1. OVERVIEW

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.

History of Python

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

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

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

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

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.

Apart from the above-mentioned features, Python has a big list of good features, few are listed below:

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.

2. ENVIRONMENT

Python is available on a wide variety of platforms including Linux and Mac OS X. Let's understand how to set up our Python environment.

Local Environment Setup

Open a terminal window and type "python" to find out if it is already installed and which version is installed.

Unix (Solaris, Linux, FreeBSD, AIX, HP/UX, SunOS, IRIX, etc.)

Win 9x/NT/2000

Macintosh (Intel, PPC, 68K)

OS/2

DOS (multiple versions)

PalmOS

Nokia mobile phones

Windows CE

Acorn/RISC OS

BeOS

Amiga

VMS/OpenVMS

QNX

VxWorks

Psion

Python has also been ported to the Java and .NET virtual machines

Getting Python

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

You can download Python documentation from 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 **http://www.python.org/download/**.

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 **http://www.python.org/download/**

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.

There are three different ways to start Python:

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/Linuxor

python% script.py              # Unix/Linuxor 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.

Macintosh: The Macintosh version of Python along with the IDLE IDE is available from the main website, downloadable as either MacBinary or BinHex'd files.

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.

Note: All the examples given in subsequent chapters are executed with Python 2.4.3 version available on CentOS flavor of Linux.

We already have set up Python Programming environment online, so that you can execute all the available examples online at the same time when you are learning theory. Feel free to modify any example and execute it online.

3. BASIC SYNTAX

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

First Python Program

Let us execute programs in different modes of programming.

Interactive Mode Programming:

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

```
$ python

Python 2.4.3 (#1, Nov 11 2010, 13:34:43)

[GCC 4.1.2 20080704 (Red Hat 4.1.2-48)] 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!";
```

If you are running new version of Python, then you 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.

Python Keywords

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	exec	Not
Assert	finally	or
Break	for	pass
Class	from	print
Continue	global	raise
def	if	return
del	import	try
elif	in	while
else	is	with
except	lambda	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:

```
if True:

    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 = "Madisetti" # 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.

To accept User Input from Console:

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\nPress the enter key to exit.")
```

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 :

    suite elif expression :

    suite  else :     suite
```

Accessing Command-Line Arguments

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

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

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

sys.argv is the list of command-line arguments.

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

Here sys.argv[0] is the program i.e. script name.

Example

Consider the following script test.py:

```
#!/usr/bin/python

 import sys

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

Now run above script as follows:

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

This produces the following result:

```
Number of arguments: 4 arguments.
```

```
Argument List: ['test.py', 'arg1', 'arg2', 'arg3']
```

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

4. VARIABLE TYPES

Variables are nothing but reserved memory locations to store values. This means 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. 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:

```
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)

long (long integers, they can also be represented in octal and hexadecimal)

float (floating point real values)

complex (complex numbers)

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 x + yj, where x is the real part and b is the imaginary part of the complex number.

Python Strings

Strings in Python are identified as a contiguous set of characters represented in the quotation marks. Python allows for either pairs of single or double quotes. Subsets of strings can be taken using the slice operator ([ ] and [:]) with indexes starting at 0 in the beginning of the string and working their way from -1 at the end.

The plus (+) sign is the string concatenation operator and the asterisk (*) is the repetition operator. For example:

```
#!/usr/bin/python

 str = 'Hello World!' print str          # Prints complete string print str[0]
# Prints first character of the string print str[2:5]     # Prints characters
starting from 3rd to 5th print str[2:]     # Prints string starting from 3rd
character print str * 2      # Prints string two times print str + "TEST" #
Prints concatenated string
```

This will produce the following result:

```
Hello World! H llo llo World!

Hello World!Hello World!
```
Hello World!TEST

Python Lists

Lists are the most versatile of Python's compound data types. A list contains items separated by commas and enclosed within square brackets ([]). To some extent, lists are similar to arrays in C. One difference between them is that all the items belonging to a list can be of different data type.

The values stored in a list can be accessed using the slice operator ([ ] and [:]) with indexes starting at 0 in the beginning of the list and working their way to end -1. The plus (+) sign is the list concatenation operator, and the asterisk (*) is the repetition operator. For example:

```
#!/usr/bin/python

 list = [ 'abcd', 786 , 2.23, 'john', 70.2 ] tinylist = [123, 'john']
```

```
 print list           # Prints complete list print list[0]       # Prints first
element of the list print list[1:3]     # Prints elements starting from 2nd
till 3rd  print list[2:]      # Prints elements starting from 3rd element print
tinylist * 2 # Prints list two times print list + tinylist # Prints concatenated
lists
```

This produces the following result:

```
['abcd', 786, 2.23, 'john', 70.200000000000003] abcd

[786, 2.23]

[2.23, 'john', 70.200000000000003]

[123, 'john', 123, 'john']
```

['abcd', 786, 2.23, 'john', 70.200000000000003, 123, 'john']

Python Tuples

A tuple is another sequence data type that is similar to the list. A tuple consists of a number of values separated by commas. Unlike lists, however, tuples are enclosed within parentheses.

The main differences between lists and tuples are: Lists are enclosed in brackets ([ ]) and their elements and size can be changed, while tuples are enclosed in parentheses (( )) and cannot be updated. Tuples can be thought of as **readonly** lists. For example:

```
#!/usr/bin/python

 tuple = ( 'abcd', 786 , 2.23, 'john', 70.2  ) tinytuple = (123, 'john')

 print tuple            # Prints complete list print tuple[0]        # Prints
first element of the list print tuple[1:3]     # Prints elements starting from
2nd till 3rd  print tuple[2:]       # Prints elements starting from 3rd element
print tinytuple * 2   # Prints list two times print tuple + tinytuple # Prints
concatenated lists
```

This produces the following result:

```
('abcd', 786, 2.23, 'john', 70.200000000000003) abcd
```

```
(786, 2.23)

(2.23, 'john', 70.200000000000003)

(123, 'john', 123, 'john')

('abcd', 786, 2.23, 'john', 70.200000000000003, 123, 'john')
```

The following code is invalid with tuple, because we attempted to update a tuple, which is not allowed. Similar case is possible with lists:

```
#!/usr/bin/python

 tuple = ( 'abcd', 786 , 2.23, 'john', 70.2  ) list = [ 'abcd', 786 , 2.23,
'john', 70.2  ] tuple[2] = 1000    # Invalid syntax with tuple list[2] = 1000
# Valid syntax with list
```

Python Dictionary

Python's dictionaries are kind of hash table type. They work like associative arrays or hashes found in Perl and consist of key-value pairs. A dictionary key can be almost any Python type, but are usually numbers or strings. Values, on the other hand, can be any arbitrary Python object.

Dictionaries are enclosed by curly braces ({ }) and values can be assigned and accessed using square braces ([]). For example:

```
#!/usr/bin/python

 dict = {} dict['one'] = "This is one" dict[2]     = "This is two" tinydict =
{'name': 'john','code':6734, 'dept': 'sales'} print dict['one']       # Prints
value for 'one' key print dict[2]              # Prints value for 2 key print
tinydict          # Prints complete dictionary print tinydict.keys()  # Prints
all the keys print tinydict.values() # Prints all the values
```

This produces the following result:

```
This is one

This is two

{'dept': 'sales', 'code': 6734, 'name': 'john'}

['dept', 'code', 'name']

['sales', 6734, 'john']
```

Dictionaries have no concept of order among elements. It is incorrect to say that the elements are "out of order"; they are simply unordered.

Data Type Conversion

Sometimes, you may need to perform conversions between the built-in types. To convert between types, you simply use the type name as a function.

There are several built-in functions to perform conversion from one data type to another. These functions return a new object representing the converted value.

Function	Description
int(x [,base])	Converts x to an integer. base specifies the base if x is a string.
long(x [,base])	Converts x to a long integer. base specifies the base if x is a string.
float(x)	Converts x to a floating-point number.
complex(real [,imag])	Creates a complex number.
str(x)	Converts object x to a string representation.
repr(x)	Converts object x to an expression string.

eval(str)	Evaluates a string and returns an object.
tuple(s)	Converts s to a tuple.
list(s)	Converts s to a list.
set(s)	Converts s to a set.
dict(d)	Creates a dictionary. d must be a sequence of (key,value) tuples.
frozenset(s)	Converts s to a frozen set.
chr(x)	Converts an integer to a character.
unichr(x)	Converts an integer to a Unicode character.
ord(x)	Converts a single character to its integer value.
hex(x)	Converts an integer to a hexadecimal string.
oct(x)	Converts an integer to an octal string.

5. 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 Operators

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.	9//2 = 4 and 9.0//2.0 = 4.0

Example

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

```
#!/usr/bin/python

 a = 21 b = 10 c = 0  c = a + b print "Line 1 - Value of c is ", c c = a - b
print "Line 2 - Value of c is ", c  c = a * b print "Line 3 - Value of c is ",
c  c = a / b print "Line 4 - Value of c is ", c

 c = a % b
```

```
print "Line 5 - Value of c is ", c



a = 2

b = 3

c = a**b

print "Line 6 - Value of c is ", c



a = 10

b = 5

c = a//b

print "Line 7 - Value of c is ", c
```

31 When you execute the above program, it produces the following result:

```
Line 1 - Value of c is 31

Line 2 - Value of c is 11

Line 3 - Value of c is 210

Line 4 - Value of c is 2

Line 5 - Value of c is 1

Line 6 - Value of c is 8

Line 7 - Value of c is 2
```

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.

Example

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

```
#!/usr/bin/python

 a = 21 b = 10 c = 0



if ( a == b ):
```

Python

```
   print "Line 1 - a is equal to b" else:

   print "Line 1 - a is not equal to b"

 if ( a != b ):

   print "Line 2 - a is not equal to b" else:

   print "Line 2 - a is equal to b"

 if ( a <> b ):

   print "Line 3 - a is not equal to b" else:

   print "Line 3 - a is equal to b"

 if ( a < b ):

   print "Line 4 - a is less than b"  else:

   print "Line 4 - a is not less than b"

 if ( a > b ):

   print "Line 5 - a is greater than b" else:

   print "Line 5 - a is not greater than b"

 a = 5;
```

```
b = 20; if ( a <= b ):

   print "Line 6 - a is either less than or equal to  b" else:

   print "Line 6 - a is neither less than nor equal to  b"

 if ( b >= a ):

   print "Line 7 - b is either greater than  or equal to b" else:

   print "Line 7 - b is neither greater than  nor equal to b"
```

When you execute the above program it produces the following result:

```
Line 1 - a is not equal to b

Line 2 - a is not equal to b
```

```
Line 3 - a is not equal to b

Line 4 - a is not less than b

Line 5 - a is greater than b

Line 6 - a is either less than or equal to b

Line 7 - b is either greater than or equal to b
```

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 / 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

Example

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

```
#!/usr/bin/python

 a = 21 b = 10 c = 0  c = a + b
```

```
print "Line 1 - Value of c is ", c

 c += a print "Line 2 - Value of c is ", c

 c *= a print "Line 3 - Value of c is ", c

 c /= a  print "Line 4 - Value of c is ", c

 c  = 2 c %= a print "Line 5 - Value of c is ", c

 c **= a print "Line 6 - Value of c is ", c

 c //= a print "Line 7 - Value of c is ", c
```

When you execute the above program, it produces the following result:

```
Line 1 - Value of c is 31

Line 2 - Value of c is 52

Line 3 - Value of c is 1092

Line 4 - Value of c is 52

Line 5 - Value of c is 2
```

```
Line 6 - Value of c is 2097152

Line 7 - Value of c is 99864
```

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) = 12 (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)

Example

```
#!/usr/bin/python

a = 60            # 60 = 0011 1100
b = 13            # 13 = 0000 1101
c = 0   c = a & b; # 12 = 0000 1100
print "Line 1 - Value of c is ", c
c = a | b;        # 61 = 0011 1101
print "Line 2 - Value of c is ", c
c = a ^ b;        # 49 = 0011 0001
print "Line 3 - Value of c is ", c
c = ~a;           # -61 = 1100 0011
print "Line 4 - Value of c is ", c
c = a << 2;       # 240 = 1111 0000
print "Line 5 - Value of c is ", c
c = a >> 2        # 15 = 0000 1111
print "Line 6 - Value of c is ", c
```

When you execute the above program it produces the following result:

```
Line 1 - Value of c is 12

Line 2 - Value of c is 61

Line 3 - Value of c is 49

Line 4 - Value of c is -61

Line 5 - Value of c is 240

Line 6 - Value of c is 15
```

Python Logical Operators

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

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

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.

Example

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

```
    print "Line 2 - b is available in the given list"



a = 2

if ( a in list ):

    print "Line 3 - a is available in the given list"

else:

    print "Line 3 - a is not available in the given list"
```

```
 a = 10 b = 20 list = [1, 2, 3, 4, 5 ];

 if ( a in list ):

   print "Line 1 - a is available in the given list" else:

   print "Line 1 - a is not available in the given list"

 if ( b not in list ):

   print "Line 2 - b is not available in the given list" else:
```

When you execute the above program it produces the following result:

```
Line 1 - a is not available in the given list

Line 2 - b is not available in the given list

Line 3 - a is available in the given list
```

Python Identity Operators

Identity operators compare the memory locations of two objects. There are two Identity operators as 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).

Example

```
#!/usr/bin/python

a = 20 b = 20  if ( a is b ):

   print "Line 1 - a and b have same identity" else:

   print "Line 1 - a and b do not have same identity"

 if ( id(a) == id(b) ):

   print "Line 2 - a and b have same identity" else:

   print "Line 2 - a and b do not have same identity"

 b = 30 if ( a is b ):

   print "Line 3 - a and b have same identity" else:

   print "Line 3 - a and b do not have same identity"

 if ( a is not b ):

   print "Line 4 - a and b do not have same identity" else:

   print "Line 4 - a and b have same identity"
```

When you execute the above program it produces the following result:

```
Line 1 - a and b have same identity

Line 2 - a and b have same identity
```

Python Operators Precedence

```
Line 3 - a and b do not have same identity

Line 4 - a and b do not have same identity
```

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

Operator	Description
**	Exponentiation (raise to the power)
~ + -	Ccomplement, unary plus and minus (method names for the last two are +@ and -@)
* / % //	Multiply, divide, modulo and floor division
+ -	Addition and subtraction
>> <<	Right and left bitwise shift
&	Bitwise 'AND'
^ |	Bitwise exclusive `OR' and regular `OR'
<= < > >=	Comparison operators
<> == !=	Equality operators
= %= /= //= -= += *= **=	Assignment operators
is is not	Identity operators

in not in	Membership operators
not or and	Logical operators

Operator precedence affects how an expression is evaluated.

For example, x = 7 + 3 * 2; here, x is assigned 13, not 20 because operator * has higher precedence than +, so it first multiplies 3*2 and then adds into 7.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom.

Example

```
#!/usr/bin/python

 a = 20 b = 10 c = 15 d = 5 e = 0  e = (a + b) * c / d        #( 30 * 15 ) / 5
print "Value of (a + b) * c / d is ",  e

 e = ((a + b) * c) / d     # (30 * 15 ) / 5 print "Value of ((a + b) * c) / d
is ",  e

 e = (a + b) * (c / d);     # (30) * (15/5) print "Value of (a + b) * (c / d)
is ",  e

 e = a + (b * c) / d;      #  20 + (150/5) print "Value of a + (b * c) / d is
",  e
```

When you execute the above program, it produces the following result:

```
Value of (a + b) * c / d is 90

Value of ((a + b) * c) / d is 90

Value of (a + b) * (c / d) is 90

Value of a + (b * c) / d is 50
```

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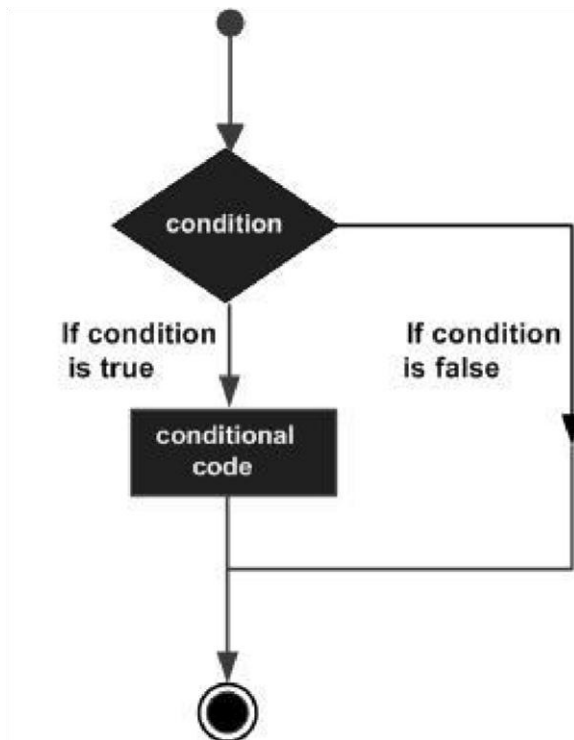
Python programming language assumes any **non-zero** and **non-null** values as TRUE, and if it is either **zero** or **null**, then it is assumed as FALSE value.

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

Statement	Description

| if statements	**if statement** consists of a boolean expression followed by one or more statements.
if...else statements	**if statement** can be followed by an optional **else statement**, which executes when the boolean expression is FALSE.
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:

If Statement

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
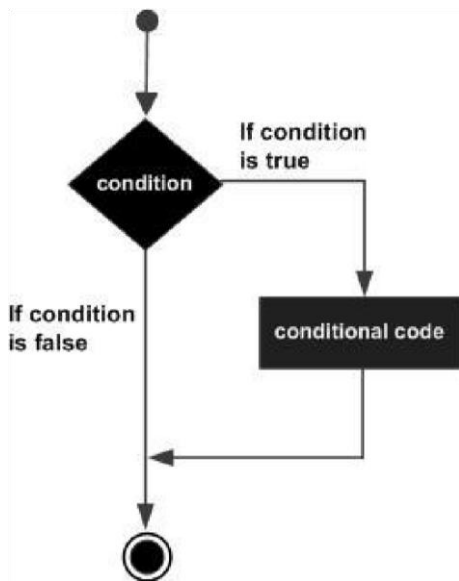
Example

```
#!/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 Statement

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**.
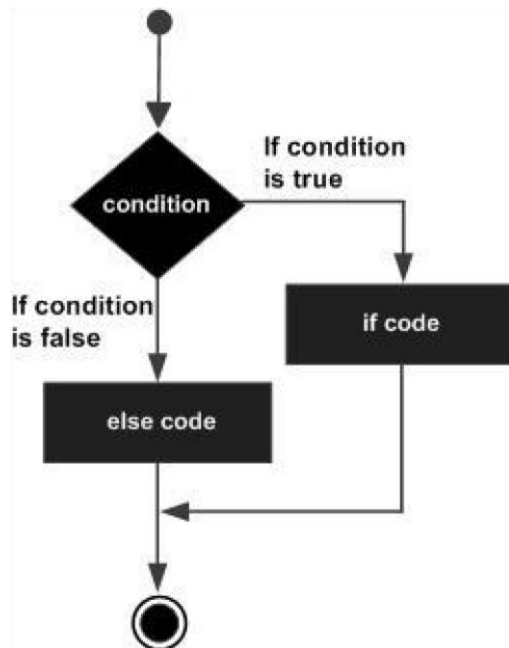
Syntax

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:

```
- Got a true expression value

100

- 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 if var == 200:    print "1 - Got a true expression value"    print
var elif var == 150:

   print "2 - Got a true expression value"    print var elif var == 100:

   print "3 - Got a true expression value"    print var else:

   print "4 - Got a false expression value"    print var

 print "Good bye!"
```

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

```
3 - Got a true expression value

100
```
Good bye!

7. LOOPS

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!
```

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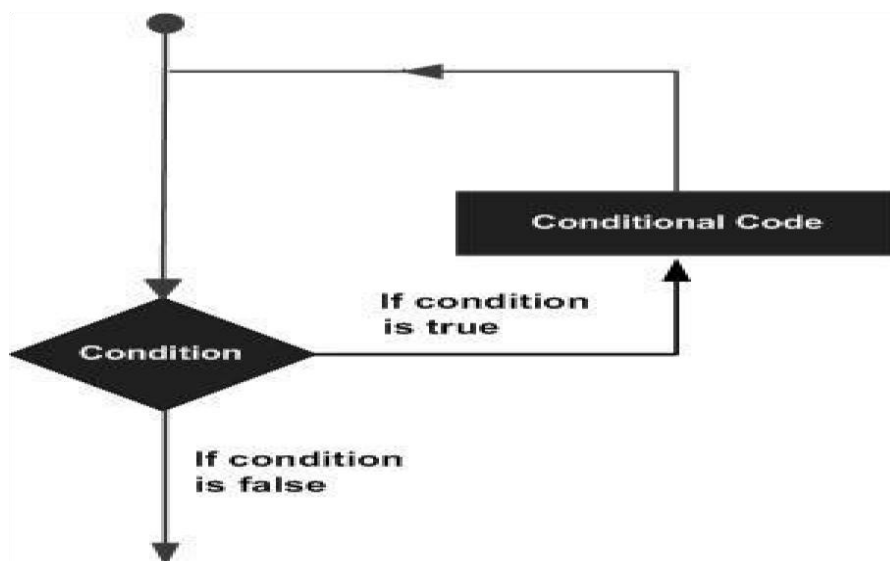
Python programming language provides following types of loops to handle looping requirements.

Loop Type	Description
while loop	Repeats a statement or group of statements while a given condition is TRUE. It tests the condition before executing the loop body.
for loop	Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.
nested loops	You can use one or more loop inside any another while, for or do..while loop.

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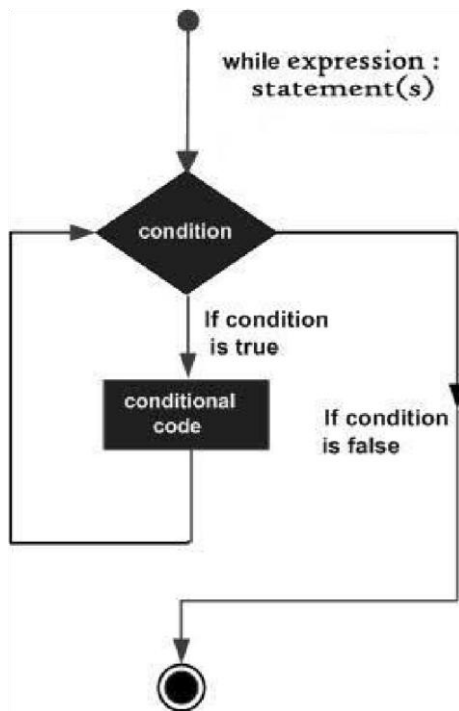
Here, key point of the *while* loop is that the loop might not ever run. When the condition is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

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 while var == 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:

```
is less than 5

is less than 5
```

```
is less than 5

is less than 5

is less than 5

is not less than 5
```

For Loop

```
for iterating_var in sequence:

    statements(s)
```

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

Syntax

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
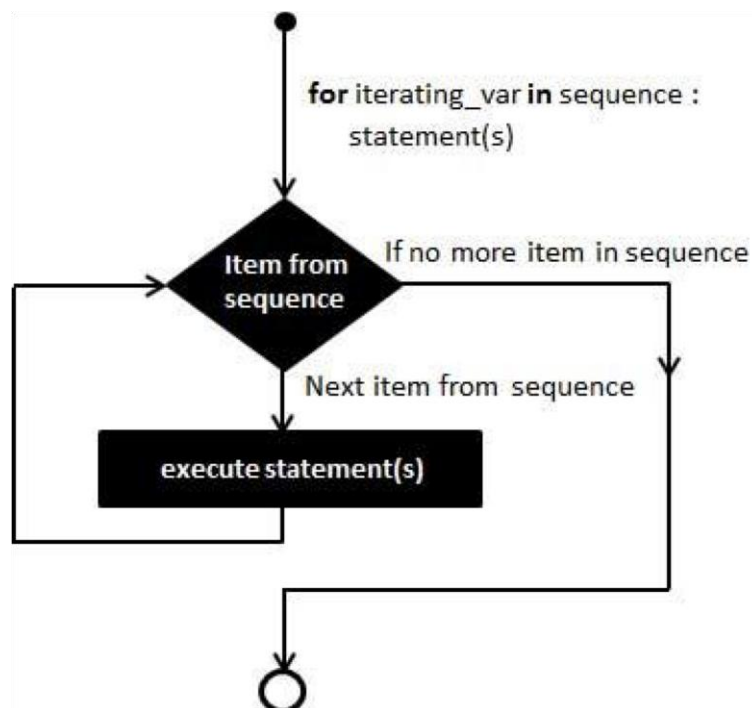
Example

```
#!/usr/bin/python

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

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

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.

```
#!/usr/bin/python

 for num in range(10,20):  #to iterate between 10 to 20    for i in range(2,num):
#to iterate on the factors of the number        if num%i == 0:        #to determine
the first factor              j=num/i              #to calculate the second factor
print '%d equals %d * %d' % (num,i,j)
```

         break #to move to the next number, the #first FOR   else:
else part of the loop   print num, 'is a prime number'

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

```
10 equals 2 * 5
```

```
11 is a prime number

12 equals 2 * 6

13 is a prime number

14 equals 2 * 7

15 equals 3 * 5

16 equals 2 * 8

17 is a prime number

18 equals 2 * 9

19 is a prime number
```

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.

Control Statement	Description
break statement	Terminates the loop statement and transfers execution to the statement immediately following the loop.
continue statement	Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.
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:

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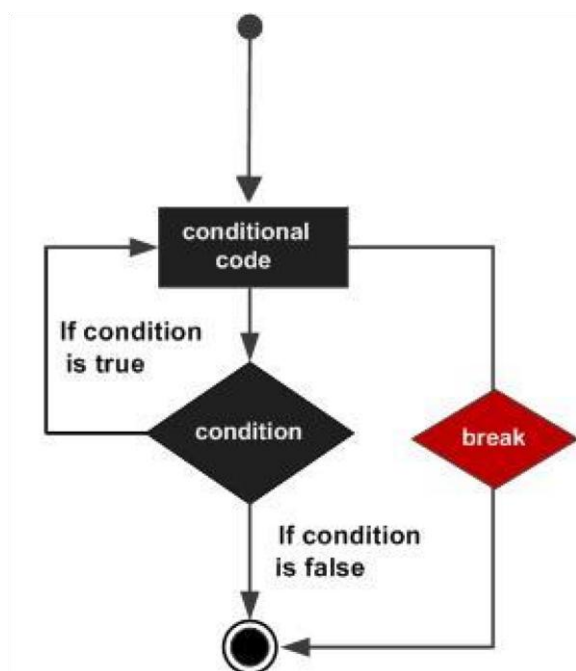
If you are using nested loops, the break statement stops the execution of the innermost loop and start executing the next line of code after the block.

