UNIT 2

FILES, EXCEPTIONS AND MODULES

PART-A
SHORT QUESTIONS WITH SOLUTIONS

Q1. What are built-in functions for files?

Answer :

The file functions are built-in and provides an interface for performing file I/O operations. There are two built-in file functions,

1. open( )
2. file( ).

1. open( )

This function is used to open a file for reading or writing. Its syntax is shown below:

file_object = open(file_name, access_mode = 'w', buffering = –1)

2. file( ) or file( ) Factory Function

The file( ) function is exactly same as open( ) function. Both of them perform the same action and one can be substituted in place of other without any side effects.

At the time of python 2.2 release, all built-in types that did not have respective built-in functions, were given factory functions. The 'file' is one such built-in type that was not associated with any built-in function. As a result file( ) factory function was provided.

Usually, open( ) function is used, for performing read/write operation on file. Whereas file( ) function is used when dealing with file objects.

Q2. Define the file system of python.

Answer :

Model Paper-I, Q1(c)

The file system of python consists of several OS modules to perform tasks related to files and directory. The OS also serves as a front end module to a pure OS-dependent module. It avoids direct usage direct usage of OS dependent modules by loading the required module for the operating system being installed.

The methods supported by the OS modules can be put into three categories. They are as follows,

1. File processing methods
2. Directory methods
3. Permissions methods.

Q3. What are the file built-in attributes?

Answer :

File objects have file attributes other than file methods. They contain auxiliary data of file objects such as filename, file mode and a flag that indicates whether an additional space is required to be displayed before any successive data items. Various attributes of file objects are as follows,

Attribute	Description
file.name	It is the name of file.
file.mode	It is the access mode using which file is opened.
file.encoding	It is a type of encoding to convert the unicode strings into byte strings.
file.closed	It returns true if file is closed, otherwise false is returned.

Q4. What are standard files?

Model Paper-I, Q1(d)

Answer :

The standard files are used for read or writing data. They are preopened and allow access of data through handlers when they are available to the users. These file handles are provided by signs module in Python. There are three types of standard files in Python, they are as follows,

1. **stdin:** The function raw_input() receives input from sys.stdin:
2. **stdout**: The print statement outputs to sys.stdout.
3. **stderr:** It holds the error information that is to be forwarded to user.

Q5. List some of the modules related to files.

Answer :

The modules related to files and input/output are as follows,

Functions	Description
csv	It allows access to comma separated value files
bz2	It allows access to BZ2 compressed files
base6u	It is encoding/decoding of binary strings to/from text strings
binascii	It is encoding/decoding of binary and ASCII encoded binary strings
filecmp	It compares directories and files.
tarfile	It reads and writes the TAR archive files
fileinput	It iterates over lines of multiple input text files

Q6. What is exception handling in python?

Answer :

Model Paper-II, Q1(c)

An Exception is a run-time error, that halts the execution of the program. It is also described as an action, that is taken outside the normal flow of control by python interpreter or programmer, because of a runtime error. It is a two-phase process. First phase happens, when a phenomenon called 'Exception condition' or 'Exceptional condition' occurs. Whenever, an interpreter detect an error, it initiates something called 'raising an exceptions'. Though the interpreter immediately stops the flow of current execution, it intimates that something is wrong and that needs to be rectified raising an exception is also known as, 'triggering', 'throwing' or 'generating'.

In second phase, actual exceptional handling takes place. Exceptions can be handled either by the python interpreter or the programmer. Once an exceptional handling is initiated, programmer dictates an interpreter about the course-of-action to be taken, in order to contain the errors. This course-of-action could be taking some corrective measures, logging the error, ignoring the error (or) aborting the program itself. The actions to subside the errors are customized, according to the nature and severity of the error.

'try_except' statement is used by Python, to handle exceptions. It also allow one to detect exceptions. try_except statement, consists of code used to monitor for exceptions. It also provides a mechanism to handle exceptions.

Q7. Define try-finally statement.

Answer :

Model Paper-III, Q1(d)

We can use finally clause with the try block try-finally statement is used, to execute some part of the code, even there exists some errors. In other ways, we can say that, it is used to maintain the consistent behaviour irrespective of the occurrence of exceptions.

The code which is to be executed is placed in finally block. If an exception occurs while executing the code in the try block, then the control jumps to the finally clause the code in finally gets executed i.e., when an exception is raised in the try block, the statements below the control are not executed and the control immediately jumps to the finally block. On completion of finally block, the exception is reraised for handling errors. The syntax of the try-finally statement is given below,

Syntax

```
try:
   try_suite
finally:
   finally_suite
```

Define,

(a) Name error

(b) Index error

Answer :

(a) **Name Error**

Python interpreter uses 'NameError', to inform a user that the requested variable is not present in any of the namespaces, perhaps it may not be declared. When programmer calls an object, interpreter will search for it in all the namespaces. If it does not find it, then NameError is generated.

Example

```
>>> x
Traceback(outtermost first):
File"<stdin>", line1, out?
NameError : name 'x' is not defined.
```

(b) **IndexError**

Indexes are accessed, within a range when a user tries to access an index, outside the valid range of a sequence, 'IndexError' is generated by the Python interpreter.

Example

```
>>>aList = [ ]
>>>aList[5]
Traceback(outtermost first):
File "<stdin>, line1, out?
IndexError : list index out of range.
```

Q9. What is a module in python?

Model Paper-III, Q1(c)

Answer :

A Module stores pieces of python code, which can be shared by other python programs. It is a mechanism, that is used to organize the code logically. Lengthy codes can be divided into several independent modules, that can interact with each other.

The code present in a module, may be a single class with its methods and data variables or a group of related classes and functions. A module can also share or use the code present in some other module. This is done, by importing a module. Hence, modules provide code reusability.

Q10. List out four features of modules.

Model Paper-II, Q1(d)

Answer :

The other features of modules are listed as follows,

1. **Auto-loaded Modules**

The modules get loaded by interpreter for system use when it begins in standard mode. All these are maintained in sys. modules. The names are keys and locations are values.

2. **Preventing Attribute Import**

If module attributes are not needed to be loaded while importing module with "from module import" then prepend under score to attribute names.

3. **Case-insensitive Import**

The imports are by default case_sensitive. To make them work properly, an environment variable with name PYTHONCASEOK is defined.

4. **Source Code Encoding**

Python allows to create module file in native encoding rather than 7-bit ASCII.

```
#!/usr/bin/env python
#_*_coding:UTF _8_*_
```

PART-B

ESSAY QUESTIONS WITH SOLUTIONS

2.1 FILES

2.1.1 File Objects

Q11. Give a detail description about file objects.

Answer : Model Paper-I, Q4(a)

File Objects in Python

File objects in python are used to read and/or write data to a file. They indicate a connection to a file on the disk. Since file is a built-in type, the need to import module is not required.

Writing to a File

A file must be created first before writing to it. This is done by creating a file object and notifying python that it is ready to be written. The following statement creates a simple file.

```
>>>my_file = file(path, "w")
```

Here, the first argument represents the path where the file is created. The argument "w" tells the python that the file is created for writing purpose. If it is not mentioned the python assumes the file is created for reading purpose.

Data into the file is written using the following statement.

```
>>>my_file.write("Python Programming\n");
```

The line breaks in python are added by the programmer itself using the escape sequence "\n". The text can be appended by using the write method again. If the string data exceeds one line then it can be continued in the next lines too.

Example

```
>>>my_file.write(" " "
... Python is a programming language
... It is easy and user-friendly
... It has many methods and
... Functions in it.
... " " ")
```

Line breaks can be added in between a multi-line string. The data can be printed using the print statement as follows,

```
>>>print>>my_file, "End of the file".
```

The output prints >>> as the prompt while >> adds the output to the file to suppress the line break, terminate the print statement by comma. After writing the text, the file must be closed by deleting the file object. This can be done by the following statement.

```
>>> del my_file.
```

Reading From a File

In order to read a file, a file object must be opened first then "r" argument must be specified. Which tells python that the file has been opened for the reading purpose.

