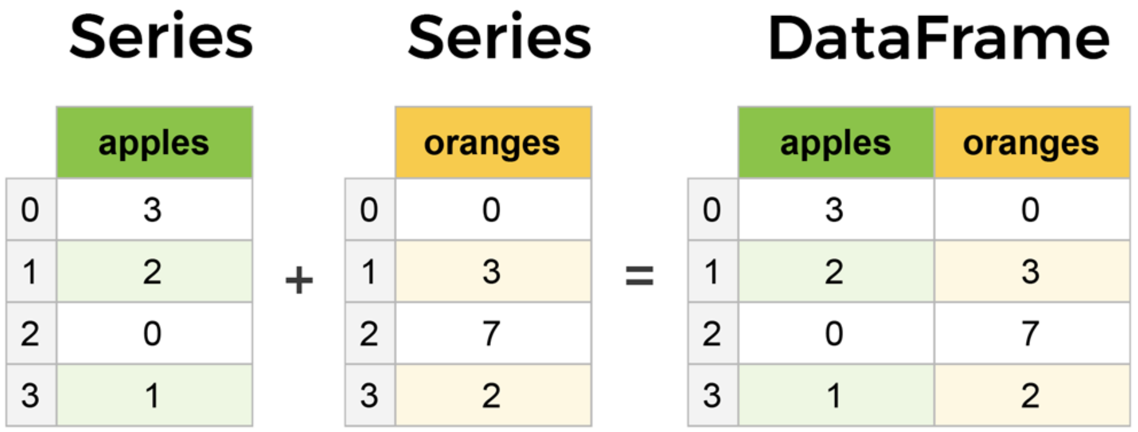
Week 2: Working with Dataframes



Recap of DataFrames

Before we continue, let's quickly review the key concepts we covered last week:

- NumPy and Pandas, two powerful data science libraries.

- A DataFrame is a two-dimensional, tabular data structure with labeled axes, consisting of rows and columns.

- In simple terms, a DataFrame is like an Excel spreadsheet—a table with rows and columns.

- Each column can have a different data type, providing flexibility in representing and analyzing data.

Why use DataFrames?

Structured Representation:

DataFrames provide a structured and organized way to represent real-world data.

Efficient Operations:

Operations on DataFrames are vectorized, enabling efficient manipulation and analysis without the need for explicit looping. The vectorized operations make them a powerful tool for handling and analyzing structured data.

Relationship with Pandas Series

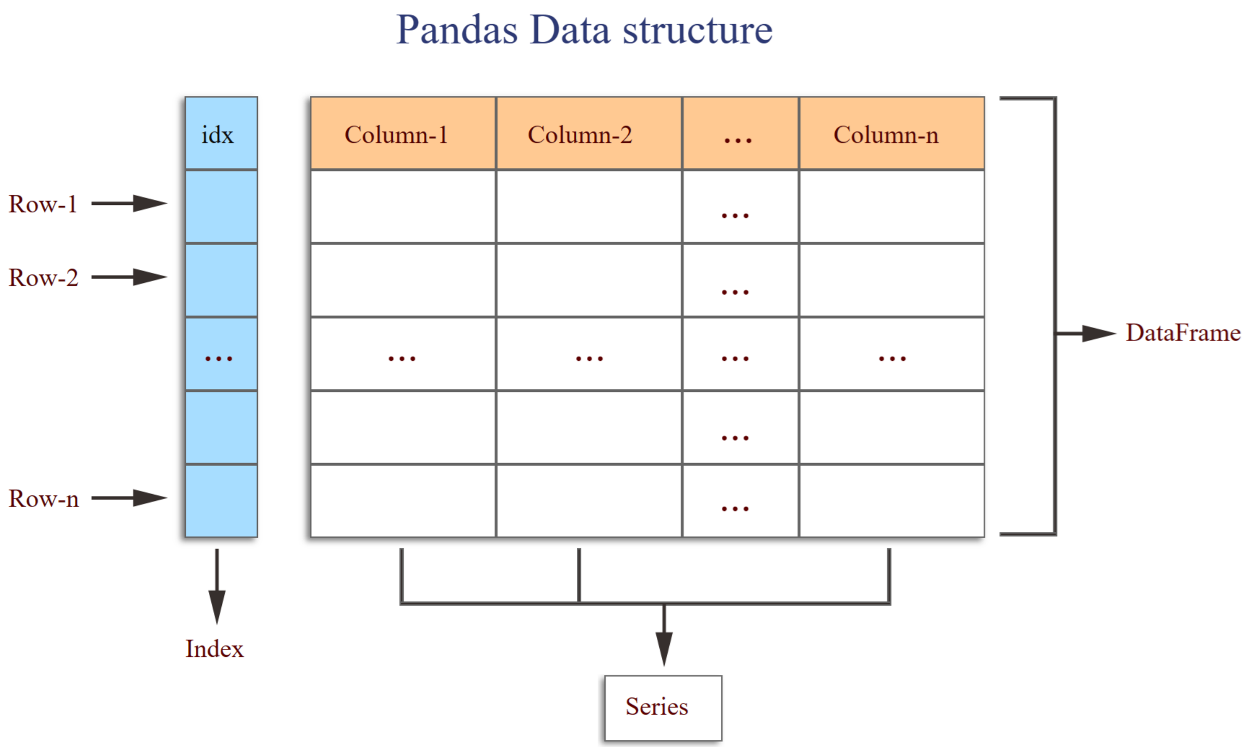
- It's essential to recognize that each column within a DataFrame is a Pandas Series.

