UNIT – 1

1.What is Python? Explain in detail.

Python is a high-level, interpreted, interactive and object-oriented scripting language. Python is designed to be highly readable. It uses English keywords frequently whereas the other languages use punctuations. It has fewer syntactical constructions than other languages.

1. Python is Interpreted − Python is processed at runtime by the interpreter. You do not need to compile your program before executing it. This is similar to PERL and PHP.

2. Python is Interactive − You can actually sit at a Python prompt and interact with the interpreter directly to write your programs.

3. Python is Object-Oriented − Python supports Object-Oriented style or technique of programming that encapsulates code within objects.

4. Python is a Beginner's Language − Python is a great language for the beginner-level programmers and supports the development of a wide range of applications from simple text processing to WWW browsers to games.

History of Python:

Python was developed by Guido van Rossum in the late eighties and early nineties at the National Research Institute for Mathematics and Computer Science in the Netherlands.

1. Python is derived from many other languages, including ABC, Modula-3, C, C++, Algol-68, Smalltalk, and Unix shell and other scripting languages.

2. Python is copyrighted. Like Perl, Python source code is now available under the GNU General Public License (GPL).

3. Python is now maintained by a core development team at the institute, although Guido van Rossum still holds a vital role in directing its progress.

4. Python 1.0 was released in November 1994. In 2000, Python 2.0 was released. Python 2.7.11 is the latest edition of Python 2.

5. Meanwhile, Python 3.0 was released in 2008. Python 3 is not backward compatible with Python 2. The emphasis in Python 3 had been on the removal of duplicate programming constructs and modules so that "There should be one -- and preferably only one -- obvious way to do it." Python 3.5.1 is the latest version of Python 3.

Python Features:

1. Easy-to-learn − Python has few keywords, simple structure, and a clearly defined syntax. This allows a student to pick up the language quickly.

2. Easy-to-read − Python code is more clearly defined and visible to the eyes.

3. Easy-to-maintain − Python's source code is fairly easy-to-maintain.

4. A broad standard library − Python's bulk of the library is very portable and cross- platform compatible on UNIX, Windows, and Macintosh.

5. Interactive Mode − Python has support for an interactive mode which allows interactive testing and debugging of snippets of code.

6. Portable − Python can run on a wide variety of hardware platforms and has the same interface on all platforms.

7. Extendable − You can add low-level modules to the Python interpreter. These modules enable programmers to add to or customize their tools to be more efficient.

8. Databases − Python provides interfaces to all major commercial databases.

9. GUI Programming − Python supports GUI applications that can be created and ported to many system calls, libraries and windows systems, such as Windows MFC, Macintosh, and the X Window system of Unix.

10. Scalable − Python provides a better structure and support for large programs than shell scripting.

2. Explain about the type of operators used in Python?

Operators:

- Operators are the constructs which can manipulate the value of operands.
- Consider the expression 4 + 5 = 9. Here, 4 and 5 are called operands and + is called operator.

Types of Operator:

Python language supports the following types of operators.

1 Arithmetic Operators

2 Comparison (Relational) Operators

3 Assignment Operators

4 Logical Operators

5 Bitwise Operators

6 Membership Operators

7 Identity Operators

Python Arithmetic Operators:

Assume variable a holds 10 and variable b holds 20, then –

Operator	Description	Example
+ Addition	Adds values on either side of the operator.	a + b =
- Subtraction	Subtracts right hand operand from left hand operand.	a – b = -
* Multiplication	Multiplies values on either side of the operator	a * b =
/ Division	Divides left hand operand by right hand operand	b / a = 2
% Modulus	Divides left hand operand by right hand operand and returns	b % a =
** Exponent	Performs exponential (power) calculation on operators	a**b =10 to the power 20

Python Comparison Operators:

These operators compare the values on either sides of them and decide the relation among them. They are also called Relational operators.

Operator	Description	Example
==	If the values of two operands are equal, then the condition becomes true.	(a == b) is not true.

!=	If values of two operands are not equal, then condition becomes true.	(a != b)
<>	If values of two operands are not equal, then condition becomes true.	(a <> b) is true. This is similar to !=
>	If the value of left operand is greater than the value of right operand, then	(a > b) is not
<	If the value of left operand is less than the value of right operand, then	(a < b) is true.
>=	If the value of left operand is greater than or equal to the value of right	(a >= b) is not
<=	If the value of left operand is less than or equal to the value of right	(a <= b)

Python Bitwise Operators:

There are following Bitwise operators supported by Python language

Operator	Description	Example
& Binary AND	Operator copies a bit to the result if it exists in both operands	(a & b) (means 0000 1100)
| Binary OR	It copies a bit if it exists in either operand.	(a | b) = 61 (means 0011 1101)
^ Binary XOR	It copies the bit if it is set in one operand but not both.	(a ^ b) = 49 (means 0011 0001)
~ Binary Ones Complement	It is unary and has the effect of 'flipping' bits.	(~a) = -61 (means 1100 0011 in 2's complement form due to

Python Logical Operators:

There are following logical operators supported by Python language. Assume variable a holds 10 and variable b holds 20 then

Operator	Description	Example
and Logical	If both the operands are true then condition becomes true.	(a and b)
or Logical OR	If any of the two operands are non-zero then condition	(a or b)
not Logical NOT	Used to reverse the logical state of its operand.	Not(a and b) is false.

Used to reverse the logical state of its operand.

