Unit-4: Python Interaction with text and CSV:

4.1 File handling (text and CSV files) using CSV module :

- A CSV file (Comma Separated Values file) is a type of plain text file that uses specific structuring to arrange tabular data. Because it's a plain text file, it can contain only actual text data—in other words, printable ASCII or Unicode characters.
- A CSV file (Comma Separated Values file) is a delimited text file that uses a comma**(,)** to separate values. It is used to store tabular data, such as a spreadsheet or database.
- Python's Built-in csv library makes it easy to read, write, and process data from and to CSV files.

Where Do CSV Files Come From?

CSV files are normally created by programs that handle large amounts of data. They are a convenient way to export data from spreadsheets and databases as well as import or use it in other programs. For example, you might export the results of a data mining program to a CSV file and then import that into a spreadsheet to analyze the data, generate graphs for a presentation, or prepare a report for publication.

CSV files are very easy to work with programmatically. Any language that supports text file input and string manipulation (like Python) can work with CSV files directly.

4.1.1 CSV module , File modes: Read , write, append

Open a CSV File

When you want to work with a CSV file, the first thing to do is to open it. You can open a file using open() built-in function specifying its name (same as a text file).

```
f = open('myfile.csv')
```

When you specify the filename only, it is assumed that the file is located in the same folder as Python. If it is somewhere else, you can specify the exact path where the file is located.

```
f = open(r'C:\Python33\Scripts\myfile.csv')
```

Remember! While specifying the exact path, characters prefaced by \ (like \n \r \t etc.) are interpreted as special characters. You can escape them using:

1. raw strings like **r'C:\new\text.txt'**
2. double backslashes like **'C:\\new\\text.txt'**

Specify File Mode

Here are five different modes you can use to open the file:

Character	Mode	Description
'r'	Read (default)	Open a file for read only
'w'	Write	Open a file for write only (overwrite)
'a'	Append	Open a file for write only (append)
'r+'	Read + Write	open a file for both reading and writing
'x'	Create	Create a new file

Here are some examples:

```
# Open a file for reading
f = open('myfile.csv')
# Open a file for writing
f = open('myfile.csv', 'w')
# Open a file for reading and writing
f = open('myfile.csv', 'r+')
```

Because read mode 'r' and text mode 't' are default modes, you do not need to specify them.

Close a CSV File

It's a good practice to close the file once you are finished with it. You don't want an open file running around taking up resources!

Use the close() function to close an open file.

```
f = open('myfile.csv')
f.close()
# check closed status
print(f.closed)
# Prints True
```

There are two approaches to ensure that a file is closed properly, even in cases of error.

The **first approach** is to use the with keyword, which Python recommends, as it automatically takes care of closing the file once it leaves the with block (even in cases of error).

The with statement

The with statement was introduced in python 2.5. The with statement is useful in the case of manipulating the files. It is used in the scenario where a pair of statements is to be executed with a block of code in between.

Syntax:

```
with open(<file name>, <access mode>) as <file-pointer>:
        #statement suite
```

- The advantage of using with statement is that it provides the guarantee to close the file regardless of how the nested block exits.
- It is always suggestible to use the with statement in the case of files because, if the break, return, or exception occurs in the nested block of code then it automatically closes the file, we don't need to write the close() function. It doesn't let the file to corrupt.

Example:

```
with open('myfile.csv') as f:
    print(f.read())
```

The **second approach** is to use the try-finally block:

```
f = open('myfile.csv')
try:
    # File operations goes here
finally:
    f.close()
```

4.2 Important Classes and Functions of CSV modules:

4.2.1 Open(), reader(), writer(), writerows(), DictReader(),DictWriter()

Read a CSV File

Suppose you have the following CSV file.

myfile.csv

```
name,age,job,city
Bob,25,Manager,Seattle
Sam,30,Developer,New York
```

You can read its contents by importing the csv module and using its **reader() method**. The **reader()** method splits each row on a specified delimiter and returns the list of strings.

```
#read_csv.py
import csv
with open('myfile.csv') as f:
   reader = csv.reader(f)
   for row in reader:
      print(row)
# Prints:
# ['name', 'age', 'job', 'city']
# ['Bob', '25', 'Manager', 'Seattle']
# ['Sam', '30', 'Developer', 'New York']
```

Write to a CSV File

To write an existing file, you must first open the file in one of the writing modes ('w', 'a' or 'r+') first. Then, use a **writer** object and its **writerow()** method to pass the data as a list of strings.

