Unit-4: Python Interaction with text and CSV:

4.1 File handling (text and CSV files) using CSV module :
- 4.1.1 CSV module , File modes: Read , write, append

4.2 Important Classes and Functions of CSV modules:
- 4.2.1 Open(), reader(), writer(), writerows(), DictReader(), DictWriter()

4.3 Dataframe Handling using Panda and Numpy:
- 4.3.1 csv and excel file extract and write using Dataframe
- 4.3.2 Extracting specific attributes and rows from dataframe.
- 4.3.3 Central Tendency measures :
  - 4.3.3.1 mean, median, mode, variance, Standard Deviation
- 4.3.4 Dataframe functions: head, tail, loc, iloc, value, to_numpy(), describe()

Python Interaction with text and CSV

Python too supports file handling and allows users to handle files i.e., to read, write, create, delete and move files, along with many other file handling options, to operate on files.

The concept of file handling has stretched over various other languages, but the implementation is either complicated or lengthy, but alike other concepts of Python, this concept here is also easy and short.

For writing to a file in Python, you would need a couple of functions such as ***Open()***, ***Write()***, and ***Read()***.

All these are built-in Python functions and don't require a module to import.

There are majorly two types of files you may have to interact with while programming.

One is the text file that contains streams of ASCII or UNICODE (UTF-8) characters. Each line ends with a newline ("\n") char, a.k.a. EOL (End of Line).

Another type of file is called binary that contains machine-readable data. It doesn't have, so-called line as there is no line-ending. Only the application using it would know about its content.

Binary files in Python

Most of the files that we see in our computer system are called binary files.
Example:
1. Document files: .pdf, .doc, .xls etc.
2. Image files: .png, .jpg, .gif, .bmp etc.
3. Video files: .mp4, .3gp, .mkv, .avi etc.
4. Audio files: .mp3, .wav, .mka, .aac etc.
5. Database files: .mdb, .accde, .frm, .sqlite etc.
6. Archive files: .zip, .rar, .iso, .7z etc.
7. Executable files: .exe, .dll, .class etc.

Text files in Python

Text files don't have any specific encoding and it can be opened in normal text editor itself.
Example:
Web standards: html, XML, CSS, JSON etc.
Source code: c, app, js, py, java etc.
Documents: txt, tex, RTF etc.
Tabular data: csv, tsv etc.
Configuration: ini, cfg, reg etc.

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

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

Python File Handling Operations

Most importantly there are 4 types of operations that can be handled by Python on files:

- Open
- Read
- Write
- Close

Other operations include:

- Rename
- Delete

Creating a File

The first step in using a file instance is to open a disk file.
In any computer language this means establishing a communication link between your code and the external file.

To create a new file I/O classes provides the member function **open().**

Syntax: **open(filename, mode)**
Here the mode refers to the **Access Mode**.

Access modes govern 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.
File handle is like a cursor, which defines from where the data has to be read or written in the file.

example : f =open("C:/Documents/Python/test.txt", "r+") or open('a.csv','r')

```
# open function for opening the file (file_open1.py)
file1 = open('a.txt','r')
for i in file1:                        # This will print every line one by one in the file
  print(i)
```

OR

```
# Python code to illustrate read() mode
file = open("a.txt", "r")
print (file.read())
```

There are 6 access modes in python.

- **Read Only ('r'):**
  Open text file for reading. The handle is positioned at the beginning of the file.
  If the file does not exists, raises I/O error.
  This is also the default mode in which file is opened.

- **Read and Write ('r+'):**
  Open the file for reading and writing.
  The handle is positioned at the beginning of the file.
  Raises I/O error if the file does not exists.

- **Write Only ('w'):**
  Open the file for writing.
  For existing file, the data is truncated and over-written.
  The handle is positioned at the beginning of the file.
  Creates the file if the file does not exists.

- **Write and Read ('w+'):**
  Open the file for reading and writing.

For existing file, data is truncated and over-written.
The handle is positioned at the beginning of the file.

- **Append Only ('a'):**
  Open the file for writing.
  The file is created if it does not exist.
  The handle is positioned at the end of the file.
  The data being written will be inserted at the end, after the existing data.

- **Append and Read ('a+'):**
  Open the file for reading and writing.
  The file is created if it does not exist.
  The handle is positioned at the end of the file.
  The data being written will be inserted at the end, after the existing data.

Reading from File

There are three ways to read data from a text file.

1. **read():**
   Returns the read bytes in form of a string. Reads n bytes, if no n specified, reads the entire file.
   File_object.read([n])
2. **readline():**
   Reads a line of the file and returns in form of a string.For specified n, reads at most n bytes. However, does not reads more than one line, even if n exceeds the length of the line.
   File_object.readline([n])

Read Lines

You can return one line by using the readline() method:

Example **Read one line of the file:**

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

By calling readline() two times, you can read the two first lines:

Example **Read two lines of the**
file:f= open("demofile.txt", "r")
print(f.readline())
print(f.readline())

3. **readlines():**
   Reads all the lines and return them as each line a string element in a list.
   File_object.readlines()

Note: '\n' is treated as a special character of two bytes.

EXample : demofile.txt

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

Read() :	**readline() :**	**readlines() :**

f = open("demofile.txt", "r")
print(f.read())
 	
output:
Hello! Welcome to demofile.txt
	f = open("demofile.txt", "r")
	print(f.readline())
	
	output:
		f = open("demofile.txt", "r")
		print(f.readlines())
		
		output:

This file is for testing purposes. Good Luck!	Hello! Welcome to demofile.txt	['Hello! Welcome to demofile.txt\n', 'This file is for testing purposes.\n', 'Good Luck!']

Writing to File

There are two ways to write in a file.

1. **write():**
   Inserts the string str1 in a single line in the text file.
   File_object.write(str1)

Example : Suppose we want to write more data to the above file using Python.

```
# Python program to demonstrate opening a file
# it's reference in the variable file1

file1 = open("myfile.txt", "a")
file1.write("\nWriting to file :)")          # Writing to file
file1.close()                               # Closing file
```

2. **writelines():**
   For a list of string elements, each string is inserted in the text file. Used to insert multiple strings at a single time.
   File_object.writelines(L) for L = [str1, str2, str3]

Note: '\n' is treated as a special character of two bytes.

4.2 Important Classes and Functions of CSV modules:

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

Python CSV Module

If you want to import or export spreadsheets and databases for use in the Python interpreter, you must rely on the CSV module, or Comma Separated Values format.

And the CSV module is a built-in function that allows Python to parse these types of files.

Python provides a CSV module to handle CSV files. To read/write data, you need to loop through rows of the CSV.
You need to use the split method to get data from specified columns.

Working with the CSV Module

To pull information from CSV files you use loop and split methods to get the data from individual columns.

CSV Functions

The CSV module includes all the necessary functions built in. They are:

- csv.reader – read data from a csv file
- csv.writer – write data to a csv file

Reading CSV Files

To pull data from a CSV file, you must use the reader function to generate a reader object.
The reader function is designed to take each line of the file and make a list of all columns.
Then, you just choose the column you want the variable data for.

Example 1 : csvfile_read.py

```
import CSV
With open('a.csv', 'r') as f:
```

```
reader = csv.reader(f)
for row in reader:
print row
```

Here, we have opened the **a.csv** file in reading mode using open() function.
Then, the csv.reader() is used to read the file, which returns an reader object.
The reader object is then iterated using a for loop to print the contents of each row.
Now, we will look at CSV files with different formats.

Example 2: Read CSV file Having Tab Delimiter [csvfile_read.py]

```
import csv

with open(a.csv', 'r') as file:
   reader = csv.reader(file, delimiter = '\t')
   for row in reader:
      print(row)
```

Output:

```
['SN', 'Name', 'Contribution']
['1', 'Linus Torvalds', 'Linux Kernel']
['2', 'Tim Berners-Lee', 'World Wide Web']
['3', 'Guido van Rossum', 'Python Programming']
```

CSV files with Custom Delimiters

By default, a comma is used as a delimiter in a CSV file.
However, some CSV files can use delimiters other than a comma.
Few popular ones are | and \t.
Suppose the **a.csv** file in **Example 1** was using **tab** as a delimiter.
To read the file, we can pass an additional delimiter parameter to the csv.reader() function.

CSV files with initial spaces

Some CSV files can have a space character after a delimiter.
When we use the default csv.reader() function to read these CSV files, we will get spaces in the output as well.
To remove these initial spaces, we need to pass an additional parameter called skipinitialspace.

Example 3: Read CSV files with initial spaces

Suppose we have a CSV file called **people.csv** with the following content:

```
SN, Name, City
1, John, Washington
2, Eric, Los Angeles
3, Brad, Texas
```

We can read the CSV file as follows:***csv_write1.py***

```
import csv
with open('people.csv', 'r') as csvfile:
   reader = csv.reader(csvfile, skipinitialspace=True)
   for row in reader:
      print(row)
```

Output

```
['SN', 'Name', 'City']
['1', 'John', 'Washington']
['2', 'Eric', 'Los Angeles']
['3', 'Brad', 'Texas']
```

The program is similar to other examples but has an additional skipinitialspace parameter which is set to True.
This allows the reader object to know that the entries have initial whitespace.
As a result, the initial spaces that were present after a delimiter is removed.

CSV files with quotes

Some CSV files can have quotes around each or some of the entries.

Using csv.reader() in minimal mode will result in output with the quotation marks.
In order to remove them, we will have to use another optional parameter called quoting.

Example 4: Read CSV files with quotes*csv_write1.py*

```
import csv
with open('person1.csv', 'r') as file:
    reader = csv.reader(file, quoting=csv.QUOTE_ALL, skipinitialspace=True)
    for row in reader:
        print(row)
```

Output

```
['SN', 'Name', 'Quotes']
['1', 'Buddha', 'What we think we become']
['2', 'Mark Twain', 'Never regret anything that made you smile']
['3', 'Oscar Wilde', 'Be yourself everyone else is already taken']
```

As you can see, we have passed csv.QUOTE_ALL to the quoting parameter. It is a constant defined by the csv module.
csv.QUOTE_ALL specifies the reader object that all the values in the CSV file are present inside quotation marks.
There are 3 other predefined constants you can pass to the quoting parameter:

- csv.QUOTE_MINIMAL - Specifies reader object that CSV file has quotes around those entries which contain special characters such as **delimiter**, **quotechar** or any of the characters in **lineterminator**.
- csv.QUOTE_NONNUMERIC - Specifies the reader object that the CSV file has quotes around the non-numeric entries.
- csv.QUOTE_NONE - Specifies the reader object that none of the entries have quotes around them.

Read CSV files with csv.DictReader()

The objects of a csv.DictReader() class can be used to read a CSV file as a dictionary.

Example 6: Python csv.DictReader()

Suppose we have a CSV file (**people.csv**) with the following entries:

Name	Age	Profession
Jack	23	Doctor
Miller	22	Engineer

Let's see how csv.DictReader() can be used.# csv.dictreader.py

```
import csv
with open("people.csv", 'r') as file:
    csv_file = csv.DictReader(file)
    for row in csv_file:
        print(dict(row))
```

Output:

```
{'Name': 'Jack', ' Age': ' 23', ' Profession': ' Doctor'}
{'Name': 'Miller', ' Age': ' 22', ' Profession': ' Engineer'}
```

As we can see, the entries of the first row are the dictionary keys. And, the entries in the other rows are the dictionary values.

Here, csv_file is a csv.DictReader() object. The object can be iterated over using a for loop. The csv.DictReader() returned an OrderedDict type for each row. That's why we used dict() to convert each row to a dictionary.
Notice that we have explicitly used the dict() method to create dictionaries inside the for loop.

```
print(dict(row))
```

Note: Starting from Python 3.8, csv.DictReader() returns a dictionary for each row, and we do not need to use dict() explicitly.

Python CSV writer

The csv.writer() method returns a writer object which converts the user's data into delimited strings on the given file-like object.

write_csv.py

```
#!/usr/bin/python3

import csv

nms = [[1, 2, 3, 4, 5, 6], [7, 8, 9, 10, 11, 12]]

f = open('numbers2.csv', 'w')

with f:

    writer = csv.writer(f)

    for row in nms:
        writer.writerow(row)
```

The script writes numbers into the numbers2.csv file. The writerow() method writes a row of data into the specified file.

```
$ cat numbers2.csv
1,2,3,4,5,6
7,8,9,10,11,12
```

It is possible to write all data in one shot. The writerows() method writes all given rows to the CSV file.

write_csv2.py

```
#!/usr/bin/python3

import csv

nms = [[1, 2, 3], [7, 8, 9], [10, 11, 12]]

f = open('numbers3.csv', 'w')
```

```
with f:

    writer = csv.writer(f)
    writer.writerows(nms)
```

The code example writes three rows of numbers into the file using the writerows() method.

Python CSV DictWriter

The csv.DictWriter class operates like a regular writer but maps Python dictionaries into CSV rows.

This class returns a writer object which maps dictionaries onto output rows.

The fieldnames parameter is a sequence of keys that identify the order in which values in the dictionary passed to the writerow() method are written to the CSV file.

Syntax: *csv.DictWriter(csvfile, fieldnames, restval=", extrasaction='raise', dialect='excel', *args, **kwds)*

Parameters:
csvfile: *A file object with write() method.*
fieldnames: *A sequence of keys that identify the order in which values in the dictionary should be passed.*
restval (optional): *Specifies the value to be written if the dictionary is missing a key in fieldnames.*
extrasaction (optional): *If a key not found in fieldnames, the optional extrasaction parameter indicates what action to take. If it is set to raise a ValueError will be raised.*
dialect (optional): *Name of the dialect to be used.*

csv.DictWriter provides two methods for writing to CSV. They are:

- **writeheader():** writeheader() method simply writes the first row of your csv file using the pre-specified fieldnames.
  Syntax:
  writeheader()
- writerows(): writerows method simply writes all the rows but in each row, it writes only the values(not keys).
  Syntax:
  writerows(mydict)

write_csv_dictionary.py

```
#!/usr/bin/python3

import csv

f = open('names.csv', 'w')

with f:

    fnames = ['first_name', 'last_name']
    writer = csv.DictWriter(f, fieldnames=fnames)

    writer.writeheader()
    writer.writerow({'first_name' : 'John', 'last_name': 'Smith'})
    writer.writerow({'first_name' : 'Robert', 'last_name': 'Brown'})
    writer.writerow({'first_name' : 'Julia', 'last_name': 'Griffin'})
```

The example writes the values from Python dictionaries into the CSV file using the csv.DictWriter.

```
writer = csv.DictWriter(f, fieldnames=fnames)
```

New csv.DictWriter is created. The header names are passed to the fieldnames parameter.

```
writer.writeheader()
```

The writeheader() method writes the headers to the CSV file.

```
writer.writerow({'first_name' : 'John', 'last_name': 'Smith'})
```

The Python dictionary is written to a row in a CSV file.

```
$ cat names.csv
first_name,last_name
John,Smith
Robert,Brown
Julia,Griffin
```

4.3 Dataframe Handling using Panda and Numpy:

DataFrame

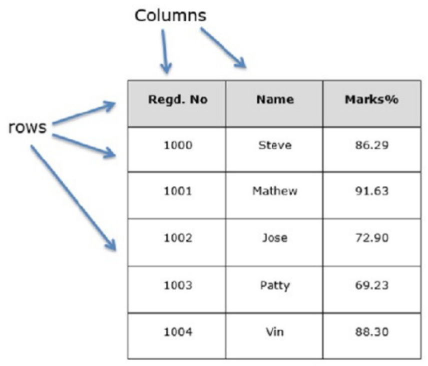
A Data frame is a two-dimensional data structure, i.e., data is aligned in a tabular fashion in rows and columns.

Features of DataFrame

- Potentially columns are of different types
- Size – Mutable
- Labeled axes (rows and columns)
- Can Perform Arithmetic operations on rows and columns

structure

Let us assume that we are creating a data frame with student's data.



Regd. No	Name	Marks%
1000	Steve	86.29
1001	Mathew	91.63
1002	Jose	72.90
1003	Patty	69.23
1004	Vin	88.30

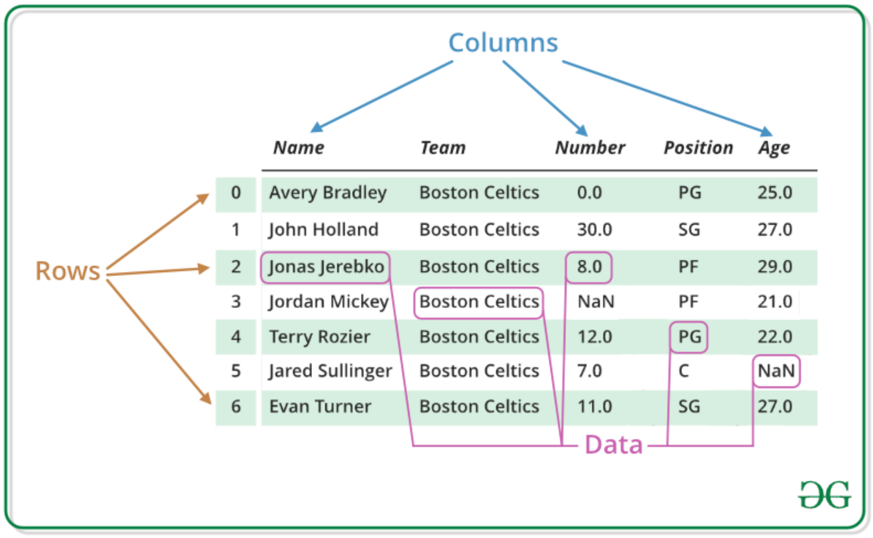
You can think of it as an SQL table or a spreadsheet data representation.

Pandas DataFrame

Pandas DataFrame is two-dimensional size-mutable, potentially heterogeneous tabular data structure with labeled axes (rows and columns).

A **Data frame** is a two-dimensional data structure, i.e., data is aligned in a tabular fashion in rows and columns.

Pandas DataFrame consists of three principal components, the data, rows, and columns.



	Name	Team	Number	Position	Age
0	Avery Bradley	Boston Celtics	0.0	PG	25.0
1	John Holland	Boston Celtics	30.0	SG	27.0
2	Jonas Jerebko	Boston Celtics	8.0	PF	29.0
3	Jordan Mickey	Boston Celtics	NaN	PF	21.0
4	Terry Rozier	Boston Celtics	12.0	PG	22.0
5	Jared Sullinger	Boston Celtics	7.0	C	NaN
6	Evan Turner	Boston Celtics	11.0	SG	27.0

pandas.DataFrame

A pandas DataFrame can be created using the following constructor −

pandas.DataFrame(data, index, columns, dtype, copy)

The parameters of the constructor are as follows −

Sr.No	Parameter & Description
1	**Data** :data takes various forms like ndarray, series, map, lists, dict, constants and also another DataFrame.
2	**Index** :For the row labels, the Index to be used for the resulting frame is Optional Default np.arange(n) if no index is passed.
3	**Columns** :For column labels, the optional default syntax is - np.arange(n). This is only true if no index is passed.
4	**Dtype** :Data type of each column.
5	**Copy** :This command (or whatever it is) is used for copying of data, if the default is False.

Create DataFrame

A pandas DataFrame can be created using various inputs like −

- **Lists**
- **dict**
- **Series**
- **Numpy ndarrays**
- **Another DataFrame**

Create an Empty DataFrame

A basic DataFrame, which can be created is an Empty Dataframe.

Example

```
#import the pandas library and aliasing as pd

import pandas as pd
df = pd.DataFrame()
print( df)

file : C:\notes\303_sqlite_python\panda\emptydf.py
```

Its output is as follows −

```
Empty DataFrame
Columns: []
Index: []
```

Create a DataFrame from Lists

The DataFrame can be created using a single list or a list of lists.

Example

```
import pandas as pd
data = [1,2,3,4,5]
df = pd.DataFrame(data)
print df

file : C:\notes\303_sqlite_python\panda\list_panda.py
```

Its output is as follows –

```
 0
0  1
1  2
2  3
3  4
4  5
```

Example

```
import pandas as pd
data = [['Alex',10],['Bob',12],['Clarke',13]]
df = pd.DataFrame(data,columns=['Name','Age'])
print( df)

file : C:\notes\303_sqlite_python\panda\list_panda.py
```

Its output is as follows –

```
     Name    Age
0    Alex    10
1    Bob     12
2    Clarke  13
```

Example

```
import pandas as pd
data = [['Alex',10],['Bob',12],['Clarke',13]]
df = pd.DataFrame(data,columns=['Name','Age'],dtype=float)
print df
```

Its output is as follows –

```
     Name   Age
0    Alex   10.0
1    Bob    12.0
2    Clarke 13.0
```

Note – Observe, the dtype parameter changes the type of Age column to floating point.

Create a DataFrame from Dict of ndarrays / Lists

All the ndarrays must be of same length.
If index is passed, then the length of the index should equal to the length of the arrays.
If no index is passed, then by default, index will be range(n), where n is the array length.

Example

```
import pandas as pd
data = {'Name':['Tom', 'Jack', 'Steve', 'Ricky'],'Age':[28,34,29,42]}
df = pd.DataFrame(data)
print( df)

file : C:\notes\303_sqlite_python\panda\panda_dict.py
```

Its output is as follows –

```
    Age    Name
0    28     Tom
1    34    Jack
2    29   Steve
3    42   Ricky
```

Note − Observe the values 0,1,2,3. They are the default index assigned to each using the function range(n).

Example

Let us now create an indexed DataFrame using arrays.

```
import pandas as pd
data = {'Name':['Tom', 'Jack', 'Steve', 'Ricky'],'Age':[28,34,29,42]}
df = pd.DataFrame(data, index=['rank1','rank2','rank3','rank4'])
print (df)

file : C:\notes\303_sqlite_python\panda\panda_dict.py
```

Its output is as follows −

```
        Age   Name
rank1    28    Tom
rank2    34   Jack
rank3    29  Steve
rank4    42  Ricky
```

Note − Observe, the index parameter assigns an index to each row.

Create a DataFrame from List of Dicts

List of Dictionaries can be passed as input data to create a DataFrame.
The dictionary keys are by default taken as column names.

Example 1

The following example shows how to create a **DataFrame** by passing a list of dictionaries.

```
import pandas as pd
data = [{'a': 1, 'b': 2},{'a': 5, 'b': 10, 'c': 20}]
df = pd.DataFrame(data)
print (df)
```

Its output is as follows −

```
   a   b     c
0  1   2   NaN
1  5  10  20.0
```

Note − Observe, NaN (Not a Number) is appended in missing areas.

Example 2

The following example shows how to create a DataFrame by passing a list of dictionaries and the row indices.

```
import pandas as pd
data = [{'a': 1, 'b': 2},{'a': 5, 'b': 10, 'c': 20}]
df = pd.DataFrame(data, index=['first', 'second'])
print (df)
```

Its output is as follows −

```
        a   b     c
first   1   2   NaN
second  5  10  20.0
```

Example 3

The following example shows how to create a DataFrame with a list of dictionaries, row indices, and column indices.

```
import pandas as pd
data = [{'a': 1, 'b': 2},{'a': 5, 'b': 10, 'c': 20}]
```

```
#With two column indices, values same as dictionary keys
df1 = pd.DataFrame(data, index=['first', 'second'], columns=['a', 'b'])

#With two column indices with one index with other name
df2 = pd.DataFrame(data, index=['first', 'second'], columns=['a', 'b1'])
print(df1)
print (df2)
```

Its output is as follows −

```
#df1 output
       a  b
first   1  2
second   5  10

#df2 output
       a  b1
first   1  NaN
second   5  NaN
```

Note − Observe, df2 DataFrame is created with a column index other than the dictionary key; thus, appended the NaN's in place. Whereas, df1 is created with column indices same as dictionary keys, so NaN's appended.

Create a DataFrame from Dict of Series

Dictionary of Series can be passed to form a DataFrame. The resultant index is the union of all the series indexes passed.

Example

```
import pandas as pd

d = {'one' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),
   'two' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}

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

Its output is as follows −

```
     one   two
a    1.0   1
b    2.0   2
c    3.0   3
d    NaN   4
```

Note − Observe, for the series one, there is no label 'd' passed, but in the result, for the d label, NaN is appended with NaN.