Basic Python

Genesis InSoft Limited

A name you can trust

1-7-1072/A, Opp. Saptagiri Theatre, RTC * Roads, Hyderabad - 500 020

Python

Genesis Computers (A unit of Genesis InSoft Limited) started its operations on March 16th 1992 in Hyderabad, India primarily as a centre for advanced software education, development and consultancy.

Training is imparted through lectures supplemented with on-line demonstrations using audio visual aids. Seminars, workshops and demonstrations are organized periodically to keep the participants abreast of the latest technologies. Genesis InSoft Ltd. is also involved in software development and consultancy.

We have implemented projects/training in technologies ranging from client server applications to web based. Skilled in PowerBuilder, Windows C SDK, VC++, C++, C, Visual Basic, Java, J2EE, XML, WML, HTML, UML, Java Script, MS.NET (C#, VB.NET and ASP.NET, ADO.NET etc), PHP, Jhoomla, Zend Framework, JQuery, ExtJS, PERL, Python, TCL/TK etc. using Oracle, MySql and SQL Server as backend databases.

Genesis has earned a reputation of being in forefront on Technology and is ranked amongst the top training institutes in Hyderabad city. The highlight of Genesis is that new emerging technologies are absorbed quickly and applied in its areas of operation.

We have on our faculty a team of highly qualified and trained professionals who have worked both in India and abroad. So far we have trained about 52,000+ students who were mostly engineers and experienced computer professionals themselves.

Tapadia (MS, USA), the founder of Genesis Computers, has about 30+ years experience in software industry. He worked for OMC computers, Intergraph India Private Ltd., Intergraph Corporation (USA), and was a consultant to D.E. Shaw India Software Private Ltd and iLabs Limited. He has more than 30 years of teaching experience, and has conducted training for the corporates like ADP, APSRTC, ARM, B.H.E.L, B2B Software Technologies, Cambridge Technology Enterprises Private Limited, CellExchange India Private Limited, Citicorp Overseas Software Limited, CMC Centre (Gachibowli, Posnett Bhavan), CommVault Systems (India) Private Limited, Convergys Information Management (India) Private Limited, D.E. Shaw India Software Private Limited, D.R.D.L, Dell EMC, Bangalore (behalf of DevelopIntelligence, USA), Deloitee Consulting India Private Limited, ELICO Limited, eSymbiosis ITES India Private Limited, Everypath Private Limited, Gold Stone Software, HCL Consulting (Chennai), iLabs Limited, Infotech Enterprises, Intelligroup Asia Private Limited, Intergraph India Private Limited, Invensys Development Centre India Private Limited, Ivy Comptech, JP Systems (India) Limited, Juno Online Services Development Private Limited, Malpani Soft Private Limited, Mars Telecom Systems Private Limited, Mentor Graphics India Private Limited, Motorola India Electronics Limited, NCR Corporation India Private Limited, Netrovert Software Private Limited, Nokia India Private Limited, Optima Software Services, Oracle India Private Limited, Polaris Software Lab Limited, Qualcomm India Private Limited (Chennai, Hyderabad, Bangalore), Qualcomm China, Quantum Softech Limited, R.C.I, Renaissance Infotech, Satyam Computers, Satyam GE, Satyam Learning Centre, SIS Software (India) Private Limited, Sriven Computers, Teradata - NCR, Tanla Solutions Limited, Timmins Training Consulting Sdn. Bhd, Kuala Lumpur, Vazir Sultan Tobacco, Verizon, Virtusa India Private Limited, Wings Business Systems Private Limited, Wipro Systems, Xilinx India Technology Services Private Limited, Xilinx Ireland, Xilinx Inc (San Jose).

Genesis InSoft Limited
1-7-1072/A, RTC * Roads, Hyderabad - 500 020

rtapadia@genesisinsoft.com
www.genesisinsoft.com

Python is a general-purpose high-level programming language. Its design philosophy emphasizes code readability. Python claims to "combine remarkable power with very clear syntax", and its standard library is large and comprehensive. Its use of indentation for block delimiters is unusual among popular programming languages.

Programming languages support decomposing problems in several different ways:

• Most programming languages are functional/procedural: programs are lists of instructions that tell the computer what to do with the program's input. C, Pascal, and even Unix shells are procedural languages.

• In declarative languages, you write a specification that describes the problem to be solved, and the language implementation figures out how to perform the computation efficiently. SQL is the declarative language; a SQL query describes the data set you want to retrieve, and the SQL engine decides whether to scan tables or use indexes, which subclauses should be performed first, etc.

• Object-oriented programs manipulate collections of objects. Objects have internal state and support methods that query or modify this internal state in some way. Smalltalk and Java are object-oriented languages. C++ and Python are languages that support object-oriented programming, but do not force the use of object-oriented features.

Python supports multiple programming paradigms and features a fully dynamic type system and automatic memory management, similar to Perl, Tcl etc. Like other dynamic languages, Python is often used as a scripting language.

Another situation: perhaps you have to work with several C libraries, and the usual C write/compile/test/re-compile cycle is slow. You need to develop software more quickly. Possibly, perhaps you have written a program that could use an extension language, and you do not want to design a language, write and debug an interpreter for it, then tie it into your application.

Python allows you to split up your program in modules that can be reused in other Python programs. It comes with a large collection of standard modules that you can use as the basis of your programs - or as examples to start learning to program in Python. There are also built-in modules that provide things like file I/O, system calls, sockets, and even interfaces to graphical user interface toolkits like Tk.

Python is an interpreted language, which can save you considerable time during program development because no compilation and linking are necessary. The interpreter can be used interactively, which makes it easy to experiment with features of the language, to write programs, or to test functions during bottom-up program development.

Python allows writing very compact and readable programs. Programs written in Python are typically much shorter than equivalent C or C++ programs, for several reasons:

- the high-level data types allow you to express complex operations in a single statement;

- statement grouping is done by indentation instead of beginning and ending brackets;
- no variable or argument declarations are necessary.

Python is extensible: if you know how to program in C it is easy to add a new built-in function or module to the interpreter, either to perform critical operations at maximum speed, or to link Python programs to libraries that may only be available in binary form (such as a vendor-specific graphics library). You can link the Python interpreter into an application written in C and use it as an extension or command language for that application.

Application of Python

Since its origin in 1989 (created by Guido van Rossum, a Dutch programmer), Python has grown to become part of web-based, desktop-based, graphic design, scientific, and computational applications. With Python available for Windows, Mac OS X and Linux / UNIX, it offers ease of development for enterprises.

1) GUI-Based Desktop Applications:
   Python has simple syntax, modular architecture, rich text processing tools and the ability to work on multiple operating systems, which make it a desirable choice for developing desktop-based applications. There are various GUI toolkits like Tkinter, PyGtk available which help developers create highly functional Graphical User Interface (GUI).

2) Web Frameworks and Web Applications:
   Python has been used to create a variety of web-frameworks including Django, Flask etc. These frameworks provide standard libraries and modules that simplify tasks related to content management, interaction with database and interfacing with different internet protocols such as HTTP, SMTP, XML-RPC, FTP and POP. Plone, a content management system; ERP5, an open source ERP which is used in aerospace, apparel and banking; and Google App engine are a few of the popular web applications based on Python.

3) Enterprise and Business Applications:
   Python is a suitable coding language for customizing larger applications. Reddit, which was originally written in Common Lips, was rewritten in Python in 2005. Python also contributed in a large part to functionality in YouTube.

4) Operating Systems:
   Python is often an integral part of Linux distributions. For instance, Ubuntu's Ubiquity Installer, and Fedora's and Red Hat Enterprise Linux's Anaconda Installer are written in Python.

5) Prototyping:
   Besides being quick and easy to learn, Python also has the open source advantage of being free with the support of a large community. This makes it the preferred choice for prototype development.

Using the Interpreter

The Python interpreter is usually installed as '/usr/local/bin/python' on those machines where it is available; putting '/usr/local/bin' in your UNIX shell's search path makes it possible to start it by typing the command to the shell or command prompt (windows)

```
Python
```

Typing an end-of-file character (Control-D on UNIX, Control-Z on Windows) at the primary prompt causes the interpreter to exit with a zero exit status. If that does not work, you can exit the interpreter by typing the following commands:

```
import sys;sys.exit()
or
exit()
```

A second way of starting the interpreter is 'python -c command [arg] ...', which executes the statement(s) in command, analogous to the shell's -c option. Since Python statements often contain spaces or other characters that are special to the shell, it is best to quote command in its entirety with double quotes.

For example, the following command causes the interpreter to exit.

```
python -c exit()
```

When known to the interpreter, the script name and additional arguments thereafter are passed to the script in the variable `sys.argv`, which is a list of strings. Its length is at least one; when no script and no arguments are given, `sys.argv[0]` is an empty string.

When commands are read from a tty, the interpreter is said to be in interactive mode. In this mode it prompts for the next command with the primary prompt, usually three greater-than signs ('>>> '); for continuation lines it prompts with the secondary prompt, by default three dots ('**...**'). The interpreter prints a welcome message stating its version number and a copyright notice before printing the first prompt.

Continuation lines are needed when entering a multi-line construct. As an example, look at this if statement:

```
d:\>python
Python 3.9.2 (tags/v3.9.2:1a79785, Feb 19 2021, 13:44:55) [MSC v.1928
64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> value=1
>>> if value:
...     print("value is 1")
...
value is 1
>>> exit()
>>> help('modules')
```

```
>>> help(str)
>>> help(int)
>>> help('builtins')
>>> help('keywords')
```

When an error occurs, the interpreter prints an error message and a stack trace. In interactive mode, it then returns to the primary prompt; when input came from a file, it exits with a nonzero exit status after printing the stack trace (exceptions handled by an except clause in a try statement are not errors in this context). Some errors are fatal and cause an exit with a nonzero exit; this applies to internal inconsistencies and some cases of running out of memory. All error messages are written to the standard error stream; normal output from the executed commands is written to standard output.

Typing the interrupt character (usually Control-C or DEL) to the primary or secondary prompt cancels the input and returns to the primary prompt. Typing an interrupt while a command is executing raises the KeyboardInterrupt exception, which may be handled by a try statement.

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.

Code 1 File: first.py

```
# first script
x = "Genesis"
print (x)

"""
multiline comments
Accepts data from console. Input returns a string. To convert it to
int use int()
"""

str = input("Enter input: ");
print(str, type(str))
print("Input is:", int(str))

D:\Python39>python first.py
```

Types and operators

Everything in Python is an object. All objects in Python can be either mutable or immutable.

When an object is instantiated, it is assigned a unique object id. Its type is defined at runtime. A mutable object can be changed after it is created, and an immutable object cannot.

Objects of built-in types like (int, float, bool, str, tuple, frozenset) are immutable. Objects of built-in types like (list, set, dict) are mutable. Custom classes are generally mutable. To simulate immutability in a class, one should override attribute setting and deletion to raise exceptions.

Class	Description	Immutable
bool	Boolean value	Yes
int	Integer	Yes
float	Float point number	Yes
list	Mutable sequence of objects	No
tuple	Immutable sequence of objects	Yes
str	Character string	Yes
dict	Associative mapping (dictionary)	No
set	Unique set of values	No
frozenset	Immutable form of set	Yes

The built-in function id() returns the identity of an object as an integer. This integer usually corresponds to the object's location in memory, although this is specific to the Python implementation and the platform being used.

Code 2 File: id type.py

```
# Immutable objects does not allow modification after creation

num1 = 10
print(id(num1))
print(type(num1))

num2 = num1

if(id(num1) == id(num2)):
  print("same ids: 1")

if(id(num1) == id(10)):
  print("same ids: 2")

num1 = num1 + 10
print(num1, num2)
print(id(num1), id(num2))
```

Besides numbers, Python can also manipulate strings, which can be expressed in several ways.

Strings

Strings can be enclosed in single quotes or double quotes.

Strings can be concatenated with the + operator, and repeated with *. Two string literals next to each other are automatically concatenated.

Strings can be subscripted (indexed); like in C, the first character has index 0. There is no separate character type; a character is simply a string of size one. Substrings can be specified with the slice notation: two indices separated by a colon.

Slice indices have useful defaults; an omitted first index defaults to zero, an omitted second index defaults to the size of the string being sliced. Strings cannot be changed. Assigning to an indexed position in the string results in an error.

Indices may be negative numbers, to start counting from the right.

```
+---+---+---+---+---+---+---+
| G | E | N | E | S | I | S |
+---+---+---+---+---+---+---+
0   1   2   3   4   5   6   7
-7 -6  -5  -4  -3  -2  -1
```

The first row of numbers gives the position of the indices 0 to 6 in the string; the second row gives the corresponding negative indices. The slice from i to j consists of all characters between the edges labeled i and j, respectively. For non-negative indices, the length of a slice is the difference of the indices, if both are within bounds. For example, the length of word[1:3] is 2.

The built-in function len() returns the length of a string.

Code 3 File: str.py

```
str1 = "Genesis"
str2 = "Genesis"
print(id(str1), id(str2))
# str1[1] = 'a'      # illegal, since strings are immmutable

# str is really a pointer, and we are simply pointing it to a
# new string created from old ones
str1 = str1[:1] + 'i' + str1[6:]

print(str1, str2)
print(id(str1), id(str2))

str = "Genesis"
print(str.index('e'))
print(str.index('es'))
print('a' in str)

print(str.upper(), str.lower())
print(str.replace('a', 'b'))
str=";"

print(str.join(('Insoft', 'hyd', 'india')))
str="Genesis"
print(str*3, len(str))
print(str[2:10])

str = "hello123"
print(str.isalnum(), str.isalpha())
```

```
val = "123"
print(val.isdigit())

str = "Genesis"
print(str.isalpha())
```

Function eval() evaluates the passed string as a Python expression and returns the result. For example, eval("1 + 2") interprets and executes the expression "1 + 2" and returns the result (3).

Since Python treats a newline as a statement terminator, and since statements are often more than (one line, many people use a backslash to continue statements).

This could be dangerous as a stray space after the \ would make this line wrong, and stray spaces are hard to see in editors. In this case, at least it would be a syntax error. It is usually much better to use the implicit continuation inside parentheses, square brackets or curly braces.

Code 4 File: two.py

```
# Using Backslash to continue statements
import os

a = 2 + \
1.5 * 3
print("a = ", a)
b = eval("3.2") + 3
print("b = ", b)

print(eval("4 < 10"))
print(eval("'hello ' * 5"))
print(eval("abs(-11)"))
print(eval('"hello".upper()'))
print(eval('os.getcwd()'))

c = eval("2.3") + 2
print("c = ", c, type(c))
d = "3.2" + "3"
print("d = ", d, type(d))

# use the implicit continuation inside parenthesis
e = ([5, 12, 13,
200])
print("e1 = ", e)
```

There are two ways to format your output; the first way is to do all the string handling yourself; using string slicing and concatenation operations you can create any layout. The standard module string contains some useful operations for padding strings to a given column width. The second way is to use the str.format() method.

How to convert values to strings? Python has ways to convert any value to a string: pass it to the repr() or str() functions.

The str() function is meant to return representations of values which are fairly human-readable, while repr() is meant to generate representations which can be read by the interpreter (or will force a SyntaxError if there is no equivalent syntax).

Almost always use str when creating output for end users.

The repr function is mainly useful for debugging and exploring. For example, if you suspect a string has non printing characters in it, or a float has a small rounding error, repr will show it; str may not.

The rjust() method of string objects, which right-justifies a string in a field of a given width by padding it with spaces on the left. There are similar methods ljust() and center(). These methods do not write anything, they just return a new string. If the input string is too long, they don't truncate it, but return it unchanged; this will mess up your column lay-out but that's usually better than the alternative, which would be lying about a value.

There is another method, zfill(), which pads a numeric string on the left with zeros. It understands about plus and minus signs.

Positional and keyword arguments can be arbitrarily combined.

An optional ':' and format specifier can follow the field name. This allows greater control over how the value is formatted. The following example truncates PI to three places after the decimal.

A trailing comma avoids the newline after the output.

Code 5 File: print.py

```
s="genesis"

"""
Name s is attached to 'genesis' string. When you call str(s) the
interpreter puts 'genesis' instead of s and then calls str('genesis').
"""

print(str(s))
print(repr(s))

for x in range(1, 5):
  print(str(x).rjust(2), str(x*x).rjust(3), sep=";", end=' ')
  # Note use of 'end' in the above
  print(str(x*x*x).rjust(4))

print()
for x in range(1, 5):
  print('{0:2d} {1:3d} {2:4d}'.format(x, x*x, x*x*x))

print()
print('12'.zfill(5))
```

```
print('3.14159265'.zfill(5))
print('-3.14'.zfill(7))

print('{0:12} and {1:10}'.format('company', 'Genesis'))

print()

print('PI value is approximately {0:.3f}'.format(3.14159265))

print('{0} is {1}'.format('Country', 'India'))
print('{1} is {0}'.format('Country', 'India'))
print('Our {key} is {data}'.format(key='country', data='india'))

print('{0} {company}'.format('Genesis', company='InSoft'))
```

if statement

Code 6 File: if.py

```
x = int(input("Please enter an integer: "))
if x < 0:
  print('Less than zero')
elif x == 0:
  print('Zero')
else:
  print('greater than zero')
```

There can be zero or more elif parts, and the else part is optional. The keyword ‘elif’ is short for ‘elseif’, and is useful to avoid excessive indentation. An if . . . elif . . . else sequence is a substitute for the switch or case statements found in other languages

for statement

The for statement in Python differs a bit from what you used in C or C++. Rather than giving the user the ability to define both the iteration step and halting condition (as C), Python’s for statement iterates over the items of any sequence (a list or a string), in the order that they appear in the sequence.

If you need to modify the list you are iterating over (for example, to duplicate selected items) you must iterate over a copy. The slice notation makes this particularly convenient:

Code 7 File: for.py

```
data = ['genesis', 'insoft', 'limited']
for x in data:
  print(x, len(x))

datacp = []

for x in data:
```

```
  if len(x) > 6: datacp.insert(0, x)

print(datacp)

months = ['jan', 'feb', 'march', 'april']
for i in range(len(months)):
  print("month {}: {}".format(i + 1, months[i]))

months = ['july', 'august', 'september', 'october']
for num, name in enumerate(months, start = 7):
  print("month {}: {}".format(num, name))
```

Python has a while construct too (though not an until). There is also a break statement like that of C/C++, used to leave loops "prematurely."

```
x = 5
while 1:
   x += 1
   if x == 8:
      print (x)
      break
```

The colon at the end of the while line defines the start of a block. Unlike languages like C/C++ or even Perl, which use braces to define blocks, Python uses a combination of a colon and indenting to define a block.

By inserting this colon a block begins on the next line, indented that line, and the lines following it, further right than the current line, in order to tell you that these lines form a block.

The break statement breaks out of the smallest enclosing for or while loop.
The continue statement continues with the next iteration of the loop.

Loop statements may have an else clause; it is executed when the loop terminates through exhaustion of the list (with for) or when the condition becomes false (with while), but not when the loop is terminated by a break statement.

The pass statement does nothing. It can be used when a statement is required syntactically but the program requires no action. For example:

```
while True:
   pass  # Busy-wait for keyboard interrupt
```

Lists

Python knows a number of compound data types, used to group together other values. The most versatile is the list, which can be written as a list of comma-separated values (items) between square brackets. List items need not all have the same type.

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

Unlike strings, which are immutable, it is possible to change individual elements of a list.

Code 8 File: list1.py

```
e = [8, 15, 3, 20]

print("e1 = ", e)
e.append(-2)
print("e2 = ", e)

del e[2]
print("e3 = ", e)

f = e[1:3]              # array "slicing": elements 1 through 3-1 = 2
print("f = ", f)

print("e4 = ", e[1:])   # all elements starting with index 1

print("e5 = ", e[:2])   # all elements upto but excluding index 2

print("e6 = ", e[-1])   # means "1 item from the right end"

e.insert(2, 25)         # insert 25 at position 2
print("e7 = ", e)

print("e8 = ", e[::2])   # all elements upto end in steps of 2

print(12 in e)          # tests for membership; true, false
print(20 in e)

# finds the index within the list of the given value
print("e9 = ", e.index(20))

e.remove(20)      # If element is not found it returns valueError
print("e10 = ", e)
```

Assignment to slices is also possible, and this can even change the size of the list.

Code 9 File: list2.py

```
a = 5
b = 10
[a, b] = [b, a]   # Elegant way to swap two variables
print("a = ", a, "b = ", b)

# Multidimensional lists can be implemented as lists
c = []
c.append([1, 2])
print("c = ", c)
c.append([3, 4])
print("c1 = ", c)
```

```
x = 4 * [2]
print("x = ", x);

y = 3 * [x]     # [[2,2,2,2],[2,2,2,2],[2,2,2,2]]
print("y = ", y);
print("y1 ", y[0][2])

print(id(y[0]))
print(id(y[1]))
print(id(y[2]))

y[0][2] = 1
print("y2 ", y)
```

Note: The problem is that assignment to y was really a list of three references to the same thing (x). When the object pointed to by y is changed, then all three rows of y are changed.

range() function

If you do need to iterate over a sequence of numbers, use the built-in function range(). It generates lists containing arithmetic progressions:

Python's range() function is an example of the use of lists, i.e. Python arrays, even though not quite explicitly. Lists are fundamental to Python.

Python's `range(1, 10)` function returns a list of consecutive integers, in this case the list [1,2,3,4,5,6,7,8,9]. Note that this is official Python notation for lists - a sequence of objects (these could be all kinds of things, not necessarily numbers), separated by commas and enclosed by brackets.

So, the for statement (`for i in range(1, 10)`) is equivalent to:
`for i in [1,2,3,4,5,6,7,8,9]`

As you can guess, this will result in 9 iterations of the loop, with i first being 1, then 2, etc.

The code
`for i in [2,3,6]:`
would give us three iterations, with i taking on the values 2, 3 and 6.

Code 10 File: sumN.py

```
for i in range(1, 5):
  print (i, " ", i * i)
print()
for i in range(0, 25, 5):
  print (i, " ", i * i)

print()
print (list(range(0, -10, -1)))
```

It is possible to nest lists (create lists containing other lists).

Code 11 File: list3.py

```
data = ["a", 'b', 10, 3]
print(data)
print(data[1], data[1:-1])
print(data[:2], ['c', 3 * 1])

data[2:3] = [20, 30]    # change items
print(data)

data[2:4] = []          # remove items
print(data)

data[2:2] = [15, 18, 22]    # insert items
print(data)
print(len(data))

# nesting of lists
a = [10, 20, 12, 3]
b = [5, a, 30]
print(b, b[1])

a.sort()
print(a)

b[1].append(25)
print(b)
```

Tuples

Tuples are like lists, but are immutable like strings, i.e. unchangeable (it is not possible to assign to the individual items of a tuple). They are enclosed by parentheses or nothing at all, rather than brackets. The parentheses are mandatory if there is an ambiguity without them, e.g. in function arguments. A comma must be used in the case of empty or single tuple, e.g. (,) and (5,).

Tuples have many uses. For example: (x, y) coordinate pairs, employee records from a database, etc. It is also possible to create tuples which contain mutable objects, such as lists.

Code 12 File: tuples.py

```
t1 = (12,5,8)
print(t1[1])
print(len(t1), max(t1), min(t1))
# t1[0] = 10 # illegal, due to immutability

t2 = ("hello", 5)
t3 = t1 + t2
print(t3)
```

```
print(5 in t1)

for val in t1:
  print(val)

print(t1[1:])
del t1
print(t3)
# print(t1)
```

Hash (Dictionary)

Another useful data type is the dictionary (hash) which is also called associative arrays. Unlike sequences, which are indexed by a range of numbers, dictionaries are indexed by keys, which can be any immutable type; strings and numbers can always be keys.

Tuples can be used as keys if they contain only strings, numbers, or tuples; if a tuple contains any mutable object either directly or indirectly, it cannot be used as a key. You can't use lists as keys, since lists can be modified in place using their append() and extend() methods, as well as slice and indexed assignments.

A dictionary is an unordered set of key:value pairs, with the requirement that the keys are unique (within one dictionary). From Python 3.6 onwards, the standard dict type maintains insertion order by default. In 3.6 this was still considered an implementation detail. Python 3.7 elevates this implementation detail to a language specification, so it is now mandatory that dict preserves order in all Python implementations compatible with that version or newer.

A pair of braces creates an empty dictionary {}. Placing a comma-separated list of key:value pairs within the braces adds initial key:value pairs to the dictionary; this is also the way dictionaries are written on output.

The main operations on a dictionary are storing a value with some key and extracting the value given the key. It is also possible to delete a key:value pair with del. If you store using a key that is already in use, the old value associated with that key is overwritten. It is an error to extract a value using a non-existent key.

The keys() method of a dictionary object returns a list of all the keys used in the dictionary, in random order (apply the sort() method to the list of keys if you want it sorted).

Deletion of an element from a dictionary can be done via pop(). The in operator works on dictionary keys.

The dict() constructor can accept an iterator that returns a finite stream of (key, value) tuples.

Code 13 File: dictionary1.py

```
# Execute on Python 2.7 compiler to see the difference in output

months = {4:'April', 2:'Feb', 5:'May'}
months[1] = 'Jan'
print(months)

print (months[2])
months[3] = 'March'

print(months)
print(months.keys())
print(months.values())

del months[2]

for key in months:
  print(key, months[key])

print(months.pop(3))
print(months)

print(5 in months)
print(3 in months)
print(months.get(1))
#del months
months.clear()
print(months)
```

Code 14 File: dictionary2.py

```
Dict = {}

# Creating a Dictionary  with Mixed keys
Dict = {'Name': 'Genesis', 1: [1, 2]}
Dict[1] = 5, 6
Dict[2] = 2, 3, 4
print("\nDictionary with the use of Mixed Keys: ")
Dict[3] = {'Nested' :{1 : 'Hyderabad', 2 : 'Telangana'}}
print(Dict)

print(Dict[3]['Nested'][2])

# Creating a Dictionary with dict() method
Dict = dict({1: 'Genesis', 2: 'Insoft', 3:'Limited'})
print("\nDictionary with the use of dict(): ")
print(Dict)

# Creating a Dictionary with each item as a Pair
Dict = dict([(1, 'Genesis'), (2, 'Hyderabad')])
print("\nDictionary with each item as a pair: ")
```

```
print(Dict)

cubes = {1: 1, 2:8, 3: 27, 4: 64, 5: 125}
for i in cubes:
  print(cubes[i])

print("Len = ", len(cubes))
```

Code 15 File: dictionary sorted.py

```
months_days = {'Jan': 31, 'Feb': 28, 'March': 31}
months_days['April'] = 30
print(months_days.keys())
print(months_days.items())
print(months_days.values())

keys = list(months_days.keys())
print(keys)
keys.sort()
for key in keys:
  print(key, months_days[key])
```

OrderedDict module

An OrderedDict is a dictionary subclass that remembers the order in which its contents are added, supporting the usual dict methods. If a new entry overwrites an existing entry, the original insertion position is left unchanged. Deleting an entry and reinserting it will move it to the end.

Code 16 File: orderDict1.py

```
from collections import OrderedDict

od = OrderedDict()
od['c'] = 1
od['b'] = 2
od['a'] = 3
print(od.items())

d = {}
d['c'] = 1
d['b'] = 2
d['a'] = 3
print(d.items())

# Execute this on Python 2.7 compiler. The output would be as follows:
# [('c', 1), ('b', 2), ('a', 3)]
# [('a', 3), ('c', 1), ('b', 2)]
```

Code 17 File: orderDict2.py

```
from collections import OrderedDict

print("Before:")
od = OrderedDict()
od['a'] = 1
od['b'] = 2
od['c'] = 3
od['d'] = 4
for key, value in od.items():
  print(key, value)

print("\nAfter:")
od['c'] = 5
for key, value in od.items():
  print(key, value)
```

Code 18 File: orderDict3.py

```
from collections import OrderedDict

print("Before deleting:")
od = OrderedDict()
od['a'] = 1
od['b'] = 2
od['c'] = 3
od['d'] = 4

for key, value in od.items():
    print(key, value)

print("\nAfter deleting:")
od.pop('c')
for key, value in od.items():
    print(key, value)

print("\nAfter re-inserting:")
od['c'] = 3
for key, value in od.items():
    print(key, value)
```

A regular dict looks at its contents when testing for equality. An OrderedDict also considers the order in which the items were added.

Code 19 File: orderDict4.py

```
import collections

d1 = {}
d1['a'] = 'A'
d1['b'] = 'B'
d1['c'] = 'C'

d2 = {}
d2['c'] = 'C'
d2['b'] = 'B'
d2['a'] = 'A'

print('dict:', d1 == d2)

d1 = collections.OrderedDict()
d1['a'] = 'A'
d1['b'] = 'B'
d1['c'] = 'C'

d2 = collections.OrderedDict()
d2['c'] = 'C'
d2['b'] = 'B'
d2['a'] = 'A'

print('OrderedDict:', d1 == d2)
```

It is possible to change the order of the keys in an OrderedDict by moving them to either the beginning or the end of the sequence using move_to_end(). The last argument tells move_to_end() whether to move the item to be the last item in the key sequence (when True) or the first (when False).

Code 20 File: orderDict5.py

```
import collections

d = collections.OrderedDict(
  [('a', 'A'), ('b', 'B'), ('c', 'C')]
)

print('Before:')
for k, v in d.items():
  print(k, v)

d.move_to_end('b')

print('\nmove_to_end():')
for k, v in d.items():
  print(k, v)
```

```
d.move_to_end('b', last=False)

print('\nmove_to_end(last=False):')
for k, v in d.items():
  print(k, v)
```

The only difference between dict() and OrderedDict() is that: OrderedDict preserves the order in which the keys are inserted. A regular dict does not track the insertion order, and iterating it gives the values in an arbitrary order. By contrast, the order the items are inserted is remembered by OrderedDict.

Functions

The keyword def introduces a function definition. It must be followed by the function name and the parenthesized list of formal parameters. The statements that form the body of the function start at the next line, and must be indented. The first statement of the function body can optionally be a string literal; this string literal is the function's documentation string, or docstring. There are tools which use docstrings to automatically produce online or printed documentation, or to let the user interactively browse through code; it's good practice to include docstrings in code that you write.

The execution of a function introduces a new symbol table used for the local variables of the function. More precisely, all variable assignments in a function store the value in the local symbol table; whereas variable references first look in the local symbol table, then in the global symbol table, and then in the table of built-in names. Thus, global variables cannot be directly assigned a value within a function (unless named in a global statement), although they may be referenced.

The actual parameters (arguments) to a function call are in the local symbol table of the called function when it is called; thus, arguments are passed using call by value (where the value is always an object reference, not the value of the object). When a function calls another function, a new local symbol table is created for that call.

The return statement returns with a value from a function, return without an expression argument returns None. Falling off the end of a procedure also returns None.

Python supports C-style "printf()", equivalent. Note the importance of writing '(5, 10)' in the following example rather than '5, 10'. In the latter case, the % operator would think that its operand was merely 5, whereas it needs a 2-element tuple. Recall that parentheses enclosing a tuple can be omitted as long as there is no ambiguity.

Code 21 File: func1.py

```
def square(a):
  '''Returns the square of a number'''
  return a * a

print(square.__doc__)
```

```
val = square(3)
print(val)
print(type(val))

val = square
print(val(2))

print("Number is %d" %5)
print("sum of %d and %d is 15" %(5, 10))
print("sum of %.2f %.2f and %.2f is 20" %(5.5, 10, 4.5))
```

The most useful form is to specify a default value for one or more arguments. This creates a function that can be called with fewer arguments than it is defined.

Code 22 File: func2.py

```
# Functions continued
def sumodd(n = 5):
  val = 0
  index = 1
  while (index <= n):
    # if even we continue with next iteration
    if (index % 2 == 0):
      index += 1
      continue

    # if odd we add it
    val += index
    index += 1
  return val

def funNotImplemented(): pass

print("sumodd is", sumodd(3))
print("sumodd is", sumodd())

funNotImplemented()
```

Keyword Argument (named argument)

Functions can also be called using keyword arguments of the form ‘keyword = value’.

Code 23 File: func3.py

```
# keyword arguments

def funckeyword(arg1, arg2='Genesis', arg3='InSoft'):
  print("arg1=", arg1, "arg2=", arg2, "arg3=", arg3)

funckeyword(10)
funckeyword(arg1="value1")
```

```
funckeyword(10, arg2="Qualcomm")
funckeyword(10, arg3="Qualcomm", arg2="Genesis")
funckeyword(arg3="Hyderabad", arg1="value1")
```

Why are the following an error?

```
funckeyword()
funckeyword(arg2 = "Qualcomm")
```

In general, an argument list must have any positional arguments followed by any keyword arguments, where the keywords must be chosen from the formal parameter names. It's not important whether a formal parameter has a default value or not. No argument may receive a value more than once - formal parameter names corresponding to positional arguments cannot be used as keywords in the same calls. Here's an example that fails due to this restriction:

Why does the following code snippet result in an error?

```
def function(a):
   pass

function(0, a = 0)
```

Multiple Function Arguments are functions which receive a variable number of arguments, using the *.

Code 24 File: func4.py

```
# variable arguments
def funcvarArgs(first, *arguments):
  print("value of first argument:", first)
  for arg in arguments:
    print(arg)

funcvarArgs(10)
funcvarArgs(20, "How")
funcvarArgs("Hello", "How", "are", "you")
```

When a final formal parameter of the form **name is present, it receives a dictionary containing all keyword arguments whose keyword does not correspond to a formal parameter. This may be combined with a formal parameter of the form *name which receives a tuple containing the positional arguments beyond the formal parameter list. (*name must occur before **name).

Code 25 File: func5.py

```
# dictionary arguments
def funcdictionary(**keywords):
  keys = keywords.keys()
  for kw in keys:
    print (kw, ':', keywords[kw])
```

```
funcdictionary(k1 = 'v1')
funcdictionary(k1='v1', k2='v2', k3='v3')

months_days = {'Jan': 31, 'Feb': 28, 'March': 31, 'April':30}
funcdictionary(**months_days)
```

Code 26 File: func6.py

```
def funcVarDictionary(first, *arguments, **keywords):
  print("value of first argument:", first)
  for arg in arguments:
    print(arg)

  keys = keywords.keys()
  for kw in keys:
    print (kw, ':', keywords[kw])

funcVarDictionary(10, "How", k1 = 'v1')
funcVarDictionary("Hello", "How", "are", "you", k1='v1', k2='v2',
k3='v3')

months_days = {'Jan': 31, 'Feb': 28, 'March': 31, 'April':30}
funcVarDictionary("months", **months_days)
```

Code 27 File: fibonacci.py

```
# Fibonacci series: the sum of two elements defines the next
def fib1():
  n1, n2 = 0, 1
  while n2 < 15:
    print(n2)
    n1, n2 = n2, n1 + n2

# Return a list containing the Fibonacci series up to n
def fib2(n):
  result = []
  a, b = 0, 1
  while b < n:
    result.append(b)
    a, b = b, a + b
  return result

print (__name__)

if __name__ == "__main__":
  fib1()
  print("After call to fib1()")
  print(fib2(20))
```

Modules

A module is a file containing Python definitions and statements. The file name is the module name with the suffix '.py' appended. Within a module, the module's name (as a string) is available as the value of the global variable __name__.

Definitions from a module can be imported into other modules or into the main module.

Lets use the fibonacci module (fibonacci.py) we looked at earlier. Now enter the Python interpreter and import this module with the following command.

```
D:\Python39>python
>>> import Fibonacci
```

This does not enter the names of the functions defined in Fibonacci directly in the current symbol table; it only enters the module name fibonacci. Using the module name you can access the functions as shown below.

```
>>> fibonacci.fib1()
>>> fibonacci.fib2(20)
```

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

```
>>> func = fibonacci.fib2(20)
>>> func
```

A module can contain executable statements as well as function definitions. These statements are intended to initialize the module. They are executed only the first time the module is imported somewhere. Each module has its own private symbol table, which is used as the global symbol table by all functions defined in the module. Thus, the author of a module can use global variables in the module without worrying about accidental clashes with a user's global variables. On the other hand you can access a module's global variables with the same notation used to refer to its functions, `modname.itemname`. Modules can import other modules. It is customary but not required to place all import statements at the beginning of a module (or script, for that matter). The imported module names are placed in the importing module's global symbol table.

There is a variant of the import statement that imports names from a module directly into the importing module's symbol table. For example:

```
>>> from fibonacci import fib1, fib2
>>> fib1()
>>> fib2(10)
```

The above does not introduce the module name from which the imports are taken in the local symbol table. Fibonacci is not defined, hence the following call would result in an error

```
>>> fibonacci.fib1()
```

There is even a variant to import all names that a module defines.

```
>>> from fibonacci import *
```

This imports all names except those beginning with an underscore (_).

Note: Change function definition `fib2(n)` to `_fib2(n)` and run the program. Check the error generated.

When a module named fibonacci is imported, the interpreter searches for a file named 'fibonacci.py' in the current directory, and then in the list of directories specified by the environment variable PYTHONPATH.

On my system it is set to '`D:\Python39\Lib;.;`' where `d:\Python39` is the folder where Python is installed. The dot (`.`)signifies the current folder.

Actually, modules are searched in the list of directories given by the variable `sys.path` which is initialized from the directory containing the input script (or the current directory), PYTHONPATH and the installation-dependent default.

Note: Because the directory containing the script being run is on the search path, it is important that the script not have the same name as a standard module, or Python will attempt to load the script as a module when that module is imported. This will generally be an error.

As an important speed-up of the start-up time for short programs that use a lot of standard modules, if a file called 'fibonacci.pyc' exists in the directory where 'fibonacci.py' is found, this is assumed to contain an already "byte-compiled" version of the module fibonacci. The modification time of the version of 'fibonacci.py' used to create 'fibonacci.pyc' is recorded in 'fibonacci.pyc', and the '.pyc' file is ignored if these don't match.

You don't need to do anything to create the 'fibonacci.pyc' file. Whenever 'fibonacci.py' is successfully compiled, an attempt is made to write the compiled version to 'fibonacci.pyc'. It is not an error if this attempt fails; if for any reason the file is not written completely, the resulting 'fibonacci.pyc' file will be recognized as invalid and thus ignored later. The contents of the 'fibonacci.pyc' file are platform independent, so a Python module directory can be shared by machines of different architectures.

To get a list of options available to generate the byte code, try the following at the command prompt.

```
Python -h
```

- When the Python interpreter is invoked with the -O flag, optimized code is generated and stored in '.pyo' files. The optimizer currently doesn't help much; it only removes assert statements. When -O is used, all bytecode is optimized; .pyc files are ignored and .py files are compiled to optimized bytecode.

- Passing two -O flags to the Python interpreter (-OO) will cause the bytecode compiler to perform optimizations that could in some rare cases result in malfunctioning programs. Currently only __doc__ strings are removed from the bytecode, resulting in more compact '.pyo' files. Since some programs may rely on having these available, you should use this option with caution.

- A program doesn't run any faster when it is read from a '.pyc' or '.pyo' file than when it is read from a '.py' file; the only thing that's faster about '.pyc' or '.pyo' files is the speed with which they are loaded.

- It is possible to have a file called 'fibonacci.pyc' (or 'fibonacci.pyo' when -O is used) without a file 'fibonacci.py' for the same module. This can be used to distribute a library of Python code in a form that is moderately hard to reverse engineer.

Code 28 File: assert.py

```
import sys

# __debug__ is true by default, unless we run
# using -O (optimized code)
# python -O assert.py

print(__debug__)

# assert comes into affect only when __debug__ is true

num = int(input('Enter a positive number: '))
print(num)
assert(num > 0), 'Only positive numbers are allowed!'

def chkassert(num):
  assert(type(num) == int)

chkassert('india')

sys.exit()
```

Standard Modules

Python comes with a library of standard modules, the Library Reference. Some modules are built into the interpreter; these provide access to operations that are not part of the core of the language but are nevertheless built in, either for efficiency or to provide access to operating system primitives such as system calls. The set of such modules is a configuration option which also depends on the underlying platform. One particular module deserves attention: sys, which is built into every Python interpreter. The variables sys.ps1 and sys.ps2 define the strings used as primary and secondary prompts.

```
D:\Python39>python
>>> import sys
```

```
>>> sys.ps1
'>>> '
>>> sys.ps2
'... '
```

The above two variables are only defined if the interpreter is in interactive mode.

The variable sys.path is a list of strings that determine the interpreter's search path for modules. It is initialized to a default path taken from the environment variable PYTHONPATH, or from a built-in default if PYTHONPATH is not set.

```
>>> sys.path
['','D:\\Python39\\Demos','D:\\Python39',
'C:\\WINDOWS\\system32\\Python39.zip', 'D:\\Python39\\DLLs',
'D:\\Python\lib', 'D:\\Python39\\lib\\plat-win',
'D:\\Python39\\lib\\site-packages']
>>>
```

You can modify it using standard list operations:

```
>>> sys.path.append('c:\demos')
```

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

```
>>> import fibonacci
>>> dir (fibonacci)
['__builtins__', '__doc__', '__file__', '__name__', '__package__',
'fib1', 'fib2', 'value']
```

Note: Execute dir(sys) to see the names defined in sys module.

Without arguments, dir() lists the names you have defined currently. It lists all types of names: variables, modules, functions, etc.

```
D:\Python39>python
>>> a=10
>>> print(a)
10
>>> import fibonacci, sys
>>> fibonacci.fib1()
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__', 'a',
'fibonacci', 'sys']
```

Code 29 File: mathdemo.py

```
from math import *

print(sqrt(25))
print(pow(2, 3))
print(pi)
print(trunc(-2.3))
print(cos(0))
print(sin(0))
print(fabs(-2.3))
print(factorial(5))
print(fmod(5, 2))
print(fmod(5, -2))
print(5 % 2)
print(5 % -2)
print(modf(7.8))
```

Namespace

A namespace is a container for a set of identifiers. Namespaces provide a level of direction to specific identifiers, thus making it possible to distinguish between identifiers with the same exact name.

In Python, namespaces are defined by the individual modules, and since modules can be contained in hierarchical packages, then namespaces are hierarchical too. In general when a module is imported then the names defined in the module are defined via that module's namespace, and are accessed in from the calling modules by using the fully qualified name.

Examples of namespaces are: the set of built-in names (functions such as abs(), and built-in exception names); the global names in a module; and the local names in a function invocation. In a sense the set of attributes of an object also form a namespace. The important thing to know about namespaces is that there is absolutely no relation between names in different namespaces; for instance, two different modules may both define a function "maximize" without confusion - users of the modules must prefix it with the module name.

Scope resolution for variable names via the LEGB rule:

We have seen that multiple namespaces can exist independently from each other and that they can contain the same variable names on different hierarchy levels. The "scope" defines on which hierarchy level Python searches for a particular "variable name" for its associated object.

Python uses the LEGB rule to search the different levels of namespaces before it finds the name-to-object mapping.

```
Local -> Enclosed -> Global -> Built-in
```

where the arrows denote the direction of the namespace-hierarchy search order.

- Local can be inside a function or class method.
- Enclosed can be its enclosing function, e.g., if a function is wrapped inside another function.
- Global refers to the uppermost level of the executing script itself, and
- Built-in are special names that Python reserves for itself.

So, if a particular name:object mapping is not found in the local namespaces, the namespaces of the enclosed scope is searched next. If the search in the enclosed scope is unsuccessful, too, Python moves on to the global namespace, and eventually, it will search the built-in namespace (if a name cannot found in any of the namespaces, a NameError is raised).

Code 30 File: nestedfunc.py

```
lst = []
num = 10

def outer(n):
  lst.append(n)
  print(lst)

  global num
  num = 20
  print(num)

  def inner():
    num = 30
    print(num)
  return inner

func = outer(2)
func()
func = outer(3)
func()
print(num)
```

Packages

Packages are a way of structuring Python's module namespace by using "dotted module names". For example, the module name A.B designates a submodule named 'B' in a package named 'A'. Just like the use of modules saves the authors of different modules from having to worry about each other's global variable names, the use of dotted module names saves the authors of multi-module packages from name clashes.

Files named __init__.py are used to mark directories on disk as a Python package directories. The primary use of __init__.py is to initialize Python packages. The easiest way to demonstrate this is to take a look at the structure of a standard Python module.

```
--+ PackageDemo
  |-- mod1.py
  |-- mod12Demo.py
```

```
|-- mod2.py
|-- __init__.py
```

As you can see in the structure above the inclusion of the __init__.py file in a directory indicates to the Python interpreter that the directory should be treated like a Python package.

The '__init__.py' can just be an empty file, but it can also execute initialization code for the package or set the __all__ variable.

Note: Python 3.3+ has Implicit Namespace Packages that allow it to create a package without an __init__.py file. This however, only applies to empty __init__.py files. So empty __init__.py files are no longer necessary and can be omitted. If you want to run a particular initialization script when the package or any of its modules or sub-packages are imported, you still require an __init__.py file

When using from package import item, the item can be either a submodule (or subpackage) of the package, or some other name defined in the package, like a function, class or variable. The import statement first tests whether the item is defined in the package; if not, it assumes it is a module and attempts to load it. If it fails to find it, an ImportError exception is raised.

Contrarily, when using syntax like import item.subitem.subsubitem, each item except for the last must be a package; the last item can be a module or a package but can't be a class or function or variable defined in the previous item.

If a package '__init__.py' code defines a list named __all__, it is taken to be the list of module names that should be imported when from package import * is encountered. It is up to the package author to keep this list up-to-date when a new version of the package is released. Package authors may also decide not to support it, if they don't see a use for importing * from their package.

If __all__ is not defined, the statement from PackageDemo import * does not import all submodules from the package; it only ensures that the package PackageDemo has been imported (possibly running its initialization code, '__init__.py') and then imports whatever names are defined in the package. This includes any names defined (and submodules explicitly loaded) by '__init__.py'. It also includes any submodules of the package that were explicitly loaded by previous import statements.