Example1:

```
#write_csv.py
import csv
with open('myfile.csv', 'w') as f:
        writer = csv.writer(f)
        writer.writerow(['Bob', '25', 'Manager', 'Seattle'])
        writer.writerow(['Sam', '30', 'Developer', 'New York'])

print("CSV file Created and Written Successfully.")
#OUTPUT
myfile.csv
Bob,25,Manager,Seattle
Sam,30,Developer,New York
```

Example2: To write multiple rows at once in CSV with Header

```
#write_csv_head.py
import csv          # importing the csv module

fields = ['Name', 'Branch', 'Year', 'CGPA']  #field names
# data rows of csv file
rows = [['Nikhil', 'COE', '2', '9.0'],
        ['Sanchit', 'COE', '2', '9.1'],
        ['Aditya', 'IT', '2', '9.3'],
        ['Sagar', 'SE', '1', '9.5'],
        ['Prateek', 'MCE', '3', '7.8'],
        ['Sahil', 'EP', '2', '9.1']]

filename = "university_records.csv"     #name of csv file
# writing to csv file
with open(filename, 'w') as csvfile:
        csvwriter = csv.writer(csvfile)    # creating a csv writer object
        csvwriter.writerow(fields)         # writing the fields
        csvwriter.writerows(rows)          # writing the data rows

print("CSV file Created and Written Successfully.")
```

Let us try to understand the above code in pieces.

- fields and rows have been already defined. fields is a list containing all the field names. rows is a list of lists. Each row is a list containing the field values of that row.

```
with open(filename, 'w') as csvfile:
   csvwriter = csv.writer(csvfile)
```

- Here, we first open the CSV file in WRITE mode. The file object is named as csvfile. The file object is converted to csv.writer object. We save the csv.writer object as csvwriter.

```
csvwriter.writerow(fields)
```

- Now we use writerow method to write the first row which is nothing but the field names.

```
csvwriter.writerows(rows)
```

- We use writerows method to write multiple rows at once.

Append a list as a new row to the existing CSV file

There are several steps to take that.

1. Import writer class from csv module
2. Open your existing CSV file in append mode(CSV file content before append)

	A	B	C	D	E	F
1	ID	NAME	RANK	ARTICLE	COUNTRY	
2	1	Sham	1190	15	Japan	
3	2	Joe	3200	3	Brazil	
4	3	Meet	1320	11	India	
5	4	Sophia	2125	7	UAE	
6	5	Amelia	2506	9	Germany	
7						

Figure: CSV file content before append

   Create a file object for this file.

3. Pass this file object to csv.writer() and get a writer object.
4. Pass the list as an argument into the writerow() function of the writer object.
   (It will add a list as a new row into the CSV file).
5. Close the file object

```
#Append a list as a new row to the existing CSV file
from csv import writer                  # Import writer class from csv module

List=[6,'William',5532,1,'UAE']         #append new List

#Open our existing CSV file in append mode & Create a file object for this file
with open('event.csv', 'a') as f_object:
        # Pass this file object to csv.writer() and get a writer object
        writer_object = writer(f_object)

        # Pass the list as an argument into the writerow()
        writer_object.writerow(List)

print("event.csv file content has been append successfully...")
```

When you are executing this program Ensure that your CSV file must be closed otherwise this program will give you a permission error.

	A	B	C	D	E	F
1	ID	NAME	RANK	ARTICLE	COUNTRY	
2	1	Sham	1190	15	Japan	
3	2	Joe	3200	3	Brazil	
4	3	Meet	1320	11	India	
5	4	Sophia	2125	7	UAE	
6	5	Amelia	2506	9	Germany	
7	6	William	5532	1	UAE	
8						

Figure: After append content in to csv file

Read a CSV File Convert Into a Dictionary

You can read CSV file directly into a dictionary using **DictReader()** method. The first row of the CSV file is assumed to contain the **column names**, which are used as **keys** for the dictionary.

```
#Read a CSV File Convert Into a Dictionary
import csv

with open('myfile.csv') as f:
    reader = csv.DictReader(f)
    for row in reader:
        print(row)

# OUTPUT:
{'name': 'Bob', 'age': '25', 'job': 'Manager', 'city': 'Seattle'}
{'name': 'Sam', 'age': '30', 'job': 'Developer', 'city': 'New York'}
```

If the CSV file doesn't have column names like the file below, you should specify your own keys by setting the optional parameter fieldnames.

myfile1.csv

```
Bob,25,Manager,Seattle
Sam,30,Developer,New York
```

```
import csv
with open('myfile1.csv') as f:
    keys = ['Name', 'Age', 'Job', 'City']
    reader = csv.DictReader(f, fieldnames=keys)
    for row in reader:
        print(row)

#OUTPUT:
{'Name': 'Mihir', 'Age': '25', 'Job': 'Manager', 'City': 'Seattle'}
{'Name': 'Tahir', 'Age': '30', 'Job': 'Developer', 'City': 'New York'}
```

Write a CSV File From a Dictionary

- Since you can read a csv file in a dictionary, it's only fair that you should be able to write it from a dictionary as well:
- You can write a CSV file from a dictionary using the **DictWriter()** method. Here it is necessary to specify the fieldname parameter. This makes sense, because without a list of fieldnames, the **DictWriter** may not know which keys to use to retrieve values from the dictionary.

