

Files

Learning objective:

After going through the chapter, the student will be able to:

- Understand the importance of data file for permanent storage of data
- Understand how standard Input/Output function work
- Distinguish between text and binary file
- Open and close a file (text and binary)
- Read and write data in file
- Write programs that manipulate data file(s)

Objective:

Python file handling is one of the essential topics for programmers and automation testers. As both of them needs to work with files either to write to a file or to read data from it. Also, if you are not already aware, I/O operations are the costliest operations where a program can stumble. Hence, you should be quite careful while implementing file handling for reporting or any other purpose. Optimizing a single file operation can help you produce an high-performing application or a robust solution for automated software testing

What is a file?

What if the data with which, we are working or producing as output is required for later use? Result processing done in Term Exam is again required for Annual Progress Report. Here if data is stored permanently, its processing would be faster. This can be done if we are able to store data in secondary storage media i.e. Hard Disk, which we know is permanent storage media. Data is stored using file(s) permanently on secondary storage media. You have already used the files to store your data permanently - when you were storing data in Word processing applications, Spreadsheets, Presentation applications, etc. All of them created data files and stored your data, so that you may use the same later on. Apart from this, you were permanently storing your python scripts (as .py extension) also.

Here is a basic definition of file handling in Python, “File is a named location on the system storage which records data for later access. It enables persistent storage in a non-volatile memory i.e. Hard disk.”

In python files are simply stream of data, so the structure of data is not stored in the file, along with data. Basic operations performed on a data file are:

- Naming a file
- Opening a file
- Reading data from the file
- Writing data in the file
- Closing a file

Using these basic operations, we can process a file in many ways, such as

Creating a file

Traversing a file for displaying the data on screen

Appending data in file

Inserting data in file

Deleting data from file

Create a copy of file

Updating data in the file, etc

File types

Python allows us to create and manage two types of file

Text:

A text file is usually considered as a sequence of lines. A line is a sequence of characters (ASCII), stored on permanent storage media. Although default character coding in python is ASCII but using constant `u` with string, supports Unicode as well. As we talk of lines in a text file, each line is terminated by a special character, known as End of Line (EOL). From strings, we know that `\n` is a newline character. So at the lowest level, a text file will be a collection of bytes. Text files are stored in human readable form and they can also be created using any text editor.

Binary:

A binary file contains arbitrary binary data i.e. numbers stored in the file, can be used for numerical operation(s). So when we work on a binary file, we have to interpret the raw bit pattern(s) read from the file into the correct type of data in our program. It is perfectly possible to interpret a stream of bytes originally written as a string, as a numeric value. But we know that will be an incorrect interpretation of data and we are not going to get desired output after the file

processing activity. So in the case of a binary file, it is extremely important that we interpret the correct data type while reading the file. Python provides a special module(s) for encoding and decoding of data for the binary file.

File Operations

Opening a file:

How do you write data to a file and read the data back from a file? You need to first create a file object that is associated with a physical file. This is called opening a file. The syntax for opening a file is:

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

The open function returns a file object for filename. The mode parameter is a string that specifies how the file will be used (for reading or writing, Note: you can find the table for more file modes)

EX:

For example, the following statement opens a file named Scores.txt in **the current directory** for reading:

```
file= open("Scores.txt", "r")
```

You can also use the absolute filename to open the file in Windows, as follows:

```
file = open("c:\pybook\Scores.txt", "r")
```

Example code for Opening a file:

```
file = open("Scores.txt", "r")
## in a open(file_name,file_accessmode) here Scores.txt file name
## r is the file access mode i.e read only
print(file.read())
## read() method will read the data from file
## print() method will prints the data from file object
file.close() ## closing the file[will be explained in the closing file]
```

File access modes

Sr.No.	Modes & Description
1	r Opens a file for reading only. The file pointer is placed at the beginning of the file. This is the default mode.

2	rb Opens a file for reading only in binary format. The file pointer is placed at the beginning of the file. This is the default mode.
3	r+ Opens a file for both reading and writing. The file pointer placed at the beginning of the file.
4	rb+ Opens a file for both reading and writing in binary format. The file pointer placed at the beginning of the file.
5	w Opens a file for writing only. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
6	w+ Opens a file for both writing and reading. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.
7	wb Opens a file for writing only in binary format. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
8	wb+ Opens a file for both writing and reading in binary format. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.
9	a Opens a file for appending. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
10	a+ Opens a file for both appending and reading. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it

	creates a new file for reading and writing.
11	ab Opens a file for appending in binary format. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
12	ab+ Opens a file for both appending and reading in binary format. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.

The File Object Attributes:

Once a file is successfully opened, a file object is returned. Using this file object, you can easily access different types of information related to that file. This information can be obtained by reading values of specific attributes of the file.

Attribute	Information Obtained
Fileobj.closed	Returns True if the file is closed and False otherwise
Fileobj.mode	Return access mode with which files has been opened
Fileobj.name	Returns name of the file

Example: Program to open a file and print its attribute values

```
infile=open("student.txt","r")
print("Name of the file: ",infile.name)
print("file is closed: ",infile.closed)
print("file has been opened in: ",infile.mode," mode")
```

Output

```
Name of the file:  student.txt
file is closed:  False
file has been opened in:  r  mode
```

Writing data in the file:

The open function creates a file object, which is an instance of the `_io.TextIOWrapper` class. This class contains the methods for reading and writing data and for closing the file

File methods:

A file object contains the methods for reading and writing data are as follows

read([number.int]): str ---> Returns the specified number of characters from the file. If the argument is omitted, the entire remaining contents in the file are read.

readline(): str ---> Returns the next line of the file as a string

readlines(): list----> Returns a list of the remaining lines in the file.

write(s: str): None --> Writes the string to the file.

close(): None----> Closes the file

write(str):

Write a string to the file. There is no return value. Due to buffering, the string may not actually show up in the file until the `flush()` or `close()` method is called.

writelines(sequence):

Write a sequence of strings to the file. The sequence can be any iterable object producing strings, typically a list of strings. There is no return value. (The name is intended to match `readlines()`; `writelines()` does not add line separators.) After a file is opened for writing data, you can use the `write` method to write a string to the file

Here is writing demo into files

```
def main():  
    # Open file for output  
    outfile = open("student.txt", "w")  
    # Write data to the file  
    outfile.write("SomeswaraRao\n")## \n is a newline character in the files  
    outfile.write("Rasagnya\n")  
    outfile.write("Sharvin\n")  
    outfile.write("Suryakala\n")  
    outfile.close()# Close the output file  
main()# Call the main function
```

The program opens a file named student.txt using the w mode for writing data (line 3). **If the file does not exist, the open function creates a new file. If the file already exists, the contents of the file will be overwritten with new data.** You can now write data to the file. When a file is opened for writing or reading, a special marker called a file pointer is positioned internally in the file. A read or write operation takes place at the pointer's location. When a file is opened, the file pointer is set at the beginning of the file. When you read or write data to the file, the file pointer moves forward

The program invokes the write method on the file object to write four strings

The program closes the file to ensure that data is written to the file

Here is writelines() demo into files

```
infile=open("write.txt","w")  
lines=["Hello world, ","Welcome to the world of python ","Enjoy learning python"]  
infile.writelines(lines)  
infile.close()
```

The program invokes the writelines method on the file object to write three strings. The program closes the file to ensure that data is written to the file

Testing a File's Existence:

To prevent the data in an existing file from being erased by accident, you should test to see if the file exists before opening it for writing. The isfile function in the os.path module can be used to determine whether a file exists.



Testing a File's Existence Code:

```
import os.path
if os.path.isfile("student.txt"):
    print("student.txt exists")
else:
    print("student.txt DOES NOT exists")
```

Here `isfile("students.txt.txt")` returns `True` if the file `students.txt` exists in the current directory.

Reading Data:

After a file is opened for reading data, you can use the read method to read a specified number of characters or all characters from the file and return them as a string, the readline() method to read the next line, and the readlines() method to read all the lines into a list of strings.

Reading Data demo code:

```
def main():
```

```
    # Open file for input
```

```
    infile = open("student.txt", "r")
```

```
    print("(1) Using read(): ")
```

```
    print(infile.read())
```

```
    """ read() method reads all characters from the file and returns them as a string"""
```

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

```
    # Open file for input
```

```
    infile = open("student.txt", "r")
```

```
    print("\n(2) Using read(number): ")
```

```
    s1=infile.read(4)
```

```
    """
```

```
    read(4) -- read(number) method to read the specified number of characters from the file.
```

```
    Invoking infile.read(4) reads 4 characters
```

```
    """
```

```
        print(s1)
```

```
        s2 = infile.read(10)
```

```
        print(repr(s2))
```

```
    """
```

```
    The repr(s)
```

```
    repr(s2)-function returns a raw string for s, which causes the escape sequence to be displayed as literals, as shown in the output.
```

```
    """
```

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

```
    # Open file for input
```

```
    infile = open("student.txt", "r")
```

```
    print("\n(3) Using readline(): ")
```

```
    line1 = infile.readline()
```

```
    line2 = infile.readline()
```

```
    line3 = infile.readline()
```

```

line4 = infile.readline()
print(repr(line1))
print(repr(line2))
print(repr(line3))
print(repr(line4))
infile.close() # Close the input file

```

```

# Open file for input
infile = open("student.txt", "r")
print("\n(4) Using readlines(): ")
print(infile.readlines())

```

readlines() method to read all lines and return a list of strings

```

infile.close() # Close the input file
main() # Call the main function

```

Output Screen shot

```

>>>
===== RESTART: C:\Users\Sharvin\Desktop\Python_Practice\Files\files.py =====
(1) Using read():
SomeswaraRao
Rasagnya
Sharvin
Suryakala

(2) Using read(number):
Some
'swaraRao\nR'

(3) Using readline():
'SomeswaraRao\n'
'Rasagnya\n'
'Sharvin\n'
'Suryakala'

(4) Using readlines():
['SomeswaraRao\n', 'Rasagnya\n', 'Sharvin\n', 'Suryakala']
>>>

```

Activate Window
Go to Settings to activate Windows

Reading all data from the file

Programs often need to read all data from a file. Here are two common approaches to accomplishing this task:

1. Use the `read()` method to read all data from the file and return it as one string.
2. Use the `readlines()` method to read all data and return it as a list of strings.

These two approaches are simple and appropriate for small files, but what happens if the file is so large that its contents cannot be stored in the memory? You can write the following loop to read one line at a time, process it, and continue reading the next line until it reaches the end of the file

Using while loop

```
infile = open("student.txt", "r")
line = infile.readline() # Read a line
while line != "":
    print(line)
    # Process the line here ...
    # Read next line
    line = infile.readline()
infile.close()
```

Using for loop

```
infile = open("student.txt", "r")
line = infile.readline()
for line in infile:
    print(line)
infile.close()
```

Appending Data

You can use the `a` mode to open a file for appending data to the end of an existing file.

```
def main():
```

```
    # Open file for appending data
    outfile = open("Info.txt", "a")
    outfile.write("\nPython is interpreted\n")
    outfile.close()# Close the file
```

Closing a file:

When you're done working, you can use the **`file.close()`** command to end things. What this does is close the file completely, terminating resources in use, in turn freeing them up for the system to deploy elsewhere.

It's important to understand that when you use the **`file.close()`** method, any further attempts to use the file object will fail.

tell():

`tell()` function tells the current position within the file at which the next read or write operation will occur. It is specified as number of bytes from the beginning of the file. When you just open a file for reading, the file pointer is positioned at location 0, which is the beginning of the file.

Program:

```
infile = open("student.txt", "r")
print("Position of file pointer before reading is: ",infile.tell())
print(infile.read(6))
print("Position of file pointer after reading is: ",infile.tell())
infile.close()
```

output:

```
Position of file pointer before reading is: 0
Somesw
Position of file pointer after reading is: 6
```

split():

Python allows you to read line(s) from a file and splits the line(treated as a string) based on a character. By default, this character is space but you can even specify any other character to split words in the string.

Program:

```
infile=open("write.txt","r")
line=infile.readline()
words=line.split()
print(words)
infile.close()
```

output:

```
['Hello', 'world,', 'Welcome', 'to', 'the', 'world', 'of', 'python', 'Enjoy', 'learning', 'python']
```

Solved Problems:

1. Write a program that counts the number of tabs, spaces and newline characters in a file.

Solution:

```
infile=open("write.txt","r")
text=infile.read()
count_tab=0
count_space=0
count_nl=0
for char in text:
    if char == '\t':
        count_tab += 1
    elif char == ' ':
        count_space += 1
    elif char == '\n':
        count_nl += 1
print("Tabs =", count_tab)
print("Spaces =", count_space)
print("New lines =", count_nl)
infile.close()
```

2. Write a program that reads a file line by line. Each line read from the file is copied to another file with line numbers specified at the beginning of the line.

Solution:

```
file1=open("file1.txt","r")
file2=open("file2.txt","w")
num=1
for line in file1:
    file2.write(str(num)+ ":" + line)
    num=num+1
file1.close()
file2.close()
```

3. Write a program that accepts filename as an input from the user. Open the file and count the number of times a character appears in the file.

Solution:

```
filename=input("Enter the filename : ")
file1=open(filename,"r")
text=file1.read()
letter=input("Enter the character to be searched: ")
count=0
for char in text:
    if char == letter:
        count +=1
print(letter, "appears ", count, "times in file")
file1.close()
```



Problem sets

1. How do you open a file for reading, for writing, and for appending, respectively?
2. When you open a file for reading, what happens if the file does not exist? When you open a file for writing, what happens if the file already exists?
3. How do you determine whether a file exists?
4. What method do you use to read 30 characters from a file?
5. What method do you use to read all data into a string?
6. What method do you use to read a line?
7. What method do you use to read all lines into a list?
8. Will your program have a runtime error if you invoke `read()` or `readline()` at the end of the file?
9. When reading data, how do you know if it is the end of the file?
10. What function do you use to write data to a file?

IT Quiz

1. Does tell() function has parameters?
 - a. Yes
 - b. No
2. Which function is used to write a sequence of strings to the file?
 - a) Writeline() b) read lines() c) Writelines() d) readline()
3. What is the meaning of 'a' mode?
 - a) Read and write b) delete c) Appending d) None of these
4. What is the use of tell() function?
 - a) Return the file's current position b) Return the files previous position c) Return the files end position d) None of these
5. What is the use of 'r' mode?
 - a) Write b) only Read c) Read and Write d) Above all
6. Which of the following function reads the text from file by line by line?
 - (a)file.read() (b)file.readLine() (c) file.readline() (d) file.readlines()
7. what is the function to read all lines from a file? When file = open(" read_it.txt", "r")
 - (a) file.read lines() (b)file.read() (c)read(file) (d) none
8. What is the new line tag in python?
 - (a) print "\n" (b) print "n" (c) "/"n") (d) new line
9. What is the function to close a file? If file=open("object.txt","r")?
 - (a) file.close() (b)object.close() (c) close() (d) none
10. Which one of the following is the correct function to read a file? When f=open("file.txt","r")
 - (a) f.read() (b) f(read) (c)read.f() (d) none
11. When we use w mode in opening file, what does it mean?
 - (a) to append text (b) to read text (c) to create new file (d) none
12. Which function reads the text from a file all lines at once?
 - (a) file.read() (b) file.readlines() (c) file.readLiine() (d) non
13. open() function in python is function
 - (a) user defined function (b) built-in-function (c) both (d) none
14. How many arguments can we pass in open function?
 - (a) 1, (b) 2 (c) 3 (d) both a and c

Using String methods in files concept

Dear students use all string methods in files that you have learned in string module, because after reading whole data in a file that is the string or file object, here are some examples

find method:

The find() method returns the lowest index of the substring if it is found in given string. If its is not found then it returns -1.

Syntax :

```
str.find(sub,start,end)
```

Parameters :

sub : It's the substring which needs to be searched in the given string.

start : Starting position where sub is needs to be checked within the string.

end : Ending position where suffix is needs to be checked within the string.

NOTE : If start and end indexes are not provided then by default it takes 0 and length-1 as starting and ending indexes where ending indexes is not included in our search.

Returns:

returns the lowest index of the substring if it is found in the given string. If it's not found then it returns -1.

```
word = 'geeks for geeks'
```

```
# returns the first occurrence of Substring
```

```
result = word.find('geeks')
```

```
print ("Substring 'geeks' found at index:", result )
```

```
result = word.find('for')
```

```
print ("Substring 'for ' found at index:", result )
```

```
# How to use find()
```

```
if (word.find('pawan') != -1):
```

```
    print ("Contains given substring ")
```

```
else:
```


```
    print ("Doesn't contain's given substring")
```

```
find() With start and end Arguments:
```

```
quote = 'Do small things with great love'
```

```
# Substring is searched in 'hings with great love'
```

```
print(quote.find('small things', 10))
```

```
# Substring is searched in ' small things with great love'
print(quote.find('small things', 2))
```

```
# Substring is searched in 'hings with great lov'
print(quote.find('o small ', 10, -1))
```

```
# Substring is searched in 'll things with'
print(quote.find('things ', 6, 20))
```

Replace

`replace()` is an inbuilt function in Python programming language that returns a copy of the string where all occurrences of a substring is replaced with another substring.

Syntax :

```
string.replace(old, new, count)
```

Parameters

old – old substring you want to replace.

new – new substring which would replace the old substring.

count – the number of times you want to replace the old substring with the new substring.

(Optional)

Return Value :

It returns a copy of the string where all occurrences of a substring is replaced with another substring.

Below is the code demonstrating `replace()` :

```
# Python3 program to demonstrate the
# use of replace() method
string = "geeks for geeks geeks geeks"
# Prints the string by replacing geeks by Geeks
print(string.replace("geeks", "Geeks"))
# Prints the string by replacing only 3 occurrence of Geeks
print(string.replace("geeks", "GeeksforGeeks", 3))
```

Python Directory and Files Management

What is Directory in Python?

If there are a large number of files to handle in your Python program, you can arrange your code within different directories to make things more manageable.

A directory or folder is a collection of files and subdirectories. Python has the `os` module, which provides us with many useful methods to work with directories (and files as well).

Get Current Directory

We can get the present working directory using the `getcwd()` method.

This method returns the current working directory in the form of a string. We can also use the `getcwdb()` method to get it as bytes object.

Ex:

```
>>> import os
>>> os.getcwd()
>>> os.getcwdb()
```

Changing Directory:

We can change the current working directory using the `chdir()` method.

The new path that we want to change to must be supplied as a string to this method. We can use both forward slash (/) or the backward slash (\) to separate path elements. It is safer to use escape sequence when using the backward slash.

```
>>> os.chdir('C:\\Python33')
>>> print(os.getcwd())
C:\\Python33
```

List Directories and Files

All files and sub-directories inside a directory can be known using the `listdir()` method. This method takes in a path and returns a list of sub directories and files in that path. If no path is specified, it returns from the current working directory.

Making a New Directory:

We can make a new directory using the `mkdir()` method. This method takes in the path of the new directory. If the full path is not specified, the new directory is created in the current working directory.

Ex:

```
>>> os.mkdir('test')
>>> os.listdir()
['test']
```

Renaming a Directory or a File:

The `rename()` method can rename a directory or a file. The first argument is the old name and the new name must be supplied as the second argument.

Ex:

```
>>> os.listdir()
['test']
>>> os.rename('test','new_one')
>>> os.listdir()
['new_one']
```

Removing Directory or File:

A file can be removed (deleted) using the `remove()` method. Similarly, the `rmdir()` method removes an empty directory.

Ex:

```
>>> os.listdir()
['new_one', 'old.txt']
>>> os.remove('old.txt')
>>> os.listdir()
['new_one']
```

```
>>> os.rmdir('new_one')
>>> os.listdir()
```

However, note that `rmdir()` method can only remove empty directories. In order to remove a non-

empty directory we can use the `rmtree()` method inside the `shutil` module.

Ex:

```
>>> os.listdir()
['test']
>>> os.rmdir('test')
Traceback (most recent call last):
...
OSError: [WinError 145] The directory is not empty: 'test'
>>> import shutil
>>> shutil.rmtree('test')
>>> os.listdir()
```

Answers to be remember

Why files?

Data used in a program is temporary; unless the data is specifically saved, it is lost when the program terminates. To permanently store the data created in a program, you need to save it in a file on a disk or some other permanent storage device. The file can be transported and can be read later by other programs.

Why exception handling?

What happens if your program tries to read data from a file but the file does not exist? Your program will be abruptly terminated. We can write a program to handle this exception so the program can continue to execute.

Absolute filename

A file is placed in a directory in the file system. An absolute filename contains a filename with its complete path and drive letter. For example, `c:\pybook\Scores.txt` is the absolute filename for the file `Scores.txt` on the Windows operating system. Absolute filenames are machine dependent. On the UNIX platform, the absolute filename may be `/home/liang/pybook/Scores.txt`

Directory path

Here, `c:\pybook` is referred to as the directory path to the file in windows operating system., where `/home/liang/pybook` is the directory path to the file `Scores.txt` in UNIX operating system

Relative filename

A relative filename is relative to its current working directory. The complete directory path for a relative file name is omitted. For example, `Scores.py` is a relative filename. If its current working directory is `c:\pybook`, the absolute filename would be `c:\pybook\Scores.py`.

Text file & Binary file

Files can be classified into text or binary files. A file that can be processed (that is, read, created, or modified) using a text editor such as Notepad on Windows or vi on UNIX is called a text file. All the other files are called binary files. For example, Python source programs are stored in text files and can be processed by a text editor, but Microsoft Word files are stored in binary files and are processed by the Microsoft Word program.

Why binary file?

Although it is not technically precise and correct, you can envision a text file as consisting of a sequence of characters and a binary file as consisting of a sequence of bits. Characters in a text file are encoded using a character encoding scheme such as ASCII and Unicode. For example, the decimal integer 199 is stored as the sequence of the three characters 1, 9, and 9, in a text file, and the same integer is stored as a byte-type value C7 in a binary file, because decimal 199 equals hex C7 ($199 = 12 \times 16^1 + 7$). The advantage of binary files is that they are more efficient to process than text files.

NOTE:

Computers do not differentiate between binary files and text files. All files are stored in binary format, and thus all files are essentially binary files. Text IO (input and output) is built upon binary IO to provide a level of abstraction for character encoding and decoding.