Python Membership Operators:

Python‘s membership operators test for membership in a sequence, such as strings, lists, or tuples. There are two membership operators as explained below –

In	Evaluates to true if it finds a variable in the specified sequence and false otherwise.	x in y, here in results in a 1 if x is a member of sequence y.
not in	Evaluates to true if it does not finds a variable in the specified sequence and false otherwise.	x not in y, here not in results in a 1 if x is not a member of

Python Identity Operators:

Identity operators compare the memory locations of two objects. There are two Identity operators explained below –

is	Evaluates to true if the variables on either side of the operator point to the same object and false otherwise.	x is y, here is results in 1 if id(x) equals id(y).
is not	Evaluates to false if the variables on either side of the operator point to the same object and true otherwise.	x is not y, here is not results in 1 if id(x) is not equal to

UNIT II

1.Demonstrate usage of exceptions in Python?

Exception:

Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. Errors detected during execution are called exceptions and are not unconditionally fatal: you will soon learn how to handle them in Python programs. Most exceptions are not handled by programs, however, and result in error messages as shown here:

```
>>>

>>> 10 * (1/0)

Traceback (most recent call last): File "<stdin>", line 1, in <module> ZeroDivisionError: division by zero

>>> 4 + spam*3

Traceback (most recent call last): File "<stdin>", line 1, in <module> NameError: name 'spam' is not defined

>>> '2' + 2

Traceback (most recent call last): File "<stdin>", line 1, in <module>

TypeError: Can't convert 'int' object to str implicitly
```

The last line of the error message indicates what happened. Exceptions come in different types, and the type is printed as part of the message: the types in the example are ZeroDivisionError, NameError and TypeError. The string printed as the exception type is the name of the built-in exception that occurred. This is true for all built-in exceptions but need not be true for user- defined exceptions (although it is a useful convention). Standard exception names are built-in identifiers (not reserved keywords).

The rest of the line provides detail based on the type of exception and what caused it.

The preceding part of the error message shows the context where the exception happened, in the form of a stack traceback. In general, it contains a stack traceback listing source lines; however, it will not display lines read from standard input.

Built-in Exceptions lists:

Handling Exceptions:

It is possible to write programs that handle selected exceptions. Look at the following example,which asks the user for input until a valid integer has been entered but allows the user to interrupt the program (using Control-C or whatever the operating system supports); note that a user- generated interruption is signaled by raising the KeyboardInterruptException.

If an exception occurs which does not match the exception named in the except clause, it is passed on to outer try statements; if no handler is found, it is an unhandled exception and execution stops with a message as shown above.

A try statement may have more than one except clause, to specify handlers for different exceptions. At most one handler will be executed. Handlers only handle exceptions that occur in the corresponding try clause, not in other handlers of the same try statement. An except clause may name multiple exceptions as a parenthesized tuple.

2.Explain in detail about Packages in Python?

A *package* is a hierarchical file directory structure that defines a single Python application environment that consists of modules and sub packages. Packages were added to Python 1.5 to aid with a variety of problems including:

Adding hierarchical organization to flat namespace Allowing developers to group-related modules

Allowing distributors to ship directories vs. bunch of files Helping resolve conflicting module names

Along with classes and modules, packages use the familiar attribute/dotted attribute notation to access their elements. Importing modules within packages use the standard importand from-importstatements.

Using from-importwith Packages

Packages also support the from-importall statement:

However, such a statement is too operating system filesystem-dependent for Python to make the determination which files to import. Thus the_all_ variable in

init.py is required. This variable contains all the module names that should be imported when the above statement is invoked if there is such a thing. It consists of a list of module names as strings.

3.Give a short note on Python built in functions?

File Built-in Function [open()]

As the key to opening file doors, the open()built-in function provides a general interface to initiate the file input/output (I/O) process. open() returns a file object on a successful opening of the file or else results in an error situation. When a failure occurs, Python generates or *raises* an IOErrorexception—we will cover errors and exceptions in the next chapter. The basic syntax of the open()built-in function is:

file_object = open(*file_name*, *access_mode*='r', *buffering*=-1)

The *file_name* is a string containing the name of the file to open. It can be a relative or absolute/full pathname. The *access_mode* optional variable is also a string, consisting of a set of flags indicating which mode to open the file with. Generally, files are opened with the modes "r," "w," or "a," representing read, write, and append, respectively.

Any file opened with mode "r" must exist. Any file opened with "w" will be truncated first if it exists, and then the file is (re)created. Any file opened with "a" will be opened for write. If the file exists, the initial position for file (write) access is set to the end-of- file. If the file does not exist, it will be created, making it the same as if you opened the file in "w" mode. If you are a C programmer, these are the same file open modes used for the C library function fopen().

There are other modes supported by fopen() that will work with Python's open(). These include the "+" for read-write access and "b" for binary access. One note regarding the binary flag: "b" is antiquated on all Unix systems which are POSIX-compliant (including Linux) because they treat all files as "binary" files, including text files. Here is an entry from the Linux manual page for fopen(), which is from where the Python open() function is derived:

The mode string can also include the letter "b" either as a last character or as a character between the characters in any of the two-character strings described above. This is strictly for compatibility with ANSI C3.159-1989 ("ANSI C") and has no effect; the "b" is ignored on all POSIX conforming systems, including Linux. (Other systems may treat text files and binary files differently, and adding the "b" may be a good idea if you do I/O to a binary file and expect that your program may be ported to non-Unix environments.)

You will find a complete list of file access modes, including the use of "b" if you choose to use it,If *access_mode*is not given, it defaults automatically to "r."

The other optional argument, *buffering,* is used to indicate the type of buffering that should be performed when accessing the file. A value of 0 means no buffering should occur, a value of 1 signals line buffering, and any value greater than 1 indicates buffered I/O with the given value as the buffer size. The lack of or a negative value indicates that the system default buffering scheme should be used, which is line buffering for any teletype or tty-like device and normal buffering for everything else. Under normal circumstances, a *buffering*value is not given, thus using the system default.

