INTRODUCTION TO

DATA SCIENCE

Data to Decisions: The Science of Insight

by Dr. Fardina Alam
Gavin Hung

CMSC320TextBook

October 6, 2025

1 Introduction

by Dr. Fardina Alam and Gavin Hung

2 What is Pandas?

Pandas is a powerful open-source Python library that provides high-performance, easy-to-use data structures and data analysis tools. The name "pandas" comes from "panel data," a term used in econometrics to describe multi-dimensional data. Developed by Wes McKinney in 2008, pandas has become the cornerstone of data manipulation and analysis in the Python ecosystem.

At its core, Pandas helps you:

- Load, clean, and transform datasets.
- Perform statistical operations efficiently.
- Handle missing or inconsistent data.
- Merge, reshape, and aggregate large datasets.

If you have ever worked with spreadsheets in Excel, Pandas offers similar functionality—but with far greater power, speed, and scalability.

3 Why Use Pandas?

Before pandas, data analysis in Python was cumbersome and required jumping between different libraries. Python users relied heavily on lists, dictionaries, and NumPy arrays for handling structured data. While these tools are powerful, they lack built-in functionality for common tasks like handling missing values, grouping data, or joining tables. Pandas solved this by providing:

- **Intuitive data structures:** DataFrames and Series that feel familiar to users from various backgrounds, useful for for working with tabular and one-dimensional data.
- **Seamless integration:** Works beautifully with other Python data science libraries (NumPy, Matplotlib, etc).
- **Powerful data manipulation:** Easy filtering, grouping, and transformation of data
- **Performance:** Built on top of highly optimized C code for speed.
- **Time series functionality:** Excellent support for working with time-based data

- **Ease of Use:** Simplifies complex operations into a few lines of readable code.

4 Installing Pandas

Before using Pandas, you need to install it. If you are using Anaconda, Pandas comes pre-installed. Otherwise, you can install it with:

pip install pandas

Or if you're using Anaconda:

conda install pandas

To confirm the installation, open a Python shell and type:

import pandas as pd print(pd.__version__)

#Loading Data with Pandas

One of Pandas' biggest strengths is its ability to easily import/export datasets from multiple formats:

- **CSV:** pd.read_csv("file.csv")
- **Excel:** pd.read_excel("file.xlsx")
- **SQL Databases:** pd.read_sql(query, connection)
- **JSON:** pd.read_json("file.json")

[2]:
```python
#Example:
import pandas as pd

df = pd.read_csv("students.csv")
print(df.head())   # Displays first 5 rows
```

```
   Unnamed: 0 default student      balance        income
0           1      No      No   729.526495  44361.625074
1           2      No     Yes   817.180407  12106.134700
2           3      No      No  1073.549164  31767.138947
3           4      No      No   529.250605  35704.493935
4           5      No      No   785.655883  38463.495879
```

5 Core Data Structures

The strength of Pandas lies in two core objects: 1. **Series:** A one-dimensional labeled array 2. **Dataframe:** A two-dimensional labeled data structure

5.1 Series: The One-Dimensional Workhorse

A Series is a one-dimensional labeled array that can hold any data type. Think of it as a single column in a spreadsheet.

Unlike some arrays that require all elements to be the same type (homogeneous), a Series can store different types of values together, such as numbers, text, or dates. Each value has a label called an index, which can be numbers, words, or timestamps, and you can use it to quickly find or select values. Here are some examples:

5.1.1 How to Create a Series

A Series can be created directly from a ***Python list***, in which case pandas automatically assigns default numeric indexes (0, 1, 2, …) to each element.

You can also create a Series from a ***Python dictionary***, where the dictionary keys become the index labels and the dictionary values become the Series values. In Python 3.7 and later, the order of the keys is preserved, so the Series keeps the same order as the dictionary

[3]:
```python
import pandas as pd

# Creating a Series from a list
temperatures = pd.Series([22, 25, 18, 30, 27],
                        index=['Mon', 'Tue', 'Wed', 'Thu', 'Fri'],
                        name='Daily_Temps')
print(temperatures)


# Creating a Series from a Dictionary

grades = {"Math": 90, "English": 85, "Science": 95}
dict_series = pd.Series(grades)

print(dict_series)
```

```
Mon    22
Tue    25
Wed    18
Thu    30
Fri    27
Name: Daily_Temps, dtype: int64
Math       90
English    85
Science    95
dtype: int64
```

[4]:
```python
import pandas as pd

# Homogeneous Series (all integers)
print("Homogeneous Series \n")
homo_series = pd.Series([10, 20, 30, 40], index=['A', 'B', 'C', 'D'])
print(homo_series)
print(f"The data type is: {homo_series.dtype}\n")
```

```python
# Heterogeneous Series (mix of int, float, string, bool)
print("Heterogeneous Series \n")
hetero_series = pd.Series([10, 20.5, 'hello', True])
print(hetero_series)
print(f"The data type is: {hetero_series.dtype}")
```

```
Homogeneous Series

A    10
B    20
C    30
D    40
dtype: int64
The data type is: int64

Heterogeneous Series

0       10
1     20.5
2    hello
3     True
dtype: object
The data type is: object
```

5.2 DataFrame: The Two-Dimensional Powerhouse

A DataFrame is a two-dimensional labeled data structure, similar to a table with rows and columns. It is the most commonly used object in Pandas.

Here are some examples:

[5]:
```python
# Creating a DataFrame from a dictionary
data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'Diana'],
    'Age': [25, 30, 35, 28],
    'City': ['New York', 'London', 'Paris', 'Tokyo']
}

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

```
      Name  Age      City
0    Alice   25  New York
1      Bob   30    London
2  Charlie   35     Paris
3    Diana   28     Tokyo
```

[6]:
```python
import pandas as pd
```

```python
# Create a simple dataset
data = {
    'Product': ['Apple', 'Banana', 'Cherry', 'Date'],
    'Price': [1.20, 0.50, 3.00, 2.50],
    'Stock': [45, 120, 15, 80]
}

# Create DataFrame
df = pd.DataFrame(data)

# Display basic information
print("Our DataFrame:")
print(df)
print("\nData types:")
print(df.dtypes)
print("\nBasic statistics:")
print(df.describe())
```

```
Our DataFrame:
  Product  Price  Stock
0   Apple    1.2     45
1  Banana    0.5    120
2  Cherry    3.0     15
3    Date    2.5     80

Data types:
Product     object
Price      float64
Stock        int64
dtype: object