The following statement opens a file for reading.

```
>>>input = file(path, "r")
```

Here, path specifies the location to the file that is to be read. An exception is raised if the file is not found. The method "readline" reads the date one line at a time.

```
>>>input.readline( )
"Python Programming\n"
```

The newline character is returned at the end of the string. The read method can also read the entire file at once. This method returns the content which has not been read yet. Consider the following statement.

```
>>>text = input.read( )
>>>print text
```

This outputs the following,

"Python is a programming language. It is easy and user-friendly. It has many methods and functions in it".

The newline characters are printed as line breaks since print statement has been used. The file must be closed after reading.

The following statement deletes the file.

```
>>> del input
```

2.1.2 File Built-in Functions, File Built-in Methods, File Built-in Attributes

Q12. Explain built-in function and methods for files.

Answer :

File Built-in Functions

The file functions are built-in and provides an interface for performing file I/O operations. There are two built-in file functions,

1. open( )
2. file( ).

1. **open( )**

This function is used to open a file for reading or writing. Its syntax is shown below:

file_object = open(file_name, access_mode = 'w', buffering = –1)

Parameters

- **file_name :** A string value that specifies the name of the file to open. It may be relative or absolute pathname.
- **access_mode :** It specifies the mode in which the file has to be opened. Python defines several access modes. For example, 'w' for writing, 'r' for reading, 'a' for append and so on. This is an optional parameter and if it is not used, the default access mode i.e., "r" will be used. The table below shows various access modes for file objects.

Mode	Action
r	Opens the file for read operation.
rU	Opens the file for read operation with support for universal NEWLINE.
w	Opens file for performing write operation. If file already exist, it is truncated before performing write.
a	Opens file for performing append operation. Data is written from the EOF.
r+	Opens the file for performing both read and write operation.
w+	Opens file for performing both read and write. Note that during write operation it behaves like 'w' mode.
a+	Opens file for performing both read and write. Note that during write operation it behaves like 'a' mode.
rb	Opens file for performing binary read.
wb	Opens file for performing binary write.
ab	Opens file for performing binary append.
rb+	It is similar to r+ mode, but treats the file as binary file.
wb+	It is similar to w+ mode, but treats the file as binary file.
ab+	It is similar to a+ mode, but treats the file as binary file.

Table: Access Modes for File open( ) Function

❖ **Buffering :** It specifies the type of buffering that should be performed. It is an optional argument. The following are various values that can be used:

Value	Action
0	No buffering
1	Line buffering
–1	System default buffering
x	Buffered I/O with
where, x > 1	Buffer size = x

Return Value

The open( ) function returns a file object, if file was opened successfully, otherwise an IOError is generated.

2. File( ) or file( ) Factory Function

The file( ) function is exactly same as open( ) function. Both of them perform the same action and one can be substituted in place of other without any side effects.

At the time of python 2.2 release, all built-in types that did not have respective built-in functions, were given factory functions. The 'file' is one such built-in type that was not associated with any built-in function. As a result file( ) factory function was provided.

Usually, open( ) function is used, for performing read/write operation on file. Whereas file( ) function is used when dealing with file objects.

File Methods

The file methods are used for reading, writing and moving within the file. File methods are categorized into the following categories.

(a) Input

The following are various input functions.

(i) read( )

This method reads bytes from a file and stores into a string. Its syntax is,

file.read(size)

The parameter 'size', specifies the number of bytes to read. If 'size' is not specified, the default i.e., '–1' will be used, in which case the file will be read to the end.

(ii) readline( )

This method, reads one line of the file. A line consists of several bytes and a line terminating character at the end. This method reads the line and returns all bytes and the line terminating character as a string.

There is also an optional 'size' parameter to this function, which specifies the number of bytes to read. If 'size' is not specified, the default i.e., –1 is assumed.

Syntax : file.readline(size)

(iii) readlines( )

This method, reads the remaining lines from the file and returns them in the form of a list containing strings. It accepts an optional parameter named 'sizeint', which specifies the maximum number of bytes to be read in whole line.

Syntax : file.readlines(sizeint = 0)

(iv) Xreadlines( )

This method, reads file as chunks of bytes instead of all lines at a time. It allows iterating over a set of lines in an efficient manner than readlines( ) method.

Syntax : file.Xreadlines( )

(v) readinto( )

This method, reads the specified number of bytes and stores it into a writable buffer object.

Syntax : file.readinto(buffer, size).

(b) Output

The following are the various output methods that are used to write data into the file.

(i) write( )

This method, accepts a string as a parameter and writes it into the file. The string can have a single or multiple lines of text such as a data block.

Syntax : file.write(string)

(ii) writelines( )

This method, accepts a list of string as an argument and writes into a file. Line termination characters may not be present between each line.

Syntax : File.writelines(seq)

Where, 'seq' is an iterable sequence of strings.

(c) Intra-file Motion

These methods are used, to move the file pointer within the file.

(i) seek( )

This method is used, to move the file pointer to different locations within the file.

Syntax : file.seek(off, whence = 0)

Where,

'off' refers to the offset, i.e., the number of bytes the file pointer should be moved. And the parameter 'whence', specifies the position from where offset should be measured, i.e., from beginning, current position or from end of the file.

Whence	Measure offset from the
0 (default)	Beginning of file
1	Current location
2	End of the file

(ii) **tell( )**

This method, returns the current location of file pointer within the file. It returns the distance in bytes from the beginning of the file. In other words, if tell( ) returns a value say 128, it means that the file pointer is currently located on 128th byte from the beginning of the file.

Syntax : file.tell( )

(d) **File Iteration Methods**

File Iteration Methods, allow us to iterate through lines of a file. It is usually done with the help of a 'for-loop'.

Prior to the release of python 2.2, iteration was performed as follows:

```
for eachLine in file1.readline( )
    #process each line
```

However in python 2.2, the call to method readline( ) was avoided and hence, we can perform iteration as follows,

```
for eachLine in file1
# process each line
```

Another method i.e., file.next( ) can also be used, to read the nextline in the file. If there are no more lines in the file, then 'stopIteration' exception is raised.

(e) **Other Methods**

(i) **close( )**

This method is used, to explicitly close a file after its usage. Python garbage collection mechanism, however implicitly closes the file, if the file object reference is decreased to zero.

Syntax : file.close( )

(ii) **fileno( )**

This method, returns the file descriptor (integer value) for the file. The descriptor is useful for performing various lower-level operations.

Syntax : file.fileno( )

(iii) **flush( )**

This method, will immediately write the contents of output buffer to the desired file.

Syntax : file.flush( )

(iv) **isatty( )**

This method, is used to determine whether the file is a tty-like device. If yes, it returns 'True', otherwise 'false'.

Syntax : file.isatty( )

(v) **truncate( )**

This method, 'truncates' (delete the contents) of the file to the specified size. If the size is not specified, then the file will be truncated till current file position.

Syntax : file.truncate(size)

Q13. Explain in brief the file system of python and file built-in methods.

Answer : Model Paper-III, Q4(a)

File System

The file system of python consists of several OS modules to perform tasks related to files and directory. The OS also serves as a front end module to a pure OS-dependent module. It avoids direct usage of OS dependent modules by loading the required module for the operating system being installed.

The methods supported by the OS modules can be put into three categories. They are as follows,

1. File processing methods
2. Directory methods
3. Permissions methods.

1. File Processing Methods

File processing methods like rename( ) and remove( ) are provided by the OS modules in order to perform file processing operations like renaming and deleting the files. The function of these methods are as follows,

rename( ) method

This method enables a user to rename a file. The general syntax of this method is,

rename(current_file_name, new_file_name)

Here, current_file_name is the current name of file and new_file_name is the new name to be given for the file.

