B.Sc. (Computer Science)
III - YEAR/ V - SEMESTER
THEORY PAPER – VI (Elective - IC)
Programming in Python

Scheme of Instruction	Scheme of Examination
Total durations Hrs : 60	Max. Marks : 100
Hours/Week : 05(3T+2P)	Internal Examination :30
Credits : 4	SBT : 10
Instruction Mode: Lecture +practical	External Examination :60
Course Code : BS.07.201.14C.T	Exam Duration : 3 Hrs
Course Objectives:	
To prepare the students with the knowledge of concepts of Programming in Python	
Course Outcomes:	
On completion of the course the student will • Be able to do basic programming in python • Gain knowledge on CGI and GUI Programming	

UNIT - I: Introduction to Programming in Python.
Introduction to Programming in Python**:**
What Is Python? Features of Python, Python environment set up: Installing Python, Running Python, Python Documentation, Structure of a Python Program
Basics of Programming in Python**:**
Input statement, output statement, variables, operators, numbers, Literals, strings, lists and tuples, dictionaries.

UNIT - II: Conditionals, Loops and Functions.
Conditionals and Loops: if statement, else Statement, elif Statement, while Statement, for Statement break Statement, continue Statement, pass Statement.
Functions: Built-in Functions, User defined functions: Defining a Function, Calling a Function, Various Function Arguments.

UNIT - III: Files, Modules and Introduction to Advanced Python.
Files**:** File Objects, File Built-in Methods, File Built-in Attributes, Standard Files, Command-line Arguments
Modules: Modules and Files, Namespaces, Importing Modules, Importing Module Attributes, Module Built-in Functions, Packages.
Introduction to Advanced Python**:** Classes and objects declaration, Inheritance, Regular Expressions.

UNIT - IV: Python GUI & CGI Programming and Python database connectivity.
Python GUI Programming (Tkinter): Tkinter Programming example, Tkinter widges, standard attributes, geometry management

Python CGI Programming: CGI Architecture, First CGI Program, HTTP Header, CGI Environment Variables, GET and POST Methods, Simple FORM Example: Using GET Method, Passing Information Using POST Method

Python database connectivity**:** Establishing connection, insert, retrieve, delete, and rollback and commit operations.

References:

1.Core Python Programming Wesley J. Chun Publisher: Prentice Hall PTR First Edition
2.T. Budd, Exploring Python, TMH, 1st Ed, 2011
3.Python Tutorial/Documentation www.python.or 2010
4.Allen Downey, Jeffrey Elkner, Chris Meyers , How to think like a computer scientist : learning with Python , Freely available online.2015
5.Web Resource: http://interactivepython.org/courselib/static/pythonds

B.Sc. (Computer Science)
III – YEAR / V - SEMESTER
PRACTICAL PAPER – VI (Elective - IC)
Programming in Python Lab Question Bank

Subject Code: BS.07.201.14C.P

1. Write a Python program to get the largest number from a list.
2. a) Write a Python script to concatenate two dictionaries to create a new one.
   b) Write a Python program to sort a dictionary by key.
3. a) Write a Python program to create a tuple with different data types.
   b) Write a Python program to find the repeated items of a tuple.
4. Write a Python program that accepts a word from the user and reverse it.
5. Write a phython program to demonstrate functions.
6. Write a phython program to demonstrate classes.
7. Write a program to perform file operations.
8. Write a Python program to check that a string contains only a certain set of characters (in this case a-z, A-Z and 0-9)
9. Develop following GUI components:
   a) create a window
   b) create a frame
   c) create canvas
   d) create message widget
   e) scale
10. Develop following GUI components:
    a) To create Entry
    b) Radio button
    c) checkbox
    d) menu
    e) Button
    f) List box
11. Develop a form and use GET Method,
12. Develop a form and Pass Information Using POST Method.
13. Write a python program to insert data into table.
14. Write a python program to retrieve data from data base
15. Write a python program to perform Rollback and commit operations

UNIT - I

Introduction to PYTHON: (introduced by Guido Van Rossum in early 80s)

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

- **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.
- **Python is Interactive** − You can actually sit at a Python prompt and interact with the interpreter directly to write your programs.
- **Python is Object-Oriented** − Python supports Object-Oriented style or technique of programming that encapsulates code within objects.
- **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.

Python Features

Python's features include −

- **Easy-to-learn** − Python has few keywords, simple structure, and a clearly defined syntax. This allows the student to pick up the language quickly.
- **Easy-to-read** − Python code is more clearly defined and visible to the eyes.
- **Easy-to-maintain** − Python's source code is fairly easy-to-maintain.
- **A broad standard library** − Python's bulk of the library is very portable and cross-platform compatible on UNIX, Windows, and Macintosh.
- **Interactive Mode** − Python has support for an interactive mode which allows interactive testing and debugging of snippets of code.
- **Portable** − Python can run on a wide variety of hardware platforms and has the same interface on all platforms.
- **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.
- **Databases** − Python provides interfaces to all major commercial databases.
- **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.
- **Scalable** − Python provides a better structure and support for large programs than shell scripting.
- It supports functional and structured programming methods as well as OOP.
- It can be used as a scripting language or can be compiled to byte-code for building large applications.
- It provides very high-level dynamic data types and supports dynamic type checking.
- IT supports automatic garbage collection.
- It can be easily integrated with C, C++, COM, ActiveX, CORBA, and Java.

Python - Environment Setup

Python is available on a wide variety of platforms including Linux and Mac OS X.

Getting Python

The most up-to-date and current source code, binaries, documentation, news, etc., is available on the official website of Python https://www.python.org/

You can download Python documentation from https://www.python.org/doc/. The documentation is available in HTML, PDF, and PostScript formats.

Installing Python

Python distribution is available for a wide variety of platforms. You need to download only the binary code applicable for your platform and install Python.

If the binary code for your platform is not available, you need a C compiler to compile the source code manually. Compiling the source code offers more flexibility in terms of choice of features that you require in your installation.

Here is a quick overview of installing Python on various platforms −

Unix and Linux Installation

Here are the simple steps to install Python on Unix/Linux machine.

- Open a Web browser and go to https://www.python.org/downloads/.
- Follow the link to download zipped source code available for Unix/Linux.
- Download and extract files.
- Editing the *Modules/Setup* file if you want to customize some options.
- run ./configure script
- make
- make install

This installs Python at standard location */usr/local/bin* and its libraries at */usr/local/lib/pythonXX* where XX is the version of Python.

Windows Installation

Here are the steps to install Python on Windows machine.

- Open a Web browser and go to https://www.python.org/downloads/.
- Follow the link for the Windows installer *python-XYZ.msi* file where XYZ is the version you need to install.
- To use this installer *python-XYZ.msi*, the Windows system must support Microsoft Installer 2.0. Save the installer file to your local machine and then run it to find out if your machine supports MSI.
- Run the downloaded file. This brings up the Python install wizard, which is really easy to use. Just accept the default settings, wait until the install is finished, and you are done.

Setting up PATH

Programs and other executable files can be in many directories, so operating systems provide a search path that lists the directories that the OS searches for executables.

The path is stored in an environment variable, which is a named string maintained by the operating system. This variable contains information available to the command shell and other programs.

The **path** variable is named as PATH in Unix or Path in Windows (Unix is case sensitive; Windows is not).

In Mac OS, the installer handles the path details. To invoke the Python interpreter from any particular directory, you must add the Python directory to your path.

Setting path at Unix/Linux

To add the Python directory to the path for a particular session in Unix −

- **In the csh shell** − type setenv PATH "$PATH:/usr/local/bin/python" and press Enter.
- **In the bash shell (Linux)** − type export ATH="$PATH:/usr/local/bin/python" and press Enter.
- **In the sh or ksh shell** − type PATH="$PATH:/usr/local/bin/python" and press Enter.
- **Note** − /usr/local/bin/python is the path of the Python directory

Setting path at Windows

To add the Python directory to the path for a particular session in Windows −

At the command prompt − type path %path%;C:\Python and press Enter.

Note − C:\Python is the path of the Python directory

Running Python

There are three different ways to start Python −

Interactive Interpreter

You can start Python from Unix, DOS, or any other system that provides you a command-line interpreter or shell window.

Enter **python** the command line.

Start coding right away in the interactive interpreter.

```
$python # Unix/Linux
        or
python% # Unix/Linux
         or
C:> python # Windows/DOS
```

Here is the list of all the available command line options −

Sr.No.	Option & Description
1	**-d** It provides debug output.
2	**-O** It generates optimized bytecode (resulting in .pyo files).
3	**-S** Do not run import site to look for Python paths on startup.
4	**-v** verbose output (detailed trace on import statements).
5	**-X** disable class-based built-in exceptions (just use strings); obsolete starting with version 1.6.
6	**-c cmd** run Python script sent in as cmd string
7	**file** run Python script from given file

Script from the Command-line

A Python script can be executed at command line by invoking the interpreter on your application, as in the following −

```
$python script.py # Unix/Linux
        or
python% script.py # Unix/Linux
        or
C: >python script.py # Windows/DOS
```

Note − Be sure the file permission mode allows execution.

Integrated Development Environment

You can run Python from a Graphical User Interface (GUI) environment as well, if you have a GUI application on your system that supports Python.

- **Unix** − IDLE is the very first Unix IDE for Python.
- **Windows** − PythonWin is the first Windows interface for Python and is an IDE with a GUI.

If you are not able to set up the environment properly, then you can take help from your system admin. Make sure the Python environment is properly set up and working perfectly fine.

Python - Basic Syntax

The Python language has many similarities to Perl, C, and Java. However, there are some definite differences between the languages.

Interactive Mode Programming

Invoking the interpreter without passing a script file as a parameter brings up the following prompt −

```
$ python
2.4.3(#1,Nov112010,13:34:43)
GCC 4.1.220080704(RedHat4.1.2-48)] on linux2
Type"help","copyright","credits"or"license"for more information.
>>>
```

Type the following text at the Python prompt and press the Enter −

```
>>>print"Hello,Python!"                #valid in python 2 and invalid in python 3
```

If you are running new version of Python, then you would need to use print statement with parenthesis as in **print ("Hello, Python!");**. However in Python version 2.4.3, this produces the following result −

```
Hello, Python!
```

Script Mode Programming

Invoking the interpreter with a script parameter begins execution of the script and continues until the script is finished. When the script is finished, the interpreter is no longer active.

Let us write a simple Python program in a script. Python files have extension **.py**. Type the following source code in a test.py file −

```
print"Hello, Python!"
```

We assume that you have Python interpreter set in PATH variable. Now, try to run this program as follows −

```
$ python test.py
```

This produces the following result −

```
Hello, Python!
```

Let us try another way to execute a Python script. Here is the modified test.py file −

```
#!/usr/bin/python

print"Hello, Python!"
```

We assume that you have Python interpreter available in /usr/bin directory. Now, try to run this program as follows −

```
$ chmod +x test.py     # This is to make file executable
$./test.py
```

This produces the following result −

```
Hello, Python!
```

Python Identifiers

A Python identifier is a name used to identify a variable, function, class, module or other object. An identifier starts with a letter A to Z or a to z or an underscore (_) followed by zero or more letters, underscores and digits (0 to 9).

Python does not allow punctuation characters such as @, $, and % within identifiers. Python is a case sensitive programming language. Thus, **Manpower** and **manpower** are two different identifiers in Python.

Here are naming conventions for Python identifiers −

- Class names start with an uppercase letter. All other identifiers start with a lowercase letter.
- Starting an identifier with a single leading underscore indicates that the identifier is private.
- Starting an identifier with two leading underscores indicates a strongly private identifier.

- If the identifier also ends with two trailing underscores, the identifier is a language-defined special name.

Reserved Words

The following list shows the Python keywords. These are reserved words and you cannot use them as constant or variable or any other identifier names. All the Python keywords contain lowercase letters only.

and	del	for	is	raise
assert	elif	from	lambda	return
break	else	global	not	try
class	except	if	or	while
continue	exec	import	pass	with
def	finally	in	print	yield

Lines and Indentation

Python provides no braces to indicate blocks of code for class and function definitions or flow control. Blocks of code are denoted by line indentation, which is rigidly enforced.

The number of spaces in the indentation is variable, but all statements within the block must be indented the same amount. For example −

```
if True:
   print "True"
else:
   print "False"
```

However, the following block generates an error −

```
ifTrue:
print"Answer"
print"True"
else:
print"Answer"
print"False"
```

Thus, in Python all the continuous lines indented with same number of spaces would form a block. The following example has various statement blocks −

Multi-Line Statements

Statements in Python typically end with a new line. Python does, however, allow the use of the line continuation character (\) to denote that the line should continue. For example −

```
total = item_one + \
        item_two + \
        item_three
```

Statements contained within the [], {}, or () brackets do not need to use the line continuation character. For example −

```
days = ['Monday', 'Tuesday', 'Wednesday','Thursday', 'Friday']
```

Quotation in Python

Python accepts single ('), double (") and triple ('' or """) quotes to denote string literals, as long as the same type of quote starts and ends the string.

The triple quotes are used to span the string across multiple lines. For example, all the following are legal −

```
word = 'word'
sentence = "This is a sentence."
```

```
paragraph = """This is a paragraph. It is
made up of multiple lines and sentences."""
```

Comments in Python

A hash sign (#) that is not inside a string literal begins a comment. All characters after the # and up to the end of the physical line are part of the comment and the Python interpreter ignores them.

```
#!/usr/bin/python

# First comment
print"Hello, Python!"# second comment
```

This produces the following result −

```
Hello, Python!
```

You can type a comment on the same line after a statement or expression −

```
name = "St.Joseph's" # This is again comment
```

You can comment multiple lines as follows −

```
# This is a comment.
# This is a comment, too.
# This is a comment, too.
# I said that already.
```

Using Blank Lines

A line containing only whitespace, possibly with a comment, is known as a blank line and Python totally ignores it.

In an interactive interpreter session, you must enter an empty physical line to terminate a multiline statement.

Waiting for the User

The following line of the program displays the prompt, the statement saying "Press the enter key to exit", and waits for the user to take action −

```
#!/usr/bin/python

raw_input ("\n\n Press the enter key to exit.")

var = input ()
```

Here, "\n\n" is used to create two new lines before displaying the actual line. Once the user presses the key, the program ends. This is a nice trick to keep a console window open until the user is done with an application.

Multiple Statements on a Single Line

The semicolon (;) allows multiple statements on the single line given that neither statement starts a new code block. Here is a sample snip using the semicolon −

```
import sys; x ='foo'; sys.stdout.write(x +'\n');
```

Multiple Statement Groups as Suites

A group of individual statements, which make a single code block are called **suites** in Python. Compound or complex statements, such as if, while, def, and class require a header line and a suite.

Header lines begin the statement (with the keyword) and terminate with a colon (:) and are followed by one or more lines which make up the suite. For example −

```
if expression :
   Block of statements
elif expression :
   Block of statements
else :
   Block of statements
```

Command Line Arguments

Many programs can be run to provide you with some basic information about how they should be run. Python enables you to do this with -h –

```
$ python -h
usage: python [option]...[-c cmd |-m mod | file |-][arg]...
Options and arguments (and corresponding environment variables):
-c cmd : program passed in as string(terminates option list)
-d     : debug output from parser (also PYTHONDEBUG=x)
-E     : ignore environment variables (such as PYTHONPATH)
-h     :print this help message and exit

[ etc.]
```

Input using the `input( )` function

A function is defined as a block of organized, reusable code used to perform a single, related action. Python has many built-in functions; you can also create your own. Python has an input function which lets you ask a user for some text input. You call this function to tell the program to stop and wait for the user to key in the data. In Python 2, you have a built-in function `raw_input()`, whereas in Python 3, you have `input()`. The program will resume once the user presses the ENTER or RETURN key. Look at this example to get input from the keyboard using Python 2 in the interactive mode. Your output is displayed in quotes once you hit the ENTER key.

```
>>> input()
I am learning at St.Joseph's #(This is where you type in)

Output:
'I am learning at St.Joseph's' #(The interpreter showing you how the input is
captured.)
```

Output using the `print()` function

To output your data to the screen, use the `print()` function. You can write `print(argument)` and this will print the `argument` in the next line when you press the `ENTER` key.

Definitions to remember: An argument is a value you pass to a function when calling it. A value is a letter or a number. A variable is a name that refers to a value. It begins with a letter. An assignment statement creates new variables and gives them values.

This syntax is valid in both Python 3.x and Python 2.x. For example, if your data is "Guido," you can put "Guido" inside the parentheses `( )` after `print`.

```
>>>print("Guido")
Guido
```

Python - Variable

Variables are nothing but reserved memory locations to store values. This means that when you create a variable you reserve some space in memory.

Based on the data type of a variable, the interpreter allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to variables, you can store integers, decimals or characters in these variables.

Assigning Values to Variables

Python variables do not need explicit declaration to reserve memory space. The declaration happens automatically when you assign a value to a variable. The equal sign (=) is used to assign values to variables.

The operand to the left of the = operator is the name of the variable and the operand to the right of the = operator is the value stored in the variable. For example −

```
#!/usr/bin/python

counter =100   # An integer assignment
miles   =1000.0# A floating point
name    ="John"# A string

print (counter)
print (miles)
print (name)
```

Here, 100, 1000.0 and "John" are the values assigned to *counter*, *miles*, and *name* variables, respectively. This produces the following result −

```
100
1000.0
John
```

Multiple Assignment

Python allows you to assign a single value to several variables simultaneously. For example −

```
a = b = c = 1
```

Here, an integer object is created with the value 1, and all three variables are assigned to the same memory location. You can also assign multiple objects to multiple variables. For example −

```
a,b,c = 1,2,"john"
```

Here, two integer objects with values 1 and 2 are assigned to variables a and b respectively, and one string object with the value "john" is assigned to the variable c.

Standard Data Types

The data stored in memory can be of many types. For example, a person's age is stored as a numeric value and his or her address is stored as alphanumeric characters. Python has various standard data types that are used to define the operations possible on them and the storage method for each of them.

Python has five standard data types −

- Numbers
- String
- List
- Tuple
- Dictionary

Python - Numbers

Number data types store numeric values. They are immutable data types, means that changing the value of a number data type results in a newly allocated object.

Number objects are created when you assign a value to them. For example −

```
var1 = 1
var2 = 10
```

You can also delete the reference to a number object by using the **del** statement. The syntax of the del statement is −

```
Syntax : del var1[,var2[,var3[....,varN]]]
```

You can delete a single object or multiple objects by using the **del** statement. For example −

```
del var
del var_a, var_b
```

Python supports four different numerical types −

- **int (signed integers)** − They are often called just integers or ints, are positive or negative whole numbers with no decimal point.
- **long (long integers)** − Also called longs, they are integers of unlimited size, written like integers and followed by an uppercase or lowercase L.
- **float (floating point real values)** − Also called floats, they represent real numbers and are written with a decimal point dividing the integer and fractional parts. Floats may also be in scientific notation, with E or e indicating the power of 10 (2.5e2 = 2.5 x 10^2 = 250).
- **complex (complex numbers)** − are of the form a + bJ, where a and b are floats and J (or j) represents the square root of -1 (which is an imaginary number). The real part of the number is a, and the imaginary part is b. Complex numbers are not used much in Python programming.

Examples

Here are some examples of numbers

int	long	float	complex
10	51924361L	0.0	3.14j
100	-0x19323L	15.20	45.j
-786	0122L	-21.9	9.322e-36j
080	0xDEFABCECBDAECBFBAEL	32.3+e18	.876j
-0490	535633629843L	-90.	-.6545+0J
-0x260	-052318172735L	-32.54e100	3e+26J
0x69	-4721885298529L	70.2-E12	4.53e-7j

- Python allows you to use a lowercase L with long, but it is recommended that you use only an uppercase L to avoid confusion with the number 1. Python displays long integers with an uppercase L.
- A complex number consists of an ordered pair of real floating point numbers denoted by a + bj, where a is the real part and b is the imaginary part of the complex number.

Number Type Conversion

Python converts numbers internally in an expression containing mixed types to a common type for evaluation. But sometimes, you need to coerce a number explicitly from one type to another to satisfy the requirements of an operator or function parameter.

- Type **int(x)** to convert x to a plain integer.

- Type **long(x)** to convert x to a long integer.
- Type **float(x)** to convert x to a floating-point number.
- Type **complex(x)** to convert x to a complex number with real part x and imaginary part zero.
- Type **complex(x, y)** to convert x and y to a complex number with real part x and imaginary part y. x and y are numeric expressions

Mathematical Functions

Python includes following functions that perform mathematical calculations.

Sr.No.	Function & Returns (description)
1	abs(x) The absolute value of x: the (positive) distance between x and zero.
2	ceil(x) The ceiling of x: the smallest integer not less than x
3	cmp(x, y) -1 if x < y, 0 if x == y, or 1 if x > y
4	exp(x) The exponential of x: e^x
5	fabs(x) The absolute value of x.
6	floor(x) The floor of x: the largest integer not greater than x
7	log(x) The natural logarithm of x, for x> 0
8	log10(x) The base-10 logarithm of x for x> 0.
9	max(x1, x2,...) The largest of its arguments: the value closest to positive infinity
10	min(x1, x2,...) The smallest of its arguments: the value closest to negative infinity
11	modf(x) The fractional and integer parts of x in a two-item tuple. Both parts have the same sign as x. The integer part is returned as a float.
12	pow(x, y) The value of x**y.
13	round(x [,n]) **x** rounded to n digits from the decimal point. Python rounds away from zero as a tie-breaker: round(0.5) is 1.0 and round(-0.5) is -1.0.
14	sqrt(x) The square root of x for x > 0

Random Number Functions

Random numbers are used for games, simulations, testing, security, and privacy applications. Python includes following functions that are commonly used.

Sr.No.	Function & Description
1	choice(seq) A random item from a list, tuple, or string.

2	randrange ([start,] stop [,step]) A randomly selected element from range(start, stop, step)
3	random() A random float r, such that 0 is less than or equal to r and r is less than 1
4	seed([x]) Sets the integer starting value used in generating random numbers. Call this function before calling any other random module function. Returns None.
5	shuffle(lst) Randomizes the items of a list in place. Returns None.
6	uniform(x, y) A random float r, such that x is less than or equal to r and r is less than y

Trigonometric Functions

Python includes following functions that perform trigonometric calculations.

Sr.No.	Function & Description
1	acos(x) Return the arc cosine of x, in radians.
2	asin(x) Return the arc sine of x, in radians.
3	atan(x) Return the arc tangent of x, in radians.
4	atan2(y, x) Return atan(y / x), in radians.
5	cos(x) Return the cosine of x radians.
6	hypot(x, y) Return the Euclidean norm, sqrt(x*x + y*y).
7	sin(x) Return the sine of x radians.
8	tan(x) Return the tangent of x radians.
9	degrees(x) Converts angle x from radians to degrees.
10	radians(x) Converts angle x from degrees to radians.

Mathematical Constants

The module also defines two mathematical constants −

Sr.No.	Constants & Description
1	**pi** The mathematical constant pi.
2	**e** The mathematical constant e.

Python - Strings

Strings are amongst the most popular types in Python. We can create them simply by enclosing characters in quotes. Python treats single quotes the same as double quotes. Creating strings is as simple as assigning a value to a variable. For example −

```
var1 = 'Hello World!'
var2 = "Python Programming"
```

Accessing Values in Strings

Python does not support a character type; these are treated as strings of length one, thus also considered a substring.

To access substrings, use the square brackets for slicing along with the index or indices to obtain your substring. For example −

```
#!/usr/bin/python

var1 = 'Hello World!'
var2 = "Python Programming"

print "var1[0]: ", var1[0]          # valid in python 2.x and invalid in python 3.x
print ("var2[1:5]: ", var2[1:5])    # valid in python 2.x and valid in python 3.x
```

When the above code is executed, it produces the following result −

```
var1[0]:  H
var2[1:5]:  ytho
```

Updating Strings

You can "update" an existing string by (re)assigning a variable to another string. The new value can be related to its previous value or to a completely different string altogether. For example −

```
#!/usr/bin/python
var1 = 'Hello World!'
print ("Updated String :- ", var1[:6] + 'Python')
```

When the above code is executed, it produces the following result −

```
Updated String :-  Hello Python
```

Escape Characters

Following table is a list of escape or non-printable characters that can be represented with backslash notation.

An escape character gets interpreted; in a single quoted as well as double quoted strings.

Backslash notation	Hexadecimal character	“Description”
\a	0x07	Bell or alert
\b	0x08	Backspace
\cx		Control-x
\C-x		Control-x

\e	0x1b	Escape
\f	0x0c	Formfeed
\M-\C-x		Meta-Control-x
\n	0x0a	Newline
\nnn		Octal notation, where n is in the range 0.7
\r	0x0d	Carriage return
\s	0x20	Space
\t	0x09	Tab
\v	0x0b	Vertical tab
\x		Character x
\xnn		Hexadecimal notation, where n is in the range 0.9, a.f, or A.F

String Special Operators

Assume string variable **a** holds 'Hello' and variable **b** holds 'Python', then −

Operator	Description	Example
+	Concatenation - Adds values on either side of the operator	a + b will give HelloPython
*	Repetition - Creates new strings, concatenating multiple copies of the same string	a*3 will give -HelloHelloHello
[]	Slice - Gives the character from the given index	a[1] will give e
[:]	Range Slice - Gives the characters from the given range	a[1:4] will give ell
in	Membership - Returns true if a character exists in the given string	H in a will give 1
not in	Membership - Returns true if a character does not exist in the given string	M not in a will give 1
r/R	Raw String - Suppresses actual meaning of Escape characters. The syntax for raw strings is exactly the same as for normal strings with the exception of the raw string operator, the letter "r," which precedes the quotation marks. The "r" can be lowercase (r) or uppercase (R) and must be placed immediately preceding the first quote mark.	print r'\n' prints \n and print R'\n'prints \n
%	Format - Performs String formatting	See at next section

String Formatting Operator

One of Python's coolest features is the string format operator %. This operator is unique to strings and makes up for the pack of having functions from C's printf() family. Following is a simple example −

```
#!/usr/bin/python

print ("My name is %s and weight is %d kg!") % ('Zara', 21)
```

When the above code is executed, it produces the following result −

```
My name is Zara and weight is 21 kg!
```

Here is the list of complete set of symbols which can be used along with % −

Format Symbol	Conversion
%c	character
%s	string conversion via str() prior to formatting
%i	signed decimal integer
%d	signed decimal integer
%u	unsigned decimal integer
%o	octal integer
%x	hexadecimal integer (lowercase letters)
%X	hexadecimal integer (UPPERcase letters)
%e	exponential notation (with lowercase 'e')
%E	exponential notation (with UPPERcase 'E')
%f	floating point real number
%g	the shorter of %f and %e
%G	the shorter of %f and %E

Other supported symbols and functionality are listed in the following table −

Symbol	Functionality
*	argument specifies width or precision
-	left justification
+	display the sign
<sp>	leave a blank space before a positive number
#	add the octal leading zero ('0') or hexadecimal leading '0x' or '0X', depending on whether 'x' or 'X' were used.
0	pad from left with zeros (instead of spaces)
%	'%%' leaves you with a single literal '%'
(var)	mapping variable (dictionary arguments)
m.n.	m is the minimum total width and n is the number of digits to display after the decimal point (if appl.)