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 while var > 0:

   print 'Current variable value :', var    var = var -1    if var == 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!
```
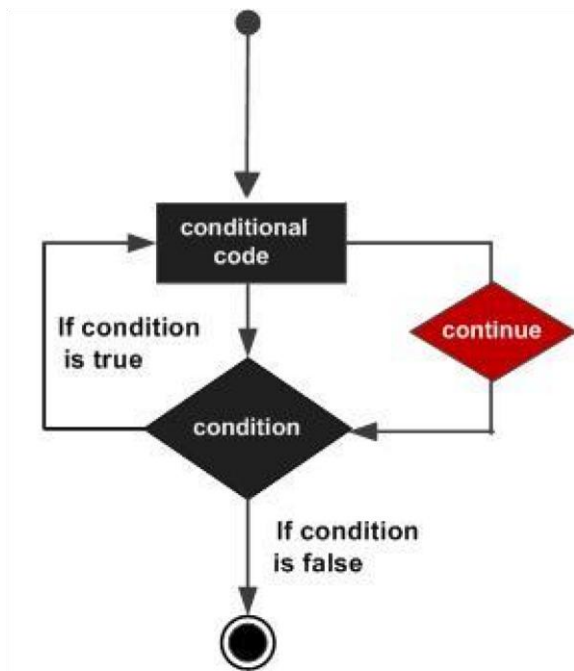
Continue Statement

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 while var > 0:

   var = var -1    if var == 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!
```

8. 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:

```
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.

Function	Returns (description)
abs(x)	The absolute value of x: the (positive) distance between x and zero.
ceil(x)	The ceiling of x: the smallest integer not less than x
cmp(x, y)	-1 if x < y, 0 if x == y, or 1 if x > y
exp(x)	The exponential of x: e^x
fabs(x)	The absolute value of x.
floor(x)	The floor of x: the largest integer not greater than x
log(x)	The natural logarithm of x, for x> 0
log10(x)	The base-10 logarithm of x for x> 0.
max(x1, x2,...)	The largest of its arguments: the value closest to positive infinity
min(x1, x2,...)	The smallest of its arguments: the value closest to negative infinity

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.
pow(x, y)	The value of x**y.
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.
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.

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

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.

Function	Description
acos(x)	Return the arc cosine of x, in radians.
asin(x)	Return the arc sine of x, in radians.
atan(x)	Return the arc tangent of x, in radians.
atan2(y, x)	Return atan(y / x), in radians.
cos(x)	Return the cosine of x radians.
hypot(x, y)	Return the Euclidean norm, sqrt(x*x + y*y).
sin(x)	Return the sine of x radians.
tan(x)	Return the tangent of x radians.
degrees(x)	Converts angle x from radians to degrees.
radians(x)	Converts angle x from degrees to radians.

Mathematical Constants

The module also defines two mathematical constants:

Constants	Description
pi	The mathematical constant pi.
e	The mathematical constant e.

9. 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] print "var2[1:5]: ", var2[1:5]
```

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*2 will give -HelloHello
[]	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.

Let us study them in detail:

1.   capitalize() Method

It returns a copy of the string with only its first character capitalized.

Syntax

```
str.capitalize()
```

Parameters

NA

Return Value

string

Example

```
#!/usr/bin/python

 str = "this is string example....wow!!!";

 print "str.capitalize() : ", str.capitalize()
```

Result

```
str.capitalize() :  This is string example....wow!!!
```

2.   center(width, fillchar) Method

The method center() returns centered in a string of length width. Padding is done using the specified fillchar. Default filler is a space.

Syntax

```
str.center(width[, fillchar])
```

Parameters

width -- This is the total width of the string.

fillchar -- This is the filler character.

Return Value

This method returns centered in a string of length width.

Example

```
#!/usr/bin/python

 str = "this is string example....wow!!!";



```
```
print "str.center(40, 'a') : ", str.center(40, 'a')
```

Result

```
str.center(40, 'a') :  aaaathis is string example....wow!!!aaaa
```

3. count(str, beg= 0,end=len(string)) Method

The method **count()** returns the number of occurrences of substring sub in the range [start, end]. Optional arguments start and end are interpreted as in slice notation.

Syntax

```
str.count(sub, start= 0,end=len(string))
```

Parameters

sub -- This is the substring to be searched.

start -- Search starts from this index. First character starts from 0 index. By default search starts from 0 index.

end -- Search ends from this index. First character starts from 0 index. By default search ends at the last index.

Return Value

Centered in a string of length width.

Example

```
#!/usr/bin/python

 str = "this is string example....wow!!!";

 sub = "i"; print "str.count(sub, 4, 40) : ", str.count(sub, 4, 40) sub =
"wow"; print "str.count(sub) : ", str.count(sub)
```

```
str.count(sub, 4, 40) :  2

str.count(sub, 4, 40) :  1
```

Result

4. decode(encoding='UTF-8',errors='strict') Method

The method **decode()** decodes the string using the codec registered for *encoding*. It defaults to the default string encoding.

Syntax

```
str.decode(encoding='UTF-8',errors='strict')
```

Parameters

encoding -- This is the encodings to be used. For a list of all encoding schemes please visit: **Standard Encodings.**

errors -- This may be given to set a different error handling scheme. The default for errors is 'strict', meaning that encoding errors raise a UnicodeError. Other possible values are 'ignore', 'replace', 'xmlcharrefreplace', 'backslashreplace' and any other name registered via codecs.register_error().

Return Value

Decoded string.

Example

```
#!/usr/bin/python

 str = "this is string example....wow!!!"; str = str.encode('base64','strict');

 print  "Encoded  String:  "  +  str;  print  "Decoded  String:  "  +
str.decode('base64','strict')
```

Result

```
Encoded String: dGhpcyBpcyBzdHJpbmcgZXhhbXBsZS4uLi53b3chISE=   Decoded String:
this is string example....wow!!!
```

5. encode(encoding='UTF-8',errors='strict') Method

The method **encode()** returns an encoded version of the string. Default encoding is the current default string encoding. The errors may be given to set a different error handling scheme. **Syntax**

```
str.encode(encoding='UTF-8',errors='strict')
```

Parameters

encoding -- This is the encodings to be used. For a list of all encoding schemes please visit **Standard Encodings.**

errors -- This may be given to set a different error handling scheme. The default for errors is 'strict', meaning that encoding errors raise a UnicodeError. Other possible values are 'ignore', 'replace', 'xmlcharrefreplace', 'backslashreplace' and any other name registered via codecs.register_error().

Return Value

Encoded string.

Example

```
#!/usr/bin/python str = "this is string example....wow!!!"; print "Encoded
String: " + str.encode('base64','strict')
```

Result

```
Encoded String: dGhpcyBpcyBzdHJpbmcgZXhhbXBsZS4uLi53b3chISE=
```

6. endswith(suffix, beg=0, end=len(string)) Method

It returns True if the string ends with the specified *suffix*, otherwise return False optionally restricting the matching with the given indices *start* and *end*.

Syntax

```
str.endswith(suffix[, start[, end]])
```

Parameters

suffix -- This could be a string or could also be a tuple of suffixes to look for.

start -- The slice begins from here.

end -- The slice ends here.

Return Value

TRUE if the string ends with the specified suffix, otherwise FALSE.

Example

```
#!/usr/bin/python

 str = "this is string example....wow!!!";

 suffix = "wow!!!"; print str.endswith(suffix); print str.endswith(suffix,20);

 suffix = "is"; print str.endswith(suffix, 2, 4); print str.endswith(suffix,
2, 6);
```

Result

```
True
```

```
True
```

```
True
```

```
False
```

7. expandtabs(tabsize=8)

It returns a copy of the string in which tab characters ie. '\t' are expanded using spaces, optionally using the given tabsize (default 8).

Syntax

```
str.expandtabs(tabsize=8)
```

Parameters

tabsize -- This specifies the number of characters to be replaced for a tab character '\t'.

Return Value

This method returns a copy of the string in which tab characters i.e., '\t' have been expanded using spaces.

Example

```
#!/usr/bin/python

 str = "this is\tstring example....wow!!!";

 print "Original string: " + str; print "Defualt exapanded tab: " +
str.expandtabs(); print "Double exapanded tab: " +  str.expandtabs(16);
```

Result

Original string: this is   string example....wow!!!

Defualt exapanded tab: this is string example....wow!!!

Double exapanded tab: this is   string example....wow!!!

8. find(str, beg=0 end=len(string))

It determines if string *str* occurs in string, or in a substring of string if starting index *beg* and ending index *end* are given.

Syntax

```
str.find(str, beg=0 end=len(string))
```

Parameters

str -- This specifies the string to be searched.

beg -- This is the starting index, by default its 0.

end -- This is the ending index, by default its equal to the lenght of the string.

Return Value

Index if found and -1 otherwise.

Example

```
The following example shows the usage of find() method.

#!/usr/bin/python

 str1 = "this is string example....wow!!!"; str2 = "exam";

 print str1.find(str2); print str1.find(str2, 10); print str1.find(str2, 40);
```

Result

15

15

-1

11.   9. index(str, beg=0, end=len(string))

It determines if string *str* occurs in string or in a substring of string if starting index *beg* and ending index *end* are given. This method is same as find(), but raises an exception if sub is not found.

Syntax

```
str.index(str, beg=0 end=len(string))
```

Parameters

str -- This specifies the string to be searched.

beg -- This is the starting index, by default its 0.

end -- This is the ending index, by default its equal to the length of the string.

Return Value

Index if found otherwise raises an exception if str is not found.

Example

```
#!/usr/bin/python

 str1 = "this is string example....wow!!!"; str2 = "exam";

 print str1.index(str2); print str1.index(str2, 10); print str1.index(str2,
40);
```

Result

```
15

15

Traceback (most recent call last):   File "test.py", line 8, in     print
str1.index(str2, 40);

ValueError: substring not found
```

```
shell returned 1
```

12.   10. isalnum() Method

It checks whether the string consists of alphanumeric characters. **Syntax**

```
str.isa1num()
```

Parameters

NA

Return Value

TRUE if all characters in the string are alphanumeric and there is at least one character, FASLE otherwise.

Example

```
#!/usr/bin/python

 str = "this2009";  # No space in this string print str.isalnum();

 str = "this is string example....wow!!!"; print str.isalnum();
```

Result

```
True
```

```
False
```

13.   11. isalpha()

Description

The method **isalpha()** checks whether the string consists of alphabetic characters only.

Syntax

Following is the syntax for **islpha()** method:

```
str.isalpha()
```

Parameters

☐ **NA**

Return Value

This method returns true if all characters in the string are alphabetic and there is at least one character, false otherwise.

Example

The following example shows the usage of isalpha() method.

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

 str = "this";  # No space & digit in this string print str.isalpha();

 str = "this is string example....wow!!!"; print str.isalpha();
```

When we run above program, it produces following result:

```
True
```

```
False
```

14.   12. isdigit()

Description

The method **isdigit()** checks whether the string consists of digits only.

Syntax

Following is the syntax for **isdigit()** method:

```
str.isdigit()
```

Parameters

☐ **NA**

Return Value

This method returns true if all characters in the string are digits and there is at least one character, false otherwise.

Example

The following example shows the usage of isdigit() method.

```
#!/usr/bin/python

str = "123456";  # Only digit in this string print str.isdigit();

str = "this is string example....wow!!!"; print str.isdigit();
```

When we run above program, it produces following result:

```
True
```

```
False
```

15.   13. islower()

Description

The method **islower()** checks whether all the case-based characters (letters) of the string are lowercase.

Syntax

Following is the syntax for **islower()** method:

```
str.islower()
```

Parameters

 NA

Return Value

This method returns true if all cased characters in the string are lowercase and there is at least one cased character, false otherwise.

Example

The following example shows the usage of islower() method.

```
#!/usr/bin/python

 str = "THIS is string example....wow!!!";  print str.islower();

 str = "this is string example....wow!!!"; print str.islower();
```

When we run above program, it produces following result:

```
False
```

```
True
```

16.   14. isnumeric()

Description

The method **isnumeric()** checks whether the string consists of only numeric characters. This method is present only on unicode objects.

Note: To define a string as Unicode, one simply prefixes a 'u' to the opening quotation mark of the assignment. Below is the example.

Syntax

Following is the syntax for **isnumeric()** method:

```
str.isnumeric()
```

Parameters

☐ **NA**

Return Value

This method returns true if all characters in the string are numeric, false otherwise.

Example

The following example shows the usage of isnumeric() method.

```
#!/usr/bin/python str = u"this2009";   print str.isnumeric(); str = u"23443434";
print str.isnumeric();
```

When we run above program, it produces following result:

```
False
```

```
True
```

17.   15. isspace() Method

Description

The method **isspace()** checks whether the string consists of whitespace. **Syntax**

Following is the syntax for **isspace()** method:

```
str.isspace()
```

Parameters

 NA

Return Value

This method returns true if there are only whitespace characters in the string and there is at least one character, false otherwise.

Example

The following example shows the usage of isspace() method.

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

```
str = "        ";

print str.isspace();



str = "This is string example....wow!!!";

print str.isspace();
```

When we run above program, it produces following result:

```
True
```

```
False
```

18.   16. istitle()

Description

The method **istitle()** checks whether all the case-based characters in the string following non-casebased letters are uppercase and all other case-based characters are lowercase.

Syntax

Following is the syntax for **istitle()** method:

```
str.istitle()
```

Parameters

□ **NA**

Return Value

This method returns true if the string is a titlecased string and there is at least one character, for example uppercase characters may only follow uncased characters and lowercase characters only cased ones.It returns false otherwise.

Example

The following example shows the usage of istitle() method.

```
#!/usr/bin/python

 str = "This Is String Example...Wow!!!";
```

```
print str.istitle();

 str = "This is string example....wow!!!"; print str.istitle();
```

When we run above program, it produces following result:

True

False

19.   17. isupper()

Description

The method **isupper()** checks whether all the case-based characters (letters) of the string are uppercase.

Syntax

Following is the syntax for **isupper()** method:

str.isupper()

Parameters

NA

Return Value

This method returns true if all cased characters in the string are uppercase and there is at least one cased character, false otherwise.

Example

The following example shows the usage of isupper() method.

```
str = "THIS is string example....wow!!!";

print str.isupper();
```

```
#!/usr/bin/python

 str = "THIS IS STRING EXAMPLE....WOW!!!";  print str.isupper();
```

When we run above program, it produces following result:

```
True
```

```
False
```

20.   18. join(seq)

Description

The method **join()** returns a string in which the string elements of sequence have been joined by *str* separator.

Syntax

Following is the syntax for **join()** method:

```
str.join(sequence)
```

Parameters

sequence -- This is a sequence of the elements to be joined.

Return Value

This method returns a string, which is the concatenation of the strings in the sequence seq. The separator between elements is the string providing this method.

Example

The following example shows the usage of join() method.

```
#!/usr/bin/python

 str = "-"; seq = ("a", "b", "c"); # This is sequence of strings.

print str.join( seq );
```

When we run above program, it produces following result:

```
a-b-c
```

21.   19. len(string)

Description

The method **len()** returns the length of the string.

Syntax

Following is the syntax for **len()** method:

```
len( str )
```

Parameters

NA

Return Value

This method returns the length of the string.

Example

The following example shows the usage of len() method.

```
#!/usr/bin/python

 str = "this is string example....wow!!!";

 print "Length of the string: ", len(str);
```

When we run above program, it produces following result:

```
Length of the string:  32
```

22.   20. ljust(width[, fillchar])

Description

The method **ljust()** returns the string left justified in a string of length *width*. Padding is done using the specified *fillchar* (default is a space). The original string is returned if width is less than len(s).

Syntax

Following is the syntax for **ljust()** method:

```
str.ljust(width[, fillchar])
```

Parameters

width -- This is string length in total after padding.

fillchar -- This is filler character, default is a space.

Return Value

This method returns the string left justified in a string of length width. Padding is done using the specified fillchar (default is a space). The original string is returned if width is less than len(s).

Example

The following example shows the usage of ljust() method.

```
#!/usr/bin/python

 str = "this is string example....wow!!!";

 print str.ljust(50, '0');
```

When we run above program, it produces following result:

```
this is string example....wow!!!000000000000000000
```

23.   21. lower()

Description

The method **lower()** returns a copy of the string in which all case-based characters have been lowercased.

Syntax

Following is the syntax for **lower()** method:

```
str.lower()
```

Parameters

NA

Return Value

This method returns a copy of the string in which all case-based characters have been lowercased.

Example

The following example shows the usage of lower() method.

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

```
str = "THIS IS STRING EXAMPLE....WOW!!!";

print str.lower();
```

When we run above program, it produces following result:

this is string example....wow!!!

24.   22. lstrip()

Description

The method **lstrip()** returns a copy of the string in which all chars have been stripped from the beginning of the string (default whitespace characters).

Syntax

Following is the syntax for **lstrip()** method:

str.lstrip([chars])

Parameters

chars -- You can supply what chars have to be trimmed.

Return Value

This method returns a copy of the string in which all chars have been stripped from the beginning of the string (default whitespace characters).

Example

The following example shows the usage of lstrip() method.

#!/usr/bin/python

```
str = "     this is string example....wow!!!     ";

print str.lstrip();

str = "88888888this is string example....wow!!!8888888";

print str.lstrip('8');
```

When we run above program, it produces following result:

```
this is string example....wow!!!

this is string example....wow!!!8888888
```

25.   23. maketrans()

Description

The method **maketrans()** returns a translation table that maps each character in the *intab*string into the character at the same position in the *outtab* string. Then this table is passed to the translate() function.

Note: Both intab and outtab must have the same length.

Syntax

Following is the syntax for **maketrans()** method:

```
str.maketrans(intab, outtab]);
```

Parameters

intab -- This is the string having actual characters.

outtab -- This is the string having corresponding mapping character.

Return Value

This method returns a translate table to be used translate() function.

Example

The following example shows the usage of maketrans() method. Under this, every vowel in a string is replaced by its vowel position:

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

```
from string import maketrans   # Required to call maketrans function.


intab = "aeiou"

outtab = "12345"

trantab = maketrans(intab, outtab)


str = "this is string example....wow!!!";

print str.translate(trantab);
```

When we run above program, it produces following result:

th3s 3s str3ng 2x1mpl2....w4w!!!

26.   24. max(str)

Description

The method **max()** returns the max alphabetical character from the string *str*.

Syntax

Following is the syntax for **max()** method:

max(str)

Parameters

☐ **str** -- This is the string from which max alphabetical character needs to be returned.

Return Value

This method returns the max alphabetical character from the string str.

Example

The following example shows the usage of max() method.

#!/usr/bin/python

```
str = "this is really a string example....wow!!!";

print "Max character: " + max(str);
str = "this is really a string example....wow!!!";

print "Min character: " + min(str);
```

```
str = "this is a string example....wow!!!";

print "Max character: " + max(str);
```

When we run above program, it produces following result:

Max character: y

Max character: x

27.   25. min(str)

Description

The method **min()** returns the min alphabetical character from the string *str*. **Syntax**

Following is the syntax for **min()** method:

```
min(str)
```

Parameters

 str -- This is the string from which min alphabetical character needs to be returned.

Return Value

This method returns the max alphabetical character from the string str.

Example

The following example shows the usage of min() method.

```
#!/usr/bin/python

 str = "this-is-real-string-example....wow!!!"; print "Min character: " +
min(str);
```

When we run above program, it produces following result:

Min character: !

```
print str.replace("is", "was");

print str.replace("is", "was", 3);
```

Min character: !

28.   26. replace(old, new [, max])

Description

The method **replace()** returns a copy of the string in which the occurrences of *old* have been replaced with *new*, optionally restricting the number of replacements to *max*.

Syntax

Following is the syntax for **replace()** method:

```
str.replace(old, new[, max])
```

Parameters

old -- This is old substring to be replaced.

new -- This is new substring, which would replace old substring.

max -- If this optional argument max is given, only the first count occurrences are replaced.

Return Value

This method returns a copy of the string with all occurrences of substring old replaced by new. If the optional argument max is given, only the first count occurrences are replaced.

Example

The following example shows the usage of replace() method.

```
#!/usr/bin/python

  str = "this is string example....wow!!! this is really string";
```

109 When we run above program, it produces following result:

thwas was string example....wow!!! thwas was really string thwas was string example....wow!!! thwas is really string

29.   27. rfind(str, beg=0,end=len(string))

Description

The method **rfind()** returns the last index where the substring str is found, or -1 if no such index exists, optionally restricting the search to string[beg:end].

Syntax

Following is the syntax for **rfind()** method:

```
str.rfind(str, beg=0 end=len(string))
```

Parameters

str -- This specifies the string to be searched.

beg -- This is the starting index, by default its 0.

end -- This is the ending index, by default its equal to the length of the string.

Return Value

This method returns last index if found and -1 otherwise.

Example

The following example shows the usage of rfind() method.

```
#!/usr/bin/python

str = "this is really a string example....wow!!!"; str = "is";

print str.rfind(str);
```

```
print str.rfind(str, 0, 10);

print str.rfind(str, 10, 0);



print str.find(str);

print str.find(str, 0, 10);

print str.find(str, 10, 0);
```

When we run above program, it produces following result:

5

5

-1

2

2

-1

30.   28. rindex(str, beg=0, end=len(string))

Description

The method **rindex()** returns the last index where the substring *str* is found, or raises an exception if no such index exists, optionally restricting the search to string[beg:end].

Syntax

Following is the syntax for **rindex()** method:

```
str.rindex(str, beg=0 end=len(string))
```

Parameters

str -- This specifies the string to be searched.

beg -- This is the starting index, by default its 0

len -- This is ending index, by default its equal to the length of the string.

Return Value

This method returns last index if found otherwise raises an exception if str is not found.

Example

The following example shows the usage of rindex() method.

```
#!/usr/bin/python

 str1 = "this is string example....wow!!!"; str2 = "is";

 print str1.rindex(str2); print str1.index(str2);
```

When we run above program, it produces following result:

5

2

31.   29. rjust(width,[, fillchar])

Description

The method **rjust()** returns the string right justified in a string of length *width*. Padding is done using the specified *fillchar* (default is a space). The original string is returned if width is less than len(s).

Syntax

Following is the syntax for **rjust()** method:

```
str.rjust(width[, fillchar])
```

Parameters

width -- This is the string length in total after padding.

fillchar -- This is the filler character, default is a space.

Return Value

This method returns the string right justified in a string of length width. Padding is done using the specified fillchar (default is a space). The original string is returned if width is less than len(s).

Example

The following example shows the usage of rjust() method.

```
#!/usr/bin/python

 str = "this is string example....wow!!!";

 print str.rjust(50, '0');
```

When we run above program, it produces following result:

000000000000000000this is string example....wow!!!

32.   30. rstrip()

Description

The method **rstrip()** returns a copy of the string in which all *chars* have been stripped from the end of the string (default whitespace characters).

Syntax

Following is the syntax for **rstrip()** method:

str.rstrip([chars])

Parameters

chars -- You can supply what chars have to be trimmed.

Return Value

This method returns a copy of the string in which all chars have been stripped from the end of the string (default whitespace characters).

Example

The following example shows the usage of rstrip() method.

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

```
str = "     this is string example....wow!!!     ";

print str.rstrip();

str = "88888888this is string example....wow!!!8888888";

print str.rstrip('8');
```

When we run above program, it produces following result:

```
     this is string example....wow!!!
```

```
88888888this is string example....wow!!!
```

33.   31. split(str="", num=string.count(str))

Description

The method **split()** returns a list of all the words in the string, using *str* as the separator (splits on all whitespace if left unspecified), optionally limiting the number of splits to *num*.

Syntax

Following is the syntax for **split()** method:

```
str.split(str="", num=string.count(str)).
```

Parameters

str -- This is any delimeter, by default it is space.

num -- this is number of lines to be made.

Return Value

This method returns a list of lines.

Example

The following example shows the usage of split() method.

```
print str.split( );

print str.split(' ', 1 );
```

```
#!/usr/bin/python

 str = "Line1-abcdef \nLine2-abc \nLine4-abcd";
```

When we run above program, it produces following result:

```
['Line1-abcdef', 'Line2-abc', 'Line4-abcd']
```

```
['Line1-abcdef', '\nLine2-abc \nLine4-abcd']
```

34.   32. splitlines(num=string.count('\n'))

Description

The method **splitlines()** returns a list with all the lines in string, optionally including the line breaks (if num is supplied and is true)

Syntax

Following is the syntax for **splitlines()** method:

```
str.splitlines( num=string.count('\n'))
```

Parameters

 num -- This is any number, if present then it would be assumed that line breaks need to be included in the lines.

Return Value

This method returns true if found matching string otherwise false.

Example

The following example shows the usage of splitlines() method.

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

```
  str = "Line1-a b c d e f\nLine2- a b c\n\nLine4- a b c d"; print str.splitlines(
 ); print str.splitlines( 0 ); print str.splitlines( 3 ); print str.splitlines(
 4 );
```

print str.splitlines(5);

When we run above program, it produces following result:

['Line1-a b c d e f', 'Line2- a b c', '', 'Line4- a b c d']

['Line1-a b c d e f', 'Line2- a b c', '', 'Line4- a b c d']

['Line1-a b c d e f\n', 'Line2- a b c\n', '\n', 'Line4- a b c d']

['Line1-a b c d e f\n', 'Line2- a b c\n', '\n', 'Line4- a b c d']

['Line1-a b c d e f\n', 'Line2- a b c\n', '\n', 'Line4- a b c d']

35.   33. startswith(str, beg=0,end=len(string))

Description

The method **startswith()** checks whether string starts with *str*, optionally restricting the matching with the given indices *start* and *end*.

Syntax

Following is the syntax for **startswith()** method:

str.startswith(str, beg=0,end=len(string));

Parameters

str -- This is the string to be checked.

beg -- This is the optional parameter to set start index of the matching boundary.

end -- This is the optional parameter to set start index of the matching boundary.

Return Value

This method returns true if found matching string otherwise false.

```
str = "this is string example....wow!!!";

print str.startswith( 'this' );

print str.startswith( 'is', 2, 4 );

print str.startswith( 'this', 2, 4 );
```

Example

The following example shows the usage of startswith() method.

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

When we run above program, it produces following result:

```
True
```

```
True
```

```
False
```

36.   34. strip([chars])

Description

The method **strip()** returns a copy of the string in which all chars have been stripped from the beginning and the end of the string (default whitespace characters). **Syntax**

Following is the syntax for **strip()** method:

```
str.strip([chars]);
```

Parameters

 chars -- The characters to be removed from beginning or end of the string.

Return Value

This method returns a copy of the string in which all chars have been stripped from the beginning and the end of the string.

Example

The following example shows the usage of strip() method.

```
#!/usr/bin/python

 str = "0000000this is string example....wow!!!0000000"; print str.strip( '0'
);
```

When we run above program, it produces following result:

```
this is string example....wow!!!
```

37.   35. swapcase()

Description

The method **swapcase()** returns a copy of the string in which all the case-based characters have had their case swapped.

Syntax

Following is the syntax for **swapcase()** method:

```
str.swapcase();
```

Parameters

NA

Return Value

This method returns a copy of the string in which all the case-based characters have had their case swapped.

Example

The following example shows the usage of swapcase() method.