Example: Write a CSV File From a Dictionary

```
#Write a CSV File From a Dictionary
import csv
with open('info.csv', mode='w') as f:
    keys = ['name', 'age', 'job', 'city']
    writer = csv.DictWriter(f, fieldnames=keys)
    writer.writeheader()    # add column names in the CSV file
    writer.writerow({'name': 'Mihir','job': 'Manager', 'city': 'Seattle', 'age': '25'})
    writer.writerow({'name': 'Tahir','job': 'Developer', 'city': 'New York', 'age': '30'})

print("csv file content has been written successfully...")
```

OUTPUT:

```
info.csv
name,age,job,city
Mihir,25,Manager,Seattle
Tahir,30,Developer,New York
```

Use a different delimiter

The comma , is not the only delimiter used in the CSV files. You can use any delimiter of your choice such as the pipe |, tab \t, colon : or semi-colon ; etc.

Example: To specify a different delimiter, use the parameter delimiter.

```
#Write_myfile_delimiter.py
# Write a file with different delimiter '|'
import csv
with open('myfilewithdeli.csv', mode='w',newline='') as f:
    writer = csv.writer(f, delimiter='|')
    writer.writerow(['Name', 'Age', 'Designation', 'City'])
    writer.writerow(['Saumil', '25', 'Manager', 'Seattle'])
    writer.writerow(['Raj', '30', 'Developer', 'New York'])

print("csv file content with delimiter has been written successfully...")
```

OUTPUT:

```
myfilewithdeli.csv
Name|Age|Designation|City
Saumil|25|Manager|Seattle
Raj|30|Developer|New York
```

Example: Let's read this file, specifying the same delimiter.

```
# Read a file with different delimiter '|'
import csv
with open('myfilewithdeli.csv') as f:
    reader = csv.reader(f, delimiter='|')
    print("****Data From CSV File****")
    for row in reader:
        print(row)
```

OUTPUT:

```
****Data From CSV File****
['Name', 'Age', 'Designation', 'City']
['Saumil', '25', 'Manager', 'Seattle']
['Raj', '30', 'Developer', 'New York']
```

4.3 Dataframe Handling using Panda and Numpy:

4.3.1 csv and excel file extract and write using Dataframe

In the real world, a Pandas DataFrame will be created by loading the datasets from existing storage, storage can be SQL Database, CSV file, and Excel file. Pandas DataFrame can be created from the lists, dictionary, and from a list of dictionary etc.

CSV File extract using Dataframe(Reading csv files with Pandas)

- The Pandas is defined as an open-source library which is built on the top of the NumPy library. It provides fast analysis, data cleaning, and preparation of the data for the user.
- Reading the csv file into a pandas DataFrame is quick and straight forward. We don't need to write enough lines of code to open, analyze, and read the csv file in pandas and it stores the data in DataFrame.
- Here, we are taking a slightly more complicated file to read, called **hrdata.csv**, which contains data of company employees.

```
Name,Hire_Date,Salary,Leaves_Remaining
John Idle,08/15/14,50000.00,10
Smith Gilliam,04/07/15,65000.00,8
Parker Chapman,02/21/14,45000.00,10
Jones Palin,10/14/13,70000.00,3
Terry Gilliam,07/22/14,48000.00,7
Michael Palin,06/28/13,66000.00,8
```

Example : Reading the csv file into a pandas DataFrame

```
#read_csv_df.py
import pandas

df = pandas.read_csv('hrdata.csv')

print("******* Extract CSV data in DATAFRAME*********")

print(df)
```

In the above code, the three lines are enough to read the file, and only one of them is doing the actual work, i.e., pandas.read_csv()

OUTPUT:

```
========== RESTART: G:\PYTHON\UNIT4_CSV\read_csv_df.py ==========
********* Extract CSV data in DATAFRAME**********
             Name Hire Date   Salary  Leaves_Remaining
0       John Idle  08/15/14  50000.0                10
1    Smith Gilliam  04/07/15  65000.0                 8
2  Parker Chapman  02/21/14  45000.0                10
3     Jones Palin  10/14/13  70000.0                 3
4    Terry Gilliam  07/22/14  48000.0                 7
5    Michael Palin  06/28/13  66000.0                 8
>>>
```

Convert Pandas DataFrame to CSV(Write Data to CSV From DATAFRAME)

The Pandas to_csv() function is used to convert the DataFrame into CSV data. To write the CSV data into a file, we can simply pass a file object to the function. Otherwise, the CSV data is returned in a string format.

