FILES, MODULES, PACKAGES

5

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 Files and Exception

- Python provides inbuilt functions for creating, writing and reading files.
- There are two types of files that can be handled in python, normal text files and binary files (written in binary language,0s and 1s).

  - **Text files:** In this type of file, Each line of text is terminated with a special character called EOL (End of Line), which is the new line character (‘\n’) in python by default.
  - **Binary files:** In this type of file, there is no terminator for a line and the data is stored after converting it into machine understandable binary language.

5.1.1 Text files

- A text file is a sequence of characters stored on a permanent medium like a hard drive, flash memory, or CD-ROM.
- To write a file, you have to open it with mode 'w' as a second parameter:

```
>>>fout = open('output.txt', 'w')
```

- If the file already exists, opening it in write mode clears out the old data and starts fresh, so be careful!
- If the file doesn’t exist, a new one is created.
- **open** returns a file object that provides methods for working with the file.
- The **write** method puts data into the file.

```
>>> line1 = "This here's the wattle,\n"
>>>fout.write(line1)
24
```

- The return value is the number of characters that were written.
- The file object keeps track of where it is, so if you call write again, it adds the new data to the end of the file.

```
>>> line2 = "the emblem of our land.\n"
>>>fout.write(line2)
24
```

- When you are done writing, you should close the file.

```
>>>fout.close()
```

5.1.2 Reading and writing files

Create a text file

- Using a simple text editor, let's create a file.
- You can give any name to the file. we are going to call it "testfile.txt".

```
file = open("testfile.txt","w")
 file.write("Hello World")
file.write("This is our new text file")
file.write("and this is another line.")
file.write("Why? Because we can.")
 file.close()
```

- Naturally, if you open the text file you will see only the text we told the interpreter to add.

```
$ cat testfile.txt
Hello World
This is our new text file
and this is another line.
Why? Because we can.
```

Reading a Text File in Python

- There are actually a number of ways to read a text file in Python, not just one.
- If you need to extract a string that contains all characters in the file, you can use the following method:

```
file.read()
```

- The full code to work with this method will look something like this:

```
file = open("testfile.text", "r")
printfile.read()
```

- The output of that command will display all the text inside the file, the same text we told the interpreter to add earlier.
- Another way to read a file is to call a certain number of characters.
- For example, with the following code the interpreter will read the first five characters of stored data and return it as a string:

```
file = open("testfile.txt", "r")
printfile.read(5)
```

- using the same *file.read()* method, only this time we specify the number of characters to process?

The output for this will look like:

```
Hello
```

- If you want to read a file line by line – as opposed to pulling the content of the entire file at once – then you use the *readline()* function.
- Each time you run the method, it will return a string of characters that contains a single line of information from the file.

```
file = open("testfile.txt", "r")
printfile.readline():
```

- This would return the first line of the file, like so:

Hello World

- If we wanted to return only the third line in the file, we would use this:

file = open("testfile.txt", "r")

printfile.readline(3):

- But what if we wanted to return every line in the file, You would use the same function, only in a new form.
- This is called the *file.readlines()* function.

file = open("testfile.txt", "r")

printfile.readlines()

- The output you would get from this is:

['Hello World', 'This is our new text file', 'and this is another line.', 'Why? Because we can.']

5.1.3 Reading Keyboard Input

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

- **raw_input**
- **input**

5.1.3.1 The *raw_input* Function

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

```
#!/usr/bin/python

str=raw_input("Enter your input: ");

print"Received input is : ",str
```

This prompts you to enter any string and it would display same string on the screen. When I typed "Hello Python!", its output is like this –

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

5.1.3.2 The *input* Function

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

```
#!/usr/bin/python

str= input("Enter your input: ");

print"Received input is : ",str
```

This would produce the following result against the entered input −

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

5.1.4 Opening and Closing Files

Until now, you have been reading and writing to the standard input and output. Now, we will see how to use actual data files.

Python provides basic functions and methods necessary to manipulate files by default. You can do most of the file manipulation using a **file** object.

5.1.4.1 The *open* Function

Before you can read or write a file, you have to open it using Python's built-in *open()* function. This function creates a **file**object, which would be utilized to call other support methods associated with it.

Syntax

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

Here are parameter details −

- **file_name** − Thefile_name argument is a string value that contains the name of the file that you want to access.
- **access_mode** − Theaccess_mode determines the mode in which the file has to be opened, i.e., read, write, append, etc. A complete list of possible values is given below in the table. This is optional parameter and the default file access mode is read (r).

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

Here is a list of the different modes of opening a file −

Sr.No.	Modes & Description
1	**r** Opens a file for reading only. The file pointer is placed at the beginning of the file. This is the default mode.
2	**rb** Opens a file for reading only in binary format. The file pointer is placed at the beginning of the file. This is the default mode.
3	**r+** Opens a file for both reading and writing. The file pointer placed at the beginning of the file.
4	**rb+** Opens a file for both reading and writing in binary format. The file pointer placed at the beginning of the file.
5	**w** Opens a file for writing only. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
6	**wb** Opens a file for writing only in binary format. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.

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

5.1.4.2. The *close()* Method

The close() method of a *file* object flushes any unwritten information and closes the file object, after which no more writing can be done.

Python automatically closes a file when the reference object of a file is reassigned to another file. It is a good practice to use the close() method to close a file.

