_ :Student Name________________________ _ :Roll No______________ _ :Section____________

Experiment No. 07

Lab 07 – Introduction to Files

Lab Objectives:

1. Introduction to Files
2. Patterns for reading Text Files.
3. Writing to a Text File.

Files 1.1

A file could be a text document or spreadsheet, an HTML file, or a Python module. Such files are referred to as text files. Text files contain a sequence of characters that are encoded using some encoding (ASCII, utf-8, etc.). A file also can be an executable application (like python.exe), an image, or an audio file. These files are referred to as binary files because they are just a sequence of bytes and there is no encoding. All files are managed by the file system, which we introduce next.

Opening and Closing a File Processing a file consists of these three steps:
1. Opening a file for reading or writing
2. Reading from the file and/or writing to the file
Closing the file .3

The built-in function open() is used to open a file, whether the file is a text file or a binary file. In order to read file example.txt, we must first open it:
infile = open('example.txt', 'r')

The function open() takes three string arguments: a file name and, optionally, a mode and an encoding; we will not discuss the encoding argument until Chapter 6. The file name is really the pathname (absolute or relative) of the file to be opened. In the last example, the file relative pathname is example.txt. Python will look for a file named example.txt in the current working directory (recall that this will be the folder containing the module that was last imported); if no such file exists, an exception occurs. For example:

```
>>> infile = open('sample.txt')
Traceback (most recent call last):
File "<pyshell#339>", line 1, in <module> infile = open('sample.txt')
IOError: [Errno 2] No such file or directory: 'sample.txt'
```

The file name could also be the absolute path of the file such as, for example
/Users/lperkovic/example.txt
on a UNIX box or

C:/Users/sfaisal/example.txt on a Windows machine.

The mode is a string that specifies how we will interact with the opened file. In function call open('example.txt', 'r'), the mode 'r' indicates that the opened file will be read from; it also specifies that the file will be read from as a text file.

In general, the mode string may contain one of r, w, a, or r+ to indicate whether the file should be opened for reading, writing, appending, or reading and writing, respectively. If missing, the default is r. In addition, t or b could also appear in the mode string: indicates that the file is a text file, while b indicates it is a binary file. If neither is present, the file will be opened as a text file. So open('example.txt', 'r') is equivalent to open('example.txt', 'rt'), which is equivalent to open('example.txt'). .This is all summarized in Table below

Mode	Description
`r`	Reading mode (default)
`w`	Writing mode; if the file already exists, its content is wiped out
`a`	Append mode; writes are appended to the end of the file
`r+`	Reading and writing mode (beyond the scope of this book)
`t`	Text mode (default)
`b`	Binary mode

The difference between opening a file as a text or binary file is that binary files are treated as a sequence of bytes and are not decoded when read or encoded when written to. Text files, however, are treated as encoded files using some encoding.

The open() function returns an object of an Input or Output Stream type that supports methods to read and/or write characters. We refer to this object as a file object. Different modes will give us file objects of different file types. Depending on the mode, the file type will support all or some of the methods described in Table 4.5. The separate read methods are used to read the content of the file in different ways. We show the difference between the there on file example.txt whose content is:

```
The 3 lines in this file end with the new line character.

There is a blank line above this line.
```

We start by opening the file for reading as a text input stream:

```
>>> infile = open('example.txt')
```

With every opened file, the file system will associate a cursor that points to a character in the file. When the file is first opened, the cursor typically points to the beginning of the file (i.e., the first character of the file), as shown in Figure 4.8. When reading the file, characters that are read are the characters that start at the cursor; if we are writing to the file, then anything we write will be written starting at the cursor position.

:Student Name_______________________ :Roll No_____________ :Section____________

We now use the read() function to read just one character. The read() function will return the first character in the file as a (one-character) string.

```
>>> infile.read(1)
'T'
```

After the character 'T' is read, the cursor will move and point to the next character, which is 'h' (i.e., the first unread character). Let’s use the read() function again, but now to read five characters at a time. What is returned is a string of the five characters following the character 'T' we initially read:

```
>>> infile.read(5)
'he 3 '
```

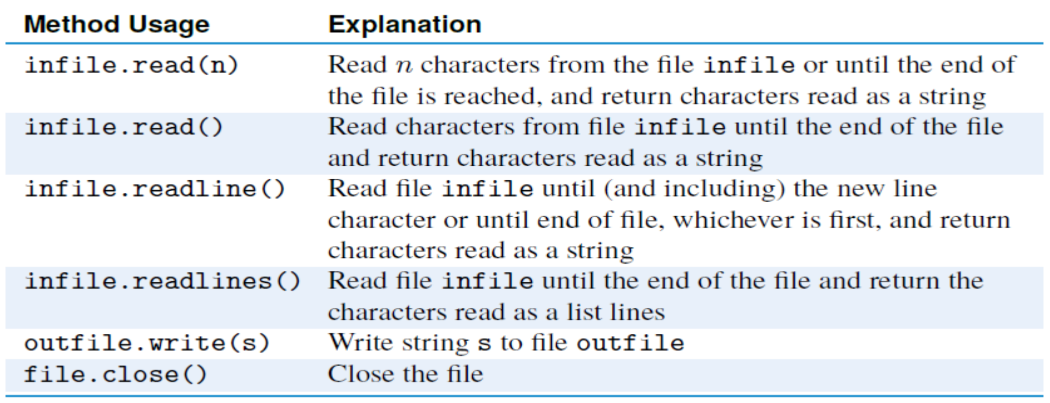
The function readline() will read characters from the file up to the end of the line (i.e., the new line character \n) or until the end of the file, whichever happens first. Note that in our case the last character of the string returned by readline() is the new line character:

```
>>> infile.readline()
```

'lines in this file end with the new line character.\n' The cursor now points to the beginning of the second line, as shown in Figure 4.8. Finally, we use the read() function without arguments to read the remainder of the file:

```
        >>> infile.read()
'\nThere is a blank line above this line.\n'
```

The cursor now points at the “End-Of-File” (EOF) character, which indicates the end of the .file



Method Usage	Explanation
`infile.read(n)`	Read n characters from the file **infile** or until the end of the file is reached, and return characters read as a string
`infile.read()`	Read characters from file **infile** until the end of the file and return characters read as a string
`infile.readline()`	Read file **infile** until (and including) the new line character or until end of file, whichever is first, and return characters read as a string
`infile.readlines()`	Read file **infile** until the end of the file and return the characters read as a list lines
`outfile.write(s)`	Write string **s** to file **outfile**
`file.close()`	Close the file

To close the opened file that infile refers to, you just do:
infile.close()

Closing a file releases the file system resources that keep track of information about the opened file (i.e., the cursor position information).

_ :Student Name_________________________ _ :Roll No_______________ _ :Section_____________

1.2 Patterns for Reading a Text File

Depending on what you need to do with a file, there are several ways to access the file content and prepare it for processing. We describe several patterns to open a file for reading and read the content of the file. We will use the file example.txt again to illustrate the patterns:

```
The 3 lines in this file end with the new line character.

There is a blank line above this line.
```

One way to access the text file content is to read the content of the file into a string object. This pattern is useful when the file is not too large and string operations will be used to process the file content. For example, this pattern can be used to search the file content or to replace every occurrence of a substring with another.

We illustrate this pattern by implementing function numChars(), which takes the name of a file as input and returns the number of characters in the file. We use the read() function to read the file content into a string:

```python
def numChars(filename):
    'returns the number of characters in file filename'
    infile = open(filename, 'r')
    content = infile.read()
    infile.close()

    return len(content)
```

When we run this function on our example file, we obtain:

```
>>> numChars('example.txt')
98
```

Write function stringCount() that takes two string inputs—a file name and a target string—and returns the number of occurrences of the target string in the file.

```
>>> stringCount('example.txt', 'line')
4
```

The file reading pattern we discuss next is useful when we need to process the words of a file. To access the words of a file, we can read the file content into a string and use the string split() function, in its default form, to split the content into a list of words. (So, our definition of a word in this example is just a contiguous sequence of nonblank characters.)

We illustrate this pattern on the next function, which returns the number of words in a file. It also prints the list of words, so we can see the list of words.

```
def numWords(filename):
    'returns the number of words in file filename'
    infile = open(filename, 'r')
    content = infile.read()       # read the file into a string
    infile.close()

    wordList = content.split()    # split file into list of words
    print(wordList)               # print list of words too
    return len(wordList)
```

Shown is the output when the function is run on our example file:

```
>>> numWords('example.txt')
['The', '3', 'lines', 'in', 'this', 'file', 'end', 'with',
 'the', 'new', 'line', 'character.', 'There', 'is', 'a',
 'blank', 'line', 'above', 'this', 'line.']
20
```

In function numWords(), the words in the list may include punctuation symbols, such as the period in 'line.'. It would be nice if we removed punctuation symbols before splitting the content into words. Doing so is the aim of the next problem.

Example:1

Write function words() that takes one input argument—a file name—and returns the list of actual words (without punctuation symbols!,.:;?) in the file.

```
>>> words('example.txt')
['The', '3', 'lines', 'in', 'this', 'file', 'end', 'with',
 'the', 'new', 'line', 'character', 'There', 'is', 'a',
 'blank', 'line', 'above', 'this', 'line']
```

Sometimes a text file needs to be processed line by line. This is done, for example, when searching a web server log file for records containing a suspicious IP address. A log file is a file in which every line is a record of some transaction (e.g., the processing of a web page request by a web server). In this third pattern, the readlines() function is used to obtain the content of the file as a list of lines. We illustrate the pattern on a simple function that counts the number of lines in a file by returning the length of this list. It also will print the list of lines so we can see what the list looks like:

```
def numLines(filename):
    'returns the number of lines in file filename'
    infile = open(filename, 'r')   # open the file and read it
    lineList = infile.readlines()  # into a list of lines
    infile.close()

    print(lineList)                # print list of lines
    return len(lineList)
```

is included Let's test the function on our example file. Note that the new line character \n :in each line

```
>>> numLines('example.txt')
['The 3 lines in this file end with the new line character.\n',
 '\n', 'There is a blank line above this line.\n']
3
```

All file processing patterns we have seen so far read the whole file content into a string or a a ,list of strings (lines). This approach is OK if the file is not too large. If the file is large better approach would be to process the file line by line; that way we avoid having the whole file in main memory. Python supports iteration over lines of a file object.We use this approach to print each line of the example file:

```
>>> infile = open('example.txt')
>>> for line in infile:
        print(line,end='')

The 3 lines in this file end with the new line character.

There is a blank line above this line.
```

the In every iteration of the for loop, the variable line will refer to the next line of the file. In first iteration, variable line refers to the line 'The three lines in ...'; in the second, it refers to .'... and in the final iteration, it refers to 'There is a blank ;'n\'
Thus, at any point in time, only one line of the file needs to be kept in memory

Example:2
Implement function myGrep() that takes as input two strings, a file name and a target string, .and prints every line of the file that contains the target string as a substring

```
>>> myGrep('example.txt', 'line')
The 3 lines in this file end with the new line character.
There is a blank line above this line.
```

_ :Student Name_________________________ _ :Roll No_______________ _ :Section_____________

1. 3 Writing to a Text File

In order to write to a text file, the file must be opened for writing:

```
>>> outfile = open('test.txt', 'w')
```

If there is no file test.txt in the current working directory, the open() function will create it. If a file text.txt exists, its content will be erased. In both cases, the cursor will point to the beginning of the (empty) file. (If we wanted to add more content to the (existing) file, we would use the mode 'a' instead of 'w'.)

Once a file is opened for writing, function write() is used to write strings to it. It will write the string starting at the cursor position. Let's start with a one-character string:

```
>>> outfile.write('T')
1
```

The value returned is the number of characters written to the file. The cursor now points to the position after T, and the next write will be done starting at that point.

```
>>> outfile.write('his is the first line.')
22
```

In this write, 22 characters are written to the first line of the file, right after T. The cursor will now point to the position after the period.

```
>>> outfile.write(' Still the first line...\n')
25
```

Everything written up until the new line character is written in the same line.With the '\n' character written, what follows will go into the second line:

```
>>> outfile.write('Now we are in the second line.\n')
31
```

The \n escape sequence indicates that we are done with the second line and will write the third line next. To write something other than a string, it needs to be converted to a string first:

```
>>> outfile.write('Non string value like '+str(5)+' must be
('converted first.\n
49
```

Here is where the string format() function is helpful. To illustrate the benefit of using string formatting, we print an exact copy of the previous line using string formatting:

```
>>> outfile.write('Non string value like {} must be converted
first.\n'.format(5))
49
```

Just as for reading, we must close the file after we are done writing:

```
>>> outfile.close()
```

The file test.txt will be saved in the current working directory and will have this content:

```
1 This is the first line. Still the first line...
2 Now we are in the second line.
3 Non string value like 5 must be converted first.
4 Non string value like 5 must be converted first.
```

_ :Student Name________________________ _ :Roll No______________ _ :Section____________

Programming Exercise

1. Write a function stats() that takes one input argument: the name of a text file. The function should print, on the screen, the number of lines, words, and characters in the file; your function should open the file only once.

```
>>> stats('example.txt')
line count: 3
word count: 20
character count: 98
```

2. Implement function distribution() that takes as input the name of a file (as a string). This one-line file will contain letter grades separated by blanks. Your function should print the distribution of grades, as shown.

```
>>> distribution('grades.txt')
6 students got A
2 students got A-
3 students got B+
2 students got B
2 students got B-
4 students got C
1 student got C-
2 students got F
```

3. Implement function duplicate() that takes as input the name (a string) of a file in the current directory and returns True if the file contains duplicate words and False otherwise.

```
>>> duplicate('Duplicates.txt')
True
>>> duplicate('noDuplicates.txt')
False
```

4. The function abc() takes the name of a file (a string) as input. The function should open the file, read it, and then write it into file abc.txt with this modification: Every occurrence of a four-letter word in the file should be replaced with string 'xxxx'.

```
>>> abc('example.txt')
```

Note that this function produces no output, but it does create file abc.txt in the current folder