Access Modes for File Objects

File Mode	*Operation*
r	open for read
w	open for write (truncate if necessary)
a	open for write (start at EOF, create if necessary)
r+	open for read and write
w+	open for read and write (see "w" above)
a+	open for read and write (see "a" above)
rb	open for binary read

Wb	open for binary write (see "w" above)
Ab	open for binary append (see "a" above)
rb+	open for binary read and write (see "r+" above)
wb+	open for binary read and write (see "w+" above)
ab+	open for binary read and write (see "a+" above)

UNIT III

1.Give a short note on Regular Expressions(Res)?

Regular Expressions (REs) provide such an infrastructure for advanced text pattern matching, extraction, and/or search-and-replace functionality. REs are simply strings which use special symbols and characters to indicate pattern repetition or to represent multiple characters so that they can "match" a set of strings with similar characteristics described by the pattern.

Python supports REs through the standard library re module. In this introductory subsection, we will give you a brief and concise introduction. Due to its brevity, only the most common aspects of REs used in every day Python programming will be covered. Your experience, will of course, vary. We highly recommend reading any of the official supporting documentation as well as external texts on this interesting subject. You will never look at strings the same way again!

Let us look at the most basic of regular expressions now to show you that although REs are sometimes considered an "advanced topic," they can also be rather simplistic. Using the standard alphabet for general text, we present some simple REs and the strings which their patterns describe. The following regular expressions are the most basic, "true vanilla," as it were. They simply consist of a string pattern which matches only one string, the string defined by the regular expression. We now present the REs followed by the strings which match them:

RE Pattern	String(s) Matched
Foo	foo
Python	Python
abc123	abc123

The first regular expression pattern from the above chart is "foo." This pattern has no special symbols to match any other symbol other than those described, so the only string which matches this pattern is the string "foo." The same thing applies to "Python" and "abc123." The power of regular expressions comes in when special characters are used to define character sets, subgroup matching, and pattern repetition. It is these special symbols that allow a RE to match a set of strings rather than a single one.

Special Symbols and Characters for REs

We will now introduce the most popular of the *metacharacters*, special characters and symbols, which give regular expressions their power and flexibility. You will find the most common of these symbols and characters.

Common Regular Expression Symbols and Special Characters

Notation	*Description*	*Example RE*
Symbols		
re_string	match literal string value *re_string*	foo
re1|re2	match literal string value *re1* or *re2*	foo|bar
.	match *any character* (except NEWLINE)	::.+::
^	match *start of string*	^Dear
$	match *end of string*	/bin/\w*sh$
*	match *0 or more* occurrences of preceding RE	[A-Za-z]\w*
+	match *1 or more* occurrences of preceding RE	\d+\.|\.\d+
?	match *0 or 1* occurrence(s) of preceding RE	goo?
{*N*}	match *N* occurrences of preceding RE	\d{3}
{*M*,*N*}	match from *M* to *N* occurrences of preceding RE	\d{5,9}
[…]	match any single character from *character* class	[aeiou]

[..*x*–*y*..]	match any single character in the *range from* x to y	[0–9], [A-Za-z]
[^…]	*do not match* any character from character class, including any ranges, if present	[^aeiou], [^A- Za-z0–9_]
(*|+|?|{})?	apply non-greedy versions of above occurrence/repetition symbols (*, +, ?, {})	.*?\w
(…)	match enclosed RE and save as *subgroup*	(\d{3})?, f(oo|u)bar
Special Characters		
\d	match any decimal *digit*, same as [0–9](\Dis inverse of \d:do not match any numeric digit)	data\d+.txt
\w	match any *alphanumeric* character, same as [A- Za-z0-9_](\W is inverse of \w)	[A-Za-z_]\w+
\s	match *any whitespace* character, same as [\n\t\r\v\f](\S is inverse of \s)	of\sthe
\b	match any *word boundary* (\Bis inverse of \b)	\bThe\b

2.Explain threads with Global interpreter lock?

Global Interpreter Lock

As an example, for any Python I/O-oriented

Execution by Python code is controlled by the *Python Virtual Machine* (a.k.a. the interpreter main loop), and Python was designed in such a way that only one thread of control may be executing in this main loop, similar to how multiple processes in a system share a single CPU. Many programs may be in memory, but only *one* is live on the CPU at any given moment. Likewise, although multiple threads may be "running" within the Python interpreter, only one thread is being executed by the interpreter at any given time.

Access to the Python Virtual Machine is controlled by a *global interpreter lock* (GIL). This lock is what ensures that exactly one thread is running. The Python Virtual Machine executes in the following manner in an MT environment:

Set the GIL,

Switch in a thread to run,

Execute for a specified number of bytecode instructions, Put the thread back to sleep (switch out thread),

Unlock the GIL, and,

Do it all over again (rinse, lather, repeat).

When a call is made to external code, i.e., any C/C++ extension built-in function, the GIL will be locked until it has completed (since there are no Python bytecodes to count as the interval). Extension programmers do have the ability to unlock the GIL however, so you being the Python developer shouldn't have to worry about your Python code locking up in those situations.

Routines (which invoke built-in operating system C code), the GIL is released before the I/O call is made, allowing other threads to run while the I/O is being performed. Code which *doesn't* have much I/O will tend to keep the processor (and GIL) to the full interval a thread is allowed before it yields. In other words, I/O-bound Python programs stand a much better chance of being able to take advantage of a multithreaded environment than CPU-bound code.