- This relationship enables us to apply operations at both the DataFrame and Series levels, providing flexibility in data manipulation.

- Columns can be extracted as individual Series, allowing for more focused operations.

Relationship with Pandas Series

Pandas Series and DataFrame



Creating DataFrames from Different Sources

- Creating a DataFrame is the first step in data analysis.

- Pandas provides flexibility in creating DataFrames from a variety of sources.

- There are multiple methods to create DataFrames, providing flexibility in handling different types of data sources:
  - Creating from NumPy arrays
  - Creating from a Python dictionary
  - Creating from a file

Creating a DataFrame from NumPy Arrays

- This is particularly useful when we already have data in the form of arrays and want to structure it into a DataFrame.

In [7]:

```
import pandas as pd
import numpy as np

# Creating a DataFrame from NumPy arrays
data = np.array([[1, 2, 3], [4, 5, 6]])
df = pd.DataFrame(data, columns=['A', 'B', 'C'])

df
```

Out[7]:

	A	B	C
0	1	2	3
1	4	5	6

Creating a DataFrame from a Dictionary

➢ We create a DataFrame from lists using a dictionary, where keys represent column names and values are lists of data.

Example:

In [1]:

```python
import pandas as pd

# Creating a DataFrame from a dictionary
data = {'Name': ['Alice', 'Bob', 'Charlie'],
        'Age': [25, 30, 35],
        'City': ['New York', 'San Francisco', 'Los Angeles']}

df = pd.DataFrame(data)

df
```

Out[1]:

	Name	Age	City
0	Alice	25	New York
1	Bob	30	San Francisco
2	Charlie	35	Los Angeles

Creating a DataFrame from a file

Reading data from various file sources:

- From CSV Files
- From Excel Files
- From JSON Files
- From SQL Databases
- From Web Scraping

Creating DataFrames from CSV Files

- Reading data from CSV files is a common practice, allowing you to create a DataFrame from external data sources.
- We use the `read_csv` method to create a DataFrame from data stored in a CSV file.

Example:

```
# Reading data from a CSV file
csv_file_path = 'path/to/your/file.csv'
df_from_csv = pd.read_csv(csv_file_path)
```

Creating DataFrames from Excel Files

- Reading data from Excel files is seamless with Pandas, making it easy to create a DataFrame from spreadsheet data.
- We use the `read_excel` method to create a DataFrame from data stored in an Excel file.