Triple Quotes

Python's triple quotes comes to the rescue by allowing strings to span multiple lines, including verbatim NEWLINEs, TABs, and any other special characters.

The syntax for triple quotes consists of three consecutive **single or double** quotes.

```
#!/usr/bin/python

para_str = """this is a long string that is made up of
several lines and non-printable characters such as
TAB ( \t ) and they will show up that way when displayed.
NEWLINEs within the string, whether explicitly given like
this within the brackets [ \n ], or just a NEWLINE within
the variable assignment will also show up.
"""
print para_str
```

When the above code is executed, it produces the following result. Note how every single special character has been converted to its printed form, right down to the last NEWLINE at the end of the string between the

"up." and closing triple quotes. Also note that NEWLINEs occur either with an explicit carriage return at the end of a line or its escape code (\n) −

```
this is a long string that is made up of
several lines and non-printable characters such as
TAB (	) and they will show up that way when displayed.
NEWLINEs within the string, whether explicitly given like
this within the brackets [
 ], or just a NEWLINE within
the variable assignment will also show up.
```

Raw strings do not treat the backslash as a special character at all. Every character you put into a raw string stays the way you wrote it −

```
#!/usr/bin/python

print 'C:\\nowhere'
```

When the above code is executed, it produces the following result −

```
C:\nowhere
```

Now let's make use of raw string. We would put expression in **r'expression'** as follows −

```
#!/usr/bin/python

print r'C:\\nowhere'
```

When the above code is executed, it produces the following result −

```
C:\\nowhere
```

Unicode String

Normal strings in Python are stored internally as 8-bit ASCII, while Unicode strings are stored as 16-bit Unicode. This allows for a more varied set of characters, including special characters from most languages in the world. I'll restrict my treatment of Unicode strings to the following −

```
#!/usr/bin/python

print (u'Hello, world!')
```

When the above code is executed, it produces the following result −

```
Hello, world!
```

As you can see, Unicode strings use the prefix u, just as raw strings use the prefix r.

Built-in String Methods

Python includes the following built-in methods to manipulate strings −

Sr.No.	Methods with Description
1	capitalize() Capitalizes first letter of string
2	center(width, fillchar) Returns a space-padded string with the original string centered to a total of width columns.

3	count(str, beg= 0,end=len(string)) Counts how many times str occurs in string or in a substring of string if starting index beg and ending index end are given.
4	decode(encoding='UTF-8',errors='strict') Decodes the string using the codec registered for encoding. encoding defaults to the default string encoding.
5	encode(encoding='UTF-8',errors='strict') Returns encoded string version of string; on error, default is to raise a ValueError unless errors is given with 'ignore' or 'replace'.
6	endswith(suffix, beg=0, end=len(string)) Determines if string or a substring of string (if starting index beg and ending index end are given) ends with suffix; returns true if so and false otherwise.
7	expandtabs(tabsize=8) Expands tabs in string to multiple spaces; defaults to 8 spaces per tab if tabsize not provided.
8	find(str, beg=0 end=len(string)) Determine if str occurs in string or in a substring of string if starting index beg and ending index end are given returns index if found and -1 otherwise.
9	index(str, beg=0, end=len(string)) Same as find(), but raises an exception if str not found.
10	isalnum() Returns true if string has at least 1 character and all characters are alphanumeric and false otherwise.
11	isalpha() Returns true if string has at least 1 character and all characters are alphabetic and false otherwise.
12	isdigit() Returns true if string contains only digits and false otherwise.
13	islower() Returns true if string has at least 1 cased character and all cased characters are in lowercase and false otherwise.
14	isnumeric() Returns true if a unicode string contains only numeric characters and false otherwise.
15	isspace() Returns true if string contains only whitespace characters and false otherwise.
16	istitle() Returns true if string is properly "titlecased" and false otherwise.
17	isupper() Returns true if string has at least one cased character and all cased characters are in uppercase and false otherwise.
18	join(seq) Merges (concatenates) the string representations of elements in sequence seq into a string, with separator string.
19	len(string) Returns the length of the string
20	ljust(width[, fillchar]) Returns a space-padded string with the original string left-justified to a total of width columns.
21	lower() Converts all uppercase letters in string to lowercase.
22	lstrip() Removes all leading whitespace in string.
23	maketrans() Returns a translation table to be used in translate function.
24	max(str) Returns the max alphabetical character from the string str.
25	min(str)

	Returns the min alphabetical character from the string str.
26	replace(old, new [, max]) Replaces all occurrences of old in string with new or at most max occurrences if max given.
27	rfind(str, beg=0,end=len(string)) Same as find(), but search backwards in string.
28	rindex(str, beg=0, end=len(string)) Same as index(), but search backwards in string.
29	rjust(width,[, fillchar]) Returns a space-padded string with the original string right-justified to a total of width columns.
30	rstrip() Removes all trailing whitespace of string.
31	split(str="", num=string.count(str)) Splits string according to delimiter str (space if not provided) and returns list of substrings; split into at most num substrings if given.
32	splitlines(num=string.count('\n')) Splits string at all (or num) NEWLINEs and returns a list of each line with NEWLINEs removed.
33	startswith(str, beg=0,end=len(string)) Determines if string or a substring of string (if starting index beg and ending index end are given) starts with substring str; returns true if so and false otherwise.
34	strip([chars]) Performs both lstrip() and rstrip() on string.
35	swapcase() Inverts case for all letters in string.
36	title() Returns "titlecased" version of string, that is, all words begin with uppercase and the rest are lowercase.
37	translate(table, deletechars="") Translates string according to translation table str(256 chars), removing those in the del string.
38	upper() Converts lowercase letters in string to uppercase.
39	zfill (width) Returns original string leftpadded with zeros to a total of width characters; intended for numbers, zfill() retains any sign given (less one zero).
40	isdecimal() Returns true if a unicode string contains only decimal characters and false otherwise.

Python - Lists

The most basic data structure in Python is the **sequence**. Each element of a sequence is assigned a number - its position or index. The first index is zero, the second index is one, and so forth.

Python has six built-in types of sequences, but the most common ones are lists and tuples, which we would see in this tutorial.

There are certain things you can do with all sequence types. These operations include indexing, slicing, adding, multiplying, and checking for membership. In addition, Python has built-in functions for finding the length of a sequence and for finding its largest and smallest elements.

Python Lists

The list is a most versatile datatype available in Python which can be written as a list of comma-separated values (items) between square brackets. Important thing about a list is that items in a list need not be of the same type.

Creating a list is as simple as putting different comma-separated values between square brackets. For example −

```
list1 = ['physics', 'chemistry', 1997, 2000];
list2 = [1, 2, 3, 4, 5 ];
list3 = ["a", "b", "c", "d"]
```

Similar to string indices, list indices start at 0, and lists can be sliced, concatenated and so on.

Accessing Values in Lists

To access values in lists, use the square brackets for slicing along with the index or indices to obtain value available at that index. For example −

```
#!/usr/bin/python

list1 = ['physics', 'chemistry', 1997, 2000];
list2 = [1, 2, 3, 4, 5, 6, 7 ];
print ("list1[0]: ", list1[0])
print ("list2[1:5]: ", list2[1:5])
```

When the above code is executed, it produces the following result −

```
list1[0]:  physics
list2[1:5]:  [2, 3, 4, 5]
```

Updating Lists

You can update single or multiple elements of lists by giving the slice on the left-hand side of the assignment operator, and you can add to elements in a list with the append() method. For example −

```
#!/usr/bin/python

list = ['physics', 'chemistry', 1997, 2000];
print ("Value available at index 2 : ")
print (list[2])
list[2] = 2001;
print ("New value available at index 2 : ")
print (list[2])
```

Note − append() method is discussed in subsequent section.

When the above code is executed, it produces the following result −

```
Value available at index 2 :
1997
New value available at index 2 :
2001
```

Delete List Elements

To remove a list element, you can use either the del statement if you know exactly which element(s) you are deleting or the remove() method if you do not know. For example −

```
#!/usr/bin/python

list1 = ['physics', 'chemistry', 1997, 2000];
print list1
del list1[2];
print "After deleting value at index 2 : "
print list1
```

When the above code is executed, it produces following result −

```
['physics', 'chemistry', 1997, 2000]
After deleting value at index 2 :
['physics', 'chemistry', 2000]
```

Note − remove() method is discussed in subsequent section.

Basic List Operations

Lists respond to the + and * operators much like strings; they mean concatenation and repetition here too, except that the result is a new list, not a string.

In fact, lists respond to all of the general sequence operations we used on strings in the prior chapter.

Python Expression	Results	Description
len([1, 2, 3])	3	Length
[1, 2, 3] + [4, 5, 6]	[1, 2, 3, 4, 5, 6]	Concatenation
['Hi!'] * 4	['Hi!', 'Hi!', 'Hi!', 'Hi!']	Repetition
3 in [1, 2, 3]	True	Membership
for x in [1, 2, 3]: print (x)	1 2 3	Iteration

Indexing, Slicing, and Matrixes

Because lists are sequences, indexing and slicing work the same way for lists as they do for strings.

Assuming following input −

```
L = ['spam', 'Spam', 'SPAM!']
```

Python Expression	Results	Description
L[2]	'SPAM!'	Offsets start at zero
L[-2]	'Spam'	Negative: count from the right
L[1:]	['Spam', 'SPAM!']	Slicing fetches sections

Built-in List Functions & Methods

Python includes the following list functions −

Sr.No.	Function with Description
1	cmp(list1, list2) Compares elements of both lists.
2	len(list) Gives the total length of the list.

3	max(list) Returns item from the list with max value.
4	min(list) Returns item from the list with min value.
5	list(seq) Converts a tuple into list.

Python includes following list methods

Sr.No.	Methods with Description
1	list.append(obj) Appends object obj to list
2	list.count(obj) Returns count of how many times obj occurs in list
3	list.extend(seq) Appends the contents of seq to list
4	list.index(obj) Returns the lowest index in list that obj appears
5	list.insert(index, obj) Inserts object obj into list at offset index
6	list.pop(obj=list[-1]) Removes and returns last object or obj from list
7	list.remove(obj) Removes object obj from list
8	list.reverse() Reverses objects of list in place
9	list.sort([func]) Sorts objects of list, use compare func if given

Python - Tuples

A tuple is a sequence of immutable Python objects. Tuples are sequences, just like lists. The differences between tuples and lists are, the tuples cannot be changed unlike lists and tuples use parentheses, whereas lists use square brackets.

Creating a tuple is as simple as putting different comma-separated values. Optionally you can put these comma-separated values between parentheses also. For example −

```
tup1 = ('physics', 'chemistry', 1997, 2000);
tup2 = (1, 2, 3, 4, 5 );
tup3 = ("a", "b", "c", "d");
```

The empty tuple is written as two parentheses containing nothing −

```
tup1 = ();
```

To write a tuple containing a single value you have to include a comma, even though there is only one value

```
tup1 = (50,);
```

Like string indices, tuple indices start at 0, and they can be sliced, concatenated, and so on.

Accessing Values in Tuples

To access values in tuple, use the square brackets for slicing along with the index or indices to obtain value available at that index. For example −

```
#!/usr/bin/python

tup1 = ('physics', 'chemistry', 1997, 2000);
tup2 = (1, 2, 3, 4, 5, 6, 7 );
print ("tup1[0]: ", tup1[0]);
print ("tup2[1:5]: ", tup2[1:5]);
```

When the above code is executed, it produces the following result −

```
tup1[0]:  physics
tup2[1:5]:  (2, 3, 4, 5)
```

Updating Tuples

Tuples are immutable which means you cannot update or change the values of tuple elements. You are able to take portions of existing tuples to create new tuples as the following example demonstrates −

```
#!/usr/bin/python

tup1 = (12, 34.56);
tup2 = ('abc', 'xyz');

# Following action is not valid for tuples
# tup1[0] = 100;

# So let's create a new tuple as follows
tup3 = tup1 + tup2;
print (tup3);
```

When the above code is executed, it produces the following result −

```
(12, 34.56, 'abc', 'xyz')
```

Delete Tuple Elements

Removing individual tuple elements is not possible. There is, of course, nothing wrong with putting together another tuple with the undesired elements discarded.

To explicitly remove an entire tuple, just use the **del** statement. For example −

```
#!/usr/bin/python

tup = ('physics', 'chemistry', 1997, 2000);
print (tup);
del tup;
print ("After deleting tup : ");
print (tup);
```

This produces the following result. Note an exception raised, this is because after **del tup** tuple does not exist any more −

```
('physics', 'chemistry', 1997, 2000)
After deleting tup :
Traceback (most recent call last):
   File "test.py", line 9, in <module>
      print tup;
```

```
NameError: name 'tup' is not defined
```

Basic Tuples Operations

Tuples respond to the + and * operators much like strings; they mean concatenation and repetition here too, except that the result is a new tuple, not a string.

In fact, tuples respond to all of the general sequence operations we used on strings in the prior chapter −

Python Expression	Results	Description
len((1, 2, 3))	3	Length
(1, 2, 3) + (4, 5, 6)	(1, 2, 3, 4, 5, 6)	Concatenation
('Hi!',) * 4	('Hi!', 'Hi!', 'Hi!', 'Hi!')	Repetition
3 in (1, 2, 3)	True	Membership
for x in (1, 2, 3): print x,	1 2 3	Iteration

Indexing, Slicing, and Matrixes

Because tuples are sequences, indexing and slicing work the same way for tuples as they do for strings. Assuming following input −

```
L = ('Spam', 'SPAM!')
```

Python Expression	Results	Description
L[2]	'SPAM!'	Offsets start at zero
L[-2]	'Spam'	Negative: count from the right
L[1:]	['Spam', 'SPAM!']	Slicing fetches sections

No Enclosing Delimiters

Any set of multiple objects, comma-separated, written without identifying symbols, i.e., brackets for lists, parentheses for tuples, etc., default to tuples, as indicated in these short examples −

```
#!/usr/bin/python

print ('abc', -4.24e93, 18+6.6j, 'xyz');
x, y = 1, 2;
print ("Value of x , y : ", x,y);
```

When the above code is executed, it produces the following result −

```
abc -4.24e+93 (18+6.6j) xyz
Value of x , y : 1 2
```

Built-in Tuple Functions

Python includes the following tuple functions −

Sr.No.	Function with Description
1	cmp(tuple1, tuple2) Compares elements of both tuples.
2	len(tuple) Gives the total length of the tuple.
3	max(tuple) Returns item from the tuple with max value.
4	min(tuple) Returns item from the tuple with min value.
5	tuple(seq) Converts a list into tuple.

Python - Dictionary

Each key is separated from its value by a colon (:), the items are separated by commas, and the whole thing is enclosed in curly braces. An empty dictionary without any items is written with just two curly braces, like this: {}.

Keys are unique within a dictionary while values may not be. The values of a dictionary can be of any type, but the keys must be of an immutable data type such as strings, numbers, or tuples.

Accessing Values in Dictionary

To access dictionary elements, you can use the familiar square brackets along with the key to obtain its value. Following is a simple example −

```
#!/usr/bin/python

dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
print "dict['Name']: ", dict['Name']
print "dict['Age']: ", dict['Age']
```

When the above code is executed, it produces the following result −

```
dict['Name']:  Zara
dict['Age']:  7
```

If we attempt to access a data item with a key, which is not part of the dictionary, we get an error as follows −

```
#!/usr/bin/python

dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}

print "dict['Alice']: ", dict['Alice']
```

When the above code is executed, it produces the following result −

```
dict['Alice']:
Traceback (most recent call last):
   File "test.py", line 4, in <module>
      print "dict['Alice']: ", dict['Alice'];
KeyError: 'Alice'
```

Updating Dictionary

You can update a dictionary by adding a new entry or a key-value pair, modifying an existing entry, or deleting an existing entry as shown below in the simple example −

```
#!/usr/bin/python

dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
dict['Age'] = 8; # update existing entry
dict['School'] = "DPS School"; # Add new entry

print "dict['Age']: ", dict['Age']
print "dict['School']: ", dict['School']
```

When the above code is executed, it produces the following result −

```
dict['Age']:  8
dict['School']:  DPS School
```

Delete Dictionary Elements

You can either remove individual dictionary elements or clear the entire contents of a dictionary. You can also delete entire dictionary in a single operation.

To explicitly remove an entire dictionary, just use the **del** statement. Following is a simple example −

```
#!/usr/bin/python

dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
del dict['Name']; # remove entry with key 'Name'
dict.clear();     # remove all entries in dict
del dict ;        # delete entire dictionary

print "dict['Age']: ", dict['Age']
print "dict['School']: ", dict['School']
```

This produces the following result. Note that an exception is raised because after **del dict** dictionary does not exist any more −

```
dict['Age']:
Traceback (most recent call last):
   File "test.py", line 8, in <module>
      print "dict['Age']: ", dict['Age'];
TypeError: 'type' object is unsubscriptable
```

Note − del() method is discussed in subsequent section.

Properties of Dictionary Keys

Dictionary values have no restrictions. They can be any arbitrary Python object, either standard objects or user-defined objects. However, same is not true for the keys.

There are two important points to remember about dictionary keys −

(a) More than one entry per key not allowed. Which means no duplicate key is allowed. When duplicate keys encountered during assignment, the last assignment wins. For example −

```
#!/usr/bin/python

dict = {'Name': 'Zara', 'Age': 7, 'Name': 'Manni'}
print "dict['Name']: ", dict['Name']
```

When the above code is executed, it produces the following result −

```
dict['Name']:  Manni
```

(b) Keys must be immutable. Which means you can use strings, numbers or tuples as dictionary keys but something like ['key'] is not allowed. Following is a simple example −

```
#!/usr/bin/python

dict = {['Name']: 'Zara', 'Age': 7}
```

```
print "dict['Name']: ", dict['Name']
```
When the above code is executed, it produces the following result −
```
Traceback (most recent call last):
   File "test.py", line 3, in <module>
      dict = {['Name']: 'Zara', 'Age': 7};
TypeError: list objects are unhashable
```

Built-in Dictionary Functions & Methods

Python includes the following dictionary functions −

Sr.No.	Function with Description
1	cmp(dict1, dict2) Compares elements of both dict.
2	len(dict) Gives the total length of the dictionary. This would be equal to the number of items in the dictionary.
3	str(dict) Produces a printable string representation of a dictionary
4	type(variable) Returns the type of the passed variable. If passed variable is dictionary, then it would return a dictionary type.

Python includes following dictionary methods −

Sr.No.	Methods with Description
1	dict.clear() Removes all elements of dictionary *dict*
2	dict.copy() Returns a shallow copy of dictionary *dict*
3	dict.fromkeys() Create a new dictionary with keys from seq and values *set* to *value*.
4	dict.get(key, default=None) For *key* key, returns value or default if key not in dictionary
5	dict.has_key(key) Returns *true* if key in dictionary *dict*, *false* otherwise
6	dict.items() Returns a list of *dict*'s (key, value) tuple pairs
7	dict.keys() Returns list of dictionary dict's keys
8	dict.setdefault(key, default=None) Similar to get(), but will set dict[key]=default if *key* is not already in dict
9	dict.update(dict2) Adds dictionary *dict2*'s key-values pairs to *dict*
10	dict.values() Returns list of dictionary *dict*'s values

Python - Basic 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.

- Arithmetic Operators
- Comparison (Relational) Operators
- Assignment Operators
- Logical Operators
- Bitwise Operators
- Membership Operators
- Identity Operators

Let us have a look on all operators one by one.

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 = 30
- Subtraction	Subtracts right hand operand from left hand operand.	a – b = -10
* Multiplication	Multiplies values on either side of the operator	a * b = 200
/ Division	Divides left hand operand by right hand operand	b / a = 2
% Modulus	Divides left hand operand by right hand operand and returns remainder	b % a = 0
** Exponent	Performs exponential (power) calculation on operators	a**b =10 to the power 20
//	Floor Division - The division of operands where the result is the quotient in which the digits after the decimal point are removed. But if one of the operands is negative, the result is floored, i.e., rounded away from zero (towards negative infinity) −	9//2 = 4 and 9.0//2.0 = 4.0, -11//3 = -4, -11.0//3 = -4.0

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.

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

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

Python Assignment Operators

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

Operator	Description	Example
=	Assigns values from right side operands to left side operand	c = a + b assigns value of a + b into c
+= Add AND	It adds right operand to the left operand and assign the result to left operand	c += a is equivalent to c = c + a
-= Subtract AND	It subtracts right operand from the left operand and assign the result to left operand	c -= a is equivalent to c = c - a
*= Multiply AND	It multiplies right operand with the left operand and assign the result to left operand	c *= a is equivalent to c = c * a
/= Divide AND	It divides left operand with the right operand and assign the result to left operand	c /= a is equivalent to c = c / ac /= a is equivalent to c = c / a
%= Modulus AND	It takes modulus using two operands and assign the result to left operand	c %= a is equivalent to c = c % a
**= Exponent AND	Performs exponential (power) calculation on operators and assign value to the left operand	c **= a is equivalent to c = c ** a
//= Floor Division	It performs floor division on operators and assign value to the left operand	c //= a is equivalent to c = c // a

Python Bitwise Operators

Bitwise operator works on bits and performs bit by bit operation. Assume if a = 60; and b = 13; Now in binary format they will be as follows −

a = 0011 1100

b = 0000 1101

---

a&b = 0000 1100

a|b = 0011 1101

a^b = 0011 0001

~a = 1100 0011

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 a signed binary number.
<< Binary Left Shift	The left operands value is moved left by the number of bits specified by the right operand.	a << 2 = 240 (means 1111 0000)
>> Binary Right Shift	The left operands value is moved right by the number of bits specified by the right operand.	a >> 2 = 15 (means 0000 1111)

Python Logical Operators

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

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 −

Operator	Description	Example
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 sequence y.

Python Identity Operators

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

Operator	Description	Example
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 id(y).

Python Operators Precedence

The following table lists all operators from highest precedence to lowest.

Sr.No.	Operator & Description
1	** Exponentiation (raise to the power)
2	~ + - Complement, unary plus and minus (method names for the last two are +@ and -@)
3	* / % // Multiply, divide, modulo and floor division
4	+ - Addition and subtraction
5	>><< Right and left bitwise shift
6	& Bitwise 'AND'
7	^ | Bitwise exclusive `OR' and regular `OR'
8	<= <>>= Comparison operators
9	<> == != Equality operators
10	= %= /= //= -= += *= **= Assignment operators
11	is is not Identity operators
12	in not in Membership operators
13	not or and Logical operators

Literals

Python Literals

Literals can be defined as a data that is given in a variable or constant.

Python support the following literals:

I. String literals:

String literals can be formed by enclosing a text in the quotes. We can use both single as well as double quotes for a String.

Eg:"Aman" , '12345'

Types of Strings:

There are two types of Strings supported in Python:

a).Single line String- Strings that are terminated within a single line are known as Single line Strings.

Eg:>>> text1='hello'

b).Multi line String- A piece of text that is spread along multiple lines is known as Multiple line String.

There are two ways to create Multiline Strings:

1). Adding black slash at the end of each line.

Eg:

```
>>> text1='hello\
user'
>>> text1
'hellouser'
>>>
```

2).Using triple quotation marks:-

Eg:

```
>>> str2='''welcome
to
SSSIT'''
>>> print str2
welcome
to
SSSIT
>>>
```

II.Numeric literals:

Numeric Literals are immutable. Numeric literals can belong to following four different numerical types.

Int(signed integers)	Long(long integers)	float(floating point)	Complex(complex)
Numbers(can be both positive and negative) with no fractional part.eg: 100	Integers of unlimited size followed by lowercase or uppercase L eg: 87032845L	Real numbers with both integer and fractional part eg: -26.2	In the form of a+bj where a forms the real part and b forms the imaginary part of complex number. eg: 3.14j

III. Boolean literals:

A Boolean literal can have any of the two values: True or False.

IV. Special literals.

Python contains one special literal i.e., None.
None is used to specify to that field that is not created. It is also used for end of lists in Python.

Eg:

```
>>> val1=10
>>> val2=None
>>> val1
10
>>> val2
>>> print val2
None
>>>
```

V.Literal Collections.

Collections such as tuples, lists and Dictionary are used in Python.

List:

- List contain items of different data types. Lists are mutable i.e., modifiable.
- The values stored in List are separated by commas(,) and enclosed within a square brackets([]). We can store different type of data in a List.
- Value stored in a List can be retrieved using the slice operator([] and [:]).
- The plus sign (+) is the list concatenation and asterisk(*) is the repetition operator.

Eg:

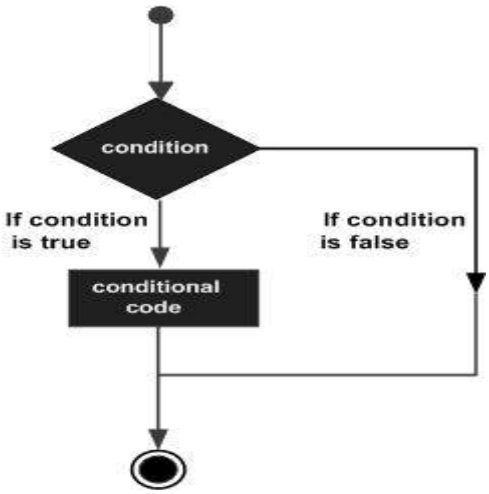
```
>>> list=['aman',678,20.4,'saurav']
>>> list1=[456,'rahul']
>>> list
['aman', 678, 20.4, 'saurav']
>>> list[1:3]
[678, 20.4]
>>> list+list1
['aman', 678, 20.4, 'saurav', 456, 'rahul']
>>> list1*2
[456, 'rahul', 456, 'rahul']
>>>
```

Python - Decision Making

Decision making is anticipation of conditions occurring while execution of the program and specifying actions taken according to the conditions.

Decision structures evaluate multiple expressions which produce TRUE or FALSE as outcome. You need to determine which action to take and which statements to execute if outcome is TRUE or FALSE otherwise.

Following is the general form of a typical decision making structure found in most of the programming languages –



Python programming language assumes any **non-zero** and **non-null** values as TRUE, and if it is either **zero** or **null**, then it is assumed as FALSE value.

Python programming language provides following types of decision making statements. Click the following links to check their detail.

Sr.No.	Statement & Description
1	**if statements** An **if statement** consists of a boolean expression followed by one or more statements.
2	**if...else statements** An **if statement** can be followed by an optional **else statement**, which executes when the boolean expression is FALSE.
3	**nested if statements** You can use one **if** or **else if** statement inside another **if** or **else if**statement(s).

Let us go through each decision making briefly –

Single Statement Suites

If the suite of an **if** clause consists only of a single line, it may go on the same line as the header statement.

Here is an example of a **one-line if** clause –

```
#!/usr/bin/python
var=100
if(var==100):print"Value of expression is 100"
print"Good bye!"
```

When the above code is executed, it produces the following result –

```
Value of expression is 100
Good bye!
```

if statements

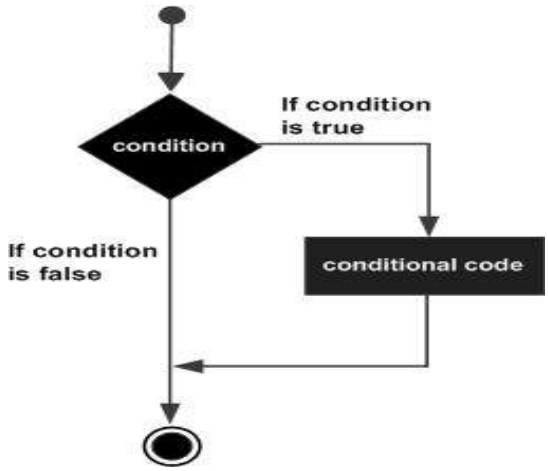
It is similar to that of other languages. The **if** statement contains a logical expression using which data is compared and a decision is made based on the result of the comparison.

Syntax :

```
if expression:
   statement(s)
```

If the boolean expression evaluates to TRUE, then the block of statement(s) inside the if statement is executed. If boolean expression evaluates to FALSE, then the first set of code after the end of the if statement(s) is executed.

Flow Diagram



Example

```
#!/usr/bin/python
var1 = 100
if var1:
   print "1 - Got a true expression value"
   print var1

var2 = 0
if var2:
   print "2 - Got a true expression value"
   print var2
print "Good bye!"
```

When the above code is executed, it produces the following result –

```
1 - Got a true expression value
100
Good bye!
```

if...else statements

An **else** statement can be combined with an **if** statement. An **else** statement contains the block of code that executes if the conditional expression in the if statement resolves to 0 or a FALSE value.

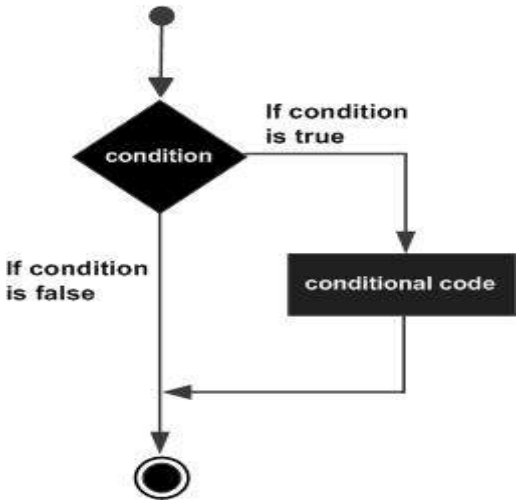
The *else* statement is an optional statement and there could be at most only one **else** statement following **if**.

Syntax : The syntax of the *if...else* statement is –

```
if expression:
```

```
   statement(s)
else:
   statement(s)
```

Flow Diagram



Example

```
#!/usr/bin/python

var1 =100
if var1:
print"1 - Got a true expression value"
print var1
else:
print"1 - Got a false expression value"
print var1

var2 =0
if var2:
print"2 - Got a true expression value"
print var2
else:
print"2 - Got a false expression value"
print var2

print"Good bye!"
```

When the above code is executed, it produces the following result –

```
1 - Got a true expression value
100
2 - Got a false expression value
0
Good bye!
```

The *elif* Statement

The **elif** statement allows you to check multiple expressions for TRUE and execute a block of code as soon as one of the conditions evaluates to TRUE.

Similar to the **else**, the **elif** statement is optional. However, unlike **else**, for which there can be at most one statement, there can be an arbitrary number of**elif** statements following an **if**.

syntax

```
if expression1:
   statement(s)
```

```
elif expression2:
   statement(s)
elif expression3:
   statement(s)
else:
   statement(s)
```

Core Python does not provide switch or case statements as in other languages, but we can use if..elif...statements to simulate switch case as follows –

Example

```
#!/usr/bin/python

var=100
ifvar==200:
print"1 - Got a true expression value"
printvar
elifvar==150:
print"2 - Got a true expression value"
printvar
elifvar==100:
print"3 - Got a true expression value"
printvar
else:
print"4 - Got a false expression value"
printvar

print"Good bye!"
```

When the above code is executed, it produces the following result –

```
3 - Got a true expression value
100
Good bye!
```

nested if statements

There may be a situation when you want to check for another condition after a condition resolves to true. In such a situation, you can use the nested **if**construct.

In a nested **if** construct, you can have an **if...elif...else** construct inside another**if...elif...else** construct.

Syntax: The syntax of the nested *if...elif...else* construct may be –

```
if expression1:
   statement(s)
   if expression2:
      statement(s)
   elif expression3:
      statement(s)
   else:
      statement(s)
   elif expression4:
      statement(s)
else:
   statement(s)
```

Example

```
#!/usr/bin/python
var=100
ifvar<200:
print"Expression value is less than 200"
ifvar==150:
print"Which is 150"
elifvar==100:
print"Which is 100"
elifvar==50:
print"Which is 50"
elifvar<50:
print"Expression value is less than 50"
else:
print"Could not find true expression"

print"Good bye!"
```

When the above code is executed, it produces following result –

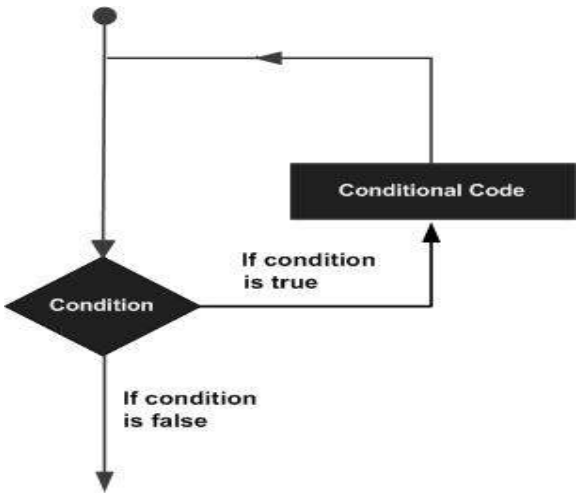
```
Expression value is less than 200
Which is 100
Good bye!
```

Python - Loops

In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on. There may be a situation when you need to execute a block of code several number of times.

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times. The following diagram illustrates a loop statement –



Python programming language provides following types of loops to handle looping requirements.

Sr.No.	Loop Type & Description
1	**while loop** Repeats a statement or group of statements while a given condition is TRUE. It tests the condition before executing the loop body.
2	**for loop** Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.

3	**nested loops** You can use one or more loop inside any another while, for or do..while loop.

Loop Control Statements

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

Python supports the following control statements. Click the following links to check their detail.

Sr.No.	Control Statement & Description
1	**break statement** Terminates the loop statement and transfers execution to the statement immediately following the loop.
2	**continue statement** Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.
3	**pass statement** The pass statement in Python is used when a statement is required syntactically but you do not want any command or code to execute.

Let us go through the loop control statements briefly

while loop

A **while** loop statement in Python programming language repeatedly executes a target statement as long as a given condition is true.

Syntax : The syntax of a **while loop** in Python programming language is –

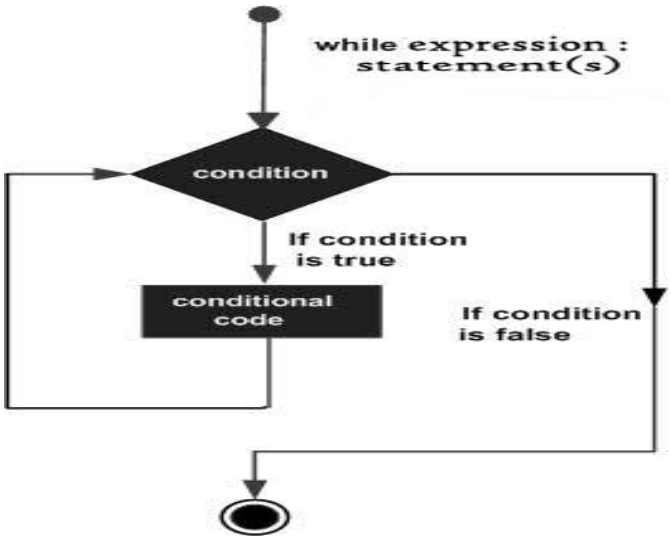
```
while expression:
   statement(s)
```

Here, **statement(s)** may be a single statement or a block of statements. The**condition** may be any expression, and true is any non-zero value. The loop iterates while the condition is true.

When the condition becomes false, program control passes to the line immediately following the loop.

In Python, all the statements indented by the same number of character spaces after a programming construct are considered to be part of a single block of code. Python uses indentation as its method of grouping statements.

Flow Diagram



Here, key point of the while loop is that the loop might not ever run. When the condition is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

Example

```
#!/usr/bin/python
count =0
while(count <9):
print'The count is:', count
   count = count +1

print"Good bye!"
```

When the above code is executed, it produces the following result –

```
The count is: 0
The count is: 1
The count is: 2
The count is: 3
The count is: 4
The count is: 5
The count is: 6
The count is: 7
The count is: 8
Good bye!
```

The block here, consisting of the print and increment statements, is executed repeatedly until count is no longer less than 9. With each iteration, the current value of the index count is displayed and then increased by 1.

The Infinite Loop

A loop becomes infinite loop if a condition never becomes FALSE. You must use caution when using while loops because of the possibility that this condition never resolves to a FALSE value. This results in a loop that never ends. Such a loop is called an infinite loop.

An infinite loop might be useful in client/server programming where the server needs to run continuously so that client programs can communicate with it as and when required.

```
#!/usr/bin/python
var=1
whilevar==1:# This constructs an infinite loop
   num = raw_input("Enter a number  :")
print"You entered: ", num
```

```
print"Good bye!"
```

When the above code is executed, it produces the following result –

```
Enter a number  :20
You entered:  20
Enter a number  :29
You entered:  29
Enter a number  :3
You entered:  3
Enter a number between :Traceback (most recent call last):
   File "test.py", line 5, in <module>
      num = raw_input("Enter a number :")
KeyboardInterrupt
```

Above example goes in an infinite loop and you need to use CTRL+C to exit the program.

Using else Statement with Loops

Python supports to have an **else** statement associated with a loop statement.

- If the **else** statement is used with a **for** loop, the **else** statement is executed when the loop has exhausted iterating the list.
- If the **else** statement is used with a **while** loop, the **else** statement is executed when the condition becomes false.

The following example illustrates the combination of an else statement with a while statement that prints a number as long as it is less than 5, otherwise else statement gets executed.

```
#!/usr/bin/python
count =0
while count <5:
print( count," is  less than 5")
   count = count +1
else:
print(count," is not less than 5")
```

When the above code is executed, it produces the following result –

```
0 is less than 5
1 is less than 5
2 is less than 5
3 is less than 5
4 is less than 5
5 is not less than 5
```

Single Statement Suites

Similar to the **if** statement syntax, if your **while** clause consists only of a single statement, it may be placed on the same line as the while header.

Here is the syntax and example of a **one-line while** clause –

```
#!/usr/bin/python

flag =1
while(flag):print'Given flag is really true!'
print"Good bye!"
```

It is better not try above example because it goes into infinite loop and you need to press CTRL+C keys to exit.

for Loop Statements

It has the ability to iterate over the items of any sequence, such as a list or a string.

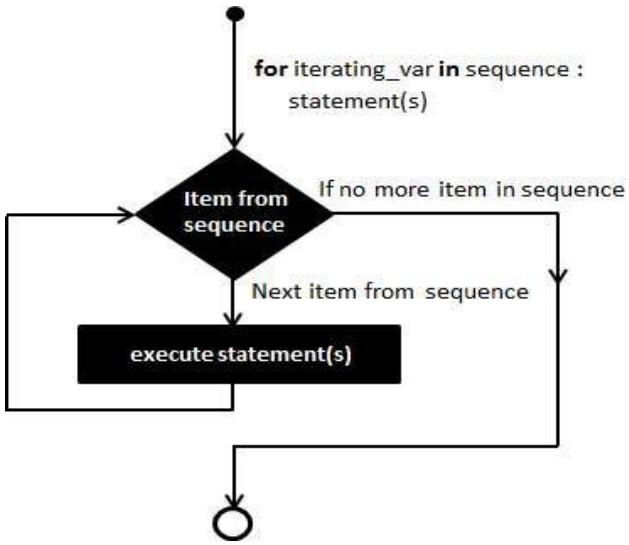
Syntax

```
for iterating_var in sequence:
```

```
   statements(s)
```

If a sequence contains an expression list, it is evaluated first. Then, the first item in the sequence is assigned to the iterating variable *iterating_var*. Next, the statements block is executed. Each item in the list is assigned to *iterating_var*, and the statement(s) block is executed until the entire sequence is exhausted.

Flow Diagram



Example

```
#!/usr/bin/python

for letter in'Python':# First Example
print'Current Letter :', letter

fruits =['banana','apple','mango']
for fruit infruits:# Second Example
print'Current fruit :', fruit

print"Good bye!"
```

When the above code is executed, it produces the following result –

```
Current Letter : P
Current Letter : y
Current Letter : t
Current Letter : h
Current Letter : o
Current Letter : n
Current fruit : banana
Current fruit : apple
Current fruit : mango
Good bye!
```

Iterating by Sequence Index

An alternative way of iterating through each item is by index offset into the sequence itself. Following is a simple example –

```
#!/usr/bin/python

fruits =['banana','apple','mango']
for index in range(len(fruits)):
print'Current fruit :', fruits[index]
```

```
print"Good bye!"
```

When the above code is executed, it produces the following result –

```
Current fruit : banana
Current fruit : apple
Current fruit : mango
Good bye!
```

Here, we took the assistance of the len() built-in function, which provides the total number of elements in the tuple as well as the range() built-in function to give us the actual sequence to iterate over.

Using else Statement with Loops

Python supports to have an else statement associated with a loop statement

- If the **else** statement is used with a **for** loop, the **else** statement is executed when the loop has exhausted iterating the list.
- If the **else** statement is used with a **while** loop, the **else** statement is executed when the condition becomes false.

The following example illustrates the combination of an else statement with a for statement that searches for prime numbers from 10 through 20.

```
lower =10
upper =20
# uncomment the following lines to take input from the user
#lower = int(input("Enter lower range: "))
#upper = int(input("Enter upper range: "))
print("Prime numbers between",lower,"and",upper,"are:")
for num in range(lower,upper +1):
for i in range(2,num):
if(num % i)==0:
break
else:
print'%d is a prime number \n'%(num),
```

When the above code is executed, it produces the following result –

```
('Prime numbers between', 10, 'and', 20, 'are:')
11 is a prime number
13 is a prime number
17 is a prime number
19 is a prime number
```

nested loops :

Python programming language allows to use one loop inside another loop. Following section shows few examples to illustrate the concept.

Syntax

```
for iterating_var in sequence:
   for iterating_var in sequence:
      statements(s)
   statements(s)
```

The syntax for a **nested while loop** statement in Python programming language is as follows –

```
while expression:
   while expression:
      statement(s)
   statement(s)
```

A final note on loop nesting is that you can put any type of loop inside of any other type of loop. For example a for loop can be inside a while loop or vice versa.

Example

The following program uses a nested for loop to find the prime numbers from 2 to 100 –

```
#!/usr/bin/python

i =2
while(i <100):
   j =2
while(j <=(i/j)):
ifnot(i%j):break
      j = j +1
if(j > i/j):print i," is prime"
   i = i +1

print"Good bye!"
```

When the above code is executed, it produces following result –

```
2 is prime
3 is prime
5 is prime
7 is prime
11 is prime
13 is prime
17 is prime
19 is prime
23 is prime
29 is prime
31 is prime
37 is prime
41 is prime
43 is prime
47 is prime
53 is prime
59 is prime
61 is prime
67 is prime
71 is prime
73 is prime
79 is prime
83 is prime
89 is prime
97 is prime
Good bye!
```

break statement

It terminates the current loop and resumes execution at the next statement, just like the traditional break statement in C.

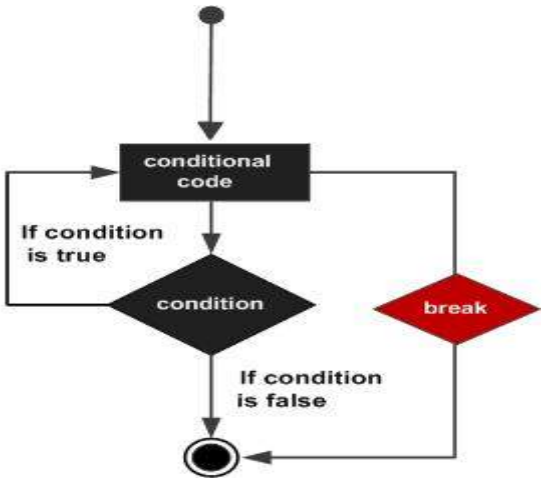
The most common use for break is when some external condition is triggered requiring a hasty exit from a loop. The **break** statement can be used in both*while* and *for* loops.

If you are using nested loops, the break statement stops the execution of the innermost loop and start executing the next line of code after the block.

Syntax : The syntax for a **break** statement in Python is as follows –

```
break
```

Flow Diagram



Example

```
#!/usr/bin/python

for letter in'Python':# First Example
if letter =='h':
break
print'Current Letter :', letter

var=10# Second Example
whilevar>0:
print'Current variable value :',var
var=var-1
ifvar==5:
break

print"Good bye!"
```

When the above code is executed, it produces the following result –

```
Current Letter : P
Current Letter : y
Current Letter : t
Current variable value : 10
Current variable value : 9
Current variable value : 8
Current variable value : 7
Current variable value : 6
Good bye!
```

continue statement

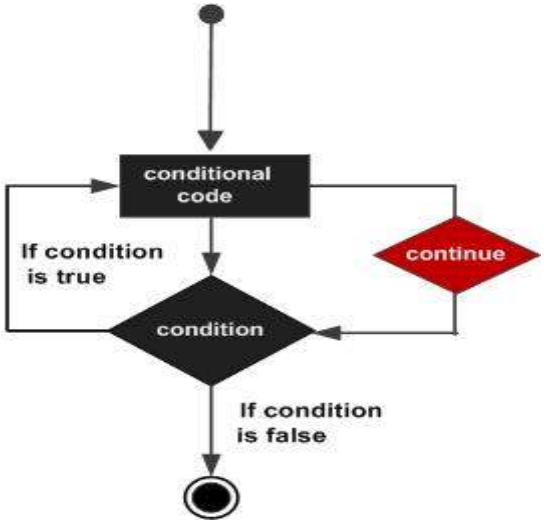
It returns the control to the beginning of the while loop.. The **continue**statement rejects all the remaining statements in the current iteration of the loop and moves the control back to the top of the loop.

The **continue** statement can be used in both *while* and *for* loops.

Syntax

```
continue
```

Flow Diagram



Example

```
#!/usr/bin/python

for letter in'Python':# First Example
if letter =='h':
continue
print'Current Letter :', letter

var=10# Second Example
whilevar>0:
var=var-1
ifvar==5:
continue
print'Current variable value :',var
print"Good bye!"
```

When the above code is executed, it produces the following result –

```
Current Letter : P
Current Letter : y
Current Letter : t
Current Letter : o
Current Letter : n
Current variable value : 9
Current variable value : 8
Current variable value : 7
Current variable value : 6
Current variable value : 4
Current variable value : 3
Current variable value : 2
Current variable value : 1
Current variable value : 0
Good bye!
```

pass Statement

It is used when a statement is required syntactically but you do not want any command or code to execute.
The **pass** statement is a *null* operation; nothing happens when it executes. The**pass** is also useful in places where your code will eventually go, but has not been written yet (e.g., in stubs for example) –

Syntax

```
pass
```

Example

```
#!/usr/bin/python

for letter in'Python':
if letter =='h':
pass
print'This is pass block'
print'Current Letter :', letter

print"Good bye!"
```

When the above code is executed, it produces following result –

```
Current Letter : P
Current Letter : y
Current Letter : t
This is pass block
Current Letter : h
Current Letter : o
Current Letter : n
Good bye!
```

File Handling in Python

What is a file?

File is a named location on disk to store related information. It is used to permanently store data in a non-volatile memory (e.g. hard disk).
Since, random access memory (RAM) is volatile which loses its data when computer is turned off, we use files for future use of the data.
When we want to read from or write to a file, we need to open it first. When we are done, it needs to be closed, so that resources that are tied with the file are freed.
Hence, in Python, a file operation takes place in the following order.

1. Open a file
2. Read or write (perform operation)
3. Close the file

Python too supports file handling and allows users to handle files i.e., to read and write files, along with many other file handling options, to operate on files. The concept of file handling has stretched over various other languages, but the implementation is either complicated or lengthy, but alike other concepts of Python, this concept here is also easy and short. Python treats file differently as text or binary and this is important. Each line of code includes a sequence of characters and they form text file. Each line of a file is terminated with a special character, called the EOL or End of Line characters like comma {,} or newline character. It ends the current line and tells the interpreter a new one has begun. Let us start with Reading and Writing files.

Python File Objects

Python has in-built functions to create and manipulate files. The io module is the default module for accessing files and you don't need to import it. The module consists of open(filename, access_mode) that returns a file object, which is called "handle". You can use this handle to read from or write to a file. Python treats the file as an object, which has its own attributes and methods.
As you already read before, there are two types of flat files, text and binary files:

1. As you might have expected from reading the previous section, text files have an End-Of-Line (EOL) character to indicate each line's termination. In Python, the new line character (\n) is default EOL terminator.
2. Since binary files store data after converting it into binary language (0s and 1s), there is no EOL character. This file type returns bytes. This is the file to be used when dealing with non-text files such as images or exe.

Opening and Closing Files

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

The *open* Function

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

Syntax

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

Here are parameter details −

- **file_name** − Thefile_name argument is a string value that contains the name of the file that you want to access.
- **access_mode** − Theaccess_mode determines the mode in which the file has to be opened, i.e., read, write, append, etc. A complete list of possible values is given below in the table. This is optional parameter and the default file access mode is read (r).
- **buffering** − If the buffering value is set to 0, no buffering takes place. If the buffering value is 1, line buffering is performed while accessing a file. If you specify the buffering value as an integer greater than 1, then buffering action is performed with the indicated buffer size. If negative, the buffer size is the system defaultbehavior.

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

Sr.No.	Modes & Description
1	**r** Opens a file for reading only. The file pointer is placed at the beginning of the file. This is the default mode.
2	**rb** Opens a file for reading only in binary format. The file pointer is placed at the beginning of the file. This is the default mode.
3	**r+** Opens a file for both reading and writing. The file pointer placed at the beginning of the file.
4	**rb+** Opens a file for both reading and writing in binary format. The file pointer placed at the beginning of the file.
5	**w** Opens a file for writing only. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
6	**wb** Opens a file for writing only in binary format. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
7	**w+** Opens a file for both writing and reading. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.
8	**wb+** Opens a file for both writing and reading in binary format. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.
9	**a** Opens a file for appending. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
10	**ab** Opens a file for appending in binary format. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
11	**a+** Opens a file for both appending and reading. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.
12	**ab+** Opens a file for both appending and reading in binary format. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.

The file Object Attributes

Once a file is opened and you have one *file* object, you can get various information related to that file.

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

Sr.No.	Attribute & Description
1	file.closed Returns true if file is closed, false otherwise.
2	file.mode Returns access mode with which file was opened.
3	file.name Returns name of the file.
4	file.softspace Returns false if space explicitly required with print, true otherwise.

Example

```
#!/usr/bin/python

# Open a file
fo = open("foo.txt","wb")
print"Name of the file: ", fo.name
print"Closed or not : ",fo.closed
print"Opening mode : ",fo.mode
print"Softspace flag : ",fo.softspace
```

This produces the following result −

```
Name of the file:  foo.txt
Closed or not :  False
Opening mode :  wb
Softspaceflag :  0
```

The close Method

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

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

Syntax

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

Example

```
#!/usr/bin/python

# Open a file
fo= open("foo.txt","wb")
print"Name of the file: ", fo.name

# Close opend file
fo.close()
```

This produces the following result −

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

Reading and Writing Files

The *file* object provides a set of access methods to make our lives easier. We would see how to use *read* and *write* methods to read and write files.

The write Method

The *write* method writes any string to an open file. It is important to note that Python strings can have binary data and not just text.

The write method does not add a newline character '\n'\n′ to the end of the string −

Syntax

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

Here, passed parameter is the content to be written into the opened file.

Example

```
#!/usr/bin/python

# Open a file
fo= open("foo.txt","wb")
fo.write("Python is a great language.\nYeah its great!!\n");

# Close opend file
fo.close()
```

The above method would create *foo.txt* file and would write given content in that file and finally it would close that file. If you would open this file, it would have following content.

```
Python is a great language.
Yeah its great!!
```

The append Method

The append function is used to append to the file instead of overwriting it.
To append to an existing file, simply open the file in append mode ("a"):

Syntax

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

Let’s see how the append mode works:

```
# Python code to illustrate append() mode
fo = open(“foo.txt”,”a”)
fo.write("This method will add this line")
fo.close()
```

The read Method

The *read* method reads a string from an open file. It is important to note that Python strings can have binary data. apart from text data.

Syntax

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

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

Example

Let's take a file *foo.txt*, which we created above.

```
#!/usr/bin/python

# Open a file
fo= open("foo.txt","r+")
str=fo.read(10);
print"Read String is : ",str
# Close opend file
fo.close()
```

This produces the following result −

```
Read String is :  Python is
```

File Handling Examples

Let's show some examples

To open a text file, use:

```
fh = open("hello.txt", "r")
```

To read a text file, use:

```
fh = open("hello.txt","r")
printfh.read()
```

To read one line at a time, use:

```
fh = open("hello".txt", "r")
printfh.readline()
```

To read a list of lines use:

```
fh = open("hello.txt.", "r")
printfh.readlines()
```

To write to a file, use:

```
fh = open("hello.txt","w")
write("Hello World")
fh.close()
```

To write to a file, use:

```
fh = open("hello.txt", "w")
lines_of_text = ["a line of text", "another line of text", "a third line"]
fh.writelines(lines_of_text)
fh.close()
```

To append to file, use:

```
fh = open("Hello.txt", "a")
```

```
write("Hello World again")
fh.close
```

To close a file, use

```
fh = open("hello.txt", "r")
printfh.read()
fh.close()
```

File Positions

The *tell* method tells you the current position within the file; in other words, the next read or write will occur at that many bytes from the beginning of the file.

The *seek*offset[,from]offset[,from] method changes the current file position. The *offset* argument indicates the number of bytes to be moved. The *from* argument specifies the reference position from where the bytes are to be moved.

If *from* is set to 0, it means use the beginning of the file as the reference position and 1 means use the current position as the reference position and if it is set to 2 then the end of the file would be taken as the reference position.

Example

Let us take a file *foo.txt*, which we created above.

```
#!/usr/bin/python

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

# Check current position
position=fo.tell();
print"Current file position : ", position

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

This produces the following result −

```
Read String is :  Python is
Current file position :  10
Again read String is :  Python is
```

Renaming and Deleting Files

Python **os** module provides methods that help you perform file-processing operations, such as renaming and deleting files.

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

The rename Method

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

Syntax

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

Example

Following is the example to rename an existing file *test1.txt* −

```
#!/usr/bin/python
importos

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

The remove Method

You can use the *remove* method to delete files by supplying the name of the file to be deleted as the argument.

Syntax

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

Example

Following is the example to delete an existing file *test2.txt* −

```
#!/usr/bin/python
importos

# Delete file test2.txt
os.remove("text2.txt")
```

Directories in Python

All files are contained within various directories, and Python has no problem handling these too. The **os** module has several methods that help you create, remove, and change directories.

The mkdir Method

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

Syntax

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

Example

Following is the example to create a directory *test* in the current directory −

```
#!/usr/bin/python
importos

# Create a directory "test"
os.mkdir("test")
```

The chdir Method

You can use the *chdir* method to change the current directory. The chdir method takes an argument, which is the name of the directory that you want to make the current directory.

Syntax

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

Example

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

```
#!/usr/bin/python
importos

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

The getcwd Method

The *getcwd* method displays the current working directory.

Syntax

```
os.getcwd()
```

Example

Following is the example to give current directory −

```
#!/usr/bin/python
importos

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

The rmdir Method

The *rmdir* method deletes the directory, which is passed as an argument in the method.

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

Syntax

```
os.rmdir('dirname')
```

Example

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

```
#!/usr/bin/python
importos

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

File & Directory Related Methods

There are three important sources, which provide a wide range of utility methods to handle and manipulate files & directories on Windows and Unix operating systems. They are as follows −

- File Object Methods: The *file* object provides functions to manipulate files.
- OS Object Methods: This provides methods to process files as well as directories.

PYTHON - MODULES

A module allows you to logically organize your Python code. Grouping related code into a module makes the code easier to understand and use. A module is a Python object with arbitrarily named attributes that you can bind and reference.

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

Example

The Python code for a module named *aname* normally resides in a file named *aname.py*. Here's an example of a simple module, support.py

```
def print_func( par ):
   print "Hello : ", par
   return
```

The *import* Statement

You can use any Python source file as a module by executing an import statement in some other Python source file. The *import* has the following syntax −

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

When the interpreter encounters an import statement, it imports the module if the module is present in the search path. A search path is a list of directories that the interpreter searches before importing a module. For example, to import the module support.py, you need to put the following command at the top of the script −

```
#!/usr/bin/python

# Import module support
import support

# Now you can call defined function that module as follows
support.print_func("Zara")
```

When the above code is executed, it produces the following result −

```
Hello : Zara
```

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

The *from...import* Statement

Python's *from* statement lets you import specific attributes from a module into the current namespace. The*from...import* has the following syntax −

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

For example, to import the function fibonacci from the module fib, use the following statement −

```
from fib import fibonacci
```

This statement does not import the entire module fib into the current namespace; it just introduces the item fibonacci from the module fib into the global symbol table of the importing module.

The *from...import* * Statement

It is also possible to import all names from a module into the current namespace by using the following import statement −

```
from modname import *
```

This provides an easy way to import all the items from a module into the current namespace; however, this statement should be used sparingly.

Locating Modules

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

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

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

The *PYTHONPATH* Variable

The PYTHONPATH is an environment variable, consisting of a list of directories. The syntax of PYTHONPATH is the same as that of the shell variable PATH.

Here is a typical PYTHONPATH from a Windows system −

```
set PYTHONPATH = c:\python20\lib;
```

And here is a typical PYTHONPATH from a UNIX system −

```
set PYTHONPATH = /usr/local/lib/python
```

Namespaces and Scoping

Variables are names identifiersidentifiers that map to objects. A *namespace* is a dictionary of variable names keyskeys and their corresponding objects valuesvalues.

A Python statement can access variables in a *local namespace* and in the *global namespace*. If a local and a global variable have the same name, the local variable shadows the global variable.

Each function has its own local namespace. Class methods follow the same scoping rule as ordinary functions.

Python makes educated guesses on whether variables are local or global. It assumes that any variable assigned a value in a function is local.

Therefore, in order to assign a value to a global variable within a function, you must first use the global statement.

The statement *global VarName* tells Python that VarName is a global variable. Python stops searching the local namespace for the variable.

For example, we define a variable *Money* in the global namespace. Within the function *Money*, we assign*Money* a value, therefore Python assumes *Money* as a local variable. However, we accessed the value of the local variable *Money* before setting it, so an UnboundLocalError is the result. Uncommenting the global statement fixes the problem.

```
#!/usr/bin/python


Money = 2000
def AddMoney():
   # Uncomment the following line to fix the code:
   # global Money
   Money = Money + 1


print Money
AddMoney()
print Money
```

The dir Function

The dir built-in function returns a sorted list of strings containing the names defined by a module.

The list contains the names of all the modules, variables and functions that are defined in a module. Following is a simple example −

Live Demo

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

```
# Import built-in module math
import math

content = dir(math)
print content
```

When the above code is executed, it produces the following result −

```
['__doc__', '__file__', '__name__', 'acos', 'asin', 'atan',
'atan2', 'ceil', 'cos', 'cosh', 'degrees', 'e', 'exp',
'fabs', 'floor', 'fmod', 'frexp', 'hypot', 'ldexp', 'log',
'log10', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh',
'sqrt', 'tan', 'tanh']
```

Here, the special string variable *__name__* is the module's name, and *__file__* is the filename from which the module was loaded.

The *globals* and *locals* Functions

The *globals* and *locals* functions can be used to return the names in the global and local namespaces depending on the location from where they are called.

If locals is called from within a function, it will return all the names that can be accessed locally from that function.

If globals is called from within a function, it will return all the names that can be accessed globally from that function.

The return type of both these functions is dictionary. Therefore, names can be extracted using the keysfunction.

The *reload* Function

When the module is imported into a script, the code in the top-level portion of a module is executed only once.

Therefore, if you want to reexecute the top-level code in a module, you can use the *reload* function. The reload function imports a previously imported module again. The syntax of the reload function is this −

```
reload(module_name)
```

Here, *module_name* is the name of the module you want to reload and not the string containing the module name. For example, to reload *hello* module, do the following −

```
reload(hello)
```

Packages in Python

A package is a hierarchical file directory structure that defines a single Python application environment that consists of modules and subpackages and sub-subpackages, and so on.

Consider a file *Pots.py* available in *Phone* directory. This file has following line of source code −

```
#!/usr/bin/python


def Pots():
   print "I'm Pots Phone"
```

Similar way, we have another two files having different functions with the same name as above −

- *Phone/Isdn.py* file having function Isdn
- *Phone/G3.py* file having function G3

Now, create one more file __init__.py in *Phone* directory −

- Phone/__init__.py

To make all of your functions available when you've imported Phone, you need to put explicit import statements in __init__.py as follows −

```
from Pots import Pots
from Isdn import Isdn
from G3 import G3
```

After you add these lines to __init__.py, you have all of these classes available when you import the Phone package.

```
#!/usr/bin/python


# Now import your Phone Package.
import Phone


Phone.Pots()
Phone.Isdn()
Phone.G3()
```

When the above code is executed, it produces the following result −

```
I'm Pots Phone
I'm 3G Phone
I'm ISDN Phone
```

In the above example, we have taken example of a single functions in each file, but you can keep multiple functions in your files. You can also define different Python classes in those files and then you can create your packages out of those classes.

Python - Functions

A function is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing.

Python gives you many built-in functions like print(), etc. but you can also create your own functions. These functions are called user-defined functions.

Defining a Function:

You can define functions to provide the required functionality. Here are simple rules to define a function in Python.

- Function blocks begin with the keyword **def** followed by the function name and parentheses (( )).
- Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses.
- The first statement of a function can be an optional statement - the documentation string of the function or *docstring*.
- The code block within every function starts with a colon (:) and is indented.
- The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

Syntax

```
def functionname( parameters ) :
   "function_docstring"
    function_suite
    return [expression]
```

By default, parameters have a positional behavior and you need to inform them in the same order that they were defined.

Example : The following function takes a string as input parameter and prints it on standard screen.

```
def printme( str ) :
   "This prints a passed string into this function"
   print (str)
   return
```

Calling a Function:

Defining a function only gives it a name, specifies the parameters that are to be included in the function and structures the blocks of code.

Once the basic structure of a function is finalized, you can execute it by calling it from another function or directly from the Python prompt. Following is the example to call printme() function −

```
#!/usr/bin/python

# Function definition is here
def printme( str ):
  #"This prints a passed string into this function"
  print str
  return;

# Now you can call printme function
printme("My first call to user defined function!")
printme("Again,The second call to the same function")
```

When the above code is executed, it produces the following result −

```
I'm first call to user defined function!
Again second call to the same function
```

Pass by reference vs value

All parameters (arguments) in the Python language are passed by reference. It means if you change what a parameter refers to within a function, the change also reflects back in the calling function. For example –

```
#!/usr/bin/python

# Function definition is here
def changeme( mylist ) :
   "This changes a passed list into this function"
   mylist.append([1,2,3,4]);
   print ("Values inside the function: ", mylist)
   return

# Now you can call changeme function
mylist = [10,20,30];
changeme( mylist );
print "Values outside the function: ", mylist
```

Here, we are maintaining reference of the passed object and appending values in the same object. So, this would produce the following result −

```
Values inside the function:  [10, 20, 30, [1, 2, 3, 4]]
Values outside the function:  [10, 20, 30, [1, 2, 3, 4]]
```

There is one more example where argument is being passed by reference and the reference is being overwritten inside the called function.

```
#!/usr/bin/python

# Function definition is here
def changeme( mylist ):
   "This changes a passed list into this function"
   mylist = [1,2,3,4]; # This would assig new reference in mylist
   print ("Values inside the function: ", mylist)
   return

# Now you can call changeme function
mylist = [10,20,30];
changeme( mylist );
print ("Values outside the function: ", mylist)
```

The parameter *mylist* is local to the function changeme. Changing mylist within the function does not affect *mylist*. The function accomplishes nothing and finally this would produce the following result −

```
Values inside the function:  [1, 2, 3, 4]
Values outside the function:  [10, 20, 30]
```

Function Arguments

You can call a function by using the following types of formal arguments −

- Required arguments
- Keyword arguments
- Default arguments
- Variable-length arguments

1. Required arguments

Required arguments are the arguments passed to a function in correct positional order. Here, the number of arguments in the function call should match exactly with the function definition.

To call the function *printme()*, you definitely need to pass one argument, otherwise it gives a syntax error as follows –

```
#!/usr/bin/python

# Function definition is here
def printme( str ):
```

```
   "This prints a passed string into this function"
   print str
   return;
# Now you can call printme function
printme()
```

When the above code is executed, it produces the following result −

```
Traceback (most recent call last):
   File "test.py", line 11, in <module>
      printme();
TypeError: printme() takes exactly 1 argument (0 given)
```

2. Keyword arguments

Keyword arguments are related to the function calls. When you use keyword arguments in a function call, the caller identifies the arguments by the parameter name.

This allows you to skip arguments or place them out of order because the Python interpreter is able to use the keywords provided to match the values with parameters. You can also make keyword calls to the *printme()* function in the following ways −

```
#!/usr/bin/python

# Function definition is here
def printme( str ):
   "This prints a passed string into this function"
   print str
   return;

# Now you can call printme function
printme( str = "My string")
```

When the above code is executed, it produces the following result −

```
My string
```

The following example gives more clear picture. Note that the order of parameters does not matter.

```
#!/usr/bin/python

# Function definition is here
def printinfo( name, age ):
   "This prints a passed info into this function"
   print "Name: ", name
   print "Age ", age
   return;

# Now you can call printinfo function
printinfo( age=50, name="miki" )
```

When the above code is executed, it produces the following result −

```
Name:  miki
Age  50
```

3. Default arguments

A default argument is an argument that assumes a default value if a value is not provided in the function call for that argument. The following example gives an idea on default arguments, it prints default age if it is not passed −

```
#!/usr/bin/python

# Function definition is here
def printinfo( name, age = 35 ):
   "This prints a passed info into this function"
   print "Name: ", name
   print "Age ", age
   return;
```

```
# Now you can call printinfo function
printinfo( age=50, name="miki" )
printinfo( name="miki" )
```

When the above code is executed, it produces the following result −

```
Name:  miki
Age  50
Name:  miki
Age  35
```

4. Variable-length arguments

You may need to process a function for more arguments than you specified while defining the function. These arguments are called *variable-length* arguments and are not named in the function definition, unlike required and default arguments.

Syntax for a function with non-keyword variable arguments is this −

```
def functionname([formal_args,] *var_args_tuple ):
   "function_docstring"
   function_suite
   return [expression]
```

An asterisk (*) is placed before the variable name that holds the values of all non keyword variable arguments. This tuple remains empty if no additional arguments are specified during the function call. Following is a simple example −

```
#!/usr/bin/python

# Function definition is here
def printinfo( arg1, *vartuple ):
   "This prints a variable passed arguments"
   print "Output is: "
   print arg1
   for var in vartuple:
      print var
   return;

# Now you can call printinfo function
printinfo( 10 )
printinfo( 70, 60, 50 )
```

When the above code is executed, it produces the following result −

```
Output is:
10
Output is:
70
60
50
```

The *Anonymous* Functions

These functions are called anonymous because they are not declared in the standard manner by using the *def* keyword. You can use the *lambda* keyword to create small anonymous functions.

- Lambda forms can take any number of arguments but return just one value in the form of an expression. They cannot contain commands or multiple expressions.
- An anonymous function cannot be a direct call to print because lambda requires an expression
- Lambda functions have their own local namespace and cannot access variables other than those in their parameter list and those in the global namespace.
- Although it appears that lambda's are a one-line version of a function, they are not equivalent to inline statements in C or C++, whose purpose is by passing function stack allocation during invocation for performance reasons.

Syntax: The syntax of *lambda* functions contains only a single statement, which is as follows −

```
lambda [arg1 [,arg2,.....argn]]:expression
```

Following is the example to show how *lambda* form of function works −

```
#!/usr/bin/python

# Function definition is here
sum = lambda arg1, arg2: arg1 + arg2;

# Now you can call sum as a function
print "Value of total : ", sum( 10, 20 )
print "Value of total : ", sum( 20, 20 )
```

When the above code is executed, it produces the following result −

```
Value of total :  30
Value of total :  40
```

The *return* Statement

The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

All the above examples are not returning any value. You can return a value from a function as follows −

```
#!/usr/bin/python

# Function definition is here
def sum( arg1, arg2 ):
   # Add both the parameters and return them."
   total = arg1 + arg2
   print "Inside the function : ", total
   return total;

# Now you can call sum function
total = sum( 10, 20 );
print "Outside the function : ", total
```

When the above code is executed, it produces the following result −

```
Inside the function :  30
Outside the function :  30
```

Scope of Variables

All variables in a program may not be accessible at all locations in that program. This depends on where you have declared a variable.

The scope of a variable determines the portion of the program where you can access a particular identifier. There are two basic scopes of variables in Python −

- Global variables
- Local variables

Global vs. Local variables

Variables that are defined inside a function body have a local scope, and those defined outside have a global scope.

This means that local variables can be accessed only inside the function in which they are declared, whereas global variables can be accessed throughout the program body by all functions. When you call a function, the variables declared inside it are brought into scope. Following is a simple example −

```
#!/usr/bin/python

total = 0; # This is global variable.
# Function definition is here
def sum( arg1, arg2 ):
   # Add both the parameters and return them."
   total = arg1 + arg2; # Here total is local variable.
   print "Inside the function local total : ", total
   return total;

# Now you can call sum function
sum( 10, 20 );
```

```
print "Outside the function global total : ", total
```

When the above code is executed, it produces the following result −

```
Inside the function local total :  30
Outside the function global total :  0
```

Built – In Functions

The Python interpreter has a number of functions and types built into it that are always available. They are listed here in alphabetical order. Python has a set of built-in functions.

The Python interpreter has a number of functions that are always available for use. These functions are called built-in functions. For example, `print()` function prints the given object to the standard output device (screen) or to the text stream file.In Python 3.6 (latest version), there are 68 built-in functions. They are listed below alphabetically along with brief description.

Built-in Functions				
abs()	delattr()	hash()	memoryview()	set()
all()	dict()	help()	min()	setattr()
any()	dir()	hex()	next()	slice()
ascii()	divmod()	id()	object()	sorted()
bin()	enumerate()	input()	oct()	staticmethod()
bool()	eval()	int()	open()	str()
breakpoint()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	

OR

Function	Description
abs()	Returns the absolute value of a number
all()	Returns True if all items in an iterable object are true
any()	Returns True if any item in an iterable object is true
ascii()	Returns a readable version of an object. Replaces none-ascii characters with escape character
bin()	Returns the binary version of a number
bool()	Returns the boolean value of the specified object
bytearray()	Returns an array of bytes
bytes()	Returns a bytes object
callable()	Returns True if the specified object is callable, otherwise False
chr()	Returns a character from the specified Unicode code.
classmethod()	Converts a method into a class method
compile()	Returns the specified source as an object, ready to be executed
complex()	Returns a complex number
delattr()	Deletes the specified attribute (property or method) from the specified object
dict()	Returns a dictionary (Array)
dir()	Returns a list of the specified object's properties and methods
divmod()	Returns the quotient and the remainder when argument1 is divided

	by argument2
enumerate()	Takes a collection (e.g. a tuple) and returns it as an enumerate object
eval()	Evaluates and executes an expression
exec()	Executes the specified code (or object)
filter()	Use a filter function to exclude items in an iterable object
float()	Returns a floating point number
format()	Formats a specified value
frozenset()	Returns a frozenset object
getattr()	Returns the value of the specified attribute (property or method)
globals()	Returns the current global symbol table as a dictionary
hasattr()	Returns True if the specified object has the specified attribute (property/method)
hash()	Returns the hash value of a specified object
help()	Executes the built-in help system
hex()	Converts a number into a hexadecimal value
id()	Returns the id of an object
input()	Allowing user input
int()	Returns an integer number
isinstance()	Returns True if a specified object is an instance of a specified object
issubclass()	Returns True if a specified class is a subclass of a specified object
iter()	Returns an iterator object
len()	Returns the length of an object
list()	Returns a list
locals()	Returns an updated dictionary of the current local symbol table
map()	Returns the specified iterator with the specified function applied to each item
max()	Returns the largest item in an iterable
memoryview()	Returns a memory view object
min()	Returns the smallest item in an iterable
next()	Returns the next item in an iterable
object()	Returns a new object
oct()	Converts a number into an octal
open()	Opens a file and returns a file object
ord()	Convert an integer representing the Unicode of the specified character
pow()	Returns the value of x to the power of y
print()	Prints to the standard output device
property()	Gets, sets, deletes a property
range()	Returns a sequence of numbers, starting from 0 and increments by 1 (by default)
repr()	Returns a readable version of an object
reversed()	Returns a reversed iterator
round()	Rounds a numbers
set()	Returns a new set object
setattr()	Sets an attribute (property/method) of an object
slice()	Returns a slice object
sorted()	Returns a sorted list
@staticmethod()	Converts a method into a static method
str()	Returns a string object
sum()	Sums the items of an iterator
tuple()	Returns a tuple
type()	Returns the type of an object
vars()	Returns the __dict__ property of an object
zip()	Returns an iterator, from two or more iterators

1. Python abs()

The python abs() is one of the most popular Python built-in functions, which returns the absolute value of a number. A negative value's absolute is that value is positive.

```
>>> abs(-7)
```

```
7
```

```
>>> abs(7)
```

```
7
```

```
>>> abs(0)
```

```
0
```

2. Python all()

Python all() function takes a container as an argument. This Built in Functions returns True if all values in a python iterable have a Boolean value of True. An empty value has a Boolean value of False.

```
>>> all({'*',',',''})
```

```
False
```

```
>>> all([' ',' ',' '])
```

```
True
```

3. Python any()

Like Python all(), it takes one argument and returns True if, even one value in the iterable has a Boolean value of True.

```
>>> any((1,0,0))
```

```
True
```

```
>>> any((0,0,0))
```

```
False
```

4. Python ascii()

Python ascii(), is important Python built-in functions, returns a printable representation of a **python object** (like a string or a **Python list**). Let's take a Romanian character.

```
>>> ascii('ș')

'\\u0219'
```

Since this was a non-ASCII character in python, the interpreter added a backslash (\) and escaped it using another backslash.

```
>>> ascii('ușor')

'u\\u0219or'
```

Let's apply it to a list.

```
>>> ascii(['s','ș'])

"['s', '\\u0219']"
```

5. Python bin()

Python bin() converts an integer to a binary string. We have seen this and other functions in our article on**Python Numbers**.

```
>>> bin(7)
```

'0b111'

We can't apply it on floats, though.

```
>>> bin(7.0)
Traceback (most recent call last):
File "<pyshell#20>", line 1, in <module>
bin(7.0)
```

TypeError: 'float' object cannot be interpreted as an integer

6. Python bool()

Python bool() converts a value to Boolean.

```
>>> bool(0.5)
```

```
True
```

```
>>> bool('')
```

```
False
```

```
>>> bool(True)
```

```
True
```

7. Python bytearray()

Python bytearray() returns a python array of a given byte size.

```
>>> a=bytearray(4)
>>> a
bytearray(b'\x00\x00\x00\x00')
>>> a.append(1)
>>> a
bytearray(b'\x00\x00\x00\x00\x01')
>>> a[0]=1
>>> a
bytearray(b'\x01\x00\x00\x00\x01')
>>> a[0]
```

```
1
```

Let's do this on a list.

```
>>> bytearray([1,2,3,4])
bytearray(b'\x01\x02\x03\x04')
```

8. Python bytes()

Python bytes() returns an immutable bytes object.

```
>>> bytes(5)
b'\x00\x00\x00\x00\x00'
>>> bytes([1,2,3,4,5])
b'\x01\x02\x03\x04\x05'
>>> bytes('hello','utf-8')
```

```
b'hello'
```

Here, utf-8 is the encoding.

Both bytes() and bytearray() deal with raw data, but bytearray() is mutable, while bytes() is immutable.

```
>>> a=bytes([1,2,3,4,5])
>>> a
b'\x01\x02\x03\x04\x05'
>>> a[4]=
```

```
3
```

```
Traceback (most recent call last):
File "<pyshell#46>", line 1, in <module>
a[4]=3
```

```
TypeError: 'bytes' object does not support item assignment
```

Let's try this on bytearray().

```
>>> a=bytearray([1,2,3,4,5])
>>> a
bytearray(b'\x01\x02\x03\x04\x05')
>>> a[4]=3
>>> a
bytearray(b'\x01\x02\x03\x04\x03')
```

9. Python callable()

Python callable() tells us if an object can be called.

```
>>> callable([1,2,3])
```

```
False
```

```
>>> callable(callable)
```

```
True
```

```
>>> callable(False)
```
False
```
>>> callable(list)
```
True
A function is callable, a list is not. Even the callable() python Built In function is callable.

10. Python chr()

Python chr() Built In function returns the character in python for an ASCII value.
```
>>> chr(65)
```
'A'
```
>>> chr(97)
```
'a'
```
>>> chr(9)
```
'\t'
```
>>> chr(48)
```
'0'

11. Python classmethod()

Python classmethod() returns a class method for a given method.
```
>>> class fruit:
def sayhi(self):
print("Hi, I'm a fruit")
>>> fruit.sayhi=classmethod(fruit.sayhi)
>>> fruit.sayhi()
```
Hi, I'm a fruit
When we pass the method sayhi() as an argument to classmethod(), it converts it into a python class method one that belongs to the class. Then, we call it like we would call any static **method in python** without an object.

12. Python compile()

Python compile() returns a Python code object. We use Python in built function to convert a string code into object code.
```
>>> exec(compile('a=5\nb=7\nprint(a+b)','','exec'))
```
12
Here, 'exec' is the mode. The parameter before that is the filename for the file form which the code is read.
Finally, we execute it using exec().

13. Python complex()

Python complex() function creates a complex number. We have seen this is our article on **Python Numbers**.
```
>>> complex(3)
```
(3+0j)
```
>>> complex(3.5)
```
(3.5+0j)
```
>>> complex(3+5j)
```
(3+5j)

14. Python delattr()

Python delattr() takes two arguments- a class, and an attribute in it. It deletes the attribute.
```
>>> class fruit:
size=7
>>> orange=fruit()
>>> orange.size
```
7
```
>>> delattr(fruit,'size')
>>> orange.size
Traceback (most recent call last):
File "<pyshell#95>", line 1, in <module>
orange.size
```
AttributeError: 'fruit' object has no attribute 'size'

15. Python dict()

Python dict(), as we have seen it, creates a **python dictionary**.

```
>>> dict()
{}
>>> dict([(1,2),(3,4)])
{1: 2, 3: 4}
```

This was about dict() Python Built In function

16. Python dir()

Python dir() returns an object's attributes.

```
>>> class fruit:
size=7
shape='round'
>>> orange=fruit()
>>> dir(orange)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'shape', 'size']
```

17. Python divmod()

Python divmod() in Python built-in functions, takes two parameters, and returns a tuple of their quotient and remainder. In other words, it returns the floor division and the modulus of the two numbers.

```
>>> divmod(3,7)
(0, 3)
>>> divmod(7,3)
(2, 1)
```

If you encounter any doubt in Python Built-in Function, Please Comment.

18. Python enumerate()

This Python Built In function returns an enumerate object. In other words, it adds a counter to the iterable.

```
>>> for i in enumerate(['a','b','c']):
print(i)
(0, 'a')
(1, 'b')
(2, 'c')
```

19. Python eval()

This Function takes a string as an argument, which is parsed as an expression.

```
>>> x=7
>>> eval('x+7')
14
>>> eval('x+(x%2)')
8
```

20. Python exec()

Python exec() runs Python code dynamically.

```
>>> exec('a=2;b=3;print(a+b)')
5
>>> exec(input("Enter your program"))
Enter your programprint(2+3)
5
```

21. Python filter()

Like we've seen in **python Lambda Expressios**, filter() filters out the items for which the condition is True.

```
>>> list(filter(lambda x:x%2==0,[1,2,0,False]))
[2, 0, False]
```

22. Python float()

This Python Built IN function converts an int or a compatible value into a float.

```
>>> float(2)
```

```
2.0
>>> float('3')
3.0
>>> float('3s')
Traceback (most recent call last):
File "<pyshell#136>", line 1, in <module>
float('3s')
ValueError: could not convert string to float: '3s'
>>> float(False)
0.0
>>> float(4.7)
4.7
```

23. Python format()

We have seen this Python built-in function, one in our lesson on **Python Strings**.

```
>>> a,b=2,3
>>> print("a={0} and b={1}".format(a,b))
a=2 and b=3
>>> print("a={a} and b={b}".format(a=3,b=4))
a=3 and b=4
```

24. Python frozenset()

Python frozenset() returns an immutable frozenset object.

```
>>> frozenset((3,2,4))
frozenset({2, 3, 4})
```

Read **Python Sets and Booleans** for more on frozenset.

25. Python getattr()

Python getattr() returns the value of an object's attribute.

```
>>> getattr(orange,'size')
7
```

26. Python globals()

This Python built-in functions, returns a dictionary of the current global symbol table.

```
>>> globals()
{'__name__': '__main__', '__doc__': None, '__package__': None, '__loader__': <class '_frozen_importlib.BuiltinImporter'>, '__spec__': None, '__annotations__': {}, '__builtins__': <module 'builtins' (built-in)>, 'fruit': <class '__main__.fruit'>, 'orange': <__main__.fruit object at 0x05F937D0>, 'a': 2, 'numbers': [1, 2, 3], 'i': (2, 3), 'x': 7, 'b': 3}
```

27. Python hasattr()

Like delattr() and getattr(), hasattr() Python built-in functions, returns True if the object has that attribute.

```
>>> hasattr(orange,'size')
True
>>> hasattr(orange,'shape')
True
>>> hasattr(orange,'color')
False
```

28. Python hash()

Python hash() function returns the hash value of an object. And in Python, everything is an object.

```
>>> hash(orange)
6263677
>>> hash(orange)
6263677
>>> hash(True)
1
>>> hash(0)
```

```
0
```

```
>>> hash(3.7)
```

```
644245917
```

```
>>> hash(hash)
```

```
25553952
```

This was all about hash() Python In Built function

29. Python help()

To get details about any module, keyword, symbol, or topic, we use the help() function.

```
>>> help()
```

```
Welcome to Python 3.6's help utility!
If this is your first time using Python, you should definitely check out the tutorial on the Internet at http://docs.python.org/3.6/tutorial/.Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules.  To quit this help utility and return to the interpreter, just type "quit".
To get a list of available modules, keywords, symbols, or topics, type "modules", "keywords", "symbols", or "topics".  Each module also comes with a one-line summary of what it does; to list the modules whose name
or summary contain a given string such as "spam", type "modules spam".
help> map
Help on class map in module builtins:

class map(object)
 |  map(func, *iterables) –> map object
 |
 |  Make an iterator that computes the function using arguments from
 |  each of the iterables.  Stops when the shortest iterable is exhausted.
 |
 |  Methods defined here:
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
 |  __iter__(self, /)
 |      Implement iter(self).
 |
 |  __new__(*args, **kwargs) from builtins.type
 |      Create and return a new object.  See help(type) for accurate signature.
 |
 |  __next__(self, /)
 |      Implement next(self).
 |
 |  __reduce__(…)
 |      Return state information for pickling.
help>
You are now leaving help and returning to the Python interpreter.If you want to ask for help on a particular object directly from the interpreter, you can type "help(object)".  Executing "help('string')" has the same effect as typing a particular string at the help> prompt.
>>>
```

30. Python hex()

Hex() Python built-in functions, converts an integer to hexadecimal.

```
>>> hex(16)
```

```
'0x10'
```

```
>>> hex(False)
```

```
'0x0'
```

31. Python id() Function

Python id() returns an object's identity.

```
>>> id(orange)
```

```
100218832
```

```
>>> id({1,2,3})==id({1,3,2})
```

```
True
```

32. Python input()

Input() Python built-in functions, reads and returns a line of string.

```
>>> input("Enter a number")
```

```
Enter a number7
‘7’
```

Note that this returns the input as a string. If we want to take 7 as an integer, we need to apply the int() function to it.

```
>>> int(input("Enter a number"))
```

```
Enter a number7
7
```

33. Python int()

Python int() converts a value to an integer.

```
>>> int('7')
```

```
7
```

34. Python isinstance()

We have seen this one in previous lessons. isinstance() takes a variable and a class as arguments. Then, it returns True if the variable belongs to the class. Otherwise, it returns False.

```
>>> isinstance(0,str)
```

```
False
```

```
>>> isinstance(orange,fruit)
```

```
True
```

35. Python issubclass()

This Python Built In function takes two arguments- two**python classes**. If the first class is a subclass of the second, it returns True. Otherwise, it returns False.

```
>>> issubclass(fruit,fruit)
```

```
True
```

```
>>> class fruit:
pass
>>> class citrus(fruit):
pass
>>> class citrus(fruit):
```

```
True
```

```
>>> issubclass(fruit,citrus)
```

```
False
```

36. Python iter()

Iter() Python built-in functions, returns a **python iterator**for an object.

```
>>> for i in iter([1,2,3]):
print(i)
```

```
1
2
3
```

37. Python len()

We’ve seen Python len() so many times by now. It returns the length of an object.

```
>>> len({1,2,2,3})
```

```
3
```

Here, we get 3 instead of 4, because the set takes the value ‘2’ only once.

38. Python list()

Python list() creates a list from a sequence of values.

```
>>> list({1,3,2,2})
```

```
[1, 2, 3]
```

39. Python locals()

This function returns a dictionary of the current local symbol table.

```
>>> locals()
```

{‘__name__’: ‘__main__’, ‘__doc__’: None, ‘__package__’: None, ‘__loader__’: <class ‘_frozen_importlib.BuiltinImporter’>, ‘__spec__’: None, ‘__annotations__’: {}, ‘__builtins__’: <module ‘builtins’ (built-in)>, ‘fruit’: <class ‘__main__.fruit’>, ‘orange’: <__main__.fruit object at 0x05F937D0>, ‘a’: 2, ‘numbers’: [1, 2, 3], ‘i’: 3, ‘x’: 7, ‘b’: 3, ‘citrus’: <class ‘__main__.citrus’>}

40. Python map()

Like filter(), map() Python built-in functions, takes a function and applies it on an iterable. It maps True or False values on each item in the iterable.

```
>>> list(map(lambda x:x%2==0,[1,2,3,4,5]))
```

[False, True, False, True, False]

41. Python max()

A no-brainer, Python max() returns the item, in a sequence, with the highest value of all.

```
>>> max(2,3,4)
```

4

```
>>> max([3,5,4])
```

5

```
>>> max('hello','Hello')
```

‘hello’

42. Python memoryview()

Python memoryview() shows us the memory view of an argument.

```
>>> a=bytes(4)
>>> memoryview(a)
<memory at 0x05F9A988>
>>> for i in memoryview(a):
print(i)
```

0
0
0
0

43. Python min()

Python min() returns the lowest value in a sequence.

```
>>> min(3,5,1)
```

1

```
>>> min(True,False)
```

False

44. Python next()

This Python Built In function returns the next element from the iterator.

```
>>> myIterator=iter([1,2,3,4,5])
>>> next(myIterator)
```

1

```
>>> next(myIterator)
```

2

```
>>> next(myIterator)
```

3

```
>>> next(myIterator)
```

4

```
>>> next(myIterator)
```

5

Now that we’ve traversed all items, when we call next(), it raises StopIteration.

```
>>> next(myIterator)
Traceback (most recent call last):
File "<pyshell#392>", line 1, in <module>
next(myIterator)
```

StopIteration

45. Python object()

Object() Python built-in functions, creates a featureless object.

```
>>> o=object()
>>> type(o)
```

<class 'object'>

```
>>> dir(o)
```

['__class__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__']

Here, the function type() tells us that it's an object. dir() tells us the object's attributes. But since this does not have the __dict__ attribute, we can't assign to arbitrary attributes.

46. Python oct()

Python oct() converts an integer to its octal representation.

```
>>> oct(7)
```

'0o7'

```
>>> oct(8)
```

'0o10'

```
>>> oct(True)
```

'0o1'

47. Python open()

Python open() lets us open a file. Let's change the current working directory to Desktop.

```
>>> import os
>>> os.chdir('C:\\Users\\lifei\\Desktop')
```

Now, we open the file 'topics.txt'.

```
>>> f=open('topics.txt')
>>> f
<_io.TextIOWrapper name='topics.txt' mode='r' encoding='cp1252'>
>>> type(f)
```

<class '_io.TextIOWrapper'>

To read from the file, we use the read() method.

```
>>> print(f.read())
```

DBMS mappings
projection
union
rdbms vs dbms
doget dopost
how to add maps
OOT
SQL queries
Join
Pattern programs
Output
Default constructor in inheritance

48. Python ord()

The function ord() returns an integer that represents the Unicode point for a given Unicode character.

```
>>> ord('A')
```

65

```
>>> ord('9')
```

57

This is complementary to chr().

```
>>> chr(65)
```

'A'

49. Python pow()

Python pow() takes two arguments- say, x and y. It then returns the value of x to the power of y.

```
>>> pow(3,4)
81
>>> pow(7,0)
1
>>> pow(7,-1)
0.14285714285714285
>>> pow(7,-2)
0.02040816326530612
```

50. Python print()

We don't think we need to explain this anymore. We've been seeing this function since the beginning of this article.

```
>>> print("Okay, next function, please!")
Okay, next function, please!
```

51.Python property()

The function property() returns a property attribute. Alternatively, we can use the syntactic sugar @property. We will learn this in detail in our tutorial on **Python Property**.

52. range()

We've taken a whole tutorial on this. Read up **range() in Python**.

```
>>> for i in range(7,2,-2):
    print(i)
7
5
3
```

53. Python repr()

Python repr() returns a representable string of an object.

```
>>> repr("Hello")
"'Hello'"
>>> repr(7)
'7'
>>> repr(False)
'False'
```

54. Python reversed()

This functions reverses the contents of an iterable and returns an iterator object.

```
>>> a=reversed([3,2,1])
>>> a
<list_reverseiterator object at 0x02E1A230>
>>> for i in a:
    print(i)
1
2
3
>>> type(a)
<class 'list_reverseiterator'>
```

55. Python round()

Python round() rounds off a float to the given number of digits (given by the second argument).

```
>>> round(3.777,2)
3.78
>>> round(3.7,3)
3.7
>>> round(3.7,-1)
0.0
>>> round(377.77,-1)
380.0
```

The rounding factor can be negative.

56. Python set()

Of course, Python set() returns a set of the items passed to it.

```
>>> set([2,2,3,1])
```

{1, 2, 3}

Remember, a set cannot have duplicate values, and isn't indexed, but is ordered. Read on **Sets and Booleans** for the same.

57. Python setattr()

Like getattr(),Python setattr() sets an attribute's value for an object.

```
>>> orange.size
```

7

```
>>> orange.size=8
>>> orange.size
```

8

58. Python slice()

Python slice() returns a slice object that represents the set of indices specified by range(start, stop, step).

```
>>> slice(2,7,2)
```

slice(2, 7, 2)

We can use this to iterate on an iterable like a **string in python**.

```
>>> 'Python'[slice(1,5,2)]
```

'yh'

59. Python sorted()

Like we've seen before, Python sorted() prints out a sorted version of an iterable. It does not, however, alter the iterable.

```
>>> sorted('Python')
```

['P', 'h', 'n', 'o', 't', 'y']

```
>>> sorted([1,3,2])
```

[1, 2, 3]

60. Python staticmethod()

Python staticmethod() creates a static method from a function. A static method is bound to a class rather than to an object. But it can be called on the class or on an object.

```
>>> class fruit:
def sayhi():
print("Hi")
>>> fruit.sayhi=staticmethod(fruit.sayhi)
>>> fruit.sayhi()
```

Hi

You can also use the syntactic sugar @staticmethod for this.

```
>>> class fruit:
@staticmethod
def sayhi():
print("Hi")
>>> fruit.sayhi()
```

Hi

61. Python str()

Python str() takes an argument and returns the string equivalent of it.

```
>>> str('Hello')
```

'Hello'

```
>>> str(7)
```

'7'

```
>>> str(8.7)
```

'8.7'

```
>>> str(False)
```

‘False’

```
>>> str([1,2,3])
```

‘[1, 2, 3]’

62. Python sum()

The function sum() takes an iterable as an argument, and returns the sum of all values.

```
>>> sum([3,4,5],3)
```

15

63. Python super()

Python super() returns a proxy object to let you refer to the parent class.

```
>>> class person:
def __init__(self):
print("A person")
>>> class student(person):
def __init__(self):
super().__init__()
print("A student")
>>> Avery=student()
```

A person
A student

64. Python tuple()

As we’ve seen in our tutorial on **Python Tuples**, the function tuple() lets us create a tuple.

```
>>> tuple([1,3,2])
```

(1, 3, 2)

```
>>> tuple({1:'a',2:'b'})
```

(1, 2)

65. Python type()

We have been seeing the type() function to check the type of object we’re dealing with.

```
>>> type({})
```

<class ‘dict’>

```
>>> type(set())
```

<class ‘set’>

```
>>> type(())
```

<class ‘tuple’>

```
>>> type((1))<span style="background-color: #fafafa;color: #333333;font-family: Verdana, Geneva, sans-serif;font-size: 16px;font-weight: inherit"> </span>
```

<class ‘int’>

```
>>> type((1,))
```

<class ‘tuple’>

66. Python vars()

Python vars() function returns the __dict__ attribute of a class.

```
>>> vars(fruit)
```

mappingproxy({‘__module__’: ‘__main__’, ‘size’: 7, ‘shape’: ’round’, ‘__dict__’: <attribute ‘__dict__’ of ‘fruit’ objects>, ‘__weakref__’: <attribute ‘__weakref__’ of ‘fruit’ objects>, ‘__doc__’: None})

67. Python zip()

Python zip() returns us an iterator of tuples.

```
>>> set(zip([1,2,3],['a','b','c']))
```

{(1, ‘a’), (3, ‘c’), (2, ‘b’)}

```
>>> set(zip([1,2],[3,4,5]))
```

{(1, 3), (2, 4)}

```
>>> a=zip([1,2,3],['a','b','c'])
```

To unzip this, we write the following code.

```
>>> x,y,z=a
```

2.
3. >>> x

(1, ‘a’)

1. >>> y

(2, ‘b’)

1. >>> z

(3, ‘c’)

Isn’t this just like tuple unpacking?

A module allows you to logically organize your Python code. Grouping related code into a module makes the code easier to understand and use. A module is a Python object with arbitrarily named attributes that you can bind and reference.

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

Python Modules

A module is a file containing Python definitions and statements. A module can define functions, classes and variables. A module can also include runnable code. Grouping related code into a module makes the code easier to understand and use.

Example:

```
# A simple module, calc.py

def add(x, y):

    return(x+y)

def subtract(x, y):

    return(x-y)
```

The *import* statement

We can use any Python source file as a module by executing an import statement in some other Python source file.

When interpreter encounters an import statement, it imports the module if the module is present in the search path. A search path is a list of directories that the interpreter searches for importing a module. For example, to import the module calc.py, we need to put the following command at the top of the script :

```
# importing  module calc.py
import calc

print(add(10, 2) )
```

Output:

12

The *from import* Statement

Python's *from* statement lets you import specific attributes from a module. The *from .. import ..* has the following syntax :

```
# importing sqrt() and factorial from the
# module math
from math import sqrt, factorial
```

```
# if we simply do "import math", then
# math.sqrt(16) and math.factorial()
# are required.
print(sqrt(16) )
print(factorial(6) )
```

Output:

4.0

720

Code Snippet illustrating python built-in modules:

```
# importing built-in module math
import math

# using square root(sqrt) function contained
# in math module
print(math.sqrt(25))

# using pi function contained in math module
print (math.pi)

# 2 radians = 114.59 degreees
print(math.degrees(2))

# 60 degrees = 1.04 radians
print(math.radians(60))
```

```
# Sine of 2 radians
printmath.sin(2)

# Cosine of 0.5 radians
printmath.cos(0.5)

# Tangent of 0.23 radians
printmath.tan(0.23)

# 1 * 2 * 3 * 4 = 24
printmath.factorial(4)


# importing built in module random
import random

# printing random integer between 0 and 5
print(random.randint(0, 5))

# print random floating point number between 0 and 1
print(random.random())

# random number between 0 and 100
print(random.random() *100)
```

```
List=[1, 4, True, 800, "python", 27, "hello"]

# using choice function in random module for choosing
# a random element from a set such as a list
print(random.choice(List))


# importing built in module datetime
import datetime
from datetime import date
import time

# Returns the number of seconds since the
# Unix Epoch, January 1st 1970
print(time.time())

# Converts a number of seconds to a date object
printdate.fromtimestamp(454554)
```

Output:

5.0

3.14159265359

114.591559026

1.0471975512

0.909297426826

0.87758256189

0.234143362351

24

3

0.401533172951

88.4917616788

True

1461425771.87

1970-01-06

Example

The Python code for a module named *aname* normally resides in a file named *aname.py*. Here's an example of a simple module, support.py

```
defprint_func( par):
print"Hello : ", par
```

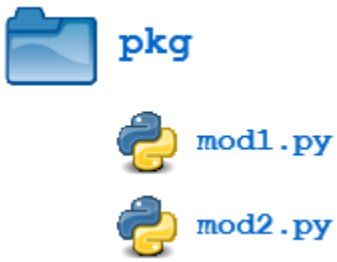
```
return
```

Python Packages

Suppose you have developed a very large application that includes many modules. As the number of modules grows, it becomes difficult to keep track of them all if they are dumped into one location. This is particularly so if they have similar names or functionality. You might wish for a means of grouping and organizing them.

Packages allow for a hierarchical structuring of the module namespace using **dot notation**. In the same way that **modules** help avoid collisions between global variable names, **packages** help avoid collisions between module names.

Creating a **package** is quite straightforward, since it makes use of the operating system's inherent hierarchical file structure. Consider the following arrangement:



Here, there is a directory named `pkg` that contains two modules, `mod1.py` and `mod2.py`. The contents of the modules are:

mod1.py

```
def foo():
print('[mod1] foo()')

class Foo:
pass
```

mod2.py

```
def bar():
print('[mod2] bar()')

class Bar:
pass
```

Given this structure, if the `pkg` directory resides in a location where it can be found (in one of the directories contained in `sys.path`), you can refer to the two **modules** with **dot notation**(`pkg.mod1`, `pkg.mod2`) and import them with the syntax you are already familiar with:

```
import<module_name>[,<module_name>...]
>>>importpkg.mod1,pkg.mod2
>>>pkg.mod1.foo()
[mod1] foo()
>>>x=pkg.mod2.Bar()
>>>x
<pkg.mod2.Bar object at 0x033F7290>
from<module_name>import<name(s)>
>>>frompkg.mod1importfoo
>>>foo()
[mod1] foo()
from<module_name>import<name>as<alt_name>
```

Package Initialization

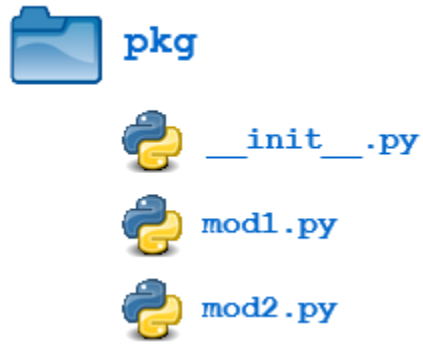
If a file named `__init__.py` is present in a package directory, it is invoked when the package or a module in the package is imported. This can be used for execution of package initialization code, such as initialization of package-level data.

For example, consider the following `__init__.py` file:

__init__.py

```
print(f'Invoking __init__.py for {__name__}')
A=['quux','corge','grault']
```

Let's add this file to the `pkg` directory from the above example:



Now when the package is imported, global list `A` is initialized:

```
>>>importpkg
Invoking __init__.py for pkg
>>>pkg.A
['quux', 'corge', 'grault']
```

A **module** in the package can access the global by importing it in turn:

mod1.py

```
deffoo():
frompkgimportA
print('[mod1] foo() / A = ',A)

classFoo:
pass
>>>frompkgimportmod1
Invoking __init__.py for pkg
>>>mod1.foo()
[mod1] foo() / A =  ['quux', 'corge', 'grault']
```

`__init__.py` can also be used to effect automatic importing of modules from a package. For example, earlier you saw that the statement `import pkg` only

places the name `pkg` in the caller's local symbol table and doesn't import any modules. But if `__init__.py` in the `pkg`directory contains the following:

__init__.py

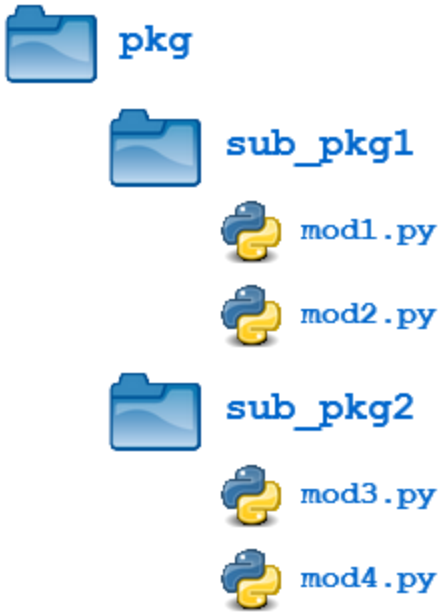
```
print(f'Invoking __init__.py for {__name__}')
importpkg.mod1,pkg.mod2
```

then when you execute `import pkg`, modules `mod1` and `mod2` are imported automatically:

```
>>>importpkg
Invoking __init__.py for pkg
>>>pkg.mod1.foo()
[mod1] foo()
>>>pkg.mod2.bar()
[mod2] bar()
```

Subpackages

Packages can contain nested **subpackages** to arbitrary depth. For example, let's make one more modification to the example **package** directory as follows:



The four modules (`mod1.py`, `mod2.py`, `mod3.py` and `mod4.py`) are defined as previously. But now, instead of being lumped together into the `pkg` directory, they are split out into two **subpackage** directories, `sub_pkg1` and `sub_pkg2`.

Importing still works the same as shown previously. Syntax is similar, but additional **dot notation** is used to separate **package** name from **subpackage** name:

```
>>>importpkg.sub_pkg1.mod1
>>>pkg.sub_pkg1.mod1.foo()
[mod1] foo()

>>>frompkg.sub_pkg1importmod2
>>>mod2.bar()
[mod2] bar()

>>>frompkg.sub_pkg2.mod3importbaz
>>>baz()
[mod3] baz()

>>>frompkg.sub_pkg2.mod4importquxasgrault
>>>grault()
[mod4] qux()
```

In addition, a module in one **subpackage** can reference objects in a **sibling subpackage** (in the event that the sibling contains some functionality that you need). For example, suppose you want to import and execute function `foo()` (defined in module `mod1`) from within module `mod3`. You can either use an **absolute import**:

pkg/sub__pkg2/mod3.py

```
defbaz():
print('[mod3] baz()')

classBaz:
pass

frompkg.sub_pkg1.mod1importfoo
foo()
>>>frompkg.sub_pkg2importmod3
[mod1] foo()
>>>mod3.foo()
[mod1] foo()
```

Or you can use a **relative import**, where `..` refers to the package one level up. From within `mod3.py`, which is in subpackage `sub_pkg2`,

- `..` evaluates to the parent package (`pkg`), and
- `..sub_pkg1` evaluates to subpackage `sub_pkg1` of the parent package.

pkg/sub__pkg2/mod3.py

```
defbaz():
print('[mod3] baz()')

classBaz:
pass

from..importsub_pkg1
```

```
print(sub_pkg1)

from..sub_pkg1.mod1importfoo
foo()
>>>frompkg.sub_pkg2importmod3
<module 'pkg.sub_pkg1' (namespace)>
[mod1] foo()
```

What Are Namespaces?

A namespace is basically a system to make sure that all the names in a program are unique and can be used without any conflict. You might already know that everything in Python—like strings, lists, functions, etc.—is an object. Another interesting fact

is that Python implements namespaces as dictionaries. There is a name-to-object mapping, with the names as keys and the objects as values. Multiple namespaces can use the same name and map it to a different object. Here are a few examples of namespaces:

- Local Namespace: This namespace includes local names inside a function. This namespace is created when a function is called, and it only lasts until the function returns.
- Global Namespace: This namespace includes names from various imported modules that you are using in a project. It is created when the module is included in the project, and it lasts until the script ends.
- Built-in Namespace: This namespace includes built-in functions and built-in exception names.

In the Mathematical Modules in Python series on Envato Tuts+, I wrote about useful mathematical functions available in different modules. For example, the math and cmath modules have a lot of functions that are common to both of them, like `log10()`, `acos()`, `cos()`, `exp()`, etc. If you are using both of these modules in the same

program, the only way to use these functions unambiguously is to prefix them with the name of the module, like `math.log10()` and `cmath.log10()`.

Python has been an object-oriented language since the time it existed. Due to this, creating and using classes and objects are downright easy. This chapter helps you become an expert in using Python's object-oriented programming support.

If you do not have any previous experience with object-oriented (OO) programming, you may want to consult an introductory course on it or at least a tutorial of some sort so that you have a grasp of the basic concepts.

However, here is a small introduction of Object-Oriented Programming (OOP) to help you –

Overview of OOP Terminology

- **Class** – A user-defined prototype for an object that defines a set of attributes that characterize any object of the class. The attributes are data members (class variables and instance variables) and methods, accessed via dot notation.
- **Class variable** – A variable that is shared by all instances of a class. Class variables are defined within a class but outside any of the class's methods. Class variables are not used as frequently as instance variables are.
- **Data member** – A class variable or instance variable that holds data associated with a class and its objects.
- **Function overloading** – The assignment of more than one behavior to a particular function. The operation performed varies by the types of objects or arguments involved.
- **Instance variable** – A variable that is defined inside a method and belongs only to the current instance of a class.
- **Inheritance** – The transfer of the characteristics of a class to other classes that are derived from it.
- **Instance** – An individual object of a certain class. An object obj that belongs to a class Circle, for example, is an instance of the class Circle.
- **Instantiation** – The creation of an instance of a class.
- **Method** – A special kind of function that is defined in a class definition.

- **Object** – A unique instance of a data structure that is defined by its class. An object comprises both data members (class variables and instance variables) and methods.
- **Operator overloading** – The assignment of more than one function to a particular operator.

Creating Classes

The *class* statement creates a new class definition. The name of the class immediately follows the keyword *class* followed by a colon as follows –

```
class ClassName:
   'Optional class documentation string'
   class_suite
```

- The class has a documentation string, which can be accessed via ***ClassName.__doc__***.
- The ***class_suite*** consists of all the component statements defining class members, data attributes and functions.

Example

Following is an example of a simple Python class –

```
class Employee:
   'Common base class for all employees'
   empCount = 0

   def __init__(self, name, salary):
      self.name = name
      self.salary = salary
      Employee.empCount += 1

   def displayCount(self):
     print "Total Employee %d" % Employee.empCount
```

```
   def displayEmployee(self):
      print "Name : ", self.name,  ", Salary: ", self.salary
```

- The variable *empCount* is a class variable whose value is shared among all the instances of a in this class. This can be accessed as *Employee.empCount* from inside the class or outside the class.
- The first method *__init__()* is a special method, which is called class constructor or initialization method that Python calls when you create a new instance of this class.
- You declare other class methods like normal functions with the exception that the first argument to each method is *self*. Python adds the *self* argument to the list for you; you do not need to include it when you call the methods.

Creating Instance Objects

To create instances of a class, you call the class using class name and pass in whatever arguments its *__init__* method accepts.

```
This would create first object of Employee class
emp1 = Employee("Zara", 2000)
This would create second object of Employee class
emp2 = Employee("Manni", 5000)
```

Accessing Attributes

You access the object's attributes using the dot operator with object. Class variable would be accessed using class name as follows –

```
emp1.displayEmployee()
emp2.displayEmployee()
print ("Total Employee %d" % Employee.empCount)
```

Now, putting all the concepts together –

```
#!/usr/bin/python3
```

```
class Employee:
   'Common base class for all employees'
   empCount = 0

   def __init__(self, name, salary):
      self.name = name
      self.salary = salary
      Employee.empCount += 1

   def displayCount(self):
     print ("Total Employee %d" % Employee.empCount)

   def displayEmployee(self):
      print ("Name : ", self.name,  ", Salary: ", self.salary)



#This would create first object of Employee class"
emp1 = Employee("Zara", 2000)
#This would create second object of Employee class"
emp2 = Employee("Manni", 5000)
emp1.displayEmployee()
emp2.displayEmployee()
print ("Total Employee %d" % Employee.empCount)
```

When the above code is executed, it produces the following result –

```
Name :  Zara ,Salary:  2000
Name :  Manni ,Salary:  5000
Total Employee 2
```

You can add, remove, or modify attributes of classes and objects at any time –

```
emp1.salary = 7000  # Add an 'salary' attribute.
emp1.name = 'xyz'  # Modify 'age' attribute.
del emp1.salary  # Delete 'age' attribute.
```

Instead of using the normal statements to access attributes, you can use the following functions –

- The **getattr(obj, name[, default])** – to access the attribute of object.
- The **hasattr(obj,name)** – to check if an attribute exists or not.
- The **setattr(obj,name,value)** – to set an attribute. If attribute does not exist, then it would be created.
- The **delattr(obj, name)** – to delete an attribute.

```
hasattr(emp1, 'salary')    # Returns true if 'salary' attribute exists

getattr(emp1, 'salary')    # Returns value of 'salary' attribute

setattr(emp1, 'salary', 7000) # Set attribute 'salary' at 7000

delattr(emp1, 'salary')    # Delete attribute 'salary'
```

Built-In Class Attributes

Every Python class keeps following built-in attributes and they can be accessed using dot operator like any other attribute –

- **__dict__** – Dictionary containing the class's namespace.
- **__doc__** – Class documentation string or none, if undefined.
- **__name__** – Class name.
- **__module__** – Module name in which the class is defined. This attribute is "__main__" in interactive mode.
- **__bases__** – A possibly empty tuple containing the base classes, in the order of their occurrence in the base class list.

For the above class let us try to access all these attributes –

```
#!/usr/bin/python3
```

```
class Employee:
   'Common base class for all employees'
   empCount = 0

   def __init__(self, name, salary):
      self.name = name
      self.salary = salary
      Employee.empCount += 1

   def displayCount(self):
     print ("Total Employee %d" % Employee.empCount)

   def displayEmployee(self):
      print ("Name : ", self.name,  ", Salary: ", self.salary)

emp1 = Employee("Zara", 2000)
emp2 = Employee("Manni", 5000)
print ("Employee.__doc__:", Employee.__doc__)
print ("Employee.__name__:", Employee.__name__)
print ("Employee.__module__:", Employee.__module__)
print ("Employee.__bases__:", Employee.__bases__)
print ("Employee.__dict__:", Employee.__dict__ )
```

When the above code is executed, it produces the following result –

```
Employee.__doc__: Common base class for all employees
Employee.__name__: Employee
Employee.__module__: __main__
Employee.__bases__: (<class 'object'>,)
Employee.__dict__: {
   'displayCount': <function Employee.displayCount at 0x0160D2B8>,
   '__module__': '__main__', '__doc__': 'Common base class for all employees',
```

```
   'empCount': 2, '__init__':
   <function Employee.__init__ at 0x0124F810>, 'displayEmployee':
   <function Employee.displayEmployee at 0x0160D300>,
   '__weakref__':
   <attribute '__weakref__' of 'Employee' objects>, '__dict__':
   <attribute '__dict__' of 'Employee' objects>
}
```

Destroying Objects (Garbage Collection)

Python deletes unneeded objects (built-in types or class instances) automatically to free the memory space. The process by which Python periodically reclaims blocks of memory that no longer are in use is termed as Garbage Collection.

Python's garbage collector runs during program execution and is triggered when an object's reference count reaches zero. An object's reference count changes as the number of aliases that point to it changes.

An object's reference count increases when it is assigned a new name or placed in a container (list, tuple, or dictionary). The object's reference count decreases when it is deleted with *del*, its reference is reassigned, or its reference goes out of scope. When an object's reference count reaches zero, Python collects it automatically.

```
a = 40      # Create object <40>

b = a       # Increase ref. count  of <40>

c = [b]     # Increase ref. count  of <40>


del a       # Decrease ref. count  of <40>

b = 100     # Decrease ref. count  of <40>

c[0] = -1   # Decrease ref. count  of <40>
```

You normally will not notice when the garbage collector destroys an orphaned instance and reclaims its space. However, a class can implement the special method *__del__()*, called a destructor, that is invoked when the instance is about to be destroyed. This method might be used to clean up any non-memory resources used by an instance.

Example

This __del__() destructor prints the class name of an instance that is about to be destroyed –

```
#!/usr/bin/python3


class Point:

   def __init__( self, x=0, y=0):

      self.x = x

      self.y = y

   def __del__(self):

      class_name = self.__class__.__name__

      print (class_name, "destroyed")


pt1 = Point()

pt2 = pt1

pt3 = pt1

print (id(pt1), id(pt2), id(pt3));   # prints the ids of the obejcts

del pt1

del pt2

del pt3
```

When the above code is executed, it produces the following result –

```
140338326963984 140338326963984 140338326963984
Point destroyed
```

Note – Ideally, you should define your classes in a separate file, then you should import them in your main program file using *import* statement.

In the above example, assuming definition of a Point class is contained in *point.py* and there is no other executable code in it.

```
#!/usr/bin/python3
```

```
import point


p1 = point.Point()
```

Class Inheritance

Instead of starting from a scratch, you can create a class by deriving it from a pre-existing class by listing the parent class in parentheses after the new class name.

The child class inherits the attributes of its parent class, and you can use those attributes as if they were defined in the child class. A child class can also override data members and methods from the parent.

Syntax

Derived classes are declared much like their parent class; however, a list of base classes to inherit from is given after the class name –

```
class SubClassName (ParentClass1[, ParentClass2, ...]):

   'Optional class documentation string'

   class_suite
```

Example

```
#!/usr/bin/python3


class Parent:        # define parent class

   parentAttr = 100

   def __init__(self):

      print ("Calling parent constructor")


   def parentMethod(self):

      print ('Calling parent method')
```

```
   def setAttr(self, attr):
      Parent.parentAttr = attr

   def getAttr(self):
      print ("Parent attribute :", Parent.parentAttr)

class Child(Parent): # define child class
   def __init__(self):
      print ("Calling child constructor")

   def childMethod(self):
      print ('Calling child method')

c = Child()          # instance of child
c.childMethod()      # child calls its method
c.parentMethod()     # calls parent's method
c.setAttr(200)       # again call parent's method
c.getAttr()          # again call parent's method
```

When the above code is executed, it produces the following result –

```
Calling child constructor
Calling child method
Calling parent method
Parent attribute : 200
```

In a similar way, you can drive a class from multiple parent classes as follows –

```
class A:        # define your class A
.....

class B:         # define your calss B
```

```
.....


class C(A, B):   # subclass of A and B

.....
```

You can use issubclass() or isinstance() functions to check a relationships of two classes and instances.

- The **issubclass(sub, sup)** boolean function returns True, if the given subclass **sub** is indeed a subclass of the superclass **sup**.
- The **isinstance(obj, Class)** boolean function returns True, if *obj* is an instance of class *Class* or is an instance of a subclass of Class

Overriding Methods

You can always override your parent class methods. One reason for overriding parent's methods is that you may want special or different functionality in your subclass.

Example

```
#!/usr/bin/python3


class Parent:        # define parent class
   def myMethod(self):
      print ('Calling parent method')


class Child(Parent): # define child class
   def myMethod(self):
      print ('Calling child method')


c = Child()          # instance of child
c.myMethod()         # child calls overridden method
```

When the above code is executed, it produces the following result –

```
Calling child method
```

Base Overloading Methods

The following table lists some generic functionality that you can override in your own classes –

Sr.No.	Method, Description & Sample Call
1	**__init__ (self [,args...])** Constructor (with any optional arguments) Sample Call : *obj = className(args)*
2	**__del__(self)** Destructor, deletes an object Sample Call : *del obj*
3	**__repr__(self)** Evaluatable string representation Sample Call : *repr(obj)*
4	**__str__(self)** Printable string representation Sample Call : *str(obj)*
5	**__cmp__ (self, x)** Object comparison Sample Call : *cmp(obj, x)*

Overloading Operators

Suppose you have created a Vector class to represent two-dimensional vectors. What happens when you use the plus operator to add them? Most likely Python will yell at you.

You could, however, define the *__add__* method in your class to perform vector addition and then the plus operator would behave as per expectation –

Example

```
#!/usr/bin/python3


class Vector:

   def __init__(self, a, b):

      self.a = a

      self.b = b


   def __str__(self):

      return 'Vector (%d, %d)' % (self.a, self.b)


   def __add__(self,other):

      return Vector(self.a + other.a, self.b + other.b)


v1 = Vector(2,10)

v2 = Vector(5,-2)

print (v1 + v2)
```

When the above code is executed, it produces the following result –

```
Vector(7,8)
```

Data Hiding

An object's attributes may or may not be visible outside the class definition. You need to name attributes with a double underscore prefix, and those attributes then will not be directly visible to outsiders.

Example

```
#!/usr/bin/python3


class JustCounter:

   __secretCount = 0


   def count(self):

      self.__secretCount += 1

      print (self.__secretCount)


counter = JustCounter()

counter.count()

counter.count()

print (counter.__secretCount)
```

When the above code is executed, it produces the following result –

```
1
2
Traceback (most recent call last):
   File "test.py", line 12, in <module>
      print counter.__secretCount
AttributeError: JustCounter instance has no attribute '__secretCount'
```

Python protects those members by internally changing the name to include the class name. You can access such attributes as *object._className__attrName*. If you would replace your last line as following, then it works for you –

```
.......................

print (counter._JustCounter__secretCount)
```

When the above code is executed, it produces the following result −

```
1
2
2
```

A *regular expression* is a special sequence of characters that helps you match or find other strings or sets of strings, using a specialized syntax held in a pattern. Regular expressions are widely used in UNIX world.

The module **re** provides full support for Perl-like regular expressions in Python. The re module raises the exception re.error if an error occurs while compiling or using a regular expression.

We would cover two important functions, which would be used to handle regular expressions. But a small thing first: There are various characters, which would have special meaning when they are used in regular expression. To avoid any confusion while dealing with regular expressions, we would use Raw Strings as **r'expression'**.

The *match* Function

This function attempts to match RE *pattern* to *string* with optional *flags*.

Here is the syntax for this function −

```
re.match(pattern, string, flags=0)
```

Here is the description of the parameters −

Sr.No.	Parameter & Description
1	**pattern** This is the regular expression to be matched.
2	**string** This is the string, which would be searched to match the pattern at the beginning of string.
3	**flags** You can specify different flags using bitwise OR (|). These are modifiers, which are listed in the table below.

The *re.match* function returns a **match** object on success, **None** on failure. We use*group(num)* or *groups()* function of **match** object to get matched expression.

Sr.No.	Match Object Method & Description
1	**group(num=0)** This method returns entire match (or specific subgroup num)
2	**groups()** This method returns all matching subgroups in a tuple (empty if there weren't any)

Example

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

```
import re

line="Cats are smarter than dogs"

matchObj=re.match( r'(.*) are (.*?) .*', line,re.M|re.I)

ifmatchObj:

print"matchObj.group() : ",matchObj.group()

print"matchObj.group(1) : ",matchObj.group(1)

print"matchObj.group(2) : ",matchObj.group(2)

else:

print"No match!!"
```

When the above code is executed, it produces following result −

```
matchObj.group() :  Cats are smarter than dogs
matchObj.group(1) :  Cats
matchObj.group(2) :  smarter
```

The *search* Function

This function searches for first occurrence of RE *pattern* within *string* with optional *flags*.

Here is the syntax for this function −

```
re.search(pattern, string, flags=0)
```

Here is the description of the parameters −

Sr.No.	Parameter & Description
1	**pattern** This is the regular expression to be matched.
2	**string** This is the string, which would be searched to match the pattern anywhere in the string.
3	**flags** You can specify different flags using bitwise OR (|). These are modifiers, which are listed in the table below.

The *re.search* function returns a **match** object on success, **none** on failure. We use *group(num)* or *groups()* function of **match** object to get matched expression.

Sr.No.	Match Object Methods & Description

1	**group(num=0)** This method returns entire match (or specific subgroup num)
2	**groups()** This method returns all matching subgroups in a tuple (empty if there weren't any)

Example

```
#!/usr/bin/python

import re

line="Cats are smarter than dogs";



searchObj=re.search( r'(.*) are (.*?) .*', line,re.M|re.I)

ifsearchObj:

print"searchObj.group() : ",searchObj.group()

print"searchObj.group(1) : ",searchObj.group(1)

print"searchObj.group(2) : ",searchObj.group(2)

else:

print"Nothing found!!"
```

When the above code is executed, it produces following result −

```
searchObj.group() :  Cats are smarter than dogs
searchObj.group(1) :  Cats
searchObj.group(2) :  smarter
```

Matching Versus Searching

Python offers two different primitive operations based on regular expressions: **match** checks for a match only at the beginning of the string, while **search**checks for a match anywhere in the string (this is what Perl does by default).

Example

```
#!/usr/bin/python

import re

line="Cats are smarter than dogs";
```

```
matchObj=re.match(r'dogs', line,re.M|re.I)

ifmatchObj:

print"match -->matchObj.group() : ",matchObj.group()

else:

print"No match!!"


searchObj=re.search(r'dogs', line,re.M|re.I)

ifsearchObj:

print"search -->searchObj.group() : ",searchObj.group()

else:

print"Nothing found!!"
```

When the above code is executed, it produces the following result −

```
No match!!
search -->matchObj.group() :  dogs
```

Search and Replace

One of the most important **re** methods that use regular expressions is **sub**.

Syntax

```
re.sub(pattern, repl, string, max=0)
```

This method replaces all occurrences of the RE *pattern* in *string* with *repl*, substituting all occurrences unless *max* provided. This method returns modified string.

Example

```
#!/usr/bin/python

import re

phone="2004-959-559 # This is Phone Number"

# Delete Python-style comments

num=re.sub(r'#.*$',"", phone)

print"Phone Num : ",num

# Remove anything other than digits
```

```
num=re.sub(r'\D',"", phone)

print"Phone Num : ",num
```

When the above code is executed, it produces the following result −

```
Phone Num :  2004-959-559
Phone Num :  2004959559
```

Regular Expression Modifiers: Option Flags

Regular expression literals may include an optional modifier to control various aspects of matching. The modifiers are specified as an optional flag. You can provide multiple modifiers using exclusive OR (|), as shown previously and may be represented by one of these −

Sr.No.	Modifier & Description
1	**re.I** Performs case-insensitive matching.
2	**re.L** Interprets words according to the current locale. This interpretation affects the alphabetic group (\w and \W), as well as word boundary behavior(\b and \B).
3	**re.M** Makes $ match the end of a line (not just the end of the string) and makes ^ match the start of any line (not just the start of the string).
4	**re.S** Makes a period (dot) match any character, including a newline.
5	**re.U** Interprets letters according to the Unicode character set. This flag affects the behavior of \w, \W, \b, \B.
6	**re.X** Permits "cuter" regular expression syntax. It ignores whitespace (except inside a set [] or when escaped by a backslash) and treats unescaped # as a comment marker.

Regular Expression Patterns

Except for control characters, **(+ ? . * ^ $ ( ) [ ] { } | \)**, all characters match themselves. You can escape a control character by preceding it with a backslash.

Following table lists the regular expression syntax that is available in Python −

Sr.No.	Pattern & Description

1	^ Matches beginning of line.
2	**$** Matches end of line.
3	**.** Matches any single character except newline. Using m option allows it to match newline as well.
4	**[...]** Matches any single character in brackets.
5	**[^...]** Matches any single character not in brackets
6	**re*** Matches 0 or more occurrences of preceding expression.
7	**re+** Matches 1 or more occurrence of preceding expression.
8	**re?** Matches 0 or 1 occurrence of preceding expression.
9	**re{ n}** Matches exactly n number of occurrences of preceding expression.
10	**re{ n,}** Matches n or more occurrences of preceding expression.
11	**re{ n, m}** Matches at least n and at most m occurrences of preceding expression.
12	**a| b** Matches either a or b.
13	**(re)** Groups regular expressions and remembers matched text.
14	**(?imx)** Temporarily toggles on i, m, or x options within a regular expression. If in parentheses, only that area is affected.

15	**(?-imx)** Temporarily toggles off i, m, or x options within a regular expression. If in parentheses, only that area is affected.
16	**(?: re)** Groups regular expressions without remembering matched text.
17	**(?imx: re)** Temporarily toggles on i, m, or x options within parentheses.
18	**(?-imx: re)** Temporarily toggles off i, m, or x options within parentheses.
19	**(?#...)** Comment.
20	**(?= re)** Specifies position using a pattern. Doesn't have a range.
21	**(?! re)** Specifies position using pattern negation. Doesn't have a range.
22	**(?> re)** Matches independent pattern without backtracking.
23	**\w** Matches word characters.
24	**\W** Matches nonword characters.
25	**\s** Matches whitespace. Equivalent to [\t\n\r\f].
26	**\S** Matches nonwhitespace.
27	**\d** Matches digits. Equivalent to [0-9].
28	**\D** Matches nondigits.
29	**\A** Matches beginning of string.

30	**\Z** Matches end of string. If a newline exists, it matches just before newline.
31	**\z** Matches end of string.
32	**\G** Matches point where last match finished.
33	**\b** Matches word boundaries when outside brackets. Matches backspace (0x08) when inside brackets.
34	**\B** Matches nonword boundaries.
35	**\n, \t, etc.** Matches newlines, carriage returns, tabs, etc.
36	**\1...\9** Matches nth grouped subexpression.
37	**\10** Matches nth grouped subexpression if it matched already. Otherwise refers to the octal representation of a character code.

Regular Expression Examples

Literal characters

Sr.No.	Example & Description
1	**python** Match "python".

Character classes

Sr.No.	Example & Description
1	**[Pp]ython** Match "Python" or "python"
2	**rub[ye]** Match "ruby" or "rube"
3	**[aeiou]**

	Match any one lowercase vowel
4	**[0-9]** Match any digit; same as [0123456789]
5	**[a-z]** Match any lowercase ASCII letter
6	**[A-Z]** Match any uppercase ASCII letter
7	**[a-zA-Z0-9]** Match any of the above
8	**[^aeiou]** Match anything other than a lowercase vowel
9	**[^0-9]** Match anything other than a digit

Special Character Classes

Sr.No.	Example & Description
1	**.** Match any character except newline
2	**\d** Match a digit: [0-9]
3	**\D** Match a nondigit: [^0-9]
4	**\s** Match a whitespace character: [\t\r\n\f]
5	**\S** Match nonwhitespace: [^ \t\r\n\f]
6	**\w** Match a single word character: [A-Za-z0-9_]
7	**\W** Match a nonword character: [^A-Za-z0-9_]

Repetition Cases

Sr.No.	Example & Description
1	**ruby?** Match "rub" or "ruby": the y is optional
2	**ruby*** Match "rub" plus 0 or more ys
3	**ruby+** Match "rub" plus 1 or more ys
4	**\d{3}** Match exactly 3 digits
5	**\d{3,}** Match 3 or more digits
6	**\d{3,5}** Match 3, 4, or 5 digits

Nongreedy repetition

This matches the smallest number of repetitions −

Sr.No.	Example & Description
1	**<.*>** Greedy repetition: matches "<python>perl>"
2	**<.*?>** Nongreedy: matches "<python>" in "<python>perl>"

Grouping with Parentheses

Sr.No.	Example & Description
1	**\D\d+** No group: + repeats \d
2	**(\D\d)+** Grouped: + repeats \D\d pair
3	**([Pp]ython(,)?)+** Match "Python", "Python, python, python", etc.

Backreferences

This matches a previously matched group again −

Sr.No.	Example & Description
1	**([Pp])ython&\1ails** Match python&pails or Python&Pails
2	**(['"])[^\1]*\1** Single or double-quoted string. \1 matches whatever the 1st group matched. \2 matches whatever the 2nd group matched, etc.

Alternatives

Sr.No.	Example & Description
1	**python|perl** Match "python" or "perl"
2	**rub(y|le))** Match "ruby" or "ruble"
3	**Python(!+|\?)** "Python" followed by one or more !or one ?

Anchors

This needs to specify match position.

Sr.No.	Example & Description
1	**^Python** Match "Python" at the start of a string or internal line
2	**Python$** Match "Python" at the end of a string or line
3	**\APython** Match "Python" at the start of a string
4	**Python\Z** Match "Python" at the end of a string
5	**\bPython\b** Match "Python" at a word boundary
6	**\brub\B** \B is nonword boundary: match "rub" in "rube" and "ruby" but not alone

7	**Python(?=!)** Match "Python", if followed by an exclamation point.
8	**Python(?!!)** Match "Python", if not followed by an exclamation point.

Special Syntax with Parentheses

Sr.No.	Example & Description
1	**R(?#comment)** Matches "R". All the rest is a comment
2	**R(?i)uby** Case-insensitive while matching "uby"
3	**R(?i:uby)** Same as above
4	**rub(?:y|le))** Group only without creating \1 backreference

PYTHON - GUI PROGRAMMING TkinterTkinter

Python provides various options for developing graphical user interfaces (GUIs). Most important are listed below.

- **Tkinter** − Tkinter is the Python interface to the Tk GUI toolkit shipped with Python. We would look this option in this chapter.
- **wxPython** − This is an open-source Python interface for wxWindows.
- **JPython** − JPython is a Python port for Java which gives Python scripts seamless access to Java class libraries on the local machine.

There are many other interfaces available, which you can find them on the net.

Tkinter Programming

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

```
#!/usr/bin/python

import tkinter
top = tkinter.Tk()
# Code to add widgets will go here...
top.mainloop()
```

This would create a following window −

Tkinter Widgets

Tkinter provides various controls, such as buttons, labels and text boxes used in a GUI application. These controls are commonly called widgets.

There are currently 15 types of widgets in Tkinter. We present these widgets as well as a brief description in the following table −

Sr.No.	Operator & Description
1	Button The Button widget is used to display buttons in your application.
2	Canvas The Canvas widget is used to draw shapes, such as lines, ovals, polygons and rectangles, in your application.
3	Checkbutton The Checkbutton widget is used to display a number of options as checkboxes. The user can select multiple options at a time.
4	Entry The Entry widget is used to display a single-line text field for accepting values from a user.
5	Frame The Frame widget is used as a container widget to organize other widgets.
6	Label The Label widget is used to provide a single-line caption for other widgets. It can also contain images.
7	Listbox The Listbox widget is used to provide a list of options to a user.
8	Menubutton The Menubutton widget is used to display menus in your application.
9	Menu The Menu widget is used to provide various commands to a user. These commands are contained inside Menubutton.
10	Message The Message widget is used to display multiline text fields for accepting values from a user.
11	Radiobutton The Radiobutton widget is used to display a number of options as radio buttons. The user can select only one option at a time.
12	Scale The Scale widget is used to provide a slider widget.
13	Scrollbar The Scrollbar widget is used to add scrolling capability to various widgets, such as list boxes.
14	Text The Text widget is used to display text in multiple lines.
15	Toplevel The Toplevel widget is used to provide a separate window container.
16	Spinbox The Spinbox widget is a variant of the standard Tkinter Entry widget, which can be used to select from a fixed number of values.
17	PanedWindow A PanedWindow is a container widget that may contain any number of panes, arranged horizontally or vertically.
18	LabelFrame A labelframe is a simple container widget. Its primary purpose is to act as a spacer or container for complex window layouts.
19	tkMessageBox This module is used to display message boxes in your applications.

Write Any 10 in Exams

Let us study these widgets in detail −

1. **Button**:To add a button in your application, this widget is used. The general syntax is:

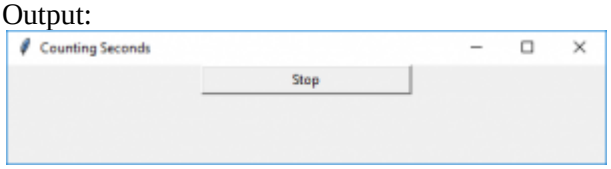
   ```
   w=Button(master, option=value)
   ```

   master is the parameter used to represent the parent window. There are number of options which are used to change the format of the Buttons. Number of options can be passed as parameters separated by commas. Some of them are listed below.

   - **activebackground**: to set the background color when button is under the cursor.
   - **activeforeground**: to set the foreground color when button is under the cursor.
   - **bg**: to set he normal background color.
   - **command**: to call a function.
   - **font**: to set the font on the button label.
   - **image**: to set the image on the button.
   - **width**: to set the width of the button.
   - **height**: to set the height of the button.