Example

```
>>import OS
>>> OS.rename('somefile', 'newfile')
```

When the above code is executed the rename( ) method renames 'somefile' to 'newfile'.

remove( ) method

This method enables a user to delete a file from the operating system. This can be done by providing file name that is to be deleted as an argument. The syntax of remove( ) method is as follows,

remove (file_name)

2. **Directory Methods**

Directory methods of OS modules allow a user to display the current directory and view the content of directory. Moreover, they also allow us to create, change and also remove the directories. Some of the directory methods are as follows,

mkdir( ) method

This method is used to create directories within current directory.

syntax: mkdir(directory_name)

getcwd( )

This method is used to display the current working directories

syntax: getcwd( )

rmdir( ) method

This method is used to remove the directory.

syntax: rmdir('directory_name')

3. **Permissions Methods**

These methods of OS modules enables a user to set and verify the permission modes. Some of the permission methods are as follows,

access( ) method

This method is used to verify the access permission. That is specified in mode argument to specified path.

syntax: access (path, mode).

chmod( )

This method is used to change the access permission of the path to specified modes.

syntax: chmod(path, mode)

unmask( )

This method is used to set the specified mask as argument.

syntax: unmask(mask)

Q14. List and explain various file built-in attributes.

Answer :

File objects have file attributes other than file methods. They contain auxiliary data of file objects such as filename, file mode and a flag that indicates whether an additional space is required to be displayed before any successive data items. Various attributes of file objects are as follows,

Attribute	Description
file.name	It is the name of file.
file.mode	It is the access mode using which file is opened.
file.encoding	It is a type of encoding to convert the unicode strings into byte strings.
file.closed	It returns true if file is closed, otherwise false is returned.

2.1.3 Standard Files, Command Line Arguments

Q15. Write about,

(i) Standard files

(ii) Command line arguments.

Answer :

(i) Standard Files

The standard files are used for read or writing data. They are preopened and allow access of data through handlers when they are available to the users. These file handles are provided by signs module in Python. There are three types of standard files in Python, they are as follows,

1. **stdin:** The function raw_input() receives input from sys.stdin:
2. **stdout:** The print statement outputs to sys.stdout.
3. **stderr:** It holds the error information that is to be forwarded to user.

The argument that are provided to the program other than script name upon invocation are called command line arguments. They can be accessed through sys.argv and are provided sys module. The argv stands for "argument vector". It consists of array of strings in which every argument is from command line. And argc is a variable that holds count of arguments. The first argc would the name of the program.

In unix based systems, the commands act as programs that accept input and perform some operation and then send output as stream of data. This is the data that is sent as input to next program. Instead of saving the output on disk space, it is first directed to next program as input. This can be implemented through command line input or standard input.

The command line arguments allow the programme to being the program with different characteristics. All these tasks are done at the background. Python maintains two modules for processing command line arguments, they are getopt and optparse.

2.1.4 File System, File Execution

Q16. Write in brief about file system. Explain briefly about file execution.

Answer :

File System

For answer refer Unit-II, Q13, Topic: File System.

File system is allowed to be accessed through Python OS module. It acts as a primary interface to operating system facilities and services from Python. The os module is a front-end to real module that is loaded. This module can be anyone from os2, Mac, nt, Dos, posix, etc.

These modules need not be imported directly. When os module is imported the required module automatically gets loaded. Other the managing the processes and process execution environment, the os module the major file system operations such as deleting files, remaining files, traversing directory tree and managing file accessibility. The file/directory access functions of os module are as follows,

File Processing

Functions	Description
*stat( )	It returns file statistics
remove( )/unlink( )	It deletes the file
symlink( )	It creates a symbolic link
walk( )	It generates filename in a directory tree
utime( )	It updates time stamp
mkfifo( )/mk nod( )	It creates named pipe/create file system node
rename( )/renames( )	It renames the file
tmpfile( )	It creates and opens new temporary file

Directories/Folders

Functions	Description
chroot( )	It changes root directory of the current process
chdir( )/fchdir( )	It changes the working directory/via a file descriptor
listdir( )	It lists all the files in directory
mkdir( )/makedir( )	It creates directory(is)
remdir( )/removedirs( )	It removes directory(is)
getcud/getcwdu( )	It returns current working directory

Access/Permissions

Functions	Description
access( )	It verifies permission modes
unmask( )	It sets the default permission modes
chmod( )	It changes permission modes
chown( )/lchown( )	It changes owner and group ID

File Descriptor Operations

Functions	Description
Open( )	It is used to open file
read( )/write( )	It reads/writes data to file descriptor
dup( )/dup2( )	It duplicates the file descriptor

Device Numbers

Functions	Description
major( )/minor( )	It extracts major/minor device number from raw device number.
makedev( )	It generates raw device number from major and minor device numbers.

File System Execution

To run an operating system command, initially invoke a binary executable or some other type of script. This involves execution of another file somewhere else on the system. Running execution of other python code might call some other python interpreter. But this might not be the case every time.

The most common execution scenarios are as follows,

- ❖ Executing another Python script.
- ❖ Creating and managing the subprocess.
- ❖ Executing a command that needs input.
- ❖ Invoking a command across the network.
- ❖ Executing a set of dynamically generated Python statements.
- ❖ Executing external command or program.
- ❖ Importing Python module.
- ❖ Executing a command that creates output and needs processing.

Execution of another Python programs involves the below steps.

1. **Import the Module**

   Initially import the module, this will cause the code at top level of that module to execute.

2. **Execute the Modules as Scripts**

   Execute the module as script from shell or DOS prompt.

2.1.5 Persistent Storage Modules, Related Modules

Q17. Discuss in brief about persistent storage modules.

Answer :

The system operations needs user to input the data. In case of iterations the same data might be entered multiple times repeatedly. The persistent storage archives the data so that it can be accessed in future rather than re entering the same data. Mostly the storage modules deal with storing the strings of data. Other than this even Python objects can be archived. Various persistent storage modules are as follows,

Module	Description
Pickle and Marshal	They allow to pickle the python objects. It converts the primitive types to binary set of bytes that can be stored or transmitted over network. Pickling is also called as serializing, flattering and marshalling. Marshal can handle the simple python objects and pickle can transform recursive objects that are multi referenced from different places and user defined classes and instances.
DBM style modules	They writes data in pure DBM format. There are various implementations such as dbm, gdbm, dumbdbm and dbhash/bsddb. They provide names space for the objects.
Schelve Module	This module makes use of any dbm module for finding the appropriate DBM module. It again use cpickle module to perform the pickling process. It only allows concurrent read access to database file.

Q18. List out various related modules.

Answer :

The modules related to files and input/output are as follows,

Functions	Description
csv	It allows access to comma separated value files
bz2	It allows access to BZ2 compressed files
base6u	It is encoding/decoding of binary strings to/from text strings
binascii	It is encoding/decoding of binary and ASCII encoded binary strings
filecmp	It compares directories and files.
tarfile	It reads and writes the TAR archive files
fileinput	It iterates over lines of multiple input text files
getopt/optparse	It provides command line argument parsing/manipulation
glob/fnmatch	It provides unix-style wildcard character matching
gzip/zlib	It reads and writes GNU zip files
shutil	It offers high level file access functionality.
c/string IO	It implements file like interface on string objects.
tempfile	It generates temporary files or file names.
zipfile	It provides tools and utilities to read and write ZIP archive files.
uu	It provides uuencode and uudecode files.

2.2 EXCEPTIONS

2.2.1 Exceptions in Python

Q19. Explain about the exception handling in Python.

Model Paper-I, Q4(b)

Answer :

Exception

An Exception is a run-time error, that halts the execution of the program. It is also described as an action, that is taken outside the normal flow of control by python interpreter or programmer, because of a runtime error. It is a two-phase process. First phase happens, when a phenomenon called 'Exception condition' or 'Exceptional condition' occurs. Whenever, an interpreter detect an error, it initiates something called 'raising an exceptions'. Though the interpreter immediately stops the flow of current execution, it intimates that something is wrong and that needs to be rectified raising an exception is also known as, 'triggering', 'throwing' or 'generating'.

In second phase, actual exceptional handling takes place. Exceptions can be handled either by the python interpreter or the programmer. Once an exceptional handling is initiated, programmer dictates an interpreter about the course-of-action to be taken, in order to contain the errors. This course-of-action could be taking some corrective measures, logging the error, ignoring the error (or) aborting the program itself. The actions to subside the errors are customized, according to the nature and severity of the error.

'try_except' statement is used by Python, to handle exceptions. It also allow one to detect exceptions. try_except statement, consists of code used to monitor for exceptions. It also provides a mechanism to handle exceptions.