Example:

```
# Reading data from an Excel file
excel_file_path = 'path/to/your/file.xlsx'
df_from_excel = pd.read_excel(excel_file_path)
```

Creating DataFrames from JSON Files

- JSON (JavaScript Object Notation) is a common data format, and Pandas supports creating DataFrames from JSON data.
- Reading data from JSON files is straightforward with Pandas, allowing you to create a DataFrame from structured JSON data.
- We use the `read_json` method to create a DataFrame from data stored in a JSON file.

Example:

```
# Reading data from a JSON file
json_file_path = 'path/to/your/file.json'
df_from_json = pd.read_json(json_file_path)
```

Creating DataFrames from SQL Databases

- Connecting to SQL databases allows for creating DataFrames from tables or query results, providing seamless integration with structured data stored in databases.
- We use the `read_sql_query` method to create a DataFrame from data retrieved from an SQL database.

Example:

```
# Reading data from an SQL database
import sqlite3

conn = sqlite3.connect('your_database.db')
query = 'SELECT * FROM your_table'
df_from_sql = pd.read_sql_query(query, conn)
```

Creating DataFrames from Web Scraping

- Web scraping enables you to extract data from websites and create DataFrames, expanding the sources of information.

- We will need other libraries like requests and BeautifulSoup.

DataFrame we will work with

DataFrame from a Dictionary

In [1]:

```
import pandas as pd

# Creating a DataFrame from a dictionary
data = {'Name': ['Alice', 'Bob', 'Charlie'],
        'Age': [25, 30, 35],
        'City': ['New York', 'San Francisco', 'Los Angeles']}

df = pd.DataFrame(data)

df
```

Out[1]:

	Name	Age	City
0	Alice	25	New York
1	Bob	30	San Francisco
2	Charlie	35	Los Angeles

Basic Operations on DataFrames

- Basic data exploration (head, tail, summary statistics)
- Accessing rows and columns
- Operations on columns
- Operations on Rows
- Filtering data:
  - Boolean indexing
  - Using conditions

Basic exploration and viewing data

➢ Last week we tried these methods and attributes on our dataframe:

- head() and tail() methods
- info() method
- describe() method
- shape attribute
- columns attribute

Example

Basic methods and attributes on our dataframe

```
df.shape
```

```
(3, 3)
```

```
df.columns
```

```
Index(['Name', 'Age', 'City'], dtype='object')
```

```
df.head()
```

	Name	Age	City
0	Alice	25	New York
1	Bob	30	San Francisco
2	Charlie	35	Los Angeles

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3 entries, 0 to 2
Data columns (total 3 columns):
 #   Column  Non-Null Count  Dtype 
---  ------  --------------  ----- 
 0   Name    3 non-null      object
 1   Age     3 non-null      int64 
 2   City    3 non-null      object
dtypes: int64(1), object(2)
memory usage: 204.0+ bytes
```

Accessing rows and columns

Now that we have a DataFrame, let's explore how we can access and retrieve specific data.

- Accessing data within a DataFrame involves retrieving specific subsets based on column names and row indices.

- This allows us to focus on relevant portions of the data.

- By specifying a column name, we can retrieve all values in that column. Additionally, using row indices allows us to fetch specific rows.

Accessing rows and columns

- Selecting Columns
- Using *loc* and *iloc* methods
- Selecting Rows
- Slicing and Indexing
- Conditional Selection

Accessing Data from Specific Column

- By using a single column name, we can retrieve all values in that column.

```
# Accessing data from specific column
# Using column name
ages = df['Age']

# When we are trying to access one column, we will get a Series
ages
```

```
0    25
1    30
2    35
Name: Age, dtype: int64
```

Accessing Data from Specific Columns

- We can extract data from specific columns by specifying the column names.

