UNIT - 5

FILES, MODULES, PACKAGES

5.1 FILES

Persistence

Most of the programs are transient which means that they run for a short time and produce some output. But, when the program terminates, their data get vanished. When the program started again, it starts with a clean slate. However, there are some other programs which are **persistent** that they run for a long time (or all the time), maintain at least few of their data in permanent storage (for example, a hard drive) and if the system set to shut down and restart, the program takes the data from where it resides.

Examples of persistent programs are operating systems that run better whenever a computer is on and web servers that run all the time and is waiting for requests to come in on the network. One of the simplest ways for programs to maintain their data is by reading and writing text files.

5.1.1 Reading and writing Operation

Standard input and output through input functions such as input () and raw_input() and output function print statement are used in file operation.

The *raw_input* Function:

The *raw_input([prompt])* function reads one line from standard input and returns it as a string (leaving the trailing newline).

```
s = raw_input("Enter your input: ");
print " Entered input is : ", s
```

Output:
Enter your input: Hello Python
Received input is : Hello Python

The input Function

The *input([prompt])* function is similar to raw_input, except that it assumes the input as a valid Python expression and returns the evaluated result.

```
s = input("Enter your input: ");
print " The output is : ", str
```

Output:

```
Enter your input: [x*5 for x in range(2,10,2)]
Recieved input is :  [10, 20, 30, 40]
```

Now, we will see how to use actual data files. A text file is a sequence of characters stored on a permanent storage medium such as hard drive, flash memory, or CD-ROM. Python offers some basic functions and methods necessary to manipulate files by default. The file manipulation can be done using a file object. The basic file operations are open, close, read and write files.

The open Function

To read or write a file, it is necessary to open it using Python's built-in function named *open()* function. The *open()* function creates a **file** object that could be used to call other methods associated with it. The syntax for *open()* function is shown below.

Syntax

```
fileobject = open(file_name [, access_mode][, buffering])
```

The parameters are explained below:

- **file_name:** The file_name argument is a string value that contains the name of the file to access.
- **access_mode:** The access_mode denotes the mode in which the file has to be opened (read, write, append, etc). A complete list of possible values is mentioned below in the table. This parameter is optional and the default file access mode is read (r).
- **buffering:** If the buffering value is set to 0, then there is no buffering takes place. If the buffering value is 1, then line buffering is performed while accessing a file. If the buffering value is set to an integer greater than 1, then buffering action is performed with the indicated buffer size. If the buffering value is negative, then the buffer size is the system default (default behavior).

The list of different file opening modes –

Modes	Description
r	Opens a file for reading only. The file pointer is placed at the beginning of the file. This is the default mode.
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.
r+	Opens a file for both reading and writing. The file pointer placed at the beginning of the file.
rb+	Opens a file for both reading and writing in binary format. The file pointer placed at the beginning of the file.
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.

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

The file Object Attributes

Once a file is opened, there would be one *file* object, to get various information related to that file.

Here is a list of all attributes related to file object:

Attribute	**Description**
file.closed	Returns true if the file is closed, otherwise false.
file.mode	Returns the file access mode with which file was opened.
file.name	Returns name of the file.
file.softspace	Returns false if space explicitly required with print, otherwise true.

The following illustrate the file attribute description using file object.

```
f = open(“file1.txt”, “w+”)
print “Name of the file: “, f.name
print “Closed or not : “, f.closed
print “Opening mode : “, f.mode
print “Softspace flag : “, f.softspace
```

```
Output:
            Name of the file:  file1.txt
            Closed or not:  False
            Opening mode:  w+
            Softspace flag:  0
```

The close() Method

The function close() of a *file* object flushes if there is any unwritten information and closes the file object when there is no more writing can be done. Python closes a file automatically when the reference object of a file is reassigned with another file. It is a good practice to use the close () method to close a file. The syntax of close () function is given below.

Syntax

```
fileObject.close();
```

The program to perform the open and close operations of a file.

```
f = open (“file1.txt”, “w+”)
print “Name of the file: “, f.name
# close opened file
f.close()
```

```
Output:
            Name of the file:  file1.txt
```

Reading and Writing Text Files

Python provides *read ()* and *write ()* methods to read and write files through file object respectively.

The write() Method

The *write()* method is used to write any string to a file which is opened. Python strings can have binary data and not just text. The *write()* method does not add a newline character (‘\n’) to the end of the string. The syntax for *write()* function is shown below.

Syntax

```
fileObject.write(string);
```

The argument passed is the content to be written into the opened file. The following program illustrates the file write operation.

```
f = open(“file1.txt”, “wb”)
print f
f.write( “Python is a programming language.\nIt is very flexible\n”);
```

```
# Close opened file
f.close()
```

Output:

```
open file 'file1.txt', mode 'wb' at 0xb7eb2410
Python is a programming language.
It is very flexible
```

The above program creates a file *file1.txt* and writes the given content in that file and finally it closes that file. If the file is opened, then it would have content what is written.

If the file already exists, then opening it in write mode erases the old data and starts fresh. If the file doesn't exist, then a new one is created.

The write method is used to put data into the file.

for example,

```
line1 = "Python is a programming language, \n"
f.write (line1)
```

Here, the file object keeps track of where it is pointing to, so if write function is called again, it adds the new data to the end of the file. For example,

```
line2 = "It is very flexible .\n"
f.write (line2)
```

If no more operations, need to be performed, then the file could be closed using file close() function.

```
f.close()
```

The read() Method

The file *read() function* reads the file contents from an open file. It is important to note that Python strings can have binary data in addition to text data. The syntax for file read() is given below.

Syntax

```
fileObject.read([count]);
```

The argument passed is the number of bytes to be read from the opened file. This method starts reading from the beginning of the file and if the argument *count* is missing, then it tries to read as much as possible, till the end of file.

Example

To read from the file file1.txt

```
# Open a file
f=open("file1.txt", "w+")
```

```
f.write(" Python is a programming language")
f.close()
f = open("file1.txt", "r+")
str = f.read(20);
print " The string read is : ", str
# Close opened file
f.close()
```

Output:

```
The string read is :   Python is a program
```

5.1.1 Format operator

The file write() function takes the argument as a string. In order to take other values in a file, it is important to convert them into strings. The easiest way to do is with *str* function as follows.

```
x = 52
f.write (str(x))
```

Here, the str function converts integer x value as string. An alternative way is to use the **format operator**, %. When this is applied to integers, % is considered as the modulus operator. But when the first operand is a string, % is considered as the format operator.

The first operand is the **format string** that contains one or more **format sequences**, to specify how the second operand is formatted. The result is a string. For example, the format sequence '%d' means that the second operand should be formatted as an integer (d stands for "decimal"): consider the simple example.

```
x = 15
print '%d' % x
```

Output:

```
15
```

The result is the string '15', which is not to be confused with the integer value 15. A format sequence can appear anywhere in the string, so you can embed a value in a sentence:

```
bugs= 10
print 'I have spotted %d bugs.' % bugs
```

Output:

```
I have spotted 10 bugs
```

If there is more than one format sequence in the string, the second argument must be a tuple. Each format sequence is matched with an element of the tuple, in sequence. The various format sequences are '%d' to format an integer, '%g' to format a floating-point number and '%s' to format a string.

```
print 'In %d years I have spotted %g %s.' % (2, 0.3, 'bugs')
```

Output:

```
In 2 years I have spotted 0.3 bugs.
```

The number of elements in the tuple has to match the number of format sequences in the string. The types of the elements have to match the format sequences also.

Example:

```
print '%d %d %d' % (1, 2)
```

Output:

```
Traceback (most recent call last):
  File "main.py", line 1, in
    print '%d %d %d' % (1, 2)
TypeError: not enough arguments for format string
```

Example:

```
print '%d' % 'dollars'
```

Output:

```
Traceback (most recent call last):
  File "main.py", line 1, in
    print '%d' % 'dollars'
TypeError: %d format: a number is required, not str
```

In the first example, there are three format sequences and only two elements; in the second, the format sequence is for integer but the element is string. The format operator is more effective, however it is difficult to use.

Python File functions:

There are various functions available with the file object.

Method	**Description**
close()	Close an open file. It has no effect if the file is already closed.
detach()	Separate the underlying binary buffer from the `TextIOBase` and return it.
fileno()	Return an integer number (file descriptor) of the file.
flush()	Flush the write buffer of the file stream.
isatty()	Return `True` if the file stream is interactive.

read(*n*)	Read atmost *n* characters form the file. Reads till end of file if it is negative or `None`.
readable()	Returns `True` if the file stream can be read from.
readline(*n*=-1)	Read and return one line from the file. Reads in at most *n* bytes if specified.
readlines(*n*=-1)	Read and return a list of lines from the file. Reads in at most *n* bytes/characters if specified.
seek(*offset*,*from*=`SEEK_SET`)	Change the file position to *offset* bytes, in reference to *from* (start, current, end).
seekable()	Returns `True` if the file stream supports random access.
tell()	Returns the current file location.
truncate(*size*=`None`)	Resize the file stream to *size* bytes. If *size* is not specified, resize to current location.
writable()	Returns `True` if the file stream can be written to.
write(*s*)	Write string *s* to the file and return the number of characters written.
writelines(*lines*)	Write a list of *lines* to the file.

File Positions

The *tell()* method gives the current object position within the file; in other words, the next read or write will occur at that many bytes from the beginning of the file. The *seek (offset [, from])* method modifies the current file position. The *offset* argument specifies the number of bytes to be moved. The *from* argument specifies the reference position from where the bytes are to be moved. If *from* is set to 0, it means use the beginning of the file as the reference position and 1 means use the current position as the reference position and if it is set to 2 then the end of the file would be taken as the reference position. The following program explains the functions of tell() and seek() functions.

```
# Open a file
f=open("file1.txt", "w+")
f.write(" Python is a programming language")
f.close()
f = open("file1.txt", "r+")
str = f.read(20);
print " The string read is : ", str
# Check current position
pos = f.tell();
print "current file position:", pos
```

```
# reposition pointer at the beginning once again
pos = f.seek(0, 0)
str = f . read (10)
print " Again the string read is:  ", str
# close opened file
f . close()
```

Output:

```
The string read is :   Python is a program
Current file position :  20
Again the string read is :   Python is
```

Renaming and Deleting Files

Python **os** module provides methods that enable us to perform file-processing operations like renaming and deleting a file. To use this module it is necessary to import the module first and then the related functions can be called.

The rename() Method

The *rename()* method takes two arguments, the current filename and the new filename.

Syntax:

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

Example:

Following is the example to rename an existing file file*1.txt*:

```
import os
# Rename a file from file1.txt to file2.txt
os.rename( "file1.txt", "file2.txt" )
```

The remove() Method

The *remove()* method can be used to delete files by supplying the name of the file to be deleted as the argument.

Syntax:

```
os.remove(file_name)
```

Example:

Following is the example to delete an existing file *file2.txt* −

```
import os
# Delete file file2.txt
os.remove("file2.txt")
os.mkdir("test")
```

Program for Mail Merge:

To send the same invitations to many people, the body of the mail does not change. Only the name (and maybe address) needs to be changed. Mail merge is a process of doing this. Instead of writing each mail separately, we have a template for body of the mail and a list of names that we merge together to form all the mails.

```
# Python program to mail merger
# Names are in the file names.txt
# Body of the mail is in body.txt
# open names.txt for reading
with open("names.txt",'r',encoding = 'utf-8') as names_file:
  # open body.txt for reading
  with open("body.txt",'r',encoding = 'utf-8') as body_file:
    # read entire content of the body
    body = body_file.read()
    # iterate over names
    for name in names_file:
      mail = "Hello "+name+body
      # write the mails to individual files
      with open(name.strip()+".txt",'w',encoding = 'utf-8') as mail_file:
        mail_file.write(mail)
```

In this program all the names are written in separate lines in the file named “names.txt”. The body of the letter is stored in the file “body.txt”

The two files are opened in reading mode and iterate over each name using a `for` loop. A new file with the name “[*name*].txt” is created, where *name* is the name of that person. Here, the `strip()` method is used to clean up leading and trailing whitespaces (reading a line from the file also reads the newline ‘\n’ character). Finally, we write the content of the mail into this file using the `write()` method.

5.1.3 Command Line Arguments

Command line arguments are what we type at the command line prompt along with the script name while we try to execute our scripts. Python like most other languages provides this feature. sys.argv is a list in Python, which contains the command line arguments passed to the script. We can count the number of arguments using len(sys.argv) function. To use sys.argv, we have to import the sys module.

```
import sys
print 'No. of arguments:', len(sys.argv)
print 'Argument List:',str(sys.argv)
```

Run the above script as follows:

```
$ python test.py arg1 arg2 arg3
```

Output:

```
Number of arguments: 4
Argument List: [‘test.py’, ‘arg1’,’arg2’,’arg3’]
```

Filenames and paths

Files are organized into **directories** (also called “folders”). Every program has a “current directory”, which is the default directory for most operations. For example, when you open a file for reading, Python looks for it in the current directory. The **os** module provides functions for working with files and directories (“os” stands for “operating system”). os.getcwd () returns the name of the current directory:

```
import os
cwd = os.getcwd()
print cwd
```

Output:

```
/web/com/1493114533_4353
```

cwd stands for “current working directory.” The result in this example is /web/com/1493114533_4353.

A string like cwd that identifies a file is called a **path**. A **relative path** starts from the current directory; an **absolute path** starts from the topmost directory in the file system. The paths we have seen so far are simple filenames, so they are relative to the current directory. To find the absolute path to a file, you can use os.path.abspath:

```
abs_path=os.path.abspath(‘file1.txt’)
print abs_path
```

Output:

```
/web/com/1493114533_4353/file1.txt
```

os.path.exists checks whether a file or directory exists:

```
print os.path.exists(‘file1.txt’)
```

Output:

```
True
```

If it exists, os.path.isdir checks whether it’s a directory:

```
print os.path.isdir(‘file1.txt’)
```

Output:

```
False
```

Similarly, os.path.isfile checks whether it's a file.

os.listdir returns a list of the files (and other directories) in the given directory:

```
>>> os.listdir(cwd)
['file1', 'file2']
```

To demonstrate these functions, the following example "walks" through a directory, prints the names of all the files, and calls itself recursively on all the directories.

```
def walk(dirname):
for name in os.listdir(dirname):
path = os.path.join(dirname, name)
if os.path.isfile(path):
print path
else:
walk(path)
```

os.path.join takes a directory and a file name and joins them into a complete path.

5.2. ERRORS AND EXCEPTION

5.2.1 Errors

Errors or mistakes in a program are often referred to as bugs. They are almost always the fault of the programmer. Debugging is the process of finding and eliminating errors. Errors can be classified into three major groups:

- Syntax errors
- Runtime errors
- Logical errors

Syntax Errors

Syntax errors, also known as parsing errors are identified by Python while parsing the program. It displays error message and exit without continuing execution process. They are similar to spelling mistakes or grammar mistakes in normal language like English. Some common Python syntax errors include:

- leaving out a keyword
- putting a keyword in the wrong place
- leaving out a symbol, such as a colon, comma or brackets
- misspelling a keyword
- incorrect indentation
- empty block

Here are some examples of syntax errors in Python:

```
a=10
b=20
if a<b
print 'a is greater'
```

Error Message:

```
File "main.py", line 3
  if a<b
       ^
```

SyntaxError: invalid syntax

The parser repeats the offending line and displays a little 'arrow' pointing at the earliest point in the line where the error was detected. The error is caused by (or at least detected at) the token *preceding* the arrow: in the example, the error is detected at the if a<b since a colon (':') is missing before it. File name and line number are printed so you know where to look in case the input came from a script.

```
if True:
        prnt 'Hello'
```

Error Message:

```
File "main.py", line 2
  prnt 'Hello'
           ^
```

SyntaxError: invalid syntax

In the above example, the error is detected at prnt 'Hello' since print is misspelled.

Logical errors

Logical errors occur due to mistake in program's logic. Here program runs without any error messages, but produces an incorrect result. These errors are difficult to fix. Here are some examples of mistakes which lead to logical errors:

- using the wrong variable name
- indenting a block to the wrong level
- using integer division instead of floating-point division
- getting operator precedence wrong
- making a mistake in a boolean expression
- off-by-one, and other numerical errors

Here is an example of logical error in Python:

```
i=1
fact=0
while i<=5:
    fact=fact*i
    i=i+1
print 'Fact:', fact
```

Sample Output:

```
Fact:0
```

In this example for computing factorial of 5, the obtained output is 0. There are no syntax errors. The wrong output occurs due to logical error *fact=0*. To compute factorial, the *fact* value must be initialized to 1. As it is assigned as 0, it results in wrong output.

5.2.2 Exceptions

An exception is an error that occurs during execution of a program. It is also called as **run time errors**. Some examples of Python runtime errors:

- division by zero
- performing an operation on incompatible types
- using an identifier which has not been defined
- accessing a list element, dictionary value or object attribute which doesn't exist
- trying to access a file which doesn't exist

An example for run time error is as follows.

```
print (10/0)
```

Error Message:

```
Traceback (most recent call last):
File "main.py", line 1, in <module>
print (10/0)
ZeroDivisionError: integer division or modulo by zero
```

Exceptions come in different types, and the type is printed as part of the message: the type in the example is ZeroDivisionError which occurs due to division by 0. The string printed as the exception type is the name of the built-in exception that occurred.

Exception refers to unexpected condition in a program. The unusual conditions could be faults, causing an error which in turn causes the program to fail. The error handling mechanism is referred to as exception handling. Many programming languages like C++, PHP, Java, Python, and many others have built-in support for exception handling.

Python has many built-in exceptions which forces your program to output an error when something in it goes wrong. When these exceptions occur, it stops the current process and passes the control to corresponding exception handler. If not handled, our program will crash.

Some of the standard exceptions available in Python are listed below.

Exception Name	**Description**
Exception	Base class for all exceptions
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 or modulo by zero takes place for all numeric types.
AssertionError	Raised in case of failure of the Assert statement
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.
IndexError	Raised when an index is not found in a sequence
KeyError	Raised when the specified key is not found in the dictionary.
NameError	Raised when an identifier is not found in the local or global namespace
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.
SyntaxError	Raised when there is an error in Python syntax
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

Handling Exceptions

The simplest way to handle exceptions is with a “try-except” block. Exceptions that are caught in try blocks are handled in except blocks. The exception handling process in Python is shown in Figure.5.1. If an error is encountered, a try block code execution is stopped and control transferred down to except block.

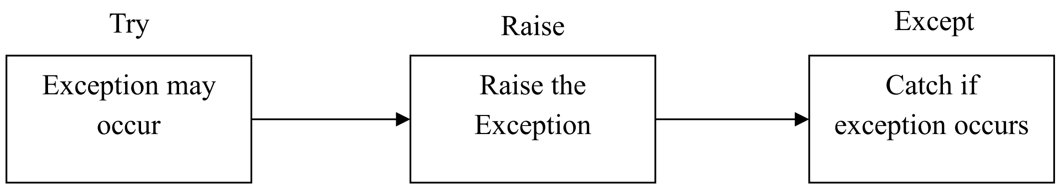

Figure 5.1. Exception Handling

Syntax:

```
try:
        # statements
        break
except ErrorName:
        # handler code
```

The *try* statement works as follows.

- First, the ***try*** *clause* (the statement(s) between the try and except keywords) is executed.
- If no exception occurs, the ***except*** *clause* is skipped and execution of the try statement is finished.
- If an exception occurs during execution of the try clause, the rest of the clause is skipped. Then if its type matches the exception named after the except keyword, the except clause is executed, and then execution continues after the try statement.
- If an exception occurs which does not match the exception named in the except clause, it is passed on to outer try statements; if no handler is found, it is an *unhandled exception* and execution stops with a message.

A simple example to handle divide by zero error is as follows.

```
(x,y) = (5,0)
try:
  z = x/y
except ZeroDivisionError:
  print "divide by zero"
```

Sample Output:

```
divide by zero
```

To display built-in error message of exception, you could have :

```
(x,y) = (5,0)
try:
  z = x/y
```

```
except ZeroDivisionError as e:
  z = e # representation: "<exceptions.ZeroDivisionError instance at 0x817426c>"
print z # output: "integer division or modulo by zero"
```

Sample Output:

```
integer division or modulo by zero
```

A *try* statement may have more than one except clause, to specify handlers for different exceptions. If an exception occurs, Python will check each *except* clause from the top down to see if the exception type matches. If none of the *except* clauses match, the exception will be considered *unhandled*, and your program will crash:

Syntax:

```
try:
        # statements
        break
except ErrorName1:
        # handler code
except ErrorName2:
        # handler code
```

A simple example to handle multiple exceptions is as follows.

```
try:
   dividend = int(input("Please enter the dividend: "))
   divisor = int(input("Please enter the divisor: "))
   print("%d / %d = %f" % (dividend, divisor, dividend/divisor))
except ValueError:
   print("The divisor and dividend have to be numbers!")
except ZeroDivisionError:
   print("The dividend may not be zero!")
```

Sample input/output (successful):

```
Please enter the dividend: 10
Please enter the divisor: 2
10 / 2 = 5.000000
```

Sample input/output (unsuccessful-divide by zero error):

```
Please enter the dividend: 10
Please enter the divisor: 0
The dividend may not be zero!
```

Sample input/output (unsuccessful-value error):

```
Please enter the dividend: ‘s’
The divisor and dividend have to be numbers!
```

An except clause may name multiple exceptions as a parenthesized tuple, for example:

```
... except (RuntimeError, TypeError, NameError):
...     #handler code
```

Example:

```
try:
    dividend = int(input(“Please enter the dividend: “))
    divisor = int(input(“Please enter the divisor: “))
    print(“%d / %d = %f” % (dividend, divisor, dividend/divisor))
except(ValueError, ZeroDivisionError):
    print(“Oops, something went wrong!”)
```

Sample Input/Output:

```
Please enter the dividend: 10
Please enter the divisor: 0
Oops, something went wrong!
```

To **catch all** types of exceptions using single except clause, simply mention *except* keyword without specifying error name. It is shown in following example.

```
try:
    dividend = int(input(“Please enter the dividend: “))
    divisor = int(input(“Please enter the divisor: “))
    print(“%d / %d = %f” % (dividend, divisor, dividend/divisor))
except:
    print(“Oops, something went wrong!”)
```

Raising Exceptions

The ***raise*** statement initiates a new exception. It allows the programmer to force a specified exception to occur. The ***raise*** statement does two things: it creates an *exception* object, and immediately leaves the expected program execution sequence to search the enclosing *try* statements for a matching *except* clause. It is commonly used for raising user defined exceptions. Two forms of the *raise* statement are:

Syntax:

```
raise ExceptionClass(value)
raise Exception
```

Example:

```
try:
        raise NameError
except NameError:
        print('Error')
```

Sample Output:

```
Error
```

raise without any arguments is a special use of python syntax. It means get the exception and re-raise it. The process is called as **reraise**. If no expressions are present, raise re-raises the last exception that was active in the current scope.

Example:

```
try:
        raise NameError
except NameError:
        print('Error')
        raise
```

Sample Output:

```
Error
Traceback (most recent call last):
File "main.py", line 2, in <module>
raise NameError('Hi')
NameError: Hi
```

In the example, *raise* statement inside *except* clause allows you to re-raise the exception *NameError*.

The else and finally statements

Two clauses that can be added optionally to *try-except* block are *else* and *finally*. ***else*** will be executed only if the *try* clause doesn't raise an exception:

```
try:
   age = int(input("Please enter your age: "))
except ValueError:
   print("Hey, that wasn't a number!")
else:
   print("I see that you are %d years old." % age)
```

Sample input/output:

```
Please enter your age: 10
I see that you are 10 years old.
Please enter your age: ‘a’
Hey, that wasn’t a number!
```

In addition to using *except* block after the *try* block, you can also use the ***finally*** block. The code in the finally block will be executed regardless of whether an exception occurs and even if we exit the block using *break*, *continue*, or *return*.

```
try:
   age = int(input(“Please enter your age: “))
except ValueError:
   print(“Hey, that wasn’t a number!”)
else:
   print(“I see that you are %d years old.” % age)
finally:
   print(“Goodbye!”)
```

Sample input/output:

```
Please enter your age: 20
I see that you are 20 years old.
Goodbye!
```

5.2.2.4. User-defined Exceptions

Python allows the user to create their custom exceptions by creating a new class. This exception class has to be derived, either directly or indirectly, from *Exception* class.

```
# define Python user-defined exceptions
class Error(Exception):
  “””Base class for other exceptions”””
  pass # null operation
class PosError(Error):
  “””Raised when the input value is positive”””
  pass
class NegError(Error):
  “””Raised when the input value is negative”””
  pass
```

```
# our main program
number = 0
while True:
  try:
    i_num = int(input("Enter a number: "))
    if i_num < number:
      raise NegError
    elif i_num > number:
      raise PosError
    break
  except PosError:
    print("This value is positive!")
    print()
  except NegError:
    print("This value is negative!")
    print()
```

Sample input/output:

```
Enter a number: 12
This value is positive!
```

In the example, the user defined exception class *Error* is derived from built-in class *Exception*. It handles two user defined exceptions: *PosError*, raised when input value is positive and *NegError*, raised when input value is negative. The *pass* keyword indicates null block. The main program reads user input and compares input value with 0. If input>0, the exception *PosError* is raised using raise keyword else the exception *NegError* is raised.

5.3 MODULES

A Python module is a file that consists of Python code. It allows us to logically arrange related code and makes the code easier to understand and use. It defines functions, classes and variables.

Python has many useful functions and resources in modules. Functions such as abs() and round() from __builtin__ module are always directly accessible in every Python code. But, the programmer must explicitly import other functions from the modules in which they are defined.

import statement

An import statement is used to import Python module in some Python source file.

The syntax is:

import module$_1$[, module$_2$[,... module$_n$]

When an 'import' statement is encountered by the interpreter, the corresponding module(s) is imported if it is available in the search path.

Example:

```
import math
```

To use a resource from a module, the following syntax is used:

```
modulename.resourcename
```

For example, math is a built-in module that offers several built-in functions for carrying out basic mathematical operations. The following code imports math module and lists a directory of its resources:

```
import math
print dir(math)
```

Output:

```
['__doc__', '__file__', '__name__', '__package__', 'acos', 'acosh', 'asin',
'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e',
'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum',
'gamma', 'hypot', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p',
'modf', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc']
```

The usage of some built-in functions of math module is shown in the following code along with its output:

```
import math       # Import built-in module math
print math.floor(6.9)
print math.ceil(6.9)
print math.pow(3,4)
```

Output:

```
6.0
7.0
81.0
```

The following table gives description about a few modules of Python:

Table 5.1: Python modules and their description

Module	Description
cmath	Mathematical operations using complex numbers
copy	Shallow copy and deep copy operations

datetime	Date and time
fileinput	Loop over standard input or list of files
keyword	Testing whether a given string is a keyword
linecache	Accessing individual lines of text files randomly
math	Basic mathematical operations
modulefinder	Finding modules
numbers	Abstract base classes for numerals
operator	Functions analogous to basic operators
py_compile	Compiling Python source code to generate byte code
statistics	Statistical operations
string	String operations

5.3.1 Writing modules

Any Python source file can be imported as a module into another Python source file. For example, consider the following code named as support.py, which is a Python source file defining two functions add() and display().

support.py

```
def add( a, b ):
  print ‘Result is ‘, a+b
  return
def display(p):
   print ‘Welcome, ‘,p
   return
```

This support.py file can be imported as a module in another Python source file and its functions can be called from the new file as shown in the following code:

```
import support    # Import module support
support.add(3,4)    # calling add() of support module with two integer values
support.add(3.5,4.7)    # calling add() of support module with two real values
support.add(‘a’,’b’)    # calling add() of support module with two character values
support.add(‘Ram’,’Kumar’)    # calling add() of support module with two string values
support.display(‘Ram’)    # calling display() of support module with a string value
```

When this code is executed, the following output is produced:

```
Result is  7
Result is  8.2
Result is  ab
Result is  RamKumar
Welcome,  Ram
```

from...import Statement

It allows us to import specific attributes from a module into the current namespace.

Syntax:

```
from modulename import name1[, name2[, ... nameN]]
```

The first statement of the following code does not import the entire module support into the current namespace; it just introduces the item add from the module support into the global symbol table of the importing module. Hence, a call to display() function generates an error as shown in the output.

```
from support import add    # Import module support
add(3,4)    # calling add() of support module with two integer values
add(3.5,4.7)    # calling add() of support module with two real values
add('a','b')    # calling add() of support module with two character values
add('Ram','Kumar')    # calling add() of support module with two string values
display('Ram')    # calling display() of support module with a string value
```

Output:

```
Result is  7
Result is  8.2
Result is  ab
Result is  RamKumar
Traceback (most recent call last):
  File "main.py", line 8, in
    display('Ram') # calling display() of support module with a string value
NameError: name 'display' is not defined
```

from...import * Statement:

It allows us to import all names from a module into the current namespace.

Syntax:

```
from modulename import *
```

Sample Code:

```
from support import *    # Import module support
add(3,4)    # calling add() of support module with two integer values
add(3.5,4.7)   # calling add() of support module with two real values
add('a','b')   # calling add() of support module with two character values
add('Ram','Kumar')   # calling add() of support module with two string values
display('Ram')   # calling display() of support module with a string value
```

Sample Output:

```
Result is  7
Result is  8.2
Result is  ab
Result is  RamKumar
Welcome,  Ram
```

Programs that will be imported as modules often use the following expression:

```
if __name__ == '__main__':

# test code
```

Here, __name__ is a built-in variable and is set when the program starts execution. If the program runs as a script, __name__ has the value __main__ and the test code is executed. Else, the test code is skipped.

Sample Code:

```
from support import *   # Import module support
if __name__ == '__main__':   # add() and display() are called only if this pgm runs as
a script.
  add(3,4)
  display('Ram')
```

Sample Output:

```
Result is  7
Welcome,  Ram
reload()
```

When the module is already imported into a script, the module is not re-read eventhough it is modified. The code of a module is executed only once. To reload the previously imported module again, the reload() function can be used.

Syntax:

```
reload(modulename)
```

Suppose we have the following code in a module named my_module.

```
print("Welcome")
```

Now, examine the following execution sequence:

```
>>> import my_module
Welcome
>>> import my_module
>>> import my_module
```

Here, the code is executed only once since the module was imported only once. If the module is subsequently changed, it has to be reloaded. For this reloading, one of the approaches is to restart the interpreter. But this does not help much. So, we can employ reload() inside the imp module as shown:

```
>>> import imp
>>> import my_module
Welcome
>>> import my_module
>>> imp.reload(my_module)
Welcome
<module 'my_module' from '.\\my_module.py'>
```

5.3.2 Locating Modules

When a module is imported, Python interpreter searches for the module in the following sequence:

(1) The current directory.

(2) If the module is not found in the current directory, each directory in the shell variable PYTHONPATH is searched.

(3) At last, Python checks the installation-dependant default directory.

The module search path is stored in sys module as **sys.path** variable. This variable contains the current directory, PYTHONPATH, and the installation-dependent default.

5.4 PACKAGES

When we have a large number of Python modules, they can be organized into packages such that similar modules are placed in one package and different modules are placed in different packages. A package is a hierarchical file directory structure that defines a single Python application environment that consists of modules, sub-packages, sub-subpackages, and so on. In another words, it is a collection of modules. When a package is imported, Python explores in list of directories on sys.path for the package subdirectory.

5.4.1 Steps to Create a Python Package

(1) Create a directory and name it with a package name.

(2) Keep subdirectories (subpackages) and modules in it.

(3) Create **__init__.py** file in the directory

This **__init__.py** file can be left empty but we generally place the initialization code with import statements to import resources from a newly created package. This file is necessary since Python will know that this directory is a Python package directory other than an ordinary directory.

Example:

Assume we are creating a package named Animals with some subpackages as shown in Figure.5.2.

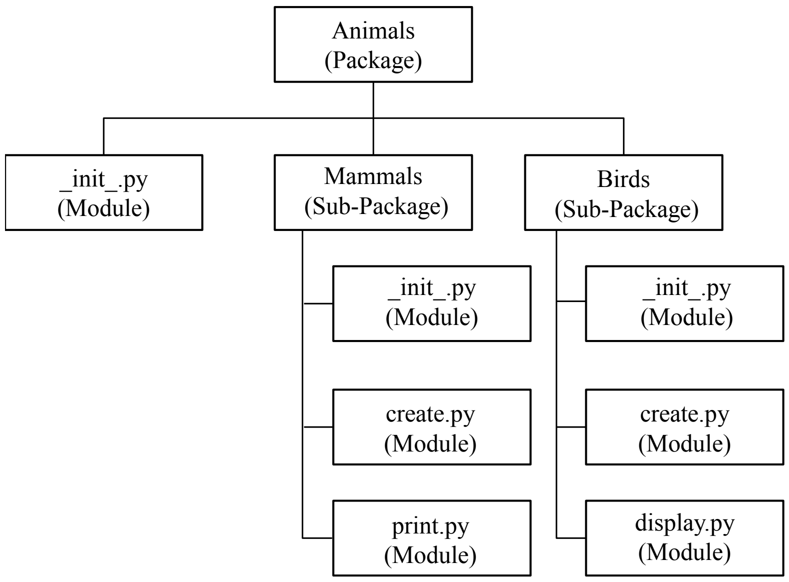

Figure 5.2. Organization of packages and modules

Modules are imported from packages using dot (.) operator.

Method 1:

Consider, we import the display.py module in the above example. It is accomplished by the following statement.

```
import Animals.Birds.display
```

Now if this display.py module contains a function named displayByName ( ) , we must use the following statement with full name to reference it.

```
Animals.Birds.display.displayByName()
```

Method 2:

On another way, we can import display.py module alone as follows:

```
from Animals.Birds import display
```

Then, we can call displayByName() function simply as shown in the following statement:

```
display.displayByName()
```

Method 3:

In the following statement, the required function alone is imported from a module within a package:

```
from Animals.Birds.display import displayByName
```

Now, this function is called directly:

```
displayByName()
```

Though this method is simple, usage of full namespace qualifier avoids confusion and prevents two similar identifier names from colliding.

In the above example, __init__.py of Animals package contains the following code:

```
from Mammals import Mammals
from Birds import Birds
```

In python, module is a single Python file and package is a directory of Python modules containing an additional `__init__.py` file. Packages can be nested to any depth, but the corresponding directories should include their own `__init__.py` file.

5.5 ILLUSTRATIVE PROGRAMS

5.5.1 Python program to handle exception when file open fails

```
try:
  fob = open("test", "r")    # open file in read mode
  fob.write("It's my test file to verify exception handling in Python!!")
except IOError:
  print "Error: can\'t find the file or read data"    # Exception occurs
else:
  print "Write operation is performed successfully on the file"  # no Exception
```

Error:

```
Error: can't find the file or read data
```

5.5.2 Python program to raise an exception when the user input is negative

```
try:
   a = int(input("Enter a positive integer value: "))
   if a <= 0:
       raise ValueError("This is not a positive number!!")
except ValueError as ve:
    print(ve)
```

Sample input:

```
Enter a positive integer value: -1
Error:
This is not a positive number!!
```

5.5.3 Python program to count number of words in a file

```
 try:
            filename = ‘GettysburgAddress.txt’ # specify your input file name
             textfile = open(filename, ‘r’)
            print(“The number of words are: “ + len(textfile.split(“ “)))
            textfile.close()
except IOError:
             print ‘Cannot open file %s for reading’ % filename
             import sys
             sys.exit(0)
```

Sample Output:

```
The number of words are: 20
```

5.5.4 Python program to count the frequency of words in a text file

```
file=open(“test.txt”,”r+”)
wordcount={}      # define a dictionary that holds words its count
for word in file.read().split():    # for loop iterates through each word is the file
   if word not in wordcount:
      wordcount[word] = 1
   else:
      wordcount[word] += 1
for k,v in wordcount.items():                          # Exception is raised
   print k, v                                            if no is negative
file.close()
```

Sample Output:

```
This 1
program 2
example 1
Python 2
```

5.5.5 Python program to copy a content of one file to another

Program 1:

```
rfile=open('/testfile.txt', 'r') try:    # open file is read mode
    reading_file=rfile.read()
    wfile=open('/testfile2.txt', 'w')    # open file is write mode
    try:
        wfile.write(reading_file)        # write into file
    finally:
        wfile.close()
finally:
    rfile.close()
```

Program 2:

```
with open("in.txt") as f:
    with open("out.txt", "w") as f1:
        for line in f:
            if "ROW" in line:
                f1.write(line)
```

Program 3:

```
# The shutil module offers a number of high-level operations on files and collections
of files
from shutil import copyfile
copyfile('test.py', 'abc.py')        # copy content of test.py to abc.py
```

5.5.6 Some Additional Programs

Python program to append text to a file and display the text

```
def file_read(fname):
        with open(fname, "w") as myfile:
                myfile.write("Python Exercises\n")
                myfile.write("Java Exercises")
        txt = open(fname)
        print(txt.read())
file_read('abc.txt')
```

Sample Output:

```
Python Exercises
Java Exercises
```

Python program to count the number of lines in a text file

```
def file_lengthy(fname):
    with open(fname) as f:
        for i, l in enumerate(f):
            pass
    return i + 1
print("Number of lines in the file: ",file_lengthy("test.txt"))
```

In this example, *f* is the file object. *enumerate(f)* iterate over lines of the file. So each time through the loop *i* gets assigned a line number, and *l* gets assigned the corresponding line from the file.

Random access file - Python program to read a random line from a file.

A **random-access** data file enables you to read or write information anywhere in the file. You can use **seek()** method to set the file marker position and **tell()** method to get the current position of the file marker. In a **sequential-access** file, you can only read and write information sequentially, starting from the beginning of the file.

```
f=open('Python_source\\test.txt','w')
f.write('DearChanna ')
f.seek(4) #move file pointer to 4th position from beginning of file
f.write(' Mr.Channa')
f.close()
f=open('Python_source\\test.txt','r')
f.read()
```

Sample Output:

```
Dear Mr.Channa
```

Program that asks the user to input customer information, call *writetofile* method to write data to the file and call *getall* method to retrieve customer information from file.

```
#write data to file
def writetofile(Name,Email='',Tel='',Address=''):
    try:
        f=open(r'customerlist.txt','a')
        f.write(Name+':'+Email+':'+Tel+':'+Address+'\n')
    except Exception:'Print error in writing to file...'
    finally:
        f.flush()
        f.close()
#Get all customers’information and display
def getall():
    f=open(r’customerlist.txt’,’r’)#open file for reading
    content=f.readlines()#read all lines
    f.close()
    return content
def add():
    Name=raw_input(‘Name:’)
    Email=raw_input(‘Email:’)
    Tel=raw_input(‘Tel:’)
    Address=raw_input(‘Address:’)
    writetofile(Name,Email,Tel,Address)
#main program
add()
 print getall()
```

With reference to previous program, write a method to search for a customer by name.

```
#Search customer by name. If match is found, it returns the position of the name
else returns -1.
def search(Name):
    global flag#declare global variable
    try:
      f=open(r’customerlist.txt’,’r’)#open file for reading
      f.seek(0)
      content=f.readline()
      while content!=’’:
         if content.find(Name)!=-1:
            print content
             flag=1
              return int(f.tell())-int(len(content)+1)  #  return the position of the
                                                           matched name
       else:
```

```
                        content=f.readline()
                                flag=0
                except Exception:print 'Error in reading file...'
                finally:
                      f.close()
                      if flag==0:
                           print 'Not found' #Inform the use if the record does not exist
                           return -1 # The record not found-->return -1
```

Using previous *search* method, write a program to delete a customer name from file.

```
#delete customer's information by name
def delete(Name):
     print search(Name)
     p=search(Name) # returns position of given customer name
     print "x=",p
     if p!=-1: #Make sure the record exists
          st=getall() #retrieve content about cutomer
          f=open(r'customerlist.txt','w')#open file for writing
          f.writelines(st)
          f.seek(p)
          f.write('*****')#write 5 starts to override the 5 characters of the name to
be  deleted
      else:
          print 'No record to delete'#Inform the use if the record does not exist
      f.close()
```

TWO MARKS QUESTION & ANSWER

1. **What is the use of modules in Python?**

   A Python module is a file that consists of Python code. It allows us to logically arrange related code and makes the code easier to understand and use. It defines functions, classes and variables. Python has many useful functions and resources in modules. Functions such as abs() and round() from_builtin module are always directly accessible in every Python code. But, the programmer must explicitly import other functions from the modules in which they are defined.

2. **Name the File-related modules in Python.**

   Python provides libraries / modules with functions that enable us to manipulate text files and binary files on file system. Using them we can create files, update their contents, copy, and delete files. The libraries are : os, os.path, and shutil. Here, os and os.path – modules include functions for accessing the filesystem shutil – module enables to copy and delete the files.

3. **Name few Python modules for Statistical, Numerical and Scientific computations.**

Module	Usage
numPy	provides an array/matrix type, and it is useful for doing computations on arrays
scipy	provides methods for doing numeric integrals and solving differential equations
pylab	generating and saving plots
matplotlib	managing data and generating plots

4. **Define Package.**

   A package is a hierarchical file directory structure that defines a single Python application environment that consists of modules, sub-packages, sub-subpackages, and so on. In another words, it is a collection of modules. When a package is imported, Python explores in list of directories on sys.path for the package subdirectory.

5. **Define Exception.**

   An exception is an error that occurs during execution of a program. It is also called as run time errors. Some examples of Python runtime errors:

   - division by zero
   - performing an operation on incompatible types

6. **How exceptions are handled in Python?**

   Python handle exceptions using "try-except" block. Exceptions that are caught in try blocks are handled in except blocks. If an error is encountered, a try block code execution is stopped and control transferred down to except block.

Syntax:

```
try:
    # statements
    break
except ErrorName:
    # handler code
```

7. **When the finally block is executed?**

   The code in the finally block will be executed regardless of whether an exception occurs and even if we exit the block using break, continue, or return.

Example:

```
try:
  age = int(input("Please enter your age: "))
except ValueError:
  print("Hey, that wasn't a number!")
finally:
  print("Goodbye!")
```

8. **List the steps involved in creating packages.**

   (1) Create a directory and name it with a package name.

   (2) Keep subdirectories (subpackages) and modules in it.

   (3) Create **init.py** file in the directory.

9. **What are command line arguments?**

   Command line arguments are the *arguments* passed into the program from the *command line.* sys.argv is a list in Python, which contains the command line arguments passed to the script. We can count the number of arguments using len(sys.argv) function. To use sys.argv, we have to import the sys module.

Example:

```
import sys
print 'No. of arguments:', len(sys.argv)
print 'Argument List:',str(sys.argv)
```

10. **Differentiate error and exception.**

    Errors or mistakes in a program are often referred to as bugs. They can be classified into three major groups: Syntax errors, Runtime errors, Logical errors. Whereas exception is an error that occurs during execution of a program. It is also called as run time errors.

11. How to catch all types of exception using single clause?

To catch all types of exceptions using single except clause, simply mention except keyword without specifying error name.

Example:

```
try:
    dividend = int(input("Please enter the dividend: "))
    divisor = int(input("Please enter the divisor: "))
    print("%d / %d = %f' % (dividend, divisor, dividend/divisor))
except:
    print("Oops, something went wrong!")
```

12. What is the use of *raise* statement?

The *raise* statement initiates a new exception specified by the programmer.

Example:

```
try:
    raise NameError
except NameError:
    print('Error')
```

13. What is the use of modules in Python?

A Python module is a file that consists of Python code. It allows us to logically arrange related code and makes the code easier to understand and use. It defines functions, classes and variables. Python has many useful functions and resources in modules. Functions such as abs() and round() from __builtin__ module are always directly accessible in every Python code. But, the programmer must explicitly import other functions from the modules in which they are defined.

14. Name the File-related modules in Python.

Python provides libraries / modules with functions that enable us to manipulate text files and binary files on file system. Using them we can create files, update their contents, copy, and delete files. The libraries are : os, os.path, and shutil. Here, os and os.path – modules include functions for accessing the filesystem shutil – module enables to copy and delete the files.

15. Name few Python modules for Statistical, Numerical and Scientific computations.

Module	**Usage**
numPy	provides an array/matrix type, and it is useful for doing computations on arrays

scipy	provides methods for doing numeric integrals and solving differential equations
pylab	generating and saving plots
matplotlib	managing data and generating plots

16. Define Package.

A package is a hierarchical file directory structure that defines a single Python application environment that consists of modules, sub-packages, sub-subpackages, and so on. In another words, it is a collection of modules. When a package is imported, Python explores in list of directories on sys.path for the package subdirectory.

17. Define Exception.

An exception is an error that occurs during execution of a program. It is also called as run time errors. Some examples of Python runtime errors:

- division by zero
- performing an operation on incompatible types

18. How exceptions are handled in Python?

Python handle exceptions using "try-except" block. Exceptions that are caught in try blocks are handled in except blocks. If an error is encountered, a try block code execution is stopped and control transferred down to except block.

Syntax:

```
try:
    # statements
    break
except ErrorName:
    # handler code
```

19. When the finally block is executed?

The code in the finally block will be executed regardless of whether an exception occurs and even if we exit the block using break, continue, or return.

Example:

```
try:
   age = int(input("Please enter your age: "))
except ValueError:
   print("Hey, that wasn't a number!")
finally:
   print("Goodbye!")
```

20. List the steps involved in creating packages.

(1) Create a directory and name it with a package name.

(2) Keep subdirectories (subpackages) and modules in it.

(3) Create **__init__.py** file in the directory.

21. What are command line arguments?

Command line arguments are the *arguments* passed into the program from the *command line.* sys.argv is a list in Python, which contains the command line arguments passed to the script. We can count the number of arguments using len(sys.argv) function. To use sys.argv, we have to import the sys module.

Example:

```
import sys
print ‘No. of arguments:’, len(sys.argv)
print ‘Argument List:’,str(sys.argv)
```

21. Differentiate error and exception.

Errors or mistakes in a program are often referred to as bugs. They can be classified into three major groups: Syntax errors, Runtime errors, Logical errors. Whereas exception is an error that occurs during execution of a program. It is also called as run time errors.

22. How to catch all types of exception using single clause?

To catch all types of exceptions using single except clause, simply mention except keyword without specifying error name.

Example:

```
try:
   dividend = int(input("Please enter the dividend: "))
   divisor = int(input("Please enter the divisor: "))
   print("%d / %d = %f" % (dividend, divisor, dividend/divisor))
except:
   print("Oops, something went wrong!")
```

23. What is the use of raise statement?

The *raise* statement initiates a new exception specified by the programmer.

Example:

```
try:
     raise NameError
except NameError:
     print(‘Error’)
```

24. Give all the file processing modes supported by Python.

Python allows you to open files in one of the three modes. They are: read-only mode, write-only mode, read-write mode, and append mode by specifying the flags “r”, “w”, “rw”, “a” respectively. A text file can be opened in any one of the above said modes by specifying the option “t” along with “r”, “w”, “rw”, and “a”, so that the preceding modes become “rt”, “wt”, “rwt”, and “at”.A binary file can be opened in any one of the above said modes by specifying the option “b” along with “r”, “w”, “rw”, and “a” so that the preceding modes become “rb”, “wb”, “rwb”, “ab”.

25. Explain the use of “with” statement.

In python generally “with” statement is used to open a file, process the data present in the file, and also to close the file without calling a close() method. “with” statement makes the exception handling simpler by providing cleanup activities.

General form of with:

```
with open(“file name”, “mode”) as file-var:
processing statements
note: no need to close the file by calling close() upon file-var.close()
```

26. Explain how to redirect the output of a python script from standout (monitor) on to a file?

They are two possible ways of redirecting the output from standout to a file.

1. Open an output file in “write” mode and the print the contents in to that file, using sys. stdout attribute.import sys

   filename = “outputfile” sys.stdout = open() print “testing”

2. You can create a python script say .py file with the contents, say print “testing” and then redirect it to the output file while executing it at the command prompt. Eg: redirect_output.py has the following code:

   print “Testing” execution: python redirect_output.py > outputfile.

27. Explain the shortest way to open a text file and its contents?

The shortest way to open a text file is by using “with” command as follows:

```
with open("file-name", "r") as fp:
fileData = fp.read()
#to print the contents of the file print(fileData)
```

28. What is the use of enumerate () in python?

Using enumerate() function you can iterate through the sequence and retrieve the index position and its corresponding value at the same time.

```
>>> for i,v in enumerate([‘Python’,’Java’,’C++’]):print(i,v)
    0 Python
```

1 Java

2 C++

29. How do you perform pattern matching in Python? Explain.

Regular Expressions/REs/ regexes enable us to specify expressions that can match specific "parts" of a given string. For instance, we can define a regular expression to match a single character or a digit, a telephone number, or an email address, etc.The Python's "re" module provides regular expression patterns and was introduce from later versions of Python 2.5. "re" module is providing methods for search text strings, or replacing text strings along with methods for splitting text strings based on the pattern defined.

30. Name few methods for matching and searching the occurrences of a pattern in a given text string?

There are 4 different methods in "re" module to perform pattern matching. They are: match() – matches the pattern only to the beginning of the String. search() – scan the string and look for a location the pattern matches findall() – finds all the occurrences of match and return them as a list

finditer() – finds all the occurrences of match and return them as an iterator.

31. Explain split(), sub() and subn() methods of Python.

To modify the strings, Python's "re" module is providing 3 methods. They are: split() – uses a regex pattern to "split" a given string into a list. sub() – finds all substrings where the regex pattern matches and then replace them with a different string subn() – it is similar to sub() and also returns the new string along with the no. of replacements.

32. How to display the contents of text file in reverse order?

1. convert the given file into a list.
2. reverse the list by using reversed()

Eg: for line in reversed(list(open("file-name","r"))):print(line)

33. What is JSON? How would convert JSON data into Python data?

JSON – stands for JavaScript Object Notation. It is a popular data format for storing data in NoSQL databases. Generally JSON is built on 2 structures.

1. A collection of <name, value> pairs.
2. An ordered list of values.

As Python supports JSON parsers, JSON-based data is actually represented as a dictionary in Python. You can convert json data into python using load() of json module.

34. What is TkInter?

TkInter is Python library. It is a toolkit for GUI development. It provides support for various GUI tools or widgets (such as buttons, labels, text boxes, radio buttons, etc) that are used in

GUI applications. The common attributes of them include Dimensions, Colors, Fonts, Cursors, etc.

35. Name and explain the three magic methods of Python that are used in the construction and initialization of custom objects.

The 3 magic methods of Python that are used in the construction and initialization of custom Objects are: init__, new, and del__.

new – this method can be considered as a "constructor". It is invoked to create an instance of a class with the statement say, myObj = MyClass () init__ — It is an "initializer"/ "constructor" method. It is invoked whenever any arguments are passed at the time of creating an object. myObj = MyClass('Pizza',25) del- this method is a "destructor" of the class. Whenever an object is deleted, invocation of del__ takes place and it defines behaviour during the garbage collection.

Note: new, del are rarely used explicitly.

36. What is Exception Handling? How do you achieve it in Python?

Exception Handling prevents the codes and scripts from breaking on receipt of an error at run -time might be at the time doing I/O, due to syntax errors, data types doesn't match. Generally it can be used for handling user inputs.

The keywords that are used to handle exceptions in Python are: try – it will try to execute the code that belongs to it. May be it used anywhere that keyboard input is required. except – catches all errors or can catch a specific error. It is used after the try block.x = 10 + 'Python' #TypeError: unsupported operand type(s) …. try:

x = 10 + 'Python'

except: print("incompatible operand types to perform sum")

raise – force an error to occur

o raise TypeError("dissimilar data types")

finally – it is an optional clause and in this block cleanup code is written here following "try" and "except".

37. Explain different ways to trigger/raise exceptions in your Python script.

The following are the two possible ways by which you can trigger an exception in your Python script. They are:

(1) raise — it is used to manually raise an exception general-form: raise exception-name ("message to be conveyed")

Eg: >>> voting_age = 15

>>> if voting_age < 18: raise ValueError("voting age should be atleast 18 and above") output: ValueError: voting age should be atleast 18 and above 2. assert statement assert statements are used to tell your program to test that condition attached to assert keyword, and trigger an exception whenever the condition becomes false. Eg: >>> a = -10 >>> assert a > 0 #to raise an exception whenever a is a negative number output: AssertionError Another

way of raising and exception can be done by making a programming mistake, but that's not usually a good way of triggering an exception.

38. How do you check the file existence and their types in python?

os.path.exists() – use this method to check for the existence of a file. It returns True if the file exists, false otherwise. Eg: import os; os.path.exists('/etc/hosts')

os.path.isfile() – this method is used to check whether the give path references a file or not. It returns True if the path references to a file, else it returns false. Eg: import os; os.path.isfile('/etc/hosts')

os.path.isdir() – this method is used to check whether the give path references a directory or not. It returns

True if the path references to a directory, else it returns false. Eg: import os; os.path.isfile('/etc/hosts')

os.path.getsize() – returns the size of the given file

os.path.getmtime() – returns the timestamp of the given path.