Basic statistics:
         Price       Stock
count  4.00000    4.000000
mean   1.80000   65.000000
std    1.15181   45.276926
min    0.50000   15.000000
25%    1.02500   37.500000
50%    1.85000   62.500000
75%    2.62500   90.000000
max    3.00000  120.000000
```

6 Pandas Basic Operations

After reading tabular data as a DataFrame, you would need to have a glimpse of the data. Pandas makes it easy to explore and manipulate. A good first step is to inspect the dataset by previewing how many rows and columns it has, what the column names are, checking dimensions, or reviewing summary information such as data types and statistics. Pandas provides convenient methods for this.

##Viewing/Exploring Data

Command	Description	Default Behavior
`df.head()`	Displays the **first rows** of the DataFrame. Useful for quickly previewing the dataset.	Shows **5 rows**
`df.tail()`	Displays the **last rows** of the DataFrame. Handy for checking the dataset's ending records.	Shows **5 rows**
`df.shape`	Returns a tuple `(rows, columns)` representing the **dimensions** of the DataFrame.	N/A
`df.info()`	Shows column names, **data types**, memory usage, and count of non-null values.	N/A
`df.dtypes`	Returns the **data type of each column** in the DataFrame.	N/A
`df.describe()`	Provides **summary statistics** (mean, std, min, max, quartiles) for numeric columns.	Includes numeric columns by default

[7]:
```python
import pandas as pd

data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [24, 30, 28],
    'Salary': [50000, 60000, 55000]
}
df = pd.DataFrame(data)

print("head")
print(df.head())
print("shape")
print(df.shape)
print("info")
print(df.info)
```

```
head
      Name  Age  Salary
0    Alice   24   50000
1      Bob   30   60000
2  Charlie   28   55000
shape
(3, 3)
info
<bound method DataFrame.info of       Name  Age  Salary
0    Alice   24   50000
1      Bob   30   60000
2  Charlie   28   55000>
```

6.1 Selecting and Indexing Data

After inspecting the structure of a DataFrame, the next step is often to select specific rows and columns (specific parts of the data). Pandas provides several approaches depending on whether you want to select columns, rows, or filter data based on conditions. It lets you choose columns, rows, or both using labels (loc), integer positions (iloc), or conditions.

###1. **Selecting Columns**

Command	Description
`df['col']`	Selects a **single column** as a Series.
`df[['col1','col2']]`	Selects **multiple columns** as a new DataFrame.

###2. **Selecting Rows**

Command	Description
`df.loc[row_label]`	Select row(s) by **label** (index name).
`df.iloc[row_index]`	Select row(s) by **integer position**.
`df.loc[0, 'col']`	Select a **specific value** by row & column.

###3. **Conditional Selection (Filtering)**

Command	Description
`df[df['col'] > 50]`	Returns rows where condition is **True**.
`df[(df['A'] > 50) & (df['B'] < 100)]`	Combine conditions with `&` (and), ‘ ‘ (or).

Follow is an example:

[8]:
```python
import pandas as pd

data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [24, 30, 28],
    'City': ['New York', 'Los Angeles', 'Chicago']
}

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

# Now, to access the columns: Select columns
df['Age']
df[['Name', 'City']]

# Select rows
```

```
df.iloc[0]         # First row
df.iloc[1:3]       # Rows 1-2
df.loc[0, 'Name'] # Specific cell

# Conditional selection
df[df['Age'] > 30]
df[(df['Age'] > 30) & (df['City'] == 'Chicago')]
```

```
      Name  Age         City
0    Alice   24     New York
1      Bob   30  Los Angeles
2  Charlie   28      Chicago
```

[8]:
```
Empty DataFrame
Columns: [Name, Age, City]
Index: []
```

#Adding a new column to DataFrame

A new column can be added to a pandas DataFrame by assigning a ***value, list, or Series*** to a new column name. If the assigned data is a list or Series, its length must match the number of rows in the DataFrame. You can also assign a single value, which will be applied to all rows.

[9]:
```
import pandas as pd

df = pd.DataFrame({
    "Name": ["Alice", "Bob", "Charlie"],
    "Math": [90, 85, 95]
})
print(df)

# Add a new column with a list
df["English"] = [88, 92, 80]

print(df)

# Add a new column with a single value
df["Pass"] = True

print(df)
```

```
      Name  Math
0    Alice    90
1      Bob    85
2  Charlie    95
      Name  Math  English
0    Alice    90       88
1      Bob    85       92
2  Charlie    95       80
```

```
      Name  Math  English  Pass
0    Alice    90       88  True
1      Bob    85       92  True
2  Charlie    95       80  True
```

7 Arithmetic Operations and Functions in Pandas

So far we explored how to inspect and select data in Pandas. Once you have access to the right rows and columns, the next step is to perform calculations and apply functions. Pandas makes this process very intuitive by allowing you to apply arithmetic directly to DataFrames or Series, and by offering tools like apply(), map(), and applymap() for more flexibility.

7.1 Arithmetic Operations on Columns

You can directly apply mathematical operations to Pandas Series or DataFrame columns. Operations are vectorized, meaning they are applied element-wise across the column.

In below example, you can notice how the operations are automatically applied to each row.

[10]:
```python
import pandas as pd

data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [24, 30, 28],
    'Salary': [50000, 60000, 55000]
}
df = pd.DataFrame(data)
print(df)

# Increase all salaries by 10%
df['Salary'] = df['Salary'] * 1.10

# Add 5 years to everyone's age
df['Age'] = df['Age'] + 5
print(df)
```

```
      Name  Age  Salary
0    Alice   24   50000
1      Bob   30   60000
2  Charlie   28   55000
      Name  Age   Salary
0    Alice   29  55000.0
1      Bob   35  66000.0
2  Charlie   33  60500.0
```

7.2 Arithmetic Between Columns

You can also perform arithmetic between two or more columns to create new features.

[11]:
```
# Create a new column 'Income_per_Age'
df['Income_per_Age'] = df['Salary'] / df['Age']
print(df)
```

```
      Name  Age   Salary  Income_per_Age
0    Alice   29  55000.0     1896.551724
1      Bob   35  66000.0     1885.714286
2  Charlie   33  60500.0     1833.333333
```

8 Applying Built-in Pandas/Numpy Functions

Pandas integrates with NumPy functions, allowing you to apply common statistics directly.

[12]:
```
import numpy as np

# Calculate average salary
print(df['Salary'].mean())

# Standard deviation of Age
print(df['Age'].std())

# Apply numpy square root
print(np.sqrt(df['Age']))
```

```
60500.0
3.0550504633038935
0    5.385165
1    5.916080
2    5.744563
Name: Age, dtype: float64
```

8.0.1 Applying Functions with apply()

Sometimes you need custom transformations. The apply() method lets you apply a function to an entire column (Series) or to each row/column in a DataFrame.

[13]:
```
# Apply to a Series
df['Age_squared'] = df['Age'].apply(lambda x: x**2)

# Apply to DataFrame across rows
df['Total'] = df[['Age','Salary']].apply(lambda row: row['Age'] +␣
 ↪row['Salary'], axis=1)
print(df)
```