Those of you interested in the source code, the interpreter main loop, and the GIL can take a look at eval_code2() routine in the Python/ceval.c file, which is the Python Virtual Machine.

Exiting Threads

When a thread completes execution of the function they were created for, they exit. Threads may also quit by calling an exit function such as thread.exit(), or any of the standard ways of exiting a Python process, i.e., sys.exit() or raising the SystemExit exception.

There are a variety of ways of managing thread termination. In most systems, when the main thread exits, all other threads die without cleanup, but for some systems, they live on. Check your operating system threaded programming documentation regarding their behavior in such occasions.

Main threads should always be good managers, though, and perform the task of knowing what needs to be executed by individual threads, what data or arguments each of the spawned threads requires, when they complete execution, and what results they provide. In so doing, those main threads can collate the individual results into a final conclusion.

Accessing Threads From Python

Python supports multithreaded programming, depending on the operating system that it is running on. It is supported on most versions of Unix, including Solaris and Linux, and

Windows. Threads are not currently available on the Macintosh platform. Python uses POSIX-compliant threads, or "pthreads," as they commonly known.

By default, threads are not enabled when building Python from source, but are available for Windows platforms automatically from the installer. To tell whether threads are installed, simply attempt to import the thread module from the interactive interpreter. No errors occur when threads are available:

```
>>> import thread

>>>
```

If your Python interpreter was not compiled with threads enabled, the module import fails:

```
>>> import  thread Traceback (innermost last):

File "<stdin>", line 1, in ?
```

ImportError: No module named thread

In such cases, you may have to recompile your Python interpreter to get access to threads. This usually involves invoking the configure script with the "--with-thread" option. Check the README file for your distribution for specific instructions on how to compile Python with threads for your system.

Due to the brevity of this chapter, we will give you only a quick introduction to threads and MT programming in Python. We refer you to the official documentation to get the full coverage of all the aspects of the threading support which Python has to offer. Also, we recommended accessing any general operating system textbook for more details on processes, interprocess communication, multi-threaded programming, and thread/process synchronization. (Some of these texts are listed in the appendix.)

Life Without Threads

For our first set of examples, we are going to use the time.sleep() function to show how threads work. time.sleep() takes a floating point argument and "sleeps" for the given number of seconds, meaning that execution is temporarily halted for the amount of time specified.

Let us create two "time loops," one which sleeps for 4 seconds and one that sleeps for 2 seconds, loop0() and loop1(),respectively. (We use the names "loop0" and "loop1" as a hint that we will eventually have a sequence of loops.) If we were to execute loop0() and loop1() sequentially in a one-process or single-threaded program,the total execution time would be at least 6 seconds.

may not be a 1-second gap between the starting of loop0() and loop1(), and other execution overhead which may cause the overall time to be bumped to 7 seconds.

Loops Executed by a Single Thread

Executes two loops consecutively in a single-threaded program. One loop must complete before the other can begin. The total elapsed time is the sum of times taken by each loop.

```
<$nopage>
#!/usr/bin/env python

```

```
from time import sleep, time, ctime

def loop0():
print 'start loop 0 at:', ctime(time())
sleep(4)
print 'loop 0 done at:', ctime(time())

def loop1():
print 'start loop 1 at:', ctime(time())
sleep(2)
print 'loop 1 done at:', ctime(time())

def main():
print 'starting…'
loop0()
loop1()
print 'all DONE at:', ctime(time())

if   name
main()
```

<$nopage>

3.Define threading module of Python?

Python provides several modules to support MT programming, including the thread, threading, and Queue modules. The thread and threading modules allow the programmer to create and manage threads. The thread module provides the basic thread and locking support, while threading provides high-level full-featured thread management. The Queue module allows the user to create a queue data structure which can be shared across multiple threads. We will take a look at these modules individually, present a good number of examples, and a couple of intermediate-sized applications.

threadModule

Let's take a look at what the thread module has to offer. In addition to being able to spawn threads, the threadmodule also provides a basic synchronization data structure called a *lock object* (a.k.a. primitive lock, simple lock, mutual exclusion lock, mutex, binary semaphore). As we mentioned earlier, such synchronization primitives go hand-in- hand with thread management. A list of the more commonly-used thread functions and

LockTypelock object methods:

threadModule and Lock Objects

Function/Method	*Description*
thread*Module Functions*	
start_new_thread(*function, args, kwargs*=None)	spawns a new thread and execute *function*with the given *args*and optional *kwargs*
allocate_lock()	allocates LockTypelock object
exit()	instructs a thread to exit
LockType*Lock Object Methods*	
acquire(*wait*=None)	attempts to acquire lock object
locked()	returns 1 if lock acquired, 0 otherwise

release()	releases lock

The key function of the thread module is start_new_thread(). Its syntax is exactly that of the apply() built-in function, taking a function along with arguments and optional keyword arguments. The difference is that instead of the main thread executing the function, a new thread is spawned to invoke the function.

Using the threadModule

The same loops from onethr.py are executed, but this time using the simple multithreaded mechanism provided by the thread module. The two loops are executed concurrently (with the shorter one finishing first, obviously), and the total elapsed time is only as long as the slowest thread rather than the total time for each separately.

<$nopage>

```
#!/usr/bin/env python

import thread
from time import sleep, time, ctime

def loop0():
print 'start loop 0 at:', ctime(time())
sleep(4)
print 'loop 0 done at:', ctime(time())

def loop1():
print 'start loop 1 at:', ctime(time())
```

```
sleep(2)

print 'loop 1 done at:', ctime(time())


def main():

print 'starting threads…'

thread.start_new_thread(loop0, ())

thread.start_new_thread(loop1, ())

sleep(6)

print 'all DONE at:', ctime(time))


if name == '_main__':

main()

<$nopage>
```

UNIT IV

1.Describe GUI programming in detail.

Graphical User Interfaces:

In graphical user interface we are going to discuss about

- Behavior of terminal based programs and GUI-based programs
- Coding simple GUI-based programs
- Other useful GUI resources

1. Behavior of terminal based programs and GUI-based programs:

Terminal-Based Version:

The terminal-based version of the bouncy program displays a greeting and then a menu of

command options. The user is prompted for a command number and then enters a number from the keyboard. The program responds by either terminating execution, prompting for the information for a bouncing ball, or printing a message indicating an unrecognized command. After the program processes a command, it displays the menu again, and the same process starts over.

This terminal-based user interface has several obvious effects on its users:

- The user is constrained to reply to a definite sequence of prompts for inputs. Once an input is entered, there is no way to back up and change it.
- To obtain results for a different set of input data, the user must wait for the command menu to be displayed again. At that point, the same command and all of the other inputs must be re-entered.
- The user can enter an unrecognized command.

Each of these effects poses a problem for users that can be solved by converting the interface to a

.

2. Coding simple GUI-based programs:

 There are many libraries and toolkits of GUI components available to the Python programmer, but in this chapter we use just two: tkinter and tkinter.messagebox.

 Both are standard modules that come with any Python installation.

 tkinter includes classes for windows and numerous types of window objects.

 tkinter.messagebox includes functions for several standard popup dialog boxes.

 We start with some short demo programs that illustrate each type of GUI component

3. Other useful GUI resources:

Other useful GUI resources are

1. Colors
2. Text Attributes
3. Sizing and Justifying an Entry
4. Sizing the Main Window
5. Grid Attributes
6. Using Nested Frames to Organize Components
7. Multi-Line Text Widgets
8. Scrolling List Boxes
9. Mouse Events
10. Keyboard Events

2.Explain about Tkinter module?

Tkinter is the standard GUI library for Python. Python when combined with Tkinter provides a fast and easy way to create GUI applications. Tkinter provides a powerful object-oriented interface to the Tk GUI toolkit.

Creating a GUI application using Tkinter is an easy task. All you need to do is perform the following steps −

- Import the *Tkinter* module.

- Create the GUI application main window

- Add one or more of the above-mentioned widgets to the GUI application.

- Enter the main event loop to take action against each event triggered by the user.

Example

python

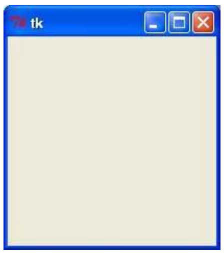
```
import Tkinter

top = Tkinter.Tk()

# Code to add widgets will go here...


top.mainloop()
```

This would create a following window −



3.Write a short note on CGI in Python?

The Web was initially developed to be a global online repository or archive of (mostly educational and research-oriented) documents. Such pieces of information generally come in the form of static text and usually in HTML (*HyperText Markup Language*). [Many documents also exist in plain text, Adobe *Portable Document Format* (PDF), or *Extensible Markup Language* (XML) format, a generalized markup language.]

HTML is not as much of a *language* as it is a text formatter, indicating changes in font types, sizes, and styles. The main feature of HTML is in its hypertext capability, document text that is in one way or another highlighted to point to another document in a related context to the original. Such a document can be accessed by a mouse click or other user selection mechanism. These (static) HTML documents live on the Web server and are sent to clients when and if requested.

As the Internet and Web services evolved, there grew a need to process user input. Online retailers needed to be able to take individual orders, and online banks and search engine portals needed to create accounts for individual users. Thus fill-out forms were invented, and became the only way a Web site can get specific information from users (until Java applets came along).This, in turn, required the HTML now be generated on the fly, for each client submitting user-specific data.

Now Web servers are only really good at one thing, getting a user request for a file and returning that file (i.e., an HTML file) to the client. They do not have the "brains" to be able to deal with user-specific data such as those which come from fields. Not being their responsibility, Web servers farm out such requests to external applications which create the dynamically-generated HTML that is returned to the client.

The entire process begins when the Web server receives a client request (i.e., GET or POST) and calls the appropriate application. It then waits for the resulting HTML— meanwhile, the client also waits. Once the application has completed, it passes the dynamically-generated HTML back to the server, who then (finally) forwards it back to the user. This process of the server receiving a form, contacting an external application, receiving and returning the newly-generated HTML takes place through what is called the Web server's *Common Gateway Interface* (CGI). An overview of how CGI works is illustrated in which shows you the execution and data flow, step-by-step from when a user submits a form until the resulting Web page is returned.

CGI Applications

A CGI application is slightly different from a typical program. The primary differences are in the input, output, and user interaction aspects of a computer program.

When a CGI script starts, it will have the additional functionality of retrieving the user- supplied data, the input for the program comes from the data via the Web client, not a user on the server machine nor a disk file.

The output differs in that any data sent to standard output will be sent back to the connected Web client rather than to the screen, GUI window, or disk file. The data that is sent back must be a set of valid headers followed by HTML. If it is not and the Web client is a browser, an error (specifically, an Internal Server Error) will occur because Web clients such as browsers understand only valid HTTP data (i.e., MIME headers and HTML).

Finally, as you can probably guess, there is no user interaction with the script. All communication occurs among the Web client (on behalf of a user), the Web server, and the CGI application.

UNIT V

1.Explain database API in Python?

The Python standard for database interfaces is the Python DB-API. Most Python database interfaces adhere to this standard.

You can choose the right database for your application. Python Database API supports a wide range of database servers such as −

GadFly

mSQL

MySQL

PostgreSQL

Microsoft SQL Server 2000

Informix

Interbase

Oracle

Sybase

Here is the list of available Python database interfaces: Python Database Interfaces and APIs. You must download a separate DB API module for each database you need to access. For example, if you need to access an Oracle database as well as a MySQL database, you must download both the Oracle and the MySQL database modules.

The DB API provides a minimal standard for working with databases using Python structures and syntax wherever possible. This API includes the following −

Importing the API module.

Acquiring a connection with the database.

Issuing SQL statements and stored procedures.

Closing the connection

We would learn all the concepts using MySQL, so let us talk about MySQLdb module.

2.What is MySQLdb?

MySQLdb is an interface for connecting to a MySQL database server from Python. It implements the Python Database API v2.0 and is built on top of the MySQL C API.

How do I Install MySQLdb?

Before proceeding, you make sure you have MySQLdb installed on your machine. Just type the following in your Python script and execute it −

```
#!/usr/bin/python


import MySQLdb
```

If it produces the following result, then it means MySQLdb module is not installed −

```
Traceback (most recent call last):
  File "test.py", line 3, in <module>
    import MySQLdb
ImportError: No module named MySQLdb
```

To install MySQLdb module, use the following command −

For Ubuntu, use the following command -

```
$ sudo apt-get install python-pip python-dev libmysqlclient-dev
```

For Fedora, use the following command -

```
$ sudo dnf install python python-devel mysql-devel redhat-rpm-config gcc
```

For Python command prompt, use the following command -

```
pip install MySQL-python
```

Note − Make sure you have root privilege to install above module.

Database Connection

Before connecting to a MySQL database, make sure of the followings −

You have created a database TESTDB.

You have created a table EMPLOYEE in TESTDB.

This table has fields FIRST_NAME, LAST_NAME, AGE, SEX and INCOME.

User ID "testuser" and password "test123" are set to access TESTDB.

Python module MySQLdb is installed properly on your machine.

You have gone through MySQL tutorial to understand MySQL Basics.

Example

Following is the example of connecting with MySQL database "TESTDB"

```
#!/usr/bin/python

import MySQLdb

# Open database connection

db = MySQLdb.connect("localhost","testuser","test123","TESTDB" )

# prepare a cursor object using cursor() method

cursor = db.cursor()

# execute SQL query using execute() method.

cursor.execute("SELECT VERSION()")

# Fetch a single row using fetchone() method.

data = cursor.fetchone()

print "Database version : %s " % data

# disconnect from server

db.close()
```

While running this script, it is producing the following result in my Linux machine.

```
Database version : 5.0.45
```

If a connection is established with the datasource, then a Connection Object is returned and saved into db for further use, otherwise db is set to None. Next, db object is used to create a cursor object, which in turn is used to execute SQL queries. Finally, before coming out, it ensures that database connection is closed and resources are released.

3.List out Database operations?

Creating Database Table

Once a database connection is established, we are ready to create tables or records into the database tables using execute method of the created cursor.

Example

Let us create Database table EMPLOYEE −

```
#!/usr/bin/python

import MySQLdb

# Open database connection

db = MySQLdb.connect("localhost","testuser","test123","TESTDB" )

# prepare a cursor object using cursor() method

cursor = db.cursor()

# Drop table if it already exist using execute() method.

cursor.execute("DROP TABLE IF EXISTS EMPLOYEE")

# Create table as per requirement

sql = """CREATE TABLE EMPLOYEE (

        FIRST_NAME  CHAR(20) NOT NULL,

        LAST_NAME  CHAR(20),
      AGE INT,
      SEX CHAR(1),
      INCOME FLOAT )"""

cursor.execute(sql)
# disconnect from server
db.close()
```

INSERT Operation

It is required when you want to create your records into a database table.

Example

The following example, executes SQL *INSERT* statement to create a record into EMPLOYEE table –

```
#!/usr/bin/python
import MySQLdb
# Open database connection
db = MySQLdb.connect("localhost","testuser","test123","TESTDB" )
# prepare a cursor object using cursor() method
cursor = db.cursor()
# Prepare SQL query to INSERT a record into the database.
sql = """INSERT INTO EMPLOYEE(FIRST_NAME,
      LAST_NAME, AGE, SEX, INCOME)
      VALUES ('Mac', 'Mohan', 20, 'M', 2000)"""
try:   # Execute the SQL command
  cursor.execute(sql)
  # Commit your changes in the database
  db.commit()
except:
   # Rollback in case there is any error
   db.rollback()


# disconnect from server
db.close()
```

Above example can be written as follows to create SQL queries dynamically −

```
#!/usr/bin/python
import MySQLdb
# Open database connection
db = MySQLdb.connect("localhost","testuser","test123","TESTDB" )
# prepare a cursor object using cursor() method
cursor = db.cursor()
# Prepare SQL query to INSERT a record into the database.
sql = "INSERT INTO EMPLOYEE(FIRST_NAME, \
       LAST_NAME, AGE, SEX, INCOME) \
       VALUES ('%s', '%s', '%d', '%c', '%d' )" % \
       ('Mac', 'Mohan', 20, 'M', 2000)
try:
   # Execute the SQL command
   cursor.execute(sql)
   # Commit your changes in the database
   db.commit()
```

```
except:
   # Rollback in case there is any error
   db.rollback()
# disconnect from server
db.close()
```

Example

Following code segment is another form of execution where you can pass parameters directly −

```
..................................
user_id = "test123"
password = "password"

con.execute('insert into Login values("%s", "%s")' % \
             (user_id, password))
..................................
```

READ Operation

READ Operation on any database means to fetch some useful information from the database.