Syntax

```
try :
try_suite       # Exceptions are detected here
except Exception[reason]:
except_suite  # code to handle exceptions.
```

Using the above syntax, we can enter a code into our program to both detect and handle the error. In the 'try_suite' block the code that may raise the exception is placed. The type of exception that should be handled is specified, through 'except Exception[reason] statement. In 'except_suite' block we define code, telling the interpreter about the actions it must take after an exception is raised.

The try_except statement, also has an optional statement called 'else'. The code in the 'else' clause is executed only when:

(i) No exceptions are raised in try ... except clause.

(ii) Try ... except suite must be successfully completed.

Example

```
# !/usr/bin/env python
import sys
try:
        f = open(fname, 'r')
except IOError:
        Print 'cannot open the file'
        sys.exit( )          # terminate the program
else :
        Print 'No exception was raised'.
```

In the above example, the interpreter will try to open a file 'fname'. If the file fname is not present or the interpreter is not able to open it, then exception IOError will be raised. The resultant output will be 'cannot open the file' and the program will be terminated. This is how exception is handled in the program. If the file is available and it is opened, then the output will be 'No exception was raised'.

2.2.2 Detecting and Handling Exceptions

Q20. Explain try_except_else_finally statement.

Answer :

try_except_else_finally

This is one of the different ways to handle the exceptions.

Its syntax is given below,

Syntax

```
try:
    try:
        x
    except MyException:
        y
    else :
        z
    finally :
        A
```

In the above syntax 'else', 'finally' clauses are optional and we can have more than one 'except', but the syntax should contain atleast one 'except' clause.

The 'else' clause executes, if no exceptions were found in the preceding try block. It is used mainly for the situations, when no exceptions are detected. Whereas, 'finally' allows the execution of code, regardless of whether or not exception occurs in the try block.

The try_except statements, allow us to detect and handle exceptions. It defines a section of code to handle exceptions and to provide mechanisms to execute handlers for exception.

```
try:
    try_code
except (Exception_a, Exception_b):
    code for Exception a and Exception b
except 'Exception_c', Argument_1:
    code for Exception c plus argument 1
else:
    no exceptions
finally:
    always execute code.
```

Q21. Explain try-finally statement.

Answer :

try-finally

We can use finally clause with the try block try-finally statement is used, to execute some part of the code, even there exists some errors. In other ways, we can say that, it is used to maintain the consistent behaviour irrespective of the occurrence of exceptions.

The code which is to be executed is placed in finally block. If an exception occurs while executing the code in the try block, then the control jumps to the finally clause the code in finally gets executed i.e., when an exception is raised in the try block, the statements below the control are not executed and the control immediately jumps to the finally block. On completion of finally block, the exception is reraised for handling errors. The syntax of the try-finally statement is given below,

Syntax

```
try:
    try_suite
finally:
    finally_suite
```

Example

```
try:
    try:
        txtfile = open('presentation.txt', 'r')
            modules = txtfile.readlines( )
    except IOError:
            log.write('no modules \n')
        finally:
            txtfile.close( )
```

It is a common practise to write the try-finally statement, nested as a part of a try-except suite.

In the above example, we tried to open the file 'txtfile' and read the modules (data).

While making an attempt to the txtfile, if an error is encountered, then the file is simply closed. If not the try block is executed and finally the file is closed.

2.2.3 Context Management, Exceptions as Strings

Q22. Explain about the following

(i) Context management

(ii) Exceptions as strings.

Answer : Model Paper-II, Q4(a)

(i) **Context Management**

Context management in python can be as follows,

1. **By Using With Statement**

Python provides a great level of abstraction. One approach is through the use of with statement. It is used when Try-except and try-finally are used together to active allocation of shared resources for execution. It then release upon completion of task - Examples of it are files, synchronization primitives, threading resources, database connections etc. Rather than reducing the code and using try-except-finally, the with statement aims to remove the Try, except and finally key words including the allocation and release code. The general format of this statement is as follows,

With context_expr [as uar]:

With _suite.

It is used with objects which support context management protocol. A simple statement. of with is as follows,

With open ('/etc/passwd', 'r') as f:

for each line in f:

statements.

The above statement opens the file and assigns fole object to f. It iterates through energy line in file and performs the specified action. The file gets closed when it is exhausted. If any exception occurs the cleanup code must be done, but the file however closes automatically.

2. **By Using Context Management Protocol**

The context manager is called when the context expression is evaluated while the execution of with statement. It provides a context object by inmaking the _context_() method. This object is used in the execution of with_ suite. The context object can act as its own manager. Therefore the context_expr can be context manager or context object scaning its manager. It has a _context_() method that returns self. When the context object is obtained special method called _enter_() is invoked. It performs all the primary things before executing with_suite. There is an optional "as var" with context_expr on the with statement line. The return value of _enter_() is assigned to var, otherwise return value is thrown away. The with_suite execute now and terminates. Then the context objects _exit_() method gets called. The method _exit_() accepts three arguments. The context lib module contains functions/decorates that can be applied over functions or objects.

(ii) **Exceptions as Strings**

The standard exceptions are implemented in the form of strings. It does not allow any relationship among exceptions. The exception classes avoid such cases. All the exceptions have now become classes in 1.5. The programmers can however generate their own exceptions as strings. But it is better to use the exception classes. Python 2.5 starts the process of deprecating the string exceptions from Python forever. It generates warnings when there are string exceptions. The catching process of exceptions generates warning. They are rarely used and therefore deprecated. The string exceptions are no longer used nowadays.

2.2.4 Raising Exceptions, Assertions

Q23. Write about raising exceptions.

Answer : Model Paper-I, Q5

Exceptions can be raised using the raise statement.

General syntax for 'raise' statement is,

raise[SomeException[, args[, traceback]]]

Arguments

(i) SomeException, is the first argument in the raise statement. If other arguments are specified, then it is necessary to mention 'SomeException'. The 'SomeException' argument, specifies the name of the exception, it may be either a string, class or instance.

If SomeException is a class

If SomeException is a class, then there is no need of additional arguments. However, if they are specified, then they must be either single object, a tuple, or an exception class instance.

If the argument is an instance, then it may be the instance of given class or derived class and in this case additional arguments are not required. If SomeException is either tuple or single to n, then class is instantiated with args as parameter to exception.

If SomeException is an Instance

No need to instantiate any thing and additional parameters must be none.

If SomeException is a String

An exception is raised, denoted by that string with optional args as parameters.

(ii) 'args' is the second argument of the 'raise' statement, which is optional. If the optional args are mentioned, then this argument stores them either as a tuple of objects or a single object.

If args is a tuple, it represents the arguments given to the handler generally error string, error number, error location.

If args is single object, then it will contain one object, i.e., a string indicating error.

(iii) 'traceback', the final argument which is also optional. When an exception is raised, traceback object is created. If we want to re-raise an exception we use the traceback object.

If the 'raise' statement does not contain any parameters, then the last exception in the current block is re-raised. If no exception was previously raised, an exception 'TypeError' is raised.

Usage of the 'raise' Statement

Different ways of using 'raise' statement is listed below.

Syntax	Action
raise exclass	Create an instance of exclass and raise an exception.
raise exclass, args	Create an instance of exclass with arguments and raise an exception.
raise exclass, args, tb	Create an instance of exclass with arguments and raise an exception. Also provides traceback object.
raise string	Raises string exception.
raise string, args	Raises string exception with arguments.
raise string, args, tb	Raises string exception with argument and provides traceback object.
raise instance	Raise exception using instance of some class.
raise	Reraise the last exception, if there is no exception raised previously then, 'TypeError' is raised.

Q24. Explain about assertions.

Answer :

Assertions

Assertions are the diagnostic predicates that should evaluate to Boolean true. If this is not the situation, the exception will be raised to indicate that the expression is false. It is similar to raise_if statement. An expression is checked and an exception is raised if the result is false. The assertions are handled using assert statement. This statement evaluates the Python expression and does not perform any action upon success. But if it is false then an Assertionerror exception is raised. The general format of it is as follows,

assert expression [, arguments]

The assert statement can be used in below ways.

```
assert 2 == 2
assert 5 + 5 == 5 * 5
assert len (['list', 18]) < 15
assert range (5) == [6, 9, 5, 4, 2]
```

The Assertionerror exception can be handled first like some other exceptions by using the TRy_except statement. But they can terminate the program, if not handled. A traceback will be generated as shown below.

```
>>>assert 1 == 0
    Traceback(innermost last):
        File "<stdin>",line1,in?
    Assertionerror
```

This statement can be even passed with Assertionerror exception. The TRy_except statement to catch the Assertionerror exception is as follows,

```
try:
    assert 1 == 0, 'one is not equal to zero'
except Assertionerror, args:
    print'%s:%s'%(args._class_._name_,args)
```

The above code generates the output as shown below,

Assertionerror: one is not equal to zero.

2.2.5 Standard Exceptions, Creating Exceptions

Q25. List out the standard exceptions.

Answer :

When a program in Python gets terminated due to certain unresolved errors, a user is confronted with a notice called traceback. It has an entire diagnostic information about the error an interpreter can hold. Traceback makes aware of a user with details like, error name, reason and error location. All the errors, irrespective of nature have a similar format. There are several built-in exceptions in python few of them are as follows,

(i) NameError

Python interpreter uses 'NameError', to inform a user that the requested variable is not present in any of the namespaces, perhaps it may not be declared. When programmer calls an object, interpreter will search for it in all the namespaces. If it does not find it, then NameError is generated.

Example

```
>>> x
Traceback(outtermost first):
File"<stdin>", line1, out?
NameError : name 'x' is not defined.
```

(ii) **ZeroDivisionError**

Python interpreter will generate this exception, when an attempt is made to divide a number by zero.

Example

```
>>10×(1/0)
Traceback(innermost last):
File "<stdin>", line1, in?
ZeroDivisionError : integer division or module by zero.
```

(iii) **SyntaxError**

SyntaxError exceptions are generated, when a wrong syntax is used. These are the only exceptions that are generated at the time of code compilation whereas, other exceptions are generated during the run time.

Example

```
>>>def safe_float(obj) :
try
SyntaxError : invalid.syntax.
```

(iv) **IndexError**

Indexes are accessed, within a range when a user tries to access an index, outside the valid range of a sequence, 'IndexError' is generated by the Python interpreter.

Example

```
>>>aList = [ ]
>>>aList[5]
Traceback(outtermost first):
File "<stdin>, line1, out?
IndexError : list index out of range.
```

(v) **KeyError**

Objects like dictionaries are mapped, based on the keys that are used, to access data values. If the keys used in mapping are incorrect or non existent, then the values will not be retrieved. In these cases, interpreter will generate 'keyError' to intimate user about the incorrect keys used in mapping.

Example

```
>>>myDict = {'client' : 'sun', 'port' : 90}
>>>print aDict['HOST']
Traceback(innermost last):
keyError : Server
```

(vi) **IOError**

Python Interpreter generates IOError, on eve of any operating system I/O error. A classic example for IOError is an attempt to open a non-existent disk file.

Example

```
file = open "logic.txt"
Traceback(innermost last):
File "<stdin>" line 1, in?
IOError : [errno4] No such file or dictionary : "logic.txt"
```

(vii) **AttributeError**

AttributeError, arises when a user tries to access an attribute of an instance, which is not defined.

Example

```
class example(object):
... pass
...
>>>inst = example( )
>>>inst.bar = 'Draft'
>>>inst.bar
'Draft'
>>>inst.foo
Traceback(innermost last):
File"<stdin>", line1, in?
AttributeError : foo
```

The above example, tries to access 'foo' attribute of 'inst' which is not defined in example class. Hence, interpreter generated 'AttributeError'.

Q26. How exceptions are created? Explain.

Answer : Model Paper-II, Q4(b)

Creating Own Exceptions

Python provides a wide range of standard exceptions, but programmers would need to use their own exceptions, that provide more detailed information than that with standard exceptions.

Consider the IOError generic exception that is concerned with input/output problems, generated due to invalid file access or through other ways of communication. A new exception can be created, to determine the source of problem, for example, **FileError**. The function of **FileError** exception is same as **IOError** exception, but with a name that provides detailed description while performing file operations.

Another user-defined exception can be created, with respect to network programming with sockets. The Python exception that handles the exceptions generated by **socket** module is, **socket.error.** The socket.error is a subclass of generic **Exception** exception. As the arguments of **socket.error** are identical to those of IOError exceptions, a new exception, **NetworkError** can be defined, as a subclass of **IOError** but providing the information as that of **socket.error**.

The following module defines and uses two new exceptions, **FileError** and **NetworkError**.

```
1.  #!/usr/bin/env python
2.  import os, socket, tempfile, errno, types
3.  class FileError(IOError):
4.  Pass
```

```
5.  Class NetworkError(IOError):
6.      Pass
7.  def updateAgs(ag, newAg = None):
8.      if isinstance(ags, IOError):
9.          my_ags = [ ]
10.         my_ags.extend([ag for ag in ags])
11.     else:
12.         my_ags = list(ags)
13.     if newAg:
14.         my_ags.append(newAg)
15.     return tuple(my_ags)
16. def fileAgs(f, mode, ags):
17.     if ags[0] = = errno.EACCES and\
18.     'access' in dir(os):
19.         ps = ' '
20.         pd = {'r': os.R_ok, 'w': os.w_ok, 'x': os.x_ok}
21.         pks = pd.keys( )
22.         pks.sort( )
23.         pks.reverse( )
24.         for each P in 'rwx'
25.             if os.access(f, pd[each P]):
26.                 ps + = eachP
27.             else
28.                 ps + = '_'
29.         if isinstance(ags, IOError):
30.             my_ags = [ ]
31.             my_ags.extend[ag for ag in ags]
32.         else:
33.             my_ags = list(ags)
34.             my_ags[1]= "%s", %s ("ps: "%s")" %(mode, my_args[1], ps)
35.             my_ags.append(ags.filename)
36.     else:
37.         my_ags = ags
38.     return tuple(my_ags)
39. def myconnection(sockt, host, port):
40.     try:
41.         sockt.connect((host, port))
42.     except socket.error, ags:
43.         my_ags = updateAgs(ags)
44.         if len(my_ags) = = 1:
45.             my_ags = (errno. ENXIO, myags[0])
46.         raise NetworkError,\
47.         updateAgs(my_ags, host+':'+str(port))
48. def myfileopen(f, mode = 'r'):
49.     try:
50.         fopen = open(f, mode)
51.     except IOError, ags:
52.         raise FileError, FileArgs(f, mode, ags)
53.     return fopen
54. def testFile( ):
55.     newfile = mktemp( )
56.     f = open(newfile, 'w')
57.     f.close( )
58.     for eachTest in ((o, 'r'), (0100, 'r'), (0400, 'w'), (0500, 'w'
59.     try:
60.         os.chmod(newfile, eachTest[0])
61.         f = myopen(newfile, eachTest[1])
62.     except FileError, ags:
63.         print "%s: %s" %\(ags._class_._name_, ags)
64.     else:
65.         print newfile, "successfully opened.Ignored permissions
66.         f.close( )
67.         os.chmod(newfile, 0777)
68.         os.unlink(newfile)
69. def testntwk( ):
70.     skt = socket.socket(socket.AF_INET, socket. SOCK_
      STREAM)
71.     for eachHost in ('deli', 'www'):
72.         try:
73.             myconnection(skt, 'deli', 8080)
74.         except NetworkError, ags:
75.             print "%s : %s" %\(ags._class_._name_, ags)
76. if _name_ = = '_main_':
77.     testfile( )
78.     testntwk( )
```

Line 1

It is the Unix start up script.

Line 2

It imports the modules, os, socket, tempfile, errno and types.

Lines 3, 4

They create the FileError, by subclassing it from the existing class **IOError**.

Lines 5, 6

They create the NetworkError, by subclassing it from the existing class **IOError**.

Lines 7-15

These lines define updateAgs( ) function, that updates the exception arguments. The updateAgs( ) function, actually obtains the arguments from original exception and adds any given information as an argument.

Lines 16-34

These lines define a fileags( ) function, that provides few arguments related to the file access.

