6...

File I/O Handling and Exception Handling

Chapter Outcomes...

- Write Python code for the given reading values from keyboard.
- Read data from the given file.
- Write the given data to a file.
- Handle the given exceptions through Python program.

Learning Objectives...

- To understand File, I/O and Exception
- To study I/O Operations like Reading Input, Printing Output etc.,
- To learn File Handling Concepts such as Opening, Reading, Writing, Renaming, Deleting, Accessing File Contents etc.
- To study Directories in Python, File and Directory related Standard Functions
- To understand Exception Handling in Python

6.0 INTRODUCTION

- A file is a collection of related data that acts as a container of storage as data permanently. The file processing refers to a process in which a program processes and accesses data stored in files.
- A file is a computer resource used for recording data in a computer storage device. The processing on a file is performed using read/write operations performed by programs.
- Python supports file handling and allows users to handle files i.e., to read and write files, along with many other file handling options, to operate on files.
- Python programming provides modules with functions that enable us to manipulate text files and binary files. Python allows us to create files, update their contents and also delete files.
- A text file is a file that stores information in the term of a sequence of characters (textual information), while a binary file stops stores data in the form of bits (0s and 1s) and used to store information in the form as text, images, audios, videos etc.

6.1 I/O OPERATIONS (READING KEYBOARD INPUT AND PRINTING TO SCREEN)

- In any programming language an interface plays a very important role. It takes data from the user (input) and displays the output.
- One of the essential operations performed in Python language is to provide input values to the program and output the data produced by the program to a standard output device (monitor).

- The output generated is always dependent on the input values that have been provided to the program. The input can be provided to the program statically and dynamically.
- In static input, the raw data does not change in every run of the program. While in dynamic input, the raw data has a tendency to change in every run of the program.
- Python language has predefined functions for reading input and displaying output on the screen, Input can also be provided directly in the program by assigning the values to the variables. Python language provides numerous built in functions that are readily available to us at Python prompt.
- Some of the functions like input() and print() are widely used for standard Input and Output (I/O) operations, respectively.

1. Output (Printing to Screen):

- The function print() is used to output data to the standard output devices i.e., monitor/screen. The output can redirect or store on to a file also.
- The message can be a string, or any other object, the object will be converted into a string before written to the screen.

Syntax: `print(object(s), separator=separator, end=end, file=file, flush=flush)`

Parameter Values:

(i) object(s): Any object, and as many as you like. Will be converted to string before printed.

(ii) sep='separator': Optional. Specify how to separate the objects, if there is more than one. Default is ''.

(iii) end='end': Optional. Specify what to print at the end. Default is '\n' (line feed).

(iv) file: Optional. An object with a write method. Default is sys.stdout.

(v) flush: Optional. A Boolean, specifying if the output is flushed (True) or buffered (False). Default is False.

Example: For output using print().

```
>>> print("Hello", "how are you?", sep=" ---")
Hello ---how are you?
>>> print(10,20,30,sep='-')
10-20-30
```

- To make the output more attractive formatting is used. This can be done by using the str.format() method.

Example: For output using format().

```
>>> a=10
>>> b=20
>>> print('Value of a is {} and b is {}'.format(a,b))
Value of a is 10 and b is 20
>>> print('I will visit {0} and {1} in summer'.format('Jammu','Kashmir')
I will visit Jammu and Kashmir in summer
>>>
```

- Alike old sprint() style used in C programming language, we can format the output in Python language also. The % operator is used to accomplish this.

Example: For output with %.

```
>>> x=12.3456789
>>> print('The value of x=%3.2f'%x)
The value of x=12.35
>>> print('The value of x=%3.4f'%x)
The value of x=12.3457
```

- The various format symbols available in Python programming are:

Sr. No.	Format Symbol	Conversion
1.	%c	Character.
2.	%s	String conversion via str() prior to formatting.
3.	%i	Signed decimal integer.
4.	%d	Signed decimal integer.
5.	%u	Unsigned decimal integer.
6.	%o	Octal integer.
7.	%x	Hexadecimal integer (lowercase letters).
8.	%X	Hexadecimal integer (UPPERcase letters).
9.	%e	Exponential notation (with lowercase 'e').
10.	%E	Exponential notation (with UPPERcase 'E').
11.	%f	Floating point real number.
12.	%g	The shorter of %f and %e.
13.	%G	The shorter of %f and %E.

2. Input (Reading Keyboard Input):

- Python provides two built-in functions to read a line of text from standard input, which by default comes from the keyboard.

(i) input(prompt): The input() function allows user input.

Syntax: `input(prompt)`

Where, Prompt is a String, representing a default message before the input.

Example: For input (prompt) method.

```
x = input('Enter your name:')
print('Hello, ' + x)
```

Output:

```
Enter your name:vijay
Hello, vijay
```

(ii) input(): The function input() always evaluates and return same type data.

Syntax: `x=input()`

- If input value is integer type then its return integer value. If input value is string type then its return string value.

Example: For reading input from keyboard.

```
>>> x=input()
5
>>> type(x)
<class 'int'>
>>> x=input()
5.6
>>> type(x)
<class 'float'>
>>> x=int(input())
5
>>> type(x)
<class 'int'>
```

```
>>> x=float(input())
2.5
>>> type(x)
<class 'float'>
```

6.2 FILES

- File is a named location on disk to store related information. It is used to permanently store data in a non-volatile memory (e.g. hard disk).
- Since, Random Access Memory (RAM) is volatile which loses its data when computer is turned OFF, we use files for future use of the data.
- Files are divided into following two categories:
  1. **Text Files:** Text files are simple texts in human readable format. A text file is structured as sequence of lines of text.
  2. **BinaryFiles:** Binary files have binary data which is understood by the computer.
- When we want to read from or write to a file we need to open it first. When we are done, it needs to be closed, so that resources that are tied with the file are freed.
- Hence, in Python a file operation takes place in the following order:
  - Open a file.
  - Read or write (perform operation).
  - Close the file.

6.2.1 Opening File in Different Modes

- All files in Python are required to be open be fore some operation (read or write) can be performed on the file. In Python programming while opening a file, file object is created, and by using this file object we can perform different set as operations on the opened file.
- Python has a built-in function open() to open a file. This function returns a file object also called a handle, as it is used to read or modify the file accordingly.

Syntax: `file object = open(file_name [, access_mode][, buffering])`

Parameters:

file_name: The file_name argument is a string value that contains the name of the file that we want to access.

access_mode: The access_mode determines the mode in which the file has to be opened, i.e., read, write, append, etc. This is optional parameter and the default file access mode is read (r).

buffering: If the buffering value is set to 0, no buffering takes place. If the buffering value is 1, line buffering is performed while accessing a file. If we specify the buffering value as an integer greater than 1, then buffering action is performed with the indicated buffer size. If negative, the buffer size is the system default (default behavior).

If the path is in current working directory, we can just provide the filename, just like in the following examples:

```
>>> file=open("sample.txt")
>>> file.read()
'Hello I am there\n'   # content of file
>>>
```

- If still we are getting “no such file or directory” error then use following command to confirm the proper file name and extension:

```
>>> import os
>>> os.listdir()
['DLLs', 'Doc', 'etc', 'include', 'Lib', 'libs', 'LICENSE.txt', 'mypkg', 'NEWS.txt',
'p1.py',  'p2.py',  'python.exe',  'python3.dll',  'python37.dll',  'pythonw.exe',
```

```
'sample.txt', 'Scripts', 'share', 'tcl', 'test.py', 'Tools', 'vcruntime140.dll',
'_ _pycache_ _']
>>>
```

- If the file resides in a directory other than that, you have to provide the full path with the file name:

```
>>> file=open("D:\\files\\sample.txt")
>>> file.read()
'Hello I am there\n'
>>>
```

- We can specify the mode while opening a file. In mode, we specify whether we want to read 'r', write 'w' or append 'a' to the file. We also specify if we want to open the file in text mode or binary mode.
- The default is reading in text mode. In this mode, we get strings when reading from the file. The binary mode returns bytes and this is the mode to be used when dealing with non-text files like image or exe files.

6.2.2 Different Modes of Opening File

- Like, C, C++, and Java, a file in Python programming can be opened in various modes depending upon the purpose. For that, the programmer needs to specify the mode whether read 'r', write 'w', or append 'a' mode.
- Apart from this, two other modes exist, which specify to open the file in text mode or binary mode.
  1. The **text mode** returns strings while reading from the file. The default is reading in text mode.
  2. The **binary mode** returns bytes and this is the mode to be used when dealing with non-text files like image or executable files.
- The text and binary modes are used in conjunction with the r, w, and a modes. The list of all the modes used in Python are given in following table:

Sr. No.	Mode	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.	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.
7.	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.
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.	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.
11.	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.
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.
13.	t	Opens in text mode (default).
14.	b	Opens in binary mode.
15.	+	Opens a file for updating (reading and writing).

Example: For different modes of opening a file.

```
f = open("test.txt")        # opening in r mode (reading only)
f = open("test.txt",'w')    # opens in w mode (writing mode)
f = open("img.bmp",'rb+')   # read and write in binary mode
```

6.2.3 Accessing File Contents using Standard Library Function

- Once, a file is opened and we have one file object, we can get various information related to that file. Here, is a list of all attributes related to file object:

Sr. No.	Attribute	Description	Example

1.	file.closed	Returns true if file is closed, false otherwise.	>>> f=open("abc.txt")
			>>> f.closed
			False

2.	file.mode	Returns access mode with which file was opened.	>>> f=open("abc.txt","w")
			>>> f.mode
			'w'

3.	file.name	Returns name of the file.	>>> f=open("abc.txt","w") >>>f.name 'abc.txt'
4.	file.encoding	The encoding of the file.	>>> f=open("abc.txt","w") >>>f.encoding cp1252

Example: For file object attribute.

```
f=open("sample.txt","r")
print(f.name)
print(f.mode)
print(f.encoding)
f.close()
print(f.closed)
```

Output:

```
sample.txt
r
cp1252
True
```

6.2.4 Closing File

- When we are done with operations to the file, we need to properly close the file.

- Closing a file will free up the resources that were tied with the file and is done using Python close() method.

Syntax: `fileObject.close()`

Example: For closing a file.

```
f=open("sample.txt")
print("Name of the file: ",f.name)
f.close()
```

Output:

```
Name of the file:  sample.txt
```

6.2.5 Writing Data to File

- The write() method writes any string to an open file. In order to write into a file in Python, we need to open it in write 'w', append 'a' or exclusive creation 'x' mode.
- The write () method writes the contents onto the file. It takes only one parameter and returns the number of characters writing to the file. The write () method is called by the file object onto which we want to write the data. We need to be careful with the 'w' mode as it will overwrite into the file if it already exists. All previous data are erased.
- We can use three methods to write to a file in Python namely, write(string) (for text), write(byte_string) (for binary) and writelines (list).

1. write(string) Method:

- The write(string) method writes the contents of string to the file, returning the number of characters written.

```
>>> f.write('This is a test\n')
15
```

Example: For write(string) method.

```
f=open("sample.txt")
print("**content of file1**")
print(f.read())
f=open("sample.txt","w")
f.write("first line\n")
f.write("second line\n")
f.write("third line\n")
f.close()
f=open("sample.txt","r")
print("**content of file1**")
print(f.read())
```

Output:

```
**content of file1**
Hello,I am There
**content of file1**
first line
second line
third line
```

2. writelines(list) Method:

- It writes 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.

Example: For writelines() method.

```
fruits=["Orange\n","Banana\n","Apple\n"]
f=open("sample.txt",mode="w+",encoding="utf-8")
f.writelines(fruits)
f.close()
f=open("sample.txt","r")
print(f.read())
```

Output:

```
Orange
Banana
Apple
>>>
```

6.2.6 Reading Data From File

- To read a file in Python, we must open the file in reading mode (r or r+). The read () method in Python programming reds the contents of the file. It returns the characters read from the file. The read () is also called by the file object from which we want to read the data.
- There are following three methods available for reading data purpose:

1. read([n]) Method:

- The read() method just outputs the entire file if number of bytes are not given in the argument. If we execute read(3), we will get back the first three characters of the file. Note: [n] means optional.

Example: for read( ) method.

```
f=open("sample.txt","r")
print(f.read(5))        # read first 5 data
print(f.read(5))        # read next five data
print(f.read())         # read rest of the file
print(f.read())
```

Output:

```
first
line
second line
third line
```

2. readline([n]) Method:

- The readline() method just output the entire line whereas readline(n) outputs at most n bytes of a single line of a file. It does not read more than one line. Once, the end of file is reached, we get empty string on further reading.

Example: For readline ( ) method.

```
f=open("sample.txt","r")
print(f.readline())     # read first line followed by\n
print(f.readline(3))
print(f.readline(5))
print(f.readline())
print(f.readline())
```

Output:

```
first line
sec
ond l
ine
```

```
third line
```

3. **readlines():** This method maintains a list of each line in the file.

Example: For roadlines() method.

```
f=open("sample.txt","r")
print(f.readlines())
```

Output:

```
['first line\n', 'second line\n', 'third line\n']
```

6.2.7 File Position

- We can change our current file cursor (position) using the seek() method. Similarly, the tell() method returns our current position (in number of bytes).
- To change the file object's position use f.seek(offset, reference_point). The position is computed from adding offset to a reference point.
- A reference_point value of 0 measures from the beginning of the file, 1 uses the current file position, and 2 uses the end of the file as the reference point.
- The reference_point can be omitted and defaults to 0, using the beginning of the file as the reference point. The reference points are 0 (the beginning of the file and is default), 1 (the current position of file) and 2 (the end of the file).
- The f.tell() returns an integer giving the file object's current position in the file represented as number of bytes from the beginning of the file when in binary mode and an opaque number when in text mode.
- In other words, the tell ( ) is used to find the current position of the file pointer in the file while the seek ( ) used to more the file pointer to the particular position.

Example 1: For file position.

```
f=open("sample.txt","r")
print(f.tell())
print(f.read())
print(f.tell())
print(f.read())   # print blank line
print(f.seek(0))
print(f.read())
```

Output:

```
0
first line
second line
third line

37
0
first line
second line
third line
>>>
```

Example 2:

```
fruits=["Orange\n","Banana\n","Apple\n"]
f=open("sample.txt",mode="w+",encoding="utf-8")
for fru in fruits:
    f.write(fru)
```

```
print("Tell the byte at which the file cursor is:",f.tell())
f.seek(0)
for line in f:
    print(line)
```

Output:

```
Tell the byte at which the file cursor is: 23
Orange
Banana
Apple
>>>
```

Other Methods of File Object:

Sr. No.	Method	Function
1.	`file.fileno()`	It returns the integer file descriptor.
2.	`file.isatty()`	It returns True if file is connected with tty (-like) device, False otherwise.
3.	`readable()`	Returns True/False whether file is readable.
4.	`writable()`	Returns True/False whether file is writable.
5.	`truncate([size)`	Truncate the file, up to specified bytes.
6.	`file.flush()`	It flushes the internal buffer memory.
7.	`file.next()`	It returns the next line from the file.

Example: For file object methods.

```
f = open("sample.txt","w+")
f.write("Line one\nLine two\nLine three")
f.seek(0)
print(f.read())
print("Is readable:",f.readable())
print("Is writeable:",f.writable())
print("File no:",f.fileno())
print("Is connected to tty-like device:",f.isatty())
f.truncate(5)
f.flush()
f.close()
```

Output:

```
Line one
Line two
Line three
Is readable: True
Is writeable: True
File no: 3
Is connected to tty-like device: False
>>>
```

Handling Files through OS Module:

- The OS module of Python allows us to perform Operating System (OS) dependent operations such as making a folder, listing contents of a folder, know about a process, end a process etc.

- It has methods to view environment variables of the operating system on which Python is working on and many more.

Sr. No.	Method	Function
1.	os.getcwd()	Show current working directory.
2.	os.path.getsize()	Show file size in bytes of file passed in parameter.
3.	os.path.isfile()	Is passed parameter a file.
4.	os.path.isdir()	Is passed parameter a folder.
5.	os.chdir()	Change directory/folder.
6.	os.rename(current,new)	Rename a file.
7.	os.remove(file_name)	Delete a file.
8.	os.makedirs()	Create a new folder.

Example: For handling files through OS module.

```
import os
os.getcwd()
print("***Contents of Present working directory***\n ", os.listdir())
print(os.path.isfile("sample.txt"))
print(os.path.isdir("sample.txt"))
```

```
***Contents of Present working directory***
 ['DLLs', 'Doc', 'etc', 'file1.txt', 'include', 'Lib', 'libs', 'LICENSE.txt',
'mypkg', 'NEWS.txt', 'p1.py', 'p2.py', 'python.exe', 'python3.dll', 'python37.dll',
'pythonw.exe', 'sample.txt', 'Scripts', 'share', 'tcl', 'test.py', 'Tools',
'vcruntime140.dll', '_ _pycache_ _']
True
False
>>>
```

6.2.8 Renaming a File

- Renaming a file in Python is done with the help of the rename() method. To rename a file in Python, the OS module needs to be imported.
- The rename() method takes two arguments, the current filename and the new filename.

Syntax: `os.rename(current_file_name, new_file_name)`

Example: For remaining files.

```
import os
print("***Contents of Present working directory***\n ",os.listdir())
os.rename("sample.txt","sample1.txt")
print("***Contents of Present working directory after rename***\n ",os.listdir())
```

Output:

```
***Contents of Present working directory***
  ['DLLs', 'Doc', 'etc', 'file1.txt', 'include', 'Lib', 'libs', 'LICENSE.txt',
'mypkg', 'NEWS.txt', 'p1.py', 'p2.py', 'python.exe', 'python3.dll', 'python37.dll',
'pythonw.exe', 'sample.txt', 'Scripts', 'share', 'tcl', 'test.py', 'Tools',
'vcruntime140.dll', '_ _pycache_ _']
***Contents of Present working directory after rename***
  ['DLLs', 'Doc', 'etc', 'file1.txt', 'include', 'Lib', 'libs', 'LICENSE.txt',
'mypkg', 'NEWS.txt', 'p1.py', 'p2.py', 'python.exe', 'python3.dll', 'python37.dll',
```

```
'pythonw.exe', 'sample1.txt', 'Scripts', 'share', 'tcl', 'test.py', 'Tools',
'vcruntime140.dll', '_ _pycache_ _']
>>>
```

6.2.9 Deleting a File

- We can use the remove() method to delete files by supplying the name of the file to be deleted as the argument.
- To remove a file, the OS module need to be imported. The remove ( ) in Python programming in used to remove the existing file with the file name.

Syntax: `os.remove(file_name)`

Example: For deleting files.

```
import os
print("***Contents of Present working directory***\n ",os.listdir())
os.remove("sample.txt")
print("***New Contents of Present working directory***\n ",os.listdir())
```

Output:

```
***Contents of Present working directory***
  ['DLLs', 'Doc', 'etc', 'file1.txt', 'include', 'Lib', 'libs', 'LICENSE.txt',
'mypkg', 'NEWS.txt', 'p1.py', 'p2.py', 'python.exe', 'python3.dll', 'python37.dll',
'pythonw.exe', 'sample.txt', 'Scripts', 'share', 'tcl', 'test.py', 'Tools',
'vcruntime140.dll', '_ _pycache_ _']
***New Contents of Present working directory***
  ['DLLs', 'Doc', 'etc', 'file1.txt', 'include', 'Lib', 'libs', 'LICENSE.txt',
'mypkg', 'NEWS.txt', 'p1.py', 'p2.py', 'python.exe', 'python3.dll', 'python37.dll',
'pythonw.exe', 'Scripts', 'share', 'tcl', 'test.py', 'Tools', 'vcruntime140.dll',
'_ _pycache_ _']
```

- The is file ( ) method in Python programming checks whether the file passed in the method exists or not. It returns true if the file exist otherwise it returns false.

6.2.10 Directories

- If there are a large number of files to handle in the Python program, we can arrange the code within different directories to make things more manageable.
- A directory or folder is a collection of files and sub directories. Python has the os module, which provides us with many useful methods to work with directories (and files as well).

6.2.10.1 Create 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.

Syntax: `os.mkdir("newdir")`

Example:

```
>>> import os
>>> os.mkdir("testdir")
```

6.2.10.2 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.

Syntax: `os.getcwd()`

Example:

```
>>> import os
```

```
>>> os.getcwd()
'C:\\Users\\ Meenakshi \\AppData\\Local\\Programs\\Python\\Python37-32'
```

6.2.10.3 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.

Syntax: `os.chdir("dirname")`

Example:

```
>>> import os
>>> os.getcwd()
'C:\\Users\\Meenakshi\\AppData\\Local\\Programs\\Python\\Python37'
>>> os.chdir("d:\IT")
>>> os.getcwd()
'd:\\IT'
>>>
```

6.2.10.4 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.

Example:

```
>>> os.listdir()
['DLLs', 'Doc', 'include', 'Lib', 'libs', 'LICENSE.txt', 'NEWS.txt', 'python.exe',
'python3.dll', 'python37.dll', 'pythonw.exe', 'Scripts', 'tcl', 'test.py', 'testdir',
'Tools', 'vcruntime140.dll']
```

6.2.10.5 Removing Directory

- The rmdir() method is used to remove directories in the current directory.

Syntax: `os.rmdir("dirname")`

Example:

```
>>> os.listdir()
['DLLs', 'Doc', 'etc', 'file1.txt', 'include', 'Lib', 'libs', 'LICENSE.txt', 'mydir',
'mypkg', 'NEWS.txt', 'p1.py', 'p2.py', 'python.exe', 'python3.dll', 'python37.dll',
'pythonw.exe', 'sample.txt', 'Scripts', 'share', 'tcl', 'test.py', 'Tools',
'vcruntime140.dll', '__pycache__']
>>> os.rmdir("mydir")
>>> os.listdir()
['DLLs', 'Doc', 'etc', 'file1.txt', 'include', 'Lib', 'libs', 'LICENSE.txt', 'mypkg',
'NEWS.txt', 'p1.py', 'p2.py', 'python.exe', 'python3.dll', 'python37.dll',
'pythonw.exe', 'sample.txt', 'Scripts', 'share', 'tcl', 'test.py', 'Tools',
'vcruntime140.dll', '__pycache__']
>>>
```

- If the directory is not empty then we will get the "The directory is not empty" error. To remove a directory, first remove all the files inside it using os.remove() method.

```
>>> os.chdir("C:\\Users\\Meenakshi\\AppData\\Local\\Programs\\Python\\Python37")
>>> os.rmdir("mydir1")
Traceback (most recent call last):
  File "<pyshell#32>", line 1, in <module>
```

```
  os.rmdir("mydir1")
OSError: [WinError 145] The directory is not empty: 'mydir1'
>>>
os.chdir("C:\\Users\\Meenakshi\\AppData\\Local\\Programs\\Python\\Python37\mydir1")
>>> os.listdir()
['sample.txt']
>>> os.remove("sample.txt")
>>> os.listdir()
[]
>>> os.chdir("C:\\Users\\Meenakshi\\AppData\\Local\\Programs\\Python\\Python37")
>>> os.rmdir("mydir1")
>>>
```

- In order to remove a non-empty directory we can use the rmtree() method inside the shutil module.

```
>>> import shutil
>>> shutil.rmtree('test')
```

6.2.11 File and Directory related Standard functions

- In this section we will study various file and directory related standard functions.

1. File Related Standard Functions:

Sr. No.	Function	Description	Examples
1.	<file.close()>	Close the file. We need to reopen it for further access.	f.close()
2.	<file.flush()>	Flush the internal buffer. It's same as the <stdio>'s <fflush()> function.	f.flush() print(f.read()) f.close()
3.	<file.fileno()>	Returns an integer file descriptor.	print(f.fileno()) f.close()
4.	<file.isatty()>	It returns true if file has a <tty> attached to it.	print(f.isatty()) f.close()
5.	<file.next()>	Returns the next line from the last offset.	try: whilef.next(): print(f.next()) except: f.close()
6.	<file.read()>	This function reads the entire file and returns a string.	lines =f.read() f.write(lines) f.close()
7.	<file.read(size)>	Reads the given no. of bytes. It may read less if EOF is hit.	text =f.read(10) print(text) f.close()
8.	file.readline()	Reads a single line and returns it as a string.	text =f.readline() print(text) f.close()
9.	<file.readline(size)>	It'll read an entire line (trailing with a	text =f.readline(20)

		new line char) from the file.	print(text) f.close()
10.	file.readlines()	Reads the content of a file line by line and returns them as a list of strings.	lines =f.readlines() f.writelines(lines) f.close()
11.	<file.readlines(size_hint)>	It calls the <readline()> to read until EOF. It returns a list of lines read from the file. If you pass <size_hint>, then it reads lines equalling the <size_hint> bytes.	text =f.readlines(25) print(text) f.close()
12.	<file.seek(offset[, from])>	Sets the file's current position.	position =f.seek(0,0); print(position) f.close()

13.	<file.tell()>	Returns the file's current position.	lines =f.read(10)
			#tell()
			print(f.tell())
			f.close()

14.	<file.truncate(size)>	Truncates the file's size. If the optional size argument is present, the file is truncated to (at most) that size.	f.truncate(10) f.close()
15.	<file.write(string)>	It writes a string to the file. And it doesn't return any value.	line ='Welcome Geeks\n' f.write(line) f.close()

16.	<file.writelines(sequence)>	Writes a sequence of strings to the file. The sequence is possibly an iterable object producing strings, typically a list of strings.	lines =f.readlines()
			#writelines()
			f.writelines(lines)
			f.close()

2. Directory Related Standard Functions:

Sr. No.	Function	Description	Examples
1.	os.getcwd()	Show current working directory.	import os os.getcwd()
2.	os.path.getsize()	Show file size in bytes of file passed in parameter.	size =os.path.getsize("sample.txt")
3.	os.path.isfile()	Is passed parameter a file.	print(os.path.isfile("sample.txt"))
4.	os.path.isdir()	Is passed parameter a folder.	print(os.path.isdir("sample.txt"))
5.	os.listdir()	Returns a list of all files and folders of present working. directory	print("***Contents of Present working directory***\n ",os.listdir())
6.	os.listdir(path)	Return a list containing the names of the entries in the directory given by path.	print("***Contents of given directory***\n ",os.listdir("testdir"))

7.	os.rename(current,new)	Rename a file.	os.rename("sample.txt","sample1.txt")
8.	os.remove(file_name)	Delete a file.	os.remove("sample.txt")
9.	os.mkdir()	Creates a single subdirectory.	os.mkdir("testdir")
10.	os.chdir(path)	Change the current working directory to path.	os.chdir("d:\IT")

Additional Programs:

1. Program to create a simple file and write some content in it.

```
print("Enter 'x' for exit.");
filename = input("Enter file name to create and write content: ");
if filename == 'x':
exit();
else:
    c = open(filename, "w");
print("\nThe file,",filename,"created successfully!");
print("Enter sentences to write on the file: ");
    sent1 = input();
c.write(sent1);
c.close();
print("\nContent successfully placed inside the file.!!");
```

Output:

```
Enter 'x' for exit.
Enter file name to create and write content: file1

The file, file1 created successfully!
Enter sentences to write on the file:
good morning

Content successfully placed inside the file.!!
>>>
```

2. Program to open a file in write mode and append some content at the end of a file.

```
print("Enter 'x' for exit.");
filename = input("Enter file name to append content: ");
if filename == 'x':
    exit();
else:
    c=open(filename, "a+");
print("Enter sentences to append on the file: ");
sent1=input();
c.write("\n");
c.write(sent1);
c.close();
print("\nContent appended to file.!!");
```

Output:

```
Enter 'x' for exit.
Enter file name to append content: file1
Enter sentences to append on the file:
good afternoon
Content appended to file.!!
```

3. Program to open a file in read mode and print number of occurrences of characters 'a'.

```
fname = input("Enter file name: ")
l=input("Enter letter to be searched:")
k = 0
with open(fname, 'r') as f:
    for line in f:
        words = line.split()
        for i in words:
            for letter in i:
                if(letter==l):
                    k=k+1
print("Occurrences of the letter:")
print(k)
```

Output:

```
Enter file name: file1
Enter letter to be searched:o
Occurrences of the letter:
7
>>>
```

6.3 EXCEPTION HANDILNG

- When we executes a Python program, there may be a few uncertain conditions which occur, known as errors. Errors also referred to as bugs that are incorrect or in accurate action that may cause the problems in the running of the program or may interrupt the execution of the programmer may provide wrong result of to r execution the program.

  There are three type of error occurs:

  1. **Compile Time error:** Occurs at the time of compilation, include due to the violation of syntax rules like missing of a colon (:) error occur.
  2. **Rein Time Errors:** Occurs during the runtime of a program, explain include error occur due to wrong input submitted to program by user.
  3. **Logical Errors:** Occurs due to wrong logic written in the program.
- Errors occurs at runtime are known as exception. Errors detected during execution of program.
- Python provides a feature (Exception handling) for handling any unreported errors in program.
- When exception occurs in the program, execution gets terminated. In such cases we get system generated error message.
- By handling the exceptions, we can provide a meaningful message to the user about the problem rather than system generated error message, which may not be understandable to the user.
- Exception can be either built-in exceptions or user defined exceptions.
- The interpreter or built-in functions can generate the built-in exceptions while user defined exceptions are custom exceptions created by the user.

Example: For exceptions.

```
>>> a=3
```

```
>>> if (a<5)
SyntaxError: invalid syntax
>>> 5/0
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    5/0
ZeroDivisionError: division by zero
```

6.3.1 Introduction

- An exception is also called ad runtime error that can halt the execution of the program.
- An exception is an error that happens/occurs during execution of a program. When that error occurs, Python generate an exception that can be handled, which avoids the normal flow of the program's instructions.
- Errors detected during execution are called exceptions. An exception is an event (usually an error), which occurs during the execution of a program that disrupts the normal flow of execution as the program.
- In Python programming we can handle exceptions using try-except statement, try-finally statement and raise statement.
- Following table list all the standard exceptions available in Python programming language:

Sr. No.	Exception	Cause of Error
1.	ArithmeticError	Base class for all errors that occur for numeric calculation.
2.	AssertionError	Raised in case of failure of the assert statement.
3.	AttributeError	Raised in case of failure of attribute reference or assignment.
4.	Exception	Base class for all exceptions.
5.	EOFError	Raised when there is no input from either the raw_input() or input() function and the end of file is reached.
6.	EnvironmentError	Base class for all exceptions that occur outside the Python environment.
7.	FloatingPointError	Raised when a floating point calculation fails.
8.	ImportError	Raised when an import statement fails.
9.	IndexError	Raised when an index is not found in a sequence.
10.	IOError	Raised when an input/ output operation fails, such as the print statement or the open() function when trying to open a file that does not exist.
11.	IndentationError	Raised when indentation is not specified properly.
12.	KeyboardInterrupt	Raised when the user interrupts program execution, usually by pressing Ctrl+c.
13.	KeyError	Raised when the specified key is not found in the dictionary.
14.	LookupError	Base class for all lookup errors.
15.	NameError	Raised when an identifier is not found in the local or global namespace.
16.	NotImplementedError	Raised when an abstract method that needs to be implemented in an inherited class is not actually implemented.

17.	OverflowError	Raised when a calculation exceeds maximum limit for a numeric type.
18.	OSError	Raised for operating system-related errors.
19.	RuntimeError	Raised when a generated error does not fall into any category.
20.	StopIteration	Raised when the next() method of an iterator does not point to any object.
21.	SystemExit	Raised by the sys.exit() function.
22.	StandardError	Base class for all built-in exceptions except StopIteration and SystemExit.
23.	SyntaxError	Raised when there is an error in Python syntax.
24.	SystemError	Raised when the interpreter finds an internal problem, but when this error is encountered the Python interpreter does not exit.
25.	SystemExit	Raised when Python interpreter is quit by using the sys.exit() function. If not handled in the code, causes the interpreter to exit.
26.	TypeError	Raised when an operation or function is attempted that is invalid for the specified data type.
27.	UnboundLocalError	Raised when trying to access a local variable in a function or method but no value has been assigned to it.
28.	ValueError	Raised when the built-in function for a data type has the valid type of arguments, but the arguments have invalid values specified.
29.	ZeroDivisionError	Raised when division or modulo by zero takes place for all numeric types.

6.3.2 Exception Handling in Python Programming

- An exception is an event, which occurs during the execution of a program that disrupts the normal flow of the program's instructions.
- In general, when a Python script encounters a situation that it cannot cope with, it raises an exception. An exception is a Python object that represents an error.
- For example, if function A calls function B which in turn calls function C and an exception occurs in function C. If it is not handled in C, the exception passes to B and then to A. When a Python script raises an exception, it must either handle the exception immediately otherwise it terminates and quits.
- The exception handling is a process that provides a way to handle exceptions that occur at runtime.
- The exception handling is done by writing exception handlers in the program. The exception handlers are blocks that execute when some exception occurs at runtime. Exception handlers displays same message that represents information about the exception.

6.3.2.1 Try-except

- In Python, exceptions can be handled using a try statement. A try block consisting of one or more statements is used by programmers to partition code that might be affected by an exception.
- A critical operation which can raise exception is placed inside the try clause and the code that handles exception is written in except clause.

- The associated except blocks are used to handle any resulting exceptions thrown in the try block.That is we want the try block to succeed and if it does not succeed, we want to control to pass to the catch block.
- If any statement within the try block throws an exception, control immediately shifts to the catch block. If no exception is thrown in the try block, the catch block is skipped.
- There can be one or more except blocks. Multiple except blocks with different exception names can be chained together.
- The except blocks are evaluated from top to bottom in the code, but only one except block is executed for each exception that is thrown.
- The first except block that specifies the exact exception name of the thrown exception is executed. If no except block specifies a matching exception name then an except block that does not have an exception name is selected, if one is present in the code.
- For handling exception in Python, the exception handler block needs to be written which consists of set of statements that need to be executed according to raised exception. There are three blocks that are used in the exception handling process, namely, try, except and finally.
  1. **try Block:** A set of statements that may cause error during runtime are to be written in the try block.
  2. **except Block:** It is written to display the execution details to the user when certain exception occurs in the program. The except block executed only when a certain type as exception occurs in the execution of statements written in the try block.
  3. **finally Block:** This is the last block written while writing an exception handler in the program which indicates the set of statements that many use to clean up to resources used by the program.

Syntax:

```
try:
   D the operations here
   ......................
except Exception1:
   If there is Exception1, then execute this block.
except Exception2:
   If there is Exception2, then execute this block.
   ......................
else:
   If there is no exception then execute this block.
```

Example: For try-except clause/statement.

```
try:
   fh = open("testfile", "w")
   fh.write("This is my test file for exception handling!!")
except IOError:
   print ("Error: can\'t find file or read data")
else:
   print ("Written content in the file successfully")
   fh.close()
```

Example: For try statement.

```
n=10
m=0
try:
    n/m
```

```
except ZeroDivisionError:
    print("Divide by zero error")
else:
    print (n/m)
```

Output:

```
Divide by zero error
```

6.3.2.2 try-Except with No Exception

- We can use try-except clause with no exception. All types of exceptions that occur are caught by the try-except statement.
- However, because it catches all exceptions, the programmer cannot identify the root cause of a problem that may occur. Hence this type of programming approach is not considered good.

Syntax:

```
try:
   D the operations here
   ......................
except:
   If there is any exception, then execute this block.
   ......................
else:
   If there is no exception then execute this block.
```

Example: For try-except statement with no exception.

```
while True:
    try:
        a=int(input("Enter an integer: "))
        div=10/a
        break
    except:
        print("Error Occurred")
        print("Please enter valid value")
        print()
print("Division is: ",div)
```

Output:

```
Enter an integer: b
Error Occurred
Please enter valid value

Enter an integer: 2.5
Error Occurred
Please enter valid value

Enter an integer: 0
Error Occurred
Please enter valid value

Enter an integer: 5
Division is:  2.0
```

6.3.2.3 try...tinally

- The try statement in Python can have an optional finally clause. This clause is executed no matter what, and is generally used to release external resources.
- The statement written in finally clause will always be executed by the interpreter, whether the try statement raises an exception or not.
- A finally block is always executed before leaving the try statement, whether an exception is occurred or not. When an exception is occurred in try block and has not been handled by an except block, it is re-raised after the finally block has been executed.
- The finally clause is also executed "on the way out" when any other clause of the try statement is left via a break, continue or return statement.

Syntax:

```
try:
   D the operations here;
   ......................
   Due to any exception, this may be skipped.
finally:
   This would always be executed.
   ......................
```

Example: For try-finally.

```
try:
   fh = open("testfile", "w")
   fh.write("This is my test file for exception handling!!")
finally:
   print ("file is closing")
   fh.close()
```

Example: Program to check for ZeroDivisionError Exception.

```
x=int(input("Enter first value:"))
y=int(input("Enter second value:"))
try:
    result=x/y
except ZeroDivisionError:
    print("Division by Zero")
else:
    print("Result is:",result)
finally:
    print("Execute finally clause")
```

Output 1:

```
Enter first value:5
Enter second value:0
Division by Zero
Execute finally clause
```

Output 2:

```
Enter first value:10
Enter second value:5
Result is: 2.0
Execute finally clause
```

6.3.3 raise Statement

- We can raise an existing exception by using raise keyword. So, we just simply write raise keyword and then the name of the exception.
- The raise statement allows the programmer to force a specified exception to occur.
- For example: We can use raise to throw an exception if age is less than 18 condition occurs.

```
while True:
   try:
       age = int(input("Enter your age for election: "))
       if age < 18:
           raise Exception
       else:
           print("you are eligible for election")
           break
   except Exception:
       print("This value is too small, try again")
```

- Output:

```
Enter your age for election: 11
This value is too small, try again
Enter your age for election: 18
you are eligible for election
>>>
```

- The raise statement can be complemented with a custom exception as explained in next section.

6.3.4 User Defined Exception

- Python has many built-in exceptions which forces the program to output an error when something in it goes wrong. However, sometimes we may need to create custom exceptions that serves the purpose.
- Python allow programmers to create their own exception class. Exceptions should typically be derived from the Exception class, either directly or indirectly. Most of the built-in exceptions are also derived from Exception class.
- User can also create and raise his/her own exception known as user defined exception.
- In the following example, we create custom exception class AgeSmallException that is derived from the base class Exception.

Example 1: Raise a user defined exception if age is less than 18.

```
# define Python user-defined exceptions
class Error(Exception):
   """Base class for other exceptions"""
   pass
class AgeSmallException(Error):
   """Raised when the input value is too small"""
   pass
# main program
while True:
   try:
       age = int(input("Enter your age for election: "))
       if age < 18:
           raise AgeSmallException
       else:
             print("you are eligible for election")
```

```
            break
    except AgeSmallException:
        print("This value is too small, try again!")
        print()
```

Output:

```
Enter your age for election: 11
This value is too small, try again!
Enter your age for election: 15
This value is too small, try again!
Enter your age for election: 18
you are eligible for election
>>>
```

Example 2: Raise a user defined exception id password is incorrect.

```
class InvalidPassword(Exception):
    pass

def verify_password(pswd):
    if str(pswd) != "abc":
        raise InvalidPassword
    else:
        print('Valid Password: '+str(pswd))

# main program
verify_password("abc")  # won't raise exception
verify_password("xyz")  # will raise exception
```

Output:

```
Valid Password: abc
Traceback (most recent call last):
  File "C:\Users\Meenakshi\AppData\Local\Programs\Python\Python37\p1.py", line 12, in
<module>
    verify_password("xyz")  # will raise exception
  File "C:\Users\Meenakshi\AppData\Local\Programs\Python\Python37\p1.py", line 6, in
verify_password
    raise InvalidPassword
InvalidPassword
>>>
```

Practice Questions

1. What is file? Enlist types of files in Python programming.
2. What is exception?
3. Explain the term exception handling in detail.
4. Explain different modes of opening a file.
5. Write the syntax of fopen(). With example.
6. What are various modes of file object? Explain any five as them.
7. Explain exception handling with example using try, except, raise keywords.
8. Explain try...except blocks for exception handling in Python.
9. Explain various built in functions and methods.
10. Explain open() and close() methods for opening and closing a file.

11. Explain any three methods associated with files in Python.
12. List and explain any five exceptions in Python.
13. List out keywords used in exception handling.
14. How python handles the exception? Explain with an example program.
15. Differentiate between an error and exception.
16. How to create a user defined exception?
17. Give the syntax and significance of input() method.
18. Give syntax of the methods which can be used to take input from the user in Python program.
19. Write a Python program which will throw exception if the value entered by user is less than zero.
20. Write a Python program to accept an integer number and use try/except to catch the exception if a floating point number is entered.
21. Write a Python program to read contents of first.txt file and write same content in second.txt file
22. Write a Python program to append data to an existing file 'python.py'. Read data to be appended from the use. Then display the contents of entire file.
23. Write a Python program to read a text file and print number of lines, words and characters.
24. Describe the term file I/O handling in detail.

■■■