```
#!/usr/bin/python

 str = "this is string example....wow!!!"; print str.swapcase();

 str = "THIS IS STRING EXAMPLE....WOW!!!"; print str.swapcase();
```

When we run above program, it produces following result:

38.   36. title()

Description

The method **title()** returns a copy of the string in which first characters of all the words are capitalized.

Syntax

Following is the syntax for **title()** method:

```
str.title();
```

Parameters

NA

Return Value

This method returns a copy of the string in which first characters of all the words are capitalized.

Example

The following example shows the usage of title() method.

```
#!/usr/bin/python

  str = "this is string example....wow!!!"; print str.title();
```

When we run above program, it produces following result:

```
This Is String Example....Wow!!!
```

39.   37. translate(table, deletechars="")

Description

The method **translate()** returns a copy of the string in which all characters have been translated using *table* (constructed with the maketrans() function in the string module), optionally deleting all characters found in the string *deletechars*.

Syntax

Following is the syntax for **translate()** method:

```
str.translate(table[, deletechars]);
```

Parameters

table -- You can use the maketrans() helper function in the string module to create a translation table.

deletechars -- The list of characters to be removed from the source string.

Return Value

This method returns a translated copy of the string.

Example

The following example shows the usage of translate() method. Under this every vowel in a string is replaced by its vowel position:

```
#!/usr/bin/python

 from string import maketrans   # Required to call maketrans function.

 intab = "aeiou" outtab = "12345" trantab = maketrans(intab, outtab)

 str = "this is string example....wow!!!"; print str.translate(trantab);
```

When we run above program, it produces following result:

```
th3s 3s str3ng 2x1mpl2....w4w!!!
```

Following is the example to delete 'x' and 'm' characters from the string:

```
#!/usr/bin/python

 from string import maketrans   # Required to call maketrans function.

 intab = "aeiou"
```
```
outtab = "12345" trantab = maketrans(intab, outtab)

 str = "this is string example....wow!!!"; print str.translate(trantab, 'xm');
```

This will produce following result:

```
th3s 3s str3ng 21pl2....w4w!!!
```

40.   38. upper()

Description

The method **upper()** returns a copy of the string in which all case-based characters have been uppercased.

Syntax

Following is the syntax for **upper()** method:

```
str.upper()
```

Parameters

NA

Return Value

This method returns a copy of the string in which all case-based characters have been uppercased.

Example

The following example shows the usage of upper() method.

```
#!/usr/bin/python

 str = "this is string example....wow!!!";

 print "str.capitalize() : ", str.upper()
```

When we run above program, it produces following result:

```
THIS IS STRING EXAMPLE....WOW!!!
```

41.   39. zfill (width)

Description

The method **zfill()** pads string on the left with zeros to fill width.

Syntax

Following is the syntax for **zfill()** method:

```
str.zfill(width)
```

Parameters

width -- This is final width of the string. This is the width which we would get after filling zeros.

Return Value

This method returns padded string.

Example

The following example shows the usage of zfill() method.

```
#!/usr/bin/python

 str = "this is string example....wow!!!";

 print str.zfill(40); print str.zfill(50);
```

When we run above program, it produces following result:

```
00000000this is string example....wow!!!
```

```
000000000000000000this is string example....wow!!!
```

42.   40. isdecimal()

Description

The method **isdecimal()** checks whether the string consists of only decimal characters. This method are present only on unicode objects.

Note: To define a string as Unicode, one simply prefixes a 'u' to the opening quotation mark of the assignment. Below is the example.

Syntax

Following is the syntax for **isdecimal()** method:

```
str.isdecimal()
```

Parameters

 NA

Return Value

This method returns true if all characters in the string are decimal, false otherwise.

Example

The following example shows the usage of isdecimal() method.

```
#!/usr/bin/python

 str = u"this2009";   print str.isdecimal();

 str = u"23443434"; print str.isdecimal();
```

When we run above program, it produces following result:

```
False
```

```
True
```

10. 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 ];
```

   124 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
```

Deleting 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.

Assume the 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 and 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.

Let us go through the functions in detail:

Cmp(list1, list2)

Description

The method **cmp()** compares elements of two lists.

Syntax

Following is the syntax for **cmp()** method:

```
cmp(list1, list2)
```

Parameters

list1 -- This is the first list to be compared.

list2 -- This is the second list to be compared.

Return Value

If elements are of the same type, perform the compare and return the result. If elements are different types, check to see if they are numbers.

If numbers, perform numeric coercion if necessary and compare.

If either element is a number, then the other element is "larger" (numbers are "smallest").

Otherwise, types are sorted alphabetically by name.

If we reached the end of one of the lists, the longer list is "larger." If we exhaust both lists and share the same data, the result is a tie, meaning that 0 is returned.

Example

The following example shows the usage of cmp() method.

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

```
list1, list2 = [123, 'xyz'], [456, 'abc']


print cmp(list1, list2);

print cmp(list2, list1);

list3 = list2 + [786];

print cmp(list2, list3)
```

When we run above program, it produces following result:

-1

1

-1

43.   len(List)

Description

The method **len()** returns the number of elements in the *list*.

Syntax

Following is the syntax for **len()** method:

```
len(list)
```

Parameters

 list -- This is a list for which number of elements to be counted.

Return Value

This method returns the number of elements in the list.

Example

The following example shows the usage of len() method.

```
#!/usr/bin/python

list1, list2 = [123, 'xyz', 'zara'], [456, 'abc']
```

 print "First list length : ", len(list1); print "Second list length : ", len(list2);

When we run above program, it produces following result:

First list length : 3

Second lsit length : 2

44.   max(list)

Description

The method **max** returns the elements from the *list* with maximum value.

Syntax

Following is the syntax for **max()** method:

max(list)

Parameters

 list -- This is a list from which max valued element to be returned.

Return Value

This method returns the elements from the list with maximum value.

Example

The following example shows the usage of max() method.

```
#!/usr/bin/python

list1, list2 = [123, 'xyz', 'zara', 'abc'], [456, 700, 200]
```

```
Max value element :  zara

Max value element :  700
```

```
 print "Max value element : ", max(list1); print "Max value element : ",
max(list2);
```

When we run above program, it produces following result:

45.   min(list)

Description

The method **min()** returns the elements from the *list* with minimum value.

Syntax

Following is the syntax for **min()** method:

```
min(list)
```

Parameters

 list -- This is a list from which min valued element to be returned.

Return Value

This method returns the elements from the list with minimum value.

Example

The following example shows the usage of min() method.

```
#!/usr/bin/python

 list1, list2 = [123, 'xyz', 'zara', 'abc'], [456, 700, 200]

 print "min value element : ", min(list1); print "min value element : ",
min(list2);
```

When we run above program, it produces following result:

```
min value element :  123 min value element :  200
```

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

Let us go through the methods in detail:

List.append(obj)

Description

The method **append()** appends a passed *obj* into the existing list.

Syntax

Following is the syntax for **append()** method:

```
list.append(obj)
```

Parameters

 obj -- This is the object to be appended in the list.

Return Value

This method does not return any value but updates existing list.

Example

The following example shows the usage of append() method.

```
#!/usr/bin/python

 aList = [123, 'xyz', 'zara', 'abc']; aList.append( 2009 ); print "Updated List
: ", aList;
```

When we run above program, it produces following result:

```
Updated List :  [123, 'xyz', 'zara', 'abc', 2009]
```

46.   list.count(obj)

Description

The method **count()** returns count of how many times *obj* occurs in list.

Syntax

Following is the syntax for **count()** method:

```
list.count(obj)
```

Parameters

 obj -- This is the object to be counted in the list.

Return Value

This method returns count of how many times obj occurs in list.

Example

The following example shows the usage of count() method.

```
#!/usr/bin/python

 aList = [123, 'xyz', 'zara', 'abc', 123];

 print "Count for 123 : ", aList.count(123); print "Count for zara : ",
aList.count('zara');
```

When we run above program, it produces following result:

Count for 123 :   2

Count for zara :   1

47.   list.extend(seq)

Description

The method **extend()** appends the contents of *seq* to list.

Syntax

Following is the syntax for **extend()** method:

```
list.extend(seq)
```

Parameters

☐ **seq** -- This is the list of elements

Return Value

This method does not return any value but add the content to existing list.

```
bList = [2009, 'manni'];

aList.extend(bList)


print "Extended List : ", aList ;
```

Example

The following example shows the usage of extend() method.

```
#!/usr/bin/python

 aList = [123, 'xyz', 'zara', 'abc', 123];
```

When we run above program, it produces following result:

```
Extended List :  [123, 'xyz', 'zara', 'abc', 123, 2009, 'manni']
```

48.   list.index(obj)

Description

The method **index()** returns the lowest index in list that *obj* appears.

Syntax

Following is the syntax for **index()** method:

```
list.index(obj)
```

Parameters

⬚ **obj** -- This is the object to be find out.

Return Value

This method returns index of the found object otherwise raise an exception indicating that value does not find.

Example

The following example shows the usage of index() method.

```
#!/usr/bin/python

 aList = [123, 'xyz', 'zara', 'abc'];

 print "Index for xyz : ", aList.index( 'xyz' ) ; print "Index for zara : ",
aList.index( 'zara' ) ;
```

When we run above program, it produces following result:

```
Index for xyz :   1
```

```
Index for zara :   2
```

49.   list.insert(index,obj)

Description

The method **insert()** inserts object *obj* into list at offset *index*.

Syntax

Following is the syntax for **insert()** method:

```
list.insert(index, obj)
```

Parameters

index -- This is the Index where the object obj need to be inserted.

obj -- This is the Object to be inserted into the given list.

Return Value

This method does not return any value but it inserts the given element at the given index.

Example

The following example shows the usage of insert() method.

```
#!/usr/bin/python

aList = [123, 'xyz', 'zara', 'abc']

aList.insert( 3, 2009)

print "Final List : ", aList
```

When we run above program, it produces following result:

```
Final List :  [123, 'xyz', 'zara', 2009, 'abc']
```

50.   list.pop(obj=list[-1])

Description

The method **pop()** removes and returns last object or *obj* from the list.

Syntax

Following is the syntax for **pop()** method:

```
list.pop(obj=list[-1])
```

Parameters

 obj -- This is an optional parameter, index of the object to be removed from the list.

Return Value

This method returns the removed object from the list.

Example

The following example shows the usage of pop() method.

```
#!/usr/bin/python

 aList = [123, 'xyz', 'zara', 'abc'];

 print "A List : ", aList.pop(); print "B List : ", aList.pop(2);
```

When we run above program, it produces following result:

```
List :   abc
```

```
List :   zara
```

51.   List.remove(obj)

Parameters

 obj -- This is the object to be removed from the list.

Return Value

This method does not return any value but removes the given object from the list.

Example

The following example shows the usage of remove() method.

```
#!/usr/bin/python

 aList = [123, 'xyz', 'zara', 'abc', 'xyz'];

 aList.remove('xyz'); print "List : ", aList; aList.remove('abc'); print "List
: ", aList;
```

When we run above program, it produces following result:

```
List :  [123, 'zara', 'abc', 'xyz']
```

```
List :  [123, 'zara', 'xyz']
```

52.   List.reverse()

Description

The method **reverse()** reverses objects of list in place.

Syntax

Following is the syntax for **reverse()** method:

```
list.reverse()
```

Parameters

NA

Return Value

This method does not return any value but reverse the given object from the list.

Example

The following example shows the usage of reverse() method.

```
#!/usr/bin/python

 aList = [123, 'xyz', 'zara', 'abc', 'xyz'];

 aList.reverse(); print "List : ", aList;
```

When we run above program, it produces following result:

```
List :  ['xyz', 'abc', 'zara', 'xyz', 123]
```

53.   list.sort([func])

Description

The method **reverse()** reverses objects of list in place.

Syntax

Following is the syntax for **reverse()** method:

```
list.reverse()
```

Parameters

NA

Return Value

This method does not return any value but reverse the given object from the list.

Example

The following example shows the usage of reverse() method.

```
#!/usr/bin/python

 aList = [123, 'xyz', 'zara', 'abc', 'xyz'];


```

aList.reverse(); print "List : ", aList;

When we run above program, it produces following result:

List : ['xyz', 'abc', 'zara', 'xyz', 123]

List : ['xyz', 'abc', 'zara', 'xyz', 123]

11. 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')
```

Deleting 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 anymore:

```
('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', '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
```

144 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.

Let us go through tuple functions briefly:

Cmp(tuple1, tuple2)

Description

The method **cmp()** compares elements of two tuples.

Syntax

Following is the syntax for **cmp()** method:

```
cmp(tuple1, tuple2)
```

Parameters

tuple1 -- This is the first tuple to be compared

tuple2 -- This is the second tuple to be compared

Return Value

If elements are of the same type, perform the compare and return the result. If elements are different types, check to see if they are numbers.

If numbers, perform numeric coercion if necessary and compare.

If either element is a number, then the other element is "larger" (numbers are "smallest").

Otherwise, types are sorted alphabetically by name.

If we reached the end of one of the tuples, the longer tuple is "larger." If we exhaust both tuples and share the same data, the result is a tie, meaning that 0 is returned.

Example

The following example shows the usage of cmp() method.

```
#!/usr/bin/python

 tuple1, tuple2 = (123, 'xyz'), (456, 'abc')

 print cmp(tuple1, tuple2); print cmp(tuple2, tuple1); tuple3 = tuple2 +
(786,); print cmp(tuple2, tuple3)
```

When we run above program, it produces following result:

-1

1

-1

54.   Len(tuple)

Description

The method **len()** returns the number of elements in the tuple.

Syntax

Following is the syntax for **len()** method:

```
len(tuple)
```

Parameters

 tuple -- This is a tuple for which number of elements to be counted.

Return Value

This method returns the number of elements in the tuple.

Example

The following example shows the usage of len() method.

```
#!/usr/bin/python

 tuple1, tuple2 = (123, 'xyz', 'zara'), (456, 'abc')

 print "First tuple length : ", len(tuple1); print "Second tuple length : ",
len(tuple2);
```

When we run above program, it produces following result:

```
First tuple length :  3

Second tuple length :  2
```

55.   Max(tuple)

Description

The method **max()** returns the elements from the tuple with maximum value. **Syntax**

Following is the syntax for **max()** method:

```
max(tuple)
```

Parameters

 tuple -- This is a tuple from which max valued element to be returned.

Return Value

This method returns the elements from the tuple with maximum value.

Example

The following example shows the usage of max() method.

```
#!/usr/bin/python

 tuple1, tuple2 = (123, 'xyz', 'zara', 'abc'), (456, 700, 200)

 print "Max value element : ", max(tuple1); print "Max value element : ",
max(tuple2);
```

When we run above program, it produces following result:

```
Max value element :  zara
```

```
Max value element :  700
```

56.   Min(tuple)

Description

The method **min()** returns the elements from the tuple with minimum value.

Syntax

Following is the syntax for **min()** method:

```
min(tuple)
```

Parameters

 tuple -- This is a tuple from which min valued element to be returned.

Return Value

This method returns the elements from the tuple with minimum value.

Example

The following example shows the usage of min() method.

```
#!/usr/bin/python

 tuple1, tuple2 = (123, 'xyz', 'zara', 'abc'), (456, 700, 200)

 print "min value element : ", min(tuple1); print "min value element : ",
min(tuple2);
```

When we run above program, it produces following result:

```
min value element :  123 min value element :  200
```

57.   Tuple(seg)

Description

The method **tuple()** compares elements of two tuples.

Syntax

Following is the syntax for **tuple()** method:

```
tuple( seq )
```

Parameters

 seq -- This is a tuple to be converted into tuple.

Return Value

This method returns the tuple.

Example

The following example shows the usage of tuple() method.

```
#!/usr/bin/python

 aList = (123, 'xyz', 'zara', 'abc'); aTuple = tuple(aList)
```

```
 print "Tuple elements : ", aTuple
```

When we run above program, it produces following result:

```
Tuple elements :  (123, 'xyz', 'zara', 'abc')
```

12. 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['Zara']:
```

```
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. For 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 anymore:

```
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:

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
```

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 and 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.

Let us go through these briefly:

Cmp(dict1, dict2)

Description

The method **cmp()** compares two dictionaries based on key and values.

Syntax

Following is the syntax for **cmp()** method:

```
cmp(dict1, dict2)
```

Parameters

dict1 -- This is the first dictionary to be compared with dict2.

dict2 -- This is the second dictionary to be compared with dict1.

Return Value

This method returns 0 if both dictionaries are equal, -1 if dict1 < dict2, and 1 if dict1 > dic2.

Example

The following example shows the usage of cmp() method.

```
#!/usr/bin/python

 dict1 = {'Name': 'Zara', 'Age': 7}; dict2 = {'Name': 'Mahnaz', 'Age': 27};
dict3 = {'Name': 'Abid', 'Age': 27}; dict4 = {'Name': 'Zara', 'Age': 7}; print
"Return Value : %d" %  cmp (dict1, dict2) print "Return Value : %d" %  cmp
(dict2, dict3) print "Return Value : %d" %  cmp (dict1, dict4)
```

When we run above program, it produces following result:

```
Return Value : -1

Return Value : 1

Return Value : 0
```

58.   len(dict)

Description

The method **len()** gives the total length of the dictionary. This would be equal to the number of items in the dictionary.

Syntax

Following is the syntax for **len()** method:

```
len(dict)
```

Parameters

 dict -- This is the dictionary, whose length needs to be calculated.

Return Value

This method returns the length.

Example

The following example shows the usage of len() method.

```
#!/usr/bin/python

  dict = {'Name': 'Zara', 'Age': 7}; print "Length : %d" % len (dict)
```

When we run above program, it produces following result:

```
Length : 2
```

59.   str(dict)

Description

The method **str()** produces a printable string representation of a dictionary.

Syntax

Following is the syntax for **str()** method:

```
str(dict)
```

Parameters

 dict -- This is the dictionary.

Return Value

This method returns string representation.

Example

The following example shows the usage of str() method.

```
#!/usr/bin/python

 dict = {'Name': 'Zara', 'Age': 7}; print "Equivalent String : %s" % str (dict)
```

When we run above program, it produces following result:

```
Equivalent String : {'Age': 7, 'Name': 'Zara'}
```

60.   type()

Description

The method **type()** returns the type of the passed variable. If passed variable is dictionary then it would return a dictionary type.

Syntax

Following is the syntax for **type()** method:

```
type(dict)
```

Parameters

 dict -- This is the dictionary.

Return Value

This method returns the type of the passed variable.

Example

The following example shows the usage of type() method.

```
#!/usr/bin/python

 dict = {'Name': 'Zara', 'Age': 7}; print "Variable Type : %s" %  type (dict)
```

When we run above program, it produces following result:

```
Variable Type : <type 'dict'>
```

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

Let us go through them briefly:

61.   dict.clear()

Description

The method **clear()** removes all items from the dictionary.

Syntax

Following is the syntax for **clear()** method:

```
dict.clear()
```

Parameters

 NA

Return Value

This method does not return any value.

Example

The following example shows the usage of clear() method.

```
#!/usr/bin/python

 dict = {'Name': 'Zara', 'Age': 7};

 print "Start Len : %d" %  len(dict) dict.clear() print "End Len : %d" %
len(dict)
```

When we run above program, it produces following result:

```
Start Len : 2

End Len : 0
```

62.   Dict.copy()

Description

The method **copy()** returns a shallow copy of the dictionary.

Syntax

Following is the syntax for **copy()** method:

```
dict.copy()
```

Parameters

 NA

Return Value

This method returns a shallow copy of the dictionary.

Example

The following example shows the usage of copy() method.

```
#!/usr/bin/python

 dict1 = {'Name': 'Zara', 'Age': 7};

 dict2 = dict1.copy() print "New Dictinary : %s" %  str(dict2)
```

When we run above program, it produces following result:

```
New Dictinary : {'Age': 7, 'Name': 'Zara'}
```

63.   Dict.fromkeys()

Description

The method **fromkeys()** creates a new dictionary with keys from *seq* and *values* set to value.

Syntax

Following is the syntax for **fromkeys()** method:

```
dict.fromkeys(seq[, value]))
```

Parameters

seq -- This is the list of values which would be used for dictionary keys preparation.

value -- This is optional, if provided then value would be set to this value

Return Value

This method returns the list.

Example

The following example shows the usage of fromkeys() method.

```
#!/usr/bin/python

 seq = ('name', 'age', 'sex')

 dict = dict.fromkeys(seq) print "New Dictionary : %s" %  str(dict)

 dict = dict.fromkeys(seq, 10) print "New Dictionary : %s" %  str(dict)
```

When we run above program, it produces following result:

```
New Dictionary : {'age': None, 'name': None, 'sex': None}

New Dictionary : {'age': 10, 'name': 10, 'sex': 10}
```

64.   Dict.get(key,default=none)

Description

The method **get()** returns a value for the given key. If key is not available then returns default value None.

Syntax

Following is the syntax for **get()** method:

```
dict.get(key, default=None)
```

Parameters

key -- This is the Key to be searched in the dictionary.

default -- This is the Value to be returned in case key does not exist.

Return Value

This method return a value for the given key. If key is not available, then returns default value None.

Example

The following example shows the usage of get() method.

```
#!/usr/bin/python

 dict = {'Name': 'Zara', 'Age': 27}

 print "Value : %s" %  dict.get('Age') print "Value : %s" %  dict.get('Sex',
"Never")
```

When we run above program, it produces following result:

```
Value : 27
```

```
Value : Never
```

65.	Dict.has_key(key)

Description

The method **has_key()** returns true if a given *key* is available in the dictionary, otherwise it returns a false.

Syntax

Following is the syntax for **has_key()** method:

```
dict.has_key(key)
```

Parameters

 key -- This is the Key to be searched in the dictionary.

Return Value

This method return true if a given key is available in the dictionary, otherwise it returns a false.

Example

The following example shows the usage of has_key() method.

```
#!/usr/bin/python

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

 print  "Value  :  %s"  %    dict.has_key('Age')  print  "Value  :  %s"  %
dict.has_key('Sex')
```

When we run above program, it produces following result:

```
Value : True
```

```
Value : False
```

66.   Dict.items()

Description

The method **items()** returns a list of dict's (key, value) tuple pairs

Syntax

Following is the syntax for **items()** method:

```
dict.items()
```

Parameters

□ **NA**

Return Value

This method returns a list of tuple pairs.

Example

The following example shows the usage of items() method.

```
#!/usr/bin/python dict = {'Name': 'Zara', 'Age': 7}
```

 print "Value : %s" % dict.items() When we run above program, it produces following result:

Value : [('Age', 7), ('Name', 'Zara')]

67.   Dict.keys()

Description

The method **keys()** returns a list of all the available keys in the dictionary.

Syntax

Following is the syntax for **keys()** method:

```
dict.keys()
```

Parameters

NA

Return Value

This method returns a list of all the available keys in the dictionary.

Example

The following example shows the usage of keys() method.

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

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

print "Value : %s" %  dict.keys()
```

When we run above program, it produces following result:

```
Value : ['Age', 'Name']
```

68.   dict.setdefault(key, default=None)

Description

The method **setdefault()** is similar to get(), but will set *dict[key]=default* if key is not already in dict.

Syntax

Following is the syntax for **setdefault()** method:

```
dict.setdefault(key, default=None)
```

Parameters

key -- This is the key to be searched.

default -- This is the Value to be returned in case key is not found.

Return Value

This method returns the key value available in the dictionary and if given key is not available then it will return provided default value.

Example

The following example shows the usage of setdefault() method.

```
#!/usr/bin/python

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

print "Value : %s" %  dict.setdefault('Age', None) print "Value : %s" %
dict.setdefault('Sex', None)
```

When we run above program, it produces following result:

```
Value : 7
```

```
Value : None
```

69.   dict.update(dict2)

Description

The method **update()** adds dictionary dict2's key-values pairs in to dict. This function does not return anything.

Syntax

Following is the syntax for **update()** method:

```
dict.update(dict2)
```

Parameters

 dict2 -- This is the dictionary to be added into dict.

Return Value

This method does not return any value.

Example

The following example shows the usage of update() method.

```
#!/usr/bin/python

 dict = {'Name': 'Zara', 'Age': 7} dict2 = {'Sex': 'female' }

 dict.update(dict2) print "Value : %s" %  dict
```

When we run above program, it produces following result:

```
Value : {'Age': 7, 'Name': 'Zara', 'Sex': 'female'}
```

70.   dict.values()

Description

The method **values()** returns a list of all the values available in a given dictionary.

Syntax

Following is the syntax for **values()** method:

```
dict.values()
```

Parameters

NA

Return Value

This method returns a list of all the values available in a given dictionary.

Example

The following example shows the usage of values() method.

```
#!/usr/bin/python

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

 print "Value : %s" %  dict.values()
```

When we run above program, it produces following result:

```
Value : [7, 'Zara']
```

13. DATE AND TIME

A Python program can handle date and time in several ways. Converting between date formats is a common chore for computers. Python's time and calendar modules help track dates and times.

What is Tick?

_____

Time intervals are floating-point numbers in units of seconds. Particular instants in time are expressed in seconds since 12:00am, January 1, 1970(epoch).

There is a popular **time** module available in Python which provides functions for working with times and for converting between representations. The function *time.time()* returns the current system time in ticks since 12:00am, January 1, 1970(epoch). **Example**

```
#!/usr/bin/python import time;  # This is required to include time module.

 ticks = time.time() print "Number of ticks since 12:00am, January 1, 1970:",
ticks
```

This would produce a result something as follows:

```
Number of ticks since 12:00am, January 1, 1970: 7186862.73399
```

Date arithmetic is easy to do with ticks. However, dates before the epoch cannot be represented in this form. Dates in the far future also cannot be represented this way - the cutoff point is sometime in 2038 for UNIX and Windows.

What is TimeTuple?

_____

Many of Python's time functions handle time as a tuple of 9 numbers, as shown below:

Index	Field	Values
0	4-digit year	2008
1	Month	1 to 12

2	Day	1 to 31
3	Hour	0 to 23
4	Minute	0 to 59
5	Second	0 to 61 (60 or 61 are leap-seconds)
6	Day of Week	0 to 6 (0 is Monday)
7	Day of year	1 to 366 (Julian day)
8	Daylight savings	-1, 0, 1, -1 means library determines DST

The above tuple is equivalent to **struct_time** structure. This structure has following attributes:

Index	Attributes	Values
0	tm_year	2008
1	tm_mon	1 to 12
2	tm_mday	1 to 31
3	tm_hour	0 to 23
4	tm_min	0 to 59
5	tm_sec	0 to 61 (60 or 61 are leap-seconds)
6	tm_wday	0 to 6 (0 is Monday)

7	tm_yday	1 to 366 (Julian day)
8	tm_isdst	-1, 0, 1, -1 means library determines DST

Getting Current Time

To translate a time instant from a *seconds since the epoch* floating-point value into a time-tuple, pass the floating-point value to a function (For example, localtime) that returns a time-tuple with all nine items valid.