```
      Name  Age   Salary  Income_per_Age  Age_squared    Total
0    Alice   29  55000.0     1896.551724          841  55029.0
1      Bob   35  66000.0     1885.714286         1225  66035.0
2  Charlie   33  60500.0     1833.333333         1089  60533.0
```

Note that, we can also apply a Function Elementwise with applymap() and to a Single Column with map() but not covering in this course.

9 Filtering Data in Pandas

Once you know how to select columns and rows, the next step is learning how to filter data. Filtering helps you focus on only the relevant part of your dataset, whether that means removing unnecessary columns, isolating rows that meet certain conditions, or preparing features for modeling.

9.1 Filtering Columns

Column filtering is about selecting only the columns you need or dropping the ones you don't. This reduces memory usage and keeps your DataFrame manageable.

[14]:
```
# Select a single column
df['Age']

# Select multiple columns
df[['Name', 'Age']]
```

[14]:
```
      Name  Age
0    Alice   29
1      Bob   35
2  Charlie   33
```

9.2 Dropping Unused Columns

[15]:
```
# Drop the 'Age_squared' column
df = df.drop(columns=['Age_squared'])
print(df)
```

```
      Name  Age   Salary  Income_per_Age    Total
0    Alice   29  55000.0     1896.551724  55029.0
1      Bob   35  66000.0     1885.714286  66035.0
2  Charlie   33  60500.0     1833.333333  60533.0
```

This is especially useful when preparing data for machine learning, where only selected features are required.

9.3 Filtering Rows (using Boolean Indexing)

Row filtering is usually done with Boolean indexing, where you apply a condition and return only the rows where that condition is true.

[16]:
```
# Filter rows where Age > 30
df[df['Age'] > 30]
```

[16]:
```
      Name  Age   Salary  Income_per_Age    Total
1      Bob   35  66000.0     1885.714286  66035.0
```

```
2  Charlie   33  60500.0     1833.333333  60533.0
```

10 Combining Multiple Conditions

You can combine conditions using & (and) or | (or).

[17]:
```
# Filter rows where Age > 30 AND Salary > 60000
df[(df['Age'] > 30) & (df['Salary'] > 60000)]
```

[17]:
```
      Name  Age   Salary  Income_per_Age    Total
1      Bob   35  66000.0     1885.714286  66035.0
2  Charlie   33  60500.0     1833.333333  60533.0
```

Remember to wrap each condition in parentheses.

11 Filtering Strings

You can filter rows where a text column contains specific values

[18]:
```
# Filter rows where Name contains "Bob"
df[df['Name'].str.contains("Bob")]
```

[18]:
```
  Name  Age   Salary  Income_per_Age    Total
1  Bob   35  66000.0     1885.714286  66035.0
```

##Unique Values and Counting

Sometimes you want to check how many unique values a column has, or count how often each appears.

[19]:
```
# Unique names
print(df['Name'].unique())

# Count frequency of each name
print(df['Name'].value_counts())
```

```
['Alice' 'Bob' 'Charlie']
Name
Alice      1
Bob        1
Charlie    1
Name: count, dtype: int64
```

12 Applying Aggregation Functions Directly to a DataFrame

One of the strengths of Pandas is that you can apply statistical and aggregation methods directly to a DataFrame or Series. These methods summarize data and provide insights without needing extra loops or manual calculations.

12.0.1 Common Aggregation Methods

Here are some of the most commonly used methods:

Method	Description	Works On
`.sum()`	Returns the **sum** of values	DataFrame / Series
`.mean()`	Returns the **average (mean)** value	DataFrame / Series
`.count()`	Counts **non-null values**	DataFrame / Series
`.min()`	Returns the **minimum** value	DataFrame / Series
`.max()`	Returns the **maximum** value	DataFrame / Series
`.std()`	Returns the **standard deviation**	DataFrame / Series
`.var()`	Returns the **variance**	DataFrame / Series
`.describe()`	Generates **summary statistics** (count, mean, std, min, quartiles, max)	DataFrame / Series

Example: Aggregating a Series

[20]:
```
import pandas as pd

# Salary data
salaries = pd.Series([50000, 60000, 55000, 65000, 70000])

print("Sum:", salaries.sum())
print("Mean:", salaries.mean())
print("Max:", salaries.max())
print("Std Dev:", salaries.std())
```

```
Sum: 300000
Mean: 60000.0
Max: 70000
Std Dev: 7905.694150420948
```

Each method is applied directly to the Series, returning a single value.

Example: Aggregating a DataFrame

[21]:
```
data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [24, 30, 28],
    'Salary': [50000, 60000, 55000]
}
df = pd.DataFrame(data)

print(df.sum(numeric_only=True))   # Sum of numeric columns
print(df.mean(numeric_only=True))  # Mean of numeric columns
print(df.describe())
```

```
Age           82
Salary    165000
```

```
dtype: int64
Age           27.333333
Salary     55000.000000
dtype: float64
             Age   Salary
count   3.000000      3.0
mean   27.333333  55000.0
std     3.055050   5000.0
min    24.000000  50000.0
25%    26.000000  52500.0
50%    28.000000  55000.0
75%    29.000000  57500.0
max    30.000000  60000.0
```

Notice how these functions automatically ignore non-numeric columns (like "Name").

13 More Advanced: Filtering Data & Apply Statistical Functions

We can combine **row filtering** with **aggregation functions** to analyze subsets of a DataFrame.

The general syntax is:

df[df['column_name'] value]['target_column'].function()

where:

- df[...] → filters the rows that meet the condition
- ['target_column'] → selects the column to aggregate
- .function() → applies the aggregation function

[22]:
```python
import pandas as pd

data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'David'],
    'Age': [24, 35, 28, 40],
    'Salary': [50000, 66000, 55000, 70000]
}
df = pd.DataFrame(data)

# Average salary of employees older than 30
avg_salary = df[df['Age'] > 30]['Salary'].mean()
print(avg_salary)

# Maximum salary for employees younger than 30
df[df['Age'] < 30]['Salary'].max()

# Count employees with salary above 60,000
df[df['Salary'] > 60000]['Name'].count()
```

```
# Standard deviation of salary for people aged 25-40
df[(df['Age'] >= 25) & (df['Age'] <= 40)]['Salary'].std()
```

```
68000.0
```

[22]:
```
7767.45346515403
```

So the syntax pattern is:

df[df[‘condition’]][‘column’].aggregation()

Expression	Meaning
`df[df['Age'] > 30]['Salary'].mean()`	Mean of Salary where Age > 30
`df[df['Salary'] > 60000]['Name'].count()`	Count of employees with Salary > 60k
`df[(df['Age'] >= 25) & (df['Age'] <= 40)]['Salary'].std()`	Standard deviation of Salary for 25–40 year olds

This pattern allows you to filter data first, then aggregate only on the rows that meet your condition.

14 Grouping Data with `groupby`

While filtering + aggregation lets us summarize a **subset** of data, the `groupby()` method allows us to compute statistics **across categories**.
This is the classic **split–apply–combine** process:

1. **Split** data into groups based on one or more columns.
2. **Apply** an aggregation function to each group.
3. **Combine** results into a new DataFrame or Series.

14.1 Basic Syntax

df.groupby(‘column_name’)[‘target_column’].aggregation_function() where: - `groupby('column_name')` → splits the data into groups.
- `['target_column']` → selects the column to aggregate.
- `.aggregation_function()` → applies functions like `mean()`, `sum()`, `count()`.

[23]:
```
## Example: Salary by Department

import pandas as pd

# Sample dataset
data = {
    'Department': ['HR','HR','IT','IT','Finance','Finance'],
    'Employee': ['Alice','Bob','Charlie','David','Eva','Frank'],
    'Salary': [50000, 52000, 60000, 62000, 58000, 60000]
}
df = pd.DataFrame(data)
print(df)

# Average salary per department
df.groupby('Department')['Salary'].mean()
```

```
  Department Employee  Salary
0         HR    Alice   50000
1         HR      Bob   52000
2         IT  Charlie   60000
3         IT    David   62000
4    Finance      Eva   58000
5    Finance    Frank   60000
```

[23]:
```
Department
Finance    59000.0
HR         51000.0
IT         61000.0
Name: Salary, dtype: float64
```

14.1.1 Grouping by Multiple Columns

[24]:
```
# Example dataset with Region added
data2 = {
    'Department': ['HR','HR','IT','IT','Finance','Finance'],
    'Region': ['East','West','East','West','East','West'],
    'Salary': [50000, 52000, 60000, 62000, 58000, 60000]
}
df2 = pd.DataFrame(data2)

# Group by Department and Region
df2.groupby(['Department','Region'])['Salary'].mean()
```

[24]:
```
Department  Region
Finance     East      58000.0
            West      60000.0
HR          East      50000.0
            West      52000.0
IT          East      60000.0
            West      62000.0
Name: Salary, dtype: float64
```

#Exporting Data in Pandas

After processing your data in Pandas, you can save it to files in various formats:

Format	Function	Key Parameters / Notes	Example Usage
CSV	`to_csv()`	`index=False` to skip row numbers, `sep='\t'` for tab-delimited	`df.to_csv('data.csv', index=False)`
Excel	`to_excel()`	`sheet_name='Sheet1'`, requires `openpyxl`	`df.to_excel('data.xlsx index=False)`
JSON	`to_json()`	`orient='records'`, `lines=True` for line-delimited JSON	`df.to_json('data.json' orient='records', lines=True)`
Pickle	`to_pickle()`	Python-specific, fast binary format	`df.to_pickle('data.pkl`
HTML	`to_html()`	Saves as an HTML table	`df.to_html('data.html' index=False)`
Parquet	`to_parquet()`	Efficient columnar format, great for big data	`df.to_parquet('data.pa index=False)`

Tip: Always choose the format based on your use case:
- CSV → universal, easy sharing
- Excel → spreadsheets
- JSON → web APIs or NoSQL
- Parquet → large datasets, high performance

[25]:
```python
# Save CSV
df.to_csv('output.csv', index=False)
print("CSV file 'output.csv' created successfully.")

# Save tab-separated CSV
df2.to_csv('output_tab.csv', sep='\t', index=False)
print("Tab-separated CSV file 'output_tab.csv' created successfully.")

# Save Excel
df.to_excel('output.xlsx', index=False, sheet_name='Sheet1')
print("Excel file 'output.xlsx' created successfully.")

# Save JSON
df2.to_json('output.json', orient='records', lines=True)
print("JSON file 'output.json' created successfully.")
```

```
CSV file 'output.csv' created successfully.
Tab-separated CSV file 'output_tab.csv' created successfully.
Excel file 'output.xlsx' created successfully.
JSON file 'output.json' created successfully.
```

15 Exporting Pandas Data in Google Colab

In Colab, you can save files directly to Google Drive. First, mount your Drive:

"'python from google.colab import drive drive.mount('/content/drive') # Follow the link and paste the authorization code

16 The pandas Ecosystem: How It Fits In

Pandas does not exist in a vacuum. It is a central hub in the Python data science stack:

- **NumPy:** Provides the foundational n-dimensional array object. Pandas DataFrames are built on top of NumPy arrays.
- **Matplotlib/Seaborn:** Used for visualization. You can plot data directly from DataFrames and Series.
- **Scikit-learn:** The premier machine learning library. It accepts DataFrames and Series as inputs for model training.
- **Jupyter Notebooks:** The ideal interactive environment for exploratory data analysis with pandas.

17 When to Use Pandas (And When Not To)

##Use pandas when:

- Working with tabular data (like spreadsheets or database tables)
- Data cleaning and preprocessing
- Exploratory data analysis
- Medium-sized datasets (up to a few gigabytes)

##Consider alternatives when:

- Working with very large datasets that don't fit in memory.
- Need extremely high performance for numerical computations (consider NumPy directly)
- Working with unstructured data like images or text

18 Key Takeaways

- Filtering + Aggregation → summarize specific rows based on conditions.
- GroupBy + Aggregation → summarize categories (all groups at once).
- Grouping can be done on one or multiple columns.

19 Summary of Pandas: Key Features at a Glance

- **Data Import/Export:** Read from and write to CSV, Excel, SQL, JSON, and many other formats
- **Data Cleaning:** Handle missing values, remove duplicates, filter outliers
- **Data Transformation:** Reshape, pivot, melt, and transform your data
- **Data Aggregation:** Group by categories and compute summary statistics

- **Time Series Analysis:** Work with dates and times effortlessly
- **Visualization Integration:** Works seamlessly with Matplotlib and Seaborn

19.1 Knowledge Check

Loading...

20 Chapter 6: Mastering SQL for Data Science with Python

The chapter introduces Structured Query Language (SQL), a powerful and essential tool for data professionals. SQL is the standard language used to communicate with and manage relational databases. This chapter focuses on its application in data science, demonstrating how to use SQL for data retrieval, manipulation, and analysis. It also covers the integration of SQL with Python, a crucial skill for any data scientist.

20.1 Introduction to SQL for Data Science

Data has become the foundation of decision-making in modern organizations. From social media platforms storing billions of user interactions to hospitals managing electronic health records, most of this information is stored in **databases**. Among different types of databases, **relational databases** are the most widely used.

Relational databases organize data into tables, which consist of **rows (records)** and **columns (attributes)**. This design reflects the way data naturally relates to entities in the real world. For example:

A retail store has customers (with IDs, names, and ages). Each customer places orders (with product details, dates, and amounts). * The relationship between customers and orders can be represented through keys.

To interact with these databases, we use a language called **Structured Query Language (SQL)**.

20.1.1 Structured Query Language (SQL).

SQL provides a standardized way to create, read, update, and delete data (commonly referred to as CRUD operations). Unlike programming languages such as Python or Java, SQL is declarative: you specify what you want, and the database figures out how to get it.

This makes it highly efficient for managing large datasets. For data science, SQL is invaluable for:

- **Data Retrieval:** Extracting specific subsets of data from large databases.
- **Data Cleaning and Transformation:** Handling missing values, standardizing formats, and creating new features.
- **Exploratory Data Analysis (EDA):** Performing quick summaries, aggregations, and data profiling.
- **Feature Engineering:** Creating new variables from existing ones before feeding them into machine learning models.

21 The Evolution of Databases and SQL

- **Early data management:** Before databases, organizations stored information in files. This approach led to redundancy, inconsistency, and inefficiency.
- **Birth of the relational model:** In 1970, Edgar F. Codd introduced the relational model, a mathematical foundation for organizing data in tables with well-defined relationships.
- **Development of SQL:** By the late 1970s, IBM developed SEQUEL (Structured English Query Language), which evolved into SQL. It became the ANSI standard in 1986.
- **SQL today:** Almost every relational database system (MySQL, PostgreSQL, Oracle, SQL Server, SQLite) supports SQL, with minor dialect differences.
- **Takeaway:** SQL is not just a programming tool—it is the backbone of modern data storage and analytics.

22 Relational Databases: Core Concepts

Relational databases are the backbone of structured data storage. They organize information in a way that ensures consistency, integrity, and efficient retrieval. The fundamental ideas of tables, keys, and relationships help us understand how real-world data is modeled.

22.0.1 Tables, Rows, and Columns

A table is like a spreadsheet.

- **Rows (records/tuples):** Each row corresponds to a single entity or instance of data. For example, one row might represent a single customer.
- **Columns (fields/attributes):** Each column stores one specific type of information about the entity, such as name, age, or gender.
- **Schema:** The structure of the table, which defines what columns exist and their data types (e.g., integer, string, date).

Example: A Customers table may contain columns such as Customer_ID, Name, Age, Gender, and Email. Each row represents one customer.

22.0.2 Keys

Keys in Relational Databases are crucial for ensuring that data remains **unique** and **consistent** across tables.

22.1 1. Primary Key (PK)

The primary key uniquely identifies each row. It contain UNIQUE values in column, and does not allows NULL values.

Here, Empid is a Primary Key. Example: **Customers Table**

Customer_ID	Name	Age	Gender
101	Alice	25	F
102	Bob	30	M
103	Charlie	28	M

- **Primary Key:** `Customer_ID`
- Ensures each customer is uniquely identifiable.

22.2 2. Foreign Key (FK)

A foreign key links one table to another. It creates a relationship between two or more tables, a primary key of one table is referred as a foreign key in another table. It can also accept multiple null values and duplicate values.

Orders Table

Order_ID	Customer_ID	Product	Quantity
5001	101	Laptop	1
5002	102	Keyboard	2
5003	101	Mouse	1

- `Customer_ID` here is a **foreign key** connecting each order to the **Customers** table.
- Prevents creating an order for a non-existent customer.

22.3 3. Composite Key

Sometimes, no single column uniquely identifies a row. Composite Key is a combination of more than one columns of a table. It can be a Candidate key and Primary key.

Enrollments Table

Student_ID	Course_ID	Grade
S001	CSE101	A
S001	MTH201	B+
S002	CSE101	A-
S002	PHY110	B

- Neither `Student_ID` nor `Course_ID` alone is unique.
- Together `(Student_ID, Course_ID)` form a **composite key**.
- Ensures a student cannot enroll in the same course twice.

22.4 4. Candidate Key

A candidate key is any column (or set of columns) that could serve as a primary key. Candidate Key(s) an identify a record uniquely in a table and which can be selected as a primary key of the table.

It contains UNIQUE values in column, and does not allows NULL values.

Here, Empid, EmpLicence and EmpPassport are candidate keys.

Example: **Employees Table**

Employee_ID	Email	SSN	Name
E001	alice@company.com	123-45-6789	Alice
E002	bob@company.com	987-65-4321	Bob
E003	charlie@company.com	111-22-3333	Charlie

- Possible unique identifiers:
  - `Employee_ID`
  - `Email`
  - `SSN`
- Each is a **candidate key**.
- One (e.g., `Employee_ID`) is chosen as the **primary key**.

Remember, Each table can have only **one Primary** key and **multiple Candidate** keys

23 PK-FK Relationships

Relational databases use **primary keys (PK)** and **foreign keys (FK)** to maintain data integrity and model relationships.

23.1 Types of Relationships

- **One-to-One (1:1):** Each person has one passport; each passport belongs to one person.
- **One-to-Many (1:N):** A customer can have many orders; each order belongs to one customer.

In this figure, a customer can have many accounts; each account belongs to one customer.

- **Many-to-Many (M:N):** Students enroll in many courses; courses have many students.

In this figure, each customer can buy more than one product and a product can be bought by many different customers.

23.2 Customers Table

Customer_ID (PK)	Name	Age	Gender
101	Alice	25	F
102	Bob	30	M
103	Charlie	28	M

23.3 Orders Table

Order_ID (PK)	Customer_ID (FK)	Product_ID (FK)	Quantity
5001	101	P100	1
5002	102	P101	2
5003	101	P102	1

- `Customer_ID` is a **foreign key** referencing `Customers.Customer_ID`.
- `Product_ID` is a **foreign key** referencing `Products.Product_ID`.

23.4 Products Table

Product_ID (PK)	Product_Name	Price
P100	Laptop	1000
P101	Keyboard	50
P102	Mouse	30

23.5 Passports Table (One-to-One Example)

Passport_ID (PK)	Customer_ID (FK)	Expiration_Date
P001	101	2030-12-31
P002	102	2031-06-30
P003	103	2030-09-15

- `Customer_ID` is a **foreign key** referencing `Customers.Customer_ID`.
- Each customer has exactly **one passport**.

23.6 Enrollments Table (Many-to-Many Example)

Student_ID	Course_ID	Grade
S001	CSE101	A

Student_ID	Course_ID	Grade
S001	MTH201	B+
S002	CSE101	A-
S002	PHY110	B

- Neither `Student_ID` nor `Course_ID` alone is unique.
- The combination `(Student_ID, Course_ID)` forms a **composite key**.
- Students can enroll in **many courses**, and courses can have **many students**.

23.7 Relationships Overview

From Table	To Table	Type	Notes
Customers	Orders	1:N	One customer → many orders
Products	Orders	1:N	One product → many orders
Customers	Passports	1:1	One customer → one passport
Students	Courses	M:N	Implemented via Enrollments table

23.8 Key Points

- **Primary Key (PK):** Unique identifier for each record. Cannot be NULL.
- **Foreign Key (FK):** Links a table to another table's primary key. Maintains referential integrity.
- **Composite Key:** Combination of columns used when a single column is not unique.
- **Candidate Key:** Any column or combination of columns that could serve as a primary key.
- **Constraints:** Rules to maintain data validity (e.g., NOT NULL, UNIQUE, CHECK, FOREIGN KEY).

23.9 The Role of SQL in Data Science

Think of SQL as your conversation partner with the data. It's a declarative language, which means you simply state your desired outcome, and the database handles the complex task of finding and organizing the data for you. This makes it incredibly efficient for handling massive datasets. A typical data science workflow using SQL might look like this:

- **Data Extraction**: You use a SELECT query to pull a specific subset of data relevant to your project.
- **Data Wrangling**: You perform initial cleaning, filtering (WHERE), and aggregation (GROUP BY) directly in the database.
- **Analysis**: The prepared data is loaded into Python (often as a Pandas DataFrame) for more sophisticated analysis, modeling, and visualization.

23.10 Core SQL Commands: Your Essential Toolkit

We have already learned that SQL is the standard language for managing and querying **relational databases**.

These core commands allow you to **create tables, insert data, retrieve information, update records, and maintain data integrity**.

Whether you are analyzing sales data, customer information, or product inventories, mastering these commands is essential for data-driven tasks.

Command	Purpose	Example
CREATE TABLE	Create a new table in the database	`sql CREATE TABLE Customers (CustomerID INT PRIMARY KEY AUTO_INCREMENT, Name VARCHAR(50), Email VARCHAR(100));`
INSERT INTO	Adds new rows of data to a table	`sql INSERT INTO Customers (CustomerID, Name, Email) VALUES (1, 'John Doe', 'john.doe@example.com');`
SELECT	Retrieves data from one or more tables	`sql SELECT Name, Email FROM Customers;`
WHERE	Filters records based on a condition	`sql SELECT Name, Email FROM Customers WHERE CustomerID = 1;`
UPDATE	Modifies existing data in a table	`sql UPDATE Customers SET Email = 'john.doe@newdomain.com' WHERE CustomerID = 1;`
DELETE	Removes rows from a table	`sql DELETE FROM Customers WHERE CustomerID = 1;`
DROP TABLE	Deletes the entire table and all its data	`sql DROP TABLE Customers;`

Notes:
- `CustomerID` is the **primary key** and uses `AUTO_INCREMENT` to generate unique IDs automatically.
- Always use `WHERE` in `UPDATE` and `DELETE` to avoid modifying all rows by mistake.
- `DROP TABLE` permanently deletes the table and its data, so use with caution.

---

24 The "Big 6" Elements of a SQL Select Statement

- **SELECT:** Specifies which columns you want to retrieve.

    – Example: `SELECT customer_id, amount FROM sales;`
- **FROM:** Specifies the table you are querying.

- **WHERE:** Filters rows based on conditions.
    – Example: `SELECT * FROM sales WHERE amount > 100 AND sale_date >= '2023-01-01';`

    – Example: `SELECT product_id, amount FROM sales ORDER BY amount DESC;`
- **GROUP BY:** Aggregates rows with the same values into summary rows. Useful for metrics like total sales per customer.

- **HAVING:** Filters results of a `GROUP BY` clause, similar to `WHERE` but for aggregated data.

- **ORDER BY:** Sorts the result set. Use `DESC` for descending and `ASC` for ascending.
- ** LIMIT:**

25 The "Big 6" Elements of a SQL SELECT Statement

When querying data in SQL, the `SELECT` statement is the foundation. It allows you to **specify what data to retrieve, from where, and how to organize it**. The six key elements (plus `LIMIT`) are essential to writing powerful queries.

Remember that, although these six key elements (SELECT, FROM, WHERE, GROUP BY, HAVING, ORDER BY) plus `LIMIT` are essential to writing powerful queries; however, **WHERE, GROUP BY, HAVING, ORDER BY, and LIMIT are optional** depending on the query's purpose.

Example: Let's create the Customers, Products, and Orders tables and insert sample data into each.

[ ]:
```python
# Step 1: Import libraries
import sqlite3
import pandas as pd