Once our database connection is established, you are ready to make a query into this database. You can use either fetchone() method to fetch single record or fetchall() method to fetech multiple values from a database table.

fetchone() − It fetches the next row of a query result set. A result set is an object that is returned when a cursor object is used to query a table.

fetchall() − It fetches all the rows in a result set. If some rows have already been extracted from the result set, then it retrieves the remaining rows from the result set.

rowcount − This is a read-only attribute and returns the number of rows that were affected by an execute() method.

Example

The following procedure queries all the records from EMPLOYEE table having salary more than 1000 −

```
#!/usr/bin/python


import MySQLdb

# Open database connection

db = MySQLdb.connect("localhost","testuser","test123","TESTDB" )

# prepare a cursor object using cursor() method

cursor = db.cursor()

sql = "SELECT * FROM EMPLOYEE \

       WHERE INCOME > '%d'" % (1000)

try:

   # Execute the SQL command

   cursor.execute(sql)

   # Fetch all the rows in a list of lists.

   results = cursor.fetchall()

   for row in results:

      fname = row[0]

      lname = row[1]

      age = row[2]

      sex = row[3]

      income = row[4]
```

```
    # Now print fetched result
    print "fname=%s,lname=%s,age=%d,sex=%s,income=%d" % \
        (fname, lname, age, sex, income )
except:
  print "Error: unable to fecth data"
# disconnect from server
db.close()
```

This will produce the following result −

```
fname=Mac, lname=Mohan, age=20, sex=M, income=2000
```

UPDATE Operation

UPDATE Operation on any database means to update one or more records, which are already available in the database.

The following procedure updates all the records having SEX as 'M'. Here, we increase AGE of all the males by one year.

Example

```
#!/usr/bin/python
import MySQLdb
# Open database connection
db = MySQLdb.connect("localhost","testuser","test123","TESTDB" )
# prepare a cursor object using cursor() method
cursor = db.cursor()
# Prepare SQL query to UPDATE required records
sql = "UPDATE EMPLOYEE SET AGE = AGE + 1
                 WHERE SEX = '%c'" % ('M')
```

```
try:   # Execute the SQL command
   cursor.execute(sql)
   # Commit your changes in the database
   db.commit()
except:
   # Rollback in case there is any error
   db.rollback()
# disconnect from server


db.close()
```

DELETE Operation

DELETE operation is required when you want to delete some records from your database. Following is the procedure to delete all the records from EMPLOYEE where AGE is more than 20 −

Example

```
#!/usr/bin/python
import MySQLdb
# Open database connection
db = MySQLdb.connect("localhost","testuser","test123","TESTDB" )
# prepare a cursor object using cursor() method
cursor = db.cursor()
# Prepare SQL query to DELETE required records
sql = "DELETE FROM EMPLOYEE WHERE AGE > '%d'" % (20)
try:
```

```
# Execute the SQL command
   cursor.execute(sql)
   # Commit your changes in the database
   db.commit()
except:
   # Rollback in case there is any error
   db.rollback()
# disconnect from server
db.close()
```

Performing Transactions

Transactions are a mechanism that ensures data consistency. Transactions have the following four properties −

Atomicity − Either a transaction completes or nothing happens at all.

Consistency − A transaction must start in a consistent state and leave the system in a consistent state.

Isolation − Intermediate results of a transaction are not visible outside the current transaction.

Durability − Once a transaction was committed, the effects are persistent, even after a system failure.

The Python DB API 2.0 provides two methods to either *commit* or *rollback* a transaction.

Example

You already know how to implement transactions. Here is again similar example −

```
# Prepare SQL query to DELETE required records
sql = "DELETE FROM EMPLOYEE WHERE AGE > '%d'" % (20)
try:
  # Execute the SQL command
  cursor.execute(sql)
  # Commit your changes in the database
  db.commit()
except:
  # Rollback in case there is any error
  db.rollback()
```

COMMIT Operation

Commit is the operation, which gives a green signal to database to finalize the changes, and after this operation, no change can be reverted back.

Here is a simple example to call commit method.

```
db.commit()
```

ROLLBACK Operation

If you are not satisfied with one or more of the changes and you want to revert back those changes completely, then use rollback() method.

Here is a simple example to call rollback() method.

```
db.rollback()
```

Disconnecting Database

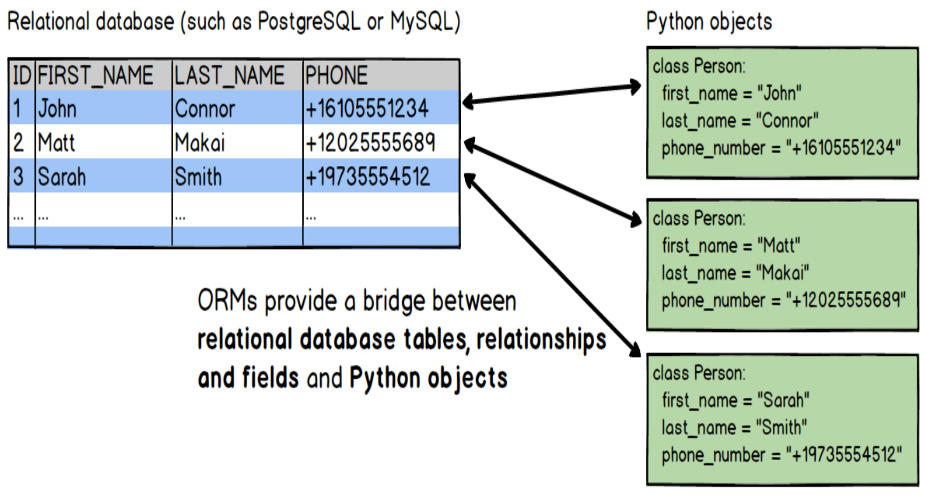
To disconnect Database connection, use close() method.

```
db.close()
```

If the connection to a database is closed by the user with the close() method, any outstanding transactions are rolled back by the DB. However, instead of depending on any of DB lower level implementation details, your application would be better off calling commit or rollback explicitly.

4.Explain ORM?

An object-relational mapper (ORM) is a code library that automates the transfer of data stored in relational databases tables into objects that are more commonly used in application code.


Relational database (such as PostgreSQL or MySQL)
Python objects
ID
FIRST_NAME
LAST_NAME
PHONE
1 John Connor +16105551234
2 Matt Makai +12025555689
3 Sarah Smith +19735554512
class Person:
first_name = "John"
last_name = "Connor"
phone_number = "+16105551234"
class Person:
first_name = "Matt"
last_name = "Makai"
phone_number = "+12025555689"
class Person:
first_name = "Sarah"
last_name = "Smith"
phone_number = "+19735554512"
ORMs provide a bridge between relational database tables, relationships and fields and Python objects

Why are ORMs useful?

ORMs provide a high-level abstraction upon a relational database that allows a developer to write Python code instead of SQL to create, read, update and delete data and schemas in their database. Developers can use the programming language they are comfortable with to work with a database instead of writing SQL statements or stored procedures.

For example, without an ORM a developer would write the following SQL statement to retrieve every row in the USERS table where the zip_code column is 94107:

```
SELECT * FROM USERS WHERE zip_code=94107;
```

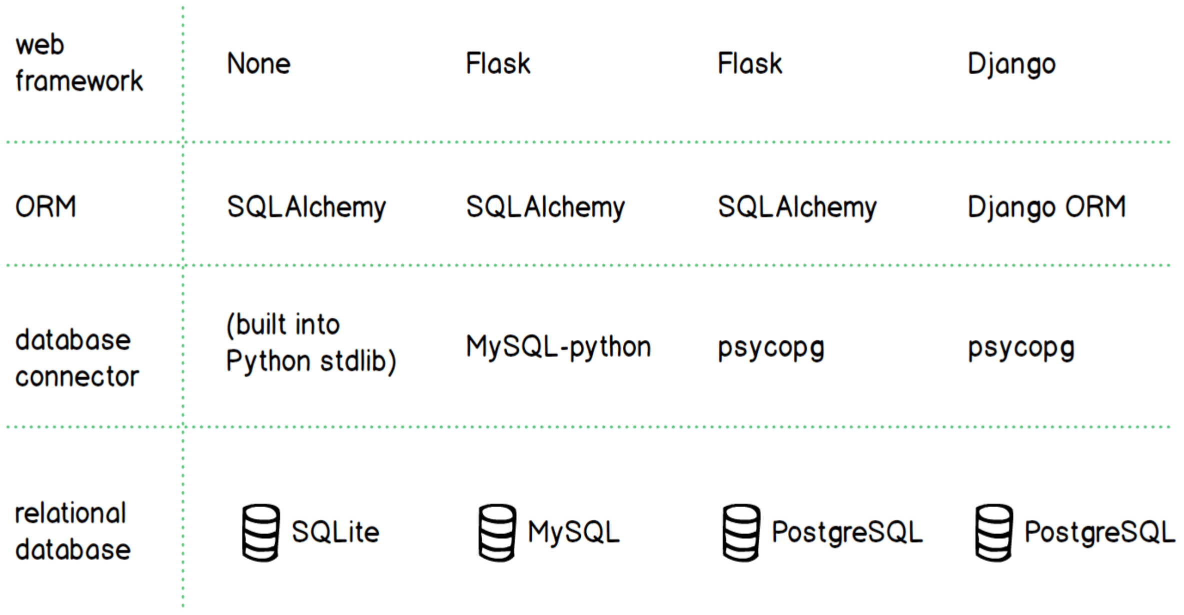
The equivalent Django ORM query would instead look like the following Python code:

```
# obtain everyone in the 94107 zip code and assign to users variable

users = Users.objects.filter(zip_code=94107)
```

The ability to write Python code instead of SQL can speed up web application development, especially at the beginning of a project. The potential development speed boost comes from not having to switch from Python code into writing declarative paradigm SQL statements. While some software developers may not mind switching back and forth between languages, it's typically easier to knock out a prototype or start a web application using a single programming language.

ORMs also make it theoretically possible to switch an application between various relational databases. For example, a developer could use SQLite for local development and MySQL in production. A production application could be switched from MySQL to PostgreSQL with minimal code modifications.



web framework	None	Flask	Flask	Django
ORM	SQLAlchemy	SQLAlchemy	SQLAlchemy	Django ORM
database connector	(built into Python stdlib)	MySQL-python	psycopg	psycopg
relational database	SQLite	MySQL	PostgreSQL	PostgreSQL

The above table shows for example that SQLAlchemy can work with varying web frameworks and database connectors. Developers can also use ORMs without a web framework, such as when creating a data analysis tool or a batch script without a user interface.

Operator	Description	Example
==	If the values of two operands are equal, then the condition becomes true.	(a == b) is not true.
!=	If values of two operands are not equal, then condition becomes true.	(a != b) is true.
<>	If values of two operands are not equal, then condition becomes true.	(a <> b) is true. This is similar to != operator.
>	If the value of left operand is greater than the value of right operand, then condition becomes true.	(a > b) is not true.
<	If the value of left operand is less than the value of right operand, then condition becomes true.	(a < b) is true.