Syntax

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

Example

```
#!/usr/bin/python
# Open a file
fo= open("foo.txt","wb")
print"Name of the file: ", fo.name

# Close opend file

fo.close()
```

This produces the following result −

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

5.1.5 The file Object Attributes

Once a file is opened and you have one file object, you can 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 file is closed, false otherwise
file.mode	Returns access mode with which file was opened
file.name	Returns name of the file
file.softspace	Returns false if space explicitly required with print, true otherwise

Example

```
#Open a file
F=open("text1.txt","wb")
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:text1.txt
Closed or not: False
Opening mode: wb
Softspace flag: 0
```

5.1.6 FILE MANIPULATIONS

5.1.6.1 File Positions

There are two major methods play the vital role, they are

(i) tell()
(ii) seek()

The **tell()** method tells you the current 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 changes the current file position. The offset argument indicates 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.

Example:

Let us take a file sam1.txt, which we created above:

```
#Open a file
fo=open("foo.txt","r+")
str=fo.read(10);
print "Read String is:",str
#Check current position
position=fo.tell();
print"Current file position:",position
```

```
#Reposition pointer at the beginning once again
position=fo.seek(0,0);
str=fo.read(10);
print "Again read String is:",str
#Close opened file
fo.close()
```

Output

```
Read String is : I will try
Current file position:10
Again read String is : I will try
```

5.1.6.2 Renaming and Deleting Files

Two file-processing operations are there, they are:

a) renaming
b) deleting or remove files

To use this module you need to import it first and then you can call any related functions.

a) 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

```
importos
#Rename a file from test1.txt to test2.txt
os.rename("test1.txt","test2.txt")
```

b) The remove() Method

This method is 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 test2.txt
importos
#delete file test2.txt
os.remove("text2.txt")
```

5.1.6.3 Directories in Python

All files are contained within various directories, and Python has to problem handling these too. The os module has several methods that help you create, remove, and change directories, they are:
a) mkdir()
b) chdir()
c) getcwd()
d) rmdir()

a) The mkdir() Method

This mkdir() method of the os module to create directories in the current directory. You need to supply an argument to this method which contains the name of the directory to be created.

Syntax:

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

Example:

```
Following is the example to create a directory test in the current directory-
importos
#Create a directory "test"
os.mkdir("test")
```

b) The chdir() Method

This chdir() method to change the current directory. The chdir() method takes an argument, which is the name of the directory that you want to make the current directory.

Syntax:

```
os.chdir("newdir")
```

Example:

Following is the example to go into "/home/newdir" directory

```
importos
#changing a directory to "/home/newdir"
os.chdir("/home/newdir")
```

c) The getcwd() Method

The getcwd() method displays the current working directory

Syntax:

```
os.getcwd()
```

Example

```
importos
#This would give location of the current directory
os.getcwd()
```

d) The rmdir() Method

This rmdir() method deletes the directory, which is passed as an argument in the method.

Before removing a directory, all the contents in it should be removed.

Syntax:

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

Example

To remove "/tmp/test" directory. It is required to give fully qualified name of the directory, otherwise it would search for that directory in the current directory.

```
importos
#this would remove "/tmp/test" directory.
os.rmdir("/tmp/test")
```

5.1.6.4. picking

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.

Example

```
# Without pickle
File1 = open("test.txt",'w')
#Write String
File1.write('hi')
# Write float
File1.write(str(56.789))
#Write dictionary
File1.write(str({'Age':23, 'Status': 'Single'}))
File1.close()
print ("Contents of file : test.txt")
File1 = open("test.txt", 'r')
print(File1.read())
File1.close()
```

Output:

```
$ python program.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

Example

```
#Use of pickle
import pickle
File1 = open("test.txt", 'w')
#Write String
pickle.dump('hi', File1)
#Write float
pickle.dump('56.789', File1)
#Write dictionary
pickle.dump({'Age':23, 'Status': 'Single'}, File1)
File1.close()
print ("Contents of file : test.txt")
File1 = open("test.txt", 'r')
print(pickle.load(File1))
File1.close()
```

Output

```
$ python program.py
Contents of file: test.txt
Hi
56.789
{'Status': 'Single', 'Age':23}
```

5.1.7 Format operator

- The argument of write has to be a string, so if we want to put other values in a file, we have to convert them to strings.
- The easiest way to do that is with str:

```
>>> x = 52
>>>fout.write(str(x))
```

- An alternative is to use the **format operator**, %. When applied to integers, % is the modulus operator.
- But when the first operand is a string, % is the format operator.
- The first operand is the **format string**, which contains one or more **format sequences**, which 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 a decimal integer:

```
>>> camels = 42
>>> '%d' % camels
'42'
```

- The result is the string '42', which is not to be confused with the integer value 42.
- A format sequence can appear anywhere in the string, so you can embed a value in a sentence:

```
>>> 'I have spotted %d camels.' % camels
'I have spotted 42 camels.'
```

- If there is more than one format sequence in the string, the second argument has to be a tuple.
- Each format sequence is matched with an element of the tuple, in order.
- The following example uses '%d' to format an integer, '%g' to format a floating-point number, and '%s' to format a string:

```
>>> 'In %d years I have spotted %g %s.' % (3, 0.1, 'camels')
'In 3 years I have spotted 0.1 camels.'
```

- The number of elements in the tuple has to match the number of format sequences in the string.
- Also, the types of the elements have to match the format sequences:

```
>>> '%d %d %d' % (1, 2)
TypeError: not enough arguments for format string
>>> '%d' % 'dollars'
TypeError: %d format: a number is required, not str
```

- In the first example, there aren't enough elements; in the second, the element is the wrong type.