```
import tkinter as tk
r = tk.Tk()
r.title('Counting Seconds')
button = tk.Button(r, text='Stop', width=25, command=r.destroy)
button.pack()
r.mainloop()
```

Output:



2. **Canvas:** It is used to draw pictures and other complex layout like graphics, text and widgets. The general syntax is:

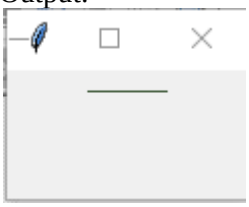
   ```
   w = Canvas(master, option=value)

   master is the parameter used to represent the parent window.
   ```

   There are number of options which are used to change the format of the widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

   - **bd**: to set the border width in pixels.
   - **bg**: to set the normal background color.
   - **cursor**: to set the cursor used in the canvas.
   - **highlightcolor**: to set the color shown in the focus highlight.
   - **width**: to set the width of the widget.
   - **height**: to set the height of the widget.

```
from tkinter import *
master = Tk()
w = Canvas(master, width=40, height=60)
w.pack()
canvas_height=20
canvas_width=200
y = int(canvas_height / 2)
w.create_line(0, y, canvas_width, y )
mainloop()
```

Output:



3. **CheckButton:** To select any number of options by displaying a number of options to a user as toggle buttons. The general syntax is:

```
w = CheckButton(master, option=value)
```

There are number of options which are used to change the format of this widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

- **Title**: To set the title of the widget.
- **activebackground**: to set the background color when widget is under the cursor.
- **activeforeground**: to set the foreground color when widget is under the cursor.
- **bg**: to set he normal backgrouSteganography
  Break

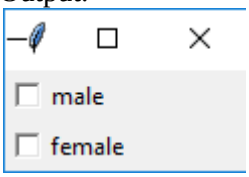
  Secret Code:

  Attach a File:nd color.

- **command**: to call a function.
- **font**: to set the font on the button label.
- **image**: to set the image on the widget.

```
from tkinter import *
master = Tk()
var1 = IntVar()
Checkbutton(master, text='male', variable=var1).grid(row=0, sticky=W)
var2 = IntVar()
Checkbutton(master, text='female', variable=var2).grid(row=1, sticky=W)
mainloop()
```

Output:



4. **Entry:**It is used to input the single line text entry from the user.. For multi-line text input, Text widget is used. The general syntax is:

```
w=Entry(master, option=value)
```

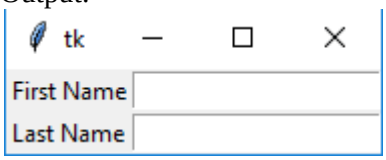
master is the parameter used to represent the parent window.
There are number of options which are used to change the format of the widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

- **bd**: to set the border width in pixels.
- **bg**: to set the normal background color.
- **cursor**: to set the cursor used.
- **command**: to call a function.
- **highlightcolor**: to set the color shown in the focus highlight.
- **width**: to set the width of the button.
- **height**: to set the height of the button.

```
from tkinter import *
master = Tk()
Label(master, text='First Name').grid(row=0)
```

```
Label(master, text='Last Name').grid(row=1)
e1 = Entry(master)
e2 = Entry(master)
e1.grid(row=0, column=1)
e2.grid(row=1, column=1)
mainloop()
```

Output:



5. **Frame:** It acts as a container to hold the widgets. It is used for grouping and organizing the widgets. The general syntax is:

   w = Frame(master, option=value)

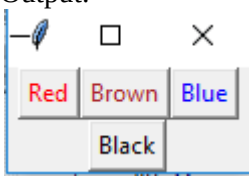
   master is the parameter used to represent the parent window.

   There are number of options which are used to change the format of the widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

   - **highlightcolor**: To set the color of the focus highlight when widget has to be focused.
   - **bd**: to set the border width in pixels.
   - **bg**: to set the normal background color.
   - **cursor**: to set the cursor used.
   - **width**: to set the width of the widget.
   - **height**: to set the height of the widget.

```
from tkinter import *

root = Tk()
frame = Frame(root)
frame.pack()
bottomframe = Frame(root)
bottomframe.pack( side = BOTTOM )
redbutton = Button(frame, text = 'Red', fg ='red')
redbutton.pack( side = LEFT)
greenbutton = Button(frame, text = 'Brown', fg='brown')
greenbutton.pack( side = LEFT )
bluebutton = Button(frame, text ='Blue', fg ='blue')
bluebutton.pack( side = LEFT )
blackbutton = Button(bottomframe, text ='Black', fg ='black')
blackbutton.pack( side = BOTTOM)
root.mainloop()
```

Output:



6. **Label**: It refers to the display box where you can put any text or image which can be updated any time as per the code.
   The general syntax is:

   w=Label(master, option=value)

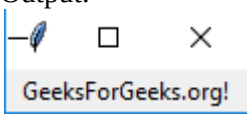
   master is the parameter used to represent the parent window.

   There are number of options which are used to change the format of the widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

- **bg**: to set he normal background color.
- **bg** to set he normal background color.
- **command**: to call a function.
- **font**: to set the font on the button label.
- **image**: to set the image on the button.
- **width**: to set the width of the button.
- **height**" to set the height of the button.

```
from tkinter import *
root = Tk()
w = Label(root, text='GeeksForGeeks.org!')
w.pack()
root.mainloop()
```

Output:



7. **Listbox**: It offers a list to the user from which the user can accept any number of options. The general syntax is:

```
w = Listbox(master, option=value)
```

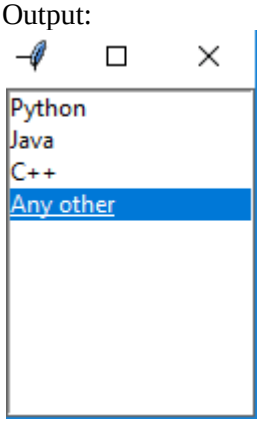
master is the parameter used to represent the parent window.

There are number of options which are used to change the format of the widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

- **highlightcolor**: To set the color of the focus highlight when widget has to be focused.
- **bg**: to set he normal background color.
- **bd**: to set the border width in pixels.
- **font**: to set the font on the button label.
- **image**: to set the image on the widget.
- **width**: to set the width of the widget.
- **height**: to set the height of the widget.

```
from tkinter import *

top = Tk()
Lb = Listbox(top)
Lb.insert(1, 'Python')
Lb.insert(2, 'Java')
Lb.insert(3, 'C++')
Lb.insert(4, 'Any other')
Lb.pack()
top.mainloop()
```

Output:



8. **MenuButton**: It is a part of top-down menu which stays on the window all the time. Every menubutton has its own functionality. The general syntax is:

   w = MenuButton(master, option=value)

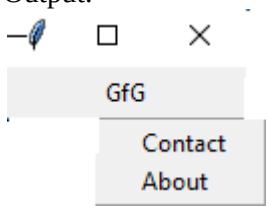
   master is the parameter used to represent the parent window.

   There are number of options which are used to change the format of the widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

   - **activebackground**: To set the background when mouse is over the widget.
   - **activeforeground**: To set the foreground when mouse is over the widget.
   - **bg**: to set he normal background color.
   - **bd**: to set the size of border around the indicator.
   - **cursor**: To appear the cursor when the mouse over the menubutton.
   - **image**: to set the image on the widget.
   - **width**: to set the width of the widget.
   - **height**: to set the height of the widget.
   - **highlightcolor**: To set the color of the focus highlight when widget has to be focused.

```
from tkinter import *

top = Tk()
mb =  Menubutton ( top, text = &amp;quot;GfG&amp;quot;)
mb.grid()
mb.menu  =  Menu ( mb, tearoff = 0 )
mb[&amp;quot;menu&amp;quot;]  =  mb.menu
cVar  = IntVar()
aVar = IntVar()
mb.menu.add_checkbutton ( label ='Contact', variable = cVar )
mb.menu.add_checkbutton ( label = 'About', variable = aVar )
mb.pack()
top.mainloop()
```

   Output:



9. **Menu**: It is used to create all kinds of menus used by the application. The general syntax is:

   w = Menu(master, option=value)

   master is the parameter used to represent the parent window.

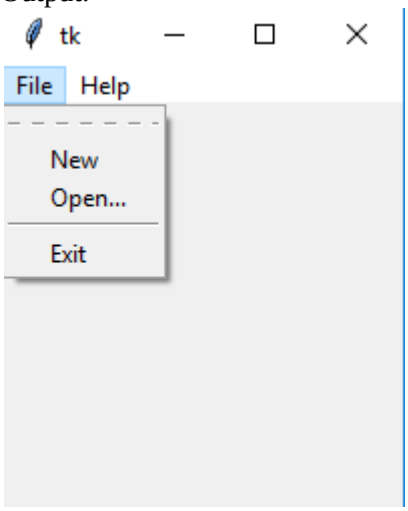
   There are number of options which are used to change the format of this widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

   - **title**: To set the title of the widget.
   - **activebackground**: to set the background color when widget is under the cursor.
   - **activeforeground**: to set the foreground color when widget is under the cursor.
   - **bg**: to set he normal background color.
   - **command**: to call a function.
   - **font**: to set the font on the button label.
   - **image**: to set the image on the widget.

```
from tkinter import *

root = Tk()
menu = Menu(root)
```

```
root.config(menu=menu)
filemenu = Menu(menu)
menu.add_cascade(label='File', menu=filemenu)
filemenu.add_command(label='New')
filemenu.add_command(label='Open...')
filemenu.add_separator()
filemenu.add_command(label='Exit', command=root.quit)
helpmenu = Menu(menu)
menu.add_cascade(label='Help', menu=helpmenu)
helpmenu.add_command(label='About')
mainloop()
```

Output:



10. **Message**: It refers to the multi-line and non-editable text. It works same as that of Label. The general syntax is:

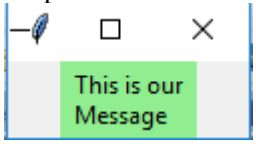
```
w = Message(master, option=value)

master is the parameter used to represent the parent window.
```

There are number of options which are used to change the format of the widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

- **bd**: to set the border around the indicator.
- **bg**: to set he normal background color.
- **font**: to set the font on the button label.
- **image**: to set the image on the widget.
- **width**: to set the width of the widget.
- **height**: to set the height of the widget.

```
from tkinter import *
main = Tk()
ourMessage ='This is our Message'
messageVar = Message(main, text = ourMessage)
messageVar.config(bg='lightgreen')
messageVar.pack( )
main.mainloop( )
```

Output:



11. **RadioButton:** It is used to offer multi-choice option to the user. It offers several options to the user and the user has to choose one option.
The general syntax is:

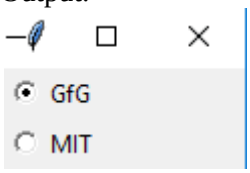
```
w = RadioButton(master, option=value)
```

There are number of options which are used to change the format of this widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

- **activebackground**: to set the background color when widget is under the cursor.
- **activeforeground**: to set the foreground color when widget is under the cursor.
- **bg**: to set he normal background color.
- **command**: to call a function.
- **font**: to set the font on the button label.
- **image**: to set the image on the widget.
- **width**: to set the width of the label in characters.
- **height**: to set the height of the label in characters.

```
from tkinter import *
root = Tk()
v = IntVar()
Radiobutton(root, text='GfG', variable=v, value=1).pack(anchor=W)
Radiobutton(root, text='MIT', variable=v, value=2).pack(anchor=W)
mainloop()
```

Output:



12. **Scale:** It is used to provide a graphical slider that allows to select any value from that scale. The general syntax is:

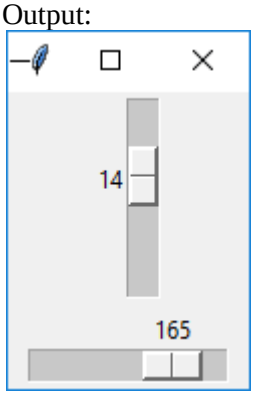
```
w = Scale(master, option=value)

master is the parameter used to represent the parent window.
```

There are number of options which are used to change the format of the widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

- **cursor**: To change the cursor pattern when the mouse is over the widget.
- **activebackground**: To set the background of the widget when mouse is over the widget.
- **bg**: to set he normal background color.
- **orient**: Set it to HORIZONTAL or VERTICAL according to the requirement.
- **from_**: To set the value of one end of the scale range.
- **to**: To set the value of the other end of the scale range.
- **image**: to set the image on the widget.
- **width**: to set the width of the widget.

```
from tkinter import *
master = Tk()
w = Scale(master, from_=0, to=42)
w.pack()
w = Scale(master, from_=0, to=200, orient=HORIZONTAL)
w.pack()
mainloop()
```

Output:



13. **Scrollbar**: It refers to the slide controller which will be used to implement listed widgets. The general syntax is:

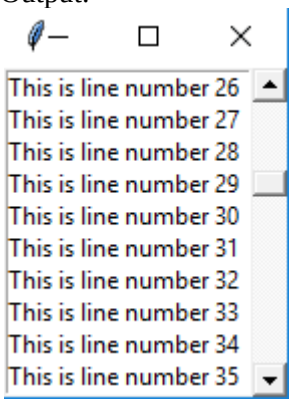
```
w = Scrollbar(master, option=value)

master is the parameter used to represent the parent window.
```

There are number of options which are used to change the format of the widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

- **width**: to set the width of the widget.
- **activebackground**: To set the background when mouse is over the widget.
- **bg**: to set he normal background color.
- **bd**: to set the size of border around the indicator.
- **cursor**: To appear the cursor when the mouse over the menubutton.

```
from tkinter import *
root = Tk()
scrollbar = Scrollbar(root)
scrollbar.pack( side = RIGHT, fill = Y )
mylist = Listbox(root, yscrollcommand = scrollbar.set )
for line in range(100):
   mylist.insert(END, 'This is line number' + str(line))
mylist.pack( side = LEFT, fill = BOTH )
scrollbar.config( command = mylist.yview )
mainloop()
```

Output:



14. **Text:** To edit a multi-line text and format the way it has to be displayed. The general syntax is:

```
w  =Text(master, option=value)
```

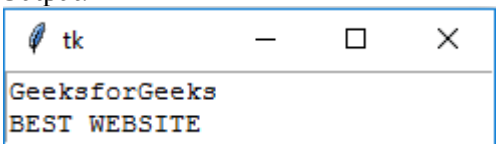
There are number of options which are used to change the format of the text. Number of options can be passed as parameters separated by commas. Some of them are listed below.

- **highlightcolor**: To set the color of the focus highlight when widget has to be focused.
- **insertbackground**: To set the background of the widget.

- **bg**: to set he normal background color.
- **font**: to set the font on the button label.
- **image**: to set the image on the widget.
- **width**: to set the width of the widget.
- **height**: to set the height of the widget.

```
from tkinter import *
root = Tk()
T = Text(root, height=2, width=30)
T.pack()
T.insert(END, 'GeeksforGeeks\nBEST WEBSITE\n')
mainloop()
```

Output:



15. **TopLevel:** This widget is directly controlled by the window manager. It don't need any parent window to work on.The general syntax is:

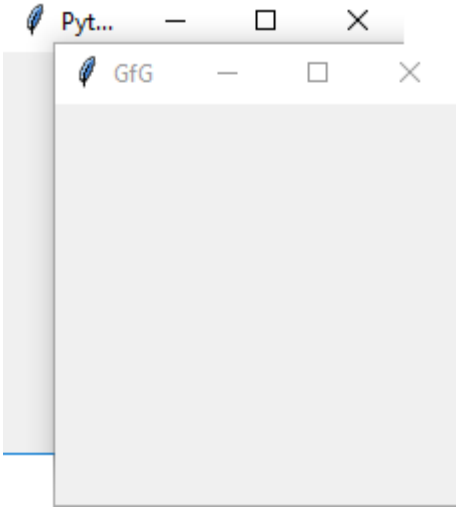
```
w = TopLevel(master, option=value)
```

There are number of options which are used to change the format of the widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

- **bg**: to set he normal background color.
- **bd**: to set the size of border around the indicator.
- **cursor**: To appear the cursor when the mouse over the menubutton.
- **width**: to set the width of the widget.
- **height**: to set the height of the widget.

```
from tkinter import *
root = Tk()
root.title('GfG')
top = Toplevel()
top.title('Python')
top.mainloop()
```

Output:



16. **SpinBox:** It is an entry of 'Entry' widget. Here, value can be input by selecting a fixed value of numbers.The general syntax is:

```
w = SpinBox(master, option=value)
```

There are number of options which are used to change the format of the widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

- **bg**: to set he normal background color.
- **bd**: to set the size of border around the indicator.
- **cursor**: To appear the cursor when the mouse over the menubutton.
- **command**: To call a function.
- **width**: to set the width of the widget.
- **activebackground**: To set the background when mouse is over the widget.
- **disabledbackground**: To disable the background when mouse is over the widget.
- **from_**: To set the value of one end of the range.
- **to**: To set the value of the other end of the range.

```
from tkinter import *
master = Tk()
w = Spinbox(master, from_ = 0, to = 10)
w.pack()
mainloop()
```

Output:

17. **PannedWindow**It is a container widget which is used to handle number of panes arranged in it. The general syntax is:

```
w = PannedWindow(master, option=value)
```

master is the parameter used to represent the parent window. There are number of options which are used to change the format of the widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

- **bg**: to set he normal background color.
- **bd**: to set the size of border around the indicator.
- **cursor**: To appear the cursor when the mouse over the menubutton.
- **width**: to set the width of the widget.
- **height**: to set the height of the widget.

```
from tkinter import *
m1 = PanedWindow()
m1.pack(fill = BOTH, expand = 1)
left = Entry(m1, bd = 5)
m1.add(left)
m2 = PanedWindow(m1, orient = VERTICAL)
m1.add(m2)
top = Scale( m2, orient = HORIZONTAL)
m2.add(top)
mainloop()
```

Output:

Standard attributes

Let us take a look at how some of their common attributes. such as sizes, colors and fonts are specified.

- Dimensions
- Colors
- Fonts

- Anchors
- Relief styles
- Bitmaps
- Cursors

Geometry Management

All Tkinter widgets have access to specific geometry management methods, which have the purpose of organizing widgets throughout the parent widget area. Tkinter exposes the following geometry manager classes: pack, grid, and place.

- The *pack* Method − This geometry manager organizes widgets in blocks before placing them in the parent widget.
- The *grid* Method − This geometry manager organizes widgets in a table-like structure in the parent widget.
- The *place* Method − This geometry manager organizes widgets by placing them in a specific position in the parent widget.

Let us study the geometry management methods briefly −

PYTHON - TKINTER PACK METHOD

This geometry manager organizes widgets in blocks before placing them in the parent widget.

Syntax

```
widget.pack( pack_options )
```

Here is the list of possible options −

- **expand** − When set to true, widget expands to fill any space not otherwise used in widget's parent.
- **fill** − Determines whether widget fills any extra space allocated to it by the packer, or keeps its own minimal dimensions: NONE defaultdefault, X fillonlyhorizontallyfillonlyhorizontally,
  Y fillonlyverticallyfillonlyvertically, or
  BOTH fillbothhorizontallyandverticallyfillbothhorizontallyandvertically.
- **side** − Determines which side of the parent widget packs against: TOP defaultdefault, BOTTOM, LEFT, or RIGHT.

Example

Try the following example by moving cursor on different buttons −

```
from Tkinter import *

root = Tk()

frame = Frame(root)

frame.pack()

bottomframe = Frame(root)

bottomframe.pack( side = BOTTOM )
```

```
redbutton = Button(frame, text="Red", fg="red")
redbutton.pack( side = LEFT)
greenbutton = Button(frame, text="green", fg="green")
greenbutton.pack( side = LEFT )
bluebutton = Button(frame, text="Blue", fg="blue")
bluebutton.pack( side = LEFT )
blackbutton = Button(bottomframe, text="Black", fg="black")
blackbutton.pack( side = BOTTOM)
root.mainloop()
```

When the above code is executed, it produces the following result −

PYTHON - TKINTER GRID METHOD

This geometry manager organizes widgets in a table-like structure in the parent widget.

Syntax

```
widget.grid( grid_options )
```

Here is the list of possible options −

- **column** − The column to put widget in; default 0 leftmostcolumnleftmostcolumn.
- **columnspan** − How many columns widgetoccupies; default 1.
- **ipadx, ipady** − How many pixels to pad widget, horizontally and vertically, inside widget's borders.
- **padx, pady** − How many pixels to pad widget, horizontally and vertically, outside v's borders.
- **row** − The row to put widget in; default the first row that is still empty.
- **rowspan** − How many rowswidget occupies; default 1.
- **sticky** − What to do if the cell is larger than widget. By default, with sticky='', widget is centered in its cell. sticky may be the string concatenation of zero or more of N, E, S, W, NE, NW, SE, and SW, compass directions indicating the sides and corners of the cell to which widget sticks.

Example

Try the following example by moving cursor on different buttons −

```
import Tkinter
root = Tkinter.Tk(  )
for r in range(3):
```

```
  for c in range(4):
    Tkinter.Label(root, text='R%s/C%s'%(r,c),
      borderwidth=1 ).grid(row=r,column=c)
root.mainloop( )
```

This would produce the following result displaying 12 labels arrayed in a 3 × 4 grid −

PYTHON - TKINTER PLACE METHOD

This geometry manager organizes widgets by placing them in a specific position in the parent widget.

Syntax

```
widget.place( place_options )
```

Here is the list of possible options −

- **anchor** − The exact spot of widget other options refer to: may be N, E, S, W, NE, NW, SE, or SW, compass directions indicating the corners and sides of widget; default is NW theupperleftcornerofwidgettheupperleftcornerofwidget
- **bordermode** − INSIDE thedefaultthedefault to indicate that other options refer to the parent's inside ignoringtheparent′sborderignoringtheparent′sborder; OUTSIDE otherwise.
- **height, width** − Height and width in pixels.
- **relheight, relwidth** − Height and width as a float between 0.0 and 1.0, as a fraction of the height and width of the parent widget.
- **relx, rely** − Horizontal and vertical offset as a float between 0.0 and 1.0, as a fraction of the height and width of the parent widget.
- **x, y** − Horizontal and vertical offset in pixels.

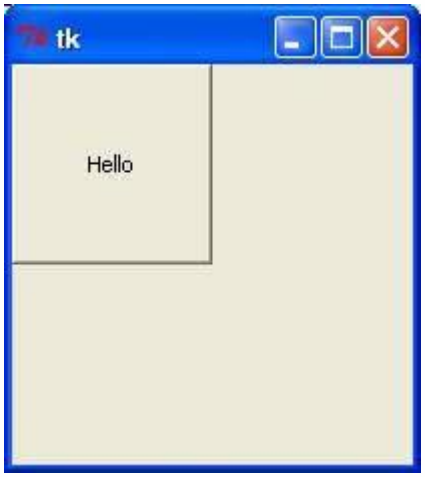
Example

Try the following example by moving cursor on different buttons −

```
from Tkinter import *
import tkMessageBox
import Tkinter
top = Tkinter.Tk()
def helloCallBack():
  tkMessageBox.showinfo( "Hello Python", "Hello World")
```

```
B = Tkinter.Button(top, text ="Hello", command = helloCallBack)

B.pack()

B.place(bordermode=OUTSIDE, height=100, width=100)

top.mainloop()
```

When the above code is executed, it produces the following result −



The Common Gateway Interface, or CGI, is a set of standards that define how information is exchanged between the web server and a custom script. The CGI specs are currently maintained by the NCSA.

What is CGI?

- The Common Gateway Interface, or CGI, is a standard for external gateway programs to interface with information servers such as HTTP servers.
- The current version is CGI/1.1 and CGI/1.2 is under progress.

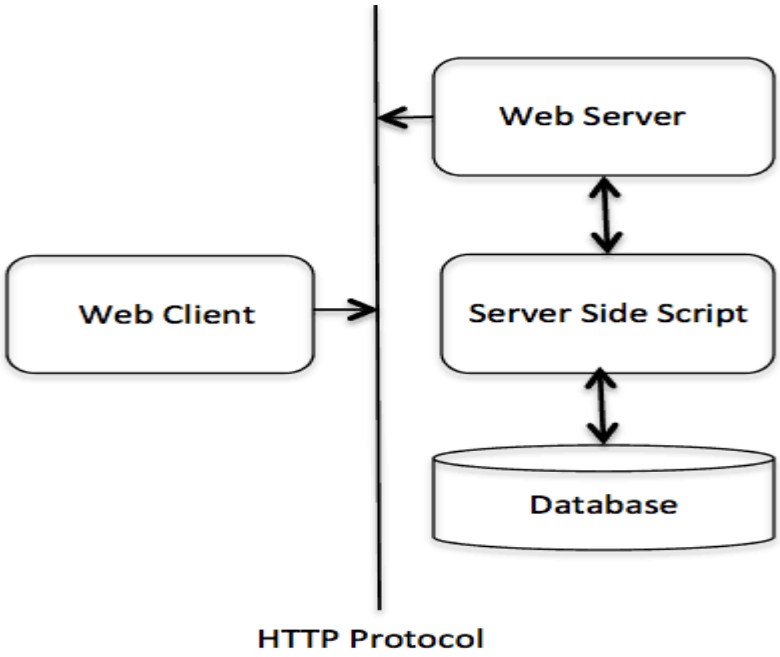
Web Browsing

To understand the concept of CGI, let us see what happens when we click a hyper link to browse a particular web page or URL.

- Your browser contacts the HTTP web server and demands for the URL, i.e., filename.
- Web Server parses the URL and looks for the filename. If it finds that file then sends it back to the browser, otherwise sends an error message indicating that you requested a wrong file.
- Web browser takes response from web server and displays either the received file or error message.

However, it is possible to set up the HTTP server so that whenever a file in a certain directory is requested that file is not sent back; instead it is executed as a program, and whatever that program outputs is sent back for your browser to display. This function is called the Common Gateway Interface or CGI and the programs are called CGI scripts. These CGI programs can be a Python Script, PERL Script, Shell Script, C or C++ program, etc.

CGI Architecture Diagram



Web Server Support and Configuration

Before you proceed with CGI Programming, make sure that your Web Server supports CGI and it is configured to handle CGI Programs. All the CGI Programs to be executed by the HTTP server are kept in a pre-configured directory. This directory is called CGI Directory and by convention it is named as /var/www/cgi-bin. By convention, CGI files have extension as. **cgi,** but you can keep your files with python extension **.py** as well.

By default, the Linux server is configured to run only the scripts in the cgi-bin directory in /var/www. If you want to specify any other directory to run your CGI scripts, comment the following lines in the httpd.conf file −

```
<Directory"/var/www/cgi-bin">

AllowOverride None

Options ExecCGI

Order allow,deny

Allow from all

</Directory>

<Directory"/var/www/cgi-bin">

Options All

</Directory>
```

Here, we assume that you have Web Server up and running successfully and you are able to run any other CGI program like Perl or Shell, etc.

First CGI Program

Here is a simple link, which is linked to a CGI script called hello.py. This file is kept in /var/www/cgi-bin directory and it has following content. Before running your CGI program, make sure you have change mode of file using **chmod755 hello.py** UNIX command to make file executable.

```
#!/usr/bin/python

print("Content-type:text/html\r\n\r\n")
print('<html>')
print('<head>')
print('<title>Hello Word - First CGI Program</title>')
print('</head>')
print('<body>')
print('<h2>Hello Word! This is my first CGI program</h2>')
print('</body>')
print('</html>')
```

If you click hello.py, then this produces the following output −

Hello Word! This is my first CGI program

This hello.py script is a simple Python script, which writes its output on STDOUT file, i.e., screen. There is one important and extra feature available which is first line to be printed **Content-type:text/html\r\n\r\n**. This line is sent back to the browser and it specifies the content type to be displayed on the browser screen. By now you must have understood basic concept of CGI and you can write many complicated CGI programs using Python. This script can interact with any other external system also to exchange information such as RDBMS.

HTTP Header

The line **Content-type:text/html\r\n\r\n** is part of HTTP header which is sent to the browser to understand the content. All the HTTP header will be in the following form −

```
HTTP Field Name: Field Content

For Example
Content-type: text/html\r\n\r\n
```

There are few other important HTTP headers, which you will use frequently in your CGI Programming.

Sr.No.	Header & Description
1	**Content-type:** A MIME string defining the format of the file being returned. Example is Content-type:text/html
2	**Expires: Date** The date the information becomes invalid. It is used by the browser to decide when a page needs to be refreshed. A valid date string is in the format 01 Jan 1998 12:00:00 GMT.
3	**Location: URL** The URL that is returned instead of the URL requested. You can use this field to redirect a request to any file.
4	**Last-modified: Date** The date of last modification of the resource.
5	**Content-length: N** The length, in bytes, of the data being returned. The browser uses this value to report the

	estimated download time for a file.
6	**Set-Cookie: String** Set the cookie passed through the *string*

CGI Environment Variables

All the CGI programs have access to the following environment variables. These variables play an important role while writing any CGI program.

Sr.No.	Variable Name & Description
1	**CONTENT_TYPE** The data type of the content. Used when the client is sending attached content to the server. For example, file upload.
2	**CONTENT_LENGTH** The length of the query information. It is available only for POST requests.
3	**HTTP_COOKIE** Returns the set cookies in the form of key & value pair.
4	**HTTP_USER_AGENT** The User-Agent request-header field contains information about the user agent originating the request. It is name of the web browser.
5	**PATH_INFO** The path for the CGI script.
6	**QUERY_STRING** The URL-encoded information that is sent with GET method request.
7	**REMOTE_ADDR** The IP address of the remote host making the request. This is useful logging or for authentication.
8	**REMOTE_HOST** The fully qualified name of the host making the request. If this information is not available, then REMOTE_ADDR can be used to get IR address.
9	**REQUEST_METHOD** The method used to make the request. The most common methods are GET and POST.
10	**SCRIPT_FILENAME** The full path to the CGI script.
11	**SCRIPT_NAME** The name of the CGI script.
12	**SERVER_NAME** The server's hostname or IP Address
13	**SERVER_SOFTWARE** The name and version of the software the server is running.

Here is small CGI program to list out all the CGI variables. Click this link to see the result Get Environment

```
#!/usr/bin/python

import os
print("Content-type: text/html\r\n\r\n";)
print("<font size=+1>Environment</font><\br>");
forparaminos.environ.keys():
print("<b>%20s</b>: %s<\br>"%(param,os.environ[param]))
```

GET and POST Methods

You must have come across many situations when you need to pass some information from your browser to web server and ultimately to your CGI Program. Most frequently, browser uses two methods two pass this information to web server. These methods are GET Method and POST Method.

Passing Information using GET method

The GET method sends the encoded user information appended to the page request. The page and the encoded information are separated by the ?character as follows −

```
http://www.test.com/cgi-bin/hello.py?key1=value1&key2=value2
```

The GET method is the default method to pass information from browser to web server and it produces a long string that appears in your browser's Location:box. Never use GET method if you have password or other sensitive information to pass to the server. The GET method has size limitation: only 1024 characters can be sent in a request string. The GET method sends information using QUERY_STRING header and will be accessible in your CGI Program through QUERY_STRING environment variable.

You can pass information by simply concatenating key and value pairs along with any URL or you can use HTML <FORM> tags to pass information using GET method.

Simple URL Example:Get Method

Here is a simple URL, which passes two values to hello_get.py program using GET method.

/cgi-bin/hello_get.py?first_name=ZARA&last_name=ALI

Below is **hello_get.py** script to handle input given by web browser. We are going to use **cgi** module, which makes it very easy to access passed information −

```
#!/usr/bin/python

# Import modules for CGI handling

importcgi,cgitb

# Create instance of FieldStorage

form=cgi.FieldStorage()

# Get data from fields

first_name=form.getvalue('first_name')

last_name=form.getvalue('last_name')

print"Content-type:text/html\r\n\r\n"

print"<html>"

print"<head>"
```

```
print"<title>Hello - Second CGI Program</title>"

print"</head>"

print"<body>"

print"<h2>Hello %s %s</h2>"%(first_name,last_name)

print"</body>"

print"</html>"
```

This would generate the following result −

Hello ZARA ALI

Simple FORM Example:GET Method

This example passes two values using HTML FORM and submit button. We use same CGI script hello_get.py to handle this input.

```
<formaction="/cgi-bin/hello_get.py"method="get">

First Name: <inputtype="text"name="first_name"><br/>

Last Name: <inputtype="text"name="last_name"/>

<inputtype="submit"value="Submit"/>

</form>
```

Here is the actual output of the above form, you enter First and Last Name and then click submit button to see the result.

First Name:

Last Name: Submit

Passing Information Using POST Method

A generally more reliable method of passing information to a CGI program is the POST method. This packages the information in exactly the same way as GET methods, but instead of sending it as a text string after a ?in the URL it sends it as a separate message. This message comes into the CGI script in the form of the standard input.

Below is same hello_get.py script which handles GET as well as POST method.

```
#!/usr/bin/python



# Import modules for CGI handling

importcgi,cgitb

# Create instance of FieldStorage

form=cgi.FieldStorage()

# Get data from fields
```

```
first_name=form.getvalue('first_name')

last_name=form.getvalue('last_name')

print"Content-type:text/html\r\n\r\n"

print"<html>"

print"<head>"

print"<title>Hello - Second CGI Program</title>"

print"</head>"

print"<body>"

print"<h2>Hello %s %s</h2>"%(first_name,last_name)

print"</body>"

print"</html>"
```

Let us take again same example as above which passes two values using HTML FORM and submit button. We use same CGI script hello_get.py to handle this input.

```
<formaction="/cgi-bin/hello_get.py"method="post">

First Name: <inputtype="text"name="first_name"><br/>

Last Name: <inputtype="text"name="last_name"/>

<inputtype="submit"value="Submit"/>

</form>
```

Here is the actual output of the above form. You enter First and Last Name and then click submit button to see the result.

First Name:

Last Name: Submit

Passing Checkbox Data to CGI Program

Checkboxes are used when more than one option is required to be selected.
Here is example HTML code for a form with two checkboxes −

```
<formaction="/cgi-bin/checkbox.cgi"method="POST"target="_blank">

<inputtype="checkbox"name="maths"value="on"/>Maths

<inputtype="checkbox"name="physics"value="on"/> Physics

<inputtype="submit"value="Select Subject"/>

</form>
```

The result of this code is the following form −

Maths Physics Select Subject

Below is checkbox.cgi script to handle input given by web browser for checkbox button.

```
#!/usr/bin/python

# Import modules for CGI handling

importcgi,cgitb

# Create instance of FieldStorage

form=cgi.FieldStorage()

# Get data from fields

ifform.getvalue('maths'):

math_flag="ON"

else:

math_flag="OFF"

ifform.getvalue('physics'):

physics_flag="ON"

else:

physics_flag="OFF"

print"Content-type:text/html\r\n\r\n"

print"<html>"

print"<head>"

print"<title>Checkbox - Third CGI Program</title>"

print"</head>"

print"<body>"

print"<h2>CheckBoxMaths is : %s</h2>"%math_flag

print"<h2>CheckBox Physics is : %s</h2>"%physics_flag

print"</body>"

print"</html>"
```

Passing Radio Button Data to CGI Program

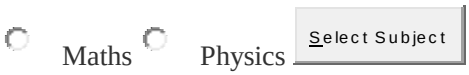
Radio Buttons are used when only one option is required to be selected.

Here is example HTML code for a form with two radio buttons −

```
<formaction="/cgi-bin/radiobutton.py"method="post"target="_blank">

<inputtype="radio"name="subject"value="maths"/>Maths

<inputtype="radio"name="subject"value="physics"/> Physics

<inputtype="submit"value="Select Subject"/>
```

```
</form>
```

The result of this code is the following form −



Below is radiobutton.py script to handle input given by web browser for radio button −

```
#!/usr/bin/python

# Import modules for CGI handling

importcgi,cgitb

# Create instance of FieldStorage

form=cgi.FieldStorage()

# Get data from fields

ifform.getvalue('subject'):

subject=form.getvalue('subject')

else:

subject="Not set"

print"Content-type:text/html\r\n\r\n"

print"<html>"

print"<head>"

print"<title>Radio - Fourth CGI Program</title>"

print"</head>"

print"<body>"

print"<h2> Selected Subject is %s</h2>"% subject

print"</body>"

print"</html>"
```

Passing Text Area Data to CGI Program

TEXTAREA element is used when multiline text has to be passed to the CGI Program.

Here is example HTML code for a form with a TEXTAREA box −

```
<formaction="/cgi-bin/textarea.py"method="post"target="_blank">

<textareaname="textcontent"cols="40"rows="4">

Type your text here...

</textarea>

<inputtype="submit"value="Submit"/>
```

```
</form>
```

The result of this code is the following form −

Below is textarea.cgi script to handle input given by web browser −

```
#!/usr/bin/python

# Import modules for CGI handling

importcgi,cgitb

# Create instance of FieldStorage

form=cgi.FieldStorage()

# Get data from fields

ifform.getvalue('textcontent'):

text_content=form.getvalue('textcontent')

else:

text_content="Not entered"

print"Content-type:text/html\r\n\r\n"

print"<html>"

print"<head>";

print"<title>Text Area - Fifth CGI Program</title>"

print"</head>"

print"<body>"

print"<h2> Entered Text Content is %s</h2>"%text_content

print"</body>"
```

Passing Drop Down Box Data to CGI Program

Drop Down Box is used when we have many options available but only one or two will be selected.
Here is example HTML code for a form with one drop down box −

```
<formaction="/cgi-bin/dropdown.py"method="post"target="_blank">

<selectname="dropdown">

<optionvalue="Maths"selected>Maths</option>

<optionvalue="Physics">Physics</option>

</select>
```

```
<inputtype="submit"value="Submit"/>

</form>
```

The result of this code is the following form −

Below is dropdown.py script to handle input given by web browser.

```
#!/usr/bin/python

# Import modules for CGI handling

importcgi,cgitb

# Create instance of FieldStorage

form=cgi.FieldStorage()

# Get data from fields

ifform.getvalue('dropdown'):

subject=form.getvalue('dropdown')

else:

subject="Not entered"

print"Content-type:text/html\r\n\r\n"

print"<html>"

print"<head>"

print"<title>Dropdown Box - Sixth CGI Program</title>"

print"</head>"

print"<body>"

print"<h2> Selected Subject is %s</h2>"% subject

print"</body>"

print"</html>"
```

Using Cookies in CGI

HTTP protocol is a stateless protocol. For a commercial website, it is required to maintain session information among different pages. For example, one user registration ends after completing many pages. How to maintain user's session information across all the web pages?

In many situations, using cookies is the most efficient method of remembering and tracking preferences, purchases, commissions, and other information required for better visitor experience or site statistics.

How It Works?

Your server sends some data to the visitor's browser in the form of a cookie. The browser may accept the cookie. If it does, it is stored as a plain text record on the visitor's hard drive. Now, when the visitor arrives

at another page on your site, the cookie is available for retrieval. Once retrieved, your server knows/remembers what was stored.

Cookies are a plain text data record of 5 variable-length fields −

- **Expires** − The date the cookie will expire. If this is blank, the cookie will expire when the visitor quits the browser.
- **Domain** − The domain name of your site.
- **Path** − The path to the directory or web page that sets the cookie. This may be blank if you want to retrieve the cookie from any directory or page.
- **Secure** − If this field contains the word "secure", then the cookie may only be retrieved with a secure server. If this field is blank, no such restriction exists.
- **Name=Value** − Cookies are set and retrieved in the form of key and value pairs.

Setting up Cookies

It is very easy to send cookies to browser. These cookies are sent along with HTTP Header before to Content-type field. Assuming you want to set UserID and Password as cookies. Setting the cookies is done as follows −

```
#!/usr/bin/python

print"Set-Cookie:UserID = XYZ;\r\n"

print"Set-Cookie:Password = XYZ123;\r\n"

print"Set-Cookie:Expires = Tuesday, 31-Dec-2007 23:12:40 GMT";\r\n"

print "Set-Cookie:Domain= www.tutorialspoint.com;\r\n"

print "Set-Cookie:Path=/perl;\n"

print "Content-type:text/html\r\n\r\n"

.....Rest of the HTML Content....
```

From this example, you must have understood how to set cookies. We use **Set-Cookie** HTTP header to set cookies.

It is optional to set cookies attributes like Expires, Domain, and Path. It is notable that cookies are set before sending magic line **"Content-type:text/html\r\n\r\n**.

Retrieving Cookies

It is very easy to retrieve all the set cookies. Cookies are stored in CGI environment variable HTTP_COOKIE and they will have following form −

```
key1 = value1;key2 = value2;key3 = value3....
```

Here is an example of how to retrieve cookies.

```
#!/usr/bin/python

# Import modules for CGI handling

fromosimport environ

importcgi,cgitb

ifenviron.has_key('HTTP_COOKIE'):

for cookie in map(strip, split(environ['HTTP_COOKIE'],';')):
```

```
(key, value )=split(cookie,'=');

if key =="UserID":

user_id= value

if key =="Password":

password= value

print"User ID  = %s"%user_id

print"Password = %s"% password
```

This produces the following result for the cookies set by above script −

```
User ID = XYZ
Password = XYZ123
```

File Upload Example

To upload a file, the HTML form must have the enctype attribute set to **multipart/form-data**. The input tag with the file type creates a "Browse" button.

```
<html>

<body>

<formenctype="multipart/form-data"

action="save_file.py"method="post">

<p>File: <inputtype="file"name="filename"/></p>

<p><inputtype="submit"value="Upload"/></p>

</form>

</body>

</html>
```

The result of this code is the following form −

File:

Upload

Above example has been disabled intentionally to save people uploading file on our server, but you can try above code with your server.

Here is the script **save_file.py** to handle file upload −

```
#!/usr/bin/python



importcgi,os

importcgitb;cgitb.enable()



form=cgi.FieldStorage()
```

```
# Get filename here.
fileitem= form['filename']

# Test if the file was uploaded
iffileitem.filename:
# strip leading path from file name to avoid
# directory traversal attacks
fn=os.path.basename(fileitem.filename)
open('/tmp/'+fn,'wb').write(fileitem.file.read())
message='The file "'+fn+'" was uploaded successfully'
else:
message='No file was uploaded'
print"""\
Content-Type: text/html\n
<html>
<body>
<p>%s</p>
</body>
</html>
"""%(message,)
```

If you run the above script on Unix/Linux, then you need to take care of replacing file separator as follows, otherwise on your windows machine above open() statement should work fine.

```
fn = os.path.basename(fileitem.filename.replace("\\", "/" ))
```

How To Raise a "File Download" Dialog Box?

Sometimes, it is desired that you want to give option where a user can click a link and it will pop up a "File Download" dialogue box to the user instead of displaying actual content. This is very easy and can be achieved through HTTP header. This HTTP header is be different from the header mentioned in previous section.

For example, if you want make a **FileName** file downloadable from a given link, then its syntax is as follows −

```
#!/usr/bin/python
# HTTP Header
print"Content-Type:application/octet-stream; name = \"FileName\"\r\n";
```

```
print"Content-Disposition: attachment; filename = \"FileName\"\r\n\n";

# Actual File Content will go here.

fo= open("foo.txt","rb")

str=fo.read();

printstr

# Close opend file

fo.close()
```

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.

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.

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.

Handling Errors

There are many sources of errors. A few examples are a syntax error in an executed SQL statement, a connection failure, or calling the fetch method for an already canceled or finished statement handle.

The DB API defines a number of errors that must exist in each database module. The following table lists these exceptions.

Sr.No.	Exception & Description
1	**Warning** Used for non-fatal issues. Must subclass StandardError.
2	**Error** Base class for errors. Must subclass StandardError.
3	**InterfaceError** Used for errors in the database module, not the database itself. Must subclass Error.
4	**DatabaseError** Used for errors in the database. Must subclass Error.
5	**DataError** Subclass of DatabaseError that refers to errors in the data.
6	**OperationalError** Subclass of DatabaseError that refers to errors such as the loss of a connection to the database. These errors are generally outside of the control of the Python scripter.
7	**IntegrityError** Subclass of DatabaseError for situations that would damage the relational integrity, such as uniqueness constraints or foreign keys.
8	**InternalError** Subclass of DatabaseError that refers to errors internal to the database module, such as a cursor no longer being active.
9	**ProgrammingError** Subclass of DatabaseError that refers to errors such as a bad table name and other things that can safely be blamed on you.
10	**NotSupportedError** Subclass of DatabaseError that refers to trying to call unsupported functionality.

Your Python scripts should handle these errors, but before using any of the above exceptions, make sure your MySQLdb has support for that exception. You can get more information about them by reading the DB API 2.0 specification.

Python MySQL Create Database

Creating a Database

To create a database in MySQL, use the "CREATE DATABASE" statement:

Example

create a database named "mydatabase":

```
import mysql.connector

mydb = mysql.connector.connect(  host="localhost",  user="yourusername",  passwd="yourpassword")
mycursor = mydb.cursor()

mycursor.execute("CREATE DATABASE  mydatabase")
```

Run example »

If the above code was executed with no errors, you have successfully created a database.

Check if Database Exists

You can check if a database exist by listing all databases in your system by using the "SHOW DATABASES" statement:

Example

Return a list of your system's databases:

```
import mysql.connector

mydb = mysql.connector.connect( host="localhost",  user="yourusername",   passwd="yourpassword" )
mycursor = mydb.cursor()
mycursor.execute("SHOW DATABASES")
for x in mycursor:
  print(x)
```

Run example »

Or you can try to access the database when making the connection:

Example

Try connecting to the database "mydatabase":

```
import mysql.connector

mydb = mysql.connector.connect( host="localhost",  user="yourusername",  passwd="yourpassword",
  database="mydatabase" )
```

Run example »

If the database does not exist, you will get an error.

sqlite3 — DB-API 2.0 interface for SQLite databases

UNIT IV

CGI (Common Gateway Interface) Programming

1) Explain the Concept CGI Programming.

Introduction:

The Common Gateway Interface, or CGI, is a set of standards that define how information is exchanged between the web server and a custom script.

What is CGI?

- The Common Gateway Interface, or CGI, is a standard for external gateway programs to interface with information servers such as HTTP servers.

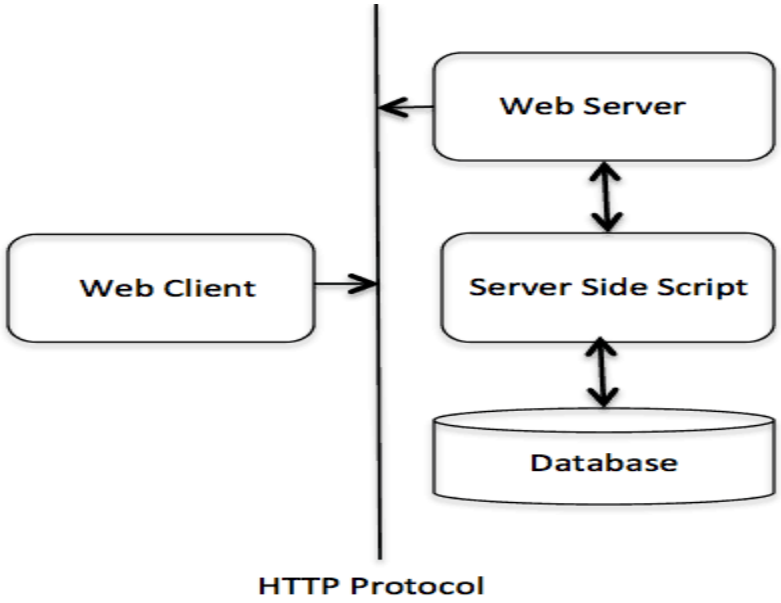
Web Browsing:

To understand the concept of CGI, let us see what happens when we click a hyper link to browse a particular web page or URL.

- Your browser contacts the HTTP web server and demands for the URL, i.e., filename.
- Web Server parses the URL and looks for the filename. If it finds that file then sends it back to the browser, otherwise sends an error message indicating that you requested a wrong file.
- Web browser takes response from web server and displays either the received file or error message.

However, it is possible to set up the HTTP server so that whenever a file in a certain directory is requested that file is not sent back; instead it is executed as a program, and whatever that program outputs is sent back for your browser to display. This function is called the Common Gateway Interface or CGI and the programs are called CGI scripts. These CGI programs can be a Python Script, PERL Script, Shell Script, C or C++ program, etc.

CGI Architecture Diagram



Working of CGI (Common Gateway Interface)

As shown in the above figure, a Web browser running on a client machine exchanges information with a Web server using the Hyper Text Transfer Protocol or HTTP. The Web server and the CGI program normally run on the same system, on which the web server resides, Depending on the type of request from the browser, the web server either provides a document from its own document directory or executes a CGI program.

The sequence of events for creating a dynamic HTML document on the fly through CGI scripting is as follows:

1. A client makes an HTTP request by means of a URL. This URL could be typed into the 'Location' window of a browser, be a hyperlink or be specified in the 'Action' attribute of an HTML <form> tag.

2. From the URL, the Web server determines that it should activate the gateway program listed in the URL and send any parameters passed via the URL to that program.

3. The gateway program processes the information and returns HTML text to the Web server. The server, in turn, adds a MIME header and returns the HTML text to the Web browser.

4. The Web browser displays the document received from the Web server.

First CGI Program

```
print("Content-type:text/html\r\n\r\n")
print('<html>')
print('<head>')
print('<title>Hello Word - First CGI Program</title>')
print('</head>')
print('<body>')
```

```
print('<h2>Hello Word! This is my first CGI program</h2>')
print('</body>')
print('</html>')
```

If you click hello.py, then this produces the following output −

Hello Word! This is my first CGI program

2) Explain CGI Environment Variables

All the CGI programs have access to the following environment variables. These variables play an important role while writing any CGI program.

S.No.	Variable Name & Description
1	**CONTENT_TYPE** The data type of the content. Used when the client is sending attached content to the server. For example, file upload.
2	**CONTENT_LENGTH** The length of the query information. It is available only for POST requests.
3	**HTTP_COOKIE** Returns the set cookies in the form of key & value pair.
4	**HTTP_USER_AGENT** The User-Agent request-header field contains information about the user agent originating the request. It is name of the web browser.
5	**PATH_INFO** The path for the CGI script.
6	**QUERY_STRING** The URL-encoded information that is sent with GET method request.
7	**REMOTE_ADDR** The IP address of the remote host making the request. This is useful logging or for authentication.
8	**REMOTE_HOST** The fully qualified name of the host making the request. If this information is not available, then REMOTE_ADDR can be used to get IR address.
9	**REQUEST_METHOD**

	The method used to make the request. The most common methods are GET and POST.
10	**SCRIPT_FILENAME** The full path to the CGI script.
11	**SCRIPT_NAME** The name of the CGI script.
12	**SERVER_NAME** The server's hostname or IP Address
13	**SERVER_SOFTWARE** The name and version of the software the server is running.

Python Programming.

SAQ's

UNIT – I

1. Briefly explain about Python programming.
2. Explain the Structure of Python program.
3. Write short notes on Input and Output Statements.
4. Write short notes on python numbers.
5. Discuss about Python Literals.
6. Explain Comments in python programming.

UNIT – II

1. Write a short note on Control flow statements.
2. Write a short note to demonstrate break, continue and pass statements.
3. Define Functions.
4. Explain Anonymous functions in python programming.
5. Explain call by value and call by reference.

UNIT – III

1. List the standard files available in python.
2. How to rename and delete files.
3. Write short notes on file positions.
4. Write the steps to import modules.
5. Briefly explain about command line arguments.
6. Define Packages, Modules and Files.
7. Write a program to demonstrate classes.
8. Write a short note on Inheritance.
9. Discuss about Directories with a syntax and example.
10. Define Namespaces.

UNIT – IV

1. Write a short on CGI in python programming (Introduction).
2. Write a sample program to demonstrate CGI.
3. Explain HTTP header in python CGI.
4. Write a short on Database access in python programming (Introduction).
5. Explain rollback and commit operations using python.

LAQ's

Unit – I

1. What is Python Programming? Explain the features of Python Programming.
2. Give the Steps for Installing Python.
3. Briefly explain about Strings in Python programming.
4. Define Operators and explain any five Operators with syntax and examples.
5. Explain briefly about Lists. Write about basic functions & methods used in lists.
6. Explain briefly about Tuples. Write about basic functions & methods used in Tuples.
7. Explain briefly about Dictionaries. Write about basic functions & methods used in Dictionaries.

Unit – II

1. Explain Conditional statements in python (if, if-else, elif, nested if).
2. Explain Looping statements in python (while, for & while – else, for - else).
3. Define Functions. Explain about Built-in functions with a syntax and example.
4. What is Function? Explain User defined functions of Python Programming.

Unit – III

1. Briefly Define File and explain File handling mechanism in python.
2. Explain briefly about Modules in python.
3. Explain briefly about Module built-in functions and packages.
4. Briefly discuss about advance python programming (Classes & Objects, Inheritance, and Regular Expressions).
5. Briefly explain about regular Expressions.

Unit – IV

1. Briefly explain about Python CGI Programming (Architecture, CGI Environment variables, GET and Post methods).
2. Explain the process to create a Simple Form using GET and POST methods.
3. Explain briefly about Database Connectivity (Establishing Connection, insert, delete, retrieve, commit, and rollback).