```python
#!/usr/bin/python import time;

 localtime  =  time.localtime(time.time())  print  "Local  current  time  :",
 localtime
```

This would produce the following result, which could be formatted in any other presentable form:

```
Local current time : time.struct_time(tm_year=2013, tm_mon=7,

tm_mday=17,   tm_hour=21,   tm_min=26,   tm_sec=3,   tm_wday=2,   tm_yday=198,
tm_isdst=0)
```

Getting Formatted Time

You can format any time as per your requirement, but simple method to get time in readable format is asctime():

```python
#!/usr/bin/python import time;

 localtime = time.asctime( time.localtime(time.time()) )
```

```
print "Local current time :", localtime
```

This would produce the following result:

```
Local current time : Tue Jan 13 10:17:09 2009
```

Getting Calendar for a Month

The calendar module gives a wide range of methods to play with yearly and monthly calendars. Here, we print a calendar for a given month (Jan 2008):

```
#!/usr/bin/python import calendar

 cal = calendar.month(2008, 1) print "Here is the calendar:" print cal;
```

This would produce the following result:

```
Here is the calendar:

    January 2008

Mo Tu We Th Fr Sa Su

    1  2  3  4  5  6

 7  8  9 10 11 12 13

14 15 16 17 18 19 20

21 22 23 24 25 26 27

28 29 30 31
```

The *time* Module

There is a popular **time** module available in Python which provides functions for working with times and for converting between representations. Here is the list of all available methods:

Sr. No.	Function with Description
1	time.altzone The offset of the local DST timezone, in seconds west of UTC, if one is defined. This is negative if the local DST timezone is east of UTC (as in Western Europe, including the UK). Only use this if daylight is nonzero.

2	time.asctime([tupletime]) Accepts a time-tuple and returns a readable 24-character string such as 'Tue Dec 11 18:07:14 2008'.
3	time.clock( ) Returns the current CPU time as a floating-point number of seconds. To measure computational costs of different approaches, the value of time.clock is more useful than that of time.time().
4	time.ctime([secs]) Like asctime(localtime(secs)) and without arguments is like asctime( )
5	time.gmtime([secs]) Accepts an instant expressed in seconds since the epoch and returns a time-tuple t with the UTC time. Note : t.tm_isdst is always 0
6	time.localtime([secs]) Accepts an instant expressed in seconds since the epoch and returns a time-tuple t with the local time (t.tm_isdst is 0 or 1, depending on whether DST applies to instant secs by local rules).
7	time.mktime(tupletime) Accepts an instant expressed as a time-tuple in local time and returns a floating-point value with the instant expressed in seconds since the epoch.
8	time.sleep(secs) Suspends the calling thread for secs seconds.
9	time.strftime(fmt[,tupletime]) Accepts an instant expressed as a time-tuple in local time and returns a string representing the instant as specified by string fmt.

10	time.strptime(str,fmt='%a %b %d %H:%M:%S %Y') Parses str according to format string fmt and returns the instant in time-tuple format.
11	time.time( ) Returns the current time instant, a floating-point number of seconds since the epoch.
12	time.tzset() Resets the time conversion rules used by the library routines. The environment variable TZ specifies how this is done.

Let us go through the functions briefly:

71.   time.altzone

Description

The method **altzone()** is the attribute of the **time** module. This returns the offset of the local DST timezone, in seconds west of UTC, if one is defined. This is negative if the local DST timezone is east of UTC (as in Western Europe, including the UK). Only use this if daylight is nonzero.

Syntax

Following is the syntax for **altzone()** method:

```
time.altzone
```

Parameters

NA

Return Value

This method returns the offset of the local DST timezone, in seconds west of UTC, if one is defined.

Example

The following example shows the usage of altzone() method.

```
#!/usr/bin/python import time

  print "time.altzone %d " % time.altzone
```

When we run above program, it produces following result:

```
time.altzone() 25200
```

72.   time.actime([tupletime])

Description

The method **asctime()** converts a tuple or struct_time representing a time as returned by gmtime() or localtime() to a 24-character string of the following form:

'Tue Feb 17 23:21:05 2009'.

Syntax

Following is the syntax for **asctime()** method:

```
time.asctime([t]))
```

Parameters

t -- This is a tuple of 9 elements or struct_time representing a time as returned by gmtime() or localtime() function.

Return Value

This method returns 24-character string of the following form: 'Tue Feb 17 23:21:05 2009'.

Example

The following example shows the usage of asctime() method.

```
#!/usr/bin/python import time

  t = time.localtime()
```

```
print "time.asctime(t): %s " % time.asctime(t)
```

When we run above program, it produces following result:

```
time.asctime(t): Tue Feb 17 09:42:58 2009
```

73.   time.clock( )

Description

The method **clock()** returns the current processor time as a floating point number expressed in seconds on **Unix**. The precision depends on that of the C function of the same name, but in any case, this is the function to use for benchmarking Python or timing algorithms.

On **Windows**, this function returns wall-clock seconds elapsed since the first call to this function, as a floating point number, based on the Win32 function QueryPerformanceCounter.

Syntax

Following is the syntax for **clock()** method:

```
time.clock()
```

Parameters

NA

Return Value

This method returns the current processor time as a floating point number expressed in seconds on *Unix* and in *Windows* it returns wall-clock seconds elapsed since the first call to this function, as a floating point number.

Example

The following example shows the usage of clock() method.

```
#!/usr/bin/python import time

  def procedure():     time.sleep(2.5)


```

```
# measure process time t0 = time.clock() procedure() print time.clock() - t0,
"seconds process time"



# measure wall time t0 = time.time() procedure() print time.time() - t0,
"seconds wall time"
```

When we run above program, it produces following result:

0.0 seconds process time

2.50023603439 seconds wall time

Note: Not all systems can measure the true process time. On such systems (including Windows), clock usually measures the wall time since the program was started.

74.   time.ctime([secs])

Description

The method **ctime()** converts a time expressed in seconds since the epoch to a string representing local time. If secs is not provided or None, the current time as returned by time() is used. This function is equivalent to asctime(localtime(secs)). Locale information is not used by ctime().

Syntax

Following is the syntax for **ctime()** method:

time.ctime([sec])

Parameters

 sec -- These are the number of seconds to be converted into string representation.

Return Value

This method does not return any value.

Example

The following example shows the usage of ctime() method.

```
#!/usr/bin/python import time

 print "time.ctime() : %s" % time.ctime()
```

When we run above program, it produces following result:

```
time.ctime() : Tue Feb 17 10:00:18 2009
```

75.   time.gmtime([secs])

Description

The method **gmtime()** converts a time expressed in seconds since the epoch to a struct_time in UTC in which the dst flag is always zero. If secs is not provided or None, the current time as returned by time() is used.

Syntax

Following is the syntax for **gmtime()** method:

```
time.gmtime([ sec ])
```

Parameters

sec -- These are the number of seconds to be converted into structure struct_time representation.

Return Value

This method does not return any value.

Example

The following example shows the usage of gmtime() method.

```
#!/usr/bin/python import time
```

```
print "time.gmtime() : %s" % time.gmtime()
```

When we run above program, it produces following result:

```
time.gmtime() : (2009, 2, 17, 17, 3, 38, 1, 48, 0)
```

76.   time.localtime([secs])

Description

The method **localtime()** is similar to gmtime() but it converts number of seconds to local time. If secs is not provided or None, the current time as returned by time() is used. The dst flag is set to 1 when DST applies to the given time.

Syntax

Following is the syntax for **localtime()** method:

```
time.localtime([ sec ])
```

Parameters

sec -- These are the number of seconds to be converted into structure struct_time representation.

Return Value

This method does not return any value.

Example

The following example shows the usage of localtime() method.

```
#!/usr/bin/python import time

  print "time.localtime() : %s" % time.localtime()
```

When we run above program, it produces following result:

```
time.localtime() : (2009, 2, 17, 17, 3, 38, 1, 48, 0)
```

77.   time.mktime(tupletime)

Description

The method **mktime()** is the inverse function of localtime(). Its argument is the struct_time or full 9-tuple and it returns a floating point number, for compatibility with time().

If the input value cannot be represented as a valid time, either *OverflowError* or *ValueError*will be raised.

Syntax

Following is the syntax for **mktime()** method:

```
time.mktime(t)
```

Parameters

t -- This is the struct_time or full 9-tuple.

Return Value

This method returns a floating point number, for compatibility with time().

Example

The following example shows the usage of mktime() method.

```
#!/usr/bin/python import time

 t = (2009, 2, 17, 17, 3, 38, 1, 48, 0) secs = time.mktime( t ) print
"time.mktime(t)  :  %f"  %    secs  print  "asctime(localtime(secs)):  %s"  %
time.asctime(time.localtime(secs))
```

When we run above program, it produces following result:

```
time.mktime(t)   :   1234915418.000000   asctime(localtime(secs)):   Tue   Feb   17
17:03:38 2009
```

78. time.sleep(secs)

Description

The method **sleep()** suspends execution for the given number of seconds. The argument may be a floating point number to indicate a more precise sleep time.

The actual suspension time may be less than that requested because any caught signal will terminate the sleep() following execution of that signal's catching routine.

Syntax

Following is the syntax for **sleep()** method:

```
time.sleep(t)
```

Parameters

t -- This is the number of seconds execution to be suspended.

Return Value

This method does not return any value.

Example

The following example shows the usage of sleep() method.

```
#!/usr/bin/python import time

 print "Start : %s" % time.ctime() time.sleep( 5 ) print "End : %s" %
time.ctime()
```

When we run above program, it produces following result:

Start : Tue Feb 17 10:19:18 2009

End : Tue Feb 17 10:19:23 2009

79.   time.strftime(fmt[,tupletime])

Description

The method **strftime()** converts a tuple or struct_time representing a time as returned by gmtime() or localtime() to a string as specified by the format argument.

If t is not provided, the current time as returned by localtime() is used. format must be a string. An exception ValueError is raised if any field in t is outside of the allowed range.

Syntax

Following is the syntax for **strftime()** method:

```
time.strftime(format[, t])
```

Parameters

t -- This is the time in number of seconds to be formatted.

format -- This is the directive which would be used to format given time. The following directives can be embedded in the format string:

Directive

%a - abbreviated weekday name

%A - full weekday name

%b - abbreviated month name

%B - full month name

%c - preferred date and time representation

%C - century number (the year divided by 100, range 00 to 99)

%d - day of the month (01 to 31)

%D - same as %m/%d/%y

%e - day of the month (1 to 31)

%g - like %G, but without the century

%G - 4-digit year corresponding to the ISO week number (see %V).

%h - same as %b

%H - hour, using a 24-hour clock (00 to 23)

%I - hour, using a 12-hour clock (01 to 12)

%j - day of the year (001 to 366)

%m - month (01 to 12)

%M - minute

%n - newline character

%p - either am or pm according to the given time value

%r - time in a.m. and p.m. notation

%R - time in 24 hour notation

%S - second

%t - tab character

%T - current time, equal to %H:%M:%S

%u - weekday as a number (1 to 7), Monday=1. Warning: In Sun Solaris Sunday=1

%U - week number of the current year, starting with the first Sunday as the first day of the first week

%V - The ISO 8601 week number of the current year (01 to 53), where week 1 is the first week that has at least 4 days in the current year, and with Monday as the first day of the week

%W - week number of the current year, starting with the first Monday as the first day of the first week

%w - day of the week as a decimal, Sunday=0

%x - preferred date representation without the time

%X - preferred time representation without the date

%y - year without a century (range 00 to 99)

%Y - year including the century

%Z or %z - time zone or name or abbreviation

%% - a literal % character

Return Value

This method does not return any value.

Example

The following example shows the usage of strftime() method.

```
#!/usr/bin/python import time

 t = (2009, 2, 17, 17, 3, 38, 1, 48, 0) t = time.mktime(t) print
time.strftime("%b %d %Y %H:%M:%S", time.gmtime(t))
```

When we run above program, it produces following result:

```
Feb 18 2009 00:03:38
```

80. time.strptime(str,fmt='%a %b %d %H:%M:%S

%Y')

Description

The method **strptime()** parses a string representing a time according to a format. The return value is a struct_time as returned by gmtime() or localtime().

The format parameter uses the same directives as those used by strftime(); it defaults to "%a %b %d %H:%M:%S %Y" which matches the formatting returned by ctime().

If string cannot be parsed according to format, or if it has excess data after parsing, ValueError is raised.

Syntax

Following is the syntax for **strptime()** method:

```
time.strptime(string[, format])
```

Parameters

string -- This is the time in string format which would be parsed based on the given format.

format -- This is the directive which would be used to parse the given string.

The following directives can be embedded in the format string:

Directive

%a - abbreviated weekday name

%A - full weekday name

%b - abbreviated month name

%B - full month name

%c - preferred date and time representation

%C - century number (the year divided by 100, range 00 to 99)

%d - day of the month (01 to 31)

%D - same as %m/%d/%y

%e - day of the month (1 to 31)

%g - like %G, but without the century

%G - 4-digit year corresponding to the ISO week number (see %V).

%h - same as %b

%H - hour, using a 24-hour clock (00 to 23)

%I - hour, using a 12-hour clock (01 to 12)

%j - day of the year (001 to 366)

%m - month (01 to 12)

%M - minute

%n - newline character

%p - either am or pm according to the given time value

%r - time in a.m. and p.m. notation

%R - time in 24 hour notation

%S - second

%t - tab character

%T - current time, equal to %H:%M:%S

%u - weekday as a number (1 to 7), Monday=1. Warning: In Sun Solaris Sunday=1

%U - week number of the current year, starting with the first Sunday as the first day of the first week

%V - The ISO 8601 week number of the current year (01 to 53), where week 1 is the first week that has at least 4 days in the current year, and with Monday as the first day of the week

%W - week number of the current year, starting with the first Monday as the first day of the first week

%w - day of the week as a decimal, Sunday=0

%x - preferred date representation without the time

%X - preferred time representation without the date

%y - year without a century (range 00 to 99)

%Y - year including the century

%Z or %z - time zone or name or abbreviation

%% - a literal % character

Return Value

This return value is struct_time as returned by gmtime() or localtime().

Example

The following example shows the usage of strptime() method.

```
#!/usr/bin/python import time

 struct_time = time.strptime("30 Nov 00", "%d %b %y") print "returned tuple:
%s " % struct_time
```

When we run above program, it produces following result:

```
returned tuple: (2000, 11, 30, 0, 0, 0, 3, 335, -1)
```

81.   time.time( )

Description

The method **time()** returns the time as a floating point number expressed in seconds since the epoch, in UTC.

Note: Even though the time is always returned as a floating point number, not all systems provide time with a better precision than 1 second. While this function normally returns non-decreasing values, it can return a lower value than a previous call if the system clock has been set back between the two calls.

Syntax

Following is the syntax for **time()** method:

```
time.time()
```

Parameters

NA

Return Value

This method returns the time as a floating point number expressed in seconds since the epoch, in UTC.

Example

The following example shows the usage of time() method.

```
#!/usr/bin/python import time

 print "time.time(): %f " %  time.time() print time.localtime( time.time() )
print time.asctime( time.localtime(time.time()) )
```

When we run above program, it produces following result:

time.time(): 1234892919.655932

(2009, 2, 17, 10, 48, 39, 1, 48, 0)

Tue Feb 17 10:48:39 2009

82.   time.tzset()

Description

The method **tzset()** resets the time conversion rules used by the library routines. The environment variable TZ specifies how this is done.

The standard format of the TZ environment variable is (whitespace added for clarity):

std offset [dst [offset [,start[/time], end[/time]]]]

std and dst: Three or more alphanumerics giving the timezone abbreviations. These will be propagated into time.tzname.

offset: The offset has the form: .hh[:mm[:ss]]. This indicates the value added the local time to arrive at UTC. If preceded by a '-', the timezone is east of the Prime Meridian; otherwise, it is west. If no offset follows *dst*, summer time is assumed to be one hour ahead of standard time.

start[/time], end[/time]: Indicates when to change to and back from DST. The format of the start and end dates are one of the following:

Jn: The Julian day n (1 <= n <= 365). Leap days are not counted, so in all years February 28 is day 59 and March 1 is day 60.

n: The zero-based Julian day (0 <= n <= 365). Leap days are counted, and it is possible to refer to February 29.

Mm.n.d: The d'th day (0 <= d <= 6) or week n of month m of the year (1 <= n <= 5, 1 <= m <= 12, where week 5 means 'the last d day in month m' which may occur in either the fourth or the fifth week). Week 1 is the first week in which the d'th day occurs. Day zero is Sunday.

time: This has the same format as offset except that no leading sign ('-' or '+') is allowed. The default, if time is not given, is 02:00:00.

Syntax

Following is the syntax for **tzset()** method:

```
time.tzset()
```

Parameters

NA

Return Value

This method does not return any value.

Example

The following example shows the usage of tzset() method.

```
#!/usr/bin/python import time import os

 os.environ['TZ']    =    'EST+05EDT,M4.1.0,M10.5.0'    time.tzset()    print
time.strftime('%X %x %Z')

 os.environ['TZ']    =    'AEST-10AEDT-11,M10.5.0,M3.5.0'    time.tzset()    print
time.strftime('%X %x %Z')
```

When we run above program, it produces following result:

```
13:00:40 02/17/09 EST
```

05:00:40 02/18/09 AEDT

There are following two important attributes available with time module:

Sr. No.	Attribute with Description
1	time.timezone Attribute time.timezone is the offset in seconds of the local time zone (without DST) from UTC (>0 in the Americas; <=0 in most of Europe, Asia, Africa).
2	time.tzname Attribute time.tzname is a pair of locale-dependent strings, which are the names of the local time zone without and with DST, respectively.

The *calendar* Module

The calendar module supplies calendar-related functions, including functions to print a text calendar for a given month or year.

By default, calendar takes Monday as the first day of the week and Sunday as the last one. To change this, call calendar.setfirstweekday() function.

Here is a list of functions available with the *calendar* module:

Sr. No.	Function with Description
1	calendar.calendar(year,w=2,l=1,c=6) Returns a multiline string with a calendar for year formatted into three columns separated by c spaces. w is the width in characters of each date; each line has length 21*w+18+2*c. l is the number of lines for each week.
2	calendar.firstweekday( ) Returns the current setting for the weekday that starts each week. By default, when calendar is first imported, this is 0, meaning Monday.

3	calendar.isleap(year)
	Returns True if year is a leap year; otherwise, False.
4	calendar.leapdays(y1,y2)
	Returns the total number of leap days in the years within range(y1,y2).
5	calendar.month(year,month,w=2,l=1)
	Returns a multiline string with a calendar for month of year, one line per week plus two header lines. w is the width in characters of each date; each line has length 7*w+6. l is the number of lines for each week.
6	calendar.monthcalendar(year,month)
	Returns a list of lists of ints. Each sublist denotes a week. Days outside month of year are set to 0; days within the month are set to their day-ofmonth, 1 and up.
7	calendar.monthrange(year,month)
	Returns two integers. The first one is the code of the weekday for the first day of the month month in year; the second one is the number of days in the month. Weekday codes are 0 (Monday) to 6 (Sunday); month numbers are 1 to 12.
8	calendar.prcal(year, w=2, l=1, c=6)
	Like print calendar.calendar(year, w, l, c).
9	calendar.prmonth(year, month, w=2, l=1)
	Like print calendar.month(year, month, w, l).
10	calendar.setfirstweekday(weekday)
	Sets the first day of each week to weekday code weekday. Weekday codes are 0 (Monday) to 6 (Sunday).

11	calendar.timegm(tupletime)
	The inverse of time.gmtime: accepts a time instant in time-tuple form and returns the same instant as a floating-point number of seconds since the epoch.
12	calendar.weekday(year,month,day)
	Returns the weekday code for the given date. Weekday codes are 0 (Monday) to 6 (Sunday); month numbers are 1 (January) to 12 (December).

Other Modules and Functions

If you are interested, then here you would find a list of other important modules and functions to play with date & time in Python:

The datetime Module

The pytz Module

The dateutil Module

14.   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.

As you already know, Python gives you many built-in functions such as print() and 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("I'm first call to user defined function!"); printme("Again 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
```

Passing by Reference Versus Passing by 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

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)
```

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
```

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
```

Variable Length Arguments

```
def functionname([formal_args,] *var_args_tuple ):

   "function_docstring"

   function_suite

   return [expression]
```

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:

An asterisk (*) is placed before the variable name that holds the values of all nonkeyword 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
```

15. 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 is 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 hello.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 (identifiers) that map to objects. A *namespace* is a dictionary of variable names (keys) and their corresponding objects (values).

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:

```
#!/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 keys() function.

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 subsubpackages, 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.

16.   FILES I/O

This chapter covers all the basic I/O functions available in Python. For more functions, please refer to standard Python documentation.

Printing to the Screen

The simplest way to produce output is using the *print* statement where you can pass zero or more expressions separated by commas. This function converts the expressions you pass into a string and writes the result to standard output as follows:

```
#!/usr/bin/python

 print "Python is really a great language,", "isn't it?";
```

This produces the following result on your standard screen:

```
Python is really a great language, isn't it?
```

Reading Keyboard Input

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

raw_input

input

The raw_input Function

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

```
#!/usr/bin/python

 str = raw_input("Enter your input: "); print "Received input is : ", str
```

This prompts you to enter any string and it would display same string on the screen.

When I typed "Hello Python!", its output is like this:

```
Enter your input: Hello Python
```

```
Received input is :  Hello Python
```

The input Function

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

```
#!/usr/bin/python

 str = input("Enter your input: "); print "Received input is : ", str
```

This would produce the following result against the entered input:

```
Enter your input: [x*5 for x in range(2,10,2)]
```

```
Recieved input is :  [10, 20, 30, 40]
```

Opening and Closing Files

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

Python provides basic functions and methods necessary to manipulate files by default. You can do your 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 builtin *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: The file_name argument is a string value that contains the name of the file that you want to access.

access_mode: The access_mode determines the mode in which the file has to be opened, i.e., read, write, append, etc. A complete list of possible values is given below in the table. This is optional parameter and the default file access mode is read (r).

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

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

Modes	Description
r	Opens a file for reading only. The file pointer is placed at the beginning of the file. This is the default mode.
rb	Opens a file for reading only in binary format. The file pointer is placed at the beginning of the file. This is the default mode.
r+	Opens a file for both reading and writing. The file pointer is placed at the beginning of the file.
rb+	Opens a file for both reading and writing in binary format. The file pointer is placed at the beginning of the file.
w	Opens a file for writing only. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
wb	Opens a file for writing only in binary format. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.

w+	Opens a file for both writing and reading. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.

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

The file Object Attributes

Once a file is opened and you have one *file* object, you can get various information related to that file. Here is a list of all attributes related to file object:

Attribute	Description
file.closed	Returns true if file is closed, false otherwise.
file.mode	Returns access mode with which file was opened.

file.name	Returns name of the file.
file.softspace	Returns false if space explicitly required with print, true otherwise.

Example

```
# Open a file 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 Softspace flag :  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();
```

```
#!/usr/bin/python


# Open a file

fo = open("foo.txt", "wb")
```

```
#!/usr/bin/python
print "Name of the file: ", fo.name
```

Example

```
# 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') 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 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 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

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

This produces the following result:

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

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])*   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 import os



# 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 import os



# 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 import os



# 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 import os



# 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 import os



# 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 import os
```

```
# This would  remove "/tmp/test"  directory.

os.rmdir( "/tmp/test"  )
```

File and Directory Related Methods

There are two important sources, which provide a wide range of utility methods to handle and manipulate files and 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.

A **file** object is created using *open* function and here is a list of functions which can be called on this object:

Sr. No.	Methods with Description
1	file.close() Close the file. A closed file cannot be read or written any more.
2	file.flush() Flush the internal buffer, like stdio's fflush. This may be a no-op on some file-like objects.
3	file.fileno() Returns the integer file descriptor that is used by the underlying implementation to request I/O operations from the operating system.
4	file.isatty() Returns True if the file is connected to a tty(-like) device, else False.

5	file.next()
	Returnss the next line from the file each time it is being called.
6	file.read([size])
	Reads at most size bytes from the file (less if the read hits EOF before obtaining size bytes).
7	file.readline([size])
	Reads one entire line from the file. A trailing newline character is kept in the string.
8	file.readlines([sizehint])
	Reads until EOF using readline() and return a list containing the lines. If the optional sizehint argument is present, instead of reading up to EOF, whole lines totalling approximately sizehint bytes (possibly after rounding up to an internal buffer size) are read.
9	file.seek(offset[,whence])
	Sets the file's current position.
10	file.tell()
	Returns the file's current position
11	file.truncate([size])
	Truncates the file's size. If the optional size argument is present, the file is truncated to (at most) that size.
12	file.write(str)
	Writes a string to the file. There is no return value.

| 13 | file.writelines(sequence)

Writes a sequence of strings to the file. The sequence can be any iterable object producing strings, typically a list of strings. |
|---|---|

Let us go through them briefly:

1. file.close()

Description

The method close() closes the opened file. A closed file cannot be read or written any more. Any operation, which requires that the file be opened will raise a *ValueError* after the file has been closed. Calling close() more than once is allowed.

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

Following is the syntax for close() method:

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

Parameters

NA

Return Value

This method does not return any value.

Example

The following example shows the usage of close() method.

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

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

When we run above program, it produces following result:

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

File.flush()

Description

The method flush() flushes the internal buffer, like stdio's fflush. This may be a noop on some file-like objects.

Python automatically flushes the files when closing them. But you may want to flush the data before closing any file.

Syntax

Following is the syntax for flush() method:

```
fileObject.flush();
```

Parameters

NA

Return Value

This method does not return any value.

Example

The following example shows the usage of flush() method.

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

# Here it does nothing, but you can call it with read operation. fo.flush()

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

When we run above program, it produces following result:

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

File.fileno()

Description

The method **fileno()** returns the integer file descriptor that is used by the underlying implementation to request I/O operations from the operating system. **Syntax**

Following is the syntax for **fileno()** method:

```
fileObject.fileno();
```

Parameters

NA

Return Value

This method returns the integer file descriptor.

Example

The following example shows the usage of fileno() method.

```
#!/usr/bin/python



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

 fid = fo.fileno() print "File Descriptor: ", fid



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

When we run above program, it produces following result:

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

```
File Descriptor:  3
```

4. File.isatty()

Description

The method **isatty()** returns True if the file is connected (is associated with a terminal device) to a tty(-like) device, else False.

Syntax

Following is the syntax for **isatty()** method:

```
fileObject.isatty();
```

Parameters

NA

Return Value

This method returns true if the file is connected (is associated with a terminal device) to a tty(-like) device, else false.

Example

The following example shows the usage of isatty() method.

```
#!/usr/bin/python



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

 ret = fo.isatty() print "Return value : ", ret



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

When we run above program, it produces following result:

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

```
Return value :  False
```

5. File.next()

Description

The method **next()** is used when a file is used as an iterator, typically in a loop, the next() method is called repeatedly. This method returns the next input line, or raises *StopIteration* when EOF is hit.

Combining next() method with other file methods like *readline()* does not work right. However, using*seek()* to reposition the file to an absolute position will flush the readahead buffer.

Syntax

Following is the syntax for **next()** method:

```
fileObject.next();
```

Parameters

NA

Return Value

This method returns the next input line.

Example

The following example shows the usage of next() method.