```
# Accessing data from specific columns
selected_columns = df[['Name', 'Age']]

# When we are trying to access more than one column, we will get a dataframe
selected_columns
```

	Name	Age
0	Alice	25
1	Bob	30
2	Charlie	35

Specific Column or Columns

- Series or DataFrame based on selected column or columns

```
# Accessing data from specific column
name_column = df['Name']

# Accessing data from specific columns
selected_columns = df[['Name', 'Age']]

print('The name_column will be Pandas Series')
print(name_column)
print()

print('The selected_columns will be Pandas DataFrame')
print(selected_columns)
print()

print(type(name_column))
print(type(selected_columns))
```

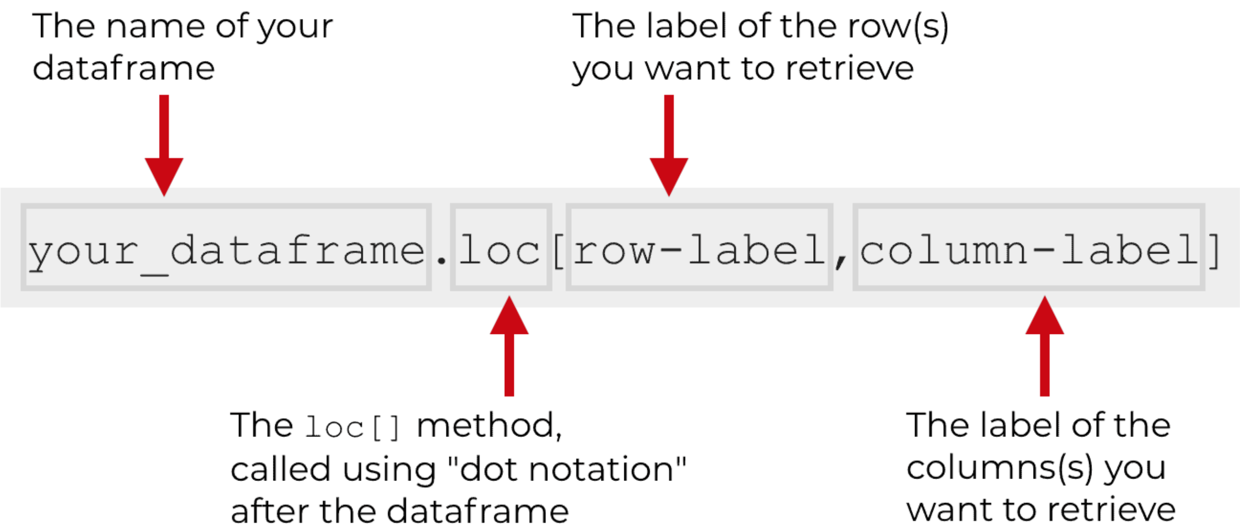
```
The name_column will be Pandas Series
0      Alice
1        Bob
2    Charlie
Name: Name, dtype: object

The selected_columns will be Pandas DataFrame
      Name  Age
0    Alice   25
1      Bob   30
2  Charlie   35

<class 'pandas.core.series.Series'>
<class 'pandas.core.frame.DataFrame'>
```

Selecting Data with *loc* and *iloc*

- Efficiently selecting specific rows and columns is essential in data analysis.
- Let's explore two powerful methods for this: loc and iloc.
- The loc and iloc methods in Pandas are used to slice a data set.
- The loc method is primarily used for Label-Based Indexing, while iloc is mainly used for Integer/Position-Based Indexing.

Understanding loc (Label-Based Indexing)

- The loc method provides label-based indexing, enabling us to select data based on row and column labels.



Selecting a Specific Row by Label

- Using loc to select the first row of the DataFrame based on the label.

Example:

```
# Selecting a specific row by label
selected_row = df.loc[0]

