Unit 5
Files, Modules, Packages

Files and exception: text files, reading and writing files, format operator; command line arguments, errors and exceptions, handling exceptions, modules, packages; Illustrative programs: word count, copy file.

5.1 BASICS OF FILE SYSTEMS

A file system is the structure in which files are given name and placed in the hard disk for storage and retrieval. All the operating systems have file systems which is used to place the files in hierarchical structure. A file system sets guidelines on naming conventions.

It specifies the maximum number of characters in a name; the characters to be used; suffix format and structure of directories.A directory or subdirectory specifies the location of a group of files. The files are placed in a directory or at the desired place in a hierarchical manner as shown in figure 5.1.

The data used in the Python programs specified in the previous units will not be stored in the data disk. They will be lost once the programs are executed. However the programs would be available if it is saved as a Python file.

5.1.1 File

A file stores related data, information, settings, or commands in secondary storage device like magnetic disks, magnetic tapes and optical disks.

A file can be a sequence of bits, bytes, lines or records depending of the application/software used to create it. For example a text file is organized as a sequence of lines.

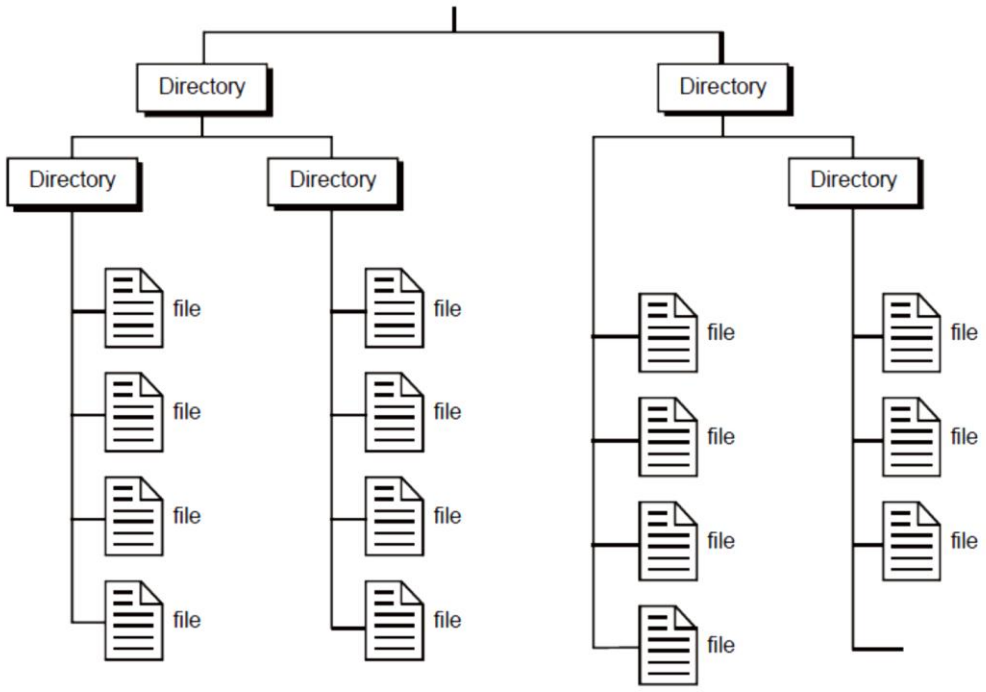

Figure 5.1 Structure of File System

5.1.2 File Types

A file is a collection of contiguous data. It might be a picture, video, text or even set of records. The type enables the operating system to choose the right software that can handle the file.

The following are some of the types of files and the applications that support those files. The extension specifies the type of file. For example .txt, .doc, .jpg etc. In the below table some of the file types are listed along with the application that can access them.

S.No	*Type*	*Application//Software*
1.	Text	Notepad
2.	Audio	MediaPlayer
3.	Video	MediaPlayer
4.	Presentations	PowerPoint
5.	DOC	Word
6.	Pictures	msPaint
7.	HTML	Browsers
8.	PDF	Acrobat Reader
9.	ZIP	WinZip
10.	Executables	Run Programs

Table 5.1 File Types

5.1.3 Ways to Access Files

The contents of the files can be accessed in the following ways;

1. Sequential access

2. Direct/Random access
3. Indexed sequential access

1. Sequential access

Sequential access is the most basic approach. The contents of the file are accessed in a sequential order. For example in a text file the lines are read one after another. Most of the compilers access the file/program to be complied in a sequential manner.

2. Direct/Random access

Random access mechanism reads the contents of the file from any point of the file. It is also called direct access because the desired contents are read/ accessed directly by specifying the address. For example in a text file every character in a line has its own address on the file. That address is used to access the characters. The address is a reference only within the file. The physical address of the character in the storage devices is never used.

3. Indexed sequential access

This mechanism is combination of both sequential and random access. An index is created for the contents of the file. The index can be accessed or searched for desired data in a sequential manner. Once the desired index value is fetched the file contents corresponding to the index is fetched in random. The index contains pointers to the location of the contents of the file that are pertaining to the index value. An index table is mandatory to use this mechanism. It has to be created by the user.

5.2 FILES IN PYTHON

A file refers to a location with filename that stores information. The storage area is non-volatile memory like hard-disk. Like a book the file must first be opened to read or write. Moreover it must be closed after the read/write operations to avoid data damages.

PROGRAM 1

```
1. # File Open Example in Python
2.
3. F_Read = open("Readme.txt", "r")
4. F_Write = open("Writeme.txt", "w")
5. print F_Read
6. print F_Write
```

EXECUTION 1

```
sh-4.3$ python program 1.py
<open file 'Readme.txt', mode 'r' at 0x7f3b747ec4b0>
<open file 'Writeme.txt', mode 'w' at 0x7f3b747ec540>
```

If the file "Readme.txt" is not present the same storage location as main.py python program; the following will be the result

EXECUTION 2

```
sh-4.3$ python program 1.py
File "main.py", line 3, in <module>
F_Read = open("Readme.txt", "r")
IOError: [Errno 2] No such file or directory: 'Readme.txt'
```

Python supports most of the file types. In this chapter only one type of file: text file alone is considered. A text file has sequence of lines in a structured manner. Each line is terminated by EOL or End of Line character. The newline character ends the current line and conveys to the interpreter that a new line has begun. Files other than text files are binary files. The binary files can be used only by an application that can interpret the file's structure.

5.2.1 File Opening Modes

The contents of a file can be read by opening the file in read mode. There are various modes to open a file. They are listed in table 5.2. For reading the modes r, r+, w+ and a+ can be used. Apart from r the rest needs the file to already exist in the system. Similarly for writing into the file; the modes are w, w+, r+ and a+. The file will be newly created in case of write mode. For binary the modes would be rb, rb+, wb and so on. The difference between r+ and w+ is that the file is overwritten by default in w+; the old data is erased.

In general the syntax for opening a file is

<FileVariableName> = open("FilePath", <Mode>)

or

<FileVariableName> = open("FilePath")

Modes	*Description*
r	Opens a file for reading only.
r+	Opens a file for both reading and writing.
w	Opens a file for writing only. Overwrites the file if the file exists.
w+	Opens a file for both writing and reading.
a	Opens a file for appending. File pointer is at the end of the file.
a+	Opens a file for both appending and reading.

Table 5.2 File Modes

Note: When the file mode is not specified the file is opened in read mode.

5.2.2 Attributes of File Object

The file take has been successfully opened possess the following attributes.

Attribute	*Description*
file.closed	If file is closed returns true else false
file.mode	Returns one of the mode listed in table 5.2
file.name	Returns name of the file.
file.softspace	Returns false if space explicitly required with print, true otherwise.

Table 5.3 Attributes of File Object

PROGRAM 2

File Object Attributes Program

```
1. # File Object Attributes
2.
3. fileRead = open("text.txt", "r+")
4.
5. print "Name of the file: ", fileRead.name
6. print "Closed or not : ", fileRead.closed
7. print "Opening mode : ", fileRead.mode
8. print "Softspace flag : ", fileRead.softspace
9.
10. #Most of the above details can be fetched by the below statement
11. print "Printing file object :", fileRead
```

EXECUTION

```
sh-4.3$ python program 2.py
Name of the file: text.txt
Closed or not : False
Opening mode : r+
Softspaceflag : 0
Printing file object :<open file 'text.txt', mode 'r+' at 0x7f3dc07824b0>
```

5.2.3 Methods in File Object

A text file is a file that contains printable characters and whitespace, organized into lines separated by newline characters. The following table lists the various methods available for a file object in Python.

The syntax to invoke these methods is

<FileVariableName>.<MethodeName>(<arguments>)

S.No	Method	Description
1.	close()	Close an opened file.
2..	read(n)	Reads at most n characters from the file.
3.	readable()	Returns True if the file stream can be read from.
4.	readline(n=-1)	Read and return one line (at most n bytes) from the file.
5.	readlines(n=-1)	Read and return a list of lines (at most n bytes) from the file.
6.	seek(offset, from = SEEK_SET)	Change the file position to offset bytes, in reference to from
7.	seekable()	Returns True if the file stream supports random access.
8.	tell()	Returns the current file location.
9.	writable()	Returns True if the file stream can be written to.
10.	write(s)	Write string s to the file and return the number of characters written.
11.	writelines(lines)	Write a list of lines to the file.
12.	flush()	Flushes the internal buffer
13.	fileno()	Returns an integer which is the file descriptor. Depends on the underlying operating system.
14.	isatty()	Returns true if the file is connected to any terminal device
15.	next()	Returns the next line from the file each time it is being called.
16.	truncate(n)	The file is truncated to at most n bytes

Table 5.4 Methods of File Object

PROGRAM 3

```
1. # File methods example
2.
3. file_RW = open("ReadWrite.txt", "r+")
4.
5. #Reads the first 10 bytes
6. str = file_RW.read(10);
7. print "Read String is : ", str
8.
9. #Writes after the 10th byte
10. file_RW.write("Now after 10\n")
11.
12. #Writes lines to file
13. file_RW.writelines(['Line 3\n', 'line 4\n', 'line 5\n'])
```

```
14.
15. # Returns the current position
16. position = file_RW.tell()
17. print "Current file position : ", position
18.
19. # Moves position back to start
20. position = file_RW.seek(0, 0)
21.
22. # Reads first line
23. str = file_RW.readline()
24. print "Read from beginning : ", str
25.
26. # From current position skip 3 bytes
27. position = file_RW.seek(3, 1)
28. print "Read remaining : ", file_RW.readlines()
29.
30. # Close open file
31. file_RW.close()
```

Contents of ReadWrite.txt:

```
1 2 3 4 5 6 7 8 9 10
Line 1
Line 2
```

EXECUTION

```
sh-4.3$ python program 3.py
Read String is : 1 2 3 4 5
Current file position : 44
Read from beginning : 1 2 3 4 5 Now after 10
Read remaining : ['e 3\n', 'line 4\n', 'line 5\n']
Contents of ReadWrite.txt after execution:
1 2 3 4 5 Now after 10
line 3
line 4
line 5
```

5.2.4 Reading Files

The text files can be read in four different ways listed below

* Using Read Method
* Using Readlines Method
* Using For Line In File Method
* Using Readline Method

The syntax for read method is

<fileVariableName>.read(<size>)

The size argument is optional. When the size is not specified the entire file is read and the contents are placed in the left hand side variable. When the size is specified; the size number of characters is read. When an empty file is read; no error is thrown. An empty string is returned.

PROGRAM 4

```
1. # Reading files using Read Method
2.
3. fileRead = open("text.txt", "r")
4.
5. #Reads the first 10 bytes
6. str = fileRead.read(10)
7. length = len(str)
8. print "First 10 characters from the file: ", str
9. print "Length of Read String : ", length
10.
11. #Reads immediately after 10th character
12. str = fileRead.read(20);
13. print "Next 20 characters : ", str
14.
15. #Without the size; the entire file is read
16. str = fileRead.read();
17. print "Read without size : ", str
18.
19. #Read after entire file is read
20. print "Content after end : ", fileRead.read()
21.
22. fileRead.close()
```

Contents of text.txt file:

```
Line1 1234567890
Line2 12345678901
Line3 12345678901
Line4 12345678901
Line5 12345678901
end
```

EXECUTION

```
sh-4.3$ python program 4.py
First 10 characters from the file: Line1 1234
Length of Read String : 10
Next 20 characters : 567890
Line2 1234567
```

Read without size : 8901
Line3 12345678901
Line4 12345678901
Line5 12345678901
end
Content after end :
The syntax for readlines method is

<fileVariableName>.readlines(<size>)

The size argument is optional. When the size is not specified the entire file is read and each line is added as an element in a list. When the size is specified; the lines that make up size bytes are read; mostly the size is equivalent to internal buffer size. When an empty file is read; no error is thrown. An empty list is returned.

PROGRAM 5

```
1. # Reading files using Readlines Method
2.
3. fileRead = open("text.txt", "r")
4.
5. #Reads the lines in file
6. Line_list = fileRead.readlines();
7. print "File lines list: ", Line_list
8.
9. # Read after end of file
10. Line_list = fileRead.readlines();
11. print "After end of file : ", Line_list
12.
13. fileRead.close()
```

Contents of text.txt file:
Line1 Hi
Line2 Bye
Line3 Fine
Line4 Good
Line5 Best
end

EXECUTION

```
sh-4.3$ python program 5.py
File lines list: ['Line1 Hi\n', 'Line2 Bye\n', 'Line3 Fine\n', 'Line4
Good\n', 'Line5 Best\n', 'end']
After end of file : []
```

The syntax for "for line in file" method is

for line in <fileVariableName>:

<process line>

Check the difference between line 7 and line 14 in the output. The comma after the print statement prints the lines from the file immediately. However a print without comma will include a new line after every line.

PROGRAM 6

```
1. # Reading contents using for loop
2.
3. fileRead = open("text.txt", "r+")
4.
5. #Read lines using for loop without comma in print
6. for line in fileRead:
7. print line
8. print "Content after end : ", fileRead.read()
9. fileRead.close()
10.
11. #Read lines using for loop with comma in print
12. fileRead = open("text.txt", "r+")
13. for line in fileRead:
14. print line,
15. print "Content after end : ", fileRead.read()
16. fileRead.close()
```

Contents of text.txt file:

```
Line1 C
Line2 Python
Line3 C#
Line4 Java
Line5 VB
Line5 C++
```

EXECUTION

```
sh-4.3$ python program 6.py
Line1 C
Line2 Python
Line3 C#
Line4 Java
Line5 VB
Line5 C++
```

Content after end :

```
Line1 C
Line2 Python
Line3 C#
```

Line4 Java
Line5 VB
Line5 C++

The syntax for readline method is

<fileVariableName>.readline(<size>)

The size argument is optional. When the size is not specified one line from the file is read and the contents are placed in the left hand side variable. When the size is specified it works as read method. The size number of characters is read. When an empty file is read; no error is thrown. An empty string is returned.

PROGRAM 7

```
1. # Reading contents using readline
2.
3. fileRead = open("text.txt", "r+")
4.
5. #Readline with size specified
6. Line = fileRead.readline(2)
7. print Line
8.
9. #Readline without size
10. Line = fileRead.readline()
11.
12. #Read entire file;till the end
13. while len(Line)>0:
14. print Line,
15. Line = fileRead.readline()
16. fileRead.close()
```

Contents of text.txt file:

Hi first line
Python is great
This is line 3
This book is good
This is line 5

EXECUTION

```
sh-4.3$ python program 7.py
Hi
first line
Python is great
This is line 3
This book is good
This is line 5
```

5.2.5 Opening Files in Different Modes

PROGRAM 8

```
1. # Read mode with different options
2.
3. fileRead = open("empty.txt", "r")
4.
5. #Read empty file
6. Line = fileRead.readline()
7. print "This is contents of empty file:", Line
8. fileRead.close()
9.
10. #Read file with contents
11. fileRead = open("text.txt", "r")
12. Line = fileRead.readline()
13. print "File with contents:",Line
14. fileRead.close()
15.
16. #Read file in current directory
17. fileRead = open("here.txt", "r")
18. Line = fileRead.readline()
19. print "Line in current directory file :", Line
20. fileRead.close()
21.
22. #Read file in different directory
23. fileRead = open("C:/MyFolder/Outside.txt", "r")
24. Line = fileRead.readline()
25. print "Line in C:/MyFolder/Outside.txt :", Line
26. fileRead.close()
27.
28. #Read file not present
29. fileRead = open("notthere.txt", "r")
30. Line = fileRead.readline()
31. print Line
32. fileRead.close()
```

Contents of empty.txt file:

Contents of text.txt file:

Hi first line
Python is great
This is line 3
This book is good
This is line 5

Contents of here.txt file:

I am in this directory...where python is installed
Line 2

Contents of Outside.txt file:

I am in C:/MyFolder/Outside.txt

EXECUTION

```
sh-4.3$ python program 8.py
This is contents of empty file:
File with contents: Hi first line
Line in current directory file : I am in this directory...where python is installed
Line in C:/MyFolder/Outside.txt :I am in C:/MyFolder/Outside.txt
Traceback (most recent call last):
File "main.py", line 24, in <module>
fileRead = open("notthere.txt", "r")
IOError: [Errno 2] No such file or directory: 'notthere.txt'
```

The below program illustrates that in write mode (w), a non existing file will be created. However an existing file opened in write mode will be completely truncated and overridden.

PROGRAM 9

```
1. # Write mode with different options
2.
3. #Open non-existing
4. fileWrite = open("notthere.txt", "w")
5. fileWrite.writelines(["Opened Newly"])
6. fileWrite.close()
7.
8. print "Contents of non-exiting file"
9. fileRead = open("notthere.txt", "r")
10. print fileRead.read()
11. fileRead.close()
12.
13. print "Contents before writing file"
14. fileRead = open("text.txt", "r")
15. print fileRead.read()
16. fileRead.close()
17.
18. #Open file with contents for write
19. fileWrite = open("text.txt", "w")
20. fileWrite.writelines(["hi"])
21. fileWrite.close()
22.
```

```
23. print "Contents after writing file"
24. fileRead = open("text.txt", "r")
25. print fileRead.read()
26. fileRead.close()
```

EXECUTION

```
sh-4.3$ python program 9.py
Contents of non-exiting file
Opened Newly
Contents before writing file
This file will be overridden
Contents after writing file
hi
```

The below program illustrates the difference between r+ and w+. In w+ the contents of the existing file will be totally removed. However in r+ the write pointer will be placed in the beginning of the file and the contents would be overridden from that position. The file A.txt had the following contents,

I am file A
My contents will be gone if opened in W or W+ mode
End

When line 4 is executed all the above contents are deleted and hence when the line 5 is executed only blank lines are read. However in r+ mode; the contents are retained. After execution of line 25 the first line alone is replaced with the given argument.

PROGRAM 10

```
1. # Comparison between w+ and r+
2.
3. print "Contents of A.txt on reading in W+"
4. fileRead = open("A.txt", "w+")
5. print fileRead.read()
6. fileRead.close()
7.
8. #Open file with contents in write+ mode
9. fileWrite = open("A.txt", "w+")
10. fileWrite.writelines(["W+\n", "check"])
11. fileWrite.close()
12.
13. print "Contents of A.txt after writing in W+"
14. fileRead = open("A.txt", "r")
15. print fileRead.read()
16. fileRead.close()
17.
18. #Open file with contents for read+
```

```
19. print "Contents of B.txt in read+ mode"
20. fileRead = open("B.txt", "r+")
21. print fileRead.read()
22. fileRead.close()
23.
24. fileWrite = open("B.txt", "r+")
25. fileWrite.writelines(["R+\n", "ready"])
26. fileWrite.close()
27.
28. print "Contents of B.txt after writing in read+ mode"
29. fileRead = open("B.txt", "r")
30. print fileRead.read()
31. fileRead.close()
```

EXECUTION

```
sh-4.3$ python program 10.py
Contents of A.txt on reading in W+
Contents of A.txt after writing in W+
W+
check
Contents of B.txt in read+ mode
Difference between r+ and w+
Only the above will be replaced
Contents of B.txt after writing in read+ mode
R+
readyce between r+ and w+
Only the above will be replaced
```

The below program illustrates the in detail the append mode. The write pointer is set to end of file when a file is opened in "a" mode. In append mode the file can never be read. The contents can only be written at the end of the file.

PROGRAM 11

```
1. # Understanding append mode
2.
3. #First open in read mode
4. print "Contents of A file before append mode"
5. fileRead = open("A.txt", "r")
6. print fileRead.read()
7. fileRead.close()
8.
9. #Next open in append mode
10. fileApp = open("A.txt", "a")
11. fileApp.writelines(['i ll be placed after the end','\n next line'])
```

```
12. fileApp.close()
13.
5.20 PROBLEM SOLVING AND PYTHON PROGRAMMING
14. #Read the file after writing
15. print "Contents of A file after writing in append mode"
16. fileRead = open("A.txt", "r")
17. print fileRead.read()
18. fileRead.close()
```

EXECUTION:

```
sh-4.3$ python program 11.py
Contents of A file before append mode
The file is going to be appended
Contents of A file after writing in append mode
The file is going to be appended all be placed after the end next line
```

The below program illustrates the difference between a and a+ mode. In a+ mode the file contents can be both read and written. The new contents are appended at the end. The read will not work fine before closing the file. The new contents get saved only after closing the file. So the file opened in a+ mode has to read either before write as in line 5 or after closing the file as in line 19.

PROGRAM 12

```
1. # Understanding difference between a and a+
2.
3. #Open file in a+ and read
4. print "Contents of A file before append mode"
5. fileRead = open("A.txt", "a+")
6. print fileRead.read()
7.
8. #Write to the file
9. fileRead.writelines(['i ll be placed after the end','\n next line'])
10.
11. #Read even before closing
12. print "Read immediately after writing"
13. print fileRead.read()
14.
15. fileRead.close()
16.
17. #Read the file after writing
18. print "Contents of A file after writing in append mode"
19. fileRead = open("A.txt", "r")
20. print fileRead.read()
21. fileRead.close()
```

EXECUTION:

```
sh-4.3$ python program 12.py
Contents of A file before append mode
The file is going to be appended
The difference between a+ and a
Read immediately after writing
Contents of A file after writing in append mode
The file is going to be appended
The difference between a+ and will be placed after the end next line
```

5.3 FORMAT OPERATOR

One of Python's very interesting features is the format operator. This operator used to print the output in a desired format as the printf() statement in C. The % operator is the format operator. It is also called as interpolation operator. The % operator is also used for modulus. The usage varies according to the operands of the operator.

The general syntax for using the format operator is

<format expression> % (v1, v2, v3...vn)

or

<format expression> % values

Where v1 to vn are variables that are formatted using the expression.

Where values is a tuple with exactly the number of items specified by the format string, or a single mapping object.

The syntax of format expression is

[key][flags][width][.precision][length type]conversion type

1. Parameters

All the parameters other than conversion type are optional. It has to be specified only when needed.

* Key: Mapping key, consisting of a parenthesised sequence of characters (for example, (somename)).
* Flags: Conversion flags, which affect the result of some conversion types.
* Width: Minimum field width. If specified as an '*' (asterisk), the actual width is read from the next element of the tuple in values, and the object to convert comes after the minimum field width and optional precision.
* Precision: Precision, given as a '.' (dot) followed by the precision. If specified as '*' (an asterisk), the actual width is read from the next element of the tuple in values, and the value to convert comes after the precision.
* Length type: Length modifier.
* Conversion type: Conversion type.

The below table give the various flags available in Python.

Flag	Use
#	The value conversion will use the "alternate form".
0	The conversion will be zero padded for numeric values.
–	The converted value is left adjusted (a space).
(a space)	A blank should be left before a positive number (or empty string) produced by a signed conversion.
+	A sign character ("+" or "–") will precede the conversion
*	Argument specifies width or precision
(var)	Mapping variable (dictionary arguments)
m.n.	m is the total width and n is the number of digits to display after the decimal point

Table 5.5 Flags in Format Expression

The below table give the various conversion types available in Python.

Conversion	Meaning
d	Signed integer decimal.
i	Signed integer decimal.
o	Unsigned octal.
u	Unsigned decimal.
x	Unsigned hexadecimal (lowercase).
X	Unsigned hexadecimal (uppercase).
e	Floating point exponential format (lowercase).
E	Floating point exponential format (uppercase).
f	Floating point decimal format.
F	Floating point decimal format.
g	Same as "e" if exponent is greater than – 4 or less than precision, "f" otherwise.
G	Same as "E" if exponent is greater than -4 or less than precision, "F" otherwise.
c	Single character (accepts integer or single character string).
r	String (converts any python object using repr()).
s	String (converts any python object using str()).
%	No argument is converted, results in a "%" character in the result.

Table 5.6 Conversion Types

5.3.1 Formatting Strings

PROGRAM 13

```
1. #Format strings
2. print '%s %s' % ('one', 'two')
3.
4. #Format strings with quotes
5. print '%s %r' % ('hi', 'bye')
6.
7. #Padding and aligning strings
8. print '%30s' % ('I ll have space before')
9.
10. #Aligning strings - Left
11. print '%-20s see the space' % ('After me')
12.
13. #Truncating long strings
14. print 'Truncate 5 characters of %r : %.5s' %
('1234567910', '1234567910')
15.
16. #Combining truncating and padding
17. print 'Truncate and pad of %r : %10.5s ' % ('Python is good','Python is good')
18.
19. #print a single character
20. print 'Covert single digit no: %r to char %c' % ('4', '4')
```

EXECUTION

```
sh-4.3$ python program 13.py
one two
hi 'bye'
I ll have space before
After me see the space
Truncate 5 characters of '1234567910' : 12345
Truncate and pad of 'Python is good' :Pytho
Covert single digit no: '4' to char 4
```

5.3.2 Formatting Numbers

PROGRAM 14

```
1. #Format numbers
2. print 'Simple integer formatting : %d' % (42)
3.
4. #Format float
5. print 'Simple Float formatting : %f' % .
(3.141592653589793)
6. print 'Float with 5 digits and 2 digits after . : %05.2f' % (3.141592653589793)
```

```
7.
8. #Padding numbers
9. print 'Padding with spaces %10d' % (45)
10. print 'Padding with zeros %04d' % (42)
11.
12. #Signed numbers
13. print 'Signed no : %+d' % (345)
14. print 'Signed no : %+d' % (-345)
15. print 'A space before positive no :% d' % (145)
16. print 'Sign before negative no :% d' % (-145)
17. print 'Using i is similar to +d : %+i' % (-345)
18.
19. #Format Exponential
20. print 'Exponential formatting : %e' % (6.171790134278593)
21. print 'Exponential formatting : %E' % (6.171790134278593)
22.
23. #Format Decimal
24. print 'Decimal : %u' % (24)
25. print 'Octal : %o' % (24)
26. print 'Hexadecimal : %x' % (1254)
27. print 'Hexadecimal : %X' % (1254)
```

EXECUTION

```
sh-4.3$ python program 14.py
Simple integer formatting : 42
Simple Float formatting : 3.141593
Float with 5 digits and 2 digits after . : 03.14
Padding with spaces 45
Padding with zeros 0042
Signed no : +345
Signed no : -345
A space before positive no : 145
Sign before negative no :-145
Using i is similar to +d : -345
Exponential formatting : 6.171790e+00
Exponential formatting : 6.171790E+00
Decimal : 24
Octal : 30
Hexadecimal : 4e6
Hexadecimal : 4E6
```

5.3.3 Formatting using Place Holders

PROGRAM 15

```
1. #Named placeholders example
2. print '<a href="%(url)s">%(url)s</a>' % {'url':'PondyExpress.com'}
3.
4. print '%(Name)s age is%(Age)d' % {'Name':'Kannan','Age': 56}
5.
6. print 'Unordered %(first)d & %(second)d' % {'second':34, 'first': 56}
```

EXECUTION

```
sh-4.3$ python program 15.py
<ahref="PondyExpress.com">PondyExpress.com</a>
Kannan age is 56
Unordered 56 & 34
```

5.3.4 Formatting in Text Files

Only strings can be written to a text file. However other data types can be written after converting it to string. The following program explains the error in line 6.

PROGRAM 16

```
1. # File example
2. file_RW = open("ReadWrite.txt", "r+")
3.
4. x = 52
5. file_RW.write (str(x))
6. file_RW.write (x) # Error - only string is accepted
7.
8. # Close the file
9. file_RW.close()
```

EXECUTION

```
sh-4.3$ python program 16.py
Traceback (most recent call last):
File "main.py", line 6, in <module>
file_RW.write (x)
TypeError: expected a character buffer object
```

To avoid such errors Python provides format operator % that can be used along with strings. This operator takes two operands; one the format string and other is a tuple of expressions. The number of expressions in the tuple has to match the number of format sequences in the string.
For example "%d %f" % (2), will give an error.
Also, the types of the expressions have to match the format sequences.
For example a string cannot be converted to integer: - "%d" % 'Python'.
The format sequence can be placed anywhere in a string. The converted expression from the tuple is appropriately placed in the string.

For example "I ate %d mangos of type %s and cost Rs. %4.2f" % (4, 'Alphonsa', 34.5)
I ate 4 mangos of type Alphonsa and cost Rs. 34.50
The below program explains how the details in a dictionary are written in formatted way. The contents when printed will also be in the formatted way.

PROGRAM 17

```
1. # File- format operator example
2.
3. file_W = open("StudentDetails.txt", "w")
4.
5. #Student details in dictionary
6. Stud_Dict = {1: ('Kavitha', [200, 198, 185]) , 2: ('Renu', [198, 176, 189]), 3: ('Arjun', [200, 200, 200])}
7.
8. file_W.write ("%-8s %-15s %-10s %-10s %-10s" % ('RollNo', 'Name', 'Maths', 'Physics', 'Chemistry'))
9. file_W.write ('\n')
10.
11. for i in Stud_Dict.keys():
12. RollNo = i
13. Val = Stud_Dict.get(i)
14. Name = Val[0]
15. Marks = Val[1]
16. file_W.write ("%-8s %-15s %-10s %-10s %-10s" % (RollNo, Name, Marks[0], Marks[1], Marks[2]))
17. file_W.write ('\n')
18.
19. file_W.write ('#')
20. file_W.close() # Close the file after writing
21.
22. # Open same file for reading
23. file_R = open("StudentDetails.txt", "r")
24. Ln = file_R.readline()
25. while Ln[0]!='#':
26. print Ln
27. Ln = file_R.readline()
28.
29. # Close file
30. file_R.close()
```

EXECUTION

```
sh-4.3$ python program 17.py
RollNo Name Maths Physics Chemistry
```

1 Kavitha 200 198 185
2 Renu 198 176 189
3 Arjun 200 200 200

5.3.5 Formatting using .format Function

Python supports format function which provides better performance than the format operator %. The below program illustrates the extra features of format function.

All the formatting done by % operator is also incorporated by the format function. It is very similar to the operator; however the conversion types are specified in curly braces. The below is the syntax for the format function

<format expression> .format (v1, v2, v3...vn)

or

<format expression> .format values

Where v1 to vn are variables that are formatted using the expression

Where values is a tuple with exactly the number of items specified by the format string, or a single mapping object.

PROGRAM 18

```
1. # Example for format function
2.
3. #Named placeholders example
4. print '{1} {0}'.format('one', 'two')
5.
6.
7. #Formatting using left, right and center alignment
8. print 'Padding left: {:>20}'.format('spaces before me')
9. print 'Padding right: {:20}#'.format('spaces after me')
10. print 'Padding with character: {:#<20}'.format('char after me')
11. print 'Padding justified:{:^20}.'.format('spaces both sides')
12.
13. #Specify space after the sign
14. print '{:=5d}'.format((- 43))
15.
16. #Formatting elements in a sequence
17. List_data = ['hi', 'bye', 'cpu', 'cse', 23, 42]
18. print 'Specifying the index inside : {d[0]} .{d[3]}'.format(d=List_data)
19.
20. #Formatting using variables
21. print 'Each curl braces is given value : .{:{}{}{}.{}}'.format(6.7789812734, '>', '+', 10, 3)
22. print 'Using variable prec : {:.{prec}} = .
{:.{prec}f}'.format('PythonManiac', 178.3468, prec=3)
```

EXECUTION

```
sh-4.3$ python program 18.py
two one
Padding left: spaces before me
Padding right: spaces after me #
Padding with character: char after me########
Padding justified: spaces both sides .
- 43
Specifying the index inside : hi cse
Each curl braces is given value : +6.78
Using variable prec :Pyt = 178.347
```

5.4 COMMAND LINE ARGUMENTS

The command line arguments are arguments send to the program being executed. sys.argv is a variable that contains the command-line arguments passed to the script.

PROGRAM 19

```
1. # Command Line Arguments Example
2. import sys
3. # Get the total number of args passed to main.py
4. No_Args = len(sys.argv)
5. # Get the arguments list
6. List_Args = str(sys.argv)
7.
8. print ("The total numbers of args passed to the
script: %d " % No_Args)
9. print ("Arguments list: %s " % List_Args)
```

EXECUTION

```
sh-4.3$ python program 19.py input.txt 678 var3
The total numbers of args passed to the script: 1
Args list: ['program 19.py', 'input.txt', '678', 'var3']
```

5.5 EXCEPTION HANDLING

5.5.1 Errors

Errors are caused by the mistakes in the program. There are three types of errors; Syntax errors, Runtime Errors and Logical Errors. The syntax errors can be rectified when the complier throws errors. The logical errors are erroneous output; the program gets executed but the output is not the desired one. A syntactically correct statement might cause errors during execution. Errors detected during execution/runtime are called exceptions.

For Eg: x = 10/0 is syntactically correct. These exceptions cause the program to halt abruptly. In order to proceed further, the exceptions must be handled. There are two types of exceptions; built-in and user-defined. The table of most widely used built-in exceptions are listed in this section.

* ***Syntax Error:***

if x<10
print X
Execution:
File "main.py", line 2
if x<10

* ***Syntax Error:***
  invalid syntax
  for i in range (4) :
  print 'hi'
  Execution:
  File "main.py", line 3
  print 'hi'
* ***Indentation Error:***
  Expected an indented block
* ***Exception Error:***
  X = '678' + 70
  Execution:
  Traceback (most recent call last):
  File "main.py", line 2, in <module>
  X = '678' + 70
  TypeError: cannot concatenate 'str' and 'int' objects
  List_T = ['22', 'hi', 'see you']
  printList_T[3]
  Execution:
  Traceback (most recent call last):
  File "main.py", line 2, in <module>
  printList_T[3]
* ***Index Error:***
  list index out of range
  X = 6
  print x
  Execution:
  Traceback (most recent call last):
  File "main.py", line 2, in <module>
  print x
* ***Name Error:***
  name 'x' is not defined
* ***Logical Errors:***
  The following code prints 16 instead of 30. The line sum_sq+sq must come inside the for loop. Logically instead of giving the sum of square we get only the square of the last number 4.

```
sum_sq = 0
for i in range(5):
sq = i**2
sum_sq += sq
printsum_sq
```

Execution:

16

5.5.2 Exceptions

The below table gives the standard exceptions available in Python.

Exception	*Description*
Exception	**Base class for all exceptions**
StopIteration	**Raised when the next() method of an iterator points to null**
SystemExit	**Raised by the sys.exit() function.**
StandardError	**Base class for all built-in exceptions except StopIteration and SystemExit.**
ArithmeticError	**Base class for all errors that occur for numeric calculation.**
OverflowError	**Raised when a calculation exceeds maximum limit for a numeric type.**
FloatingPointError	**Raised when a floating point calculation fails.**
ZeroDivisionError	**Raised when division by zero takes place**
AssertionError	**Raised in case of failure of the Assert statement.**
AttributeError	**Raised in case of failure of attribute reference**
EOFError	**Raised when there is no input from either the raw_input() or input() function and the end of file is reached.**
ImportError	**Raised when an import statement fails.**
KeyboardInterrupt	**Raised when Ctrl+c is pressed to interrupt program execution**
LookupError	**Base class for all lookup errors.**
IndexError	**Raised when an index is not found in a sequence**
KeyError	**Raised when specified key is not found in the dictionary.**
NameError	**Raised when an identifier is not found in the local or global namespace.**
UnboundLocalError	**Raised when a local variable has no value assigned to it.**

EnvironmentError	Base class for all exceptions that occur outside the Python environment.
IOError	Raised for operating system-related errors.
SyntaxError	Raised when there is an error in Python syntax.
IndentationError	Raised when indentation is not specified properly.
SystemError	Raised when the interpreter finds an internal problem, but when this error is encountered the Python interpreter does not exit.
SystemExit	Raised when Python interpreter is quit by using the sys.exit() function. If not handled in the code, causes the interpreter to exit.
TypeError	Raised when an operation or function is attempted that is invalid for the specified data type.
ValueError	Raised when the built-in function for a data type has the valid type of arguments, but the arguments have invalid values specified.
RuntimeError	Raised when a generated error does not fall into any category.
NotImplementedError	Raised when an abstract method that needs to be implemented in an inherited class is not actually implemented.

Table 5.7 Standard Exceptions

5.5.3 Handling Exceptions

The try and except statements are used to handle the runtime errors. The syntax for normal try-except block is given below

try : # lines of code that might encounter runtime error
except : # lines of code that will be executed when runtime error occurs

In the below program; line 6 will never be executed if the line 4 is absent. Only when an exception happens, the except block would be executed. Moreover the lines after the error line will never be executed in the try block; note line 5 was not executed.

PROGRAM 20

```
1. # Divide by Zero exception
2. try:
3. print "Hello World"
4. x= 10/0
5. print "Never Executed"
6. except:
7. print "This is an error message!"
```

EXECUTION

```
sh-4.3$ python program 20.py
This is an error message!
```

Multiple except statements can be used; however the exception type must be different for except statement.

PROGRAM 21

```
1. # Multiple except statements
2.
3. try:
4. A = int(input("Please enter a no: "))
5. B = int(input("Please enter a no: "))
6. print ("Sum of %d and %d = %d" % (A, B, A+B))
7. print ("Divison of %d by %d = %d" % (A, B, A/B))
8. # Action for Name Error
9. except NameError:
10. print("The inputs must be numbers!")
11. # Action for Arithmetic Error
12. except ArithmeticError:
13. print("Divide by Zero")
```

EXECUTION 1 :

```
sh-4.3$ python program 21.py
Please enter a no: 4
Please enter a no: 0
Sum of 4 and 0 = 4
Divide by Zero
```

EXECUTION 2 :

```
sh-4.3$ python program 21.py
Please enter a no: 3
Please enter a no: r
The inputs must be numbers!
```

In the try-except-else; else block is executed when there is no exception but before the finally-clause. The purpose is to add the code that has to be executed before finalization in a separate block. The code in the else block can be very well written inside try block. However using try-except-else gives a better clarity.

try : # Code that might throw exceptions of type I or II
except EXCEPTION I : # handle_the_exception I
except EXCEPTION II : # handle_the_exception II
else : # Code that has to be executed before ending
finally : # Code that has to be executed always

PROGRAM 22

```
1. try:
2. file_read= open("test.txt", "r")
```

```
3. X = file_read.read()
4. X = X+8
5. except IOError:
6. print "Error: can\'t find file or read data"
7. except :
8. print "Some other error"
9. else:
10. print "Content of file", X
11. finally:
12. file_read.close()
13. print "Finally Close the file"
```

EXECUTION 1 :

When test.txt exists; error in line 4: cannot add string with integer

```
sh-4.3$ python program 22.py
Some other error
Finally Close the file
```

EXECUT ION 2 :

When test.txt doesn't exist ; file is not opened and hence closing in 12 is error

```
sh-4.3$ python program 22.py
Error: can't find file or read data
Traceback (most recent call last):
File "program 22.py", line 12, in <module>
file_read.close()
NameError: name 'file_read' is not defined
```

5.5.4 User Defined Exception

Python allows user to define new exceptions. However these exceptions are derived from the standard exceptions only. For example a student working in Cloud can defined CloudException, but must derive from the Exception class.

PROGRAM 23

```
1. #User Defined Exception
2.
3. #CloudException is extension of standard base class Exception
4. class CloudException(Exception):
5. #The init function exists in Exception class, it is modified
6. def __init__(self, arg):
7. self.args = arg
8. try:
9. #Raise New exception
10. raise CloudException("Cloud Error")
```

```
11. except CloudException,e:
12. # e is a list
13. print e
```

EXECUTION

```
sh-4.3$ python program 23.py
('C', 'l', 'o', 'u', 'd', ' ', 'E', 'r', 'r', 'o', 'r')
```

5.5.5 Raising an Exception

The raise statement in python is used to forcefully invoke an exception. The syntax for raising an exception is

raise<ExceptionName>

the ExceptionName can the exceptions listed in table 5.6 or some used defined exception.

PROGRAM 24

```
1. # Raising standard exceptions
2. try:
3. raise ValueError('Value Error needs to be handled')
4. raise Exception('Generic Exception')
5. # Since ValueError is subclass of Exception -this works
6. except Exception as error:
7. print('caught the error: ' + repr(error))
```

EXECUTION

```
sh-4.3$ python program 24.py
caught the error: ValueError('Value Error needs to be handled',)
```

In the below program,the except clause does not work as the ValueError is specific and Exception is a generic one.

PROGRAM 25

```
1. # Raising Exception and specifying subclass of Exception
2. try:
3. raise Exception('Generic Exception')
4. except ValueError as e:
5. print('we will not catch e')
```

EXECUTION

```
sh-4.3$ python program 25.py
Traceback (most recent call last):
File "main.py", line 4, in <module>
raise Exception('Generic Exception')
Exception: Generic Exception
```

5.6 MODULES

Modules help a programmer in organising his programs logically. This grouping of programs makes understanding and modification of code easier. The module has named attributes that can be bond and referenced. A module is simple python code that can consist of functions, classes and variable. Any program that has been discussed in this book can be treated as a module. The significance of a module is better understood by the usage of import statements. There are several ways to invoke or use the functions in a module in another python program.

1. Import Statement
2. from...import Statement
3. The from...import * Statement

5.6.1 The Import Statement

Any function in a python program can be reused in another python program by using import statement. The import has the following syntax:

import module1[, module2[,... moduleN]

The module is searched for in the search path specified in the first line of the program. In the below program the path is specified as the bin folder. The interpreter searches the entire list of directories specified in the path.

PROGRAM 26

```
1. #!/usr/bin/python
2. # Import module Lib
3. import Lib
4. # Use book index function of Lib
5. Print(Lib.BookIndex("Check")
```

Program of Module Lib

```
1. def BookIndex( BookName ):
2. return "BookIndex = 110"+BookName
```

EXECUTION

```
sh-4.3$ python program 26.py
BookIndex = 110Check
```

A module is loaded only once, regardless of the number of times it is imported. This prevents the module execution from happening over and over again if multiple imports occur.

5.6.2 The from...import Statement

The specific functions/ classes/ variables can be imported using from...import statement. The syntax for this statement is

from<modname> import <name1>[, <name2>[, ...<nameN>]]

This is very useful with the modules are very large and only very few functions are being used in the current program.

5.6.3 The from...import * Statement

There are situations where all the functions of the module be used in the current program. In that situation the wildcard * is used to specify as the following syntax;

frommodname import *

This method is very rarely used. As the modules are always a bundle of large number of logically related functions, classes and variables.

5.6.4 Locating Modules

The modules are searched in the following manner

First : The current directory.

Second: Each directory in the shell variable PYTHONPATH.

Third : Checks the default path. For example in UNIX, this default path is normally /usr/local/lib/python/.

The module search path is generally stored in the system module sys in the sys.path variable. The sys.path variable contains the current directory, PYTHONPATH, and the installation-dependent default.The PYTHONPATH is an environment variable, consisting of a list of directories. The syntax of PYTHONPATH is the same as that of the shell variable PATH.

Here is a typical PYTHONPATH from a Windows system:

set PYTHONPATH=c:\python20\lib;

And here is a typical PYTHONPATH from a UNIX system:

set PYTHONPATH=/usr/local/lib/python

5.7 PACKAGES IN PYTHON

A package is a collection of modules in directories that give a package hierarchy. When a complex application/program is created it is better to be organized. The package gives the hierarchical file directory structure of the Python application environment that consists of modules and subpackages and sub-subpackages, and so on.

A package is a directory which contains a special file called __init__.py. For example; if the project team has decided to maintain all the modules in "MyProject" directory, a folder in the name of "MyProject" is created and the file __init__.py is placed in that folder. All the modules and packages pertaining to the project are placed the folder MyProject. The file__init__.py is mostly empty; however the initialization code for that package can be placed.The below picture gives the hierarchical structure of package "MyProject". The package has two normal programs and two sub-packages.

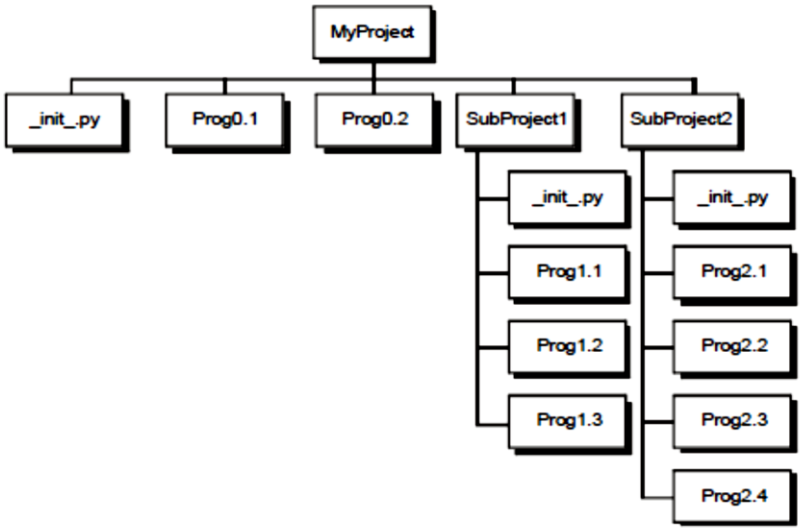

Figure 5.2 Structure of Package

5.7.1 Importing Module from a Package

The modules in package can be used by importing then in the Python program. The "." operator is used to import the modules. The syntax for importing is given below;

import<PackageName>.{<SubPackageName>.}<ModuleName>

The {} braces specify that there can be any number of sub packages. There are different ways in which the functions the modules are invoked. The full name can be used to invoke function. For Example to invoke a function start in Prog1_1 of SubProject1 package;

Option 1:

```
import MyProject.SubProject1.Prog1_1
MyProject.SubProject1.Prog1_1.start()
```

Option 2:

```
fromMyProject.SubProject1 import Prog1_1
Prog1_1.start()
```

Option 3:

```
from MyProject.SubProject1.Prog1_1 import start
start()
```

5.8 PICKLING

The write methods of the file object in Python accepts only string datatype. Any other datatype has to be converted to string before writing in the file. For example; FileEg.write(str([1,2,3])). However when the read methods of the file objects are used to read the above writes; only string format is fetched. Moreover each time the file write is invoked the position of write depends on the file pointer. The below program illustrates the issue. All the datatypes are returned back as string only.

PROGRAM 27

```
1. # Without pickle
```

```
2. File1 = open("test.txt", 'w')
3. #Write String
4. File1.write('hi')
5. #Write float
6. File1.write(str(56.789))
7. #Write dictionary
8. File1.write(str({'Age':23, 'Status': 'Single'}))
9. File1.close()
10.
11. print ("Contents of file : test.txt")
12. File1 = open("test.txt", 'r')
13. print(File1.read())
14. File1.close()
```

EXECUTION

```
sh-4.3$ python program 27.py
Contents of file :test.txt
hi56.789{'Status': 'Single', 'Age': 23}
```

Python consists of a standard module; pickle that preserves the data structures in the file. The files are opened as usual. However when writing and reading the following syntax is used;
For Write :pickle.dump(<datatype variable>, <filename>)
For Read :pickle.load(<filename>)

The following program illustrates the use of pickle module.
PROGRAM 28

```
1. # Use of pickle
2. import pickle
3. File1 = open("test.txt", 'w')
4. #Write String
5. pickle.dump('hi', File1)
6. #Write float
7. pickle.dump(56.789, File1)
8. #Write dictionary
9. pickle.dump({'Age':23, 'Status': 'Single'}, File1)
10. File1.close()
11.
12. print ("Contents of file : test.txt")
13. File1 = open("test.txt", 'r')
14. print(pickle.load(File1))
15. File1.close()
```

EXE CUT ION

```
sh-4.3$ python program 28.py
```

Contents of file : test.txt
hi
56.789
{'Status': 'Single', 'Age': 23}

5.9 ILLUSTRATIVE PROGRAMS

5.9.1 File Compare

The following program illustrates the file comparison between two text files.
The difference in each line is specified as a list.
PROGRAM 29

```
1. # Open file for reading in text mode
2. File1 = open("Version1.txt")
3. File2 = open("Version2.txt")
4. Diff_File = open("Difference.txt",'w')
5.
6. # Read the first line from the files
7. File1Ln = File1.readline()
8. File2Ln = File2.readline()
9.
10. # Initialize counter for line number
11. line_no = 1
12.
13. # Check for end of file
14. while File1Ln != '' or File2Ln != '':
15.
16. #Remove the whitespaces
17. File1Ln = File1Ln.rstrip()
18. File2Ln = File2Ln.rstrip()
19.
20. # If not equal check each word
21. if File1Ln != File2Ln:
22.
23. wordsList1 = File1Ln.split()
24. wordsList2 = File2Ln.split()
25. DiffList = [a for a in wordsList1+wordsList2 if (a
not in wordsList1) or (a not in wordsList2)]
26. TempStr = 'Line No:' + str(line_no) +str(DiffList)
27. Diff_File.write(TempStr)
28. #Read the next line from the file
29. File1Ln = File1.readline()
30. File2Ln = File2.readline()
31.
```

```
32. #Increment line counter
33. line_no += 1
34. # Close the files
35. File1.close()
36. File2.close()
37. Diff_File.close()
38.
39. Diff_File = open("Difference.txt",'r')
40. print 'The difference between Version1.txt and Version2.txt'
41. print
42. print Diff_File.read()
43. Diff_File.close()
```

Contents if Version1.txt:

```
Hi this is file Compare
The line is matching
Python is too Great
Difference is interesting
End
```

Contents if Version2.txt:

```
Hi this is file copy
The line is matching
Python is Great
Difference is interesting
Bye
```

EXECUTION

```
sh-4.3$ python program 29.py
The difference between Version1.txt and Version2.txt
Line No:1 Difference = ['Compare', 'copy']
Line No:3 Difference = ['too']
Line No:5 Difference = ['End', 'Bye']
```

5.9.2 File Copy

Using Module

Python has standard module; shutil module - to handle operations on file or collection of files. There are several functions for file copy and movement. In this program the function shutil.copyfile(sorc, dest) is used.

shutil.copyfile(sorc, dest)

The contents of the file named sorc are copied to the file named dest. The dest must be the complete target file name. When the dest is already existing the contents are completely removed. Errors is raised when any of the following criteria is met

PROGRAM 30

1. Both dest and sorc are the same file
2. When the write permission is denied for dest location
3. sorc and dest are not string

```
1. # File Copy using shutil module
2. from shutil import copyfile
3.
4. # Get the file names
5. sourcefile = input("Enter source file name: ")
6. destinationfile=input("Enter destination file name: ")
7.
8. #copyfile is a function in shutil module
9. copyfile(sourcefile, destinationfile)
10. print("File copied successfully!")
11. print
12. print("Contents of destination file :")
13. print
14.
15. #Read the file contents of destination
16. FileRead = open(destinationfile, "r")
17. print(FileRead.read())
18. FileRead.close()
```

Contents of 'A.txt':

```
python program
I am using module
Copy is very simple
why try new
old is gold
```

EXECUTION

```
sh-4.3$ python program 30.py
Enter source file name: 'A.txt'
Enter destination file name: 'B.txt'
File copied successfully!
Contents of destination file python program
I am using module
Copy is very simple
why try new
old is gold
```

Contents of 'B.txt' after execution:

```
python program
I am using module
Copy is very simple
why try new
```

old is gold

Without using module

PROGRAM 31

```
1. # File Copy without module
2.
3. # Get the file names
4. sorc = input("Enter source file name: ")
5. dest = input("Enter destination file name: ")
6.
7. sourcefile = open(sorc,'r')
8. destinationfile = open(dest,'w')
9.
10. #Copy every line from source to destination
11. for line in sourcefile:
12. destinationfile.write(line)
13.
14. #Close files
15. sourcefile.close()
16. destinationfile.close()
17.
18. print("File copied successfully!")
19. print
20. print ("Contents of destination file")
21.
22. #Read the file contents of destination
23. FileRead = open(dest, "r")
24. print(FileRead.read())
25. FileRead.close()
```

Contents of 'A.txt':

python program
I am using module
Copy is very simple
why try new
old is gold

EXECUTION

```
sh-4.3$ python program 31.py
Enter source file name: 'A.txt'
Enter destination file name: 'B.txt'
File copied successfully!
Contents of destination file
python program
```

I am using module
Copy is very simple
why try new
old is gold
Contents of 'B.txt' after execution:
python program
I am using module
Copy is very simple
why try new
old is gold

5.9.3 Words Count

Using Module

Python has standard module; the collections module is used for the container datatypes like list, dict, set and tuple.

collections.Counter(str)

The Counter function in the collections module returns a dictionary object and takes up string as argument. The dictionary has elements that are keys and values pairs in which the keys are words and value is the count of words in the string.

PROGRAM 32

```
1. #Words Count using module collections
2. from collections import Counter
3. FileName = input("Enter the file name : ")
4. CntFile = open(FileName, 'r')
5.
6. print("Number of words in the file :")
7. print(Counter(CntFile.read().split()))
8. CntFile.close()
```

Contents of 'A.txt' after execution:

python program
I am using module that is very simple
module is very simple module
Old module is very simple

EXECUTION

```
sh-4.3$ python program 32.py
Enter the file name : 'A.txt'
Number of words in the file : Counter({'module': 4, 'is': 3, 'very': 3,
'simple': 3, 'Old': 1, 'that': 1, 'python': 1, 'am': 1, 'I': 1, 'program': 1,
'using': 1})
```

Without using Module

PROGRAM 33

```
1. #Word Count without module
2. FileName = 'Source.txt'
3.
4. Count_Dict = {}
5. CntFile = open(FileName,'r')
6.
7. #For each line in the file count
8. for line in CntFile:
9. #Split line into words
10. word_list = line.split()
11. #For every word in the list
12. for word in word_list:
13. # First time adding into dictionary
14. if word not in Count_Dict:
15. Count_Dict[word] = 1
16. else:
17. # update dictionary with incremented count.
18. Count_Dict[word] = Count_Dict[word] + 1
19.
20. print ("Count of each word in the file :")
21. print('{:15}{:3}'.format('Word','Frequency'))
22. print('-' * 25)
23.
24. # printing the words and its occurrence.
25. for (word,count) in Count_Dict.items():
26. print('{:15}{:3}'.format(word,count))
```

EXECUTION

```
sh-4.3$ python program 33.py
Count of each word in the file :
Word Frequency
------------------------------
Old 1
python 1
is 4
module 4
very 4
without 1
using 1
simple 4
```

TWO MARKS QUESTIONS WITH ANSWERS

1. Define File.

A file refers to a location with filename that stores information. The storage area is non-volatile memory like hard-disk.A file stores related data, information, settings, or commands in secondary storage device like magnetic disks, magnetic tapes and optical disks.A file can be a sequence of bits, bytes, lines or records depending of the application/software used to create it. For example a text file is organized as a sequence of lines.

2. List the File Opening Modes.

Modes	*Description*
r	Opens a file for reading only.
r+	Opens a file for both reading and writing.
w	Opens a file for writing only. Overwrites the file if the file exists.
w+	Opens a file for both writing and reading.
a	Opens a file for appending. File pointer is at the end of the file.
a+	Opens a file for both appending and reading.

3. List the different ways to read a file.

The text files can be read in four different ways listed below

* Using Read Method
* Using ReadlinesMethod
* Using For Line In File Method
* Using Readline Method

4. What is the difference between append and write mode?

The write pointer is set to end of file when a file is opened in "a" mode. In append mode the file can never be read. The contents can only be written at the end of the file. In the write mode the write pointer is set to the beginning of the file.

5. What are the attributes of file objects?

Attribute	*Description*
file.closed	If file is closed returns true else false
file.mode	Returns one of the mode listed in table 5.2
file.name	Returns name of the file.
file.softspace	Returns false if space explicitly required with print, true otherwise.

6. List the methods in file objects.

S.No	Method	Description
1.	close()	Close an opened file.
2..	read(n)	Reads at most n characters from the file.
3.	readable()	Returns True if the file stream can be read from.
4.	readline(n=-1)	Read and return one line (at most n bytes) from the file.
5.	readlines(n=-1)	Read and return a list of lines (at most n bytes) from the file.
6.	seek(offset, from = SEEK_SET)	Change the file position to offset bytes, in reference to from
7.	seekable()	Returns True if the file stream supports random access.
8.	tell()	Returns the current file location.
9.	writable()	Returns True if the file stream can be written to.
10.	write(s)	Write string s to the file and return the number of characters written.
11.	writelines(lines)	Write a list of lines to the file.
12.	flush()	Flushes the internal buffer
13.	fileno()	Returns an integer which is the file descriptor. Depends on the underlying operating system.
14.	isatty()	Returns true if the file is connected to any terminal device
15.	next()	Returns the next line from the file each time it is being called.
16.	truncate(n)	The file is truncated to at most n bytes

7. Differentiate Errors and Exceptions.

Errors are caused by the mistakes in the program. There are three types of errors; Syntax errors, Runtime Errors and Logical Errors. The syntax errors can be rectified when the complier throws errors. The logical errors are erroneous output; the program gets executed but the output is not the desired one. A syntactically correct statement might cause errors during execution. Errors detected during execution/runtime are called exceptions.

8. Illustrate try-except-else.

```
1. # Example for try-except-else
2.
3. try:
4. file_read= open("test.txt", "r")
5. X = file_read.read()
6. X = X+8
7. except IOError:
8. print "Error: can\'t find file or read data"
9. except :
10. print "Some other error"
```

```
11. else:
12. print "Content of file", X
13. finally:
14. file_read.close()
15. print "Finally Close the file"
```

EXECUTION 1 :

When test.txt exists; error in line 4: cannot add string with integer

```
sh-4.3$ python main.py
Some other error
Finally Close the file
```

EXECUTION 2 :

When test.txt doesn't exist ; file is not opened and hence closing in 12 is error

```
sh-4.3$ python main.py
Error: can't find file or read data
Traceback (most recent call last):
File "main.py", line 12, in <module>
file_read.close()
NameError: name 'file_read' is not defined
```

9. Define Modules.

In Python, module is the way to structure program. Each Python program file is a module, which imports other modules like objects and attributes.

10. Define Packages.

A package is a collection of modules in directories that give a package hierarchy. When a complex application/program is created it is better to be organized. The package gives the hierarchical file directory structure of the Python application environment that consists of modules and subpackages and sub-subpackages, and so on.

11. Define Pickling.

Pickle module accepts any Python object and converts it into a string representation and dumps it into a file by using dump function, this process is called pickling. While the process of retrieving original Python objects from the stored string representation is called unpickling.

12. Give the mechanism to handle exceptions.

The try and except statements are used to handle the runtime errors. The syntax for normal try-except block is given below

```
try:
    # lines of code that might encounter runtime error
except:
    # lines of code that will be executed when runtime error occurs
```

In the below program; line 6 will never be executed if the line 4 is absent. Only when an exception happens, the except block would be executed. Moreover the lines after the error line will never be executed in the try block; note line 5 was not executed.

```
1. # Divide by Zero exception
2. try:
3. print "Hello World"
4. x= 10/0
5. print "Never Executed"
6. except:
7. print "This is an error message!"
```

EXECUTION

```
sh-4.3$ python main.py
This is an error message!
```

13. How to raise an exception?

The raise statement in python is used to forcefully invoke an exception. The syntax for raising an exception is

raise<ExceptionName>

theExceptionName can the exceptions or some used defined exception.

14. What is __init__.py used for?

It declares that the given directory is a package. When a file _init_.py is placed in a folder XYZ; it means that folder XYZ is a package.

REVIEW QUESTIONS

1. Illustrate with suitable examples various modes of opening Files.
2. Explain the difference between append mode and write mode.
3. Write a python program to find the longest word in a file.
4. Differentiate Errors and Exceptions with suitable examples.
5. Explain the different try clauses.
6. Explain with examples Modules and Packages.
7. Explain in detail the format operator.