# Step 2: Connect to SQLite database
conn = sqlite3.connect('shop_data.db')
cursor = conn.cursor()

# Step 3: Create tables
cursor.execute('''
CREATE TABLE IF NOT EXISTS Customers (
    CustomerID INTEGER PRIMARY KEY,
    Name TEXT,
    Age INTEGER,
    Gender TEXT
);
''')
```

```
cursor.execute('''
CREATE TABLE IF NOT EXISTS Products (
    ProductID INTEGER PRIMARY KEY,
    Product_Name TEXT,
    Price REAL
);
''')

cursor.execute('''
CREATE TABLE IF NOT EXISTS Orders (
    OrderID INTEGER PRIMARY KEY,
    CustomerID INTEGER,
    ProductID INTEGER,
    Quantity INTEGER,
    FOREIGN KEY(CustomerID) REFERENCES Customers(CustomerID),
    FOREIGN KEY(ProductID) REFERENCES Products(ProductID)
);
''')
cursor.execute('''
CREATE TABLE IF NOT EXISTS sales (
    order_id INTEGER PRIMARY KEY,
    customer_id INTEGER,
    sale_date TEXT,
    product_id INTEGER,
    amount REAL
);
''')



# Step 4: Insert sample data

sales_data = [
    (1, 101, '2023-01-01', 1, 150.00),
    (2, 102, '2023-01-02', 2, 200.50),
    (3, 101, '2023-01-03', 3, 75.25),
    (4, 103, '2023-01-04', 1, 150.00),
    (5, 102, '2023-01-05', 2, 200.50)
]

cursor.executemany("INSERT OR IGNORE INTO sales VALUES (?, ?, ?, ?, ?)",␣
  ↪sales_data)


customers_data = [
    (101, 'Alice', 25, 'F'),
    (102, 'Bob', 30, 'M'),
    (103, 'Charlie', 28, 'M'),
    (104, 'Diana', 22, 'F')
]
cursor.executemany("INSERT OR IGNORE INTO Customers VALUES (?, ?, ?, ?)",␣
 ↪customers_data)

products_data = [
    (1, 'Laptop', 1000),
    (2, 'Monitor', 200),
    (3, 'Mouse', 30)
]
cursor.executemany("INSERT OR IGNORE INTO Products VALUES (?, ?, ?)",␣
 ↪products_data)

orders_data = [
    (5001, 101, 1, 1),
    (5002, 102, 2, 2),
    (5003, 101, 3, 1),
    (5004, 103, 1, 1),
    (5005, 104, 2, 1)
]
cursor.executemany("INSERT OR IGNORE INTO Orders VALUES (?, ?, ?, ?)",␣
 ↪orders_data)

conn.commit()
print("Tables created and sample data inserted successfully!")
```

```
Tables created and sample data inserted successfully!
```

25.1 1. SELECT

Specifies which **columns** you want to retrieve from a table. *Example:*
"'sql SELECT customer_id, amount FROM sales;

[ ]:
```
query_select = "SELECT * FROM sales;"
df_select = pd.read_sql_query(query_select, conn)
print(df_select)
```

```
   order_id  customer_id   sale_date  product_id  amount
0         1          101  2023-01-01           1  150.00
1         2          102  2023-01-02           2  200.50
2         3          101  2023-01-03           3   75.25
3         4          103  2023-01-04           1  150.00
4         5          102  2023-01-05           2  200.50
```

25.2 2. FROM

Specifies the table(s) you are querying.

"'sql SELECT customer_id, amount FROM sales;

```
[ ]: query_select = "SELECT customer_id, amount FROM sales;"
     df_select = pd.read_sql_query(query_select, conn)
     print(df_select)
```

```
   customer_id  amount
0          101  150.00
1          102  200.50
2          101   75.25
3          103  150.00
4          102  200.50
```

25.3 3. WHERE

Filters rows based on conditions. Only rows that satisfy the condition are returned.

"'sql SELECT * FROM sales WHERE amount > 100 AND sale_date >= '2023-01-01';

```
[ ]: query_select = "SELECT * FROM sales WHERE amount > 100 AND sale_date >=␣
      ↪'2023-01-01'"
     df_select = pd.read_sql_query(query_select, conn)
     print(df_select)
```

```
   order_id  customer_id   sale_date  product_id  amount
0         1          101  2023-01-01           1   150.0
1         2          102  2023-01-02           2   200.5
2         4          103  2023-01-04           1   150.0
3         5          102  2023-01-05           2   200.5
```

25.4 4. GROUPBY

Aggregates rows with the same values into summary rows, such as totals, averages, or counts.

"'sql SELECT customer_id, SUM(amount) AS total_sales FROM sales GROUP BY customer_id;

```
[ ]: query_select = """
     SELECT customer_id, SUM(amount) AS total_sales
     FROM sales
     GROUP BY customer_id;
     """
     df_select = pd.read_sql_query(query_select, conn)
     print(df_select)
```

```
   customer_id  total_sales
0          101       225.25
1          102       401.00
2          103       150.00
```

25.5 5. HAVING

Filters results after aggregation. Similar to WHERE, but operates on aggregated data.

Remember, while `WHERE` filters **raw rows before aggregation**, `HAVING` filters **groups created by `GROUP BY`**.

Connection to GROUP BY:
- `GROUP BY` creates aggregated groups (e.g., total sales per customer).
- `HAVING` applies conditions on these aggregated values. Without `GROUP BY`, `HAVING` can still work on aggregate functions applied to the entire table.

"'sql SELECT customer_id, SUM(amount) AS total_sales FROM sales GROUP BY customer_id HAVING SUM(amount) > 100;

[ ]:
```
query_select = """
SELECT customer_id, SUM(amount) AS total_sales
FROM sales
GROUP BY customer_id
HAVING SUM(amount) > 100;
"""
df_select = pd.read_sql_query(query_select, conn)
print(df_select)
```

```
   customer_id  total_sales
0          101       225.25
1          102       401.00
2          103       150.00
```

25.6 6. ORDERBY

Sorts the result set by one or more columns.

"'sql SELECT product_id, amount FROM sales ORDER BY amount DESC; – DESC for descending, ASC for ascending

[ ]:
```
query_select = """
SELECT product_id, amount
FROM sales
ORDER BY amount DESC;
"""
df_select = pd.read_sql_query(query_select, conn)
print(df_select)
```

```
   product_id  amount
0           2  200.50
1           2  200.50
2           1  150.00
3           1  150.00
4           3   75.25
```

25.7 Some More SQL Essentials

25.7.1 DISTINCT

Returns **unique values** from a column, removing duplicates.

"'sql – Get unique customer IDs SELECT DISTINCT customer_id FROM sales;

[ ]:
```
query_select = """
SELECT DISTINCT customer_id
FROM sales;
"""
df_select = pd.read_sql_query(query_select, conn)
print(df_select)
```

```
   customer_id
0          101
1          102
2          103
```

25.7.2 COUNT

Counts the number of rows that satisfy a condition.

"'sql – Count total sales SELECT COUNT(*) AS total_sales FROM sales;

– Count number of unique customers SELECT COUNT(DISTINCT customer_id) AS unique_customers FROM sales;

[ ]:
```
query_select = """
SELECT COUNT(DISTINCT customer_id) AS unique_customers
FROM sales;
"""
df_select = pd.read_sql_query(query_select, conn)
print(df_select)
```

```
   unique_customers
0                 3
```

25.7.3 LIMIT

Restricts the number of rows returned, useful for sampling or previewing data. "'sql – Get the 10 most recent sales SELECT * FROM sales ORDER BY sale_date DESC LIMIT 10;

"'sql – Count unique customers but only show the first 5 results SELECT customer_id, COUNT(*) AS num_sales FROM sales GROUP BY customer_id ORDER BY num_sales DESC LIMIT 5;

[ ]:
```
query_select = """
SELECT customer_id, COUNT(*) AS num_sales
FROM sales
GROUP BY customer_id
ORDER BY num_sales DESC
```

```
LIMIT 5;
"""
df_select = pd.read_sql_query(query_select, conn)
print(df_select)
```

```
   customer_id  num_sales
0          102          2
1          101          2
2          103          1
```

Aggregations and Filtering - Use aggregation functions like `SUM()`, `COUNT()`, `AVG()`, `MIN()`, `MAX()` to summarize data.
- `GROUP BY` allows you to compute metrics per category (e.g., total sales per customer).
- `HAVING` filters aggregated results (useful when you want to filter groups, unlike `WHERE` which filters raw rows).

"'sql – Find total sales per customer SELECT customer_id, SUM(amount) AS total_sales FROM sales GROUP BY customer_id;

– Find customers with total sales greater than 200 SELECT customer_id, SUM(amount) AS total_sales FROM sales GROUP BY customer_id HAVING SUM(amount) > 200;

[ ]:
```
query_select = """
SELECT customer_id, SUM(amount) AS total_sales
FROM sales
GROUP BY customer_id
HAVING SUM(amount) > 200;
"""
df_select = pd.read_sql_query(query_select, conn)
print(df_select)
```

```
   customer_id  total_sales
0          101       225.25
1          102       401.00
```

26 SQL JOINs: Combining Data from Multiple Tables

In relational databases, data is often split across multiple tables. **JOINs** allow you to combine rows from two or more tables based on related columns (usually keys).

26.1 1. INNER JOIN

Returns only the rows where there is a **match in both tables**.

"'sql SELECT o.Order_ID, c.Name, p.Product_Name FROM Orders o INNER JOIN Customers c ON o.Customer_ID = c.CustomerID INNER JOIN Products p ON o.Product_ID = p.ProductID;

26.2 2. LEFT JOIN (or LEFT OUTER JOIN)

Returns **all rows from the left table**, and **matched rows from the right table**. If there is no match, the right table columns return NULL.

"'sql SELECT o.Order_ID, c.Name, p.Product_Name FROM Orders o INNER JOIN Customers c ON o.Customer_ID = c.CustomerID INNER JOIN Products p ON o.Product_ID = p.ProductID;

Here:

- All customers are shown, even if they haven't placed any orders.
- Orders columns for customers with no orders will be NULL.

26.3 3. RIGHT JOIN (or RIGHT OUTER JOIN)

Returns all **rows from the right table, and matched rows from the left table**. If there is no match, the left table columns return NULL.

"'sql SELECT o.Order_ID, c.CustomerID, c.Name FROM Orders o RIGHT JOIN Customers c ON o.Customer_ID = c.CustomerID;

Here, All customers appear, even if they have no orders (similar to LEFT JOIN but reversed table order).

26.4 4. FULL OUTER JOIN

Returns **all rows from both tables**, with NULL for missing matches on either side.

"'sql SELECT c.CustomerID, c.Name, o.Order_ID FROM Customers c FULL OUTER JOIN Orders o ON c.CustomerID = o.Customer_ID;

Here, we combines the effect of LEFT and RIGHT JOIN. Customers without orders and orders without customers are included with NULL in the missing columns.

Below is a full example of creating and manipulating a table in SQL.

###EXAMPLE:

[ ]:
```python
# Step 1: Import necessary libraries
import sqlite3
import pandas as pd

# Step 2: Connect to a SQLite database
conn = sqlite3.connect('sales_data.db')
cursor = conn.cursor()

# Step 3: Create tables
cursor.execute('''
CREATE TABLE IF NOT EXISTS sales (
    order_id INTEGER PRIMARY KEY,
    customer_id INTEGER,
    product_id INTEGER,
    sale_date TEXT,
    amount REAL
);
''')

cursor.execute('''
CREATE TABLE IF NOT EXISTS products (
    product_id INTEGER PRIMARY KEY,
    product_name TEXT
);
''')

# Step 4: Insert data
sales_data = [
    (1, 101, 1, '2023-01-01', 150.00),
    (2, 102, 2, '2023-01-02', 200.50),
    (3, 101, 3, '2023-01-03', 75.25),
    (4, 103, 1, '2023-01-04', 150.00),
    (5, 102, 2, '2023-01-05', 200.50)
]
cursor.executemany("INSERT OR IGNORE INTO sales VALUES (?, ?, ?, ?, ?)",␣
 ↪sales_data)

products_data = [
    (1, 'Laptop'),
    (2, 'Monitor'),
    (3, 'Mouse')
]
cursor.executemany("INSERT OR IGNORE INTO products VALUES (?, ?)",␣
 ↪products_data)

conn.commit()
print("Database populated successfully!")

# Step 5: Simple SQL query
query_1 = "SELECT * FROM sales WHERE amount > 150;"
df_high_sales = pd.read_sql_query(query_1, conn)
print("\n--- Sales with Amount > $150 ---")
print(df_high_sales)

# Step 6: JOIN query
query_2 = """
SELECT s.order_id, s.sale_date, s.amount, p.product_name
FROM sales AS s
JOIN products AS p
```

```
ON s.product_id = p.product_id;
"""
df_sales = pd.read_sql_query(query_2, conn)
print("\n--- Sales with Product Names ---")
print(df_sales)

# Step 7: GROUP BY query
query_3 = """
SELECT customer_id, SUM(amount) AS total_amount
FROM sales
GROUP BY customer_id
ORDER BY total_amount DESC;
"""
df_summary = pd.read_sql_query(query_3, conn)
print("\n--- Total Sales per Customer ---")
print(df_summary)

# Step 8: HAVING query (filter aggregated results)
query_4 = """
SELECT customer_id, SUM(amount) AS total_amount
FROM sales
GROUP BY customer_id
HAVING SUM(amount) > 200
ORDER BY total_amount DESC;
"""
df_having = pd.read_sql_query(query_4, conn)
print("\n--- Customers with Total Sales > $200 ---")
print(df_having)

# Step 9: Close connection
conn.close()
print("\nConnection to database closed.")
```

```
Database populated successfully!

--- Sales with Amount > $150 ---
   order_id  customer_id  product_id   sale_date  amount
0         2          102           2  2023-01-02   200.5
1         5          102           2  2023-01-05   200.5

--- Sales with Product Names ---
   order_id   sale_date  amount product_name
0         1  2023-01-01  150.00       Laptop
1         2  2023-01-02  200.50      Monitor
2         3  2023-01-03   75.25        Mouse
3         4  2023-01-04  150.00       Laptop
4         5  2023-01-05  200.50      Monitor
```

```
--- Total Sales per Customer ---
   customer_id  total_amount
0          102        401.00
1          101        225.25
2          103        150.00

--- Customers with Total Sales > $200 ---
   customer_id  total_amount
0          102        401.00
1          101        225.25

Connection to database closed.
```

27 Key Takeaways: SQL

1. **Relational Databases Structure**
   - Data is organized into **tables**, consisting of **rows** (records) and **columns** (attributes).
   - **Primary Keys (PK)** uniquely identify rows, and **Foreign Keys (FK)** link tables to maintain **referential integrity**.
2. **Core SQL Commands**
   - `CREATE TABLE` – define a new table.
   - `INSERT INTO` – add rows of data.
   - `SELECT` – retrieve data.
   - `WHERE` – filter rows.
   - `UPDATE` / `DELETE` – modify or remove data.
   - `DROP TABLE` – remove a table permanently.
3. **The "Big 6" Elements of a SELECT Statement**
   - **SELECT:** Choose columns.
   - **FROM:** Specify tables.
   - **WHERE:** Filter rows.
   - **GROUP BY:** Aggregate rows.
   - **HAVING:** Filter aggregated results.
   - **ORDER BY:** Sort results.
   - **LIMIT:** Restrict the number of rows returned.
4. **JOINs for Combining Tables**

- **INNER JOIN:** Only matching rows.
- **LEFT JOIN:** All left table rows, matched right table rows.
- **RIGHT JOIN:** All right table rows, matched left table rows.
- **FULL OUTER JOIN:** All rows from both tables, NULL for missing matches.

5. **Aggregations and Filtering**
   - Use `SUM()`, `COUNT()`, `AVG()`, `MIN()`, `MAX()` for aggregation.
   - Use `GROUP BY` to summarize data per category.
   - Use `HAVING` to filter **after aggregation** (cannot use WHERE for aggregated results).
6. **SQL in Python with Pandas**
   - `sqlite3` allows creating a **lightweight database** in Colab.
   - Use `pd.read_sql_query()` to **load SQL query results directly into a DataFrame** for analysis.
   - Combining SQL + Pandas enables **powerful data workflows** in Python.
7. **Best Practices**
   - Always use **WHERE** when updating or deleting rows.
   - Use **table aliases** for readability in JOINs.
   - Use **LIMIT** when exploring large datasets to preview data efficiently.
   - Test queries on **sample data** before running on full datasets.

Conclusion:
By mastering table creation, data insertion, SELECT statements, JOINs, aggregation, and integration with Pandas, you can perform **complex data analysis** efficiently in SQL and Python. This chapter lays the foundation for building **real-world data pipelines and analytical workflows**.

27.1 Knowledge Check

Loading…

Loading…