print(selected_row)
```

```
Name        Alice
Age            25
City     New York
Name: 0, dtype: object
```

Selecting Specific Rows and Columns by Labels

- We can also use loc to select specific rows and columns by providing lists of labels.

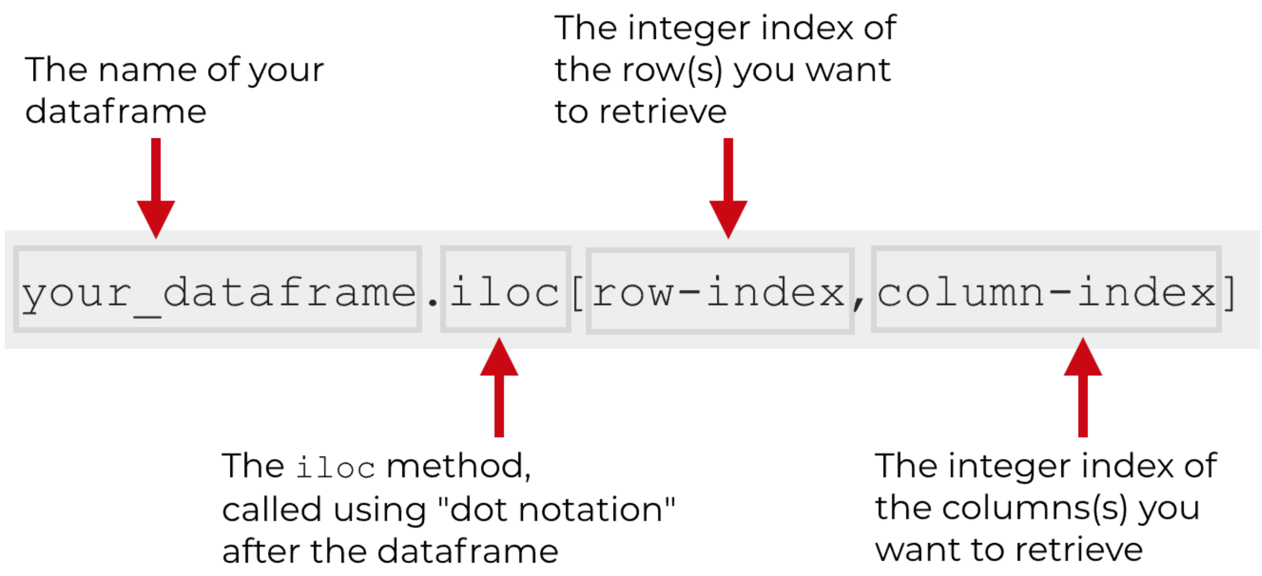
Example:

```
# Selecting specific rows and columns by labels
selected_data = df.loc[[0, 2], ['Name', 'Age']]

print(selected_data)
```

```
      Name  Age
0    Alice   25
2  Charlie   35
```

Understanding iloc (Position-Based Indexing)

➢ The iloc method provides integer-location based indexing, enabling us to select data based on integer positions.



Selecting a Specific Row by Position

- Using iloc to select the first row of the DataFrame based on its position.

Example:

```
# Selecting a specific row by position
selected_row_pos = df.iloc[0]

print(selected_row_pos)
```

```
Name       Alice
Age           25
City    New York
Name: 0, dtype: object
```

Selecting Specific Rows and Columns by Positions

- We can also use iloc to select specific rows and columns by providing lists of integer positions.

Example:

```
# Selecting specific rows and columns by positions
selected_data_pos = df.iloc[[0, 2], [0, 1]]

print(selected_data_pos)
```

```
      Name  Age
0    Alice   25
2  Charlie   35
```

Comparing loc and iloc

- Understanding the differences between loc and iloc is crucial for precise data selection:

  - loc uses labels, while iloc uses integer positions.

  - Inclusive on both ends (rows and columns) for slices with loc, exclusive for iloc.

Choosing between loc and iloc depends on the context of your data analysis.

- Use loc when working with labels and specific rows/columns.

- Use iloc when working with integer positions and specific rows/columns.

Slicing with loc and iloc

- Compare slicing with loc and iloc.
- Note the inclusivity difference in the specified ranges.

Example:

```
# Slicing with loc
sliced_data_loc = df.loc[1:3, 'Name':'City']

# Slicing with iloc
sliced_data_iloc = df.iloc[1:3, 0:3]

print(sliced_data_loc)
print()
print(sliced_data_iloc)
```

```
      Name  Age           City
1      Bob   30  San Francisco
2  Charlie   35    Los Angeles

      Name  Age           City
1      Bob   30  San Francisco
2  Charlie   35    Los Angeles
```

Conditional Selection

- It is a powerful feature that allows us to filter rows based on specific conditions.