Lines 17, 18

They check, whether a permission error is obtained and determines the available permissions. If those checks fail, a dictionary is set up for creating a new string that stores the actual file permissions.

Lines 20

It checks, whether the underlying system supports os.access( ) function in order to determine the file permission for any file.

Lines 24-28

These lines, construct a permission string. The permissions for read, write and execute are displayed as *r*, *w* and *x*, respectively, in the string. If any of these permissions is not granted, then a dash(–) is placed, accordingly. For example, the string '–wx' indicates, that access has been granted to write and execute but not to read.

Lines 29-33

These lines create a temporary argument list.

Line 34

The error string is updated to hold the permission string.

Line 35

The filename is included in the argument list.

Line 38

All the arguments are returned as a tuple.

Lines 39-47

These lines define myconnection( ) function. The myconnection( ) function is similar to a standard socket **connect( )** method, that generates IOError type exception for a network connection failure. It also includes host name and port number as arguments.

When a network connection failure occurs, the actual host-port combination along with error number and error string can be used by any database or name service. Moreover, if a host cannot be found, an (error number, error) string pair is provided.

Line 48-53

These lines define myfileopen( ) function. The myfileopen( ) function is similar, to standard file **open( )** method. If the opening of a file generates an IOError exception, new error and customized arguments are returned through fileArgs( ) method.

Lines 54-68

These lines define testFile( ) function. In this method, a temporary file is created, through **tempfile** module. If an error occurs in creating this file, the program terminates. Then, four different permission configurations 0, 0100, 0400 and 0500 are considered and the file is for each of these configuration, in an invalid mode.

0 – Indicates no permission

0100 – Indicates executive-only permission

0400 – Indicates read-only permission

0500 – Indicates, read-only and execute permission (since, 0400 + 0100 = 0500)

The permission modes of a file are changed, using os.chmod( ) function. If an error occurs, the detailed information is displayed, specifying the classname and the associated arguments.

If the file opening is successful, permissions would not have been considered. The user is informed about this, by displaying a message and then closing the file. Later, all the file permissions, are enabled and then the file is removed through os.unlink( ) function.

Lines 69-75

These lines, define testntwk( ) function. The testntwk( ) function, tests the **NetworkError** exception. In this function, a socket object is created to establish connection with host in another network, without involving server that can accept such a request.

Lines 76-78

This script when invoked, executes testFile( ) and testntwk( ) functions.

The output of the above script is given below,

$myexc.py

FileError: [Errno 13] 'r' Permission denied (ps: '---'):
'/usr/tmp/@18506.1'
FileError: [Errno 13] 'r' Permission denied (ps: '--x'):
'/usr/tmp/@18506.1'
FileError: [Errno 13] 'w' Permission denied (ps: 'r--'):
'/usr/tmp/@18506.1'
FileError: [Errno 13] 'w' Permission denied (ps: 'r-x'):
'/usr/tmp/@18506.1'
NetworkError:[Errno 146] Connection refused 'deli:8080'
NetworkError :[Errno 6] host not found: 'www: 8080'

2.2.6 Why Exceptions (Now)?, Why Exceptions at All?

Q27. What is the necessity of exception handling?

Answer :

Exception handling is necessary for error management without which, the programmers will not have control over their code.

For example, if our application is GUI-based and uses several resources. If by mistake a user possess an interrupt key (like Alt + F4), then application closes abruptly without performing clean-up (i.e., releasing resources, closing open files etc). This will result in data loss, corruption and inconsistency. In such a case, exception handling would be very useful, since, it provides a mechanism to take actions such as executing clean-up code or prompting users whether they really want to exit from the application etc.

If we do not have the concept of "exception handling", then, we would have to write functions such that they return some special value (for example 'None') if an error occurs during execution. Now, the programmer has to write code for checking each and every return value. If that return value is equal to special value, then it means an error has occurred and should be handled. This way of handling errors is complex and unstructured. Moreover, it will be a problem if special value (say 'None'), is the actual value that function wants to return. All the above problems can be solved easily, by using exception handling mechanism.

Exception handling is also useful, in error propagation. That is, when an error occurs in some function, the error data is transmitted to its caller function, which handles that error situation. If this error propagation mechanism is implemented using normal python's constructs, then it would become lengthy, tedious and complex. The python's exception handling' mechanism deals with exception propagation easily.

2.2.7 Exceptions and the Sys Module, Related Modules

Q28. Write about exceptions and sys module and list out the related modules.

Answer :

Exceptions and the Sys Module

The information about the exception can be obtained by accessing the exc_info() function in sys module. It provides a 3 - tuple of data:

```
>>>try:
    float ('xyz')
    except:
    import sys
    exc_tuple = sys.exc_info()
>>>print exc_tuple
    (<class exceptions.value error at f9838),<exceptions.value error instance at 122fa8>,<traceback object at
    10dc18>)
```

```
>>>for eachitem in exc_tuple:
    print eachitem
    exception.value error
    invalid literal for float(): xyz
    <traceback object at 10de18>
```

The sys.exc_info() generates the following,

- exc_type: exception class object
- exc_traceback: traceback object
- exc_value: exception class instance object

Related Modules

The related modules are as follows,

Module	Description
exception	It provides built-in functions.
sys	It consists of various exception-related objects and functions.
contextlib	It provides context object utilities to be used with the with statement.

2.3 MODULES

2.3.1 Modules and Files, Namespaces

Q29. Define modules. Explain about modules and files.

Model Paper-II, Q5

Answer :

Modules

A Module stores pieces of python code, which can be shared by other python programs. It is a mechanism, that is used to organize the code logically. Lengthy codes can be divided into several independent modules, that can interact with each other.

The code present in a module, may be a single class with its methods and data variables or a group of related classes and functions. A module can also share or use the code present in some other module. This is done, by importing a module. Hence, modules provide code reusability.

Modules and Files

Modules organize the files logically where as files organize the file physically. Every file is considered as individual module. The name of the file will be appended with .py extension. Various concepts of modules and files are as follows,

1. **Namespaces**

Namespaces are separate set of mappings from names to objects. The attribute name will be prepended with module name. For example, the atoi() function in string module is called.

```
string.atoi()
```

It allows only one module with the specified name. Therefore name collisions will not occur in it. Every module defines its own unique namespace. The attributes might have name conflicts so, the fully qualified name referring to an object through dotted attribute notion will prevent the exact and conflicting match.

2. **Search Path and Path Search**

The process of importing module needs a process called path search. It checks for "predefined areas" of file system to search for my module.py file to load "mymodule" module. The predefined areas are not more than set of directories belonging to Python search path. A default search path will be defined in compilation or installation process by default. This search path can be modified in either of two places. One is PYTHONPATH environment variable that is set in shell or command-line interpreter that invokes Python. It contains colon-delimeted set of directory paths. When the interpreter begins, the path can be accessed and stored in sys module as sys.path variable.

```
>>>sys.path
[' ', '/usr/local/lib/p2.x/','/usr/local/lib/p2.x/plat-sunor5','/user/local/lib/p2.x/lib-tk','/usr/local/lib/p2.x/lib-dynload','/user/local/lib/p2.x/site-packages',]
```

Now the module can be loaded by importing. In case of duplicate modules, the interpreter loads the first module. The sys.modules contains module names and physical location as keys and values respectively.

Q30. Discuss about namespaces.

Answer : Model Paper-III, Q4(b)

Namespaces

Namespace is the process of mapping the names to objects. And the process of adding name to namespace is called binding identifier to object. Modifying the mapping of a name is called rebinding and removing a name is called binding identifier to object. Modifying the mapping of a name is called unbinding. Namespaces are of three types, they are local, global and built-ins. The Python interpreter loads the built-ins first. They contain the names in _builtins_ module. The global namespace will be then loaded for executing the module. It becomes active namespace when execution begins. The namespace are truly the mappings between the names and objects. These names can be accessed based on physical location. The relationship between namespace and variable scope is illustrated in below figure.

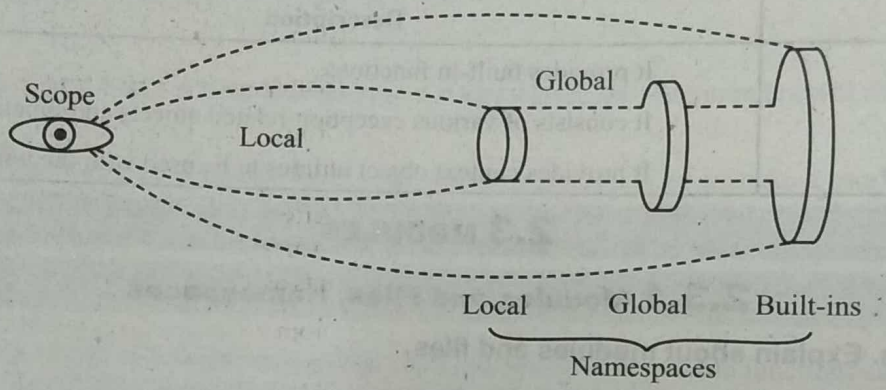

Figure: Namespaces Versus Variable Scope

Every namespace is a self contained unit. But from scoping point of view the things seem to be different. All the names within the local namespace are in my local scope. The names out of it are in global scope. The local namespaces and scope are transient since the function calls pass by, but global and built-ins namespaces just remain.

Working of scoping rules completely depend upon the name lookup. To access the attribute, the interpreter must find it in any of three namespaces. The search starts with local namespace. The global namespace will be searched if the attribute is not found. If this is not successful then the final frontier is built-ins namespace.

Consider the below code,

```
def fn():
    print"/n calling fn()...."
    x = 10
    print "in fn(), x is", x
    x = 50
    print "in _main_' x is", x
    fn()
```

The output of the above code will be,

```
in _main_, x is 50
calling fn()....
in fn(), x is 10
```

Here *x* is local namespace of fn() that overrides the global *x* variable. The lookup has found *x* in local namespace first even though global namespace exists. It overrides the global one.

3.2 Importing Modules, Importing Module Attributes

Q31. Describe how modules are imported in python.

Answer : **Model Paper-III, Q5**

Importing Modules

In order to use the code present in some module, we need to import that module in our program. This can be done by using the import statement which has the following syntax:

```
import <module_name>
```

Example

```
import myModule
```

Python also allows multiple modules to be imported in a single line as follows,

Syntax

```
import<module_name1>[, <module_name2>[, ... <module>]]
```

Example

```
import myModule, moduleA, moduleB
```

When Python interpreter encounters the import statement, it loads and executes the module being imported. Scoping rules, however are applicable. That is, if module is imported from a function, it gets local scope. And if it is imported from top-level then, it gets global scope.

There is no hard-and-fast rule for writing import statements, However, the best practice is to import one module in one line and that the following order is maintained.

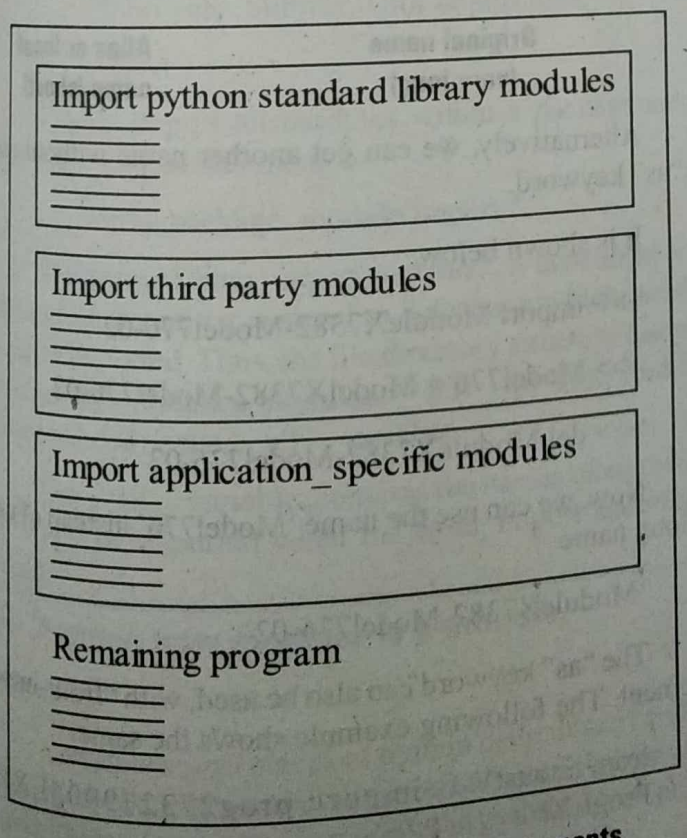

Figure: Ordering of Module Import Statements

There are different varieties of import statements. Some of them are discussed below:

(i) from_import Statement

This statement is used, to import specific elements from the module instead of importing the complete module. It is done as follows:

Syntax

```
from<module_name> import <element1> [,....<elementN>]
```

Example

```
from myModule import func1, func2, func3
```

(ii) Multi-line Import Statement

When several elements are being imported using a single "from-import" statement, then the line gets long. Hence, we can use a NEWLINE escaping backslash character to import in multi-lines.

Example

```
from myModule import classA, func1, \ classB, classImp, func_Set
```

Alternatively, we can use the Python's standard grouping mechanism i.e., parentheses to write multi-line import statements as shown below.

```
from myModule import(classA, func1, clasB, classImp, func_set)
```

However the most simple and obvious option is to use separate import statements as follows:

```
from myModule import classA, func1
from myModule import classB, classImp
from myModule import func_set
```

(iii) Import Statement with 'as'

The word 'as', is an extension to import statement. It gives another name (or alias), to the modules or elements that are imported. In other words, we can bind a local name for the imported items. This is very useful, when the original name of the element being imported is either long or meaningless and we want to give it a short and more meaningful name. It is used as follows:

```
import ModuleX7382-Model776-02 as Model776
```

Original name (very long) **Alias or local name (short)**

Alternatively, we can get another name without using the "as" keyword.

It is shown below.

```
>>> import ModuleX7382-Model776-02
>>> Model776 = ModelX7382-Model776-02
>>> del ModuleX7382-Model776-02
```

Now, we can use the name 'Model776' instead of using the long name

'ModuleX7382-Model776-02'.

The "as" keyword can also be used, with "from-import" statement. The following example shows the same.

from ExaustV-1 import prog273279905LXI as ProgLXI

Q32. Draw and explain the namespace relationship between an application and an imported module.

Answer :

When a module is imported for the first time it creates the namespace same as calling a function creates a namespace.

A module is imported using the import statement. When this statement is executed it assigns the module namespace to the module name.

The module namespace contains the names bounded in the module.

To access the names present in module namespace, we use the syntax. Module_name.Name.

For example, to access the variable '*a*' defined in module module1 we write,

```
>>>import module1
>>> module1.x
```

Example

Consider a module that defines two functions and an application that imports this module to call a function.

```
#module1.py
print "Module1 is executing"
def similarcase(x):
return x.islower( ) or x.isupper( )
def variouscase(x):
if similar case(x):
print x, "it does not have any variations"
else
print x, "It contains both lower and uppercase"
# application.py
import module1
module1.variouscase("Hai students"):
module1.various("hai students")
module1.various("HAI STUDENTS")
```

Here similar case( ) is a very simple function which returns true and prints. "It does not have any variations", if the string x that is passed as an argument has either all lower case or all upper case. Various case( ) function defined by the module. Checks whether the string has mixed case or not.

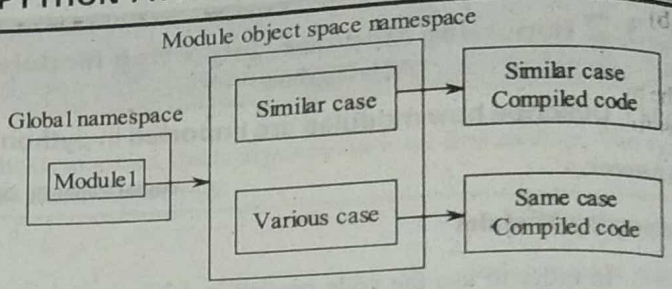

Figure: Relationship between Application and Module

When an application imports module (example module1) then the module in the local namespace of the application is bound to reference to the module and in particular it namespace. For instance, the reference to module1.variouscase is resolved by the interpreter by first looking up module1 in the local namespace then looking up variouscase function in the namespace to which module1 is bound.

Q33. Explain the following statements,

(a) Import... as

(b) From ... import.

Answer :

(a) Import Statement with 'as'

The word 'as', is an extension to import statement. It gives another name (or alias), to the modules or elements that are imported. In other words, we can bind a local name for the imported items. This is very useful, when the original name of the element being imported is either long or meaningless and we want to give it a short and more meaningful name. It is used as follows:

```
import  ModuleX7382-Model776-02   as  Model776
```

Original name (very long) — **Alias or local name (short)**

Alternatively, we can get another name without using the "as" keyword.

It is shown below.

```
>>> import ModuleX7382-Model776-02
>>> Model776 = ModelX7382-Model776-02
>>> del ModuleX7382-Model776-02
```

Now, we can use the name 'Model776' instead of using the long name

'ModuleX7382-Model776-02'.

The "as" keyword can also be used, with "from-import" statement. The following example shows the same.

from ExaustV-1 import prog273279905LXI as ProgLXI

(b) from_import Statement

This statement is used, to import specific elements from the module instead of importing the complete module. It is done as follows:

Syntax

```
from<module_name> import <element1>
[,....<elementN>]
```

Example

```
from myModule import func1, func2, func3
```

Import Statements

The import statement can be used, to import modules and subpackages. For example, if year is a top-level package containing subject, unit etc., as subpackages, then a subpackage unit can be imported using import statement as follows,

Import year. Subject. Unit

Year. Subject. Unit. Topic( )

The from_import statement can be used, to import modules and subpackages in a variety of ways.

First, import just a top-level package and reference down the subpackage tree structure.

from year import subject

Subject. Unit. Topic( )

Second, import top-level package and subpackage and reference down the subpackage tree structure.

from year. Subject import unit

Unit. Topic( )

Third, import the top-level package and all subpackages and reference only the module.

from year. Subject. Unit import topic

topic( )

Fourth, import all modules within a package using a single from_import statement.

from package. module import *.

This form of from_import statement, is dependent on file system of an operating system, to determine which, modules should be imported. Thus, the file directory structure hierarchy should include initialize modules, __init__.py files. And these files should define the __all__ variable.

The __all__ variable, contains the names of all modules that should be imported when the from_import statement is invoked.

Q34. Explain how modular is a module.

Answer :

A module stores pieces of python code. It can be shared or used by the code present in some other module by importing module. Thus, it is easy to use modules in python which provide code reusability

The author of a module can protect the module of stopping other code, that imports the module from using or accessing its namespace. But since python is designed in such a way that most of its internals are open it may temp the users to access other modules which becomes a disadvantage (i.e., no security) in python. The namespace associated with the modules are accessible for both reading and writing .That is, the names of a module not only can be accessed but also can be binded with new values. In order to program effectively there must be effective coupling between the components of the program. The following example shows how python's module boundaries are overridden.

```
#
#abc.py
#
def f1(i, j)
print "Function f1:", i, j"
#
# xyz.py
#
import abc
def f2(k, l)
print "Function f2:", k, l
abc.f1("first", "second")
abc.f1 = f2
abc.f1("third", "fourth")
```

The output after running xyz.py will be

Function f1:first, second

Function f2: third, fourth

Q35. Explain reload( ) function with an example.

Answer :

In python, if the program or a module is used by many different programs, then it should be executed repeatedly during testing. This causes difficulty for the progammers to understand how program runs.

When the module is imported for the first time it will be executed. This is done by writing an import statement on the module containing the program.

But, if same module is used for the second time, it does not work.

Thus, reload( ) function is used to inform the interpreter to execute the entire module code again.

Example

```
>>import module1   //imports the module for the
                   //first time. module1 executing

>>> reload module1 // reloads the module when used
                   //for second time.
```

<module 'module1' from

'/home/projects/python/module.pyc'>

The use of reload( ) in the above code is it re-executes the module1, forcefully.

The reload( ) function performs another import on an already imported module. There are two criteria that should be met before using reload( ) built-in function. They are as follows,

1. Module shouldn't be imported using 'from-import' and it should be fully imported and successfully loaded.
2. The argument to reload( ), should not be a string of module name but it should be the module itself.

It should be called like reload(sys) instead or reload ('sys'). Code present in the module is executed only once when it is imported, second import merely binds the module name, but doesn't re-execute the code.

2.3.3 Module Built-in Functions, Packages, Other Features of Modules

Q36. List and explain various module built-in functions.

Answer :

The process of importing the modules has the functional support from the system. These types of functions are illustrated as follows,

1. _import_()

This function is used to import the module. It allows overriding if the user is inclined to develop his or her own importation algorithm. The general format of _import_() is as follows,

```
_import_(module_name[, globals[, locals[, fromlist]]])
```

Here the module name is the variable indicating module name, globals is dictionary of current names in global symbol table, names is dictionary of current names in local symbol table and fromlist is list of symbols to be imported.

2. globals() and locals()

These functions are used to return the dictionaries of global and local namespaces. The namespaces in a function are returned by locals() and namespaces that are globally accessible are returned by globals()

```
def fn()
    print '\n calling fn()...'
    str = 'x'
    num = 64
    print "Globals of fn()", globals().keys()
    print "Locals of fn()", locals().keys()
    print "Globals of _main_'s", globals().keys()
    print "Local of _main_'s", locals().keys()
    fn()
```

The output of above code will be $ namespaces.py

```
globals of _main_'s : ['_doc_', 'fn', '_name_', '_builtins_']
locals of _main_'s : ['_doc_', 'foo', '_name_', '_builtins_']
calling fn()....
globals of fn()'s : ['_doc_', 'fn', '_name_', _builtins_']
locals of fn()'s : ['num, 'str']
```

The reload() function can reimport the previously imported module. The general format of it is as follows,

```
reload (module)
```

Answer :

Package

A package can be defined as a hierarchical file directory structure that defines single Python application environment containing modules and subpackages. They use familiar attribute/dotted attribute notion for accessing the elements. Importing of modules in packages can be done using import and from _import statements.

Consider the below directory structure,

```
Fruits/
    _init_.py
    common_util.py
Apple/
    _init_.py
    x1.py
Mango/
    _init_.py
    x2.py
Grapes/
    _init_.py
    x3.py
```

Here, fruits is top-level package and apple, mango etc., are subpackages. They can be accessed as shown below,

```
import fruits.mango.x2
```

Even from_import can be used in the same way

```
from fruits import mango.x2
```

Similarly from_import can be used in various ways. An example is from-import all statement.

```
from package.module import*
```

Import of packages and subpackages is mandatory but it ends up in clashes. For this purpose the imports are classified as absolute. The names must be packages or modules must be accessed through the Python path. But the subpackages can be accessed through sys.path. The absolute import takes away some privileges of module writer of packages. To over come this loss relative import is used. It modifies the import statement to indicate the location of module in subpackage.

Q38. Explain about other features of modules.

Answer :

The other features of modules are listed as follows,

1. **Auto-loaded Modules**

The modules get loaded by interpreter for system use when it begins in standard mode. All these are maintained in sys. modules. The names are keys and locations are values.

2. **Preventing Attribute Import**

If module attributes are not needed to be loaded while importing module with "from module import" then prepend under score to attribute names.

3. **Case-insensitive Import**

The imports are by default case_sensitive. To make them work properly, an environment variable with name PYTHONCASEOK is defined.

4. Source Code Encoding

Python allows to create module file in native encoding rather than 7-bit ASCII.

```
#! /usr/bin/env python
#_*_coding:UTF_8_*_
```

5. Import Cycles

Python allows to use import cycles.

```
from xyz import xyz
def util()
pass
def pqr():
xyz()
pqr()
```

There will be circular import of xyz in traceback. A solution for this problem is to first move import into xyz() function declaration.

```
def xyz()
import pqr.
pqr.xyz()
```

6. Module Execution

Python module can be executed in number of ways. One way is script invocation via command line or shell execfile(), module import, interpreter_m option etc.