```
#!/usr/bin/python


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


# Assuming file has following 5 lines

# This is 1st line

# This is 2nd line

# This is 3rd line

# This is 4th line # This is 5th line

 for index in range(5):    line = fo.next()    print "Line No %d - %s" % (index,
line)
```

```
# Close opend file
```

fo.close()

When we run above program, it produces following result:

Name of the file: foo.txt

Line No 0 - This is 1st line

Line No 1 - This is 2nd line

Line No 2 - This is 3rd line

Line No 3 - This is 4th line

Line No 4 - This is 5th line

6. File.read([size])

Description

The method **read()** reads at most *size* bytes from the file. If the read hits EOF before obtaining size bytes, then it reads only available bytes.

Syntax

Following is the syntax for **read()** method:

fileObject.read(size);

Parameters

size -- This is the number of bytes to be read from the file.

Return Value

This method returns the bytes read in string.

Example

The following example shows the usage of read() method.

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

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


# Assuming file has following 5 lines

# This is 1st line

# This is 2nd line

# This is 3rd line

# This is 4th line # This is 5th line

 line = fo.read(10) print "Read Line: %s" % (line)



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

When we run above program, it produces following result:

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

```
Read Line: This is 1s
```

7. File.readline([size])

Description

The method **readline()** reads one entire line from the file. A trailing newline character is kept in the string. If the *size* argument is present and non-negative, it is a maximum byte count including the trailing newline and an incomplete line may be returned.

An empty string is returned only when EOF is encountered immediately. **Syntax**

Following is the syntax for **readline()** method:

```
fileObject.readline( size );
```

Parameters

size -- This is the number of bytes to be read from the file.

Return Value

This method returns the line read from the file.

Example

The following example shows the usage of readline() method.

```
#!/usr/bin/python


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


# Assuming file has following 5 lines

# This is 1st line

# This is 2nd line

# This is 3rd line

# This is 4th line # This is 5th line

 line = fo.readline() print "Read Line: %s" % (line)

 line = fo.readline(5) print "Read Line: %s" % (line)


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

When we run above program, it produces following result:

```
Name of the file:  foo.txt

Read Line: This is 1st line

Read Line: This
```

8. file.readline([sizehint])

Description

The method readline() reads one entire line from the file. A trailing newline character is kept in the string. If the *size* argument is present and non-negative, it is a maximum byte count including the trailing newline and an incomplete line may be returned.

An empty string is returned only when EOF is encountered immediately. **Syntax**

Following is the syntax for readline() method:

```
fileObject.readline( size );
```

Parameters

size -- This is the number of bytes to be read from the file.

Return Value

This method returns the line read from the file.

Example

The following example shows the usage of readline() method.

```
#!/usr/bin/python



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

```
# Assuming file has following 5 lines

# This is 1st line
```

```
# This is 2nd line

# This is 3rd line

# This is 4th line # This is 5th line

 line = fo.readline() print "Read Line: %s" % (line)

 line = fo.readline(5) print "Read Line: %s" % (line)




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

When we run above program, it produces following result:

Name of the file: foo.txt

Read Line: This is 1st line

Read Line: This

9. file.seek(offset[,whence])

Description

The method seek() sets the file's current position at the offset. The whence argument is optional and defaults to 0, which means absolute file positioning, other values are 1 which means seek relative to the current position and 2 means seek relative to the file's end.

There is no return value. Note that if the file is opened for appending using either 'a' or 'a+', any seek() operations will be undone at the next write.

If the file is only opened for writing in append mode using 'a', this method is essentially a no-op, but it remains useful for files opened in append mode with reading enabled (mode 'a+').

If the file is opened in text mode using 't', only offsets returned by tell() are legal. Use of other offsets causes undefined behavior.

Note that not all file objects are seekable.

Syntax

Following is the syntax for seek() method:

```
fileObject.seek(offset[, whence])
```

Parameters

offset -- This is the position of the read/write pointer within the file.

whence -- This is optional and defaults to 0 which means absolute file positioning, other values are 1 which means seek relative to the current position and 2 means seek relative to the file's end.

Return Value

This method does not return any value.

Example

The following example shows the usage of seek() method.

```
#!/usr/bin/python


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


# Assuming file has following 5 lines

# This is 1st line

# This is 2nd line

# This is 3rd line

# This is 4th line # This is 5th line

 line = fo.readline()
```

```
print "Read Line: %s" % (line)




# Again set the pointer to the beginning fo.seek(0, 0) line = fo.readline()
print "Read Line: %s" % (line)




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

When we run above program, it produces following result:

```
Name of the file:  foo.txt

Read Line: This is 1st line Read Line: This is 1st line
```

10.   file.tell()

Description

The method tell() returns the current position of the file read/write pointer within the file.

Syntax

Following is the syntax for tell() method:

```
fileObject.tell()
```

Parameters

NA

Return Value

This method returns the current position of the file read/write pointer within the file.

Example

The following example shows the usage of tell() method.

```
#!/usr/bin/python



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



# Assuming file has following 5 lines

# This is 1st line

# This is 2nd line

# This is 3rd line

# This is 4th line # This is 5th line

 line = fo.readline() print "Read Line: %s" % (line)



# Get the current position of the file.

pos = fo.tell() print "Current Position: %d" % (pos)



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

When we run above program, it produces following result:

```
Name of the file:  foo.txt

Read Line: This is 1st line

Current Position: 18
```

11.   file.truncate([size])

Description

The method truncate() truncates the file's size. If the optional size argument is present, the file is truncated to (at most) that size..

The *size* defaults to the current position. The current file position is not changed. Note that if a specified*size* exceeds the file's current size, the result is platform-dependent.

Note: This method would not work in case file is opened in read-only mode. **Syntax**

Following is the syntax for truncate() method:

```
fileObject.truncate( [ size ])
```

Parameters

size -- If this optional argument is present, the file is truncated to (at most) that size.

Return Value

This method does not return any value.

Example

The following example shows the usage of truncate() method.

```
#!/usr/bin/python


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


# Assuming file has following 5 lines

# This is 1st line

# This is 2nd line

# This is 3rd line

# This is 4th line

# This is 5th line
```

```
 line = fo.readline() print "Read Line: %s" % (line)
```

```
# Now truncate remaining file. fo.truncate()



# Try to read file now line = fo.readline() print "Read Line: %s" % (line)



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

When we run above program, it produces following result:

```
Name of the file:  foo.txt Read Line: This is 1st line

Read Line:
```

12.   file.write(str)

Description

The method write() writes a string *str* to the file. There is no return value. Due to buffering, the string may not actually show up in the file until the flush() or close() method is called.

Syntax

Following is the syntax for **write()** method:

```
fileObject.write( str )
```

Parameters

str -- This is the String to be written in the file.

Return Value

This method does not return any value.

Example

The following example shows the usage of write() method.

```
#!/usr/bin/python



# Open a file in write mode fo = open("foo.txt", "rw+") print "Name of the
file: ", fo.name



# Assuming file has following 5 lines

# This is 1st line

# This is 2nd line

# This is 3rd line

# This is 4th line # This is 5th line

 str = "This is 6th line"

# Write a line at the end of the file. fo.seek(0, 2) line = fo.write( str )



# Now read complete file from beginning. fo.seek(0,0) for index in range(6):
line = fo.next()    print "Line No %d - %s" % (index, line)
```

```
# Close opend file

fo.close()
```

When we run above program, it produces following result:

```
Name of the file:  foo.txt

Line No 0 - This is 1st line

Line No 1 - This is 2nd line

Line No 2 - This is 3rd line Line No 3 - This is 4th line




Line No 4 - This is 5th line

Line No 5 - This is 6th line
```

13. file.writelines(sequence)

Description

The method writelines() writes a sequence of strings to the file. The sequence can be any iterable object producing strings, typically a list of strings. There is no return value.

Syntax

Following is the syntax for writelines() method:

```
fileObject.writelines( sequence )
```

Parameters

sequence -- This is the Sequence of the strings.

Return Value

This method does not return any value.

Example

The following example shows the usage of writelines() method.

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

```
# Open a file in witre mode fo = open("foo.txt", "rw+") print "Name of the
file: ", fo.name

# Assuming file has following 5 lines

# This is 1st line

# This is 2nd line

# This is 3rd line

# This is 4th line # This is 5th line

 seq = ["This is 6th line\n", "This is 7th line"] # Write sequence of lines at
the end of the file. fo.seek(0, 2) line = fo.writelines( seq )




# Now read complete file from beginning. fo.seek(0,0) for index in range(7):
line = fo.next()    print "Line No %d - %s" % (index, line)




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

When we run above program, it produces following result:

```
Name of the file:  foo.txt

Line No 0 - This is 1st line
```

OS Object Methods

```
Line No 1 - This is 2nd line

Line No 2 - This is 3rd line

Line No 3 - This is 4th line

Line No 4 - This is 5th line

Line No 5 - This is 6th line

Line No 6 - This is 7th line
```

This provides methods to process files as well as directories.

Sr. No.	Methods with Description
1	os.access(path,mode) Use the real uid/gid to test for access to path.
2	os.chdir(path) Change the current working directory to path
3	os.chflags(path, flags) Set the flags of path to the numeric flags.
4	os.chmod(path, mode) Change the mode of path to the numeric mode.
5	os.chown(path, uid, gid) Change the owner and group id of path to the numeric uid and gid.
6	os.chroot(path) Change the root directory of the current process to path.
7	os.close(fd) Close file descriptor fd.

8	os.closerange(fd_low, fd_high) Close all file descriptors from fd_low (inclusive) to fd_high (exclusive), ignoring errors.
9	os.dup(fd) Return a duplicate of file descriptor fd.
10	os.dup2(fd, fd2) Duplicate file descriptor fd to fd2, closing the latter first if necessary.
11	os.fchdir(fd) Change the current working directory to the directory represented by the file descriptor fd.
12	os.fchmod(fd, mode) Change the mode of the file given by fd to the numeric mode.
13	os.fchown(fd, uid, gid) Change the owner and group id of the file given by fd to the numeric uid and gid.
14	os.fdatasync(fd) Force write of file with filedescriptor fd to disk.
15	os.fdopen(fd[, mode[, bufsize]]) Return an open file object connected to the file descriptor fd.
16	os.fpathconf(fd, name) Return system configuration information relevant to an open file. name specifies the configuration value to retrieve.

17	os.fstat(fd) Return status for file descriptor fd, like stat().
18	os.fstatvfs(fd) Return information about the filesystem containing the file associated with file descriptor fd, like statvfs().
19	os.fsync(fd) Force write of file with filedescriptor fd to disk.
20	os.ftruncate(fd, length) Truncate the file corresponding to file descriptor fd, so that it is at most length bytes in size.
21	os.getcwd() Return a string representing the current working directory.
22	os.getcwdu() Return a Unicode object representing the current working directory.

23	os.isatty(fd) Return True if the file descriptor fd is open and connected to a tty(-like) device, else False.
24	os.lchflags(path, flags) Set the flags of path to the numeric flags, like chflags(), but do not follow symbolic links.
25	os.lchmod(path, mode) Change the mode of path to the numeric mode.
26	os.lchown(path, uid, gid) Change the owner and group id of path to the numeric uid and gid. This function will not follow symbolic links.
27	os.link(src, dst) Create a hard link pointing to src named dst.
28	os.listdir(path) Return a list containing the names of the entries in the directory given by path.
29	os.lseek(fd, pos, how) Set the current position of file descriptor fd to position pos, modified by how.
30	os.lstat(path) Like stat(), but do not follow symbolic links.
31	os.major(device) Extract the device major number from a raw device number.
32	os.makedev(major, minor) Compose a raw device number from the major and minor device numbers.

33	os.makedirs(path[, mode]) Recursive directory creation function.
34	os.minor(device) Extract the device minor number from a raw device number .
35	os.mkdir(path[, mode]) Create a directory named path with numeric mode mode.
36	os.mkfifo(path[, mode]) Create a FIFO (a named pipe) named path with numeric mode mode. The default mode is 0666 (octal).
37	os.mknod(filename[, mode=0600, device]) Create a filesystem node (file, device special file or named pipe) named filename.
38	os.open(file, flags[, mode]) Open the file file and set various flags according to flags and possibly its mode according to mode.
39	os.openpty() Open a new pseudo-terminal pair. Return a pair of file descriptors (master, slave) for the pty and the tty, respectively.
40	os.pathconf(path, name) Return system configuration information relevant to a named file.
41	os.pipe() Create a pipe. Return a pair of file descriptors (r, w) usable for reading and writing, respectively.

42	os.popen(command[, mode[, bufsize]])
	Open a pipe to or from command.
43	os.read(fd, n)
	Read at most n bytes from file descriptor fd. Return a string containing the bytes read. If the end of the file referred to by fd has been reached, an empty string is returned.
44	os.readlink(path)
	Return a string representing the path to which the symbolic link points.
45	os.remove(path)
	Remove the file path.
46	os.removedirs(path)
	Remove directories recursively.
47	os.rename(src, dst)
	Rename the file or directory src to dst.
48	os.renames(old, new)
	Recursive directory or file renaming function.
49	os.rmdir(path)
	Remove the directory path
50	os.stat(path)
	Perform a stat system call on the given path.
51	os.stat_float_times([newvalue])

		Determine whether stat_result represents time stamps as float objects.
52	os.statvfs(path)	Perform a statvfs system call on the given path.
53	os.symlink(src, dst)	Create a symbolic link pointing to src named dst.
54	os.tcgetpgrp(fd)	Return the process group associated with the terminal given by fd (an open file descriptor as returned by open()).
55	os.tcsetpgrp(fd, pg)	Set the process group associated with the terminal given by fd (an open file descriptor as returned by open()) to pg.
56	os.tempnam([dir[, prefix]])	Return a unique path name that is reasonable for creating a temporary file.
57	os.tmpfile()	Return a new file object opened in update mode (w+b).
58	os.tmpnam()	Return a unique path name that is reasonable for creating a temporary file.
59	os.ttyname(fd)	Return a string which specifies the terminal device associated with file descriptor fd. If fd is not associated with a terminal device, an exception is raised.

60	os.unlink(path) Remove the file path.
61	os.utime(path, times) Set the access and modified times of the file specified by path.
62	os.walk(top[, topdown=True[, onerror=None[, followlinks=False]]]) Generate the file names in a directory tree by walking the tree either topdown or bottom-up.
63	os.write(fd, str) Write the string str to file descriptor fd. Return the number of bytes actually written.

17. EXCEPTIONS

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

Exception Handling: This would be covered in this tutorial. Here is a list standard Exceptions available in Python: Standard Exceptions.

Assertions: This would be covered in Assertions in Python tutorial.

List of Standard Exceptions:

EXCEPTION NAME	DESCRIPTION
Exception	Base class for all exceptions
StopIteration	Raised when the next() method of an iterator does not point to any object.
SystemExit	Raised by the sys.exit() function.
StandardError	Base class for all built-in exceptions except StopIteration and SystemExit.
ArithmeticError	Base class for all errors that occur for numeric calculation.
OverflowError	Raised when a calculation exceeds maximum limit for a numeric type.
FloatingPointError	Raised when a floating point calculation fails.
ZeroDivisonError	Raised when division or modulo by zero takes place for all numeric types.
AssertionError	Raised in case of failure of the Assert statement.

AttributeError	Raised in case of failure of attribute reference or assignment.
EOFError	Raised when there is no input from either the raw_input() or input() function and the end of file is reached.
ImportError	Raised when an import statement fails.
KeyboardInterrupt	Raised when the user interrupts program execution, usually by pressing Ctrl+c.
LookupError	Base class for all lookup errors.
IndexError	Raised when an index is not found in a sequence.
KeyError	Raised when the specified key is not found in the dictionary.
NameError	Raised when an identifier is not found in the local or global namespace.
UnboundLocalError	Raised when trying to access a local variable in a function or method but no value has been assigned to it.
EnvironmentError	Base class for all exceptions that occur outside the Python environment.
IOError	Raised when an input/ output operation fails, such as the print statement or the open() function when trying to open a file that does not exist.
OSError	Raised for operating system-related errors.
SyntaxError	Raised when there is an error in Python syntax.

IndentationError	Raised when indentation is not specified properly.
SystemError	Raised when the interpreter finds an internal problem, but when this error is encountered the Python interpreter does not exit.
SystemExit	Raised when Python interpreter is quit by using the sys.exit() function. If not handled in the code, causes the interpreter to exit.
TypeError	Raised when an operation or function is attempted that is invalid for the specified data type.
ValueError	Raised when the built-in function for a data type has the valid type of arguments, but the arguments have invalid values specified.
RuntimeError	Raised when a generated error does not fall into any category.
NotImplementedError	Raised when an abstract method that needs to be implemented in an inherited class is not actually implemented.

Assertions in Python

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

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

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

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

The *assert* Statement

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

The syntax for assert is:

```
assert Expression[, Arguments]
```

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

Example

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

```
#!/usr/bin/python

 def KelvinToFahrenheit(Temperature):

   assert (Temperature >= 0),"Colder than absolute zero!"     return ((Temperature-273)*1.8)+32

 print KelvinToFahrenheit(273) print int(KelvinToFahrenheit(505.78)) print KelvinToFahrenheit(-5)
```

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

```
32.0

451

Traceback (most recent call last):   File "test.py", line 9, in <module>
print KelvinToFahrenheit(-5)

  File "test.py", line 4, in KelvinToFahrenheit     assert (Temperature >=
0),"Colder than absolute zero!"

AssertionError: Colder than absolute zero!
```

What is Exception?

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

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

Handling an Exception

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

Syntax

Here is simple syntax of *try....except...else* blocks:

```
try:

   You do your operations here;

   .................... except ExceptionI:

   If there is ExceptionI, then execute this block.

except ExceptionII:

   If there is ExceptionII, then execute this block.

   ....................

else:

   If there is no exception then execute this block.
```

Here are few important points about the above-mentioned syntax:

A single try statement can have multiple except statements. This is useful when the try block contains statements that may throw different types of exceptions.

You can also provide a generic except clause, which handles any exception.

After the except clause(s), you can include an else-clause. The code in the else-block executes if the code in the try: block does not raise an exception.

The else-block is a good place for code that does not need the try: block's protection.

Example

This example opens a file, writes content in the, file and comes out gracefully because there is no problem at all:

```
#!/usr/bin/python

 try:

   fh = open("testfile", "w")
```
```
   fh.write("This is my test file for exception handling!!") except IOError:

   print "Error: can\'t find file or read data" else:

   print "Written content in the file successfully"    fh.close()
```

This produces the following result:

```
Written content in the file successfully
```

Example

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

```
#!/usr/bin/python

 try:

   fh = open("testfile", "r")    fh.write("This is my test file for exception
handling!!") except IOError:

   print "Error: can\'t find file or read data" else:

   print "Written content in the file successfully"
```

This produces the following result:

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

The *except* Clause with No Exceptions

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

```
try:

   You do your operations here;

   ..................... except:

   If there is any exception, then execute this block.

   ..................... else:

   If there is no exception then execute this block.
```

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

The *except* Clause with Multiple Exceptions

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

```
try:

   You do your operations here;

   .....................

except(Exception1[, Exception2[,...ExceptionN]]):

   If there is any exception from the given exception list,     then execute
this block.

   ..................... else:

   If there is no exception then execute this block.
```

The try-finally Clause

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

```
try:

   You do your operations here;

   .....................

   Due to any exception, this may be skipped. finally:

   This would always be executed.

   .....................
```

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

Example

```
#!/usr/bin/python

 try:

   fh = open("testfile", "w")    fh.write("This is my test file for exception
handling!!") finally:

   print "Error: can\'t find file or read data"
```

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

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

Same example can be written more cleanly as follows:

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

```
 try:

   fh = open("testfile", "w")    try:

      fh.write("This is my test file for exception handling!!")    finally:
```

```
    print "Going to close the file"        fh.close() except IOError:

  print "Error: can\'t find file or read data"
```

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

Argument of an Exception

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

```
try:

   You do your operations here;    .....................

except ExceptionType, Argument:

   You can print value of Argument here...
```

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

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

Example

Following is an example for a single exception:

```
#!/usr/bin/python



# Define a function here. def temp_convert(var):    try:
```

```
        return int(var)     except ValueError, Argument:

        print "The argument does not contain numbers\n", Argument

# Call above function here.

temp_convert("xyz");
```

This produces the following result:

```
The argument does not contain numbers invalid literal for int() with base 10:
'xyz'
```

Raising an Exception

You can raise exceptions in several ways by using the raise statement. The general syntax for the **raise** statement is as follows. **Syntax**

```
raise [Exception [, args [, traceback]]]
```

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

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

Example

An exception can be a string, a class or an object. Most of the exceptions that the Python core raises are classes, with an argument that is an instance of the class. Defining new exceptions is quite easy and can be done as follows:

```
def functionName( level ):    if level < 1:

      raise "Invalid level!", level

      # The code below to this would not be executed

      # if we raise the exception
```

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

```
try:

   Business Logic here... except "Invalid level!":   Exception handling here...

else:

   Rest of the code here...
```

User-Defined Exceptions

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

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

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

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

```
class Networkerror(RuntimeError):

   def __init__(self, arg):        self.args = arg
```

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

```
try:

   raise Networkerror("Bad hostname") except Networkerror,e:

   print e.args
```

18.   CLASSES AND OBJECTS

Python has been an object-oriented language since it existed. Because of 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 small introduction of Object-Oriented Programming (OOP) to bring you at speed:

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's 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 the 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 instances of a 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/python

 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.age = 7  # Add an 'age' attribute. emp1.age = 8  # Modify 'age' attribute.
del emp1.age  # 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, 'age')    # Returns true if 'age' attribute exists getattr(emp1,
'age')     # Returns value of 'age' attribute setattr(emp1, 'age', 8) # Set
attribute 'age' at 8 delattr(empl, 'age')    # Delete attribute 'age'
```

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/python

 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

 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__: ()

Employee.__dict__: {'__module__': '__main__', 'displayCount':

<function displayCount at 0xb7c84994>, 'empCount': 2,

'displayEmployee': <function displayEmployee at 0xb7c8441c>,

'__doc__': 'Common base class for all employees',

'__init__': <function __init__ at 0xb7c846bc>}
```

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 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's 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. But 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/python

 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 following result:

```
3083401324 3083401324 3083401324
```

```
Point destroyed
```

Note: Ideally, you should define your classes in separate file, then you should import them in your main program file using *import* statement.

Class Inheritance

___

Instead of starting from scratch, you can create a class by deriving it from a preexisting 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/python

 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
```

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
```

  269 .....

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 because you may want special or different functionality in your subclass.

Example

```
#!/usr/bin/python

 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

Following table lists some generic functionality that you can override in your own classes:

Sr. No.	Method, Description, and 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/python

 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 are not be directly visible to outsiders.

Example

```
#!/usr/bin/python

 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

19. REGULAREXPRESSIONS

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:

Parameter	Description
pattern	This is the regular expression to be matched.
string	This is the string, which would be searched to match the pattern at the beginning of string.
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.

Match Object Methods	Description
group(num=0)	This method returns entire match (or specific subgroup num)
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)

 if matchObj:

    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:

Parameter	Description
pattern	This is the regular expression to be matched.
string	This is the string, which is searched to match the pattern anywhere in the string.
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.

