df.isnull()
```

	Year	Age	Role	City	Education	Experience	Salary
0	False	False	False	False	False	False	False
1	False	False	False	False	False	False	False
2	False	False	False	False	False	False	True
3	False	False	False	False	False	False	False
4	False	False	False	False	True	False	True
5	False	False	False	False	False	False	False
6	False	False	False	False	False	False	False
7	False	False	False	False	False	False	False
8	False	False	False	True	True	False	False
9	False	False	False	False	False	False	False

sum()

- Summing the null values in each column.

```
df.isnull().sum()
```

```
Year          0
Age           0
Role          0
City          1
Education     2
Experience    0
Salary        2
dtype: int64
```

any() and all()

- Checking if any or all values in a column or DataFrame are null.

```
df.isnull().any()
```

```
Year          False
Age           False
Role          False
City           True
Education      True
Experience    False
Salary         True
dtype: bool
```

```
df.isnull().all()
```

```
Year          False
Age           False
Role          False
City          False
Education     False
Experience    False
Salary        False
dtype: bool
```

Common Methods for Handling Missing Data

Null Value Filling:

- Fills null (NaN) values with a specified value or using a specified method.

```
df['column_name'].fillna(value, inplace=True)
```

Fills null values with a specified value

In this example, we fill null values in the Education column with a specific value which is ‘MBO’.

```
df['Education'].fillna('MBO', inplace=True)

df
```

	Year	Age	Role	City	Education	Experience	Salary
0	2018	26	IT consultant	Utrecht	WO	1	2150.0
1	2018	24	Business Analyst	Amsterdam	WO	0	2700.0
2	2019	32	Software Engineer	Utrecht	HBO	8	NaN
3	2019	31	System Engineering	Eindhoven	WO	6	4250.0
4	2020	38	Solution Architect	Randstad	MBO	12	NaN
5	2020	33	IT Manager	Randstad	WO	9	5300.0
6	2020	25	Front-end Developer	Eindhoven	HBO	2	2400.0
7	2021	39	Lead Developer	Randstad	MBO	15	7700.0
8	2021	24	Lead Developer	NaN	MBO	6	7679.0
9	2021	24	Front-end Developer	Rotterdam	HBO	1	1900.0

Imputing with Mean/Median

- Replacing missing values with the mean or median of the column.

```
df['column_name'].fillna(df['column_name'].mean())
```

Example:

```
df['Salary'].fillna(df['Salary'].mean())
```

```
0    2150.000
1    2700.000
2    4259.875
3    4250.000
4    4259.875
5    5300.000
6    2400.000
7    7700.000
8    7679.000
9    1900.000
Name: Salary, dtype: float64
```

Imputing with Mean/Median

By using inplace = True, we modify the ‘Salary’ column in our dataframe.

```
df['Salary'].fillna(df['Salary'].mean(), inplace=True)

df
```

	Year	Age	Role	City	Education	Experience	Salary
0	2018	26	IT consultant	Utrecht	WO	1	2150.000
1	2018	24	Business Analyst	Amsterdam	WO	0	2700.000
2	2019	32	Software Engineer	Utrecht	HBO	8	4259.875
3	2019	31	System Engineering	Eindhoven	WO	6	4250.000
4	2020	38	Solution Architect	Randstad	MBO	12	4259.875
5	2020	33	IT Manager	Randstad	WO	9	5300.000
6	2020	25	Front-end Developer	Eindhoven	HBO	2	2400.000
7	2021	39	Lead Developer	Randstad	MBO	15	7700.000
8	2021	24	Lead Developer	NaN	MBO	6	7679.000
9	2021	24	Front-end Developer	Rotterdam	HBO	1	1900.000

Forward and Backward Fill

```
df['column_name'].fillna(method='ffill', inplace=True)  # forward fill
```

- Filling null values with the previous or next valid value.

```
df['Salary'].fillna(method='ffill', inplace=True)  # forward fill

df
```

	Year	Age	Role	City	Education	Experience	Salary
0	2018	26	IT consultant	Utrecht	WO	1	2150.0
1	2018	24	Business Analyst	Amsterdam	WO	0	2700.0
2	2019	32	Software Engineer	Utrecht	HBO	8	2700.0
3	2019	31	System Engineering	Eindhoven	WO	6	4250.0
4	2020	38	Solution Architect	Randstad	NaN	12	4250.0
5	2020	33	IT Manager	Randstad	WO	9	5300.0
6	2020	25	Front-end Developer	Eindhoven	HBO	2	2400.0
7	2021	39	Lead Developer	Randstad	MBO	15	7700.0
8	2021	24	Lead Developer	NaN	NaN	6	7679.0
9	2021	24	Front-end Developer	Rotterdam	HBO	1	1900.0

Dropping Null Values

- Removing rows or columns with missing data.

```
df.dropna(inplace=True)

df
```

	Year	Age	Role	City	Education	Experience	Salary
0	2018	26	IT consultant	Utrecht	WO	1	2150.000
1	2018	24	Business Analyst	Amsterdam	WO	0	2700.000
2	2019	32	Software Engineer	Utrecht	HBO	8	4259.875
3	2019	31	System Engineering	Eindhoven	WO	6	4250.000
4	2020	38	Solution Architect	Randstad	MBO	12	4259.875
5	2020	33	IT Manager	Randstad	WO	9	5300.000
6	2020	25	Front-end Developer	Eindhoven	HBO	2	2400.000
7	2021	39	Lead Developer	Randstad	MBO	15	7700.000
9	2021	24	Front-end Developer	Rotterdam	HBO	1	1900.000

3. Removing Duplicates

Identifying Duplicates

- Use the duplicated() method to identify duplicate rows.

```
# Identifying duplicate rows
duplicate_rows = df[df.duplicated()]
```

```
# Identifying duplicate rows
duplicate_rows = df[df.duplicated()]

duplicate_rows
```

Year	Age	Role	City	Education	Experience	Salary

Dataframe with duplicated rows

I duplicated a row manually in our Excel file.

```
df = pd.read_excel('dataset_10_nan_dup.xlsx')

df
```

	Year	Age	Role	City	Education	Experience	Salary
0	2018	26	IT consultant	Utrecht	WO	1	2150.0
1	2018	24	Business Analyst	Amsterdam	WO	0	2700.0
2	2019	32	Software Engineer	Utrecht	HBO	8	NaN
3	2019	31	System Engineering	Eindhoven	WO	6	4250.0
4	2020	38	Solution Architect	Randstad	NaN	12	NaN
5	2020	33	IT Manager	Randstad	WO	9	5300.0
6	2020	25	Front-end Developer	Eindhoven	HBO	2	2400.0
7	2020	25	Front-end Developer	Eindhoven	HBO	2	2400.0
8	2021	39	Lead Developer	Randstad	MBO	15	7700.0
9	2021	24	Lead Developer	NaN	NaN	6	7679.0
10	2021	24	Front-end Developer	Rotterdam	HBO	1	1900.0

Identifying Duplicates

Now if I use duplicated() method, it will identify duplicate rows in dataframe.

```
# Identifying duplicate rows
duplicate_rows = df[df.duplicated()]

duplicate_rows
```

	Year	Age	Role	City	Education	Experience	Salary
7	2020	25	Front-end Developer	Eindhoven	HBO	2	2400.0

Removing Duplicates

- Use drop_duplicates() to remove duplicate rows.

```
# without inplace = True we need to assign the returned dataframe to a new variable
# df_no_duplicates = df.drop_duplicates()

df.drop_duplicates(inplace=True)

df
```

	Year	Age	Role	City	Education	Experience	Salary
0	2018	26	IT consultant	Utrecht	WO	1	2150.0
1	2018	24	Business Analyst	Amsterdam	WO	0	2700.0
2	2019	32	Software Engineer	Utrecht	HBO	8	NaN
3	2019	31	System Engineering	Eindhoven	WO	6	4250.0
4	2020	38	Solution Architect	Randstad	NaN	12	NaN
5	2020	33	IT Manager	Randstad	WO	9	5300.0
6	2020	25	Front-end Developer	Eindhoven	HBO	2	2400.0
8	2021	39	Lead Developer	Randstad	MBO	15	7700.0
9	2021	24	Lead Developer	NaN	NaN	6	7679.0
10	2021	24	Front-end Developer	Rotterdam	HBO	1	1900.0

Remove duplicates and reset the index

- When we drop the rows from DataFrame, by default, it keeps the original row index as is.

- If we need to reset the index of the resultant DataFrame, we can do that using the ignore_index parameter of drop_duplicate() method.

  - If ignore_index=True, it reset the row labels of resultant DataFrame to 0, 1, …, n – 1.
  - If ignore_index=False it does not change the original row index. By default, it is False.

Remove duplicates and reset the index

- Using ignore_index=True, reset the row labels.

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

df
```

	Year	Age	Role	City	Education	Experience	Salary
0	2018	26	IT consultant	Utrecht	WO	1	2150.0
1	2018	24	Business Analyst	Amsterdam	WO	0	2700.0
2	2019	32	Software Engineer	Utrecht	HBO	8	NaN
3	2019	31	System Engineering	Eindhoven	WO	6	4250.0
4	2020	38	Solution Architect	Randstad	NaN	12	NaN
5	2020	33	IT Manager	Randstad	WO	9	5300.0
6	2020	25	Front-end Developer	Eindhoven	HBO	2	2400.0
7	2021	39	Lead Developer	Randstad	MBO	15	7700.0
8	2021	24	Lead Developer	NaN	NaN	6	7679.0
9	2021	24	Front-end Developer	Rotterdam	HBO	1	1900.0

Remove duplicates and reset the index

- You might want to reset the indices. The reset_index() method can be used for this purpose.

```
df.drop_duplicates(inplace=True)
# drop=True means do not try to insert index into dataframe columns. This resets the index to the default integer index.
df.reset_index(drop=True, inplace=True)

df
```

	Year	Age	Role	City	Education	Experience	Salary
0	2018	26	IT consultant	Utrecht	WO	1	2150.0
1	2018	24	Business Analyst	Amsterdam	WO	0	2700.0
2	2019	32	Software Engineer	Utrecht	HBO	8	NaN
3	2019	31	System Engineering	Eindhoven	WO	6	4250.0
4	2020	38	Solution Architect	Randstad	NaN	12	NaN
5	2020	33	IT Manager	Randstad	WO	9	5300.0
6	2020	25	Front-end Developer	Eindhoven	HBO	2	2400.0
7	2021	39	Lead Developer	Randstad	MBO	15	7700.0
8	2021	24	Lead Developer	NaN	NaN	6	7679.0
9	2021	24	Front-end Developer	Rotterdam	HBO	1	1900.0

4. Outliers

Outliers

What Are Outliers?

- Outliers are data points that are significantly different from the majority of the other data points in a dataset that can affect the accuracy of analyses.
- They are unusual or exceptional values that stand out.
- Detecting and addressing outliers is important for obtaining meaningful insights from data.

Example:

Imagine you have a dataset of people's ages. Most people may be in their 20s, 30s, or 40s, but if there's a data point indicating someone is 150 years old, that could be an outlier.

Why Do Outliers Matter?

Impact on Analysis:

- Outliers can distort statistical analyses and machine learning models.

- They can heavily influence average values, making them less representative of the majority.

Example:

If you're calculating the average salary in a company and there's an outlier with an extremely high salary, the average might not reflect the typical salary for most employees.

Detecting Outliers

- Common methods for detecting outliers include using statistical measures like z-scores or the Interquartile Range (IQR).

- These methods help identify values that deviate significantly from the norm.

- Once identified, outliers can be handled by removing them, transforming them, or applying statistical methods to mitigate their impact.

Example: If most monthly salaries in a dataset are between €2000 and €8000, a salary of €70,000 might be flagged as an outlier.

Data Transformation

- Data transformation involves modifying the structure or values of data to make it more suitable for analysis or modeling.

- Common data transformation techniques include:
  - Normalization
  - Scaling
  - Applying custom functions
  - Binning and Discretization

Normalization and Scaling

- Normalization:

Adjusting values to a common scale, often between 0 and 1.

```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
df['column_name_normalized'] = scaler.fit_transform(df[['column_name']])
```

- Standard Scaling (Z-score normalization):

Transforming values to have a mean of 0 and a standard deviation of 1.

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
df['column_name_scaled'] = scaler.fit_transform(df[['column_name']])
```

Applying Custom Functions

- Using custom functions to transform data based on specific requirements.

```
# Example custom function: Square root transformation
def sqrt_transform(x):
    return x ** 0.5

df['column_name_sqrt'] = df['column_name'].apply(sqrt_transform)
```

Binning and Discretization

- Grouping continuous data into discrete bins or intervals.

```python
# Example binning into three categories
bins = [0, 25, 50, 100]
labels = ['Low', 'Medium', 'High']
df['column_name_bin'] = pd.cut(df['column_name'], bins=bins, labels=labels)
```

5. Merging and concatenating DataFrames

Merging and Concatenating DataFrames

- Merging and Concatenating are essential operations when working with multiple datasets.
- Combining information from different DataFrames allows for more comprehensive analysis.

Merging dataframe:

- Merging allows us to combine data from different sources or based on common columns.
- Different types of merges are available, such as:
  - Inner
  - Outer
  - Left
  - Right

Types of Merges

Inner: Retains only the common rows between two DataFrames.

```
merged_inner = pd.merge(df1, df2, on='common_column', how='inner')
```

Outer: Includes all rows from both DataFrames, filling in missing values with NaN.

```
merged_outer = pd.merge(df1, df2, on='common_column', how='outer')
```

Left: Includes all rows from the left DataFrame and matching rows from the right DataFrame.

```
merged_left = pd.merge(df1, df2, on='common_column', how='left')
```

Right: Includes all rows from the right DataFrame and matching rows from the left DataFrame.

```
merged_right = pd.merge(df1, df2, on='common_column', how='right')
```

Examples of merging types

- Let's consider two following sample DataFrames:

```
# Sample DataFrame 1
data1 = {'ID': [1, 2, 3],
         'Name': ['Alice', 'Bob', 'Charlie']}
df1 = pd.DataFrame(data1)
df1
```

	ID	Name
0	1	Alice
1	2	Bob
2	3	Charlie

```
# Sample DataFrame 2
data2 = {'ID': [2, 3, 4],
         'Age': [25, 30, 22]}
df2 = pd.DataFrame(data2)
df2
```

	ID	Age
0	2	25
1	3	30
2	4	22

Example

Inner:

```
merged_inner = pd.merge(df1, df2, on='ID', how='inner')

merged_inner
```

	ID	Name	Age
0	2	Bob	25
1	3	Charlie	30

Outer :

```
merged_outer = pd.merge(df1, df2, on='ID', how='outer')

merged_outer
```

	ID	Name	Age
0	1	Alice	NaN
1	2	Bob	25.0
2	3	Charlie	30.0
3	4	NaN	22.0

Example

Left:

```
merged_left = pd.merge(df1, df2, on='ID', how='left')

merged_left
```

	ID	Name	Age
0	1	Alice	NaN
1	2	Bob	25.0
2	3	Charlie	30.0

Right:

```
merged_right = pd.merge(df1, df2, on='ID', how='right')

merged_right
```

	ID	Name	Age
0	2	Bob	25
1	3	Charlie	30
2	4	NaN	22

Merging on Multiple Columns

```
merged_multi = pd.merge(df1, df2, on=['column1', 'column2'], how='inner')
```

- Let's add a new column called City to our dataframes.

```
# Adding another column to df1
df1['City'] = ['NY', 'SF', 'LA']

# Adding another column to df2
df2['City'] = ['SF', 'LA', 'TX']
```

```
# Merging on multiple columns
merged_multi = pd.merge(df1, df2, on=['ID', 'City'], how='inner')

merged_multi
```

	ID	Name	City	Age
0	2	Bob	SF	25
1	3	Charlie	LA	30

Handling Duplicate Column Names

- Addressing the issue of duplicate column names in merged DataFrames:

```
merged_duplicates = pd.merge(df1, df2, on='common_column', suffixes=('_left', '_right'))
```

```
# Adding another column with the same name to df2
df2['Name'] = ['Charlie', 'Dave', 'Eva']

# Merging with duplicate column names
merged_duplicates = pd.merge(df1, df2, on='ID', suffixes=('_left', '_right'))

merged_duplicates
```

	ID	Name_left	Age	Name_right
0	2	Bob	25	Charlie
1	3	Charlie	30	Dave

Combining DataFrames with Concatenate

- Concatenating DataFrames is useful when combining information from multiple sources.

```
# Creating DataFrames from dictionaries
data1 = {'Name': ['Alice', 'Bob', 'Charlie'],
        'Age': [25, 30, 35],
        'City': ['New York', 'San Francisco', 'Los Angeles']}

data2 = {'Name': ['Eva', 'John'],
         'Age': [28, 32],
         'City': ['Paris', 'London']}

df1 = pd.DataFrame(data1)
df2 = pd.DataFrame(data2)

# Concatenating DataFrames row-wise
concatenated_df = pd.concat([df1, df2], ignore_index=True)

concatenated_df
```

	Name	Age	City
0	Alice	25	New York
1	Bob	30	San Francisco
2	Charlie	35	Los Angeles
3	Eva	28	Paris
4	John	32	London

6. GroupBy Operations

Grouping and aggregation

- Grouping data based on specific columns
- Aggregating data (groupby)
- Applying aggregation functions

A new dataframe to work with groupby

- Let's use more meaningful column names for the sample DataFrame and perform GroupBy operations:

```
data = {'City': ['NY', 'SF', 'NY', 'SF', 'NY', 'SF', 'NY', 'SF'],
        'Product': ['A', 'B', 'A', 'B', 'A', 'B', 'A', 'B'],
        'Sales': [10, 15, 20, 25, 30, 35, 40, 45]}

df = pd.DataFrame(data)

df
```

	City	Product	Sales
0	NY	A	10
1	SF	B	15
2	NY	A	20
3	SF	B	25
4	NY	A	30
5	SF	B	35
6	NY	A	40
7	SF	B	45

GroupBy

- GroupBy is a powerful operation in pandas for:
  - Splitting
  - Applying
  - Combining data
- It allows you to group data based on one or more criteria. Basic Syntax:

```
grouped_data = df.groupby('Category')
```

```
grouped_by_city = df.groupby('City')

grouped_by_city
```

```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x000002434EF47DD0>
```

```
grouped_by_city_sales = df.groupby('City')['Sales']

grouped_by_city_sales
```

```
<pandas.core.groupby.generic.SeriesGroupBy object at 0x000002434EF9E410>
```

Common Aggregation Functions

groupby() function is applied to a DataFrame, followed by an aggregation function.

Sum:

- Groups and calculates the sum for each group.

```
sum_by_category = df.groupby('Category')['Value'].sum()
```

Example:

Sum of Sales by City

```
sum_by_city = df.groupby('City')['Sales'].sum()

sum_by_city
```

```
City
NY    100
SF    120
Name: Sales, dtype: int64
```

Common Aggregation Functions

Mean (Average):

- Calculates the mean for each group.

```
avg_by_category = df.groupby('Category')['Value'].mean()
```

Example:

Average Sales by City

```
avg_by_city = df.groupby('City')['Sales'].mean()

avg_by_city
```

```
City
NY    25.0
SF    30.0
Name: Sales, dtype: float64
```

Common Aggregation Functions

Count:

- Counts the number of occurrences in each group.

```
count_by_category = df.groupby('Category')['Value'].count()
```

Example:

Count of Sales by City

```
count_by_city = df.groupby('City')['Sales'].count()

count_by_city
```

```
City
NY    4
SF    4
Name: Sales, dtype: int64
```

Common Aggregation Functions

Min and Max:

Finds the minimum and maximum values in each group.

```
min_by_category = df.groupby('Category')['Value'].min()
max_by_category = df.groupby('Category')['Value'].max()
```

Example:

Min and Max Sales by City:

```
min_by_city = df.groupby('City')['Sales'].min()
max_by_city = df.groupby('City')['Sales'].max()

min_by_city
```

```
City
NY    10
SF    15
Name: Sales, dtype: int64
```

Multiple Grouping Columns

- You can group by multiple columns to create hierarchical groupings.

```
grouped_multiple = df.groupby(['Category', 'Subcategory'])['Value'].mean()
```

Example: Average Sales by City and Product:

```
grouped_multiple = df.groupby(['City', 'Product'])['Sales'].mean()

grouped_multiple
```

```
City  Product
NY    A          25.0
SF    B          30.0
Name: Sales, dtype: float64
```

GroupBy and Agg Function

- Using the agg() function to perform multiple aggregation operations simultaneously.

```
agg_operations = df.groupby('Category').agg({'Value': ['sum', 'mean', 'count']})
```

Example:

Aggregation Operations by City

```
agg_operations = df.groupby('City').agg({'Sales': ['sum', 'mean', 'count']})

agg_operations
```

	Sales		
	sum	mean	count
City			
NY	100	25.0	4
SF	120	30.0	4

7. Apply and Map Functions

apply() function in Pandas

- The apply() and map() functions in Pandas are powerful tools for manipulating and transforming data in DataFrames.

- The apply() function is both a DataFrame method and a Series method in Pandas.
- Its behavior depends on the axis along which it is applied:
- It can be applied to both rows and columns, and it's particularly useful when you want to perform a custom operation on each element or row/column of a DataFrame.

apply() function in Pandas

DataFrame Method:

When you apply apply() on a DataFrame, you can use it to operate on either rows or columns.

```
DataFrame.apply(func, axis=0)
```

- func: The function to apply
- axis: The axis along which the function will be applied (0 for columns, 1 for rows)

➢ By default (axis=0), it applies the function to each column. In this case, the function receives a Series representing a column.

➢ When axis=1, it applies the function to each row. In this case, the function receives a Series representing a row.

```
df.apply(my_function, axis=0)  # Apply along columns
df.apply(my_function, axis=1)  # Apply along rows
```

apply() function in Pandas

Series Method:

- When you apply apply() on a Series, it operates element-wise on each element of the Series.
- The function provided to apply() for a Series will receive each individual element of the Series.

```
series.apply(my_function)  # Apply element-wise on each element of the Series
```

- So, depending on whether you apply it to a DataFrame or a Series, and on the specified axis, apply() can be used for various purposes, making it a versatile tool in data manipulation with Pandas.

Example

➢ Let's say we have a DataFrame containing student scores, and we want to calculate the percentage for each student.

```
data = {'Student_ID': [1, 2, 3, 4, 5],
        'Math_Score': [85, 90, 78, 92, 88],
        'English_Score': [75, 80, 85, 88, 92]}

df = pd.DataFrame(data)

# Define a function to calculate percentage
def calculate_percentage(row):
    total_score = row['Math_Score'] + row['English_Score']
    return (total_score / 200) * 100

# Apply the function to create a new 'Percentage' column
df['Percentage'] = df.apply(calculate_percentage, axis=1)

df
```

	Student_ID	Math_Score	English_Score	Percentage
0	1	85	75	80.0
1	2	90	80	85.0
2	3	78	85	81.5
3	4	92	88	90.0
4	5	88	92	90.0

map() function in Pandas

- The map function in Pandas is used for substituting each value in a Series with another value.

- It's often used to replace values in a column based on a mapping provided.

```
Series.map(arg, na_action=None)
```

- arg: A function, a dict, or a Series
- na_action: How to handle missing values (default is None)

Example

- Let's consider a scenario where we want to map numerical scores to letter grades using the map function:

```python
# Sample DataFrame
data = {'Student_ID': [1, 2, 3, 4, 5],
        'Math_Score': [85, 90, 78, 92, 88]}

df = pd.DataFrame(data)

# Define a mapping for letter grades with a function
def map_letter_grade(score):
    if score >= 90:
        return 'A'
    elif score >= 80:
        return 'B'
    elif score >= 70:
        return 'C'
    elif score >= 60:
        return 'D'
    else:
        return 'F'

# Map the letter grades using the 'Math_Score' column
df['Letter_Grade'] = df['Math_Score'].map(map_letter_grade)

df
```

	Student_ID	Math_Score	Letter_Grade
0	1	85	B
1	2	90	A
2	3	78	C
3	4	92	A
4	5	88	B

8. Feature Engineering

Feature Engineering

- Creating new meaningful features from existing ones to improve analysis.

For example:

- Data Scaling and Normalization:

Scale numerical features to the same range for fair comparisons.

- Handling Categorical Data:

Encode categorical variables for analysis.

Creating New Features

- Let's use a different DataFrame to create new features. These examples demonstrate how you can create new features based on different scenarios using a DataFrame related to student scores.

```
data = {'Student_ID': [1, 2, 3, 4, 5],
        'Math_Score': [85, 90, 78, 92, 88],
        'English_Score': [75, 80, 85, 88, 92],
        'Gender': ['Male', 'Female', 'Male', 'Male', 'Female'],
        'Class': ['A', 'B', 'A', 'B', 'B']}

df = pd.DataFrame(data)

df
```

	Student_ID	Math_Score	English_Score	Gender	Class
0	1	85	75	Male	A
1	2	90	80	Female	B
2	3	78	85	Male	A
3	4	92	88	Male	B
4	5	88	92	Female	B

Average Score

- Calculate the average score for each student.

```
df['Average_Score'] = df[['Math_Score', 'English_Score']].mean(axis=1)

df
```

	Student_ID	Math_Score	English_Score	Gender	Class	Average_Score
0	1	85	75	Male	A	80.0
1	2	90	80	Female	B	85.0
2	3	78	85	Male	A	81.5
3	4	92	88	Male	B	90.0
4	5	88	92	Female	B	90.0

Pass/Fail Status

- Determine if a student passed or failed based on a threshold.

```
pass_threshold = 85
df['Pass_Status'] = df['Average_Score'].apply(lambda x: 'Pass' if x >= pass_threshold else 'Fail')

df
```

	Student_ID	Math_Score	English_Score	Gender	Class	Average_Score	Pass_Status
0	1	85	75	Male	A	80.0	Fail
1	2	90	80	Female	B	85.0	Pass
2	3	78	85	Male	A	81.5	Fail
3	4	92	88	Male	B	90.0	Pass
4	5	88	92	Female	B	90.0	Pass

Gender Encoding

- Encode the 'Gender' column into numerical values.

```
df['Gender_Code'] = df['Gender'].map({'Male': 0, 'Female': 1})

df
```

	Student_ID	Math_Score	English_Score	Gender	Class	Average_Score	Pass_Status	Gender_Code
0	1	85	75	Male	A	80.0	Fail	0
1	2	90	80	Female	B	85.0	Pass	1
2	3	78	85	Male	A	81.5	Fail	0
3	4	92	88	Male	B	90.0	Pass	0
4	5	88	92	Female	B	90.0	Pass	1