Example: To convert a DataFrame to CSV String and File.

```
#write_csv_df.py
#convert a DataFrame to CSV String and File.

import pandas as pd
dict = {'Name': ['Jemil', 'Pratham'], 'ID': [101, 102],
        'Language': ['Python', 'JavaScript']}

df = pd.DataFrame(dict)                  # to Create DataFrame from Dictionary

print('DataFrame Values:\n', df)

csv_data = df.to_csv()           # default CSV in string format

print('\n CSV Data in String Values:\n', csv_data)

with open('mydata.csv','w') as f:          #Write to CSV File
   df.to_csv(f)

print("CSV File has been Created Successfully..")
```

OUTPUT:

```
DataFrame Values:
       Name   ID    Language
0     Jemil  101      Python
1   Pratham  102  JavaScript

 CSV Data in String Values:
 ,Name,ID,Language
0,Jemil,101,Python
1,Pratham,102,JavaScript
```

Excel file Read(Extract) and Write using Dataframe

- Python provides the **Openpyxl** module, which is used to deal with Excel files without involving third-party Microsoft application software.
- By using this module, we have control over excel without open the application.
- It is used to perform excel tasks such as read data from excel file, or write data to the excel file, draw some charts, accessing excel sheet, renaming sheet, modification (adding and deleting) in excel sheet, formatting, styling in the sheet, and any other task.
- Openpyxl is very efficient to perform these tasks for you.
- Data scientists often use the Openpyxl to perform different operations such as data copying to data mining as well as data analysis.

Read Excel with Python Pandas

- Read Excel files (extensions:.xlsx, .xls) with Python Pandas. To read an excel file as a DataFrame, use the pandas **read_excel()** method.
- You can read the first sheet, specific sheets, multiple sheets or all sheets. Pandas converts this to the DataFrame structure, which is a tabular like structure.
- Create an excel file with two sheets, sheet1 and sheet2. You can use any Excel supporting program like Microsoft Excel or Google Sheets.

The contents of each are as follows:

Sheet1: Test1 Marks

	A	B	C	D	E
1		Sci	Maths	Comp	
2	Mihir	98	96	65	
3	Sameer	87	88	75	
4	Pooja	56	87	91	

Sheet2: Test2 Marks

	A	B	C	D	E
1		Sci	Maths	Comp	
2	Mihir	88	76	97	
3	Sameer	89	69	63	
4	Pooja	48	68	87	
5					

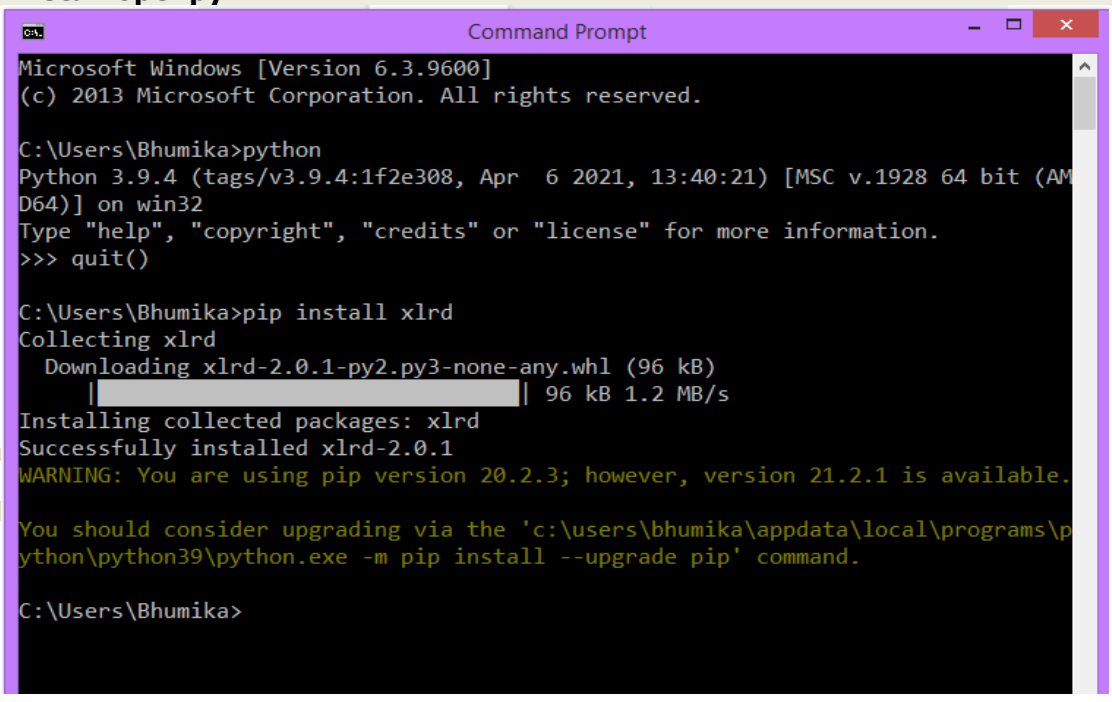
How to install xlrd?