Match Object Methods	Description
group(num=0)	This method returns entire match (or specific subgroup num).
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)

if searchObj:

   print "searchObj.group() : ", searchObj.group()
```

```
   print "searchObj>gseap(h0bj.grospé)chObj.geenp(0bj.group()

elsprint "searchObj.group(2) : ", searchObj.group(2)

elsprint "Nothing found!!"

   print "Nothing found!!"
```

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
```

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) if matchObj:

   print "match --> matchObj.group() : ", matchObj.group() else:

   print "No match!!"

 searchObj = re.search( r'dogs', line, re.M|re.I) if searchObj:
```

   277 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:

Modifier	Description
re.I	Performs case-insensitive matching.

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).
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).
re.S	Makes a period (dot) match any character, including a newline.
re.U	Interprets letters according to the Unicode character set. This flag affects the behavior of \w, \W, \b, \B.
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:

Pattern	Description
^	Matches beginning of line.
$	Matches end of line.
.	Matches any single character except newline. Using m option allows it to match newline as well.
[...]	Matches any single character in brackets.
[^...]	Matches any single character not in brackets
re*	Matches 0 or more occurrences of preceding expression.
re+	Matches 1 or more occurrence of preceding expression.
re?	Matches 0 or 1 occurrence of preceding expression.
re{ n}	Matches exactly n number of occurrences of preceding expression.
re{ n,}	Matches n or more occurrences of preceding expression.
re{ n, m}	Matches at least n and at most m occurrences of preceding expression.
a| b	Matches either a or b.
(re)	Groups regular expressions and remembers matched text.

(?imx)	Temporarily toggles on i, m, or x options within a regular expression. If in parentheses, only that area is affected.
(?-imx)	Temporarily toggles off i, m, or x options within a regular expression. If in parentheses, only that area is affected.
(?: re)	Groups regular expressions without remembering matched text.
(?imx: re)	Temporarily toggles on i, m, or x options within parentheses.
(?-imx: re)	Temporarily toggles off i, m, or x options within parentheses.
(?#...)	Comment.
(?= re)	Specifies position using a pattern. Doesn't have a range.
(?! re)	Specifies position using pattern negation. Does not have a range.
(?> re)	Matches independent pattern without backtracking.
\w	Matches word characters.
\W	Matches non-word characters.
\s	Matches whitespace. Equivalent to [\t\n\r\f].
\S	Matches non-whitespace.
\d	Matches digits. Equivalent to [0-9].
\D	Matches non-digits.

\A	Matches beginning of string.
\Z	Matches end of string. If a newline exists, it matches just before newline.
\z	Matches end of string.
\G	Matches point where last match finished.
\b	Matches word boundaries when outside brackets. Matches backspace (0x08) when inside brackets.
\B	Matches non-word boundaries.
\n, \t, etc.	Matches newlines, carriage returns, tabs, etc.
\1...\9	Matches nth grouped subexpression.
\10	Matches nth grouped subexpression if it matched already. Otherwise refers to the octal representation of a character code.

Regular-Expression Examples

Literal characters

Example	Description
python	Match "python".

Character classes

Example	Description
[Pp]ython	Match "Python" or "python"

rub[ye]	Match "ruby" or "rube"
[aeiou]	Match any one lowercase vowel
[0-9]	Match any digit; same as [0123456789]
[a-z]	Match any lowercase ASCII letter
[A-Z]	Match any uppercase ASCII letter
[a-zA-Z0-9]	Match any of the above
[^aeiou]	Match anything other than a lowercase vowel
[^0-9]	Match anything other than a digit

Special Character Classes

Example	Description
.	Match any character except newline
\d	Match a digit: [0-9]
\D	Match a non-digit: [^0-9]
\s	Match a whitespace character: [\t\r\n\f]
\S	Match non-whitespace: [^ \t\r\n\f]
\w	Match a single word character: [A-Za-z0-9_]

\W	Match a non-word character: [^A-Za-z0-9_]

Repetition Cases

Example	Description
ruby?	Match "rub" or "ruby": the y is optional.
ruby*	Match "rub" plus 0 or more ys.
ruby+	Match "rub" plus 1 or more ys.
\d{3}	Match exactly 3 digits.
\d{3,}	Match 3 or more digits.
\d{3,5}	Match 3, 4, or 5 digits.

Nongreedy repetition

This matches the smallest number of repetitions:

Example	Description
<.*>	Greedy repetition: matches "<python>perl>".
<.*?>	Nongreedy: matches "<python>" in "<python>perl>".

Grouping with Parentheses

Example	Description
\D\d+	No group: + repeats \d.

(\D\d)+	Grouped: + repeats \D\d pair.
([Pp]ython(,)?)+	Match "Python", "Python, python, python", etc.

Backreferences

This matches a previously matched group again:

Example	Description
([Pp])ython&\1ails	Match python&pails or Python&Pails.
(['"])[^\1]*\1	Single or double-quoted string. \1 matches whatever the 1st group matched. \2 matches whatever the 2nd group matched, etc.

Alternatives

Example	Description
python|perl	Match "python" or "perl".
rub(y|le))	Match "ruby" or "ruble".
Python(!+|\?)	"Python" followed by one or more ! or one ?

Anchors

This needs to specify match position.

Example	Description
^Python	Match "Python" at the start of a string or internal line.
Python$	Match "Python" at the end of a string or line.

\APython	Match "Python" at the start of a string.
Python\Z	Match "Python" at the end of a string.
\bPython\b	Match "Python" at a word boundary.
\brub\B	\B is non-word boundary: match "rub" in "rube" and "ruby" but not alone.
Python(?=!)	Match "Python", if followed by an exclamation point.
Python(?!!)	Match "Python", if not followed by an exclamation point.

Special Syntax with Parentheses

Example	Description
R(?#comment)	Matches "R". All the rest is a comment.
R(?i)uby	Case-insensitive while matching "uby".
R(?i:uby)	Same as above
rub(?:y|le))	Group only without creating \1 back reference.

Python

25. XML PROCESSING

XML is a portable, open source language that allows programmers to develop applications that can be read by other applications, regardless of operating system and/or developmental language.

What is XML?

The Extensible Markup Language (XML) is a markup language much like HTML or SGML. This is recommended by the World Wide Web Consortium and available as an open standard.

XML is extremely useful for keeping track of small to medium amounts of data without requiring a SQL-based backbone.

XML Parser Architectures and APIs:

The Python standard library provides a minimal but useful set of interfaces to work with XML.

The two most basic and broadly used APIs to XML data are the SAX and DOM interfaces.

Simple API for XML (SAX) : Here, you register callbacks for events of interest and then let the parser proceed through the document. This is useful when your documents are large or you have memory limitations, it parses the file as it reads it from disk and the entire file is never stored in memory.

Document Object Model (DOM) API : This is a World Wide Web Consortium recommendation wherein the entire file is read into memory and stored in a hierarchical (tree-based) form to represent all the features of an XML document.

SAX obviously cannot process information as fast as DOM can when working with large files. On the other hand, using DOM exclusively can really kill your resources, especially if used on a lot of small files.

SAX is read-only, while DOM allows changes to the XML file. Since these two different APIs literally complement each other, there is no reason why you can not use them both for large projects.

For all our XML code examples, let's use a simple XML file *movies.xml* as an input:

```
<collection shelf="New Arrivals">

<movie title="Enemy Behind">

    <type>War, Thriller</type>

    <format>DVD</format>

    <year>2003</year>

    <rating>PG</rating>

    <stars>10</stars>

    <description>Talk about a US-Japan war</description>

</movie>

<movie title="Transformers">

    <type>Anime, Science Fiction</type>

    <format>DVD</format>

    <year>1989</year>

    <rating>R</rating>

    <stars>8</stars>

    <description>A schientific fiction</description>

</movie>

    <movie title="Trigun">

    <type>Anime, Action</type>

    <format>DVD</format>

    <episodes>4</episodes>

    <rating>PG</rating>

    <stars>10</stars>
```

```
    <type>Comedy</type>

    <format>VHS</format>

    <rating>PG</rating>

    <stars>2</stars>

    <description>Viewable boredom</description>

</movie>

</collection>
```

```
    <description>Vash the Stampede!</description>

</movie>

<movie title="Ishtar">
```

Parsing XML with SAX APIs

SAX is a standard interface for event-driven XML parsing. Parsing XML with SAX generally requires you to create your own ContentHandler by subclassing xml.sax.ContentHandler.

Your *ContentHandler* handles the particular tags and attributes of your flavor(s) of XML. A ContentHandler object provides methods to handle various parsing events. Its owning parser calls ContentHandler methods as it parses the XML file.

The methods *startDocument* and *endDocument* are called at the start and the end of the XML file. The method *characters(text)* is passed character data of the XML file via the parameter text.

The ContentHandler is called at the start and end of each element. If the parser is not in namespace mode, the methods *startElement(tag, attributes)* and *endElement(tag)* are called; otherwise, the corresponding methods *startElementNS* and *endElementNS* are called. Here, tag is the element tag, and attributes is an Attributes object.

Here are other important methods to understand before proceeding:

The *make_parser* Method

Following method creates a new parser object and returns it. The parser object created will be of the first parser type the system finds.

```
xml.sax.make_parser( [parser_list] )
```

Here is the detail of the parameters:

 parser_list: The optional argument consisting of a list of parsers to use which must all implement the make_parser method.

The *parse* Method

Following method creates a SAX parser and uses it to parse a document.

```
xml.sax.parse( xmlfile, contenthandler[, errorhandler])
```

Here is the detail of the parameters:

xmlfile: This is the name of the XML file to read from.

contenthandler: This must be a ContentHandler object.

errorhandler: If specified, errorhandler must be a SAX ErrorHandler object.

The *parseString* Method

There is one more method to create a SAX parser and to parse the specified **XML string**.

```
xml.sax.parseString(xmlstring, contenthandler[, errorhandler])
```

Here is the detail of the parameters:

xmlstring: This is the name of the XML string to read from.

contenthandler: This must be a ContentHandler object.

errorhandler: If specified, errorhandler must be a SAX ErrorHandler object.

Example

```
#!/usr/bin/python

 import xml.sax

 class MovieHandler( xml.sax.ContentHandler ):

   def __init__(self):          self.CurrentData = ""          self.type = ""
self.format = ""
```

```
      self.year = ""            self.rating = ""            self.stars = ""
self.description = ""



   # Call when an element starts    def startElement(self, tag, attributes):

      self.CurrentData = tag       if tag == "movie":

         print "*****Movie*****"                    title = attributes["title"]
print "Title:", title



   # Call when an elements ends     def endElement(self, tag):          if
self.CurrentData == "type":          print "Type:", self.type         elif
self.CurrentData == "format":         print "Format:", self.format        elif
self.CurrentData == "year":

         print "Year:", self.year        elif self.CurrentData == "rating":
print "Rating:", self.rating       elif self.CurrentData == "stars":

         print "Stars:", self.stars               elif self.CurrentData ==
"description":
```

```
         print "Description:", self.description          self.CurrentData = ""



   # Call when a character is read     def characters(self, content):          if
self.CurrentData == "type":

         self.type = content        elif self.CurrentData == "format":

         self.format = content        elif self.CurrentData == "year":

         self.year = content        elif self.CurrentData == "rating":

         self.rating = content        elif self.CurrentData == "stars":

         self.stars = content        elif self.CurrentData == "description":

         self.description = content

   if ( __name__ == "__main__"):     # create an XMLReader       parser =
xml.sax.make_parser()

   #                 turn                off                namepsaces
parser.setFeature(xml.sax.handler.feature_namespaces, 0)

   # override the default ContextHandler      Handler = MovieHandler()
parser.setContentHandler( Handler )
```

```
***parseMovie***("movies.xml")
```

```
Title: Enemy Behind

Type: War, Thriller

Format: DVD

Year: 2003

Rating: PG

Stars: 10

Description: Talk about a US-Japan war

*****Movie*****

Title: Transformers

Type: Anime, Science Fiction

Format: DVD

Year: 1989

Rating: R

Stars: 8

Description: A schientific fiction

*****Movie*****
*****Movie*****
Title: Trigun
Title: Ishtar
Type: Anime, Action
Type: Comedy
Format: DVD
Format: VHS
Rating: PG
Rating: PG
Stars: 10
Stars: 2
Description: Vash the Stampede!
Description: Viewable boredom
```

This would produce following result:

For a complete detail on SAX API documentation, please refer to standard Python SAX APIs.

Parsing XML with DOM APIs

The Document Object Model (DOM) is a cross-language API from the World Wide Web Consortium (W3C) for accessing and modifying XML documents.

The DOM is extremely useful for random-access applications. SAX only allows you a view of one bit of the document at a time. If you are looking at one SAX element, you have no access to another.

Here is the easiest way to quickly load an XML document and to create a minidom object using the xml.dom module. The minidom object provides a simple parser method that quickly creates a DOM tree from the XML file.

The sample phrase calls the parse(file [,parser]) function of the minidom object to parse the XML file designated by file into a DOM tree object.

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

 from xml.dom.minidom import parse import xml.dom.minidom



# Open XML document using minidom parser DOMTree = xml.dom.minidom.parse("movies.xml") collection = DOMTree.documentElement if collection.hasAttribute("shelf"):
```

```
    print "Root element : %s" % collection.getAttribute("shelf")
```

```
Root element : New Arrivals

#*Get*all*the*movies in the collection

motes = collection.getElementsByTagName("movie")

Type: War, Thriller

#oPrint detail of each movie.

for movie in movies:

    print "*****Movie*****"

    if movie.hasAttribute("title"):

        print "Title: %s" % movie.getAttribute("title")


    type = movie.getElementsByTagName('type')[0]

    print "Type: %s" % type.childNodes[0].data

    format = movie.getElementsByTagName('format')[0]

    print "Format: %s" % format.childNodes[0].data

    rating = movie.getElementsByTagName('rating')[0]

    print "Rating: %s" % rating.childNodes[0].data

    description = movie.getElementsByTagName('description')[0]

    print "Description: %s" % description.childNodes[0].data
```

352 This would produce the following result:

```
Description: Talk about a US-Japan war

*****Movie*****

Title: Transformers

Type: Anime, Science Fiction

Format: DVD

Rating: R

Description: A schientific fiction

*****Movie*****

Title: Trigun

Type: Anime, Action

Format: DVD

Rating: PG

Description: Vash the Stampede!

*****Movie*****

Title: Ishtar

Type: Comedy

Format: VHS

Rating: PG

Description: Viewable boredom
```

For a complete detail on DOM API documentation, please refer to standard [Python DOM APIs](#).

26. GUI PROGRAMMING

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**http://wxpython.org**.

JPython: JPython is a Python port for Java which gives Python scripts seamless access to Java class libraries on the local machine **http://www.jython.org**.

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:

Operator	Description
Button	The Button widget is used to display buttons in your application.
Canvas	The Canvas widget is used to draw shapes, such as lines, ovals, polygons and rectangles, in your application.
Checkbutton	The Checkbutton widget is used to display a number of options as checkboxes. The user can select multiple options at a time.

Entry	The Entry widget is used to display a single-line text field for accepting values from a user.
Frame	The Frame widget is used as a container widget to organize other widgets.

Label	The Label widget is used to provide a single-line caption for other widgets. It can also contain images.
Listbox	The Listbox widget is used to provide a list of options to a user.
Menubutton	The Menubutton widget is used to display menus in your application.
Menu	The Menu widget is used to provide various commands to a user. These commands are contained inside Menubutton.
Message	The Message widget is used to display multiline text fields for accepting values from a user.
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.
Scale	The Scale widget is used to provide a slider widget.
Scrollbar	The Scrollbar widget is used to add scrolling capability to various widgets, such as list boxes.
Text	The Text widget is used to display text in multiple lines.
Toplevel	The Toplevel widget is used to provide a separate window container.
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.
PanedWindow	A PanedWindow is a container widget that may contain any number of panes, arranged horizontally or vertically.

LabelFrame	A labelframe is a simple container widget. Its primary purpose is to act as a spacer or container for complex window layouts.
tkMessageBox	This module is used to display message boxes in your applications.

Let us study these widgets in detail:

1. Button

The Button widget is used to add buttons in a Python application. These buttons can display text or images that convey the purpose of the buttons. You can attach a function or a method to a button which is called automatically when you click the button.

Syntax

Here is the simple syntax to create this widget:

```
w = Button ( master, option=value, ... )
```

Parameters

master: This represents the parent window.

options: Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Option	Description
activebackground	Background color when the button is under the cursor.
activeforeground	Foreground color when the button is under the cursor.
bd	Border width in pixels. Default is 2.

bg	Normal background color.
command	Function or method to be called when the button is clicked.
fg	Normal foreground (text) color.
font	Text font to be used for the button's label.
height	Height of the button in text lines (for textual buttons) or pixels (for images).
highlightcolor	The color of the focus highlight when the widget has focus.
image	Image to be displayed on the button (instead of text).
justify	How to show multiple text lines: LEFT to left-justify each line; CENTER to center them; or RIGHT to right-justify.
padx	Additional padding left and right of the text.
pady	Additional padding above and below the text.
relief	Relief specifies the type of the border. Some of the values are SUNKEN, RAISED, GROOVE, and RIDGE.
state	Set this option to DISABLED to gray out the button and make it unresponsive. Has the value ACTIVE when the mouse is over it. Default is NORMAL.
underline	Default is -1, meaning that no character of the text on the button will be underlined. If nonnegative, the corresponding text character will be underlined.

width	Width of the button in letters (if displaying text) or pixels (if displaying an image).
wraplength	If this value is set to a positive number, the text lines will be wrapped to fit within this length.

Methods

Following are commonly used methods for this widget:

Medthod	Description
flash()	Causes the button to flash several times between active and normal colors. Leaves the button in the state it was in originally. Ignored if the button is disabled.
invoke()	Calls the button's callback, and returns what that function returns. Has no effect if the button is disabled or there is no callback.

Example

Try the following example yourself:

```
import Tkinter import tkMessageBox top = Tkinter.Tk()

 def helloCallBack():

   tkMessageBox.showinfo( "Hello Python", "Hello World")

B = Tkinter.Button(top, text ="Hello", command = helloCallBack)




B.pack() top.mainloop()
```

When the above code is executed, it produces the following result:
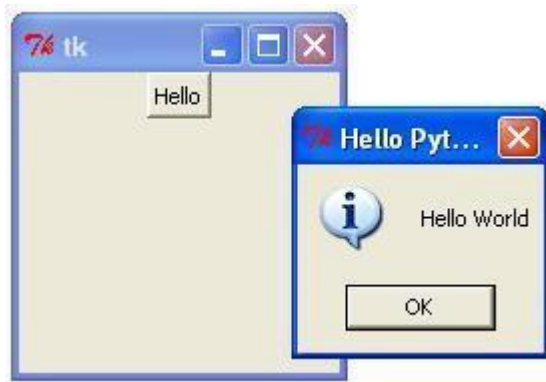
2. Canvas

The Canvas is a rectangular area intended for drawing pictures or other complex layouts. You can place graphics, text, widgets2. or frames on a Canvas. **Syntax**

Here is the simple syntax to create this widget:

```
w = Canvas ( master, option=value, ... )
```

Parameters

master: This represents the parent window.

options: Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Option	Description
bd	Border width in pixels. Default is 2.
bg	Normal background color.
confine	If true (the default), the canvas cannot be scrolled outside of the scrollregion.
cursor	Cursor used in the canvas like *arrow, circle, dot etc.*

height	Size of the canvas in the Y dimension.
highlightcolor	Color shown in the focus highlight.
relief	Relief specifies the type of the border. Some of the values are SUNKEN, RAISED, GROOVE, and RIDGE.
scrollregion	A tuple (w, n, e, s) that defines over how large an area the canvas can be scrolled, where w is the left side, n the top, e the right side, and s the bottom.
width	Size of the canvas in the X dimension.
xscrollincrement	If you set this option to some positive dimension, the canvas can be positioned only on multiples of that distance, and the value will be used for scrolling by scrolling units, such as when the user clicks on the arrows at the ends of a scrollbar.
xscrollcommand	If the canvas is scrollable, this attribute should be the .set() method of the horizontal scrollbar.
yscrollincrement	Works like xscrollincrement, but governs vertical movement.
yscrollcommand	If the canvas is scrollable, this attribute should be the .set() method of the vertical scrollbar.

The Canvas widget can support the following standard items: arc . Creates an arc item, which can be a chord, a pieslice or a simple arc.

```
coord = 10, 50, 240, 210   arc = canvas.create_arc(coord, start=0, extent=150,
fill="blue")
```

image . Creates an image item, which can be an in stance of either the BitmapImage or the PhotoImage classes.

```
filename = PhotoImage(file = "sunshine.gif")

image = canvas.create_image(50, 50, anchor=NE,  image=filename)
```

line . Creates a line item.

```
line = canvas.create_line(x0, y0, x1, y1, ...,  xn, yn, options)
```

oval . Creates a circle or an ellipse at the given coordinates. It takes two pairs of coordinates; the top left and bottom right corners of the bounding rectangle for the oval.

```
oval = canvas.create_oval(x0, y0, x1, y1, options)
```

polygon . Creates a polygon item that must have at least three vertices.

```
oval = canvas.create_polygon(x0, y0, x1, y1,...xn, yn, options)
```

Example

Try the following example yourself:

```
import Tkinter import tkMessageBox

 top = Tkinter.Tk()

C = Tkinter.Canvas(top, bg="blue", height=250, width=300)

 coord = 10,  50,  240,  210  arc = C.create_arc(coord,  start=0,  extent=150,
fill="red")

C.pack() top.mainloop()
```

When the above code is executed, it produces the following result:
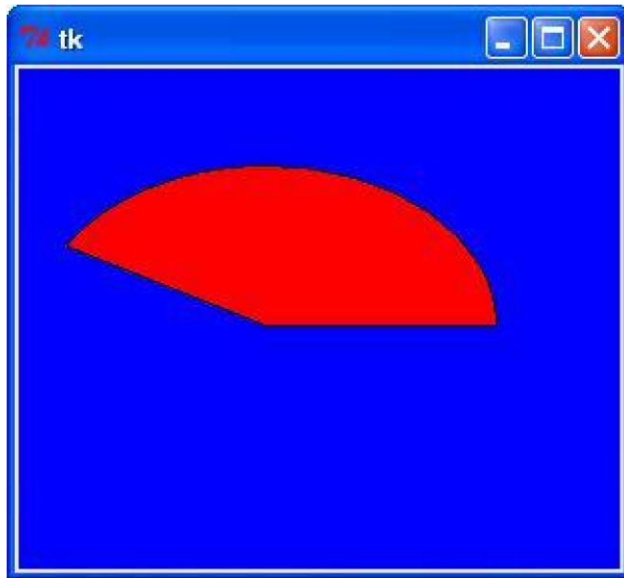
3. Checkbutton

The Checkbutton widget is used to display a number of options to a user as toggle buttons. The user can then select one or more options by clicking the button corresponding to each option.

You can also display images in place of text.

Syntax

Here is the simple syntax to create this widget:

```
w = Checkbutton ( master, option, ... )
```

Parameters

master: This represents the parent window.

options: Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Option	Description
activebackground	Background color when the checkbutton is under the cursor.

activeforeground	Foreground color when the checkbutton is under the cursor.
bg	The normal background color displayed behind the label and indicator.
bitmap	To display a monochrome image on a button.
bd	The size of the border around the indicator. Default is 2 pixels.
command	A procedure to be called every time the user changes the state of this checkbutton.
cursor	If you set this option to a cursor name (*arrow, dot etc.), the mouse cursor will change to that pattern when it is over the checkbutton.*
disabledforeground	The foreground color used to render the text of a disabled checkbutton. The default is a stippled version of the default foreground color.
font	The font used for the text.
fg	The color used to render the text.
height	The number of lines of text on the checkbutton. Default is 1.
highlightcolor	The color of the focus highlight when the checkbutton has the focus.
image	To display a graphic image on the button.

justify	If the text contains multiple lines, this option controls how the text is justified: CENTER, LEFT, or RIGHT.
offvalue	Normally, a checkbutton's associated control variable will be set to 0 when it is cleared (off). You can supply an alternate value for the off state by setting offvalue to that value.
onvalue	Normally, a checkbutton's associated control variable will be set to 1 when it is set (on). You can supply an alternate value for the on state by setting onvalue to that value.
padx	How much space to leave to the left and right of the checkbutton and text. Default is 1 pixel.
pady	How much space to leave above and below the checkbutton and text. Default is 1 pixel.
relief	With the default value, relief=FLAT, the checkbutton does not stand out from its background. You may set this option to any of the other styles
selectcolor	The color of the checkbutton when it is set. Default is selectcolor="red".
selectimage	If you set this option to an image, that image will appear in the checkbutton when it is set.
state	The default is state=NORMAL, but you can use state=DISABLED to gray out the control and make it unresponsive. If the cursor is currently over the checkbutton, the state is ACTIVE.
text	The label displayed next to the checkbutton. Use newlines ("\n") to display multiple lines of text.

underline	With the default value of -1, none of the characters of the text label are underlined. Set this option to the index of a character in the text (counting from zero) to underline that character.
variable	The control variable that tracks the current state of the checkbutton. Normally this variable is an *IntVar*, and 0 means cleared and 1 means set, but see the offvalue and onvalue options above.
width	The default width of a checkbutton is determined by the size of the displayed image or text. You can set this option to a number of characters and the checkbutton will always have room for that many characters.
wraplength	Normally, lines are not wrapped. You can set this option to a number of characters and all lines will be broken into pieces no longer than that number.

Methods

Following are commonly used methods for this widget:

Medthod	Description
deselect()	Clears (turns off) the checkbutton.
flash()	Flashes the checkbutton a few times between its active and normal colors, but leaves it the way it started.
invoke()	You can call this method to get the same actions that would occur if the user clicked on the checkbutton to change its state.
select()	Sets (turns on) the checkbutton.
toggle()	Clears the checkbutton if set, sets it if cleared.

Example

Try the following example yourself:

```
from Tkinter import *
```

```
import tkMessageBox import Tkinter

 top = Tkinter.Tk()

CheckVar1 = IntVar()

CheckVar2 = IntVar()

C1  =  Checkbutton(top,  text  =  "Music",  variable  =  CheckVar1,  \
onvalue = 1, offvalue = 0, height=5, \                    width = 20)

C2  =  Checkbutton(top,  text  =  "Video",  variable  =  CheckVar2,  \
onvalue = 1, offvalue = 0, height=5, \                    width = 20)

C1.pack() C2.pack() top.mainloop()
```

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

4. Entry

The Entry widget is used to accept single-line text strings from a user.

If you want to display multiple lines of text that can be edited, then you should use the *Text* widget.

If you want to display one or more lines of text that cannot be modified by the user, then you should use the *Label* widget.

Syntax

Here is the simple syntax to create this widget:

```
w = Entry( master, option, ... )
```

Parameters

☐ **master:** This represents the parent window.

options: Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Option	Description
bg	The normal background color displayed behind the label and indicator.
bd	The size of the border around the indicator. Default is 2 pixels.
command	A procedure to be called every time the user changes the state of this checkbutton.
cursor	If you set this option to a cursor name (*arrow, dot etc.*), the mouse cursor will change to that pattern when it is over the checkbutton.
font	The font used for the text.
exportselection	By default, if you select text within an Entry widget, it is automatically exported to the clipboard. To avoid this exportation, use exportselection=0.
fg	The color used to render the text.

highlightcolor	The color of the focus highlight when the checkbutton has the focus.
justify	If the text contains multiple lines, this option controls how the text is justified: CENTER, LEFT, or RIGHT.
relief	With the default value, relief=FLAT, the checkbutton does not stand out from its background. You may set this option to any of the other styles
selectbackground	The background color to use displaying selected text.
selectborderwidth	The width of the border to use around selected text. The default is one pixel.
selectforeground	The foreground (text) color of selected text.
show	Normally, the characters that the user types appear in the entry. To make a .password. entry that echoes each character as an asterisk, set show="*".
state	The default is state=NORMAL, but you can use state=DISABLED to gray out the control and make it unresponsive. If the cursor is currently over the checkbutton, the state is ACTIVE.
textvariable	In order to be able to retrieve the current text from your entry widget, you must set this option to an instance of the StringVar class.
width	The default width of a checkbutton is determined by the size of the displayed image or text. You can set this option to a number of characters and the checkbutton will always have room for that many characters.

| xscrollcommand | If you expect that users will often enter more text than the onscreen size of the widget, you can link your entry widget to a scrollbar. |

Methods

Following are commonly used methods for this widget:

Medthod	Description
delete (first, last=None)	Deletes characters from the widget, starting with the one at index first, up to but not including the character at position last. If the second argument is omitted, only the single character at position first is deleted.
get()	Returns the entry's current text as a string.
icursor (index)	Set the insertion cursor just before the character at the given index.
index (index)	Shift the contents of the entry so that the character at the given index is the leftmost visible character. Has no effect if the text fits entirely within the entry.
insert (index, s)	Inserts string s before the character at the given index.
select_adjust (index)	This method is used to make sure that the selection includes the character at the specified index.
select_clear()	Clears the selection. If there isn't currently a selection, has no effect.
select_from (index)	Sets the ANCHOR index position to the character selected by index, and selects that character.
select_present()	If there is a selection, returns true, else returns false.

select_range (start, end)	Sets the selection under program control. Selects the text starting at the start index, up to but not including the character at the end index. The start position must be before the end position.
select_to (index)	Selects all the text from the ANCHOR position up to but not including the character at the given index.
xview (index)	This method is useful in linking the Entry widget to a horizontal scrollbar.
xview_scroll (number, what)	Used to scroll the entry horizontally. The what argument must be either UNITS, to scroll by character widths, or PAGES, to scroll by chunks the size of the entry widget. The number is positive to scroll left to right, negative to scroll right to left.

Example

Try the following example yourself:

```python
from Tkinter import *

 top = Tk()

L1 = Label(top, text="User Name")

L1.pack( side = LEFT)

E1 = Entry(top, bd =5)



E1.pack(side = RIGHT)

 top.mainloop()
```

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

5. Frame

The Frame widget is very important for the process of grouping and organizing other widgets in a somehow friendly way. It works like a container, which is responsible for arranging the position of other widgets.

It uses rectangular areas in the screen to organize the layout and to provide padding of these widgets. A frame can also be used as a foundation class to implement complex widgets.

Syntax

Here is the simple syntax to create this widget:

```
w = Frame ( master, option, ... )
```

Parameters:

master: This represents the parent window.

options: Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Option	Description
bg	The normal background color displayed behind the label and indicator.
bd	The size of the border around the indicator. Default is 2 pixels.
cursor	If you set this option to a cursor name (*arrow, dot etc.*), the mouse cursor will change to that pattern when it is over the checkbutton.
height	The vertical dimension of the new frame.
highlightbackground	Color of the focus highlight when the frame does not have focus.

highlightcolor	Color shown in the focus highlight when the frame has the focus.
highlightthickness	Thickness of the focus highlight.
relief	With the default value, relief=FLAT, the checkbutton does not stand out from its background. You may set this option to any of the other styles
width	The default width of a checkbutton is determined by the size of the displayed image or text. You can set this option to a number of characters and the checkbutton will always have room for that many characters.

