UNIT-4 (Files)

Introduction to Files, modes, types of files, File handling functions: open(), close(), read(), readline(), readlines(); write(), writeline(), append(); seek(), tell(), flush(); file copy using shutil (), delete a file (os.remove ());
Pandas-DataFrame creation with dictionaries, list of dictionaries, dictionary of series, renaming columns and rows labels; Importing data from CSV to DataFrame (Pandas), Inspecting data in DataFrame (head (), tail (), info()), Statistical summary (describe ()); Slicing and Sorting in Pandas; Modifying DataFrames, Data Cleaning in Pandas;

4.0 Introduction to files:

Files are a way to store data permanently on a storage device. Unlike variables that hold data temporarily during the execution of a program, files ensure that data persists even after the program has ended.

Python provides built-in support for file handling using functions that allow you to read from and write to files, as well as perform other file operations.

4.0.1 Need of file handling:

All programs need the input to process and output to display data. And everything needs a file as name storage compartments on computers that are managed by OS. Though variables provide us a way to store data while the program runs, if we want out data to persist even after the termination of the program, we have to save it to a file.

4.0.2 Definition of a file: A file is a named location on disk to store related information. Files are used to store data permanently, unlike variables, which are temporary. Several types of files are:

- **Text Files:** Human-readable files. They contain plain text. For example, .txt, .csv, etc.
- **Binary Files:** Not human-readable. They contain data in binary form, such as images, executable files, etc.

4.1 File Modes

- r: Read (default mode). Opens the file for reading.
- w: Write. Opens the file for writing (creates a new file or truncates existing one).
- a: Append. Opens the file for appending new data.
- x: Exclusive creation. Fails if the file already exists.
- b: Binary mode.
- t: Text mode (default).
- +: Open a file for updating (both reading and writing).

4.2 File Handling Functions – Opening a file:

1. File Opening: The open() function is used to open the file. It returns a file object, which has a read() method for reading the content of the file.

Syntax:

```
f = open(filename, mode)
```

The open() function takes two elementary parameters for file handling:

1. The file_name includes the file extension and assumes the file is in the current working directory. If the file location is elsewhere, provide the absolute or relative path.
2. The mode is an optional parameter that defines the file opening method. The table below outlines the different possible options:

Example: First get to know the current working directory or change it to the preferred location.

```
import os
print(os.getcwd())
os.chdir('D:/Work/Python')
print(os.getcwd())
```

```
D:\Work\Python
D:\Work\Python
```

```
f = open("demofile.txt", "r")
print(f.read())
```

```
Hello! Welcome to demofile.txt
This file is for testing purposes.
Good Luck!
```

Note: Make sure the file exists, or else you will get an error.

2. Reading from a file:

The read() method to read the entire content of a file or the readline() method to read one line at a time.

Example:

```
#Read only some parts of the file
f=open("demofile.txt", "r")
print(f.read(5))
```

```
Hello
```

```
#Read one line at a time
f=open("demofile.txt", "r")
print(f.readline())
print(f.readline())
```

```
Hello! Welcome to demofile.txt

This file is for testing purposes.
```

By looping through the lines of the file, you can read the whole file, line by line:

Example:

Loop through the file line by line

```
f = open("demofile.txt", "r")
for x in f:
  print(x)
```

```
Hello! Welcome to demofile.txt

This file is for testing purposes.

Good Luck!
```

3. Writing text to a file:

To write to an existing file, you must add a parameter to the open() function:

- "a" - Append - will append to the end of the file
- "w" - Write - will overwrite any existing content

Example: Using write() method

```
f = open("demofile2.txt", "x")
f = open("demofile2.txt", "a")
f.write("Now the file has more content!")
f.close()

#open and read the file after the appending:
f = open("demofile2.txt", "r")
print(f.read())
```

```
Now the file has more content!
```

Example: Using writelines() method

```
f = open("demofile2.txt", "w")
f.write("Now the file has more content!")
f.close()

#Writes a list of strings to the file.
f = open("demofile2.txt", "w")
lines = ['Hello, World!\n', 'Welcome to file handling in Python.\n']
f.writelines(lines)
f = open("demofile2.txt", "r")
print(f.read())
```

```
Hello, World!
Welcome to file handling in Python.
```

4. Writing number to a file:

The write()function expects to only write strings. If we want to write numbers to a file, we will need to "cast" them as strings using the function str().

```
# Open a file in write mode
file = open('number.txt', 'w')

# The number to write to the file
numbers = range(0, 15)

# Convert the number to a string and write it to the file
for number in numbers:
    file.write(str(number))

# Close the file
file.close()

# Open the file in read mode to verify the content
file = open('number.txt', 'r')
content = file.read()
print(f'The number in the file is: {content}')

# Close the file
file.close()
```

```
The number in the file is: 012345678910111213 14
```

5. Closing a file: The close() method closes an open file. You should always close your files, in some cases, due to buffering; changes made to a file may not show until you close the file.

Example: Close a file after it has been opened.

6. Appending data:

While reading or writing to a file, access mode governs the type of operations possible in the opened file. It refers to how the file will be used once it's opened. These modes also define the location of the File Handle in the file.

The definition of these access modes is as follows:

1. Append Only ('a'): Open the file for writing.
2. Append and Read ('a+'): Open the file for reading and writing.

When the file is opened in append mode in Python, the handle is positioned at the end of the file. The data being written will be inserted at the end, after the existing data.

Example:

```
file1 = open("myfile.txt", "w")
L = ["This is MVGR\n", "College of Engineering\n", "Autonomous\n"]
file1.writelines(L)
file1.close()

#Append adds at last
file1 = open("myfile.txt", "a")
file1.write("Today\n")
file1.close()

file1 = open("myfile.txt", "r")
print("Reading after appending")
print(file1.read())
print()
file1.close()

#Wrtie overwrites
file1 = open("myfile.txt", "w")
file1.write("Tomorrow\n")
file1.close()

file1 = open("myfile.txt", "r")
print("Readlines after writing")
print(file1.read())
print()
file1.close()
```

Output:

```
Reading after appending
This is MVGR
College of Engineering
Autonomous
Today


Readlines after writing
Tomorrow
```

7. seek(), tell() and flush() methods:

a) seek():

In Python, seek() function is used to change the position of the File Handle to a given specific position. File handle is like a cursor, which defines from where the data has to be read or written in the file.

Syntax: f.seek(offset, from_what), where f is file pointer

Parameters:

1. **Offset:** Number of positions to move forward
2. **from_what:** It defines point of reference.

The reference point is selected by the from_what argument. It accepts three values:

- 0: sets the reference point at the beginning of the file
- 1: sets the reference point at the current file position
- 2: sets the reference point at the end of the file

By default from_what argument is set to 0.

b) tell():

- A method for file objects to return the current file pointer position.

c) Flushing a file (flush()):

- Flushes the internal buffer, ensuring all data is written to the file. flush() forces the buffered content to be written to the file immediately.

Example:

Program to demonstrate the seek(), tell() and flush() functions:

```
# Open a file in write mode
file = open('demofile2.txt', 'w')

# Write some content to the file
file.write('Hello, World!\n')
file.write('This is an example of seek(), tell(), and flush().\n')

# Use flush() to ensure the content is written to the file
file.flush()

# Use tell() to get the current file pointer position
position = file.tell()
print(f'Current file pointer position: {position}')

# Use seek() to move the file pointer to the beginning of the file
file.seek(0)
print(f'File pointer position after seek: {file.tell()}')

# Close the file
#file.close()

# Open the file in read mode to verify the content
file = open('demofile2.txt', 'r')
print(file.read())
file.close()
```

```
Current file pointer position: 67
File pointer position after seek: 0
Hello, World!
This is an example of seek(), tell(), and flush().
```

8. shutil(): Copying a file using shutil.copy():

A method from the shutil module to copy the contents of the source file to the destination file.

Syntax: import shutil

shutil.copy("source", "destination"**)**

Example:

```
import shutil
shutil.copy('myfile.txt', 'destination.txt')
```

```
'destination.txt'
```

9. os.remove(path):

A method from the os module to delete a file from the filesystem.

Example:

```
import os
os.remove('myfile.txt')
```

```
---------------------------------------------------------------------------
FileNotFoundError                         Traceback (most recent call last)
Cell In[12], line 2
      1 import os
----> 2 os.remove('myfile.txt')

FileNotFoundError: [WinError 2] The system cannot find the file specified: 'myfile.txt'
```

4.3 Introduction to Pandas:

PANDAS (PANel DAta) is a high-level data manipulation tool used for analyzing data. It is very easy to import and export data using Pandas library which has a very rich set of functions. It is built on packages like NumPy and Matplotlib and gives us a single, convenient place to do most of our data analysis and visualization work. Pandas has three important data structures, namely – Series, DataFrame and Panel to make the process of analyzing data organized, effective and efficient.

The need for Pandas is when NumPy can be used for data analysis. Following are some of the differences between Pandas and Numpy:

- A Numpy array requires homogeneous data, while a Pandas DataFrame can have different data types (float, int, string, datetime, etc.).
- Pandas have a simpler interface for operations like file loading, plotting, selection, joining, GROUP BY, which come very handy in data-processing applications.
- Pandas DataFrames (with column names) make it very easy to keep track of data.
- Pandas is used when data is in Tabular Format, whereas Numpy is used for numeric array-based data manipulation.

4.3.1 DataFrame creation with dictionaries:

Recall that Python dictionary has key: value pairs and a value can be quickly retrieved when its key is known. Dictionary keys can be used to construct an index for a Series.

Example:

```
import pandas as pd

data = {'Name': ['Ram', 'Raj', 'Raghavendra'],
        'Age': [28, 24, 35],
       'city': ['Vizag','Vizianagaram','Vijayawada']}
df = pd.DataFrame(data)
print(df)
```

```
          Name  Age          city
0          Ram   28         Vizag
1          Raj   24  Vizianagaram
2  Raghavendra   35    Vijayawada
```

4.3.2 DataFrame creation from a list of dictionaries:

```
data = [{'Name': 'Ram', 'Age': 28, 'city': 'Vizag'},
        {'Name': 'Raj', 'Age': 24, 'city': 'Vizianagaram'},
        {'Name': 'Raghavendra', 'Age': 35, 'city': 'Vijayawada'}]
df = pd.DataFrame(data)
print(df)
```

```
          Name  Age          city
0          Ram   28         Vizag
1          Raj   24  Vizianagaram
2  Raghavendra   35    Vijayawada
```

4.3.3 DataFrame creation from a dictionary of series:

```
data = {'Name': pd.Series(['Ram', 'Raj', 'Raghavendra']),
        'Age': pd.Series([28, 24, 35]),
       'city': pd.Series(['Vizag', 'Viziangaram', 'Vijayawada'])}
df = pd.DataFrame(data)
print(df)
```

```
          Name  Age         city
0          Ram   28        Vizag
1          Raj   24  Viziangaram
2  Raghavendra   35   Vijayawada
```

4.3.4 Renaming Columns and Row labels of a dataframe:

```
import pandas as pd

data = {'Name': ['Ram', 'Raj', 'Raghavendra'],
        'Age': [28, 24, 35],
       'city': ['Vizag','Viziangaram','Vijayawada']}
df = pd.DataFrame(data)

# Renaming columns
df.rename(columns={'Name': 'Full Name', 'Age': 'Years', 'city': 'Location'}, inplace=True)
print(df)
```

```
     Full Name  Years     Location
0          Ram     28        Vizag
1          Raj     24  Viziangaram
2  Raghavendra     35   Vijayawada
```

In the above example, the inplace=True parameter is used in various methods to apply changes directly to the DataFrame without returning a new object. When we use this parameter, the original DataFrame is modified, and the method returns None.

Renaming row labels

```
import pandas as pd

data = {'Name': ['Ram', 'Raj', 'Raghavendra'],
        'Age': [28, 24, 35],
       'city': ['Vizag','Viziangaram','Vijayawada']}
df = pd.DataFrame(data)

# Renaming row labels
df.rename(index={0: 'row1', 1: 'row2', 2: 'row3'}, inplace=True)
print(df)
```

```
             Name  Age         city
row1          Ram   28        Vizag
row2          Raj   24  Viziangaram
row3  Raghavendra   35   Vijayawada
```

4.3.5 Importing data from CSV to DataFrame (Pandas):

We can create a DataFrame by importing data from CSV files where values are separated by commas. Similarly, we can also store or export data in a DataFrame as a .csv file.

Let us assume that we have the following data in a csv file named student_performance.csv stored in the folder C:/Desktop.

We can load the data from the student_performance.csv file into a DataFrame, say marks using Pandas read_csv() function as shown below:

```
import pandas as pd

# Replace 'your_file.csv' with the path to your CSV file
df = pd.read_csv('student_performance.csv')#
df
```

	Student_ID	Gender	Study_Hours_per_Week	Attendance_Rate	Past_Exam_Scores
0	S147	Male	31	68.267841	86
1	S136	Male	16	78.222927	73
2	S209	Female	21	87.525096	74
3	S458	Female	27	92.076483	99
4	S078	Female	37	98.655517	63

4.3.5.1 Exporting a DataFrame to a CSV File

To export a DataFrame to a CSV file, you can use the df.to_csv() function.

Example:

```
import pandas as pd

# Creating a sample DataFrame
data = {'A': [1, 2, 3, 4], 'B': [5, 6, 7, 8], 'C': [9, 10, 11, 12]}
df = pd.DataFrame(data)

# Exporting the DataFrame to a CSV file
df.to_csv('exported_file.csv', index=False)
```

We can exclusively specify column names using the parameter “names” while creating the DataFrame using the read_csv() function.

Example:

In the following statement, names parameter is used to specify the labels for columns of the DataFrame marks1:

```
import pandas as pd
df = pd.read_csv("student_performance.csv", names= ['SID', 'Category', 'StudyHrs_PerWeek', 'Attendance_Percentage', 'AB', 'CD', 'EF', 'GH','IJ', 'KL'])
df
```

	SID	Category	StudyHrs_PerWeek	Attendance_Percentage	AB	CD	EF	GH	IJ	KL
0	S147	Male	31	68.267841	86	High School	Yes	Yes	63	Pass
1	S136	Male	16	78.222927	73	PhD	No	No	50	Fail
2	S209	Female	21	87.525096	74	PhD	Yes	No	55	Fail
3	S458	Female	27	92.076483	99	Bachelors	No	No	65	Pass
4	S078	Female	37	98.655517	63	Masters	No	Yes	70	Pass
...	...	...	...	...	...	...	...	...	...	...
703	S492	Male	14	84.658761	78	PhD	Yes	No	50	Fail
704	S301	Male	35	60.278990	83	Masters	No	No	62	Pass
705	S473	Male	25	98.384969	75	Bachelors	Yes	No	57	Fail
706	S307	Female	21	96.148012	84	Bachelors	Yes	No	65	Pass
707	S046	Female	22	80.404392	93	Bachelors	Yes	No	55	Fail

708 rows × 10 columns

4.3.6 Inspecting Data in a DataFrame:

Inspecting data in a panda DataFrame is crucial to understanding its structure, contents, and potential issues. Here are some common methods and examples to help you inspect a DataFrame:

1. **Viewing the first few rows:**
   - **Example:**

```
import pandas as pd
df = pd.read_csv("student_performance.csv", header=0)

# View the first 5 rows
df.head()
```

	Student_ID	Gender	Study_Hours_per_Week	Attendance_Rate	Past_Exam_Scores
0	S147	Male	31	68.267841	86
1	S136	Male	16	78.222927	73
2	S209	Female	21	87.525096	74
3	S458	Female	27	92.076483	99
4	S078	Female	37	98.655517	63

2. **Viewing the last few rows:**
   - **Example:**

```
import pandas as pd
df = pd.read_csv("student_performance.csv", header=0)

# View the last 5 rows
df.tail()
```

	Student_ID	Gender	Study_Hours_per_Week	Attendance_Rate
703	S492	Male	14	84.658761
704	S301	Male	35	60.278990
705	S473	Male	25	98.384969
706	S307	Female	21	96.148012
707	S046	Female	22	80.404392

3. **Getting dataframe information:**
   - **Example:**

```
import pandas as pd
df = pd.read_csv("student_performance.csv", header=0)

# Get a summary of the dataframe
df.info()

# Get a summary statistics (concise) of the dataframe
df.describe()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 708 entries, 0 to 707
Data columns (total 10 columns):
 #   Column                      Non-Null Count  Dtype
---  ------                      --------------  -----
 0   Student_ID                  708 non-null    object
 1   Gender                      708 non-null    object
 2   Study_Hours_per_Week        708 non-null    int64
 3   Attendance_Rate             708 non-null    float64
 4   Past_Exam_Scores            708 non-null    int64
 5   Parental_Education_Level    708 non-null    object
 6   Internet_Access_at_Home     708 non-null    object
 7   Extracurricular_Activities  708 non-null    object
 8   Final_Exam_Score            708 non-null    int64
 9   Pass_Fail                   708 non-null    object
dtypes: float64(1), int64(3), object(6)
memory usage: 55.4+ KB
```

	Study_Hours_per_Week	Attendance_Rate	Past_Exam_Scores	Final_Exam_Score
count	708.000000	708.000000	708.000000	708.000000
mean	26.132768	78.107722	77.871469	58.771186
std	8.877727	13.802802	14.402739	6.705877
min	10.000000	50.116970	50.000000	50.000000
25%	19.000000	67.550094	65.000000	52.000000
50%	27.000000	79.363046	79.000000	59.500000
75%	34.000000	89.504232	91.000000	64.000000
max	39.000000	99.967675	100.000000	77.000000

4.3.7 Slicing and Sorting in a DataFrame:

1. Slicing DataFrames:

Slicing the dataframe allows you to select specific rows and columns from a DataFrame.

a) Selecting Specific Columns:

Example:

```
import os
os.chdir("C:/Users/K LEELA PRASAD/OneDrive/Desktop")

df=pd.read_csv("student_performance.csv", header=0)
df
# Slicing to select specific columns using column names directly
selected_columns = df[['Student_ID', 'Gender', 'Attendance_Rate']]
selected_columns
```

	Student_ID	Gender	Attendance_Rate
0	S147	Male	68.267841
1	S136	Male	78.222927
2	S209	Female	87.525096
3	S458	Female	92.076483
4	S078	Female	98.655517
...	...	...	...
703	S492	Male	84.658761
704	S301	Male	60.278990
705	S473	Male	98.384969
706	S307	Female	96.148012
707	S046	Female	80.404392

708 rows × 3 columns

b) Using loc and iloc:

In pandas, loc and iloc are used to access subsets of a DataFrame, but they operate in different ways:

i) loc:

- The loc function is label-based, meaning it selects data by the labels of rows and columns. The end-point is inclusive. For example, df.loc[0:2] includes rows 0, 1, and 2.

ii) iloc:

- The iloc function is integer position-based, meaning it selects data by the integer positions of rows and columns. The end-point is exclusive. For example, df.iloc[0:2] includes rows 0 and 1, but not 2.

Example:

```
import pandas as pd
df = pd.read_csv("student_performance.csv", header=0)

# Using loc (label-based slicing)
print(df.loc[0:5, ['Student_ID', 'Gender']])

# Using iloc (integer-based slicing)
print(df.iloc[0:5, 0:2])
```

```
  Student_ID  Gender
0       S147    Male
1       S136    Male
2       S209  Female
3       S458  Female
4       S078  Female
5       S417    Male
  Student_ID  Gender
0       S147    Male
1       S136    Male
2       S209  Female
3       S458  Female
4       S078  Female
```

2. Sorting DataFrames:

Sorting allows you to order the data based on values in one or more columns.

a) Sorting by a Single Column:

```
import pandas as pd
df = pd.read_csv("student_performance.csv", header=0)

# Sort by column1 in ascending order
df.sort_values(by='Student_ID')
```

	Student_ID	Gender	Study_Hours_per_Week	Attendance_Rate
366	S001	Male	38	68.836948
573	S002	Female	30	87.478915
168	S003	Male	32	69.649472
693	S004	Male	10	91.458211
668	S005	Male	12	78.454073
...	...	...	...	...
245	S496	Male	31	79.413188
633	S497	Male	10	99.967675
271	S498	Female	27	77.790329
74	S499	Male	24	73.660506
659	S500	Female	26	65.642743

```
import pandas as pd
df = pd.read_csv("student_performance.csv", header=0)

# Sort by column1 in descending order
df.sort_values(by='Student_ID', ascending=False)
```

	Student_ID	Gender	Study_Hours_per_Week	Attendance_Rate
659	S500	Female	26	65.642743
74	S499	Male	24	73.660506
271	S498	Female	27	77.790329
633	S497	Male	10	99.967675
245	S496	Male	31	79.413188

b) Sorting by Multiple Columns:

```
import pandas as pd
df = pd.read_csv("student_performance.csv", header=0)

# Sort by column1 in ascending order, then by column2 in descending order
df.sort_values(by=['Student_ID', 'Gender'], ascending=[True, False])
```

	Student_ID	Gender	Study_Hours_per_Week	Attendance_Rate	Past_Exam_Scores
366	S001	Male	38	68.836948	64
573	S002	Female	30	87.478915	56
168	S003	Male	32	69.649472	58
693	S004	Male	10	91.458211	100
668	S005	Male	12	78.454073	97
...	...	...	...	...	...
245	S496	Male	31	79.413188	67
633	S497	Male	10	99.967675	55
271	S498	Female	27	77.790329	67
74	S499	Male	24	73.660506	68
659	S500	Female	26	65.642743	71

708 rows × 10 columns

4.3.8 Modifying DataFrames:

Modifying a DataFrame allows you to add, update, or remove data.

1. Adding a New Column:

Example:

```
import pandas as pd
df = pd.read_csv("student_performance.csv", header=0)

# Add a new column with default values
df['new_column'] = 0

# Add a new column with calculated values
df['new_column'] = df['Student_ID'] + df['Gender']
df
```

Output:

Final_Exam_Score	Pass_Fail	new_column
63	Pass	S147Male
50	Fail	S136Male
55	Fail	S209Female
65	Pass	S458Female
70	Pass	S078Female
...	...	...
50	Fail	S492Male
62	Pass	S301Male
57	Fail	S473Male
65	Pass	S307Female
55	Fail	S046Female

2. Removing a column:

Example:

```
import pandas as pd
df = pd.read_csv("student_performance.csv", header=0)

# Remove a single column
#df.drop(columns=['Student_ID'], inplace=True)

# Remove multiple columns
df.drop(columns=['Student_ID', 'Gender'], inplace=True)
df
```

	Study_Hours_per_Week	Attendance_Rate	Past_Exam_Scores
0	31	68.267841	86
1	16	78.222927	73
2	21	87.525096	74
3	27	92.076483	99
4	37	98.655517	63
...	...	...	...
703	14	84.658761	78
704	35	60.278990	83
705	25	98.384969	75

3. Removing rows by Labels

a) Removing single by label

You can remove rows by their labels using the drop() method and specifying the row labels.

```
import pandas as pd
df = pd.read_csv("student_performance.csv")
df_dropped = df.drop(0)
df_dropped
```

	Student_ID	Gender	Study_Hours_per_Week	Attendance_Rate	Past_Exam_Scores
1	S136	Male	16	78.222927	73
2	S209	Female	21	87.525096	74
3	S458	Female	27	92.076483	99
4	S078	Female	37	98.655517	63
5	S417	Male	30	84.159193	77
...	...	...	...	...	...
703	S492	Male	14	84.658761	78
704	S301	Male	35	60.278990	83
705	S473	Male	25	98.384969	75
706	S307	Female	21	96.148012	84
707	S046	Female	22	80.404392	93

707 rows × 10 columns

In the above dataframe removing a single row by label using drop method.

b) Removing multiple rows by labels

```
import pandas as pd
df = pd.read_csv("student_performance.csv")

# removing multiple rows by label
df_dropped = df.drop([2,3,4])
df_dropped
```

	Student_ID	Gender	Study_Hours_per_Week	Attendance_Rate	Past_Exam_Scores
0	S147	Male	31	68.267841	86
1	S136	Male	16	78.222927	73
5	S417	Male	30	84.159193	77
6	S302	Male	24	89.389494	95
7	S009	Male	31	50.683598	78
...	...	...	...	...	...
703	S492	Male	14	84.658761	78
704	S301	Male	35	60.278990	83
705	S473	Male	25	98.384969	75
706	S307	Female	21	96.148012	84
707	S046	Female	22	80.404392	93

'05 rows × 10 columns

In the above dataframe removing multiple rows by label using drop method.

4. Removing Rows by Conditions:

You can remove rows based on conditions using boolean indexing.

Example:

```
import os
os.chdir("C:/Users/K LEELA PRASAD/OneDrive/Desktop")
import pandas as pd
df = pd.read_csv("student_performance.csv")

# Removing rows where column 'Study_Hours_per_Week' is less than 25
df_filtered = df[df['Study_Hours_per_Week'] <= 25]
df_filtered
```

	Student_ID	Gender	Study_Hours_per_Week	Attendance_Rate	Past_Exam_Scores
1	S136	Male	16	78.222927	73
2	S209	Female	21	87.525096	74
6	S302	Male	24	89.389494	95
11	S059	Male	11	92.104571	95
12	S315	Male	25	78.881598	81
...	...	...	...	...	...
702	S358	Female	22	93.349688	76
703	S492	Male	14	84.658761	78
705	S473	Male	25	98.384969	75
706	S307	Female	21	96.148012	84
707	S046	Female	22	80.404392	93

319 rows × 10 columns

5. Removing Rows by Slicing:

You can also remove rows by slicing the DataFrame.

Example:

```
import os
os.chdir("C:/Users/K LEELA PRASAD/OneDrive/Desktop")

# Removing the first two rows using slicing
df_sliced = df.iloc[2:]
df_sliced
```

	Student_ID	Gender	Study_Hours_per_Week	Attendance_Rate	Past_Exam_Scores
2	S209	Female	21	87.525096	74
3	S458	Female	27	92.076483	99
4	S078	Female	37	98.655517	63
5	S417	Male	30	84.159193	77
6	S302	Male	24	89.389494	95
...	...	...	...	...	...
703	S492	Male	14	84.658761	78
704	S301	Male	35	60.278990	83
705	S473	Male	25	98.384969	75
706	S307	Female	21	96.148012	84
707	S046	Female	22	80.404392	93

706 rows × 10 columns

In the above dataframe, the statement df_sliced = df.iloc[2:] creates a new DataFrame df_sliced that includes all the rows from the third row (index 2) onwards of the original DataFrame df.

4.3.9 Data Cleaning:

Data cleaning involves handling missing values, duplicates, and incorrect data.

1. Handling Missing Values:

Empty cells (missing values) can be handled using various methods such as filling them with a specific value or removing rows/columns with empty cells.

Example1: (With reference to the data frame "student_performance.csv")

```
import pandas as pd
df = pd.read_csv("student_performance.csv", header=0)

# Drop rows with missing values
df.dropna(inplace=True)

# Fill missing values with a specific value
df.fillna(0, inplace=True)

# Fill missing values with the mean of the column
df['column1'].fillna(df['column1'].mean(), inplace=True)
```

Example2:

```
import pandas as pd

# Creating a sample DataFrame with missing values
data = {'A': [1, 2, None, 4], 'B': [None, 6, 7, 8], 'C': [9, 10, 11, None]}
df = pd.DataFrame(data)

# Filling missing values with a specific value (e.g., 0)
df_filled = df.fillna(0)
df_filled
```

	A	B	C
0	1.0	0.0	9.0
1	2.0	6.0	10.0
2	0.0	7.0	11.0
3	4.0	8.0	0.0

Example: Removing Rows/Columns with Empty Cells:

```
import pandas as pd

# Creating a sample DataFrame with missing values
data = {'A': [1, 2, None, 4], 'B': [None, 6, 7, 8], 'C': [9, 10, 11, None]}
df = pd.DataFrame(data)

# Removing rows with any missing values
df_dropped_rows = df.dropna()
print(df_dropped_rows)

# Removing columns with any missing values
df_dropped_cols = df.dropna(axis=1)
print(df_dropped_cols)
```

```
     A    B     C
1  2.0  6.0  10.0
Empty DataFrame
Columns: []
Index: [0, 1, 2, 3]
```

2. Handling wrong formats:

Sometimes data might be in the wrong format, such as a numerical value stored as a string. We can convert data types using the astype() method.

```
# Creating a sample DataFrame with wrong format
data = {'A': ['1', '2', '3', '4'], 'B': ['5', '6', 'seven', '8']}
df = pd.DataFrame(data)

# Converting column 'A' to integers
df['A'] = df['A'].astype(int)
df

# Converting column 'B' to integers (handling errors)
df['B'] = pd.to_numeric(df['B'], errors='coerce')
df
```

	A	B
0	1	5.0
1	2	6.0
2	3	NaN
3	4	8.0

3. Handling Wrong Data:

Wrong data can be corrected or removed based on certain conditions.

Example:

```
# Creating a sample DataFrame with wrong data
data = {'A': [1, -2, 3, -4], 'B': [5, 6, -7, 8]}
df = pd.DataFrame(data)

# Replacing wrong data with NaN (e.g., negative values)
df_corrected = df.where(df > 0, None)
print(df_corrected)

# Removing rows with wrong data (e.g., negative values)
df_filtered = df[(df['A'] > 0) & (df['B'] > 0)]
print(df_filtered)
```

```
     A    B
0  1.0  5.0
1  NaN  6.0
2  3.0  NaN
3  NaN  8.0
   A  B
0  1  5
```

4. Handling Duplicates:

Duplicates can be removed using the drop_duplicates() method.

Example:

```
# Creating a sample DataFrame with duplicates
data = {'A': [1, 2, 2, 4], 'B': [5, 6, 6, 8]}
df = pd.DataFrame(data)

# Removing duplicate rows
df_unique = df.drop_duplicates()
print(df_unique)
```

```
   A  B
0  1  5
1  2  6
3  4  8
```

5. Replacing Incorrect Data:

```
import pandas as pd
df = pd.read_csv("student_performance.csv", header=0)

# Replace specific values in a column
df['Student_ID'].replace('S147', 'S14700', inplace=True)
df
```

Output:

	Student_ID	Gender	Study_Hours_per_Week	Attendance_Rate
0	S14700	Male	31	68.267841
1	S136	Male	16	78.222927
2	S209	Female	21	87.525096

```
import pandas as pd
df = pd.read_csv("student_performance.csv", header=0)

# Replace specific values in a column
df['Student_ID'].replace('S147', 'S14700', inplace=True)
df
```

```
C:\Users\K LEELA PRASAD\AppData\Local\Temp\ipykernel_4672\2866
s through chained assignment using an inplace method.
The behavior will change in pandas 3.0. This inplace method wi
es as a copy.

For example, when doing 'df[col].method(value, inplace=True)',
tead, to perform the operation inplace on the original object.


  df['Student_ID'].replace('S147', 'S14700', inplace=True)
```

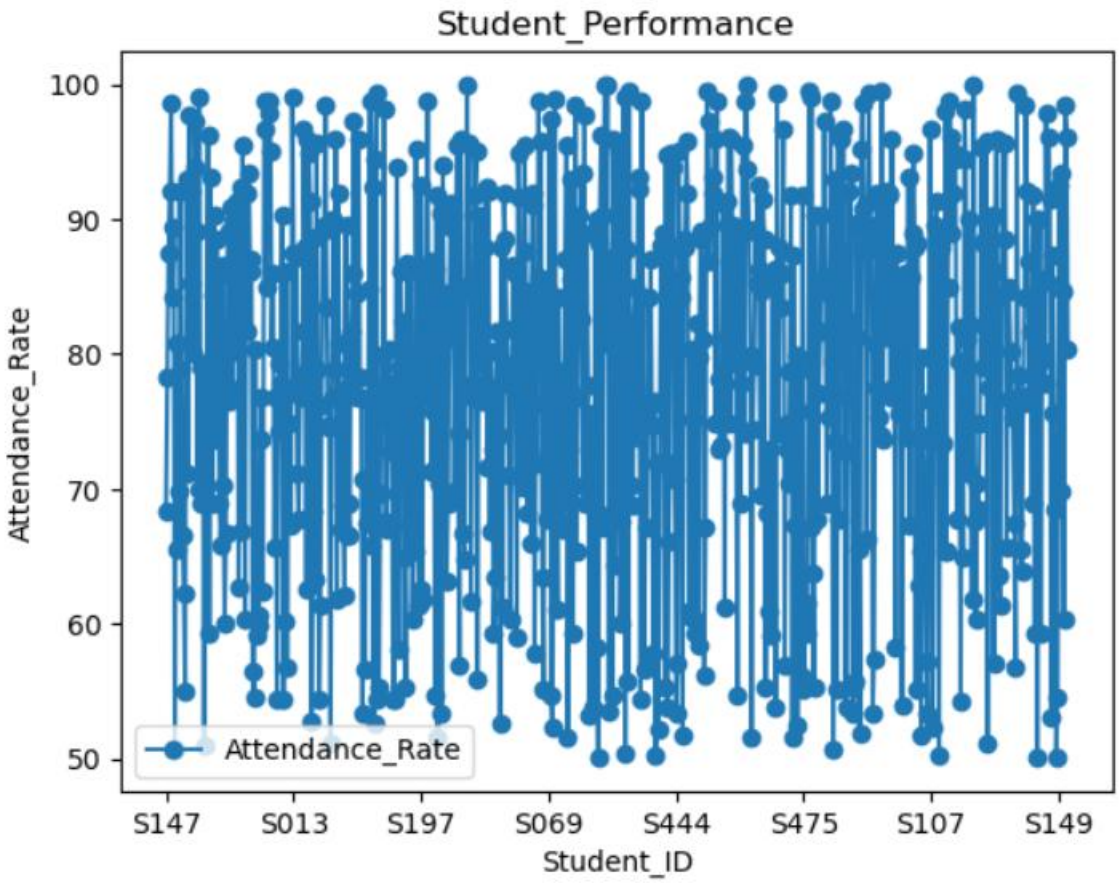
	Student_ID	Gender	Study_Hours_per_Week	Attendance_Rate
0	S14700	Male	31	68.267841
1	S136	Male	16	78.222927
2	S209	Female	21	87.525096

6. Visualizing the DataFrame:

```
import os
os.chdir("C:/Users/K LEELA PRASAD/OneDrive/Desktop")
import pandas as pd
import matplotlib.pyplot as plt
df=pd.read_csv("student_performance.csv", header=0)
df
# Step 1: Plot the Data
df.plot(x='Student_ID', y='Attendance_Rate', kind='line', marker='o')

# Step 2: Add Titles and Labels
plt.title('Student_Performance')
plt.xlabel('Student_ID')
plt.ylabel('Attendance_Rate')

# Step 3: Show the Graph
plt.show()
```



Explanation:

1. Import Necessary Libraries: pandas for data manipulation and matplotlib.pyplot for plotting.
2. Create a DataFrame: We create a DataFrame with sample data.
3. Plot the Data: Use the plot() method of the DataFrame to create a line graph. Specify the columns for the x and y axes, the kind of graph, and any markers.
4. Add Titles and Labels: Use matplotlib functions to add titles and labels to the graph.
5. Show the Graph: Finally, use plt.show() to display the graph.

Types of Graphs

You can create various types of graphs by changing the kind parameter in the plot() method:

1. Line Graph: kind='line'
2. Bar Graph: kind='bar'
3. Horizontal Bar Graph: kind='barh'
4. Histogram: kind='hist'
5. Box Plot: kind='box'
6. Area Plot: kind='area'
7. Scatter Plot: kind='scatter' (requires x and y parameters)