```
df['Age'] > 25
```

```
0    False
1     True
2     True
Name: Age, dtype: bool
```

- This condition checks each row in the 'Age' column and returns a boolean Series indicating whether the condition is met for each row.

```
filtered_data_series = df['Age'] > 25
```

```
filtered_data_series
```

```
0    False
1     True
2     True
Name: Age, dtype: bool
```

```
print(type(filtered_data_series))
```

```
<class 'pandas.core.series.Series'>
```

Filtering Data with Conditional Selection

- The boolean Series is then used to filter rows in the DataFrame.
- The condition is specified within square brackets, resulting in a DataFrame that satisfies the given criteria.
- Only rows where the condition is True are included in the filtered_data DataFrame.

```
filtered_data = df[df['Age'] > 25]

filtered_data
```

	Name	Age	City
1	Bob	30	San Francisco
2	Charlie	35	Los Angeles

Practical Use Cases:

- This technique is beneficial for extracting subsets of data that meet specific criteria.

- Common use cases include filtering data based on numerical conditions, categorical values, or combinations of conditions.

```
# Example: Retrieving data where 'City' is 'New York'
ny_data = df[df['City'] == 'New York']

ny_data
```

	Name	Age	City
0	Alice	25	New York

Complex Conditional Selection

- Conditions can be combined using logical operators (& for AND, | for OR) to create more complex filtering criteria.

```
# Example: Retrieving data where 'Age' is greater than 25 and 'City' is 'New York'
complex_condition = df[(df['Age'] > 20) & (df['City'] == 'New York')]

complex_condition
```

	Name	Age	City
0	Alice	25	New York

Using loc with Conditional Selection

➢ Using the loc method and conditional selection, we can do the same work as we did to filter rows and extract a subset of data from the entire dataframe.

```
# Using loc for subset data selection
filtered_data = df.loc[df['Age'] > 25]

filtered_data
```

	Name	Age	City
1	Bob	30	San Francisco
2	Charlie	35	Los Angeles

```
# Using loc for subset data selection
ny_data = df.loc[df['City'] == 'New York']

ny_data
```

	Name	Age	City
0	Alice	25	New York

Using loc with Conditional Selection

Example:

Use loc to select rows where the age is greater than 30 and display data in the ‘Name’ and ‘City’ columns, not the entire dataframe.

```
# Using loc for label-based selection
selected_rows_labels = df.loc[df['Age'] > 30, ['Name', 'City']]

selected_rows_labels
```

	Name	City
2	Charlie	Los Angeles

Slicing with loc (label-based selection)

dataframe.loc[<row selection> , <column selection>]

Single row Index/Label : ‘Bob’
List of row labels: [‘Alice’, ‘Bob’]
Slice of rows: ‘Alice’ : ‘Charlie’
Logical/Boolean index: dataframe[‘Age’] == 10

Single column name: ‘Name’
List of column names: [‘Name’, ‘Age’]
Slice of columns: ‘Name’ : ’City’

Slicing with iloc (position-based selection)

dataframe.iloc[<row selection> , <column selection>]

Single row selection: 1	Single column selections: 1
Integer list of rows: [0,1,2]	Integer list of columns: [0,1,2]
Slice of rows: 0:2	Slice of columns: 0:2

Operations on Columns:

- Adding a New Column
- Removing Columns
- Renaming Columns
- Modifying Values in Columns:

Adding Columns

- In addition to modifying existing data, we can add new columns to our DataFrame to incorporate additional information relevant to our analysis.

- The ability to add new columns is essential for incorporating additional information into your DataFrame.

- Adding columns allows us to include new data or derived features in our analysis.

Adding a Column with Default Values

- We can add a new column with default values for all rows.
- This is useful for marking or categorizing data.

Example:

```
# Adding a column with default values
df['New Column'] = 'Default Value'

df
```

	Name	Age	City	Age Doubled	New Column
0	Alice	25	New York	50	Default Value
1	Bob	30	San Francisco	60	Default Value
2	Charlie	35	Los Angeles	70	Default Value

Adding a Calculated Column

➢ Here, we add a new column ‘Age Doubled’ by performing a calculation based on an existing column ‘Age’.

Example:

```
# Adding a calculated column
df['Age Doubled'] = df['Age'] * 2

df
```

	Name	Age	City	Age Doubled
0	Alice	25	New York	50
1	Bob	30	San Francisco	60
2	Charlie	35	Los Angeles	70

Removing a Single Column

- Removing unnecessary or redundant columns is crucial for streamlining your DataFrame and focusing on relevant information.

- We can remove a single column, ‘New Column’, using the *drop* method with the columns parameter.

Example:

```
# Removing a single column
df.drop(columns=['New Column'], inplace=True)

df
```

	Name	Age	City	Age Doubled
0	Alice	25	New York	50
1	Bob	30	San Francisco	60
2	Charlie	35	Los Angeles	70

Removing Multiple Columns

- If we need to remove multiple columns, we can provide a list of column names to the drop method.

Example:

```
# Removing multiple columns
columns_to_remove = ['Age', 'Age Doubled']
df.drop(columns=columns_to_remove, inplace=True)

df
```

	Name	City
0	Alice	New York
1	Bob	San Francisco
2	Charlie	Los Angeles

- Removing columns should be done thoughtfully. Ensure that the columns being removed are genuinely unnecessary for your analysis to avoid data loss.

Renaming a Single Column

- Renaming columns allows us to provide more meaningful and concise names, improving the interpretability of our DataFrame.
- We rename the column 'Name' to 'Full Name' using the rename method with a dictionary of column name mappings.

Example:

```
# Renaming a single column
df.rename(columns={'Name': 'Full Name'}, inplace=True)

df
```

	Full Name	Age	City
0	Alice	25	New York
1	Bob	30	San Francisco
2	Charlie	35	Los Angeles

Renaming Multiple Columns

- We can rename multiple columns simultaneously by providing a dictionary of new column names.

Example:

```
# Renaming multiple columns
new_column_names = {'Age': 'Years', 'City': 'Location'}
df.rename(columns=new_column_names, inplace=True)

df
```

	Full Name	Years	Location
0	Alice	25	New York
1	Bob	30	San Francisco
2	Charlie	35	Los Angeles

Aggregating Numeric Columns

- Aggregating data across columns is essential for summarizing information and creating new insights.

Example:

We aggregate numeric columns, calculating the total and average age of individuals in the DataFrame.

```
# Aggregating numeric columns
total_age = df['Age'].sum()
average_age = df['Age'].mean()

print(f'Total Age: {total_age}, Average Age: {average_age}')
```

```
Total Age: 90, Average Age: 30.0
```

Aggregating Categorical Columns

- For categorical columns like 'City', we can use the value_counts method to get a count of unique values.

Example:

```
# Aggregating categorical columns
city_counts = df['City'].value_counts()

print(city_counts)
```

```
City
New York         1
San Francisco    1
Los Angeles      1
Name: count, dtype: int64
```

Modifying Values in a Columns:

- Data within a DataFrame can be modified by assigning new values to specific columns.

Here, we square the ages of individuals in the DataFrame.

This illustrates how we can modify existing data within a column.

```
# Changing column values
df['Age'] = df['Age'] ** 2

df
```

	Name	Age	City
0	Alice	625	New York
1	Bob	900	San Francisco
2	Charlie	1225	Los Angeles

Operations on Rows:

- Creating a New Row
- Removing Rows
- Modifying Values in a Row
- Copying Rows:
- Iterating Over Rows:

Adding a New Row:

- To add a new row in a DataFrame, you can use the loc method to add a new entry based on the index.

Example:

Adding a New Row to the end of dataframe

```
# Creating a new row
new_data = {'Name': 'Emma', 'Age': 27, 'City': 'Berlin'}
df.loc[len(df)] = new_data

df
```

	Name	Age	City
0	Alice	25	New York
1	Bob	30	San Francisco
2	Charlie	35	Los Angeles
3	Emma	27	Berlin

Removing Rows

➢ To delete specific rows, you can use the drop method or boolean indexing.

- Deleting Rows by Index using drop method:

```
# Deleting rows by index
df = df.drop(index=[0, 2])

df
```

	Name	Age	City
1	Bob	30	San Francisco

Removing Rows

- Deleting Rows based on a Condition:

```
# Deleting rows based on a condition
df = df[df['City'] != 'Berlin']

df
```

	Name	Age	City
0	Alice	25	New York
1	Bob	30	San Francisco
2	Charlie	35	Los Angeles

Modifying Values in a Row:

- To modify values in a specific row, you can use the loc method along with the column names.

Example:

```
# Modifying values in a row
df.loc[df['Name'] == 'Emma', 'Age'] = 28

df
```

	Name	Age	City
0	Alice	25	New York
1	Bob	30	San Francisco
2	Charlie	35	Los Angeles
3	Emma	28	Berlin

Copying Rows:

- To create a copy of a specific row, you can use the copy method.

Example:

```
# Copying a row
copied_row = df.loc[df['Name'] == 'Bob'].copy()

copied_row
```

	Name	Age	City
1	Bob	30	San Francisco

Iterating Over Rows:

- While it's generally advised to avoid explicit iteration over DataFrame rows, you can use the iterrows method if necessary.

Example:

```
# Iterating over rows (not recommended for large DataFrames)
for index, row in df.iterrows():
    print(f"Index: {index}, Name: {row['Name']}, Age: {row['Age']}, City: {row['City']}")
```

```
Index: 0, Name: Alice, Age: 25, City: New York
Index: 1, Name: Bob, Age: 30, City: San Francisco
Index: 2, Name: Charlie, Age: 35, City: Los Angeles
```

- Remember, Pandas is designed to perform vectorized operations, and explicit iteration over rows is not always the most efficient way to manipulate data. It's recommended to use vectorized operations whenever possible for better performance.

Exporting Data:

➢ Pandas provides convenient methods for writing a DataFrame to different file formats, making it easy to export your data for further analysis or sharing.

- To CSV
- To Excel
- To SQL Database
- To JSON

Writing to CSV or Excel

- Use the to_csv method to export a DataFrame to a CSV file.

```
# Writing to a CSV file
df.to_csv('output_file.csv', index=False)
```

- Use the to_excel method to export a DataFrame to an Excel file.

```
# Writing to an Excel file
df.to_excel('output_file.xlsx', index=False)
```

- The index parameter is set to False to exclude the index column from the file.

Writing to JSON

- Use the to_json method to export a DataFrame to a JSON file.

```
# Writing to a JSON file
df.to_json('output_file.json', orient='records')
```

- The orient parameter specifies the format of the JSON file. 'records' is a common choice.

Exercise

- DataFrame Operations

  - Add a new column named ‘Bonus’ with values calculated by ‘Experience’ column values multiplied by 2.
  - Delete the newly created column 'Bonus' from the DataFrame.
  - Delete the employee with ‘Role’ as ‘Helpdeskmanager’ from the DataFrame.
  - Modify the ‘Salary’ of the employee in ‘Amsterdam’ to 20% more.

Exercise

- Selecting / Filtering

  - Select and show only the employees (rows) with ‘Experience’ greater than 5 years.
  - Select and show only the employees (rows) with ‘Salary’ less than 4000.
  - Select and show only the employees (rows) who are ‘Product Owner’.
  - Select and show only the employees (rows) who have a ‘HBO’ education.
  - Select and show only the employees (rows) who are not in ‘Amsterdam’.

Exercise

- Advanced Conditional Selection

  - Select and show rows where ‘Salary’ is between 4000 and 5000.
  - Select and show rows where ‘Role’ is either ‘Product Owner’ or ‘IT Consultant’.
  - Select and show rows where ‘City’ is ‘Amsterdam’ and ‘Experience’ is less than 10 years, or ‘City’ is ‘Rotterdam’.
  - Select and show rows where ‘Role’ is ‘Product Owner’ and ‘Experience’ is greater than 5 or ‘Salary’ is greater than 4500.
  - Select and show rows where ‘Education’ is ‘MBO’ and ‘City’ is not ‘Apeldoorn’ or ‘Experience’ is less than 8.