Note: You can move the PackageDemo to Lib\site-packages folder within a python installation.

Code 31 File: mod1.py (PackageDemo folder)

```
import PackageDemo.mod2

def f():
  global x
  x = 6

def getX():
  return x

def main():
  x = 5
  f()
  print(x)
  PackageDemo.mod2.g()
  x += 2
  print(x)
```

Code 32 File: mod2.py (PackageDemo folder)

```
import PackageDemo.mod1

def g():
  x = 10
  print(x)
  print(PackageDemo.mod1.getX())
```

Code 33 File: mod12Demo.py (PackageDemo folder)

```
from PackageDemo import *

if __name__ == '__main__':
   mod1.main()
```

Note: Execute the above using at the command prompt

```
python PackageDemo/mod12Demo.py
```

File I/O:
Python provides basic functions and methods necessary to manipulate files using a file object.

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

```
fileobject = open(file_name [, access_mode][, buffering])
```

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 will take place. If the buffering value is 1, line buffering will be performed while accessing a file. If you specify the buffering value as an integer greater than 1, then buffering action will be 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 will be at the beginning of the file.
rb+	Opens a file for both reading and writing in binary format. The file pointer will be 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.

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

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.

Reading one line at a time has the nice quality that not all the file needs to fit in memory at one time - handy if you want to look at every line in a 10 gigabyte file without using 10 gigabytes of memory. The f.readlines() method reads the whole file into memory and returns its contents as a list of its lines. The f.read() method reads the whole file into a single string, which can be a handy way to deal with the text all at once.

For writing, f.write(string) method is the easiest way to write data to an open output file.
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.

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.

Code 34 File: fileio1.py

```
import io
print (io.DEFAULT_BUFFER_SIZE)

filein = open('first.py')
fileout = open('firstold.py', 'w')

print(type(filein))

def read():
  print(filein.read())
  filein.seek(0,0)

def readlines():
  print(filein.readlines())
```

```
  position = filein.seek(0, 0);

def readline():
  for line in filein:
    fileout.write(line)
  position = filein.seek(0, 0);

read()
readlines()
readline()

position = filein.tell();
print("File position 1:", position)

def readchars(n):
  str = filein.read(n);
  position = filein.tell();
  print("File position 2:", str, " ", position)

readchars(20)

print(filein.mode)
print(filein.closed)
print(filein.name)

filein.close()
fileout.close()

print(filein.closed)
```

File: customer.txt

```
100,ravi,hyderabad,sales
200,subbu,mumbai,marketing
300,kavita,hyderabad,hr
400,ram,pune,sales
500,bharat,chennai,finance
```

Code 35 File: customer.py

```
import io
import os

filein = open('customer.txt', 'r')

for line in filein:
  line = line.rstrip(os.linesep)
  data = line.split(',')
  print(data, data[0], data[3])

filein.close()
```

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 and then you can call any related functions.

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

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

The os module has several methods that help you create, remove and change directories.

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.

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.

The getcwd() method displays the current working directory.

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.

Code 36 File: fileio2.py

```
import os
import shutil

# Create a directory 'test' and copy all files from current folder to
'test'
os.mkdir("test")
lst = os.listdir(".")
for f in lst:
   if(os.path.isfile(f)):
      shutil.copy(f, 'test')


# Change to test directory and list its contents
os.chdir("test")
print(os.getcwd())

os.chdir("..")
lst = os.listdir("test")
print (lst)


# Remove all .py files from test directory
os.chdir("test")
count = 0
```

```
def removefile(extension):
  for item in lst:
    if item.endswith(extension):
      global count
      count += 1
      print(item)
      os.remove(item)

removefile(".py")

print("count of python files:", count)

# Go to parent directory and remove the test folder
os.chdir("..")
os.rmdir("test")
```

Built in tools

These libraries help you with Python development: the debugger enables you to step through code, analyze stack frames and set breakpoints etc.

The module pdb defines an interactive source code debugger for Python programs. It supports setting (conditional) breakpoints and single stepping at the source line level, inspection of stack frames, source code listing, and evaluation of arbitrary Python code in the context of any stack frame.

The debugger is extensible - it is actually defined as the class Pdb. The extension interface uses the modules bdb and cmd.

The debugger's prompt is (Pdb). pdb.py can also be invoked as a script to debug other scripts.

```
python -m pdb func2.py
```

`s(tep)`
Execute the current line, stop at the first possible occasion (either in a function that is called or on the next line in the current function).

`n(ext)`
Continue execution until the next line in the current function is reached or it returns. (The difference between next and step is that step stops inside a called function, while next executes called functions at (nearly) full speed, only stopping at the next line in the current function.)

`unt(il)`
Continue execution until the line with the line number greater than the current one is reached or when returning from current frame.

`r(eturn)`
Continue execution until the current function returns.

`c(ont(inue))`
Continue execution, only stop when a breakpoint is encountered.

`l(ist) [first[, last]]`
List source code for the current file. Without arguments, list 11 lines around the current line or continue the previous listing. With one argument, list 11 lines around at that line. With two arguments, list the given range; if the second argument is less than the first, it is interpreted as a count.

`a(rgs)`
Print the argument list of the current function.
`p(rint) expression`
Evaluate the expression in the current context and print its value.

`run [args ...]`
Restart the debugged python program (with without an argument). History, breakpoints, actions and debugger options are preserved. “restart” is an alias for “run”.

`q(uit)`
Quit from the debugger. The program being executed is aborted.

The distutils package provides support for building and installing additional modules into a Python installation. The new modules may be either 100% pure Python, or may be extension modules written in C, or may be collections of Python packages which include modules coded in both Python and C.

As a developer, your responsibilities are:
- write a setup script (setup.py by convention)
- (optional) write a setup configuration file
- create a source distribution
- (optional) create one or more built (binary) distributions

Code 37 File: setup.py

```
from distutils.core import setup
setup(name='sub',
  version='1.0',
  author='Raj Tapadia',
  author_email='rtapadia@genesisinsoft.com',
  url='http://www.genesisinsoft.com/',
  packages=['sub'],
  package_dir={'sub': 'PackageDemo'},
  )
```

To create a source distribution for this module, you would create a setup script, setup.py, containing the above code, and run:

```
python setup.py sdist
```

which will create an archive file (e.g., tar on Unix, ZIP file on Windows) containing your setup script setup.py, and your package sub. The archive file will be named sub-1.0.tar.gz (or .zip), and will unpack into a directory sub-1.0.

To create a windows msi file (setup file), use the following command:

```
python setup.py bdist_msi
```

If an end-user wishes to install your sub package, all they have to do is run this command:

```
python setup.py install
```

which will ultimately copy sub package to the appropriate directory for third-party modules in their Python installation.