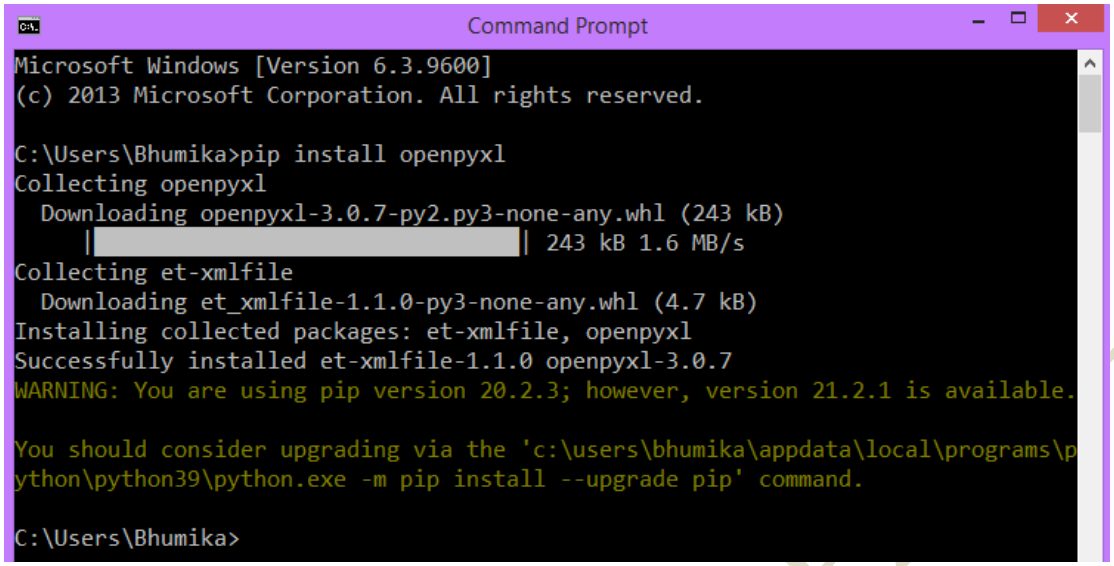
- Pandas method read_excel() uses a library called xlrd internally.
- xlrd is a library for reading (input) Excel files (.xlsx, .xls) in Python.
- If you call pandas.read_excel() method in an environment where xlrd is not installed, you will receive an error message similar to the following:

```
ImportError: Install xlrd >= 0.9.0 for Excel support
```

xlrd can be installed with pip. (pip3 depending on the environment)

```
$ pip install xlrd
$ pip install openpyxl
```





- **Read excel :** Specify the path or URL of the Excel file in the first argument. If there are multiple sheets in Excel Workbook then only the first sheet is used by pandas. It reads as DataFrame.

Example : To Read(Load/Extract) Excel File Using Dataframe

```
import pandas as pd
df = pd.read_excel('sample.xlsx')
print(df)
```

The code above outputs the excel sheet content.

In the below example:

- Select sheets to read by index: sheet_name = [0,1,2] means the first three sheets.
- Select sheets to read by name: sheet_name = ['User_info', 'compound']. This method requires you to know the sheet names in advance.
- Select all sheets: sheet_name = None.

Get Specific sheet

You can specify the sheet to read with the argument sheet_name. Specify by number starting at 0.

Example : To Read(Load/Extract) Specific Excel Sheet Using Dataframe

```
import pandas as pd

#it print data of second sheet
df_sheet_index = pd.read_excel('sample.xlsx', sheet_name=1)

# Specify by sheet name
#df_sheet_index = pd.read_excel('sample.xlsx', sheet_name='Sheet2')

print(df_sheet_index)
```

Load(Read) multiple sheets

- It is also possible to specify a list in the argumentsheet_name. It is OK even if it is a number of 0 starting or the sheet name.
- The specified number or sheet name is the key and the excel sheet data as pandas Dataframe. The DataFrame is read as the ordered dictionary with the value.

Example: Read multiple Excel sheets using Dataframe

```
#Load multiple Excel sheets using Dataframe
import pandas as pd

df_sheet_multi = pd.read_excel('sample.xlsx', sheet_name=[0, 1])
print(df_sheet_multi)

print("\nExcel Sheet1 Content:")
print(df_sheet_multi[0])

print("\nExcel Sheet2 Content:")
print(df_sheet_multi[1])

print(type(df_sheet_multi[1]))
# <class 'pandas.core.frame.DataFrame'>
```

Load all sheets

If sheet_name argument is none, all sheets are read.

```
df_sheet_all = pd.read_excel('sample.xlsx', sheet_name=None)
print(df_sheet_all)
```

In this case, the **sheet name** becomes the key.

❖ Write data in to Excel with Python Pandas

- Write Excel with Python Pandas. You can write any data (lists, strings, numbers etc) to Excel, by first converting it into a Pandas DataFrame and then writing the DataFrame to Excel.
- To export a Pandas DataFrame as an Excel file (extension: .xlsx, .xls), use the to_excel() method.

How to install installxlwt, openpyxl?

to_excel() uses a library called xlwt and openpyxl internally.

- xlwt is used to write .xls files (formats up to Excel2003)
- openpyxl is used to write .xlsx (Excel2007 or later formats).

Both can be installed with pip. (pip3 depending on the environment)

```
$ pip install xlwt
$ pip install openpyxl
```

Write DataFrame to Excel file

- Importing openpyxl is required if you want to append it to an existing Excel file described at the end.
- You can specify a path as the first argument of the to_excel() method.
- The argument new_sheet_name is the name of the sheet. If omitted, it will be named Sheet1.

Example: Write DataFrame to Excel file.

```
import pandas as pd
import openpyxl

df = pd.DataFrame([[45, 42, 48], [35, 34, 32], [46, 42, 38]],
                  index=['Mohit', 'Roy', 'Darshita'],
                  columns=['Maths', 'Science', 'Computer'])

print("*******Pandas Dataframe**********")
print(df)

#Write Datafarme to Excel file
df.to_excel('pandas_to_excel.xlsx', sheet_name='TEST1_Marks')
print("Excel File content has been written Successfully.")
```

Note that the data in the original file is deleted when overwriting.

If you do not need to write index (row name), columns (column name), the argument index, columns is False.

```
df.to_excel('pandas_to_excel_no_index_header.xlsx', index=False, header=False)
```

Write multiple DataFrames to Excel files

The ExcelWriter object allows you to use multiple **pandas.DataFrame** objects can be exported to separate sheets. Then use the ExcelWriter() function like below example.

Example:Write multiple Dateframe to Excel using ExcelWriter()

```
import pandas as pd
import openpyxl
df1 = pd.DataFrame([[45, 42, 48], [35, 34, 32], [46, 42, 38]],
                  index=['Mohit', 'Roy', 'Darshita'],
                  columns=['Maths', 'Science', 'Computer'])

df2 = pd.DataFrame([[30, 30,30], [22, 33, 12], [25, 43, 28]],
                  index=['Mohit', 'Roy', 'Darshita'],
                  columns=['Maths', 'Science', 'Computer'])

print("*******Pandas Dataframe1**********")
print(df1)
print("*******Pandas Dataframe2**********")
print(df2)
with pd.ExcelWriter('pandas_to_excel2.xlsx') as writer:
   df1.to_excel(writer, sheet_name='TEST1')
   df2.to_excel(writer, sheet_name='TEST2')

print("Excel File content has been written Successfully.")
```

OUTPUT:

```
*******Pandas Dataframe1**********
          Maths  Science  Computer
Mohit        45       42        48
Roy          35       34        32
Darshita     46       42        38
*******Pandas Dataframe2**********
          Maths  Science  Computer
Mohit        30       30        30
Roy          22       33        12
Darshita     25       43        28
Excel File content has been written Successfully.
```

4.3.2 Extracting specific attributes and rows from dataframe

A Data frame is a two-dimensional data structure, i.e., data is aligned in a tabular fashion in rows and columns. We can perform basic operations on rows/columns like selecting, deleting, adding, and renaming.

Column(attribute) Selection: In Order to select a column in Pandas DataFrame, we can either access the columns by calling them by their columns name.

Example: To select specific columns in Pandas DataFrame.

```
#sp_col_dic.py

import pandas as pd

# Define a dictionary containing employee data
data = {'Name':['Ronit','Jay', 'Raj','Ruju'],
        'Age':[27, 24, 22, 32],
        'Address':['Delhi', 'Kanpur', 'Allahabad', 'Kannauj'],
        'Qualification':['Msc', 'MA', 'MCA', 'Phd']}

# Convert the dictionary into DataFrame
df = pd.DataFrame(data)

#Select All Columns
print("\n********Employee Data*********")
print(df)

# select two columns
print("\n*****Employee Data******")
print(df[['Name', 'Qualification']])
```

Selecting a single columns

In order to select a single column, we simply put the name of the column in-between the brackets.

Example: to select a single column in Pandas DataFrame

```
# importing pandas package
import pandas as pd

# making data frame from csv file
data = pd.read_csv("hrdata.csv", index_col ="Name")

# retrieving columns by indexing operator
first = data['Hire_Date']

print(first)
```

Row Selection: Pandas provide a unique method to retrieve rows from a Data frame. **DataFrame.loc[]** method is used to retrieve rows from Pandas DataFrame. Rows can also be selected by passing integer location to an **iloc[]** function.(Read Sem2-204-Unit5)

Pandas **DataFrame.loc** attribute access a group of rows and columns by label(s) or a boolean array in the given DataFrame.