5.1.8 Command line arguments

- Python provides a **getopt** module that helps you parse command-line options and arguments.

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

- The Python **sys** module provides access to any command-line arguments via the **sys.argv**.
- This serves two purposes −

  - sys.argv is the list of command-line arguments.
  - len(sys.argv) is the number of command-line arguments.

- Here sys.argv[0] is the program ie. script name.

Example

Consider the following script test.py −

```
#!/usr/bin/python

import sys

print'Number of arguments:',len(sys.argv),'arguments.'
print'Argument List:',str(sys.argv)
```

- Now run above script as follows

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

- This produce following result −

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

- As mentioned above, first argument is always script name and it is also being counted in number of arguments.

Parsing Command-Line Arguments

- Python provided a **getopt** module that helps you parse command-line options and arguments.
- This module provides two functions and an exception to enable command line argument parsing.

getopt.getopt method

- This method parses command line options and parameter list. Following is simple syntax for this method

```
getopt.getopt(args, options,[long_options])
```

Here is the detail of the parameters −

- **args**: This is the argument list to be parsed.

- **options**: This is the string of option letters that the script wants to recognize, with options that require an argument should be followed by a colon (:).
- **long_options**: This is optional parameter and if specified, must be a list of strings with the names of the long options, which should be supported. Long options, which require an argument should be followed by an equal sign ('='). To accept only long options, options should be an empty string.
- This method returns value consisting of two elements: the first is a list of **(option, value)** pairs. The second is the list of program arguments left after the option list was stripped.
- Each option-and-value pair returned has the option as its first element, prefixed with a hyphen for short options (e.g., '-x') or two hyphens for long options (e.g., '--long-option').

5.2.Errors and exceptions

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. The process of finding and eliminating errors is called debugging. Errors can be classified into three major groups:

a) **Syntax errors**
b) **Runtime errors**
c) **Logical errors**

a) Syntax errors:

Python will find these kinds of errors when it tries to parse your program, and exit with an error message without running anything. Syntax errors are mistakes in the use of the Python language, and are analogous to spelling or grammar mistakes in a language like English:

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

Python will do its best to tell you where the error is located, but sometimes its messages can be misleading. For example, if you forget to escape a quotation mark inside a string you may get a syntax error referring to a place later in your code, even though that is not the real source of the problem. If you can’t see anything wrong on the line specified in the error message, try

backtracking through the previous few lines. As you program more, you will get better at identifying and fixing errors.

Examples:

Syntax errors in Python:

```
myfunction(x,y):
        return x + y
else:
        print ("Hello!")
if mark >=50
        print("You passed!")
if arriving:
        print("Hi!")
else:
        print("Bye!")
if flag:
print("Flag is set!")
```

b) Runtime errors

If a program is syntactically correct that is, free of syntax errors it will be run by the Python interpreter. However, the program may exit unexpectedly during execution if it encounters a runtime error a problem which was not detected when the program was parsed, but is only revealed when a particular line is executed. When a program comes to a halt because of a runtime error, we say that it has crashed.

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.

Runtime errors often creep in if you don't consider all possible values that a variable could contain, especially when you are processing user input. You should always try to add checks to your code to make sure that it can deal with bad input and edge cases gracefully. We will look at this in more detail in the chapter about exception handling.

c) Logical errors

Logical errors are the most difficult to fix. They occur when the program runs without crashing, but produces an incorrect result. The error is caused by a mistake in the program's logic. You won't get an error message, because no syntax or runtime error has occurred. You will have to find the problem on your own by reviewing all the relevant parts of your code – although some tools can flag suspicious code which looks like it could cause unexpected behavior.

Examples:

- 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

If you misspell an identifier name, you may get a runtime error or a logical error, depending on whether the misspelled name is defined.

5.2.2 HANDLING EXCEPTIONS

An exception is an event, which occurs during the execution of a program that disrupts the normal flow of the program's instructions. In general, when a Python script encounters a situation that it cannot cope with, it raises an exception. An exception is a Python object that represents an error.

When a Python script raises an exception, it must either handle the exception immediately otherwise it terminates and quits.

Python provides two very important features to handle any unexpected error in the Python programs and to add debugging capabilities in them.

- Exception Handling
- Assertions

5.2.2.1 Standard Exceptions

Exception name	Description
Exception	Base class for all exceptions
StopIteration	Raised when the next() method of an iterator does not point to any object
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 or modulo by zero takes place for all numeric types
AssertionError	Raised in case of failure of the Assert statement
AttributeError	Raised in case of failure of attribute reference or assignment
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 the user interrupts program execution, usually by pressing Ctrl+c
LookupError	Base class for all lookup errors
IndexError KeyError	Raised when an index is not found in a sequence. 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
UnboundLocalError EnvironmentError	Raised when trying to access a local variable in a function or method but no value has been assigned to it. Base class for all exceptions that occur outside the Python environment
IOError 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. Raised for operating system-related errors
SyntaxError IndentationError	Raised when there is an error in Python syntax. 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.

5.2.2.2 Assertions

An assertion is a sanity-check that you can turn on or turn off when you are done with your testing of the program.

The easiest way to think of an assertion is to liken it to raise-if statement(or to be more accurate, a raise-if-not statement). An expression is tested, and if the result comes up false, an exception is raised.

Assertions are carried out by the assert statement, the newest keyword to Python introduced in version 1.5

Programmers often place assertions at the start of a function to check for valid input, and after a function call to check for valid output.

The assert Statement:

When it encounters an assert statement, Python evaluates the accompanying expression, which is hopefully true. If the expression is false, Python raises an AssertionError exception

Syntax:

Assert Expression(, Arguments)

If the assertion fails, Python uses ArgumentExpression as the argument for the AssertionError. AssertionError exceptions can be caught and handled like any other exception using the try-except statement, but if not handled, they will terminate the program and produce a traceback.

Example:

Here is a function that converts a temperature from degrees Kelvin to degrees Fahrenheit. Since zero degrees Kelvin is as cold as it gets, the function bails out if it sees a negative temperature.

```
defKelvinToFahrenheit(Temperature):
assert(Temperature>=0),"Colder than absolute zero!"
return((Temperature-273)*1.8)+32
printKelvinToFahrenheit(273)
printint(KelvinToFahrenheit(505.78))
printKelvinToFahrenheit(-5)
```

Output

```
32.0
451
Traceback (most recent call last):
File "test.py", line 9, in
printKelvinToFahrenheit
assert(Temperature >=0), "Colder than absolute zero!"
AssertionError: Colder than absolute zero!
```

5.2.2.3 LOGIC-HANDLING EXCEPTIONS

If you have some suspicious code that may raise an exception, you can defend your program by placing the suspicious code in a try: block. After the try: block, include an except: statement, followed by a block of code which handles the problem as elegantly as possible.

Syntax

```
try:
You do your operations here:
……..
except Exception1:
if there is Exception1, then execute this block
exceptExceptionII:
if there is ExceptionII, then execute this block.
…….
else:
if there is no exception then execute this block.
```

Important points about syntax:

- A single try statement can have multiple except statements. This is useful when the try block contains statements that they may throw different types of exceptions.
- You can also provide a generic except clause, which handles any exception.

- After the except clause(s), you can include an else-clause. The code in the else-block executes if the code in the try: block does not raise an exception.
- The else-block is a good place for code that does not need the try: block's protection.

Example 1

```
try:
fh=open("testfile","w")
fh.write("This is my test file for exception handling!!")
exceptIOError:
print "Error: can't open file in write mode
else:
print "Written content in the file successfully"
fh.close()
```

Output:
Written content in the file successfully.

Example 2:

This example tries to open a file where you do not have write permission, so it raises an exception:

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

Output:
Error: can't find file or read data.

The except Clause with No Exceptions

You can also use the except statement with no exceptions defines as follows:

```
try:
You do your operations here:
……
except:
If there is any exception, then execute this block.
……..
else:
if there is no exception then execute this block.
```

This kind of a try-except statement catches all the exceptions that occur. Using this kind of try-except statement is not considered a good programming practice though, because it catches all exceptions but does not make the programmer identify the root cause of the problem that may occur.

The except Clause with Multiple Exceptions

You can also use the same except statement to handle multiple exceptions as follows:

```
try:
You do your operations here:
……
except(Exception1(, Exception2(,…ExceptionN))):
if there is any exception from the given exception list,
then execute this block.
……..
else:
if there is no exception then execute this block.
```

The try-finally Clause

You can use a finally: block along with a try: block. The finally block is a place to put any code that must execute, whether the try-block raised an exception or not. The syntax of the try – finally statement is this.

```
try:
You do your operations here:
…….
Due to any exception, this may be skipped.
finally:
this would always be executed.
……
```

Note that you can provide except clause(s), or a finally clause, but not both. You cannot use else clause as well along with a finally clause.

Example:

```
try:
fh=open(“testfile”,”w”)
fh.write(“This is my test file for exception handling!!”)
```

```
finally:
print “Error: can’t find file or read data”
```

If you do not have permission to open the file in writing mode, then this will produce the following result:

Output:

Error: can’t find file or read data.

Example

```
try:
fh=open(“testfile”,”w”)
try:
fh.write(“This is my test file for exception handling!!”)
finally:
print “Going to close the file”
fh.close()
exceptIOError:
print “Error: can’t find file or read data”
```

When an exception is thrown in the try block, the execution immediately passes to the finally block. After all the statements in the finally block are executed, the exception is raised again and is handled in the except statements if present in the next higher layer of the try-except statement.

5.2.2.4. Argument of an Exception

An exception can have an argument, which is a value that gives additional information about the problem. The contents of the argument vary by exception. You can capture an exception’s argument by supplying a variable in the except clause as follows.

```
try:
You do your operation here.
……..
exceptExceptionType, Argument
```

You can print value of Argument here…

If you write the code to handle a single exception, you can have a variable follow the name of the exception in the except statement. If you are trapping multiple exceptions, you can have a variable follow the tuple of the exception.

The variable receives the value of the exception mostly containing the cause of the exception. The variable can receive a single value or multiple values in the form of a tuple. This tuple usually contains the error string, the error number, and an error location.

Example:

Following is an example for a single exception:
#Define a function here
def temp_convert(var):
try:
return int(var)
exceptValueError,Argument:
print “The argument does not contain numbers\n”, Argument
#Call abpve function here
temp_convert(“xyz”):
Output:
The argument does not contain numbers
invalid literal for int() with base 10: ‘xyz’

5.2.2.5. Raising an Exception

You can raise exceptions in several ways by using the raise statement.

The general

Syntax

raise(Exception(, args(, traceback)))

Here, Exception is the type of exception(for example, NameError) and argument is a value for the exception argument. The argument is optional: if not supplied, the exception argument is None.

The final argument, traceback, is also optional (and rarely used in practice), and if present, is the traceback object used for the exception.

Example:

An exception can be a string, a class or an object. Most of the exceptions that the Python core raises areclasses, with an argument that is an instance of the class. Defining new exceptions is quite easy and can be done as follows:
def functionName(level):
if level <1:
raise “Invalid level”, level
#the code below to this would not be executed
#if we raise the exception

Note:

In order to catch an exception, an "except" clause must refer to the same exception thrown either class object or simply string. For example, to capture above exception, we must write the except clause as follows

```
try:
Business logic here….
except "Invalid level!":
Exception handling here….
else:
Rest of the code here….
```

5.2.2.6.User-defined exceptions

Python also allows you to create your own exceptions by deriving classes from the standard built-in exceptions.

Here is an example related to RuntimeError. Here, a class is created that is subclassed from RunTimeError. This is useful when you need to display more specific information when an exception is caught.

In the try block, the user-defined exception is raised and caught in the except block.

The variable c is used to create an instance of the class Networkerror.

```
class Networkerror(RuntimeError):
def_init_(self,org):
self.args=arg
```

So once you defined above class, you can raise the exception as follows:

```
try:
raise Networkerror("Bad hostname")
exceptNetworkerror,e:
printe.args
```

Exception getopt.GetoptError

- This is raised when an unrecognized option is found in the argument list or when an option requiring an argument is given none.
- The argument to the exception is a string indicating the cause of the error. The attributes **msg** and **opt** give the error message and related option

Example

- Consider we want to pass two file names through command line and we also want to give an option to check the usage of the script. Usage of the script is as follows −

```
usage: test.py -i <inputfile>-o <outputfile>
```

Here is the following script to test.py −

```
#!/usr/bin/python

import sys,getopt

def main(argv):
inputfile="
outputfile="
try:
    opts,args=getopt.getopt(argv,"hi:o:",["ifile=","ofile="])
exceptgetopt.GetoptError:
print'test.py -i <inputfile> -o <outputfile>'
sys.exit(2)
for opt,argin opts:
if opt =='-h':
print'test.py -i <inputfile> -o <outputfile>'
sys.exit()
elif opt in("-i","--ifile"):
inputfile=arg
elif opt in("-o","--ofile"):
outputfile=arg
print'Input file is "',inputfile
print'Output file is "',outputfile

if __name__ =="__main__":
  main(sys.argv[1:])
```

- Now, run above script as follows –

```
$ test.py -h
usage: test.py -i <inputfile> -o <outputfile>

$ test.py -i BMP -o
usage: test.py -i <inputfile> -o <outputfile>

$ test.py -i inputfile
Input file is " inputfile
Output file is "
```

5.3 Modules

- A module is a file containing Python definitions and statements.
- The file name is the module name with the suffix .py appended.
- Within a module, the module's name (as a string) is available as the value of the global variable __name__.
- For instance, use your favorite text editor to create a file called fibo.py in the current directory with the following contents:

 # Fibonacci numbers module

```
deffib(n):# write Fibonacci series up to n
a,b=0,1
whileb<n:
print(b,end=' ')
a,b=b,a+b
print()
```

```
deffib2(n):# return Fibonacci series up to n
result=[]
a,b=0,1
whileb<n:
result.append(b)
a,b=b,a+b
returnresult
```

- Now enter the Python interpreter and import this module with the following command:

```
>>>importfibo
```

- This does not enter the names of the functions defined in fibo directly in the current symbol table; it only enters the module name fibo there.
- Using the module name you can access the functions:

```
>>>fibo.fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>>fibo.fib2(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>>fibo.__name__
```

```
'fibo'
```

- If you intend to use a function often you can assign it to a local name:

```
>>>fib=fibo.fib
>>>fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

- Modules are used to categorize code in Python into smaller part.
- A module is simply a file, where classes, functions and variables are defined.
- Grouping similar code into a single file makes it easy to access.
- If the content of a book is not indexed or categorized into individual chapters, then the book might have turned boring and hectic.
- Hence, dividing book into chapters made it easy to understand.
- In the same sense python modules are the files which have similar code.
- Thus module is simplify a python code where classes, variables and functions are defined.

Advantage:

Python provides the following advantages for using module:

1) Reusability: Module can be used in some other python code. Hence it provides the facility of code reusability.

2) Categorization: Similar type of attributes can be placed in one module.

5.3.1 Writing modules

- Any file that contains Python code can be imported as a module.
- For example, suppose you have a file named wc.py with the following code:

```
deflinecount(filename):
count = 0
for line in open(filename):
count += 1
return count
print(linecount('wc.py'))
```

- If you run this program, it reads itself and prints the number of lines in the file, which is 7.

- You can also import it like this:

```
>>> import wc
7
```

- Now you have a module object wc:

```
>>>wc
<module 'wc' from 'wc.py'>
```

- The module object provides linecount:

```
>>>wc.linecount('wc.py')
7
```

- So that's how you write modules in Python.

5.3.2 Importing a Module

- There are different ways by which you we can import a module. These are as follows:

1) Using import statement:

"import" statement can be used to import a module.

Syntax:

```
import <file_name1, file_name2,...file_name(n)="">
</file_name1,>
```

Example:

```
def add(a,b):
    c=a+b
    print c

    return
```

- Save the file by the name addition.py. To import this file "import" statement is used.

```
import addition
addition.add(10,20)
addition.add(30,40)
```

- Create another python file in which you want to import the former python file.
- For that, import statement is used as given in the above example.
- The corresponding method can be used by file_name.method ().
- (Here, addition. add (), where addition is the python file and add () is the method defined in the file addition.py)

```
Output:
>>>
30
70
>>>
```

- You can access any function which is inside a module by module name and function name separated by dot.
- It is also known as period.
- Whole notation is known as dot notation.
- Example of importing multiple modules:

Example:

1) msg.py:

```
def msg_method():
   print "Today the weather is rainy"
   return
```

2) display.py:

```
def display_method():
   print "The weather is Sunny"
   return
```

3) multiimport.py:

```
import msg,display
msg.msg_method()
display.display_method()
```

Output:

```
>>>
Today the weather is rainy
The weather is Sunny
>>>
```

2) Using from..import statement:

- from..import statement is used to import particular attribute from a module.
- In case you do not want whole of the module to be imported then you can use from ?import statement.

Syntax:

```
from  <module_name> import <attribute1,attribute2,attribute3,...attributen>
</attribute1,attribute2,attribute3,...attributen></module_name>
```

Example:

area.py

Eg:

```
def circle(r):
    print 3.14*r*r
    return

def square(l):
    print l*l
    return

def rectangle(l,b):
    print l*b
    return

def triangle(b,h):
    print 0.5*b*h
    return
```

area1.py

```
from area import square,rectangle
square(10)
rectangle(2,5)
```

Output:

```
>>>
100
10
>>>
```

3) To import whole module:

- You can import whole of the module using "from? import *"

Syntax:

```
from <module_name> import *
</module_name>
```

- Using the above statement all the attributes defined in the module will be imported and hence you can access each attribute.

1) area.py

Same as above example

2) area1.py

```
from area import *
square(10)
rectangle(2,5)
circle(5)
triangle(10,20)
```

Output:

```
>>>
100
10
78.5
100.0
>>>
```

Built in Modules in Python:

- There are many built in modules in Python.
- Some of them are as follows:math, random , threading , collections , os , mailbox , string , time , tkinter etc..
- Each module has a number of built in functions which can be used to perform various functions.

1) math:

- Using math module , you can use different built in mathematical functions.

Functions:

Function	Description
ceil(n)	Returns the next integer number of the given number
sqrt(n)	Returns the Square root of the given number.
exp(n)	Returns the natural logarithm e raised to the given number
floor(n)	Returns the previous integer number of the given number.
log(n,baseto)	Returns the natural logarithm of the number.
pow(baseto, exp)	Returns basetoraised to the exp power.
sin(n)	Returns sine of the given radian.
cos(n)	Returns cosine of the given radian.
tan(n)	Returns tangent of the given radian.

Useful Example of math module:

Example:

```
import math
a=4.6
print math.ceil(a)
print math.floor(a)
b=9
print math.sqrt(b)
print math.exp(3.0)
print math.log(2.0)
print math.pow(2.0,3.0)
print math.sin(0)
print math.cos(0)
```

```
print math.tan(45)
```

Output:

```
>>>
5.0
4.0
3.0
20.0855369232
0.69314718056
8.0
0.0
1.0
1.61977519054
>>>
```

Constants:

- The math module provides two constants for mathematical Operations:

Constants	Descriptions
Pi	Returns constant ? = 3.14159...
ceil(n)	Returns constant e= 2.71828...

Example:

```
import math
print math.pi
print math.e
```

Output:

```
>>>
3.14159265359
2.71828182846
>>>
```

2) random:

- The random module is used to generate the random numbers. It provides the following two built in functions:

Function	Description
random()	It returns a random number between 0.0 and 1.0 where 1.0 is exclusive.
randint(x,y)	It returns a random number between x and y where both the numbers are inclusive.

Example

```
import random
print random.random()
print random.randint(2,8)
```

Output:

```
>>>
0.797473843839
7
>>>
```

5.3.3 The dir() Function

- The built-in function dir() is used to find out which names a module defines. It returns a sorted list of strings.

```
>>>
>>>importfibo,sys
>>>dir(fibo)
['__name__', 'fib', 'fib2']
>>>dir(sys)
['__displayhook__', '__doc__', '__egginsert', '__excepthook__',
 '__loader__', '__name__', '__package__', '__plen', '__stderr__',
 '__stdin__', '__stdout__', '_clear_type_cache', '_current_frames',
```

```
'_debugmallocstats', '_getframe', '_home', '_mercurial', '_xoptions',
'abiflags', 'api_version', 'argv', 'base_exec_prefix', 'base_prefix',
'builtin_module_names', 'byteorder', 'call_tracing', 'callstats',
'copyright', 'displayhook', 'dont_write_bytecode', 'exc_info',
'excepthook', 'exec_prefix', 'executable', 'exit', 'flags', 'float_info',
'float_repr_style', 'getcheckinterval', 'getdefaultencoding',
'getdlopenflags', 'getfilesystemencoding', 'getobjects', 'getprofile',
'getrecursionlimit', 'getrefcount', 'getsizeof', 'getswitchinterval',
'gettotalrefcount', 'gettrace', 'hash_info', 'hexversion',
'implementation', 'int_info', 'intern', 'maxsize', 'maxunicode',
'meta_path', 'modules', 'path', 'path_hooks', 'path_importer_cache',
'platform', 'prefix', 'ps1', 'setcheckinterval', 'setdlopenflags',
'setprofile', 'setrecursionlimit', 'setswitchinterval', 'settrace',
'stderr', 'stdin', 'stdout', 'thread_info', 'version', 'version_info',
'warnoptions']
```

- Without arguments, dir() lists the names you have defined currently:

```
>>>a=[1,2,3,4,5]
>>>importfibo
>>>fib=fibo.fib
>>>dir()
['__builtins__', '__name__', 'a', 'fib', 'fibo', 'sys']
```

- Note that it lists all types of names: variables, modules, functions, etc.
- dir() does not list the names of built-in functions and variables. If you want a list of those, they are defined in the standard module builtins:

```
>>>importbuiltins
>>>dir(builtins)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException',
 'BlockingIOError', 'BrokenPipeError', 'BufferError', 'BytesWarning',
 'ChildProcessError', 'ConnectionAbortedError', 'ConnectionError',
 'ConnectionRefusedError', 'ConnectionResetError', 'DeprecationWarning',
 'EOFError', 'Ellipsis', 'EnvironmentError', 'Exception', 'False',
 'FileExistsError', 'FileNotFoundError', 'FloatingPointError',
 'FutureWarning', 'GeneratorExit', 'IOError', 'ImportError',
 'ImportWarning', 'IndentationError', 'IndexError', 'InterruptedError',
 'IsADirectoryError', 'KeyError', 'KeyboardInterrupt', 'LookupError',
 'MemoryError', 'NameError', 'None', 'NotADirectoryError', 'NotImplemented',
 'NotImplementedError', 'OSError', 'OverflowError',
 'PendingDeprecationWarning', 'PermissionError', 'ProcessLookupError',
 'ReferenceError', 'ResourceWarning', 'RuntimeError', 'RuntimeWarning',
 'StopIteration', 'SyntaxError', 'SyntaxWarning', 'SystemError',
```

```
'SystemExit', 'TabError', 'TimeoutError', 'True', 'TypeError',
'UnboundLocalError', 'UnicodeDecodeError', 'UnicodeEncodeError',
'UnicodeError', 'UnicodeTranslateError', 'UnicodeWarning', 'UserWarning',
'ValueError', 'Warning', 'ZeroDivisionError', '_', '__build_class__',
'__debug__', '__doc__', '__import__', '__name__', '__package__', 'abs',
'all', 'any', 'ascii', 'bin', 'bool', 'bytearray', 'bytes', 'callable',
'chr', 'classmethod', 'compile', 'complex', 'copyright', 'credits',
'delattr', 'dict', 'dir', 'divmod', 'enumerate', 'eval', 'exec', 'exit',
'filter', 'float', 'format', 'frozenset', 'getattr', 'globals', 'hasattr',
'hash', 'help', 'hex', 'id', 'input', 'int', 'isinstance', 'issubclass',
'iter', 'len', 'license', 'list', 'locals', 'map', 'max', 'memoryview',
'min', 'next', 'object', 'oct', 'open', 'ord', 'pow', 'print', 'property',
'quit', 'range', 'repr', 'reversed', 'round', 'set', 'setattr', 'slice',
'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'vars',
'zip']
```

5.4 Packages

- A Package is simply a collection of similar modules, sub-packages etc..

Steps to create and import Package:

1) Create a directory, say Info

2) Place different modules inside the directory. We are placing 3 modules msg1.py, msg2.py and msg3.py respectively and place corresponding codes in respective modules. Let us place msg1() in msg1.py, msg2() in msg2.py and msg3() in msg3.py.

3) Create a file __init__.py which specifies attributes in each module.

4) Import the package and use the attributes using package.

Have a look over the example:

1) Create the directory:

```
import os
os.mkdir("Info")
```

2) Place different modules in package: (Save different modules inside the Info package)

msg1.py

```
def msg1():
   print "This is msg1"
```

msg2.py

```
def msg2():
   print "This is msg2"
```

msg3.py

```
def msg3():
   print "This is msg3"
```

3) Create __init__.py file:

```
from msg1 import msg1
from msg2 import msg2
from msg3 import msg3
```

4)Import package and use the attributes:

```
import Info
Info.msg1()
Info.msg2()
Info.msg3()
```

Output:

```
>>>
This is msg1
This is msg2
This is msg3
>>>
```

- What is __init__.py file? __init__.py is simply a file that is used to consider the directories on the disk as the package of the Python.
- It is basically used to initialize the python packages.

5.5 Illustrative programs

5.5.1 Program for Word Count

```
defwordCount():
wordstring = 'it was the best of times it was the worst of times '
wordstring += 'it was the age of wisdom it was the age of foolishness'

   wordlist = wordstring.split()

wordfreq = []                          # create an empty list
   for w in wordlist:
wordfreq.append(wordlist.count(w))

   print("String\n" + wordstring +"\n")
   print("List\n" + str(wordlist) + "\n")
   print("Frequencies\n" + str(wordfreq) + "\n")
   print("Pairs\n" + str(zip(wordlist, wordfreq)))

print wordCount()
```

OUTPUT:

```
String
it was the best of times it was the worst of times it was the age of wisdom it was the age of foolis
hness

List
['it', 'was', 'the', 'best', 'of', 'times', 'it', 'was', 'the', 'worst', 'of', 'times', 'it', 'was', 'the', 'age', 'of', 'wis
dom', 'it', 'was', 'the', 'age', 'of', 'foolishness']

Frequencies
[4, 4, 4, 1, 4, 2, 4, 4, 4, 1, 4, 2, 4, 4, 4, 2, 4, 1, 4, 4, 4, 2, 4, 1]

Pairs
[('it', 4), ('was', 4), ('the', 4), ('best', 1), ('of', 4), ('times', 2), ('it', 4), ('was', 4), ('the', 4), ('worst', 1),
 ('of', 4), ('times', 2), ('it', 4), ('was', 4), ('the', 4), ('age', 2), ('of', 4), ('wisdom', 1), ('it', 4), ('was', 4)
, ('the', 4), ('age', 2), ('of', 4), ('foolishness', 1)]
```

5.5.2 Program for copy file

- First create a file **in.txt** with some contents

```
defcopyFile():
   with open('C:\Users\Python\Desktop\in.txt') as f:
      with open('C:\Users\Python\Desktop\out.txt',"w") as f1:
         for line in f:
            f1.write(line)
copyFile()
```

OUTPUT:

The contents from the file **in.txt** will be copied to the file **out.txt**

Part A

1. What are the types of files that can be handles in python?

There are two types of files that can be handled in python.

1. Text files

2. Binary files

2. Define text file.

A test file is a sequence of characters stored on a permanent medium like a hard drive, flash memory or CD-ROM.

3. What is mode of the file and its types?

The mode tells the interpreter and developer which way the file will be used.

The modes are:

'r' – Read mode

'w' – Write mode

'a' – Append mode

'r+' – Special read and write mode

4. Define module.

A module is a file containing Python definitions and statements.

The file name is the module name with the suffix .py appended.

Within a module, the module's name(as a string) is available as the value of the global variable_name_.

Modules are used to categorize code in python into smaller part.

A module is a Python object with arbitrarily named attributes that you can bind and reference.

Simply, a module is a file consisting of Python code. A module can define functions, classes and variables.

A module can also include runnable code.

5. What are the advantages for using module?

1. Reusability

2. Categorization

6. What are the ways a module can be imported?

1. Using import statement

2. Using from..import statement

7. Mention the built-in modules in python

There are many built-in modules in python.

Some of them are as follows: math, random, threading, collections, os, mailbox, string, time, tkinter, etc.

Each module has a number of built-in functions which can be used to perform various functions.

8. Define package.

A package is simply a collection of similar modules, sub packages etc..

9. What are the steps involved to create and import package?

1. Create a directory, say Info

2. Place different modules inside the directory.

3. Create a file_init_.py

4. Import the package and use the attributes using package.

10. How to locate the modules in python?

When you import a module, the Python interpreter searches for the module in the following sequences −

- The current directory.
- If the module isn't found, Python then searches each directory in the shell variable PYTHONPATH.
- If all else fails, Python checks the default path. On UNIX, this default path is normally /usr/local/lib/python/.

The module search path is stored in the system module sys as the **sys.path** variable. The

sys.pathvariable contains the current directory, PYTHONPATH, and the installation-dependent default.

11. What are the advantages of files?

1. When the data is stored in a file, it is stored permanently.

2. The files in the data can bbeutilizzed as and wwhen required.

3. It is possible to update the data.

4. Files are highly useful to store huge amount of data.

12. Write the syntax for write() method and read() method.

fileObject.write(string)

fileObject.read([count])

13. Define syntax errors.

Syntax errors, also known as parsing errors, are perhaps the most common kind of complaint you get while you are still learning Python.

>>> while True print ('Hello Python')

Syntax Error: invalid syntax

>>>

14. Mention the order for the search in python module.

1. The current directory.

2. PYTHONPATH(an environment variable with a list of directory).

3. The installation dependent default directory.

15. Define package.

A package is a directory that contains modules. Having a directory of modules allows us to have modules contained within other modules.This allows us to use qualified module names, clarifying the organization of our software.

16. Define Namespaces.

Variables are names or identifiers that map to objects. A namespace is a dictionary of variable names/keys and their corresponding objects values. Each function has its own local namespace.

17. Write the syntax for the rename() method and the remove() method

os.rename(current_file_name, new_file_name)

os.remove(file_name)

18. Mention the attributes related to file object.

File.closed

file.mode

file.name

file.softspace

Part B

1. What is a Files. Explain in detail
2. What are exception
3. Discuss the various file operations in detail with example.
4. Define command line arguments. Discuss briefly abot the same.
5. define errors and discuss briefly.
6. Define exceptions and discuss briefly.
7. explain handling exceptions in detail.
8. explain modules in detail with suitable examples
9. Discuss about the packages in python.
10. Illustrative programs: word count
11. Illustrative programs: copy file.

Glossary

persistent: Pertaining to a program that runs indefinitely and keeps at least some of its data in permanent storage.

format operator: An operator, %, that takes a format string and a tuple and generates a string that includes the elements of the tuple formatted as specified by the format string.

format string: A string, used with the format operator, that contains format sequences.

format sequence: A sequence of characters in a format string, like %d, that specifies how a value should be formatted.

text file: A sequence of characters stored in permanent storage like a hard drive.

directory: A named collection of files, also called a folder.

path: A string that identifies a file.

relative path: A path that starts from the current directory.

absolute path: A path that starts from the topmost directory in the file system.

catch: To prevent an exception from terminating a program using the try and except statements.

database: A file whose contents are organized like a dictionary with keys that correspond to values.

bytes object: An object similar to a string.

shell: A program that allows users to type commands and then executes them by starting other programs.

pipe object: An object that represents a running program, allowing a Python program to run commands and read the results.