1. Introduction

We can use concept of file handling to receive input from and writing output to files. A file stores data on a permanent medium such as a disk. When compared to keyboard input from a human user, the main advantages of taking input data from a file are the following:

- The data set can be much larger.
- The data can be input much more quickly and with less chance of error.
- The data can be used repeatedly with the same program or with different programs.

File handling is an important part of any software application and Python has several functions for creating, reading, updating, and deleting files.

The key function for working with files in Python is the **open()** function. The *open()* function takes two parameters: *filename* and *mode*. The four different modes for opening a file are:

- Read("r") – Default value. Opens a file for reading, error if the file does not exist.
- Append ("a") – Opens a file for appending, creates the file if it does not exist.
- Write ("w") – Opens a file for writing, creates the file if it does not exist.
- Create ("x") – Creates the specified file for writing, returns an error if the file exists.
- (+) - Open a file for updating (reading and writing).

In addition, we can specify if the file should be handled as binary or text mode as given below.

- Text("t") – Default value. Text mode.
- Binary ("b") – Binary mode such as images.

For example, **open("img.bmp ", "r+b")** opens the img.bmp file for reading and writing in binary mode.

2. Reading Files

There are several ways to read data from an input file. The simplest way is to use the file method **read()** to input the entire contents of the file as a single string. If the file contains multiple lines of text, the newline characters will be embedded in this string. For example,

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

By default the *read()* method returns the whole text, but we can also specify how many characters we want to return. For example,

```
f = open("test.txt", "r")
print(f.read(4))
```

We can use **readline()** method to return one line. For example,

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

By looping through the lines of the file, we can read the whole file, line by line. For example,

```
f = open("test.txt", "r")
```

```
for x in f:
   print(x)
```

It is a good practice to always close the file when we are done with it. For example, *f.close()*.

3. Writing to Files

Data can be output to a file using a file object. To write to an existing file, you must add a parameter to the open() function: "a" will append to the end of the file and "w" will overwrite any existing content. For example,

```
f = open("test.txt", "a")
f.write("\nThis line will be added")
f.close()
f = open("test.txt", "r")
print(f.read())
```

4. Deleting Files

To delete a file, we must import the OS module, and run its os.remove() function. For example,

```
import os
os.remove("test.txt")
```

To avoid getting an error, we can check if the file exists before we try to delete it. For example,

```
import os
if os.path.exists("test.txt"):
  os.remove("demofile.txt")
else:
  print("The file does not exist")
```

To delete an empty folder, use the os.rmdir() method. For example,

```
import os
os.rmdir("test")
```

5. Some Common Functions in OS Module

This module provides a portable way of using operating system dependent functionality. Some common functions of this module are:

- **getcwd()** – Returns the path of the current working directory.
- **listdir(path)** – Returns a list of the names in directory named path.
- **mkdir(path)** – Creates a new directory named path and places it in the current working directory.
- **rename(old, new)** – Renames the file or directory named old to new.
- **exists(path)** – Returns True if path exists and False otherwise.
- **isdir(path)** – Returns True if path names a directory and False otherwise.
- **isfile(path)** – Returns True if path names a file and False otherwise.
- **getsize(path)** – Returns the size of the object names by path in bytes.

6. Reading and Writing CSV Files

CSV stands for comma-separated values. A CSV file is a delimited text file that uses a comma to separate values. A CSV file consists of one or more lines. Each line is a data record. And each data record consists of one or more values separated by commas.

In addition, all the lines of a CSV file have the same number of values. Typically, you use a CSV file to store tabular data in plain text. The CSV file format is quite popular and supported by many software applications such as Microsoft Excel and Google Spreadsheet.

6.1. Reading a CSV file

To read a CSV file in python, first we import the **csv** module and use built-in **open()** function to open the csv file in read mode. For example,

```
import csv
f = open("test.csv")
```

If the csv file contains UTF8 characters, you need to specify the encoding as given below.

```
import csv
f = open("test.csv", encoding="UTF8")
```

Now, reader function of the csv module is used to returns csv reader object as given below.

```
import csv
f = open("test.csv", encoding="UTF8")
csv_reader = csv.reader(f)
```

The csv_reader is an iterable object of lines from the CSV file. Therefore, you can iterate over the lines of the CSV file using a for loop as shown below.

```
import csv
f = open("test.csv")
csv_reader = csv.reader(f)
for line in csv_reader:
    print(line)
```

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. For example,

```
import csv
f = open("test.csv")
csv_reader = csv.reader(f)
for line in csv_reader:
    for x in range(3):
        print(line[x])
```

Finally, always close the file once you're no longer access it by calling the **close()** method.

It'll be easier to use the with statement so that you don't need to explicitly call the close() method. For example,

```
import csv
with open("test.csv","r")as f:
    csv_reader = csv.reader(f)
    for line in csv_reader:
        print(line)
```

To separate the header and data, you use the **enumerate()** function to get the index of each line. The *enumerate()* is a built-in function that allows you to loop over an iterable object and keep track of how many iterations have occurred. Its syntax is, ***enumerate(iterable, start)***, where iterable is an iterable object and start is the start number. Default start number is zero. For example,

```
import csv
with open("test.csv","r")as f:
    csv_reader = csv.reader(f)
    for line_no, line in enumerate(csv_reader, 1):
        if line_no == 1:
            print('Header:')
            print(line)  # header
            print('Data:')
        else:
            print(line)  # data
```

Another way to skip the header is to use the **next()** function. The *next()* function forwards to the reader to the next line. For example,

```
import csv
with open("test.csv","r")as f:
    csv_reader = csv.reader(f)
    next(csv_reader)
    for line in csv_reader:
        print(line)
```

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. 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 whose keys are specified by the values of the first line. For example,

```
import csv
with open("test.csv","r")as f:
    csv_reader = csv.DictReader(f)
    for line in csv_reader:
        print(line['ID'])
```

6.2. Writing a CSV file

The steps we follow to write data into a CSV file are:

- First, open the CSV file for writing (w mode) by using the open() function.
- Second, create a CSV writer object by calling the writer() function of the csv module.
- Third, write data to CSV file by calling the writerow() or writerows() method of the CSV writer object.
- Finally, close the file once you complete writing data to it.

For example,

```
import csv
header = ['name', 'area', 'country_code2', 'country_code3']
data = ['Afghanistan', 652090, 'AF', 'AFG']
with open('test.csv', 'w', encoding='UTF8', newline='') as f:
    writer = csv.writer(f)
    writer.writerow(header)
    writer.writerow(data)
```

To write multiple rows to a CSV file at once, you use the writerows() method of the CSV writer object. For example,

```
import csv
header = ['name', 'area', 'country_code2', 'country_code3']
data = [
    ['Albania', 28748, 'AL', 'ALB'],
    ['Algeria', 2381741, 'DZ', 'DZA'],
    ['American Samoa', 199, 'AS', 'ASM'],
    ['Andorra', 468, 'AD', 'AND'],
    ['Angola', 1246700, 'AO', 'AGO']
]
with open('countries.csv', 'w', encoding='UTF8', newline='') as f:
    writer = csv.writer(f)
    writer.writerow(header)
    writer.writerows(data)
```

If each row of the CSV file is a dictionary, you can use the **DictWriter** class of the csv module to write the dictionary to the CSV file. For example,

```
import csv
header = ['name', 'area', 'country_code2', 'country_code3']
rows = [
    {'name': 'Albania',
    'area': 28748,
    'country_code2': 'AL',
    'country_code3': 'ALB'},
    {'name': 'Algeria',
    'area': 2381741,
    'country_code2': 'DZ',
    'country_code3': 'DZA'},
    {'name': 'American Samoa',
    'area': 199,
    'country_code2': 'AS',
    'country_code3': 'ASM'}
]
with open('countries.csv', 'w', encoding='UTF8', newline='') as f:
    writer = csv.DictWriter(f, fieldnames=header)
    writer.writeheader()
    writer.writerows(rows)
```