Example

Try the following example yourself:

```
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()
```

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

6. Label

This widget implements a display box where you can place text or images. The text displayed by this widget can be updated at any time you want.

It is also possible to underline part of the text (like to identify a keyboard shortcut) and span the text across multiple lines.

Syntax

Here is the simple syntax to create this widget:

```
w = Label ( master, option, ... )
```

Parameters

master: This represents the parent window.

options: Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Option	Description
anchor	This options controls where the text is positioned if the widget has more space than the text needs. The default is anchor=CENTER, which centers the text in the available space.
bg	The normal background color displayed behind the label and indicator.
bitmap	Set this option equal to a bitmap or image object and the label will display that graphic.
bd	The size of the border around the indicator. Default is 2 pixels.

cursor	If you set this option to a cursor name (*arrow, dot etc.*), the mouse cursor will change to that pattern when it is over the checkbutton.
font	If you are displaying text in this label (with the text or textvariable option, the font option specifies in what font that text will be displayed.
fg	If you are displaying text or a bitmap in this label, this option specifies the color of the text. If you are displaying a bitmap, this is the color that will appear at the position of the 1-bits in the bitmap.
height	The vertical dimension of the new frame.
image	To display a static image in the label widget, set this option to an image object.
justify	Specifies how multiple lines of text will be aligned with respect to each other: LEFT for flush left, CENTER for centered (the default), or RIGHT for right-justified.
padx	Extra space added to the left and right of the text within the widget. Default is 1.
pady	Extra space added above and below the text within the widget. Default is 1.
relief	Specifies the appearance of a decorative border around the label. The default is FLAT; for other values.
text	To display one or more lines of text in a label widget, set this option to a string containing the text. Internal newlines ("\n") will force a line break.

textvariable	To slave the text displayed in a label widget to a control variable of class *StringVar*, set this option to that variable.
underline	You can display an underline (_) below the nth letter of the text, counting from 0, by setting this option to n. The default is underline=-1, which means no underlining.
width	Width of the label in characters (not pixels!). If this option is not set, the label will be sized to fit its contents.
wraplength	You can limit the number of characters in each line by setting this option to the desired number. The default value, 0, means that lines will be broken only at newlines.

Example

Try the following example yourself:

```
from Tkinter import *

 root = Tk()

 var = StringVar() label = Label( root, textvariable=var, relief=RAISED )

 var.set("Hey!? How are you doing?") label.pack() root.mainloop()
```

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

7. Listbox

The Listbox widget is used to display a list of items from which a user can select a number of items

Syntax

Here is the simple syntax to create this widget:

```
w = Listbox ( master, option, ... )
```

Parameters

master: This represents the parent window.

options: Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Option	Description
bg	The normal background color displayed behind the label and indicator.
bd	The size of the border around the indicator. Default is 2 pixels.
cursor	The cursor that appears when the mouse is over the listbox.
font	The font used for the text in the listbox.
fg	The color used for the text in the listbox.
height	Number of lines (not pixels!) shown in the listbox. Default is 10.
highlightcolor	Color shown in the focus highlight when the widget has the focus.
highlightthickness	Thickness of the focus highlight.
relief	Selects three-dimensional border shading effects. The default is SUNKEN.

selectbackground	The background color to use displaying selected text.
selectmode	Determines how many items can be selected, and how mouse drags affect the selection: BROWSE: Normally, you can only select one line out of a listbox. If you click on an item and then drag to a different line, the selection will follow the mouse. This is the default.
	SINGLE: You can only select one line, and you can't drag the mouse.wherever you click button 1, that line is selected. MULTIPLE: You can select any number of lines at once. Clicking on any line toggles whether or not it is selected. EXTENDED: You can select any adjacent group of lines at once by clicking on the first line and dragging to the last line.
width	The width of the widget in characters. The default is 20.
xscrollcommand	If you want to allow the user to scroll the listbox horizontally, you can link your listbox widget to a horizontal scrollbar.
yscrollcommand	If you want to allow the user to scroll the listbox vertically, you can link your listbox widget to a vertical scrollbar.

Methods

Methods on listbox objects include:

Option	Description
activate (index)	Selects the line specifies by the given index.

curselection()	Returns a tuple containing the line numbers of the selected element or elements, counting from 0. If nothing is selected, returns an empty tuple.
delete (first, last=None)	Deletes the lines whose indices are in the range [first, last]. If the second argument is omitted, the single line with index first is deleted.
get (first, last=None)	Returns a tuple containing the text of the lines with indices from first to last, inclusive. If the second argument is omitted, returns the text of the line closest to first.
index (i)	If possible, positions the visible part of the listbox so that the line containing index i is at the top of the widget.
insert (index, *elements)	Insert one or more new lines into the listbox before the line specified by index. Use END as the first argument if you want to add new lines to the end of the listbox.
nearest (y)	Return the index of the visible line closest to the ycoordinate y relative to the listbox widget.
see (index)	Adjust the position of the listbox so that the line referred to by index is visible.
size()	Returns the number of lines in the listbox.
xview()	To make the listbox horizontally scrollable, set the command option of the associated horizontal scrollbar to this method.

xview_moveto (fraction)	Scroll the listbox so that the leftmost fraction of the width of its longest line is outside the left side of the listbox. Fraction is in the range [0,1].
xview_scroll (number, what)	Scrolls the listbox horizontally. For the what argument, use either UNITS to scroll by characters, or PAGES to scroll by pages, that is, by the width of the listbox. The number argument tells how many to scroll.
yview()	To make the listbox vertically scrollable, set the command option of the associated vertical scrollbar to this method.
yview_moveto (fraction)	Scroll the listbox so that the top fraction of the width of its longest line is outside the left side of the listbox. Fraction is in the range [0,1].
yview_scroll (number, what)	Scrolls the listbox vertically. For the what argument, use either UNITS to scroll by lines, or PAGES to scroll by pages, that is, by the height of the listbox.\n\nThe number argument tells how many to scroll.

Example

Try the following example yourself:

```
from Tkinter import * import tkMessageBox import Tkinter

 top = Tk()



Lb1 = Listbox(top)

Lb1.insert(1, "Python")

Lb1.insert(2, "Perl")
```

```
Lb1.insert(3, "C")

Lb1.insert(4, "PHP")

Lb1.insert(5, "JSP")

Lb1.insert(6, "Ruby")



Lb1.pack() top.mainloop()
```

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

8. MenuButton

A menubutton is the part of a drop-down menu that stays on the screen all the time. Every menubutton is associated with a Menu widget that can display the choices for that menubutton when the user clicks on it.

Syntax

Here is the simple syntax to create this widget:

```
w = Menubutton ( master, option, ... )
```

Parameters

master: This represents the parent window.

options: Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Option	Description
activebackground	The background color when the mouse is over the menubutton.
activeforeground	The foreground color when the mouse is over the menubutton.
anchor	This options controls where the text is positioned if the widget has more space than the text needs. The default is anchor=CENTER, which centers the text.
bg	The normal background color displayed behind the label and indicator.
bitmap	To display a bitmap on the menubutton, set this option to a bitmap name.
bd	The size of the border around the indicator. Default is 2 pixels.
cursor	The cursor that appears when the mouse is over this menubutton.
direction	Set direction=LEFT to display the menu to the left of the button; use direction=RIGHT to display the menu to the right of the button; or use direction='above' to place the menu above the button.
disabledforeground	The foreground color shown on this menubutton when it is disabled.
fg	The foreground color when the mouse is not over the menubutton.

height	The height of the menubutton in lines of text (not pixels!). The default is to fit the menubutton's size to its contents.
highlightcolor	Color shown in the focus highlight when the widget has the focus.
image	To display an image on this menubutton,
justify	This option controls where the text is located when the text doesn't fill the menubutton: use justify=LEFT to left-justify the text (this is the default); use justify=CENTER to center it, or justify=RIGHT to right-justify.
menu	To associate the menubutton with a set of choices, set this option to the Menu object containing those choices. That menu object must have been created by passing the associated menubutton to the constructor as its first argument.
padx	How much space to leave to the left and right of the text of the menubutton. Default is 1.
pady	How much space to leave above and below the text of the menubutton. Default is 1.
relief	Selects three-dimensional border shading effects. The default is RAISED.
state	Normally, menubuttons respond to the mouse. Set state=DISABLED to gray out the menubutton and make it unresponsive.
text	To display text on the menubutton, set this option to the string containing the desired text. Newlines ("\n") within the string will cause line breaks.

textvariable	You can associate a control variable of class StringVar with this menubutton. Setting that control variable will change the displayed text.
underline	Normally, no underline appears under the text on the menubutton. To underline one of the characters, set this option to the index of that character.
width	The width of the widget in characters. The default is 20.
wraplength	Normally, lines are not wrapped. You can set this option to a number of characters and all lines will be broken into pieces no longer than that number.

Example

Try the following example yourself:

```
from Tkinter import * import tkMessageBox import Tkinter

 top = Tk()
```

```
mb=  Menubutton ( top, text="condiments", relief=RAISED ) mb.grid() mb.menu   =
Menu ( mb, tearoff = 0 ) mb["menu"]  =  mb.menu

     mayoVar  = IntVar() ketchVar = IntVar()

 mb.menu.add_checkbutton ( label="mayo",                          variable=mayoVar
)            mb.menu.add_checkbutton            (            label="ketchup",
variable=ketchVar )

 mb.pack() top.mainloop()
```

When the above code is executed, it produces the following result:
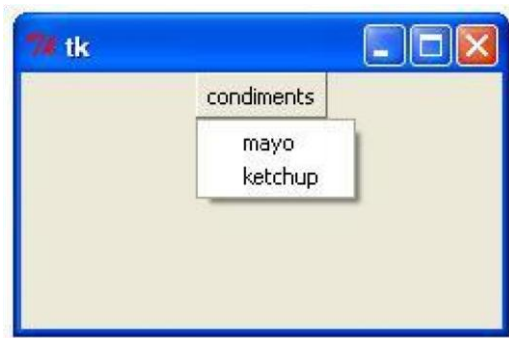
9. Menu

The goal of this widget is to allow us to create all kinds of menus that can be used by our applications. The core functionality provides ways to create three menu types: pop-up, toplevel and pull-down.

It is also possible to use other extended widgets to implement new types of menus, such as the *OptionMenu* widget, which implements a special type that generates a pop-up list of items within a selection.

Syntax

Here is the simple syntax to create this widget:

```
w = Menu ( master, option, ... )
```

Parameters

master: This represents the parent window.

options: Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Option	Description
activebackground	The background color that will appear on a choice when it is under the mouse.
activeborderwidth	Specifies the width of a border drawn around a choice when it is under the mouse. Default is 1 pixel.

activeforeground	The foreground color that will appear on a choice when it is under the mouse.
bg	The background color for choices not under the mouse.
bd	The width of the border around all the choices. Default is 1.
cursor	The cursor that appears when the mouse is over the choices, but only when the menu has been torn off.
disabledforeground	The color of the text for items whose state is DISABLED.
font	The default font for textual choices.
fg	The foreground color used for choices not under the mouse.
postcommand	You can set this option to a procedure, and that procedure will be called every time someone brings up this menu.
relief	The default 3-D effect for menus is relief=RAISED.
image	To display an image on this menubutton.
selectcolor	Specifies the color displayed in checkbuttons and radiobuttons when they are selected.
tearoff	Normally, a menu can be torn off, the first position (position 0) in the list of choices is occupied by the tear-off element, and the additional choices are added starting at position 1. If you set tearoff=0, the menu will not have a tear-off feature, and choices will be added starting at position 0.

title	Normally, the title of a tear-off menu window will be the same as the text of the menubutton or cascade that lead to this menu. If you want to change the title of that window, set the title option to that string.

Methods

These methods are available on Menu objects:

Option	Description
add_command (options)	Adds a menu item to the menu.
add_radiobutton(options)	Creates a radio button menu item.
add_checkbutton(options)	Creates a check button menu item.
add_cascade(options)	Creates a new hierarchical menu by associating a given menu to a parent menu
add_separator()	Adds a separator line to the menu.
add(type, options)	Adds a specific type of menu item to the menu.
delete(startindex [, endindex])	Deletes the menu items ranging from startindex to endindex.
entryconfig(index, options)	Allows you to modify a menu item, which is identified by the index, and change its options.
index(item)	Returns the index number of the given menu item label.

insert_separator (index)	Insert a new separator at the position specified by index.
invoke (index)	Calls the command callback associated with the choice at position index. If a checkbutton, its state is toggled between set and cleared; if a radiobutton, that choice is set.
type (index)	Returns the type of the choice specified by index: either "cascade", "checkbutton", "command", "radiobutton", "separator", or "tearoff".

Example

Try the following example yourself:

```
from Tkinter import * def donothing():

    filewin = Toplevel(root)        button = Button(filewin, text="Do nothing
button")     button.pack()

    root = Tk() menubar = Menu(root) filemenu = Menu(menubar, tearoff=0)
filemenu.add_command(label="New", command=donothing)
```

```
filemenu.add_command(label="Open",                              command=donothing)
filemenu.add_command(label="Save",                              command=donothing)
filemenu.add_command(label="Save             as...",            command=donothing)
filemenu.add_command(label="Close", command=donothing)

 filemenu.add_separator()

 filemenu.add_command(label="Exit",                             command=root.quit)
menubar.add_cascade(label="File",  menu=filemenu) editmenu  =  Menu(menubar,
tearoff=0) editmenu.add_command(label="Undo", command=donothing)

 editmenu.add_separator()

 editmenu.add_command(label="Cut",                              command=donothing)
editmenu.add_command(label="Copy",                             command=donothing)
editmenu.add_command(label="Paste",                            command=donothing)
```

```
editmenu.add_command(label="Delete",                        command=donothing)
editmenu.add_command(label="Select All", command=donothing)

 menubar.add_cascade(label="Edit", menu=editmenu) helpmenu = Menu(menubar,
tearoff=0)    helpmenu.add_command(label="Help    Index",   command=donothing)
helpmenu.add_command(label="About...",                      command=donothing)
menubar.add_cascade(label="Help", menu=helpmenu)
```

root.config(menu=menubar) root.mainloop()

When the above code is executed, it produces the following result:
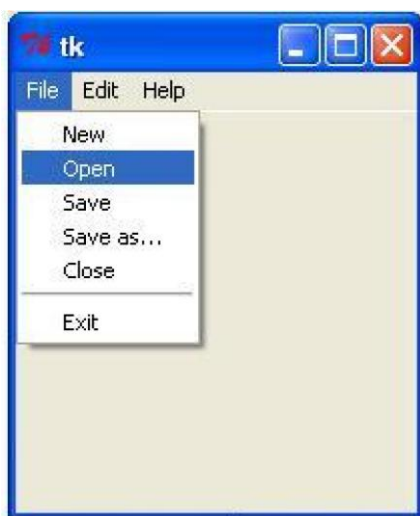
10.   Message

This widget provides a multiline and noneditable object that displays texts, automatically breaking lines and justifying their contents.

Its functionality is very similar to the one provided by the Label widget, except that it can also automatically wrap the text, maintaining a given width or aspect ratio.

Syntax

Here is the simple syntax to create this widget:

```
 w = Message ( master, option, ... )
```

Parameters

11.**master:** This represents the parent window.

12.**options:** Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Option	Description
anchor	This options controls where the text is positioned if the widget has more space than the text needs. The default is anchor=CENTER, which centers the text in the available space.
bg	The normal background color displayed behind the label and indicator.
bitmap	Set this option equal to a bitmap or image object and the label will display that graphic.
bd	The size of the border around the indicator. Default is 2 pixels.
cursor	If you set this option to a cursor name (*arrow, dot etc.*), the mouse cursor will change to that pattern when it is over the checkbutton.
font	If you are displaying text in this label (with the text or textvariable option, the font option specifies in what font that text will be displayed.
fg	If you are displaying text or a bitmap in this label, this option specifies the color of the text. If you are displaying a bitmap, this is the color that will appear at the position of the 1-bits in the bitmap.
height	The vertical dimension of the new frame.
image	To display a static image in the label widget, set this option to an image object.

justify	Specifies how multiple lines of text will be aligned with respect to each other: LEFT for flush left, CENTER for centered (the default), or RIGHT for right-justified.
padx	Extra space added to the left and right of the text within the widget. Default is 1.
pady	Extra space added above and below the text within the widget. Default is 1.
relief	Specifies the appearance of a decorative border around the label. The default is FLAT; for other values.
text	To display one or more lines of text in a label widget, set this option to a string containing the text. Internal newlines ("\n") will force a line break.
textvariable	To slave the text displayed in a label widget to a control variable of class *StringVar*, set this option to that variable.
underline	You can display an underline (_) below the nth letter of the text, counting from 0, by setting this option to n. The default is underline=-1, which means no underlining.
width	Width of the label in characters (not pixels!). If this option is not set, the label will be sized to fit its contents.
wraplength	You can limit the number of characters in each line by setting this option to the desired number. The default value, 0, means that lines will be broken only at newlines.

Example

Try the following example yourself:

```
from Tkinter import *

root = Tk()

var = StringVar() label = Message( root, textvariable=var, relief=RAISED )

var.set("Hey!? How are you doing?") label.pack() root.mainloop()
```

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

11. Radiobutton

This widget implements a multiple-choice button, which is a way to offer many possible selections to the user and lets user choose only one of them.

In order to implement this functionality, each group of radiobuttons must be associated to the same variable and each one of the buttons must symbolize a single value. You can use the Tab key to switch from one radionbutton to another. **Syntax**

Here is the simple syntax to create this widget:

```
w = Radiobutton ( master, option, ...  )
```

Parameters

master: This represents the parent window.

options: Here is the list of most commonly used options for this widget.

These options can be used as key-value pairs separated by commas.

Option	Description
activebackground	The background color when the mouse is over the radiobutton.

activeforeground	The foreground color when the mouse is over the radiobutton.
anchor	If the widget inhabits a space larger than it needs, this option specifies where the radiobutton will sit in that space. The default is anchor=CENTER.
bg	The normal background color behind the indicator and label.
bitmap	To display a monochrome image on a radiobutton, set this option to a bitmap.
borderwidth	The size of the border around the indicator part itself. Default is 2 pixels.
command	A procedure to be called every time the user changes the state of this radiobutton.
cursor	If you set this option to a cursor name (*arrow, dot etc.*), the mouse cursor will change to that pattern when it is over the radiobutton.
font	The font used for the text.
fg	The color used to render the text.
height	The number of lines (not pixels) of text on the radiobutton. Default is 1.
highlightbackground	The color of the focus highlight when the radiobutton does not have focus.
highlightcolor	The color of the focus highlight when the radiobutton has the focus.

image	To display a graphic image instead of text for this radiobutton, set this option to an image object.
justify	If the text contains multiple lines, this option controls how the text is justified: CENTER (the default), LEFT, or RIGHT.
padx	How much space to leave to the left and right of the radiobutton and text. Default is 1.
pady	How much space to leave above and below the radiobutton and text. Default is 1.
relief	Specifies the appearance of a decorative border around the label. The default is FLAT; for other values.
selectcolor	The color of the radiobutton when it is set. Default is red.
selectimage	If you are using the image option to display a graphic instead of text when the radiobutton is cleared, you can set the selectimage option to a different image that will be displayed when the radiobutton is set.
state	The default is state=NORMAL, but you can set state=DISABLED to gray out the control and make it unresponsive. If the cursor is currently over the radiobutton, the state is ACTIVE.
text	The label displayed next to the radiobutton. Use newlines ("\n") to display multiple lines of text.
textvariable	To slave the text displayed in a label widget to a control variable of class *StringVar*, set this option to that variable.

underline	You can display an underline (_) below the nth letter of the text, counting from 0, by setting this option to n. The default is underline=-1, which means no underlining.
value	When a radiobutton is turned on by the user, its control variable is set to its current value option. If the control variable is an *IntVar*, give each radiobutton in the group a different integer value option. If the control variable is a *StringVar*, give each radiobutton a different string value option.
variable	The control variable that this radiobutton shares with the other radiobuttons in the group. This can be either an IntVar or a StringVar.
width	Width of the label in characters (not pixels!). If this option is not set, the label will be sized to fit its contents.
wraplength	You can limit the number of characters in each line by setting this option to the desired number. The default value, 0, means that lines will be broken only at newlines.

Methods

Methods	Description
deselect()	Clears (turns off) the radiobutton.
flash()	Flashes the radiobutton a few times between its active and normal colors, but leaves it the way it started.
invoke()	You can call this method to get the same actions that would occur if the user clicked on the radiobutton to change its state.
select()	Sets (turns on) the radiobutton.

Example:

Try the following example yourself:

```
from Tkinter import *

 def sel():

   selection = "You selected the option " + str(var.get())    label.config(text
= selection)

 root = Tk() var = IntVar()

R1   =   Radiobutton(root,   text="Option   1",   variable=var,   value=1,
command=sel)

R1.pack( anchor = W )




R2   =   Radiobutton(root,   text="Option   2",   variable=var,   value=2,
command=sel)

R2.pack( anchor = W )
```
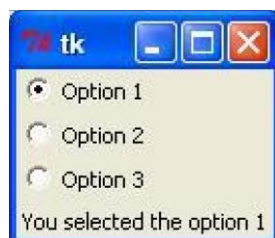
```
R3   =   Radiobutton(root,   text="Option   3",   variable=var,   value=3,
command=sel)

R3.pack( anchor = W)

 label = Label(root) label.pack() root.mainloop()
```

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

12. Scale

The Scale widget provides a graphical slider object that allows you to select values from a specific scale.

Syntax

Here is the simple syntax to create this widget:

```
w = Scale ( master, option, ... )
```

Parameters

master: This represents the parent window.

options: Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Option	Description
activebackground	The background color when the mouse is over the scale.
bg	The background color of the parts of the widget that are outside the trough.
bd	Width of the 3-d border around the trough and slider. Default is 2 pixels.
command	A procedure to be called every time the slider is moved. This procedure will be passed one argument, the new scale value. If the slider is moved rapidly, you may not get a callback for every possible position, but you'll certainly get a callback when it settles.
cursor	If you set this option to a cursor name (*arrow, dot etc.*), the mouse cursor will change to that pattern when it is over the scale.

digits	The way your program reads the current value shown in a scale widget is through a control variable. The control variable for a scale can be an IntVar, a DoubleVar (float), or a StringVar. If it is a string variable, the digits option controls how many digits to use when the numeric scale value is converted to a string.
font	The font used for the label and annotations.
fg	The color of the text used for the label and annotations.
from_	A float or integer value that defines one end of the scale's range.
highlightbackground	The color of the focus highlight when the scale does not have focus.
highlightcolor	The color of the focus highlight when the scale has the focus.
label	You can display a label within the scale widget by setting this option to the label's text. The label appears in the top left corner if the scale is horizontal, or the top right corner if vertical. The default is no label.
length	The length of the scale widget. This is the x dimension if the scale is horizontal, or the y dimension if vertical. The default is 100 pixels.
orient	Set orient=HORIZONTAL if you want the scale to run along the x dimension, or orient=VERTICAL to run parallel to the yaxis. Default is horizontal.
relief	Specifies the appearance of a decorative border around the label. The default is FLAT; for other values.

repeatdelay	This option controls how long button 1 has to be held down in the trough before the slider starts moving in that direction repeatedly. Default is repeatdelay=300, and the units are milliseconds.
resolution	Normally, the user will only be able to change the scale in whole units. Set this option to some other value to change the smallest increment of the scale's value. For example, if from_=-1.0 and to=1.0, and you set resolution=0.5, the scale will have 5 possible values: -1.0, -0.5, 0.0, +0.5, and +1.0.
showvalue	Normally, the current value of the scale is displayed in text form by the slider (above it for horizontal scales, to the left for vertical scales). Set this option to 0 to suppress that label.
sliderlength	Normally the slider is 30 pixels along the length of the scale. You can change that length by setting the sliderlength option to your desired length.
state	Normally, scale widgets respond to mouse events, and when they have the focus, also keyboard events. Set state=DISABLED to make the widget unresponsive.
takefocus	Normally, the focus will cycle through scale widgets. Set this option to 0 if you don't want this behavior.
tickinterval	To display periodic scale values, set this option to a number, and ticks will be displayed on multiples of that value. For example, if from_=0.0, to=1.0, and tickinterval=0.25, labels will be displayed along the scale at values 0.0, 0.25, 0.50,
	0.75, and 1.00. These labels appear below the scale if horizontal, to its left if vertical. Default is 0, which suppresses display of ticks.

to	A float or integer value that defines one end of the scale's range; the other end is defined by the from_ option, discussed above. The to value can be either greater than or less than the from_ value. For vertical scales, the to value defines the bottom of the scale; for horizontal scales, the right end.
troughcolor	The color of the trough.
variable	The control variable for this scale, if any. Control variables may be from class IntVar, DoubleVar (float), or StringVar. In the latter case, the numerical value will be converted to a string.
width	The width of the trough part of the widget. This is the x dimension for vertical scales and the y dimension if the scale has orient=HORIZONTAL. Default is 15 pixels.

Methods

Scale objects have these methods:

Methods	Description
get()	This method returns the current value of the scale.
set (value)	Sets the scale's value.

Example

Try the following example yourself:

```
from Tkinter import *

 def sel():

    selection = "Value = " + str(var.get())
```

```
   label.config(text = selection)

root = Tk() var = DoubleVar() scale = Scale( root, variable = var )
scale.pack(anchor=CENTER)

button    =    Button(root,    text="Get    Scale    Value",    command=sel)
button.pack(anchor=CENTER)

label = Label(root) label.pack()

root.mainloop()
```

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

13. Scrollbar

This widget provides a slide controller that is used to implement vertical scrolled widgets, such as Listbox, Text and Canvas. Note that you can also create horizontal scrollbars on Entry widgets.

Syntax:

Here is the simple syntax to create this widget:

```
w = Scrollbar ( master, option, ... )
```

Parameters

master: This represents the parent window.

options: Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Option	Description
activebackground	The color of the slider and arrowheads when the mouse is over them.
bg	The color of the slider and arrowheads when the mouse is not over them.
bd	The width of the 3-d borders around the entire perimeter of the trough, and also the width of the 3-d effects on the arrowheads and slider. Default is no border around the trough, and a 2-pixel border around the arrowheads and slider.
command	A procedure to be called whenever the scrollbar is moved.
cursor	The cursor that appears when the mouse is over the scrollbar.
elementborderwidth	The width of the borders around the arrowheads and slider. The default is elementborderwidth=-1, which means to use the value of the borderwidth option.
highlightbackground	The color of the focus highlight when the scrollbar does not have focus.
highlightcolor	The color of the focus highlight when the scrollbar has the focus.
highlightthickness	The thickness of the focus highlight. Default is 1. Set to 0 to suppress display of the focus highlight.

jump	This option controls what happens when a user drags the slider. Normally (jump=0), every small drag of the slider causes the command callback to be called. If you set this option to 1, the callback isn't called until the user releases the mouse button.
orient	Set orient=HORIZONTAL for a horizontal scrollbar, orient=VERTICAL for a vertical one.
repeatdelay	This option controls how long button 1 has to be held down in the trough before the slider starts moving in that direction repeatedly. Default is repeatdelay=300, and the units are milliseconds.
repeatinterval	repeatinterval
takefocus	Normally, you can tab the focus through a scrollbar widget. Set takefocus=0 if you don't want this behavior.
troughcolor	The color of the trough.
width	Width of the scrollbar (its y dimension if horizontal, and its x dimension if vertical). Default is 16.