Example: To retrieve unique rows from a Data frame.

```
#sp_rows.py
import pandas as pd
```

```
# making data frame from csv file
df = pd.read_csv("hrdata.csv", index_col ="Name")

print(df)
print(type(df))

# retrieving row by loc method (series)
first=df.loc["Parker Chapman"]
second = df.loc["Terry Gilliam"]

print(first, "\n\n", second)

# A slice object with labels
first = df.loc["Parker Chapman":"Michael Palin"]
```

Indexing a DataFrame using .iloc[ ] :
This function allows us to retrieve rows and columns by position. In order to do that, we'll need to specify the positions of the rows that we want, and the positions of the columns that we want as well. The df.iloc indexer is very similar to df.loc but only uses integer locations to make its selections.
Selecting a single row:
In order to select a single row using .iloc[], we can pass a single integer to .iloc[] function.
Example: To select row using index.

```
#iloc.py
import pandas as pd

# making data frame from csv file
df = pd.read_csv("hrdata.csv", index_col ="Name")
print(df)

# retrieving rows by iloc method
row = df.iloc[2]

print('\n\n',row)
print('\nType of row',type(row))
```


4.3.3 Central Tendency measures :
4.3.3.1 mean, median, mode, variance, Standard Deviation
Read Sem2-204-Unit5

4.3.4 Dataframe functions: head, tail, loc, iloc, value,to_numpy(), describe()
1. Pandas DataFrame.head() : The head() returns the first n rows for the object based on position. If your object has the right type of data in it, it is useful for quick testing. This method is used for returning top n (by default value 5) rows of a data frame or series.
Syntax:

```
DataFrame.head(n=5)
```

Parameters
n: It refers to an integer value that returns the number of rows.

Return
It returns the DataFrame with top n rows.

2. Pandas DataFrame.tail() :The tail() function is used to get the last n rows. This function returns last n rows from the object based on position. It is useful for quickly verifying data, for example, after sorting or appending rows.

Syntax:

```
DataFrame.tail(n=5)
```

Parameters
n-number of rows to select

Returns: type of caller
The last n rows of the caller object.

Example: By using the head() in the below example, we showed only top 2 rows and last 2 rows from the dataset.

```
# importing pandas module
import pandas as pd

# making data frame
df = pd.read_csv("hrdata.csv")
print(df)

# calling head() method storing in new variable
df_top  = df.head(2)
print(df_top)

# calling head() method storing in new variable
df_bottom  = df.tail(2)
print(df_bottom)
```

3. Pandas DataFrame.loc property:
The loc property is used to access a group of rows and columns by label(s) or a boolean array.
.loc[] is primarily label based, but may also be used with a boolean array.

Allowed inputs are:

- A single label, e.g. 5 or 'a', (note that 5 is interpreted as a label of the index, and never as an integer position along the index).
- A list or array of labels, e.g. ['a', 'b', 'c'].
- A slice object with labels, e.g. 'a':'f'.
- A boolean array of the same length as the axis being sliced, e.g. [True, False, True].
- A callable function with one argument (the calling Series or DataFrame) and that returns valid output for indexing (one of the above)

Syntax:

```
DataFrame.loc
```

Raises: KeyError
when any items are not found

Example: To retrieve unique rows from a Data frame.

```
#sp_rows.py
import pandas as pd

# making data frame from csv file
df = pd.read_csv("hrdata.csv", index_col ="Name")

print(df)

# retrieving row by loc method (series)
first=df.loc["Parker Chapman"]
second = df.loc["Terry Gilliam"]

print(first, "\n\n", second)
```

4. **Pandas DataFrame.iloc :** The iloc property returns purely integer-location based indexing for selection by position.

.iloc[] is primarily integer position based (from 0 to length-1 of the axis), but may also be used with a boolean array.
Allowed inputs are:

- An integer, e.g. 5.
- A list or array of integers, e.g. [4, 3, 0].
- A slice object with ints, e.g. 1:7.
- A boolean array.
- A callable function with one argument (the calling Series or DataFrame) and that returns valid output for indexing (one of the above). This is useful in method chains, when you don't have a reference to the calling object, but would like to base your selection on some value.

iloc will raise IndexError if a requested indexer is out-of-bounds, except slice indexers which allow out-of-bounds indexing (this conforms with python/numpy slice semantics).

Example : To select row using index.

```
#iloc.py
import pandas as pd

# making data frame from csv file
df = pd.read_csv("hrdata.csv", index_col ="Name")
print(df)

# retrieving rows by iloc method
row = df.iloc[2]

print('\n\n',row)

print('\nType of row',type(row))
```

5. DataFrame - values property

The values property is used to get a Numpy representation of the DataFrame.
Only the values in the DataFrame will be returned, the axes labels will be removed.

Syntax:

```
DataFrame.values
```

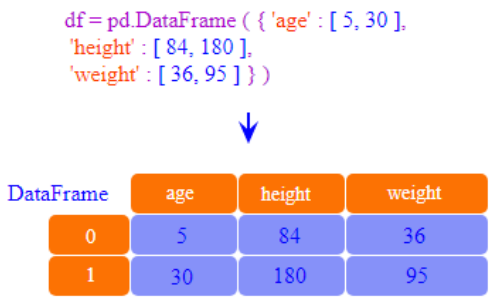
Returns: numpy.ndarray
The values of the DataFrame.

Examples: A DataFrame where all columns are the same type (e.g., int64) results in an array of the same type.

```
import numpy as np
import pandas as pd
df = pd.DataFrame({'age':    [ 5,  30],
                   'height': [84, 180],
                   'weight': [36, 95]})
print(df)

print(df.values)
```

OUTPUT:



6. pandas.DataFrame.to_numpy(): Pandas is great for working with tables, but sometimes you need to use the full force of a statistical package to get the job done. That's where turning your DataFrame into a NumPy array comes.

Syntax:

```
DataFrame.to_numpy(dtype=None, copy=False,na_value= <no_default>)
```

Parameters

- dtype – For if you need to specify the type of data that you're passing to .to_numpy(). You likely won't need to set this parameter
- copy (Default: False) – This parameter isn't widely used either. Setting copy=True will return a full exact copy of a NumPy array. Copy=False will potentially return a view of your NumPy array instead. If you don't know what the difference is, it's ok and feel free not to worry about it.
- na_value – The value to use for missing values. The default value depends on dtype and the dtypes of the DataFrame columns. The value to use when you have NAs. By default Pandas will return the NA default for that column data type. If you wanted to specify another value, go ahead and get desire result.

Example: Converting a DataFrame to Numpy Array using to_numpy().

```
#Example1:Converting a DataFrame to Numpy Array

import pandas as pd
df = pd.DataFrame([('PVR Cinema', 'Restaurant', pd.NA),
                   ('Jalaram', 'Restaurant', 224.0),
                   ('500 Club', 'Bar', 80.5),
                   ('The Square', pd.NA, 25.30)],
                   columns=('Name', 'Type', 'AvgBill'))
print('********DataFrame Values**********')
print(df)

x = df.to_numpy()
print('\nArray Values :\n',x)
print(type(x))
```

NOTE: Example2: Converting a DataFrame to Numpy Array and setting NA values.

```
x = df.to_numpy(na_value='NO_DATA')
```

7. pandas.DataFrame.describe():

The describe() function is used to generate descriptive statistics that summarize the central tendency, dispersion and shape of a dataset's distribution, excluding NaN values.

Syntax:

```
DataFrame.describe(percentiles=None, include=None, exclude=None)
```

Parameters:

- **percentile:** It is an optional parameter which is a list like data type of numbers that should fall between 0 and 1. Its default value is [.25, .5, .75], which returns the 25th, 50th, and 75th percentiles.
- **include:** It is also an optional parameter that includes the list of the data types while describing the DataFrame. Its default value is None.
- exclude: It is also an optional parameter that exclude the list of data types while describing DataFrame. Its default value is None.

Returns:

It returns the statistical summary of the Series and DataFrame.

Example :

```
import numpy as np
import pandas as pd

a1 = pd.Series([1, 2, 3])
print('\n a1 result:\n',a1.describe())

a2 = pd.Series(['p', 'q', 'q', 'r'])
print('\n a2 result:\n',a2.describe())

df = pd.DataFrame({'categorical': pd.Categorical(['s','t','u']),
                   'numeric': [1, 2, 3],
                   'object': ['p', 'q', 'r'] })
```

```
#Describing a DataFrame.
#By default only numeric fields are returned:
print('DataFrame Describe:\n',df.describe())

#Including only numeric columns in a DataFrame description.
print(df.describe(include=[np.number]))

#Including only string columns in a DataFrame description:
print(df.describe(include=[np.object]))

#Including only categorical columns from a DataFrame description:
print(df.describe(include=['category']))

#Describing all columns of a DataFrame regardless of data type:
print(df.describe(include='all'))

#Excluding numeric columns from a DataFrame description:
print(df.describe(exclude=[np.number]))
```

OUTPUT:

```
df = pd.DataFrame ( { 'categorical' : pd. Categorical ( [ 's', 't', 'u' ] ) ,
 'numeric' : [ 2, 3, 4 ],
 'object': ['p', 'q', 'r' ] } )
df.describe ( )
```

'numeric' : [2, 3, 4]

DataFrame df

	numeric	
count	3.0	3 numbers
mean	3.0	mean or average
std	1.0	Standard Deviation
min	2.0	minimum value
25%	2.5	25th percentiles
50%	3.0	50th percentiles
75%	3.5	75th percentiles
max	4.0	maximum value

N.B. : by default describe returns only numeric field