Pandas

Pandas is an open-source Python library widely used for data manipulation and analysis.

It provides data structures and functions for efficiently handling structured data, such as tabular data (similar to a spreadsheet or SQL table).

In [1]:
```
In [1]: import numpy as np

In [2]: import pandas as pd
```

Object creation in pandas

Object creation in pandas typically involves creating and working with two main data structures:

Series and DataFrame.

1- Series

A one-dimensional labeled array that can hold data of any type.

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

# Creating a Series from a list
data = [10,20,30,40,50]
series = pd.Series(data)
series
```

Out[3]:
```
0    10
1    20
2    30
3    40
4    50
dtype: int64
```

In [7]:
```
s = pd.Series([1, 3, 5, np.nan, 6, 8])
s
```

Out[7]:
```
0    1.0
1    3.0
2    5.0
3    NaN
4    6.0
5    8.0
dtype: float64
```

2- DataFrame

A two-dimensional labeled data structure with columns that can hold different types of data.

In [5]:
```
# Creating a DataFrame from a dictionary
data = {'Name': ['Uzair', 'Rana', 'Muslim'],
        'age': [23,30,21]}
df = pd.DataFrame(data)
df
```

Out[5]:

	Name	age
0	Uzair	23
1	Rana	30
2	Muslim	21

In [6]:
```
# DataFrame with more control (using index and columns parameters)

data = [[10,20,30],
        [40,50,60]]
columns = ['A', 'B', 'C']
index = ['Row 1', 'Row 2']
df = pd.DataFrame(data, columns=columns, index = index)
df
```

Out[6]:

	A	B	C
Row 1	10	20	30
Row 2	40	50	60

Viewing data

You can use various methods to view data in pandas, such as head, tail, and sample.

1- head

View the first few rows of the DataFrame.

In [9]:
```
import pandas as pd

data = {'Name': ['Uzair', 'Rana', 'Muslim','ahmad','ali','umar'],
        'age': [23,30,21,24,25,26]}
df = pd.DataFrame(data)

print(df.head())
```

```
     Name  age
0   Uzair   23
1    Rana   30
2  Muslim   21
3   ahmad   24
4     ali   25
```

2- tail

View the last few rows of the DataFrame.

In [10]:
```
print(df.tail(2))  # View the last 2 rows
```

```
   Name  age
4   ali   25
5  umar   26
```

3- sample

View a random sample of rows from the DataFrame.

In [11]:
```
print(df.sample(2))  # View a random sample of 2 rows
```

```
     Name  age
2  Muslim   21
4     ali   25
```

In [12]:
```
# shows a quick statistics summary of data
df.describe()
```

Out[12]:

	age
count	6.000000
mean	24.833333
std	3.060501
min	21.000000
25%	23.250000
50%	24.500000
75%	25.750000
max	30.000000

In [13]:
```
# transposing your data
df.T
```

Out[13]:

	0	1	2	3	4	5
Name	Uzair	Rana	Muslim	ahmad	ali	umar
age	23	30	21	24	25	26

Selection

Selection in pandas involves accessing specific data within a DataFrame using various methods, such as loc and iloc.

1- loc

Select data by label (row and column labels).

In [19]:
```
import pandas as pd

data = {'Name': ['Uzair', 'Rana', 'Muslim'],
        'age': [23,30,21]}
df = pd.DataFrame(data, index =['A', 'B', 'C'])
df
```

Out[19]:

	Name	age
A	Uzair	23
B	Rana	30
C	Muslim	21

In [20]:
```python
# Select data using label-based indexing
selected_row = df.loc['B']  # selecxt a specific row
selected_cell = df.loc['A', 'age']  # select a specific cell
selected_slice = df.loc['A':'B', 'Name':'Age'] # select a range of rows and columns
print(selected_row)
print(selected_cell)
print(selected_slice)
```

```
Name    Rana
age       30
Name: B, dtype: object
23
Empty DataFrame
Columns: []
Index: [A, B]
```

2- iloc

Select data by integer location (row and column indices).

In [22]:
```python
selected_row = df.iloc[1]            # Select a specific row by index
selected_cell = df.iloc[0, 1]        # Select a specific cell by indices
selected_slice = df.iloc[0:2, 0:2]   # Select a range of rows and columns by indices
print(selected_row)
print(selected_cell)
print(selected_slice)
```

```
Name    Rana
age       30
Name: B, dtype: object
23
    Name  age
A  Uzair   23
B   Rana   30
```

3- Boolean Indexing

Select data based on a condition.

In [23]:
```
# Select rows where Age is greater than 25
slected_data = df[df['age'] > 25 ]
slected_data
```

Out[23]:

	Name	age
B	Rana	30

In [26]:
```
# select specific column by column name
df["Name"]
```

Out[26]:
```
A     Uzair
B      Rana
C    Muslim
Name: Name, dtype: object
```

In [27]:
```
# select rows by index
df[0:3]
```

Out[27]:

	Name	age
A	Uzair	23
B	Rana	30
C	Muslim	21

Missing Data

np.nan stands for "Not a Number" and is a special floating-point value used in NumPy and pandas to represent missing or undefined data. It is often used to indicate missing or null values in numeric arrays and data structures.

Dealing with missing data

Dealing with missing data is a crucial aspect of data analysis. Pandas provides various methods to handle missing data, such as using dropna, fillna, and isna

dropna Remove rows or columns with missing values.

fillna Fill missing values with specified values.

isna Check for missing values.

In [30]:
```
# dropna
import pandas as pd
import numpy as np

data = {'A': [1, 2, np.nan, 4],
        'B': [5, np.nan, np.nan, 8]}
df = pd.DataFrame(data)
df
```

Out[30]:

	A	B
0	1.0	5.0
1	2.0	NaN
2	NaN	NaN
3	4.0	8.0

In [31]:
```
# Drop rows with any missing values
df_cleaned = df.dropna()

# Drop columns with any missing values
df_cleaned_columns = df.dropna(axis=1)
print(df_cleaned)
print(df_cleaned_columns)
```

```
     A    B
0  1.0  5.0
3  4.0  8.0
Empty DataFrame
Columns: []
Index: [0, 1, 2, 3]
```

In [33]:
```
# fillna

# Fill missing values with a specific value
df_filled = df.fillna(0)

# Fill missing values with column means

column_means = df.mean()
df_filled_with_means = df.fillna(column_means)
print(df_filled)
print(df_filled_with_means)
```

```
     A    B
0  1.0  5.0
1  2.0  0.0
2  0.0  0.0
3  4.0  8.0
          A    B
0  1.000000  5.0
1  2.000000  6.5
2  2.333333  6.5
3  4.000000  8.0
```

In [34]:
```python
# isna

# Check for missing values in the DataFrame
missing_values = df.isna()

# Check for missing values in a specific column
missing_values_column = df['A'].isna()
print(missing_values)
print(missing_values_column)
```

```
       A      B
0  False  False
1  False   True
2   True   True
3  False  False
0    False
1    False
2     True
3    False
Name: A, dtype: bool
```

Operations in pandas

Operations in pandas refer to various data manipulation tasks that can be performed on DataFrame and Series objects, such as arithmetic operations, aggregation, and transformations.

1- Arithmetic Operations

In [36]:
```python
import pandas as pd

data = {'A': [10, 20, 30],
        'B': [5, 15, 25]}
df = pd.DataFrame(data)

# Adding two columns element-wise
df['C'] = df['A'] + df['B']

# Performing arithmetic operation on a column
df['D'] = df['C'] * 2

print(df['C'])
print(df['D'])
```

```
0    15
1    35
2    55
Name: C, dtype: int64
0     30
1     70
2    110
Name: D, dtype: int64
```

2- Aggregation

In [38]:
```
# Calculating the mean of each column
column_means = df.mean()

# Calculating the sum of each row
row_sums = df.sum(axis=1)

print(column_means)
print(row_sums)
```

```
A    20.0
B    15.0
C    35.0
D    70.0
dtype: float64
0     60
1    140
2    220
dtype: int64
```

3- Transformations

In [39]:
```
# Applying a function element-wise to a column
df['A_squared'] = df['A'].apply(lambda x: x**2)

# Using the applymap function to apply a function to all elements

df_squared = df.applymap(lambda x: x**2)
print(df['A_squared'])
print(df_squared)
```

```
0    100
1    400
2    900
Name: A_squared, dtype: int64
     A    B     C      D  A_squared
0  100   25   225    900      10000
1  400  225  1225   4900     160000
2  900  625  3025  12100     810000
```

Stats

In pandas, you can perform various statistical computations on DataFrame and Series objects using built-in methods.

1- mean

Calculate the mean (average) of data.

In [40]:
```
import pandas as pd

data = {'A': [10, 20, 30, 40, 50]}
df = pd.DataFrame(data)


mean_value = df['A'].mean()
print(df)
print(mean_value)
```

```
    A
0  10
1  20
2  30
3  40
4  50
30.0
```

2- median

Calculate the median (middle value) of data.

In [42]:
```
median_value = df['A'].median()
median_value
```

Out[42]:
```
30.0
```

3- std

Calculate the standard deviation of data.

In [43]:
```
std_deviation = df['A'].std()
std_deviation
```

Out[43]:
```
15.811388300841896
```

4- describe

Generate summary statistics of data.

In [44]:
```
summary_stats = df.describe()
summary_stats
```

Out[44]:

	A
count	5.000000
mean	30.000000
std	15.811388
min	10.000000
25%	20.000000
50%	30.000000
75%	40.000000
max	50.000000

5- correlation

Compute the correlation between columns.

In [45]:
```
correlation_matrix = df.corr()
correlation_matrix
```

Out[45]:

	A
A	1.0

String Methods

In pandas, string methods can be applied to Series containing string data to perform various text-based operations. These methods allow you to manipulate, clean, and analyze textual data efficiently.

In [46]:
```
import pandas as pd

data = {'Name': ['Ali', 'Bilal', 'Moeen']}
df = pd.DataFrame(data)

# Convert all names to uppercase

df['Uppercase Name'] = df['Name'].str.upper()
print(df)
print(df['Uppercase Name'])
```

```
    Name Uppercase Name
0    Ali            ALI
1  Bilal          BILAL
2  Moeen          MOEEN
0      ALI
1    BILAL
2    MOEEN
Name: Uppercase Name, dtype: object
```

In [47]:
```
df['Lowercase Name'] = df['Name'].str.lower()
print(df['Lowercase Name'])
```

```
0      ali
1    bilal
2    moeen
Name: Lowercase Name, dtype: object
```

Merge

In pandas, the merge function is used to combine two or more DataFrames based on common columns or indices. This operation is similar to SQL JOIN operations.

In [49]:
```
import pandas as pd

# Create the employee DataFrame
employee_data = {'EmployeeID': [1, 2, 3, 4],
                 'Name': ['Ali', 'Bilal', 'umar', 'Dauood'],
                 'DepartmentID': [101, 102, 101, 103]}
employees = pd.DataFrame(employee_data)

# Create the department DataFrame
department_data = {'DepartmentID': [101, 102, 103],
                   'DepartmentName': ['HR', 'IT', 'Finance']}
departments = pd.DataFrame(department_data)

# Merge the DataFrames based on 'DepartmentID'

merged_df = pd.merge(employees, departments, on = 'DepartmentID')
print(employees)
print(departments)
print(merged_df)
```

```
   EmployeeID    Name  DepartmentID
0           1     Ali           101
1           2   Bilal           102
2           3    umar           101
3           4  Dauood           103
   DepartmentID DepartmentName
0           101             HR
1           102             IT
2           103        Finance
   EmployeeID    Name  DepartmentID DepartmentName
0           1     Ali           101             HR
1           3    umar           101             HR
2           2   Bilal           102             IT
3           4  Dauood           103        Finance
```

Concatenating

Concatenating in pandas involves combining two or more DataFrames along rows or columns. The concat function is used for this purpose.

In [50]:
```python
import pandas as pd

data1 = {'A': [1, 2, 3],
         'B': [4, 5, 6]}
df1 = pd.DataFrame(data1)

data2 = {'A': [7, 8, 9],
         'B': [10, 11, 12]}
df2 = pd.DataFrame(data2)

# Concatenate along rows

concatenated_df = pd.concat([df1, df2])
print(concatenated_df)
```

```
   A   B
0  1   4
1  2   5
2  3   6
0  7  10
1  8  11
2  9  12
```

Joining

Joining in pandas refers to combining DataFrames based on index or columns using the join function. It's similar to merging but is based on the index or specified columns.

In [62]:
```
import pandas as pd

data1 = {'A': [1, 2, 3],
         'B': [4, 5, 6]}
df1 = pd.DataFrame(data1, index=['row1', 'row2', 'row3'])

data2 = {'C': [7, 8, 9],
         'D': [10, 11, 12]}
df2 = pd.DataFrame(data2, index=['row1', 'row2', 'row3'])

print(df1)
print(df2)
# Join based on index

joined_df = df.join(df2)
```

```
      A  B
row1  1  4
row2  2  5
row3  3  6
      C   D
row1  7  10
row2  8  11
row3  9  12
```

Grouping

Grouping in pandas involves splitting data into groups based on some criteria and then applying a function to each group. The groupby function is used for this purpose.

In [64]:
```python
import pandas as pd

data = {'Category': ['Electronics', 'Clothing', 'Electronics', 'Clothing', 'Electronics'],
        'Product': ['Laptop', 'Shirt', 'Phone', 'Pants', 'Tablet'],
        'Price': [1000, 25, 800, 30, 300]}
df = pd.DataFrame(data)

# Group by 'Category' and calculate total sales (sum of 'Price') in each category

grouped_df = df.groupby('Category')['Price'].sum()
print(df)
print(grouped_df)
```

```
      Category Product  Price
0  Electronics  Laptop   1000
1     Clothing   Shirt     25
2  Electronics   Phone    800
3     Clothing   Pants     30
4  Electronics  Tablet    300
Category
Clothing         55
Electronics    2100
Name: Price, dtype: int64
```

Reshaping

Reshaping in pandas involves changing the layout of your data, such as converting between wide and long formats. The pivot and melt functions are used for reshaping data.

The pivot function is used to reshape data from long to wide format, and the melt function is used to reshape data from wide to long format.

In [65]:
```
import pandas as pd

data = {'Month': ['Jan', 'Feb', 'Mar'],
        'Product_A': [100, 150, 200],
        'Product_B': [50, 80, 120]}
df_wide = pd.DataFrame(data)

df_wide
```

Out[65]:

	Month	Product_A	Product_B
0	Jan	100	50
1	Feb	150	80
2	Mar	200	120

In [66]:
```
# Convert wide format to long format using the melt function

df_long = pd.melt(df_wide, id_vars=['Month'], var_name= 'Product', value_name = 'Sales')
print(df_long)
```

```
  Month    Product  Sales
0   Jan  Product_A    100
1   Feb  Product_A    150
2   Mar  Product_A    200
3   Jan  Product_B     50
4   Feb  Product_B     80
5   Mar  Product_B    120
```

Stacking

Stacking in pandas refers to pivoting the innermost level of column labels to the innermost level of row labels, effectively transforming columns into rows. The stack function is used for this purpose.

The stack function in pandas is used to pivot the columns of a DataFrame into rows

In [67]:
```
import pandas as pd

data = {'Month': ['Jan', 'Feb', 'Mar'],
        ('Sales', 'Product_A'): [100, 150, 200],
        ('Sales', 'Product_B'): [50, 80, 120]}
df = pd.DataFrame(data)
df
```

Out[67]:

	Month	(Sales, Product_A)	(Sales, Product_B)
0	Jan	100	50
1	Feb	150	80
2	Mar	200	120

In [69]:
```
# Stack the 'Sales' level of columns into rows

stacked_df = df.stack()
print(stacked_df)
```

```
0  Month                 Jan
   (Sales, Product_A)    100
   (Sales, Product_B)     50
1  Month                 Feb
   (Sales, Product_A)    150
   (Sales, Product_B)     80
2  Month                 Mar
   (Sales, Product_A)    200
   (Sales, Product_B)    120
dtype: object
```

Pivot tables

Pivot tables in pandas allow you to summarize and analyze data by creating a new table from an existing DataFrame based on specified columns and aggregation functions. The pivot_table function is used for this purpose.

In [70]:
```
import pandas as pd

data = {'Month': ['Jan', 'Feb', 'Jan', 'Feb'],
        'Product': ['A', 'A', 'B', 'B'],
        'Sales': [100, 150, 50, 80]}
df = pd.DataFrame(data)

df
```

Out[70]:

	Month	Product	Sales
0	Jan	A	100
1	Feb	A	150
2	Jan	B	50
3	Feb	B	80

In [71]:
```
# Create a pivot table to summarize sales by product and month
pivot_table = df.pivot_table(index='Month', columns='Product', values='Sales', aggfunc='sum')
print(pivot_table)
```

```
Product    A   B
Month
Feb      150  80
Jan      100  50
```

Time series

Time series in pandas involves working with data that is indexed or organized by time or date. The DateTimeIndex in pandas makes it convenient to handle time-related data.

In [72]:
```python
import pandas as pd

# Sample time-based data
data = {'Value': [10, 15, 20, 25, 30]}
time_index = pd.to_datetime(['2023-01-01', '2023-01-02', '2023-01-03', '2023-01-04', '2023-01-05'])

# Create a DataFrame with DateTimeIndex

df = pd.DataFrame(data, index = time_index)
print(df)
```

```
            Value
2023-01-01     10
2023-01-02     15
2023-01-03     20
2023-01-04     25
2023-01-05     30
```

Categoricals

Categoricals in pandas are a data type that represents categorical data with a fixed number of distinct values. They provide efficient storage and some performance benefits when working with large datasets.

Categoricals in pandas are a data type used to efficiently represent and manage categorical data.

In [73]:
```python
import pandas as pd

data = {'Category': ['A', 'B', 'A', 'C', 'B']}
df = pd.DataFrame(data)

# Convert 'Category' column to categorical

df['Category'] = df['Category'].astype('category')
print(df['Category'].dtype)
```

```
category
```

In [74]:
```python
import pandas as pd

# Sample data with repeated categories
data = {'Category': ['A', 'B', 'A', 'C', 'B']}
df = pd.DataFrame(data)

# Convert 'Category' column to categorical and set a specific order

df['Category'] = pd.Categorical(df['Category'], categories=['C', 'A', 'B'], ordered = True)
# Display the DataFrame
print(df)
```

```
  Category
0        A
1        B
2        A
3        C
4        B
```

In [75]:
```python
print(df['Category'])
```

```
0    A
1    B
2    A
3    C
4    B
Name: Category, dtype: category
Categories (3, object): ['C' < 'A' < 'B']
```

Plotting

Plotting in pandas allows you to create various types of plots and visualizations directly from DataFrame and Series data using built-in plotting functions.

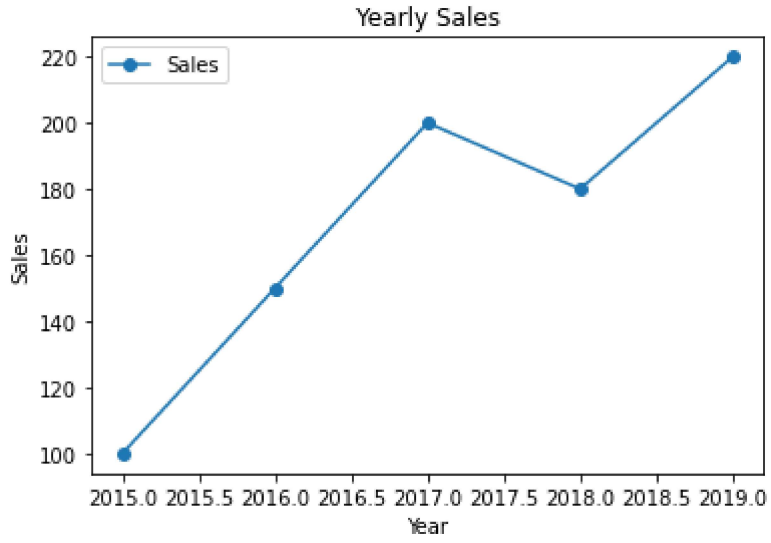
In [76]:
```
import pandas as pd
import matplotlib.pyplot as plt

# Sample data
data = {'Year': [2015, 2016, 2017, 2018, 2019],
        'Sales': [100, 150, 200, 180, 220]}
df = pd.DataFrame(data)

# Create a line plot

df.plot(x='Year', y='Sales', kind='line',marker='o')
plt.title('Yearly Sales')
plt.xlabel('Year')
plt.ylabel('Sales')

plt.show()
```



Importing and exporting data

Importing and exporting data in pandas involves reading data from various file formats (CSV, Excel, SQL, etc.) into DataFrame objects and saving DataFrame data back to files.

Pandas provides functions like read_csv, read_excel, and to_csv for importing and exporting data, respectively.

In [79]:

```
import pandas as pd

# Import data from a CSV file
df_imported = pd.read_csv('Automobile_data.csv')

# Display the imported DataFrame
print(df_imported)
df_imported.head()
```

```
     symboling normalized-losses         make fuel-type aspiration  \
0            3                 ?  alfa-romero       gas        std
1            3                 ?  alfa-romero       gas        std
2            1                 ?  alfa-romero       gas        std
3            2               164         audi       gas        std
4            2               164         audi       gas        std
..         ...               ...          ...       ...        ...
200         -1                95        volvo       gas        std
201         -1                95        volvo       gas      turbo
202         -1                95        volvo       gas        std
203         -1                95        volvo    diesel      turbo
204         -1                95        volvo       gas      turbo

    num-of-doors   body-style drive-wheels engine-location  wheel-base  ...  \
0            two  convertible          rwd           front        88.6  ...
1            two  convertible          rwd           front        88.6  ...
2            two    hatchback          rwd           front        94.5  ...
3           four        sedan          fwd           front        99.8  ...
4           four        sedan          4wd           front        99.4  ...
..           ...          ...          ...             ...         ...  ...
200         four        sedan          rwd           front       109.1  ...
201         four        sedan          rwd           front       109.1  ...
202         four        sedan          rwd           front       109.1  ...
203         four        sedan          rwd           front       109.1  ...
204         four        sedan          rwd           front       109.1  ...

     engine-size  fuel-system  bore  stroke compression-ratio horsepower  \
0            130         mpfi  3.47    2.68               9.0        111
1            130         mpfi  3.47    2.68               9.0        111
2            152         mpfi  2.68    3.47               9.0        154
3            109         mpfi  3.19     3.4              10.0        102
4            136         mpfi  3.19     3.4               8.0        115
..           ...          ...   ...     ...               ...        ...
200          141         mpfi  3.78    3.15               9.5        114
201          141         mpfi  3.78    3.15               8.7        160
202          173         mpfi  3.58    2.87               8.8        134
203          145          idi  3.01     3.4              23.0        106
204          141         mpfi  3.78    3.15               9.5        114

     peak-rpm city-mpg highway-mpg  price
0        5000       21          27  13495
1        5000       21          27  16500
2        5000       19          26  16500
```

```
3        5500        24           30  13950
4        5500        18           22  17450
..        ...       ...          ...    ...
200      5400        23           28  16845
201      5300        19           25  19045
202      5500        18           23  21485
203      4800        26           27  22470
204      5400        19           25  22625

[205 rows x 26 columns]
```

Out[79]:

	symboling	normalized-losses	make	fuel-type	aspiration	num-of-doors	body-style	drive-wheels	engine-location	wheel-base	...	engine-size	fuel-system	bore	stroke	com
0	3	?	alfa-romero	gas	std	two	convertible	rwd	front	88.6	...	130	mpfi	3.47	2.68	
1	3	?	alfa-romero	gas	std	two	convertible	rwd	front	88.6	...	130	mpfi	3.47	2.68	
2	1	?	alfa-romero	gas	std	two	hatchback	rwd	front	94.5	...	152	mpfi	2.68	3.47	
3	2	164	audi	gas	std	four	sedan	fwd	front	99.8	...	109	mpfi	3.19	3.4	
4	2	164	audi	gas	std	four	sedan	4wd	front	99.4	...	136	mpfi	3.19	3.4	

5 rows × 26 columns

HDF5

HDF5 (Hierarchical Data Format version 5) is a file format that supports the storage of large and complex datasets. In pandas, you can use the HDFStore class to work with HDF5 files.

In [80]:
```python
import pandas as pd

# Sample DataFrame
data = {'Name': ['Ali', 'Burhan', 'saeed'],
        'Age': [25, 30, 22]}
df = pd.DataFrame(data)

# Write DataFrame to an HDF5 file

with pd.HDFStore('data.h5', mode='w') as store:
    store['my_data'] = df
# Read DataFrame from the HDF5 file
with pd.HDFStore('data.h5', mode='r') as store:
    df_retrieved = store['my_data']

print(df_retrieved)
```

```
     Name  Age
0     Ali   25
1  Burhan   30
2   saeed   22
```

In [81]:
```python
import pandas as pd

# Sample data
data = {'Name': ['Alice', 'Bob', 'Charlie'],
        'Age': [25, 30, 22]}
df = pd.DataFrame(data)

# Write DataFrame to an Excel file
excel_writer = pd.ExcelWriter('data.xlsx', engine='xlsxwriter')
df.to_excel(excel_writer, sheet_name='Sheet1', index=False)
excel_writer.save()

# Read DataFrame from the Excel file
df_read = pd.read_excel('data.xlsx', sheet_name='Sheet1')

print(df_read)
```

```
      Name  Age
0    Alice   25
1      Bob   30
2  Charlie   22
```

In [ ]: