Practical 5: Data Loading, Storage and File Formats

This notebook explores various methods for loading, storing, and working with different file formats in data analysis.

In [19]:
```
# Import required libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import json
import os

# Display version information
print(f"Pandas version: {pd.__version__}")
print(f"NumPy version: {np.__version__}")
```

```
Pandas version: 2.2.3
NumPy version: 2.2.4
```

1. Reading and Writing CSV Files

CSV (Comma Separated Values) is one of the most common formats for storing tabular data.

In [20]:
```
# Creating sample data for CSV
data = {
    'Name': ['John', 'Emma', 'Michael', 'Sophia', 'William'],
    'Age': [25, 28, 32, 22, 35],
    'City': ['New York', 'Boston', 'Chicago', 'Miami', 'Seattle'],
    'Salary': [75000, 82000, 95000, 65000, 105000]
}

# Create a DataFrame
df = pd.DataFrame(data)
print("Original DataFrame:")
display(df)
```

```
Original DataFrame:
```

	Name	Age	City	Salary
0	John	25	New York	75000
1	Emma	28	Boston	82000
2	Michael	32	Chicago	95000
3	Sophia	22	Miami	65000
4	William	35	Seattle	105000

In [21]:
```
# Writing to CSV file
df.to_csv('employee_data.csv', index=False)
print("CSV file created successfully!")

# Reading from CSV file
df_from_csv = pd.read_csv('employee_data.csv')
print("\nDataFrame read from CSV:")
display(df_from_csv)
```

```
CSV file created successfully!

DataFrame read from CSV:
```

	Name	Age	City	Salary
0	John	25	New York	75000
1	Emma	28	Boston	82000
2	Michael	32	Chicago	95000
3	Sophia	22	Miami	65000
4	William	35	Seattle	105000

2. Working with Excel Files

Excel files are widely used in business. Pandas makes it easy to read and write Excel files.

In [22]:
```
# Install openpyxl package which is required for pandas to work with Excel f
try:
    import openpyxl
    print("openpyxl is already installed.")
except ImportError:
    print("Installing openpyxl...")
    !pip install openpyxl
    print("openpyxl installed successfully!")
```

```
openpyxl is already installed.
```

In [23]:
```
# Get the existing Excel file in the workspace
excel_file = 'Student_Records.xlsx'

# Check if the file exists
if os.path.exists(excel_file):
    # Reading Excel file
    df_excel = pd.read_excel(excel_file)
    print(f"Data from {excel_file}:")
    display(df_excel)
else:
    print(f"File {excel_file} not found. Creating a new Excel file.")
    # Writing to Excel file
    df.to_excel('new_employee_data.xlsx', index=False, sheet_name='Employees
    print("Excel file created successfully!")
```

```
Data from Student_Records.xlsx:
```

	Sr. No.	Enrolment Number	Department Name	Student Name	Current Semester	Email ID
0	1	ENR00001	Information Technology	Student_1	6	student1@example.com
1	2	ENR00002	Information Technology	Student_2	6	student2@example.com
2	3	ENR00003	Information Technology	Student_3	6	student3@example.com
3	4	ENR00004	Information Technology	Student_4	6	student4@example.com
4	5	ENR00005	Information Technology	Student_5	6	student5@example.com
5	6	ENR00006	Information Technology	Student_6	6	student6@example.com
6	7	ENR00007	Information Technology	Student_7	6	student7@example.com
7	8	ENR00008	Information Technology	Student_8	6	student8@example.com
8	9	ENR00009	Information Technology	Student_9	6	student9@example.com
9	10	ENR00010	Information Technology	Student_10	6	student10@example.com
10	11	ENR00011	Information Technology	Student_11	6	student11@example.com
11	12	ENR00012	Information Technology	Student_12	6	student12@example.com
12	13	ENR00013	Information Technology	Student_13	6	student13@example.com
13	14	ENR00014	Information Technology	Student_14	6	student14@example.com
14	15	ENR00015	Information Technology	Student_15	6	student15@example.com
15	16	ENR00016	Information Technology	Student_16	6	student16@example.com
16	17	ENR00017	Information Technology	Student_17	6	student17@example.com
17	18	ENR00018	Information Technology	Student_18	6	student18@example.com
18	19	ENR00019	Information Technology	Student_19	6	student19@example.com
19	20	ENR00020	Information Technology	Student_20	6	student20@example.com

3. JSON Format

JSON (JavaScript Object Notation) is a lightweight data-interchange format that is easy for humans to read and write.

In [24]:
```
# Converting DataFrame to JSON
json_data = df.to_json(orient='records')
print("JSON Data:")
print(json_data)

# Writing to a JSON file
df.to_json('employee_data.json', orient='records')
print("\nJSON file created successfully!")

# Reading from a JSON file
df_from_json = pd.read_json('employee_data.json')
print("\nDataFrame read from JSON:")
display(df_from_json)
```

```
JSON Data:
[{"Name":"John","Age":25,"City":"New York","Salary":75000},{"Name":"Emma","A
ge":28,"City":"Boston","Salary":82000},{"Name":"Michael","Age":32,"City":"Ch
icago","Salary":95000},{"Name":"Sophia","Age":22,"City":"Miami","Salary":650
00},{"Name":"William","Age":35,"City":"Seattle","Salary":105000}]

JSON file created successfully!

DataFrame read from JSON:
```

	Name	Age	City	Salary
0	John	25	New York	75000
1	Emma	28	Boston	82000
2	Michael	32	Chicago	95000
3	Sophia	22	Miami	65000
4	William	35	Seattle	105000

4. NumPy Binary Format

NumPy provides functionality to save and load arrays in binary format, which is efficient for large numerical datasets.

In [25]:
```
# Create a NumPy array
arr = np.array([10, 20, 30, 40, 50])
print("Original NumPy array:")
print(arr)

# Save array to .npy file
```

```python
np.save('array_data.npy', arr)
print("\nArray saved to .npy file successfully!")

# Load array from .npy file
loaded_arr = np.load('array_data.npy')
print("\nArray loaded from .npy file:")
print(loaded_arr)
```

```
Original NumPy array:
[10 20 30 40 50]

Array saved to .npy file successfully!

Array loaded from .npy file:
[10 20 30 40 50]
```

In [26]:

```python
# Loading an existing NumPy array file if it exists
try:
    existing_array = np.load('some_array.npy')
    print("Loaded existing array from workspace:")
    print(existing_array)
except:
    print("Could not load 'some_array.npy'")

# Try loading the NPZ archive if it exists
try:
    archive = np.load('array_archive.npz')
    print("\nLoaded NPZ archive from workspace:")
    print("Arrays in the archive:")
    print(list(archive.files))
    for file in archive.files:
        print(f"\nArray '{file}':")
        print(archive[file])
except:
    print("\nCould not load 'array_archive.npz'")
```

```
Loaded existing array from workspace:
[0 1 2 3 4 5 6 7 8 9]

Loaded NPZ archive from workspace:
Arrays in the archive:
['a', 'b']

Array 'a':
[0 1 2 3 4 5 6 7 8 9]

Array 'b':
[0 1 2 3 4 5 6 7 8 9]
```

5. Creating and Loading Multiple Arrays with NPZ Format

In [27]:

```python
# Create multiple arrays
array1 = np.array([1, 2, 3, 4, 5])
array2 = np.random.rand(3, 3)
```

```python
array3 = np.zeros((2, 4))

# Save multiple arrays to a single .npz file
np.savez('multiple_arrays.npz', array1=array1, array2=array2, array3=array3)
print("Multiple arrays saved to .npz file successfully!")

# Load arrays from .npz file
loaded_arrays = np.load('multiple_arrays.npz')
print("\nLoaded arrays from .npz file:")
print("Available arrays:", loaded_arrays.files)

print("\nArray1:")
print(loaded_arrays['array1'])
print("\nArray2:")
print(loaded_arrays['array2'])
print("\nArray3:")
print(loaded_arrays['array3'])
```

```
Multiple arrays saved to .npz file successfully!

Loaded arrays from .npz file:
Available arrays: ['array1', 'array2', 'array3']

Array1:
[1 2 3 4 5]

Array2:
[[0.78675189 0.33494394 0.87334975]
 [0.01240706 0.89054059 0.07736432]
 [0.87632852 0.54557831 0.55906091]]

Array3:
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]]
```

6. HDF5 Format with PyTables

HDF5 is a file format designed for storing and managing large amounts of data. PyTables is a package for managing hierarchical datasets.

In [28]:

```python
# Install pytables if not already installed
try:
    import tables
    print("PyTables is already installed.")
except ImportError:
    print("Installing PyTables...")
    !pip install tables
    import tables
    print("PyTables installed successfully!")
```

```
PyTables is already installed.
```

In [29]:

```python
# Create a HDF5 file using PyTables
df_expanded = pd.DataFrame({
    'A': np.random.randn(100000),
```

```
    'B': np.random.randn(100000),
    'C': np.random.randn(100000),
    'D': np.random.randn(100000)
})

# Save DataFrame to HDF5 format
df_expanded.to_hdf('data.h5', key='df', mode='w')
print("DataFrame saved to HDF5 file successfully!")

# Read DataFrame from HDF5 format
df_from_hdf = pd.read_hdf('data.h5', 'df')
print("\nDataFrame head from HDF5:")
display(df_from_hdf.head())

print(f"\nShape of the DataFrame: {df_from_hdf.shape}")
```

```
DataFrame saved to HDF5 file successfully!

DataFrame head from HDF5:
```

	A	B	C	D
0	1.256710	-1.688748	1.562620	1.527452
1	-0.873625	1.433457	1.517999	-1.119706
2	0.235480	-0.485339	0.304667	0.846970
3	-0.514288	0.281177	1.614501	1.280056
4	1.588940	0.210281	1.913426	0.585414

```
Shape of the DataFrame: (100000, 4)
```

7. SQL Database Connection

Pandas can interact with SQL databases through SQLAlchemy.

In [30]:

```
# Import SQLite module
import sqlite3

# Create a connection to a SQLite database (in-memory for demonstration)
conn = sqlite3.connect(':memory:')

# Write DataFrame to SQL database
df.to_sql('employees', conn, index=False)
print("DataFrame written to SQL database successfully!")

# Read from SQL database
query = "SELECT * FROM employees WHERE Age > 25"
df_from_sql = pd.read_sql(query, conn)
print("\nEmployees older than 25:")
display(df_from_sql)

# Close the connection
conn.close()
```

```
DataFrame written to SQL database successfully!

Employees older than 25:
```

	Name	Age	City	Salary
0	Emma	28	Boston	82000
1	Michael	32	Chicago	95000
2	William	35	Seattle	105000

8. Dealing with Different File Encodings

In [31]:

```
# Create a DataFrame with special characters
data_special = {
    'Name': ['José', 'Müller', 'François', 'Søren', 'Jürgen'],
    'Country': ['Spain', 'Germany', 'France', 'Denmark', 'Germany']
}

df_special = pd.DataFrame(data_special)
print("DataFrame with special characters:")
display(df_special)

# Save with UTF-8 encoding
df_special.to_csv('special_chars_utf8.csv', index=False, encoding='utf-8')
print("\nSaved with UTF-8 encoding")

# Read back with UTF-8 encoding
df_utf8 = pd.read_csv('special_chars_utf8.csv', encoding='utf-8')
print("\nRead with UTF-8 encoding:")
display(df_utf8)
```

```
DataFrame with special characters:
```

	Name	Country
0	José	Spain
1	Müller	Germany
2	François	France
3	Søren	Denmark
4	Jürgen	Germany

```
Saved with UTF-8 encoding

Read with UTF-8 encoding:
```

	Name	Country
0	José	Spain
1	Müller	Germany
2	François	France
3	Søren	Denmark
4	Jürgen	Germany

9. Working with Compressed Files

In [32]:

```
# Generate a larger DataFrame for compression demonstration
rows = 10000
large_df = pd.DataFrame({
    'id': range(rows),
    'value': np.random.randn(rows),
    'category': np.random.choice(['A', 'B', 'C', 'D'], size=rows)
})

# Check uncompressed size
large_df.to_csv('large_data.csv', index=False)
uncompressed_size = os.path.getsize('large_data.csv')
print(f"Uncompressed size: {uncompressed_size / 1024:.2f} KB")

# Check gzip compressed size
large_df.to_csv('large_data.csv.gz', index=False, compression='gzip')
gzip_size = os.path.getsize('large_data.csv.gz')
print(f"Gzip compressed size: {gzip_size / 1024:.2f} KB")

# Check zip compressed size
large_df.to_csv('large_data.csv.zip', index=False, compression='zip')
zip_size = os.path.getsize('large_data.csv.zip')
print(f"Zip compressed size: {zip_size / 1024:.2f} KB")

# Calculate compression ratios
print(f"\nGzip compression ratio: {uncompressed_size / gzip_size:.2f}x")
print(f"Zip compression ratio: {uncompressed_size / zip_size:.2f}x")

# Read from compressed file
df_from_gzip = pd.read_csv('large_data.csv.gz', compression='gzip')
print("\nSuccessfully read data from compressed gzip file.")
print("First 5 rows:")
display(df_from_gzip.head())
```

```
Uncompressed size: 268.91 KB
Gzip compressed size: 118.69 KB
Gzip compressed size: 118.69 KB
Zip compressed size: 118.92 KB

Gzip compression ratio: 2.27x
Zip compression ratio: 2.26x

Successfully read data from compressed gzip file.
First 5 rows:
Zip compressed size: 118.92 KB

Gzip compression ratio: 2.27x
Zip compression ratio: 2.26x

Successfully read data from compressed gzip file.
First 5 rows:
```

	id	value	category
0	0	0.032863	B
1	1	0.900876	A
2	2	0.393806	C
3	3	1.658165	B
4	4	0.305339	A