Unit IV

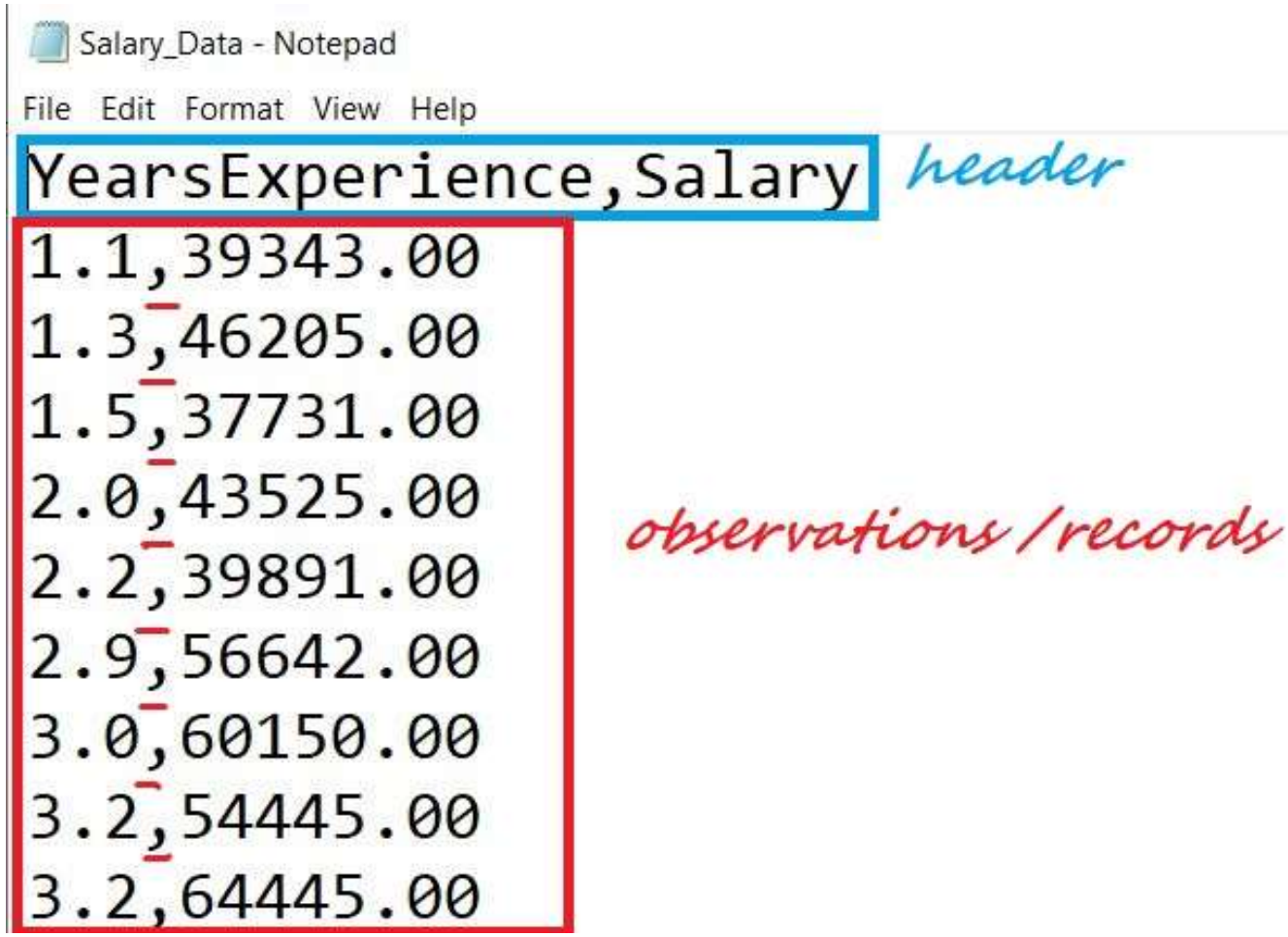
Introduction

- A file is a collection of data stored in one unit, identified by a filename. It can be a document, picture, audio or video stream, data library, application, or other collection of data.
- •Example: Suppose you want your python script to fetch data from the internet and then process that data. Now if data is small then this processing can be done every time you run the script but in case of huge data repetitive processing cannot be performed, hence the processed data needs to be stored. **This is where data storage or writing to a file comes into the picture.**

- An important component of an operating system is its file and directories.
- A file is a location on disk that stores related information and has a name. A hard-disk is non-volatile, and we use files to organize our data in different directories on a hard-disk.
- Since Random Access Memory (RAM) is volatile (which loses its data when the computer is turned off), we use files for future use of the data by permanently storing them.

What is a CSV file?

- **CSV** (Comma Separated Values) is a simple **file format** used to store tabular data, such as a spreadsheet or database.
- A CSV file stores tabular data (numbers and text) in plain text.
- Each line of the file is a data record.
- Each record consists of one or more fields, separated by commas.
- The use of the comma as a field separator is the source of the name for this file format.


Salary_Data - Notepad
File Edit Format View Help
YearsExperience,Salary
1.1,39343.00
1.3,46205.00
1.5,37731.00
2.0,43525.00
2.2,39891.00
2.9,56642.00
3.0,60150.00
3.2,54445.00
3.2,64445.00
header
observations / records

	A	B	C	D
1	name	area	country_code2	country_code3
2	Afghanistan	652090	AF	AFG
3	Albania	28748	AL	ALB
4	Algeria	2381741	DZ	DZA
5	American Samoa	199	AS	ASM
6	Andorra	468	AD	AND
7	Angola	1246700	AO	AGO
8	Anguilla	96	AI	AIA

- As you can see, the elements of a CSV file are separated by commas. Here, , is a delimiter.
- You can have any single character as your delimiter as per your needs.

Why is CSV File Format Used?

- CSV is a plain-text file which makes it easier for data interchange and also easier to import onto spreadsheet or database storage.
- For example: You might want to export the data of a certain statistical analysis to CSV file and then import it to the spreadsheet for further analysis. Overall it makes users working experience very easy programmatically.
- Any language supporting a text file or string manipulation like Python can work with CSV files directly.

Working with CSV files in Python

- While we could use the built-in open() function to work with CSV files in Python, there is a dedicated csv module that makes working with CSV files much easier.
- Before we can use the methods to the csv module, we need to import the module first using:
- **import csv**

- **#opening CSV file**
- import csv
- file = open("User_Data.csv")
- type(file)
- #_io.TextIOWrapper” which is a file object that is returned by the open() method.

- We can use csv.reader() to read from a CSV file
- The complete syntax of the csv.reader() function is:
- **csv.reader(csvfile, dialect='excel', **optional_parameters)**
- **Dialect** helps in grouping together many specific formatting patterns like delimiter, skipinitialspace, quoting, escapechar into a single dialect name.
- It can then be passed as a parameter to multiple writer or reader instances.

```
#reading CSV file
import csv
file = open("User_Data.csv")
#pass the file object to the reader() function of the csv module.
#The reader() function returns a csv reader object
csvreader = csv.reader(file)
```

```
#The csv_reader is an iterable object of lines from the CSV file.
#you can iterate over the lines of the CSV file using a for loop
for row in csvreader:
        print(row)
#Each line is a list of values.
#To access each value, you use the square bracket notation [].
#The first value has an index of 0.
#The second value has an index of 1, and so on.
```

- In the above example, we are using the csv.reader() function in default mode for CSV files having comma delimiter.
- However, the function is much more customizable.
- Suppose our CSV file was using **tab** as a delimiter. To read such files, we can pass optional parameters to the csv.reader() function

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 a csv file in 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

Extract the field names

- Create an empty list called header. Use the next() method to obtain the header.
- The .next() method returns the current row and moves to the next row.
- The first time you run next() it returns the header and the next time you run it returns the first record and so on.

- **#extracting the header**

```
import csv
header = []
file = open("User_Data.csv")
csvreader = csv.reader(file)
header = next(csvreader)
print(header)
```

Extract the rows/records

- Create an empty list called rows and iterate through the csvreader object and append each row to the rows list.

```
import csv
file = open("User_Data.csv")
csvreader = csv.reader(file)
rows = []
for row in csvreader:
        rows.append(row)
rows
```

using with() statement

- **Syntax: with open(filename, mode) as alias_filename:**
- **Modes:**
- ‘r’ – to read an existing file,
  ‘w’ – to create a new file if the given file doesn’t exist and write to it,
  ‘a’ – to append to existing file content,
  ‘+’ – to create a new file for reading and writing

- #using with statement

```
import csv
rows = []
with open("User_Data.csv", "r") as file:
    csvreader = csv.reader(file)
    header = next(csvreader)
    for row in csvreader:
        rows.append(row)
print(header)
print(rows)
```

Reading a CSV file using the DictReader class

When you use the csv.reader() function, you can access values of the CSV file using the bracket notation such as line[0], line[1], and so on. However, using the csv.reader() function has two main limitations:

- First, the way to access the values from the CSV file is not so obvious. For example, the line[0] implicitly means the country name. It would be more expressive if you can access the country name like line['country_name'].
- Second, when the order of columns from the CSV file is changed or new columns are added, you need to modify the code to get the right data.

- This is where the DictReader class comes into play. The DictReader class also comes from the csv module.
- The DictReader class allows you to create an object like a regular CSV reader. But it maps the information of each line to a dictionary (dict) whose keys are specified by the values of the first line.
- **Syntax:** csv.DictReader(file, fieldnames=None, restkey=None, restval=None, dialect='excel', *arguments, **keywords)

Example:-

```
import csv
with open("User_Data.csv", 'r') as file:
    csv_file = csv.DictReader(file)

    for row in csv_file:
        print(row)
```

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`
- 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 using dialect

- The CSV file has initial spaces, quotes around each entry, and uses a | delimiter.
- Instead of passing three individual formatting patterns, let's look at how to use dialects to read this file.

- import csv

```
csv.register_dialect('myDialect', delimiter='|', skipinitialspace=True, quoting=csv.QUOTE_ALL)
with open('office.csv', 'r') as csvfile:
reader = csv.reader(csvfile, dialect='myDialect’)
for row in reader:
    print(row)
```

the csv.register_dialect() function is used to define a custom dialect. It has the following syntax:

csv.register_dialect(name[, dialect[, **fmtparams]])

Writing CSV files in Python

- To write to a CSV file in Python, we can use the csv.writer() function.
- The csv.writer() function returns a writer object that converts the user's data into a delimited string. This string can later be used to write into CSV files using the writerow() function.
- Syntax:
- **csv.writer(csvfile, dialect='excel', **optional_parameters)**

- **Parameters**
- **csvfile** - file object having write() method
- **dialect** - It is an optional parameter that specifies the name of the dialect to be used
- **optionalparameters** - Certain optional formatting parameters that are used to overwrite the parameters specified in the dialect

- The csv.writer class is used to insert data in CSV files. It returns a writer object which is responsible for converting the user’s data into a delimited string.
- A CSV file object should be opened with newline='' otherwise newline characters inside the quoted fields will not be interpreted correctly

csv.writer class provides two methods for writing data in a CSV file, which include the following

- **writerow():** This method is used when we want to write only a single row at a time in our CSV file. writerow() can be used to write field rows.
- **Syntax**

writerow(fields)

writerows(): This method is used to write multiple rows at a time. writerows() can be used to write a list of rows.

Syntax

writerows(rows)

```
#writing into CSV file
import csv
with open('studentdata.csv', 'w') as file:
    writer = csv.writer(file)
    writer.writerow(["SN", "Eno", "Name"])
    writer.writerow([1, 101, "abc"])
    writer.writerow([2, 102, "efg"])
```

Writing multiple rows with writerows()

- If we need to write the contents of the 2-dimensional list to a CSV file
- #writerows()

```
import csv
csv_rowlist = [["SN", "Eno", "Name"], [1, 101, "abc"],
               [2, 102, "efg"]]
with open('studentdata1.csv', 'w') as file:
    writer = csv.writer(file)
    writer.writerows(csv_rowlist)
```

Writing to a CSV File with Tab Delimiter

```
import csv
with open('protagonist.csv', 'w') as file:
writer = csv.writer(file, delimiter = '\t')
 writer.writerow(["SN", "Eno", "Name"])
 writer.writerow([1, 101, "abc"])
 writer.writerow([2, 102, "efg"])
```

Using csv.DictWriter class

- This class returns a writer object which maps dictionaries onto output rows
- **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)