Methods

Scrollbar objects have these methods:

Methods	Description
get()	Returns two numbers (a, b) describing the current position of the slider. The a value gives the position of the left or top edge of the slider, for horizontal and vertical scrollbars respectively; the b value gives the position of the right or bottom edge.

set (first, last)	To connect a scrollbar to another widget w, set w's xscrollcommand or yscrollcommand to the scrollbar's set() method. The arguments have the same meaning as the values returned by the get() method.

Example

Try the following example yourself:

```
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()
```
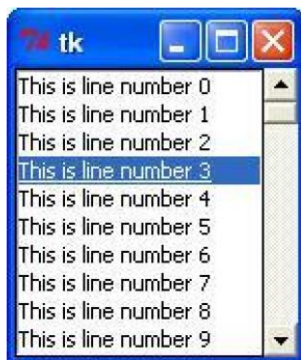
When the above code is executed, it produces the following result:

14. Text

Text widgets provide advanced capabilities that allow you to edit a multiline text and format the way it has to be displayed, such as changing its color and font.

You can also use elegant structures like tabs and marks to locate specific sections of the text, and apply changes to those areas. Moreover, you can embed windows and

images in the text because this widget was designed to handle both plain and formatted text.

Syntax

Here is the simple syntax to create this widget:

```
w = Text ( master, option, ... )
```

Parameters

master: This represents the parent window.

options: Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Option	Description
bg	The default background color of the text widget.
bd	The width of the border around the text widget. Default is 2 pixels.
cursor	The cursor that will appear when the mouse is over the text widget.
exportselection	Normally, text selected within a text widget is exported to be the selection in the window manager. Set exportselection=0 if you don't want that behavior.
font	The default font for text inserted into the widget.
fg	The color used for text (and bitmaps) within the widget. You can change the color for tagged regions; this option is just the default.

height	The height of the widget in lines (not pixels!), measured according to the current font size.
highlightbackground	The color of the focus highlight when the text widget does not have focus.
highlightcolor	The color of the focus highlight when the text widget has the focus.
highlightthickness	The thickness of the focus highlight. Default is 1. Set highlightthickness=0 to suppress display of the focus highlight.
insertbackground	The color of the insertion cursor. Default is black.
insertborderwidth	Size of the 3-D border around the insertion cursor. Default is 0.
insertofftime	The number of milliseconds the insertion cursor is off during its blink cycle. Set this option to zero to suppress blinking. Default is 300.
insertontime	The number of milliseconds the insertion cursor is on during its blink cycle. Default is 600.
insertwidth	Width of the insertion cursor (its height is determined by the tallest item in its line). Default is 2 pixels.
padx	The size of the internal padding added to the left and right of the text area. Default is one pixel.

pady	The size of the internal padding added above and below the text area. Default is one pixel.
relief	The 3-D appearance of the text widget. Default is relief=SUNKEN.
selectbackground	The background color to use displaying selected text.
selectborderwidth	The width of the border to use around selected text.
spacing1	This option specifies how much extra vertical space is put above each line of text. If a line wraps, this space is added only before the first line it occupies on the display. Default is 0.
spacing2	This option specifies how much extra vertical space to add between displayed lines of text when a logical line wraps. Default is 0.
spacing3	This option specifies how much extra vertical space is added below each line of text. If a line wraps, this space is added only after the last line it occupies on the display. Default is 0.
state	Normally, text widgets respond to keyboard and mouse events; set state=NORMAL to get this behavior. If you set state=DISABLED, the text widget will not respond, and you won't be able to modify its contents programmatically either.
tabs	This option controls how tab characters position text.
width	The width of the widget in characters (not pixels!), measured according to the current font size.

wrap	This option controls the display of lines that are too wide. Set wrap=WORD and it will break the line after the last word that will fit. With the default behavior, wrap=CHAR, any line that gets too long will be broken at any character.
xscrollcommand	To make the text widget horizontally scrollable, set this option to the set() method of the horizontal scrollbar.
yscrollcommand	To make the text widget vertically scrollable, set this option to the set() method of the vertical scrollbar.

Methods

Text objects have these methods:

Methods & Description

delete(startindex   [,endindex])

This method deletes a specific character or a range of text.

get(startindex   [,endindex])

This method returns a specific character or a range of text.

index(index)

Returns the absolute value of an index based on the given index.

insert(index   [,string]…)

This method inserts strings at the specified index location.

see(index)

This method returns true if the text located at the index position is visible.

We have the following methods available when handling marks:

Methods & Description
index(mark) Returns the line and column location of a specific mark.
mark_gravity(mark   **[,gravity])** Returns the gravity of the given mark. If the second argument is provided, the gravity is set for the given mark.
mark_names() Returns all marks from the Text widget.
mark_set(mark,   **index)** Informs a new position to the given mark.
mark_unset(mark) Removes the given mark from the Text widget.

Tags are used to associate names to regions of text   which makes easy the task of
Text widgets support three distinct helper structures: Marks, Tabs, and Indexes:
Marks are used to bookmark positions between two characters within a given text.

modifying the display settings of specific text areas. Tags are also used to bind event callbacks to specific ranges of text.

Following are the available methods for handling tabs:

Methods and Description
tag_add(tagname, startindex[,endindex] ...) This method tags either the position defined by startindex, or a range delimited by the positions startindex and endindex.
tag_config

You can use this method to configure the tag properties, which include, justify(center, left, or right), tabs(this property has the same functionality of the Text widget tabs's property), and underline(used to underline the tagged text).

tag_delete(tagname)

This method is used to delete and remove a given tag.

tag_remove(tagname [,startindex[.endindex]] ...)

After applying this method, the given tag is removed from the provided area without deleting the actual tag definition.

Example

Try the following example yourself:

```
from Tkinter import *

 def onclick():

   pass    root = Tk() text = Text(root) text.insert(INSERT, "Hello.....")
text.insert(END, "Bye Bye.....") text.pack()
```

```
text.tag_add("here",    "1.0",    "1.4")   text.tag_add("start",    "1.8",   "1.13")
text.tag_config("here",             background="yellow",             foreground="blue")
text.tag_config("start",        background="black",        foreground="green")
root.mainloop()
```

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

15. TopLevel

Toplevel widgets work as windows that are directly managed by the window manager. They do not necessarily have a parent widget on top of them.

Your application can use any number of top-level windows.

Syntax

Here is the simple syntax to create this widget:

```
w = Toplevel ( option, ... )
```

Parameters:

 options: Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Option	Description
bg	The background color of the window.
bd	Border width in pixels; default is 0.
cursor	The cursor that appears when the mouse is in this window.
class_	Normally, text selected within a text widget is exported to be the selection in the window manager. Set exportselection=0 if you don't want that behavior.
font	The default font for text inserted into the widget.
fg	The color used for text (and bitmaps) within the widget. You can change the color for tagged regions; this option is just the default.
height	Window height.

relief	Normally, a top-level window will have no 3-d borders around it. To get a shaded border, set the bd option larger that its default value of zero, and set the relief option to one of the constants.
width	The desired width of the window.

Methods

Toplevel objects have these methods:

Methods and Description
deiconify()   Displays the window, after using either the iconify or the withdraw methods.
frame()   Returns a system-specific window identifier.
group(window)   Adds the window to the window group administered by the given window.
iconify()   Turns the window into an icon, without destroying it.
protocol(name, function)   Registers a function as a callback which will be called for the given protocol.
iconify()   Turns the window into an icon, without destroying it.
state()

Returns the current state of the window. Possible values are normal, iconic, withdrawn and icon.

transient([master])

Turns the window into a temporary(transient) window for the given master or to the window's parent, when no argument is given.

withdraw()

Removes the window from the screen, without destroying it.

maxsize(width, height)

Defines the maximum size for this window.

minsize(width, height)

Defines the minimum size for this window.

positionfrom(who)

Defines the position controller.

resizable(width, height)

Defines the resize flags, which control whether the window can be resized.

sizefrom(who)

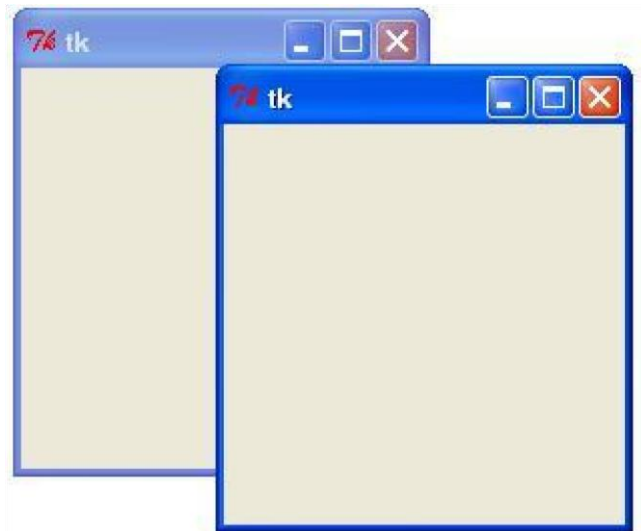Defines the size controller.

title(string)

Defines the window title.

Example

Try following example yourself:

```
from Tkinter import *

 root = Tk() top = Toplevel()

 top.mainloop()
```

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

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.

Syntax

Here is the simple syntax to create this widget:

```
w = Spinbox( master, option, ... )
```

Parameters

master: This represents the parent window.

options: Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Option	Description

activebackground	The color of the slider and arrowheads when the mouse is over them.
bg	The color of the slider and arrowheads when the mouse is not over them.
bd	The width of the 3-d borders around the entire perimeter of the trough, and also the width of the 3-d effects on the arrowheads and slider. Default is no border around the trough, and a 2-pixel border around the arrowheads and slider.
command	A procedure to be called whenever the scrollbar is moved.
cursor	The cursor that appears when the mouse is over the scrollbar.
disabledbackground	The background color to use when the widget is disabled.
disabledforeground	The text color to use when the widget is disabled.
fg	Text color.
font	The font to use in this widget.
format	Format string. No default value.
from_	The minimum value. Used together with to to limit the spinbox range.
justify	Default is LEFT
relief	Default is SUNKEN.

repeatdelay	Together with repeatinterval, this option controls button auto-repeat. Both values are given in milliseconds.
repeatinterval	See repeatdelay.
state	One of NORMAL, DISABLED, or "readonly". Default is NORMAL.
textvariable	No default value.
to	See from.
validate	Validation mode. Default is NONE.
validatecommand	Validation callback. No default value.
values	A tuple containing valid values for this widget. Overrides from/to/increment.
vcmd	Same as validatecommand.
width	Widget width, in character units. Default is 20.
wrap	If true, the up and down buttons will wrap around.
xscrollcommand	Used to connect a spinbox field to a horizontal scrollbar. This option should be set to the set method of the corresponding scrollbar.

Methods

Spinbox objects have these methods:

Methods and Description
delete(startindex [,endindex]) This method deletes a specific character or a range of text.
get(startindex [,endindex]) This method returns a specific character or a range of text.
identify(x, y) Identifies the widget element at the given location.
index(index) Returns the absolute value of an index based on the given index.
insert(index [,string]...) This method inserts strings at the specified index location.
invoke(element) Invokes a spinbox button.

Example

Try the following example yourself:

```
from Tkinter import *

 master = Tk()

 w = Spinbox(master, from_=0, to=10)

w.pack()  mainloop()
```

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

17. PanelWindow

A PanedWindow is a container widget that may contain any number of panes, arranged horizontally or vertically.

Each pane contains one widget and each pair of panes is separated by a moveable (via mouse movements) sash. Moving a sash causes the widgets on either side of the sash to be resized.

Syntax

Here is the simple syntax to create this widget:

```
w = PanedWindow( master, option, ... )
```

Parameters

18.   **master:** This represents the parent window.

19. **options:** Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Option	Description
bg	The color of the slider and arrowheads when the mouse is not over them.
bd	The width of the 3-d borders around the entire perimeter of the trough, and also the width of the 3-d effects on the arrowheads and slider. Default is no border around the trough, and a 2-pixel border around the arrowheads and slider.

borderwidth	Default is 2.
cursor	The cursor that appears when the mouse is over the window.
handlepad	Default is 8.
handlesize	Default is 8.

height	No default value.
orient	Default is HORIZONTAL.
relief	Default is FLAT.
sashcursor	No default value.
sashrelief	Default is RAISED.
sashwidth	Default is 2.
showhandle	No default value
width	No default value.

Methods

PanedWindow objects have these methods:

Methods and Description
add(child,   options) Adds a child window to the paned window.
get(startindex   [,endindex]) This method returns a specific character or a range of text.
config(options) Modifies one or more widget options. If no options are given, the method returns a dictionary containing all current option values.

Example

Try the following example yourself. Here's how to create a 3-pane widget:

```
from Tkinter import *

 m1 = PanedWindow()
```

417

```
m1.pack(fill=BOTH, expand=1)

 left = Label(m1, text="left pane") m1.add(left)

 m2 = PanedWindow(m1, orient=VERTICAL) m1.add(m2)

 top = Label(m2, text="top pane") m2.add(top)

 bottom = Label(m2, text="bottom pane") m2.add(bottom)

 mainloop()
```

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

20. LabelFrame

A labelframe is a simple container widget. Its primary purpose is to act as a spacer or container for complex window layouts.

This widget has the features of a frame plus the ability to display a label.

Syntax

Here is the simple syntax to create this widget:

```
w = LabelFrame( master, option, ... )
```

Parameters

21.   **master:** This represents the parent window.

22. **options:** Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Option	Description
bg	The normal background color displayed behind the label and indicator.
bd	The size of the border around the indicator. Default is 2 pixels.
cursor	If you set this option to a cursor name (*arrow, dot etc.*), the mouse cursor will change to that pattern when it is over the checkbutton.
font	The vertical dimension of the new frame.
height	The vertical dimension of the new frame.
labelAnchor	Specifies where to place the label.
highlightbackground	Color of the focus highlight when the frame does not have focus.
highlightcolor	Color shown in the focus highlight when the frame has the focus.
highlightthickness	Thickness of the focus highlight.

relief	With the default value, relief=FLAT, the checkbutton does not stand out from its background. You may set this option to any of the other styles
text	Specifies a string to be displayed inside the widget.
width	Specifies the desired width for the window.

Example

Try the following example yourself. Here is how to create a labelframe widget:
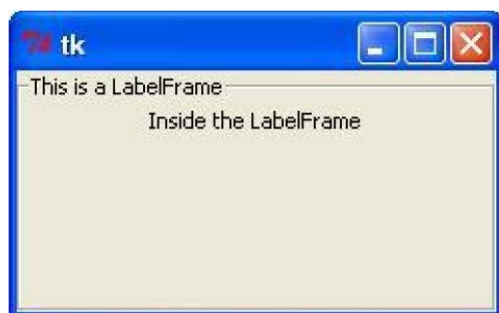
```
from Tkinter import *

 root = Tk()

 labelframe    =    LabelFrame(root,    text="This    is    a    LabelFrame")
labelframe.pack(fill="both", expand="yes")

  left = Label(labelframe, text="Inside the LabelFrame") left.pack()

  root.mainloop()
```

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

23. tkMessageBox

The tkMessageBox module is used to display message boxes in your applications. This module provides a number of functions that you can use to display an appropriate message.

Some of these functions are showinfo, showwarning, showerror, askquestion, askokcancel, askyesno, and askretryignore.

Syntax:

Here is the simple syntax to create this widget:

```
tkMessageBox.FunctionName(title, message [, options])
```

Parameters

24. **FunctionName:** This is the name of the appropriate message box function.

25. **title:** This is the text to be displayed in the title bar of a message box.

26. **message:** This is the text to be displayed as a message.

27. **options:** options are alternative choices that you may use to tailor a standard message box. Some of the options that you can use are default and parent. The default option is used to specify the default button, such as ABORT, RETRY, or IGNORE in the message box. The parent option is used to specify the window on top of which the message box is to be displayed.

You could use one of the following functions with dialogue box:

28. showinfo()

29. showwarning()

30. showerror ()

31. askquestion()

32. askokcancel()

33. askyesno ()

34. askretrycancel ()

Example

Try the following example yourself:

```
import Tkinter import tkMessageBox

 top = Tkinter.Tk() def hello():
```

```
    tkMessageBox.showinfo("Say Hello", "Hello World")



B1 = Tkinter.Button(top, text = "Say Hello", command = hello)

B1.pack()

 top.mainloop()
```

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

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

Let us study them briefly:

Dimensions

Various lengths, widths, and other dimensions of widgets can be described in many different units.

35. If you set a dimension to an integer, it is assumed to be in pixels.

36. You can specify units by setting a dimension to a string containing a number followed by.

Character	Description
c	Centimeters
i	Inches
m	Millimeters
p	Printer's points (about 1/72")

Length options:

Tkinter expresses a length as an integer number of pixels. Here is the list of common length options:

37. **borderwidth:** Width of the border which gives a three-dimensional look to the widget.

38. **highlightthickness:** Width of the highlight rectangle when the widget has focus.

39. **padX padY:** Extra space the widget requests from its layout manager beyond the minimum the widget needs to display its contents in the x and y directions.

40. selectborderwidth: Width of the three-dimentional border around selected items of the widget.

41. **wraplength:** Maximum line length for widgets that perform word wrapping.

42. **height:** Desired height of the widget; must be greater than or equal to 1.

43. **underline:** Index of the character to underline in the widget's text (0 is the first character, 1 the second one, and so on).

44. **width:** Desired width of the widget.

45. Colors :Tkinter represents colors with strings.

There are two general ways to specify colors in Tkinter:

46. You can use a string specifying the proportion of red, green and blue in hexadecimal digits. For example, "#fff" is white, "#000000" is black, "#000fff000" is pure green, and "#00ffff" is pure cyan (green plus blue).

47. You can also use any locally defined standard color name. The colors "white", "black", "red", "green", "blue", "cyan", "yellow", and "magenta" will always be available.

Color options

The common color options are:

48. **activebackground:** Background color for the widget when the widget is active.

49. **activeforeground:** Foreground color for the widget when the widget is active.

50. **background:** Background color for the widget.

This can also be represented as *bg*.

51. **disabledforeground:** Foreground color for the widget when the widget is disabled.

52. **foreground:** Foreground color for the widget.

This can also be represented as *fg*.

53. **highlightbackground:** Background color of the highlight region when the widget has focus.

54. **highlightcolor:** Foreground color of the highlight region when the widget has focus.

55. **selectbackground:** Background color for the selected items of the widget.

56. **selectforeground:** Foreground color for the

selected items of the widget.

57. Fonts

There may be up to three ways to specify type style.

Simple Tuple Fonts

As a tuple whose first element is the font family, followed by a size in points, optionally followed by a string containing one or more of the style modifiers bold, italic, underline and overstrike.

Example

58.   ("Helvetica", "16") for a 16-point Helvetica regular.

59. ("Times", "24", "bold italic") for a 24-point

Times bold italic.

Font object Fonts

You can create a "font object" by importing the tkFont module and using its Font class constructor:

```
import tkFont

  font = tkFont.Font ( option, ... )
```

Here is the list of options:

60.   **family:** The font family name as a string.

61. **size:** The font height as an integer in points. To get a font n pixels high, use -n.

62. **weight:** "bold" for boldface, "normal" for

regular weight.

63. **slant:** "italic" for italic, "roman" for unslanted.

64.   **underline:** 1 for underlined text, 0 for normal.

65.   **overstrike:** 1 for overstruck text, 0 for normal. Example

```
helv36 = tkFont.Font(family="Helvetica",size=36,weight="bold")
```

X Window Fonts

If you are running under the X Window System, you can use any of the X font names.

For example, the font named "-*-lucidatypewriter-medium-r-*-*-*-140-*-*-*-*-**" is the author's favorite fixed-width font for onscreen use. Use the *xfontsel* program to help you select pleasing fonts.

66. Anchors

Anchors are used to define where text is positioned relative to a reference point.

Here is list of possible constants, which can be used for Anchor attribute.

67.   NW

68.   N

69.   NE

70.   W

71.   CENTER

72.   E

73.   SW

74.   S

75.   SE

For example, if you use CENTER as a text anchor, the text will be centered horizontally and vertically around the reference point.

Anchor NW will position the text so that the reference point coincides with the northwest (top left) corner of the box containing the text.

Anchor W will center the text vertically around the reference point, with the left edge of the text box passing through that point, and so on.

If you create a small widget inside a large frame and use the anchor=SE option, the widget will be placed in the bottom right corner of the frame. If you used anchor=N instead, the widget would be centered along the top edge.

Example

The anchor constants are shown in this diagram:
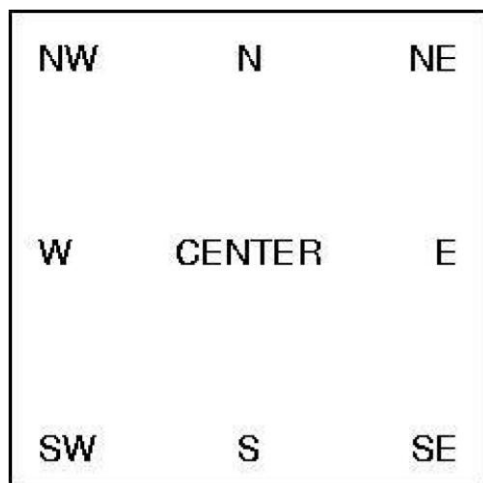
76. Relief styles

The relief style of a widget refers to certain simulated 3-D effects around the outside of the widget. Here is a screenshot of a row of buttons exhibiting all the possible relief styles:

Here is list of possible constants which can be used for relief attribute.

77.   FLAT

78.   RAISED

79.   SUNKEN

80.   GROOVE
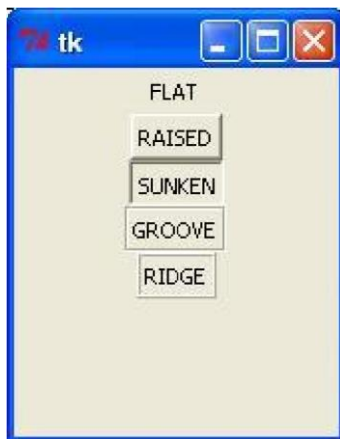
81.   RIDGE

Example

```
from Tkinter import * import Tkinter
```

```
top = Tkinter.Tk()
```

```
B1 = Tkinter.Button(top, text ="FLAT", relief=FLAT )

B2 = Tkinter.Button(top, text ="RAISED", relief=RAISED )

B3 = Tkinter.Button(top, text ="SUNKEN", relief=SUNKEN )

B4 = Tkinter.Button(top, text ="GROOVE", relief=GROOVE )

B5 = Tkinter.Button(top, text ="RIDGE", relief=RIDGE )



B1.pack()

B2.pack()

B3.pack()

B4.pack() B5.pack() top.mainloop()
```

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

This attribute to displays a bitmap. There are following type of bitmaps available:

82.   Bitmaps

83.   "error"

84.   "gray75"

85.   "gray50"

86.   "gray25"

87.   "gray12"

88.   "hourglass"

89.   "info"

90.   "questhead"

91.   "question"

92.   "warning"

Example

```
from Tkinter import * import Tkinter

 top = Tkinter.Tk()



B1     =     Tkinter.Button(top,    text    ="error",    relief=RAISED,\
bitmap="error")

B2     =     Tkinter.Button(top,    text    ="hourglass",    relief=RAISED,\
bitmap="hourglass")

B3     =     Tkinter.Button(top,    text    ="info",    relief=RAISED,\
bitmap="info")

B4     =     Tkinter.Button(top,    text    ="question",    relief=RAISED,\
bitmap="question")

B5     =     Tkinter.Button(top,    text    ="warning",    relief=RAISED,\
bitmap="warning")
```
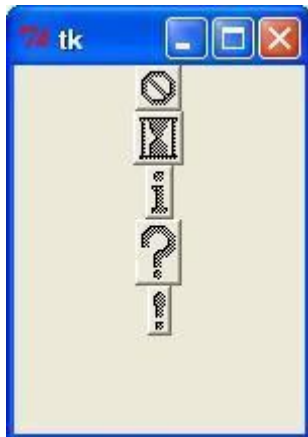
```
B1.pack()

B2.pack()

B3.pack()
```

B4.pack() B5.pack() top.mainloop()

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

93. Cursors

Python Tkinter supports quite a number of different mouse cursors available. The exact graphic may vary according to your operating system.

Here is the list of interesting ones:

94.   "arrow"

95.   "circle"

96.   "clock"

97.   "cross"

98.   "dotbox"

99. "exchange" 100.   "fleur"

101.   "heart"

102. "heart"

103. "man"

104. "mouse"

105. "pirate"

106. "plus"

107.   **"shuttle"**

108.   **"sizing"**

109.   **"spider"**

110.   **"spraycan"**

111.   **"star"**

112.   **"target"**

113.   **"tcross"**

114.   **"trek"**

115.   **"watch"**

Example

Try the following example by moving cursor on different buttons:

```
from Tkinter import * import Tkinter

 top = Tkinter.Tk()



B1     =    Tkinter.Button(top,    text    ="circle",    relief=RAISED,\
 cursor="circle")
```

```
B2     =      Tkinter.Button(top,    text    ="plus",    relief=RAISED,\
cursor="plus")


B1.pack() B2.pack() top.mainloop()
```

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.

116. The *pack()* Method - This geometry manager organizes widgets in blocks before placing them in the parent widget.

117. **The *grid()* Method** - This geometry manager

organizes widgets in a table-like structure in the parent widget.

118. 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:

pack()

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:

119. **expand:** When set to true, widget expands to fill any space not otherwise used in widget's parent.

120. **fill:** Determines whether widget fills any extra space allocated to it by the packer, or keeps its own minimal dimensions: NONE (default), X (fill only horizontally), Y (fill only vertically), or BOTH (fill both horizontally and vertically).

121. side: **Determines which side of the parent widget packs against: TOP (default), 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="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()
```

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

grid()

Here is the list of possible options:

122. **column :** The column to put widget in; default

0 (leftmost column).

123. **columnspan:**   How many columns widgetoccupies; default 1.

124. **ipadx, ipady :**How many pixels to pad widget, horizontally and vertically, inside widget's borders.

125. padx, pady : **How many pixels to pad widget, horizontally and vertically, outside v's borders.**

126. **row:** The row to put widget in; default the first row that is still empty.

127. **rowspan :** How many rowswidget occupies; default 1.

128. **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 x 4 grid:

129.   3. place()

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:

130. **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 (the upper left corner of widget)

131. bordermode : **INSIDE (the default) to indicate that other options refer to the parent's inside (ignoring the parent's border); OUTSIDE otherwise.**

132. **height, width :** Height and width in pixels.

133. **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.

134. **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.

135. **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: