

DIGITAL NOTES ON DATA STRUCTURES USING PYTHON

**B.TECH II YEAR - II SEM
(2019-20)**



DEPARTMENT OF INFORMATION TECHNOLOGY

**MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY
(Autonomous Institution – UGC, Govt. of India)**

(Affiliated to JNTUH, Hyderabad, Approved by AICTE - Accredited by NBA & NAAC – ‘A’ Grade - ISO 9001:2015 Certified)
Maisammaguda, Dhulapally (Post Via. Hakimpet), Secunderabad– 500100, Telangana State, INDIA.



MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY
DEPARTMENT OF INFORMATION TECHNOLOGY

(R18A0553) DATA STRUCTURES USING PYTHON

UNIT I

Introduction to Python, Installation and Working with Python, Understanding Python variables Python basic Operators, Understanding python blocks, Python Data Types: Declaring and using Numeric data types: int, float, complex, Using string data type and string operations.

UNIT II

Control Flow- if, if-elif-else, loops ,For loop using ranges, string ,Use of while loops in python, Loop manipulation using pass, continue, break and else, Programming using Python conditional and loops block, Python arrays.

UNIT III

Functions -Calling Functions, Passing Arguments, Keyword Arguments, Default Arguments, Variable-length arguments, Anonymous Functions, Fruitful Functions(Function Returning Values), Scope of the Variables in a Function - Global and Local Variables. Powerful Lambda function in python.

UNIT IV

Data Structures-List Operations, Slicing, Methods; Tuples, Sets, Dictionaries, Sequences. Comprehensions, Dictionary manipulation, list and dictionary in build functions

UNIT V

Sorting: Bubble Sort, Selection Sort, Insertion Sort, Merge sort, Quick sort, Linked Lists, Stacks, Queues

TEXT BOOKS:

1. Allen B. Downey, ``Think Python: How to Think Like a Computer Scientist``, 2nd edition, Updated for Python 3, Shroff/O'Reilly Publishers, 2016.
2. R. Nageswara Rao, “Core Python Programming”, dreamtech
3. Python Programming: A Modern Approach, Vamsi Kurama, Pearson

REFERENCE BOOKS:

1. Core Python Programming, W.Chun, Pearson.
2. Introduction to Python, Kenneth A. Lambert, Cengage
3. Learning Python, Mark Lutz, Orielly



MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY
DEPARTMENT OF INFORMATION TECHNOLOGY

INDEX

S. No	Unit	Topic	Page no
1	I	Introduction to Python, Installation of Python	1
2	I	Variables in Python	13
3	I	Operators in Python	21
4	I	Data types in Python	16
5	II	Control Flow Statements	28
6	II	Arrays in Python	42
7	III	Functions	45
8	III	Scope of Variables	52
9	III	Anonymous Functions	55
10	IV	Lists	61
11	IV	Tuples	68
12	IV	Dictionaries	73
13	V	Sorting Techniques	76
14	V	Linked Lists	79
15	V	Stacks	84
16	V	Queues	86



Unit-I



Definition:

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.
- At the time when he began implementing Python, Guido van Rossum was also reading the published scripts from "Monty Python's Flying Circus" (a BBC comedy series from the seventies, in the unlikely case you didn't know). It occurred to him that he needed a name that was short, unique, and slightly mysterious, so he decided to call the language Python.
- 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 1.0 was released on **20 February, 1991**.
- Python 2.0 was released on **16 October 2000** and had many major new features, including a cycle detecting garbage collector and support for Unicode. With this release the development process was changed and became more transparent and community-backed.
- Python 3.0 (which early in its development was commonly referred to as Python 3000 or py3k), a major, backwards-incompatible release, was released on **3 December 2008** after a long period of testing. Many of its major features have been back ported to the



backwards-compatible Python 2.6.x and 2.7.x version series.

- In January 2017 Google announced work on a Python 2.7 to go transcompiler, which The Register speculated was in response to Python 2.7's planned end-of-life.

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.

Need of Python Programming

➤ Software quality

Python code is designed to be *readable*, and hence reusable and maintainable—much more so than traditional scripting languages. The uniformity of Python code makes it easy to understand, even if you did not write it. In addition, Python has deep support for more advanced *software reuse* mechanisms, such as object-oriented (OO) and function programming.

➤ Developer productivity

Python boosts developer productivity many times beyond compiled or statically typed languages such as C, C++, and Java. Python code is typically *one-third to* less to debug, and less to maintain after the fact. Python programs also run immediately, without the lengthy compile and link steps required by some other tools, further boosting programmer speed. *Program portability* Most Python programs run unchanged on *all major computer platforms*. Porting Python code between Linux and Windows, for example, is usually just a matter of copying a script's code between machines.

➤ Support libraries

Python comes with a large collection of prebuilt and portable functionality, known as the *standard library*. This library supports an array of application-level programming tasks, from text pattern matching to network scripting. In addition, Python can be extended with both home grown libraries and a vast collection of third-party

application support software. Python's *third-party domain* offers tools for website construction, numeric programming, serial port access, game development, and much more (see ahead for a sampling).

➤ **Component integration**

Python scripts can easily communicate with other parts of an application, using a variety of integration mechanisms. Such integrations allow Python to be used as a product *customization and extension* tool. Today, Python code can invoke C and C++ libraries, can be called from C and C++ programs, can integrate with Java and .NET components, can communicate over frameworks such as COM and Silverlight, can interface with devices over serial ports, and can interact over networks with interfaces like SOAP, XML-RPC, and CORBA. It is not a standalone tool.

➤ **Enjoyment**

Because of Python's ease of use and built-in toolset, it can make the act of programming *more pleasure than chore*. Although this may be an intangible benefit, its effect on productivity is an important asset. Of these factors, the first two (quality and productivity) are probably the most compelling benefits to most Python users, and merit a fuller description.

➤ **It's Object-Oriented**

Python is an object-oriented language, from the ground up. Its class model supports advanced notions such as polymorphism, operator overloading, and multiple inheritance; yet in the context of Python's dynamic typing, object-oriented programming (OOP) is remarkably easy to apply. Python's OOP nature makes it ideal as a scripting tool for object-oriented systems languages such as C++ and Java. For example, Python programs can subclass (specialized) classes implemented in C++ or Java.

➤ **It's Free**

Python is freeware—something which has lately been come to be called *open source software*. As with Tcl and Perl, you can get the entire system for free over the Internet. There are no restrictions on copying it, embedding it in your systems, or shipping it with your products. In fact, you can even sell Python, if you're so inclined. But don't get the wrong idea: "free" doesn't mean "unsupported". On the contrary, the Python online community responds to user queries with a speed that most commercial software vendors would do well to notice.

➤ **It's Portable**

Python is written in portable ANSI C, and compiles and runs on virtually every major platform in use today. For example, it runs on UNIX systems, Linux, MS-DOS, MS-Windows (95, 98, NT), Macintosh, Amiga, Be-OS, OS/2, VMS, QNX, and more. Further, Python programs are automatically compiled to portable *bytecode*, which runs the same on any platform with a compatible version of Python installed (more on this in the section "It's easy to use"). What that means is that Python programs that use the core language run the same on UNIX, MS-Windows, and any other system with a Python interpreter.

➤ **It's Powerful**

From a features perspective, Python is something of a hybrid. Its tool set places it between traditional scripting languages (such as Tcl, Scheme, and Perl), and systems

languages (such as C, C++, and Java). Python provides all the simplicity and ease of use of a scripting language, along with more advanced programming tools typically found in systems development languages.

➤ **Automatic memorymanagement**

Python automatically allocates and reclaims ("garbage collects") objects when no longer used, and most grow and shrink on demand; Python, not you, keeps track of low-level memory details.

➤ **Programming-in-the-large support**

Finally, for building larger systems, Python includes tools such as modules, classes, and exceptions; they allow you to organize systems into components, do OOP, and handle events gracefully.

➤ **It's Mixable**

Python programs can be easily "glued" to components written in other languages. In technical terms, by employing the Python/C integration APIs, Python programs can be both extended by (called to) components written in C or C++, and embedded in (called by) C or C++ programs. That means you can add functionality to the Python system as needed and use Python programs within other environments or systems.

➤ **It's Easy to Use**

For many, Python's combination of rapid turnaround and language simplicity make programming more fun than work. To run a Python program, you simply type it and run it. There are no intermediate compile and link steps (as when using languages such as C or C++). As with other interpreted languages, Python executes programs immediately, which makes for both an interactive programming experience and rapid turnaround after program changes. Strictly speaking, Python programs are compiled (translated) to an intermediate form called *bytecode*, which is then run by the interpreter.

➤ **It's Easy to Learn**

This brings us to the topic of this book: compared to other programming languages, the core Python language is amazingly easy to learn. In fact, you can expect to be coding significant Python programs in a matter of days (and perhaps in just hours, if you're already an experienced programmer).

➤ **Internet Scripting**

Python comes with standard Internet utility modules that allow Python programs to communicate over sockets, extract form information sent to a server-side CGI script, parse HTML, transfer files by FTP, process XML files, and much more. There are also a number of peripheral tools for doing Internet programming in Python. For instance, the HTMLGen and pythondoc systems generate HTML files from Python class-based descriptions, and the JPython system mentioned above provides for seamless Python/Java integration.

➤ **Database Programming**

Python's standard pickle module provides a simple object-persistence system: it allows programs to easily save and restore entire Python objects to files. For more traditional database demands, there are Python interfaces to Sybase, Oracle, Informix, ODBC, and more. There is even a portable SQL database API for Python that runs the same on a variety of underlying database systems, and a system named *gadfly* that

implements an SQL database for Python programs.

➤ **Image Processing, AI, Distributed Objects, Etc.**

Python is commonly applied in more domains than can be mentioned here. But in general, many are just instances of Python's component integration role in action. By adding Python as a frontend to libraries of components written in a compiled language such as C, Python becomes useful for scripting in a variety of domains. For instance, image processing for Python is implemented as a set of library components implemented in a compiled language such as C, along with a Python frontend layer on top used to configure and launch the compiled components.

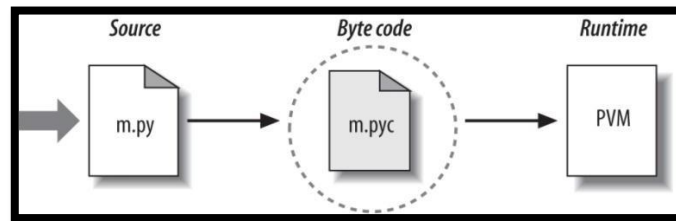
Who Uses Python Today?

1. *Google* makes extensive use of Python in its web search systems.
2. The popular *YouTube* video sharing service is largely written in Python.
3. The *Dropbox* storage service codes both its server and desktop client software primarily in Python.
4. The *Raspberry Pi* single-board computer promotes Python as its educational language.
5. The widespread *BitTorrent* peer-to-peer file sharing system began its life as a Python program.
6. Google's *App Engine* web development framework uses Python as an application language.
7. *Maya*, a powerful integrated 3D modeling and animation system, provides a Python scripting API.
8. *Intel*, *Cisco*, *Hewlett-Packard*, *Seagate*, *Qualcomm*, and *IBM* use Python for hardware testing.
9. *NASA*, *Los Alamos*, *Fermilab*, *JPL*, and others use Python for scientific programming tasks.

Byte code Compilation:

Python first compiles your source code (the statements in your file) into a format known as byte code. Compilation is simply a translation step, and byte code is a lower-level, platform independent representation of your source code. Roughly, Python translates each of your source statements into a group of byte code instructions by decomposing them into individual steps. This byte code translation is performed to speed execution — byte code can be run much more quickly than the original source code statements in your textfile.

The Python Virtual Machine:



Once your program has been compiled to byte code (or the byte code has been loaded from existing .pycfile), it is shipped off for execution to something generally known as the python virtual machine (PVM).

Applications of Python:

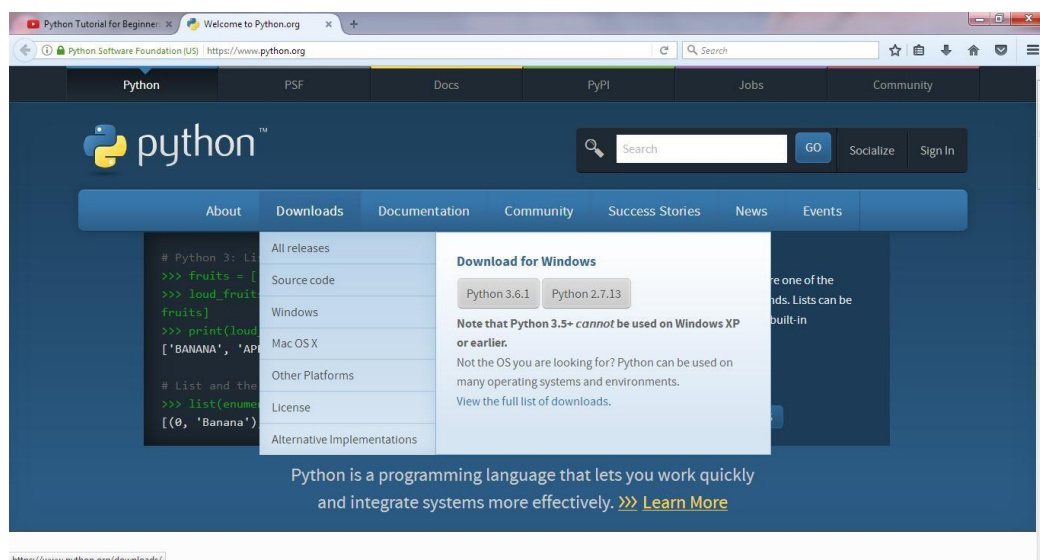
1. SystemsProgramming
2. GUIs
3. InternetScripting
4. ComponentIntegration
5. DatabaseProgramming
6. RapidPrototyping
7. Numeric and ScientificProgramming

What Are Python's Technical Strengths?

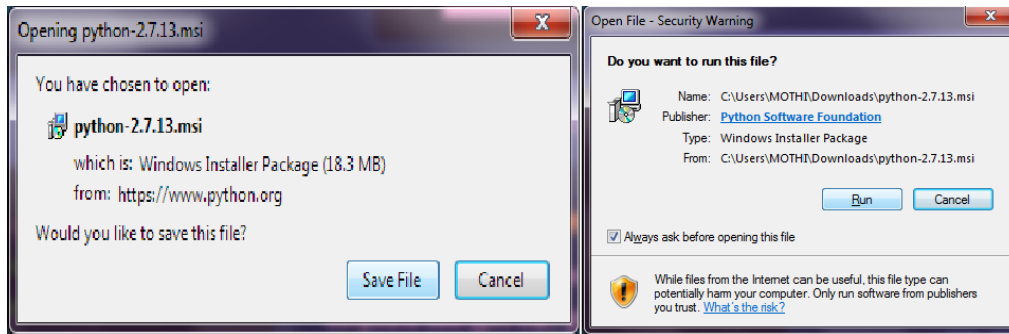
1. It's Object-Oriented andFunctional
2. It'sFree
3. It'sPortable
4. It'sPowerful
5. It'sMixable
6. It's Relatively Easy toUse
7. It's Relatively Easy toLearn

Download and installation Python software:

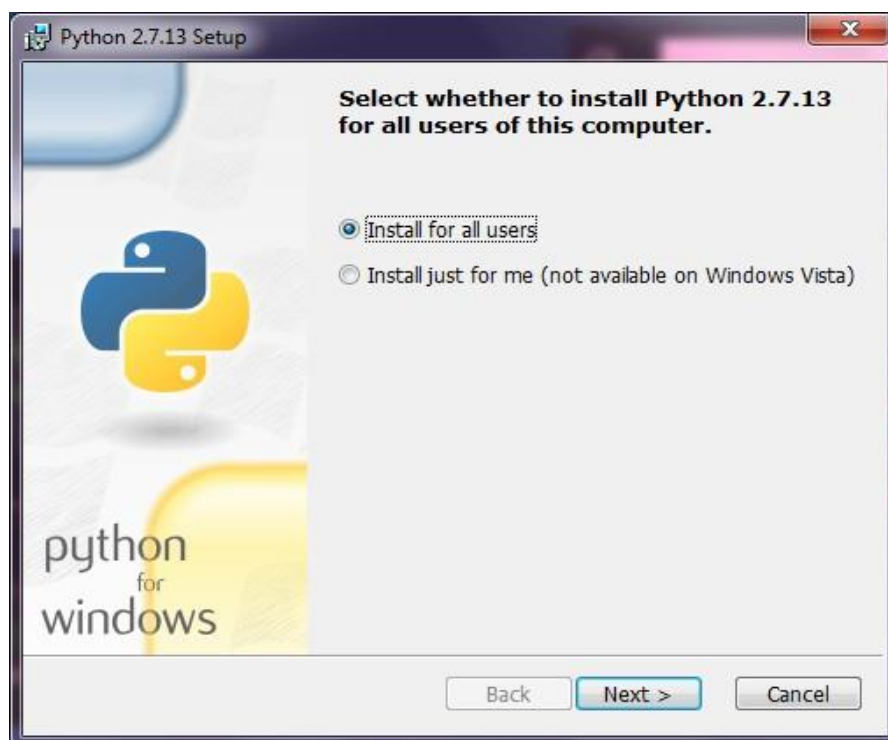
Step 1: Go to website www.python.org and click downloads select version which you want.



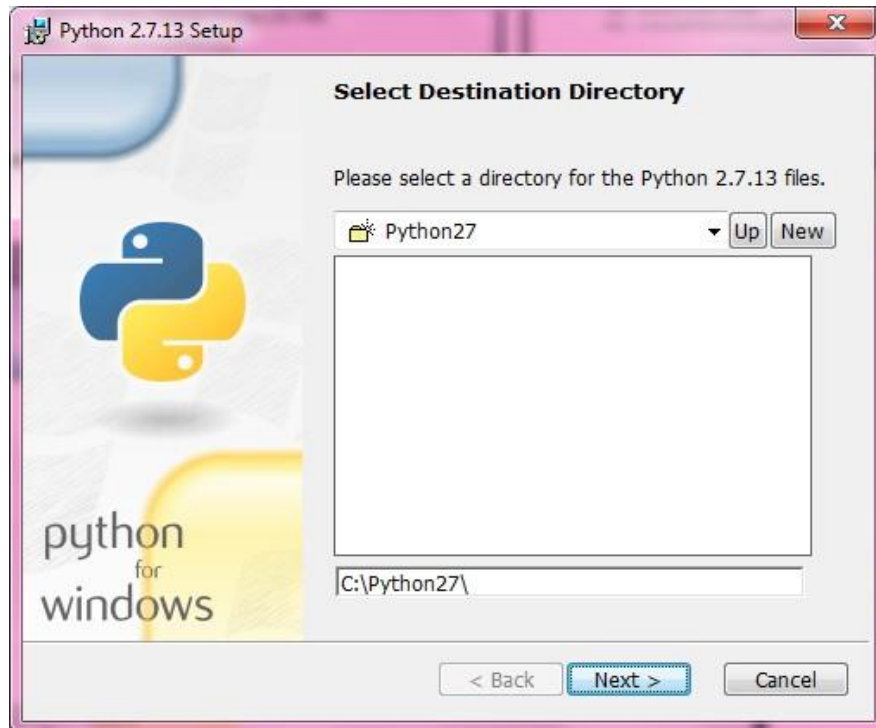
Step 2: Click on **Python 2.7.13** and download. After download open the file.



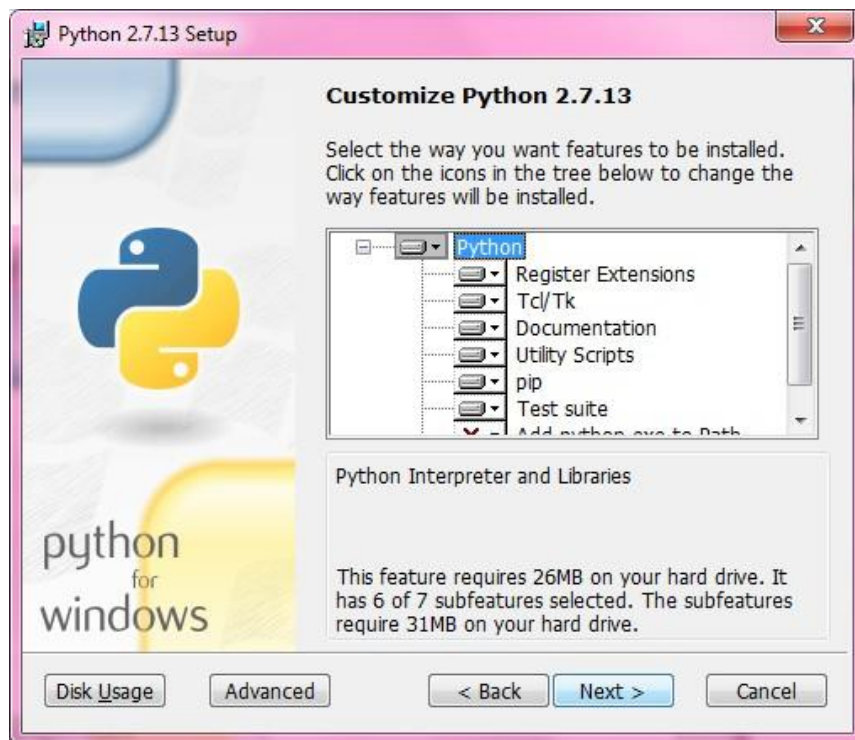
Step 3: Click on **Next** to continue.



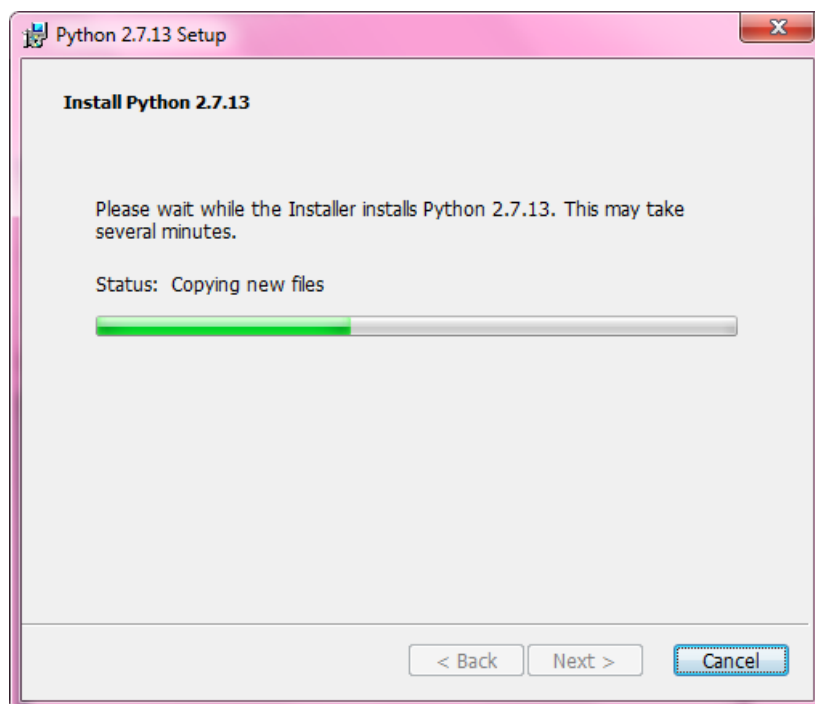
Step 4: After installation location will be displayed. The Default location is **C:\Python27**. Click on next to continue.



Step 5: After the python interpreter and libraries are displayed for installation. Click on Next to continue.



Step 6: The installation has been processed.

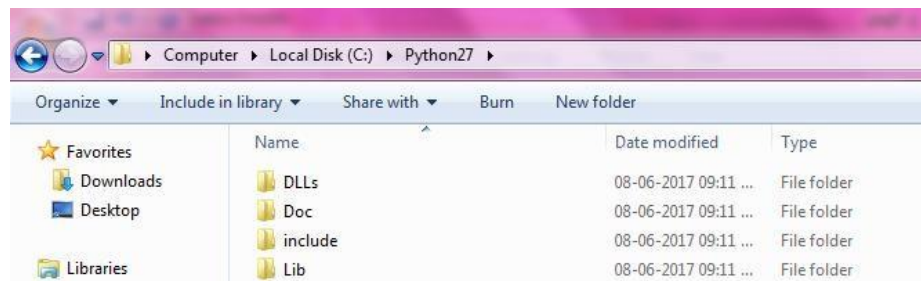


Step 7: Click the **Finish** to complete the installation.

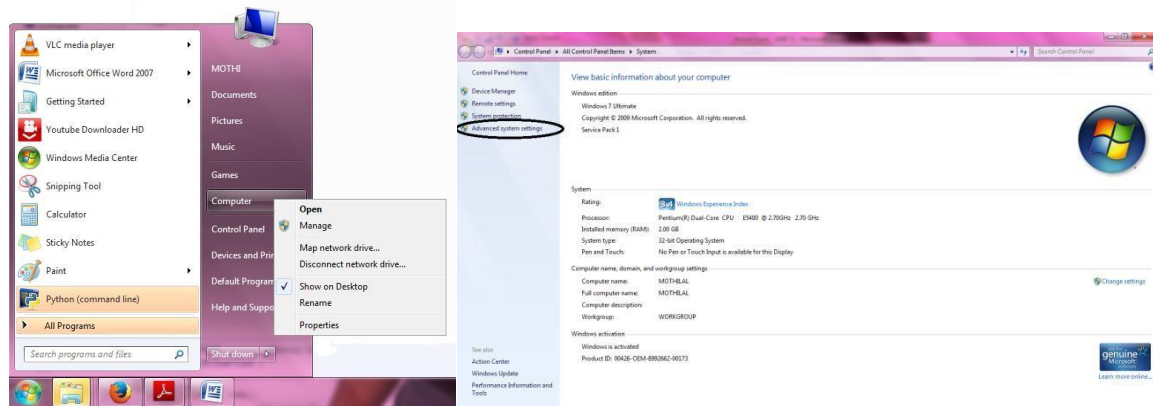


Setting up PATH to python:

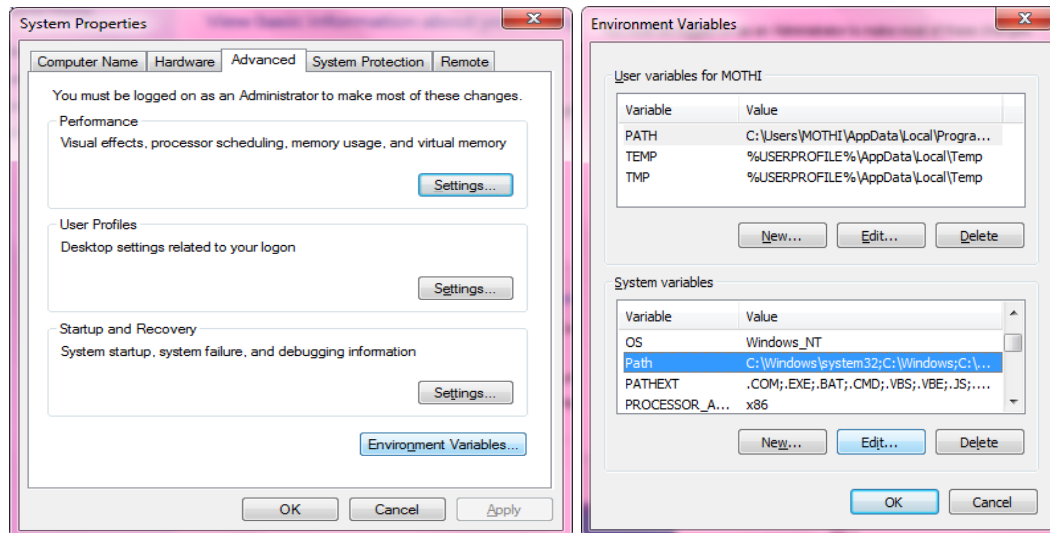
- Programs and other executable files can be in many directories, so operating systems provide a search path that lists the directories that the OS searches for executables.
- The path is stored in an environment variable, which is a named string maintained by the operating system. This variable contains information available to the command shell and other programs.
- Copy the Python installation location `C:\Python27`



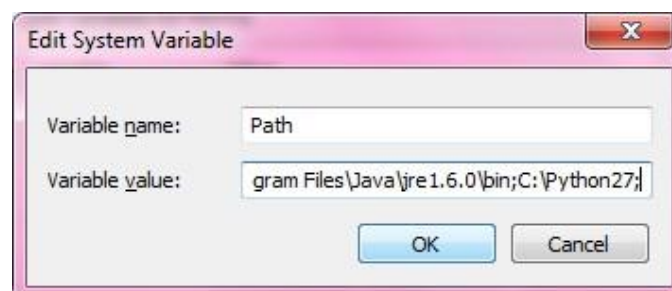
- Right-click the My Computer icon on your desktop and choose **Properties**. And then select **Advanced Systemproperties**.



- Goto **Environment Variables** and goto **System Variables** select **Path** and click on **Edit**.



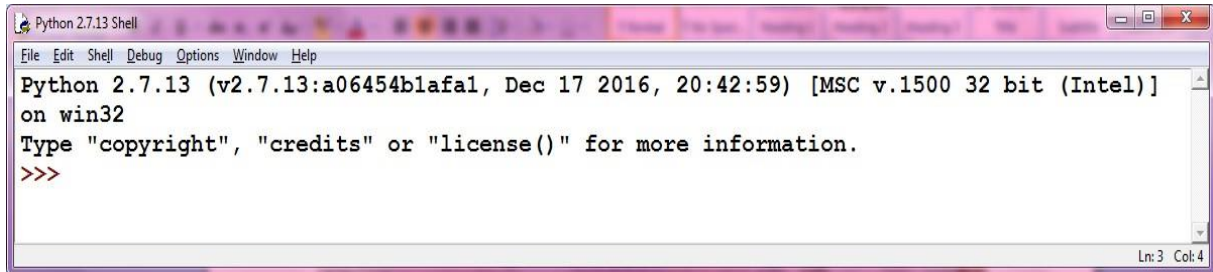
- Add semicolon (;) at end and copy the location **C:\Python27** and give semicolon (;) and click **OK**.



Running Python:

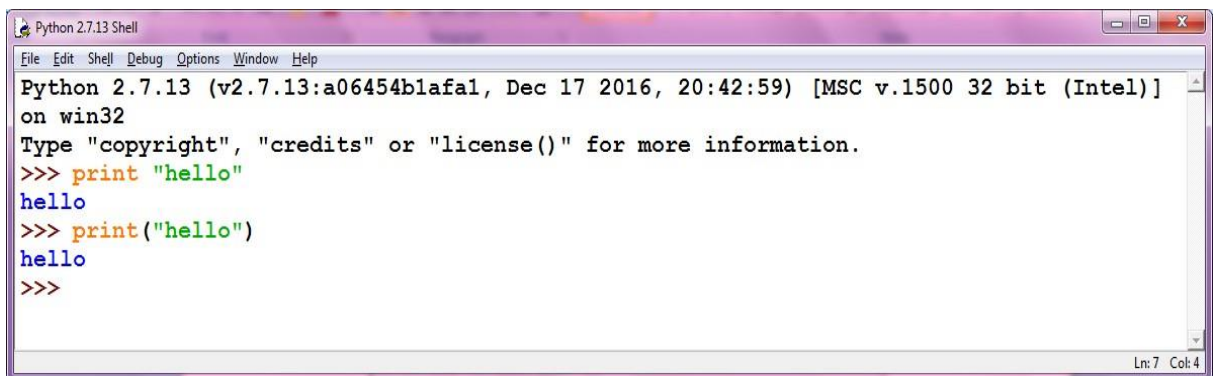
a. Running PythonInterpreter:

Python comes with an interactive interpreter. When you type python in your shell or command prompt, the python interpreter becomes active with a >>>prompt and waits for your commands.



```
Python 2.7.13 Shell
File Edit Shell Debug Options Window Help
Python 2.7.13 (v2.7.13:a06454b1afaf1, Dec 17 2016, 20:42:59) [MSC v.1500 32 bit (Intel)]
on win32
Type "copyright", "credits" or "license()" for more information.
>>>
```

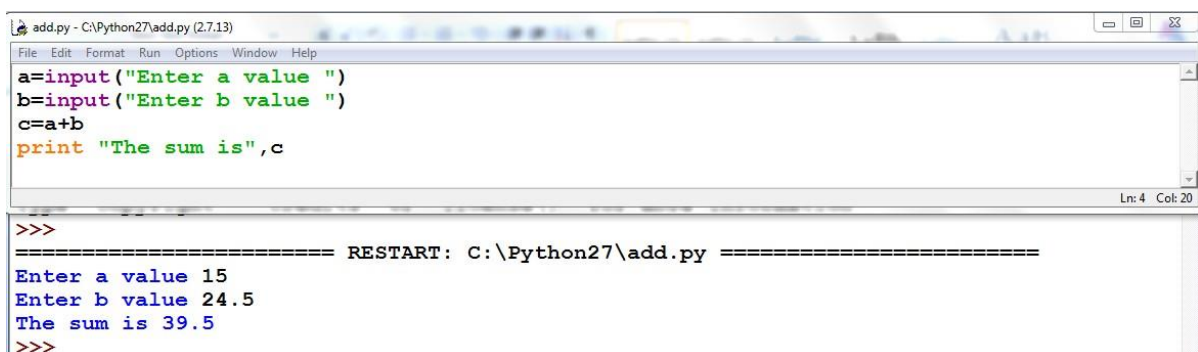
Now you can type any valid python expression at the prompt. Python reads the typed expression, evaluates it and prints the result.



```
Python 2.7.13 Shell
File Edit Shell Debug Options Window Help
Python 2.7.13 (v2.7.13:a06454b1afaf1, Dec 17 2016, 20:42:59) [MSC v.1500 32 bit (Intel)]
on win32
Type "copyright", "credits" or "license()" for more information.
>>> print "hello"
hello
>>> print("hello")
hello
>>>
```

b. Running Python Scripts inIDLE:

- GotoFile menu click on New File (CTRL+N) and write the code and save add.py
a=input("Enter a value")
b=input("Enter b value ")
c=a+b
print "The sum is",c
- And run the program by pressing F5 or Run→RunModule.




```
add.py - C:\Python27\add.py (2.7.13)
File Edit Format Run Options Window Help
a=input("Enter a value ")
b=input("Enter b value ")
c=a+b
print "The sum is",c

>>>
===== RESTART: C:\Python27\add.py =====
Enter a value 15
Enter b value 24.5
The sum is 39.5
>>>
```


c. Running python scripts in CommandPrompt:

- Before going to run we have to check the PATH in environment variables.
- Open your text editor, type the following text and save it as hello.py.
`print "hello"`
- And run this program by calling `python hello.py`. Make sure you change to the directory where you saved the file before doing it.



```
Administrator: Command Prompt
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\MOTHI>python hello.py
hello

C:\Users\MOTHI>
```

Variables:

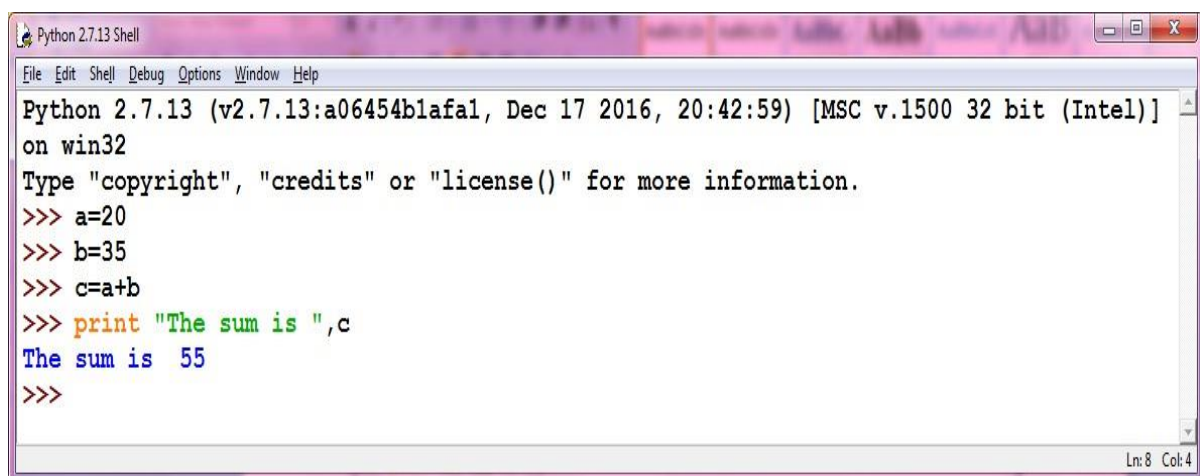
Variables are nothing but reserved memory locations to store values. This means that when you create a variable you reserve some space in memory.

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

Assigning Values to Variables

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

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



```
Python 2.7.13 Shell
File Edit Shell Debug Options Window Help
Python 2.7.13 (v2.7.13:a06454b1afaf, Dec 17 2016, 20:42:59) [MSC v.1500 32 bit (Intel)]
on win32
Type "copyright", "credits" or "license()" for more information.
>>> a=20
>>> b=35
>>> c=a+b
>>> print "The sum is ",c
The sum is  55
>>>
```

Multiple Assignments to variables:

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, 2.5, "mothi"

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.

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

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

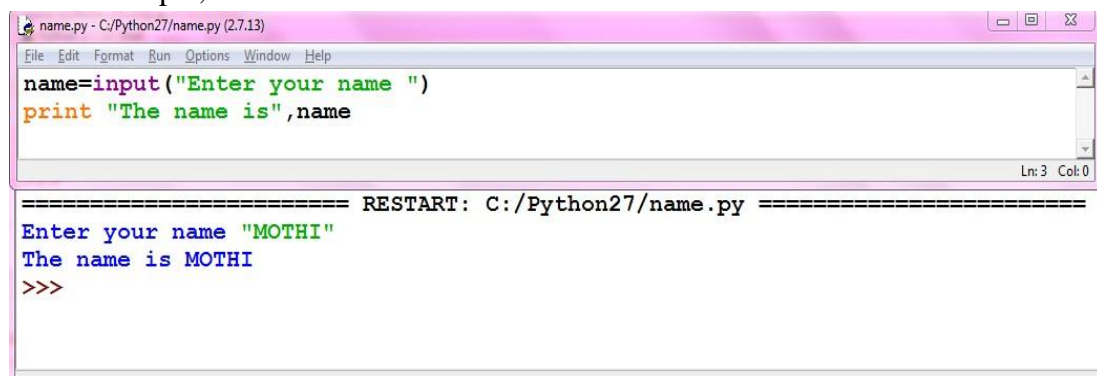
contain lowercase letters only.

INPUT Function:

To get input from the user you can use the input function. When the input function is called the program stops running the program, prompts the user to enter something at the keyboard by printing a string called the prompt to the screen, and then waits for the user to press the Enter key. The user types a string of characters and presses enter. Then the input function returns that string and Python continues running the program by executing the next statement after the input statement.

Python provides the function input(). input has an optional parameter, which is the prompt string.

For example,



The screenshot shows a Python IDE window titled 'name.py - C:/Python27/name.py (2.7.13)'. The code in the editor is:

```
name=input("Enter your name ")
print "The name is",name
```

Below the code editor, the output of the program is displayed:

```
===== RESTART: C:/Python27/name.py =====
Enter your name "MOTHI"
The name is MOTHI
>>>
```

OUTPUT function:

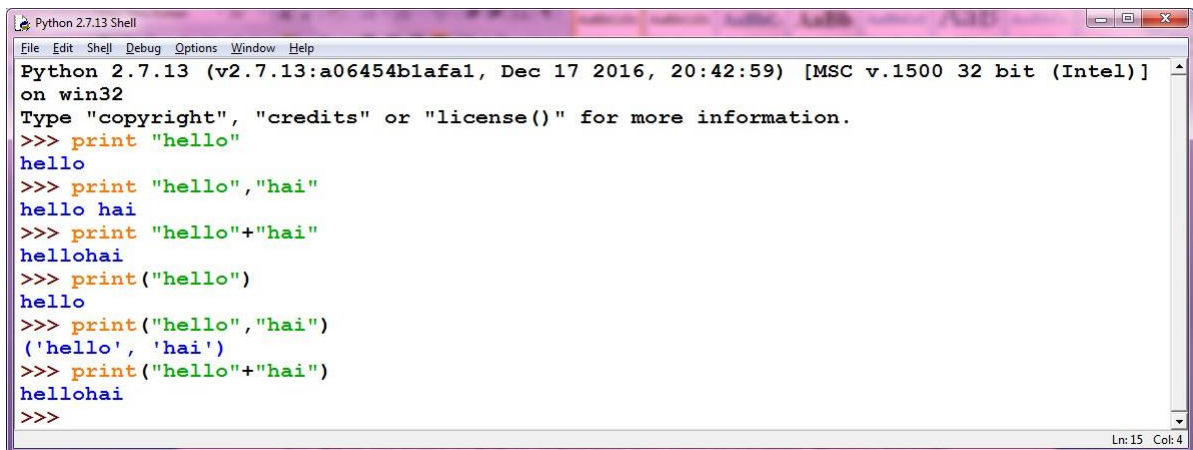
We use the `print()` function or `print` keyword to output data to the standard output device (screen). This function prints the object/string written in function.

The actual syntax of the `print()` function is

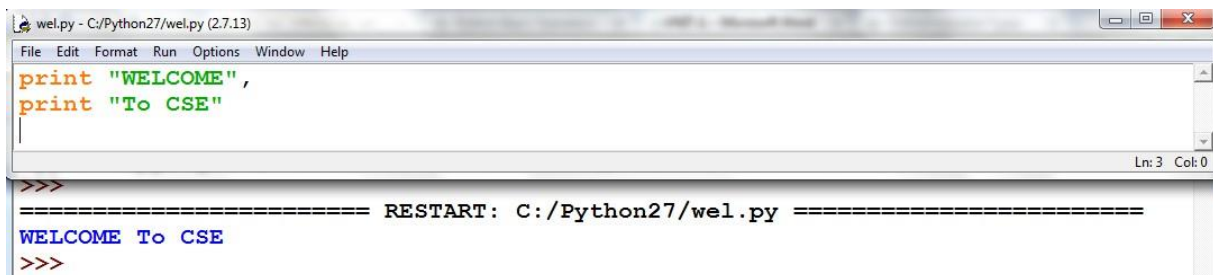
`print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)`

Here, `objects` is the value(s) to be printed.

The `sep` separator is used between the values. It defaults into a space character. After all values are printed, `end` is printed. It defaults into a new line (`\n`).



```
Python 2.7.13 Shell
File Edit Shell Debug Options Window Help
Python 2.7.13 (v2.7.13:a06454b1afaf1, Dec 17 2016, 20:42:59) [MSC v.1500 32 bit (Intel)]
on win32
Type "copyright", "credits" or "license()" for more information.
>>> print "hello"
hello
>>> print "hello", "hai"
hello hai
>>> print "hello"+"hai"
hellohai
>>> print("hello")
hello
>>> print("hello", "hai")
('hello', 'hai')
>>> print("hello"+"hai")
hellohai
>>>
```



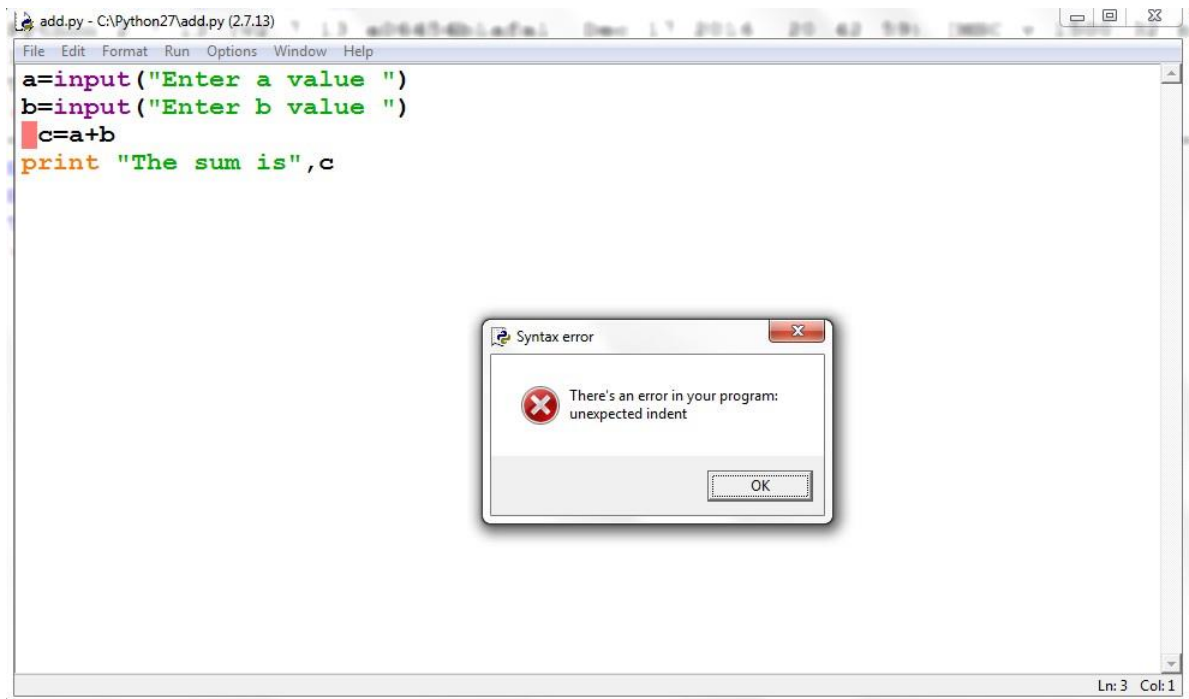
```
wel.py - C:/Python27/wel.py (2.7.13)
File Edit Format Run Options Window Help
print "WELCOME",
print "To CSE"
>>>

===== RESTART: C:/Python27/wel.py =====
WELCOME To CSE
>>>
```

Indentation

Code blocks are identified by indentation rather than using symbols like curly braces. Without extra symbols, programs are easier to read. Also, indentation clearly identifies which block of code a statement belongs to. Of course, code blocks can consist of single statements, too. When one is new to Python, indentation may come as a surprise. Humans generally prefer to avoid change, so perhaps after many years of coding with brace delimitation, the first impression of using pure indentation may not be completely positive. However, recall that two of Python's features are that it is simplistic in nature and easy to read.

Python does not support braces to indicate blocks of code for class and function definitions or flow control. Blocks of code are denoted by line indentation. All the continuous lines indented with same number of spaces would form a block. Python strictly follows indentation rules to indicate the blocks.



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
- Boolean
- List
- Tuple
- Set
- Dictionary

Python Numbers:

Number data types store numeric values. Number objects are created when you assign a value to them.

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)

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.

For example:

Program:

```
a = 3
b = 2.65
c = 98657412345L
d = 2+5j
print "int is",a
print "float is",b
print "long is",c
print "complex is",d
```

Output:

```
int is 3
float is 2.65
long is 98657412345
complex is (2+5j)
```

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:

Program:

```
str ="WELCOME"
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 + "CSE" # Prints concatenated string
```

Output:

```
WELCOME
W
LCO
LCOME
WELCOMEWELCOME
WELCOMECSE
```

Built-in String methods for Strings:

SNO	Method Name	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	<code>count(str, beg=0, end=len(string))</code>	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	<code>decode(encoding='UTF-8', errors='strict')</code>	Decodes the string using the codec registered for encoding. Encoding defaults to the default string encoding.
5	<code>encode(encoding='UTF-8', errors='strict')</code>	Returns encoded string version of string; on error, default is to raise a Value Error unless errors is given with 'ignore' or 'replace'.
6	<code>endswith(suffix, beg=0, end=len(string))</code>	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	<code>expandtabs(tabsize=8)</code>	Expands tabs in string to multiple spaces; defaults to 8 spaces per tab if tabsize not provided.
8	<code>find(str, beg=0, end=len(string))</code>	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	<code>index(str, beg=0, end=len(string))</code>	Same as find(), but raises an exception if str not found.
10	<code>isalnum()</code>	Returns true if string has at least 1 character and all characters are alphanumeric and false otherwise.
11	<code>isalpha()</code>	Returns true if string has at least 1 character and all characters are alphabetic and false otherwise.
12	<code>isdigit()</code>	Returns true if string contains only digits and false otherwise.
13	<code>islower()</code>	Returns true if string has at least 1 cased character and all cased characters are in lowercase and false otherwise.
14	<code>isnumeric()</code>	Returns true if a unicode string contains only numeric characters and false otherwise.
15	<code>isspace()</code>	Returns true if string contains only whitespace characters and false otherwise.
16	<code>istitle()</code>	Returns true if string is properly "titlecased" and false otherwise.
17	<code>isupper()</code>	Returns true if string has at least one cased character and all cased characters are in uppercase and false otherwise.
18	<code>join(seq)</code>	Merges (concatenates) the string representations of elements in sequence seq into a string, with separator string.
19	<code>len(string)</code>	Returns the length of the string.
20	<code>ljust(width[, fillchar])</code>	Returns a space-padded string with the original string left-justified to a total of width columns.
21	<code>lower()</code>	Converts all uppercase letters in string to lowercase.
22	<code>lstrip()</code>	Removes all leading whitespace in string.
23	<code>maketrans()</code>	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 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.

Example:

```

str1="welcome"
print "Capitalize function---",str1.capitalize()
print str1.center(15,"*")
print "length is",len(str1)
print "count function---",str1.count('e',0,len(str1))
print "endswith function---",str1.endswith('me',0,len(str1))
print "startswith function---",str1.startswith('me',0,len(str1))
print "find function---",str1.find('e',0,len(str1))
str2="welcome2017"
print "isalnum function---",str2.isalnum()
print "isalpha function---",str2.isalpha()
print "islower function---",str2.islower()
print "isupper function---",str2.isupper()

```



```

str3="          welcome"
print "lstrip function---",str3.lstrip()
str4="77777777cse777777";
print "lstrip function---",str4.lstrip('7')
print "rstrip function---",str4.rstrip('7')
print "strip function---",str4.strip('7')
str5="welcome to java"
print "replace function---",str5.replace("java","python")

```

Output:

```

Capitalize function--- Welcome
****welcome****
length is 7
count function--- 2
endswith function--- True
startswith function--- False
find function--- 1
isalnum function--- True
isalpha function--- False
islower function--- True
isupper function--- False
lstrip function---
welcome lstrip function-
-- cse777777
rstrip function--- 77777777cse
strip function--- cse
replace function--- welcome to python

```

Python Boolean:

Booleans are identified by True or False.

Example:

Program:

```

a = True
b = False
print a
print b

```

Output:

```

True
False

```

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. For example, it is not possible to perform "2"+4 since one operand is integer and the other is string type. To perform this we have convert string to integer i.e., `int("2") + 4 =6`.

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.
long(x [,base])	Converts x to a long integer.
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.

Types of Operators:

Python language supports the following types of operators.

- ArithmeticOperators +, -, *, /, %, **, //
- Comparison(Relational)Operators =, !=, <, >, <=, >=
- AssignmentOperators =, +=, -=, *=, /=, %=, **=, //=
- LogicalOperators **and, or, not**
- BitwiseOperators **&, |, ^, ~, <<, >>**
- MembershipOperators **in, notin**
- IdentityOperators **is, is not**

Arithmetic Operators:

Some basic arithmetic operators are +, -, *, /, %, **, and //. You can apply these operators on numbers as well as variables to perform corresponding operations.

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 $9.0//2.0 = 4.0$
-------------------	--	---------------------------------

Program:

```

a = 21
b = 10
print "Addition is", a + b
print "Subtraction is ", a - b
print "Multiplication is ", a * b
print "Division is ", a / b
print "Modulus is ", a % b
a = 2
b = 3
print "Power value is ", a ** b
a = 10
b = 4
print "Floor Division is ", a // b

```

Output:

```

Addition is 31
Subtraction is 11
Multiplication is 210
Division is 2
Modulus is 1
Power value is 8
Floor Division is 2

```

Comparison (Relational) Operators

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

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

```

a=20
b=30
if a < b:
    print "b is big"
elif a > b:
    print "a is big"
else:
    print "Both are equal"

```

Output:

b is big

Assignment Operators

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:

```

a=82
b=27
a += b
print a
a=25
b=12
a -= b
print a
a=24
b=4
a *= b

```

```

print a
a=4
b=6
a **= b
print a

```

Output:

```

109
13
96
4096

```

Logical Operators

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.

Example:

```

a=20
b=10
c=30
if a >= b and a >= c:
    print "a isbig"
elif b >= a and b >= c:
    print "b isbig"
else:
    print "c is big"

```

Output:

```

c is big

```

Bitwise Operators

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 rightoperand.	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)

Membership Operators

Python's membership operators test for membership in a sequence, such as strings, lists, or tuples.

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:

```
a = 3
list = [1, 2, 3, 4, 5 ];
if ( a in list ):
    print "available"
else:
    print " not available"
```

Output:

available

Identity Operators

Identity operators compare the memory locations of two objects.

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:

```
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"
```

Python Operators Precedence

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

Operator	Description
()	Parenthesis
**	Exponentiation (raise to the power)
~ x, +x, -x	Complement, unary plus and minus
* / % //	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

Expression:

An expression is a combination of variables constants and operators written according to the syntax of Python language. In Python every expression evaluates to a value i.e., every expression results in some value of a certain type that can be assigned to a variable. Some examples of Python expressions are shown in the table given below.

Algebraic Expression	Python Expression
$a \times b - c$	<code>a * b - c</code>
$(m + n) (x + y)$	<code>(m + n) * (x + y)</code>
(ab / c)	<code>a * b / c</code>
$3x^2 + 2x + 1$	<code>3*x*x+2*x+1</code>
$(x / y) + c$	<code>x / y + c</code>

Evaluation of Expressions

Expressions are evaluated using an assignment statement of the form

Variable = expression

Variable is any valid C variable name. When the statement is encountered, the expression is evaluated first and then replaces the previous value of the variable on the left hand side. All variables used in the expression must be assigned values before evaluation is attempted.

Example:

```
a=10
b=22
c=34
x=a*b+c
y=a-b*c
z=a+b+c*c-a
print "x=",x
print "y=",y
print "z=",z
```

Output:

```
x= 254
y=-738
z= 1178
```

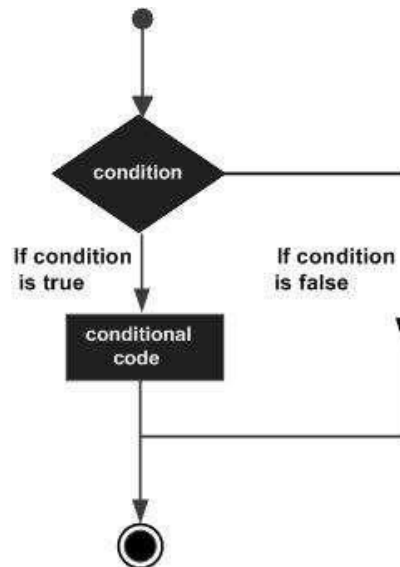
Unit-II

Decision Making:

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

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

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

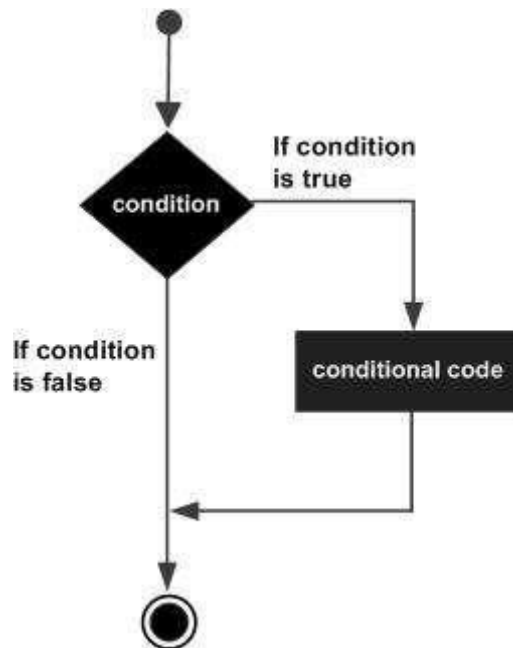


Python programming language assumes any non-zero and non-null values as True, and if it is either zero or null, then it is assumed as Falsevalue.

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).

The 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 condition:
    statements
  
```

First, the condition is tested. If the condition is True, then the statements given after colon (:) are executed. We can write one or more statements after colon (:).

Example:

```

a=10
b=15
if a < b:
    print "B is big"
    print "B value is",b
  
```

Output:

```

B is big
B value is 15
  
```

The if ... else statement

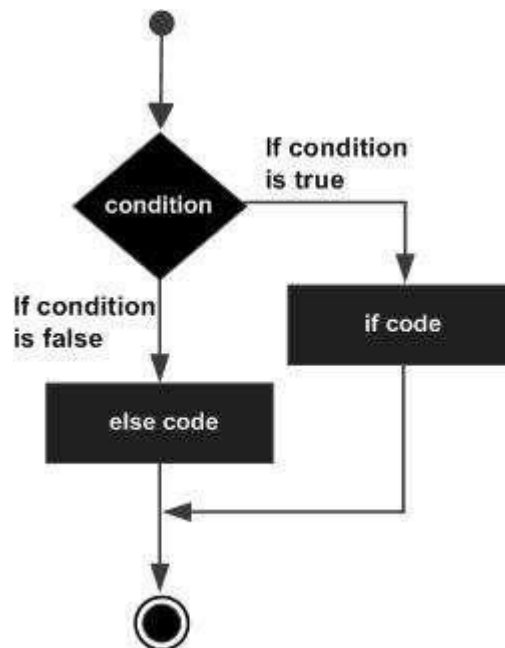
An **else** statement can be combined with an **if** statement. An **else** statement contains the block of code that executes if the conditional expression in the if statement resolves to 0 or a FALSE value.

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

Syntax:

```

if condition:
    statement(s)
else:
    statement(s)
  
```



Example:

```

a=48
b=34
if a < b:
    print "B is big"
    print "B value is",b
else:
    print "A is big"
    print "A value is", a
print "END"
  
```

Output:

```

A is big
A value is 48
END
  
```

Q) Write a program for checking whether the given number is even or not.

Program:

```

a=input("Enter a value: ")
if a%2==0:
    print "a is EVEN number"
else:
    print "a is NOT EVEN Number"
  
```

Output-1:

```

Enter avalue: 56
a isEVENNumber
  
```

Output-2:

```

Enter a value:27
a is NOT EVENNumber
  
```

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 condition1:
    statement(s)
elif condition2:
    statement(s)
else:
    statement(s)
```

Example:

```
a=20
b=10
c=30
if a >= b and a >= c:
    print "a is big"
elif b >= a and b >= c:
    print "b is big"
else:
    print "c is big"
```

Output:

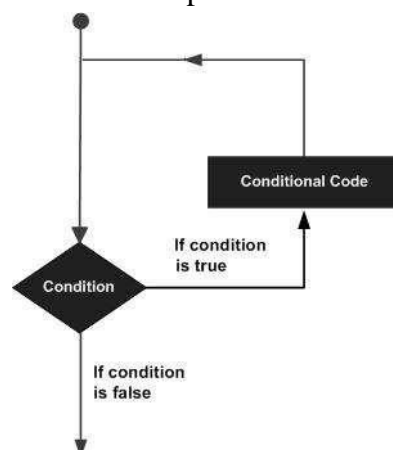
c is big

Decision Loops

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

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

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



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

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 loop.

The *while* Loop

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

Syntax

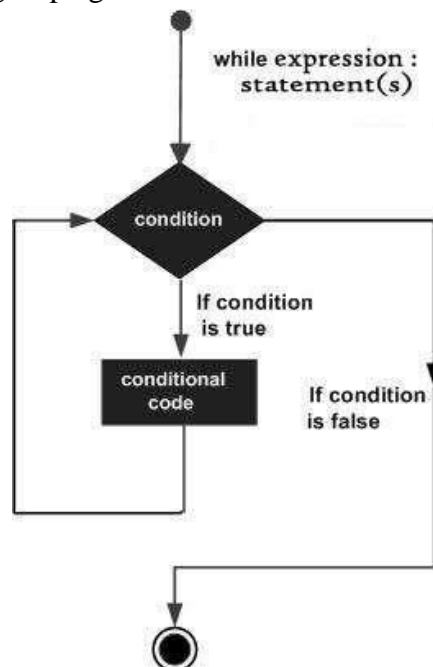
The syntax of a **while** loop in Python programming language is:

```
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.



Example-1:

```
i=1
while i < 4:
    print
    ii+=1
    print "END"
```

Example-2:

```
i=1
while i < 4:
    print
    ii+=1
    print "END"
```

Output-1:

```
1
END
2
END
3
END
```

Output-2:

```
1
2
3
END
```

Q) Write a program to display factorial of a given number.

Program:

```
n=input("Enter the number: ")
f=1
while n>0:
    f=f*n
    n=n-1
print "Factorial is",f
```

Output:

```
Enter the number: 5
Factorial is 120
```

The for loop:

The *for* loop is useful to iterate over the elements of a sequence. It means, the *for* loop can be used to execute a group of statements repeatedly depending upon the number of elements in the sequence. The *for* loop can work with sequence like string, list, tuple, range etc.

The syntax of the *for* loop is given below:

```
for var in sequence:
    statement (s)
```

The first element of the sequence is assigned to the variable written after „for“ and then the statements are executed. Next, the second element of the sequence is assigned to the variable and then the statements are executed second time. In this way, for each element of the sequence, the statements are executed once. So, the *for* loop is executed as many times as there are number of elements in thesequence.

Example-1:

```
for irange(1,5):  
    print i  
    print "END"
```

Output-1:

```
1  
END  
2  
END  
3  
END
```

Example-2:

```
for irange(1,5):  
    print i  
    print "END"
```

Output-2:

```
1  
2  
3  
END
```

Example-3:

```
name= "python"  
    for letter  
        inname:
```

Output-3:

```
p  
y  
t  
h  
o  
n
```

Example-4:

```
for x in range(10,0,-1):  
    print x,
```

Output-4:

```
10 9 8 7 6 5 4 3 2 1
```

Q) Write a program to display the factorial of given number.

Program:

```
n=input("Enter the number: ")  
f=1  
for i in range(1,n+1):  
    f=f*i  
print "Factorial is",f
```

Output:

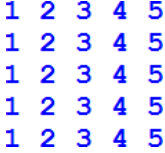
```
Enter the number: 5  
Factorial is 120
```

Nested Loop:

It is possible to write one loop inside another loop. For example, we can write a for loop inside a while loop or a for loop inside another for loop. Such loops are called “nested loops”.


Example-1:

```
for i in range(1,6):  
    for j in range(1,6):  
        print j,  
        print ""
```




Example-2:

```
for i in range(1,6):  
    for j in range(1,6):  
        print "*",  
        print ""
```




Example-3:

```
for i in range(1,6):  
    for j in range(1,6):  
        if i==1 or j==1 or i==5 or j==5:  
            print "*",  
        else:  
            print " ",  
        print ""
```



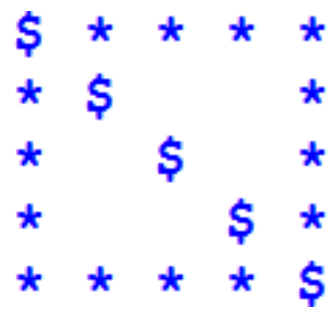
Example-4:

```
for i in range(1,6):  
    for j in range(1,6):  
        if i==j:  
            print "*",  
        elif i==1 or j==1 or i==5 or j==5:  
            print "*",  
        else:  
            print " ",  
        print ""
```



Example-5:

```
for i in range(1,6):  
    for j in range(1,6):  
        if i==j:  
            print "$",  
        elif i==1 or j==1 or i==5 or j==5:  
            print "*",  
        else:  
            print " ",  
        print ""
```



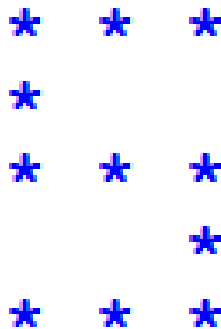
Example-6:

```
for i in range(1,6):
    for j in range(1,4):
        if i==1 or j==1 or i==5:
            print "*",
        else:
            print " ",
    print ""
```



Example-7:

```
for i in range(1,6):
    for j in range(1,4):
        if i==2 and j==1:
            print "*",
        elif i==4 and j==3:
            print "*",
        elif i==1 or i==3 or i==5:
            print "*",
        else:
            print " ",
    print ""
```



Example-8:

```
for i in range(1,6):
    for j in range(1,4):
        if i==1 or j==1 or i==3 or i==5:
            print "*",
        else:
            print " ",
    print ""
```



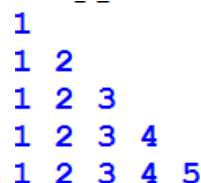
Example-9:

```
for i in range(1,6):
    for c in range(i,6):
        print "",
    for j in range(1,i+1):
        print "*",
    print ""
```



Example-10:

```
for i in range(1,6):
    for j in range(1,i+1):
        print j,
    print ""
```



Example-11:

```
a=1
for i in range(1,5):
    for j in range(1,i+1):
        print a,
        a=a+1
    print ""
```

```
1
2 3
4 5 6
7 8 9 10
```

1) Write a program for print given number is prime number or not using for loop.

Program:

```
n=input("Enter the n value")
count=0
for i in range(2,n):
    if n%i==0:
        count=count+1
        break
if count==0:
    print "Prime Number"
else:
    print "Not Prime Number"
```

Output:

```
Enter n value: 17
Prime Number
```

2) Write a program print Fibonacci series and sum the even numbers. Fibonacci series is 1,2,3,5,8,13,21,34,55

```
n=input("Enter n value ")
f0=1
f1=2
sum=f1
print f0,f1,
for i in range(1,n-1):
    f2=f0+f1
    print f2,
    f0=f1
    f1=f2
    if f2%2==0:
        sum+=f2
print "\nThe sum of even Fibonacci numbers is", sum
```

Output:

```
Enter n value 10
1 2 3 5 8 13 21 34 55 89
The sum of even fibonacci numbers is 44
```

3) Write a program to print n prime numbers and display the sum of primenumbers.

Program:

```
n=input("Enter the range: ")
sum=0
for num in range(1,n+1):
    for i in range(2,num):
        if (num % i) == 0:
            break
    else:
        print num,
        sum += num
print "\nSum of prime numbers is",sum
```

Output:

```
Enter the range: 21
1 2 3 5 7 11 13 17 19
Sum of prime numbers is 78
```

4) Using a for loop, write a program that prints out the decimal equivalents of 1/2, 1/3, 1/4, ..., 1/10

Program:

```
for i in range(1,11):
    print "Decimal Equivalent of 1/",i,"is",1/float(i)
```

Output:

```
Decimal Equivalent of 1/ 1 is 1.0
Decimal Equivalent of 1/ 2 is 0.5
Decimal Equivalent of 1/ 3 is 0.333333333333
Decimal Equivalent of 1/ 4 is 0.25
Decimal Equivalent of 1/ 5 is 0.2
Decimal Equivalent of 1/ 6 is 0.166666666667
Decimal Equivalent of 1/ 7 is 0.142857142857
Decimal Equivalent of 1/ 8 is 0.125
Decimal Equivalent of 1/ 9 is 0.111111111111
Decimal Equivalent of 1/ 10 is 0.1
```

5) Write a program that takes input from the user until the user enters -1. After display the sum of numbers.

Program:

```
sum=0
while True:
    n=input("Enter the number: ")
    if n== -1:
        break
    else:
        sum+=n
print "The sum is",sum
```

Output:

```
Enter the number: 1
Enter the number: 5
Enter the number: 6
Enter the number: 7
Enter the number: 8
Enter the number: 1
Enter the number: 5
Enter the number: -1
The sum is 33
```

6) Write a program to display the following sequence.

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Program:

```
ch='A'
for j in range(1,27):
    print ch,
    ch=chr(ord(ch)+1)
```

7) Write a program to display the following sequence.

```
A
A B
A B C
A B C D
A B C D E
```

Program:

```
for i in range(1,6):
    ch='A'
    for j in range(1,i+1):
        print ch,
        ch=chr(ord(ch)+1)
    print ""
```

8) Write a program to display the following sequence.

A
B C
D E F
G H I J
K L M N O

Program:

```
ch='A'
for i in range(1,6):
    for j in range(1,i+1):
        print ch,
        ch=chr(ord(ch)+1)
    print ""
```

9) Write a program that takes input string user and display that string if string contains at least one Uppercase character, one Lowercase character and onedigit.

Program:

```
pwd=input("Enter the password:")
u=False
l=False
d=False
for i in range(0,len(pwd)):
    if pwd[i].isupper():
        u=True
    elifpwd[i].islower():
        l=True
    elifpwd[i].isdigit():
        d=True
if u==True and l==True and d==True:
    print pwd.center(20,"*")
else:
    print "Invalid Password"
```

Output-1:

```
Enter the password:"Mothi556"
*****Mothi556*****
```

Output-2:

```
Enter the password:"mothilal"
Invalid Password
```

10) Write a program to print sum of digits.

Program:

```
n=input("Enter the number: ")
sum=0
while n>0:
    r=n%10
    sum+=r
    n=n/10
print "sum is",sum
```

Output:

```
Enter the number: 123456789
sum is 45
```

11) Write a program to print given number is Armstrong or not.

Program:

```
n=input("Enter the number: ")
sum=0
t=n
while n>0:
    r=n%10
    sum+=r*r*r
    n=n/10
if sum==t:
    print "ARMSTRONG"
else:
    print "NOT ARMSTRONG"
```

Output:

```
Enter the number: 153
ARMSTRONG
```

12) Write a program to take input string from the user and print that string after removing vowels.

Program:

```
st=input("Enter the string:")
st2=""
for i in st:
    if i not in "aeiouAEIOU":
        st2=st2+i
print st2
```

Output:

```
Enter the string:"Welcome to you"
Wlcm t y
```


Arrays:

An array is an object that stores a group of elements of same datatype.

- Arrays can store only one type of data. It means, we can store only integer type elements or only float type elements into an array. But we cannot store one integer, one float and one character type element into the same array.
- Arrays can increase or decrease their size dynamically. It means, we need not declare the size of the array. When the elements are added, it will increase its size and when the elements are removed, it will automatically decrease its size in memory.

Advantages:

- Arrays are similar to lists. The main difference is that arrays can store only one type of elements; whereas, lists can store different types of elements. When dealing with a huge number of elements, arrays use less memory than lists and they offer faster execution than lists.
- The size of the array is not fixed in python. Hence, we need not specify how many elements we are going to store into an array in the beginning.
- Arrays can grow or shrink in memory dynamically (during runtime).
- Arrays are useful to handle a collection of elements like a group of numbers or characters.
- Methods that are useful to process the elements of any array are available in „array“ module.

Creating an array:

Syntax:

```
arrayname = array(type code, [elements])
```

The type code „i“ represents integer type array where we can store integer numbers. If the type code is „f“ then it represents float type array where we can store numbers with decimal point.

Type code	Description	Minimum size in bytes
„b“	Signed integer	1
„B“	Unsigned integer	1
„i“	Signed integer	2
„I“	Unsigned integer	2
„l“	Signed integer	4
„L“	Unsigned integer	4
„f“	Floating point	4
„d“	Double precision floating point	8
„u“	Unicode character	2

Example:

The type code character should be written in single quotes. After that the elements should be written in inside the square braces [] as

```
a = array ( „i“, [4,8,-7,1,2,5,9] )
```

Importing the Array Module:

There are two ways to import the array module into our program.

The first way is to import the entire array module using import statement as,

import array

when we import the array module, we are able to get the „array“ class of that module that helps us to create an array.

a = array.array('i', [4,8,-7,1,2,5,9])

Here the first „array“ represents the module name and the next „array“ represents the class name for which the object is created. We should understand that we are creating our array as an object of array class.

The next way of importing the array module is to write:

from array import *

Observe the „*“ symbol that represents „all“. The meaning of this statement is this: import all (classes, objects, variables, etc) from the array module into our program. That means significantly importing the „array“ class of „array“ module. So, there is no need to mention the module name before our array name while creating it. We can create arrays as:

a = array('i', [4,8,-7,1,2,5,9])

Example:

```
from array import *
arr = array('i', [4,8,-7,1,2,5,9])
for i in arr:
    print i,
```

Output:

4 8 -7 1 2 5 9

Indexing and slicing of arrays:

An *index* represents the position number of an element in an array. For example, when we creating following integer type array:

a = array('i', [10,20,30,40,50])

Python interpreter allocates 5 blocks of memory, each of 2 bytes size and stores the elements 10, 20, 30, 40 and 50 in these blocks.

10	20	30	40	50
a[0]	a[1]	a[2]	a[3]	a[4]

Example:

```
from array import *
a=array('i', [10,20,30,40,50,60,70])
print "length is",len(a)
print " 1st position character", a[1]
print "Characters from 2 to 4", a[2:5]
print "Characters from 2 to end", a[2:]
print "Characters from start to 4",a[:5]
print "Characters from start to end",a[:]
```

```

a[3]=45
a[4]=55
print "Characters from start to end after modifications ",a[:]

```

Output:

```

length is 7
1st position character 20
Characters from 2 to 4 array('i', [30, 40, 50])
Characters from 2 to end array('i', [30, 40, 50, 60, 70])
Characters from start to 4 array('i', [10, 20, 30, 40, 50])
Characters from start to end array('i', [10, 20, 30, 40, 50, 60, 70])
Characters from start to end after modifications array('i', [10, 20, 30, 45, 55, 60, 70])

```

Array Methods:

Method	Description
a.append(x)	Adds an element x at the end of the existing array a.
a.count(x)	Returns the number of occurrences of x in the array a.
a.extend(x)	Appends x at the end of the array a. „x“ can be another array or iterable object.
a.fromfile(f,n)	Reads n items from from the file object f and appends at the end of the array a.
a.fromlist(l)	Appends items from the l to the end of the array. l can be any list or iterable object.
a.fromstring(s)	Appends items from string s to end of the array a.
a.index(x)	Returns the position number of the first occurrence of x in the array. Raises „ValueError“ if not found.
a.pop(x)	Removes the item x from the array a and returns it.
a.pop()	Removes last item from the array a
a.remove(x)	Removes the first occurrence of x in the array. Raises „ValueError“ if not found.
a.reverse()	Reverses the order of elements in the array a.
a.tofile(f)	Writes all elements to the file f.
a.tolist()	Converts array „a“ into a list.
a.tostring()	Converts the array into a string.

Unit-III

FUNCTIONS:

A function is a block of organized, reusable code that is used to perform a single, related action.

- Once a function is written, it can be reused as and when required. So, functions are also called reusable code.
- Functions provide modularity for programming. A module represents a part of the program. Usually, a programmer divides the main task into smaller sub tasks called modules.
- Code maintenance will become easy because of functions. When a new feature has to be added to the existing software, a new function can be written and integrated into the software.
- When there is an error in the software, the corresponding function can be modified without disturbing the other functions in the software.
- The use of functions in a program will reduce the length of the program.

As you already know, Python gives you many built-in functions like `sqrt()`, etc. but you can also create your own functions. These functions are called *user-defined functions*.

Difference between a function and a method:

A function can be written individually in a python program. A function is called using its name. When a function is written inside a class, it becomes a „method“. A method is called using object name or class name. A method is called using one of the following ways:

Objectname.methodname()

Classname.methodname()

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:

```
def add(a,b):  
    """This function sum thenumbers"""  
    c=a+b  
    print c  
    return
```

Here, „*def*” represents starting of function. „*add*” is function name. After this name, parentheses () are compulsory as they denote that it is a function and not a variable or something else. In the parentheses we wrote two variables „*a*” and „*b*” these variables are called „parameters”. A parameter is a variable that receives data from outside a function. So, this function receives two values from outside and those are stored in the variables „*a*” and „*b*”. After parentheses, we put colon (:) that represents the beginning of the function body. The function body contains a group of statements called „suite”.

Calling Function:

A function cannot run by its own. It runs only when we call it. So, the next step is to call function using its name. While calling the function, we should pass the necessary values to the function in the parentheses as:

```
add(5,12)
```

Here, we are calling „*add*” function and passing two values 5 and 12 to that function. When this statement is executed, the python interpreter jumps to the function definition and copies the values 5 and 12 into the parameters „*a*” and „*b*” respectively.

Example:

```
def add(a,b):  
    """This function sum the numbers"""  
    c=a+b  
    print c  
add(5,12) # 17
```

Returning Results from a function:

We can return the result or output from the function using a „*return*” statement in the function body. When a function does not return any result, we need not write the return statement in the body of the function.

Q) Write a program to find the sum of two numbers and return the result from the function.

```
def add(a,b):  
    """This function sum the numbers"""  
    c=a+b  
    return c  
print add(5,12) #17  
print add(1.5,6)#6.5
```

Returning multiple values from a function:

A function can return a single value in the programming languages like C, C++ and JAVA. But, in python, a function can return multiple values. When a function calculates multiple results and wants to return the results, we can use return statement as:

return a, b, c

Here, three values which are in „a“, „b“ and „c“ are returned. These values are returned by the function as a tuple. To grab these values, we can use three variables at the time of calling the function as:

x, y, z = functionName()

Here, „x“, „y“ and „z“ are receiving the three values returned by the function.

Example:

```
def calc(a,b):
    c=a+b
    d=a-b
    e=a*b
    return
c,d,e,x,y,z=calc(5,8)
print
"Addition=",x
print "Subtraction=",y
```

Functions are First Class Objects:

In Python, functions are considered as first class objects. It means we can use functions as perfect objects. In fact when we create a function, the Python interpreter internally creates an object. Since functions are objects, we can pass a function to another function just like we pass an object (or value) to a function. The following possibilities are:

- It is possible to assign a function to a variable.
- It is possible to define one function inside another function.
- It is possible to pass a function as parameter to another function.
- It is possible that a function can return another function.

To understand these points, we will take a few simple programs.

Q) A python program to see how to assign a function to a variable. def

```
display(st):
    return "hai"+st
x=display("cse")
printx
```

Output:haicse

Q) A python program to know how to define a function inside another function. def

```
display(st):
    def message():
        return "how r u?"
    res=message()+st
    return res
x=display("cse")
printx
```

Output: how r u?cse

Q) A python program to know how to pass a function as parameter to another function. def

```
display(f):  
    return "hai"+f def  
message():  
    return "how r u?"  
fun=display(message())  
printfun
```

Output: haihow ru?

Q) A python program to know how a function can return anotherfunction.

```
defdisplay():  
    def message():  
        return "how r u?"  
    return message fun=display()  
printfun()
```

Output: how ru?

Pass by Value:

Pass by value represents that a copy of the variable value is passed to the function and any modifications to that value will not reflect outside the function. In python, the values are sent to functions by means of object references. We know everything is considered as an object in python. All numbers, strings, tuples, lists and dictionaries are objects.

If we store a value into a variable as:

x=10

In python, everything is an object. An object can be imagined as a memory block where we can store some value. In this case, an object with the value „10“ is created in memoryforwhichaname„x“isattached.So,10 istheobject and„x“isthenamortaggiven to that object. Also, objects are created on heap memory which is a very huge memory that depends on the RAM of our computer system.

Example: A Python program to pass an integer to a function and modifyit.

```
defmodify(x):  
    x=15  
    print "inside",x,id(x)  
x=10  
modify(x)  
print "outside",x,id(x)
```

Output:

inside 15 6356456
outside 10 6356516

From the output, we can understand that the value of „x“ in the function is 15 and that is not available outside the function. When we call the modify() function and pass „x“ as:

modify(x)

We should remember that we are passing the object references to the modify() function. The object is 10 and its references name is „x“. This is being passed to the modify() function. Inside the function, we are using:

x=15

This means another object 15 is created in memory and that object is referenced by the name „x“. The reason why another object is created in the memory is that the integer objects are immutable (not modifiable). So in the function, when we display „x“ value, it will display 15. Once we come outside the function and display „x“ value, it will display numbers of „x“ inside and outside the function, and we see different numbers since they are different objects.

In python, integers, floats, strings and tuples are immutable. That means their data cannot be modified. When we try to change their value, a new object is created with the modified value.

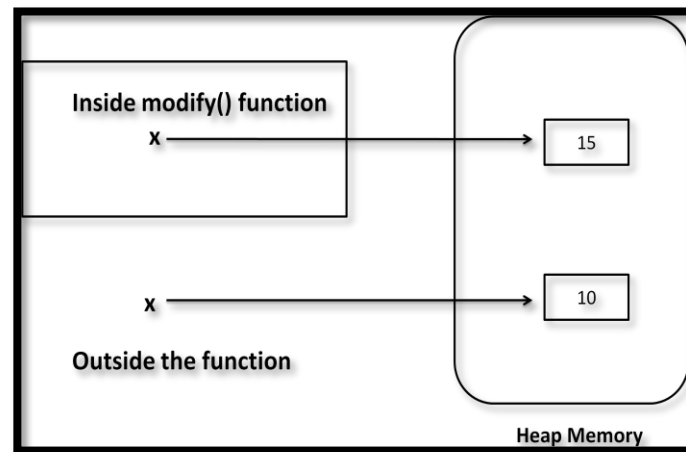


Fig. Passing Integer to a Function

Pass by Reference:

Pass by reference represents sending the reference or memory address of the variable to the function. The variable value is modified by the function through memory address and hence the modified value will reflect outside the function also.

In python, lists and dictionaries are mutable. That means, when we change their data, the same object gets modified and new object is not created. In the below program, we are passing a list of numbers to modify () function. When we append a new element to the list, the same list is modified and hence the modified list is available outside the function also.

Example: A Python program to pass a list to a function and modify it.

```
def modify(a):
    a.append(5)
    print "inside",a,id(a)
a=[1,2,3,4]
modify(a)
print "outside",a,id(a)
```

Output:

```
inside [1, 2, 3, 4, 5] 45355616
outside [1, 2, 3, 4, 5] 45355616
```

In the above program the list „a“ is the name or tag that represents the list object. Before calling the modify() function, the list contains 4 elements as: **a=[1,2,3,4]**

Inside the function, we are appending a new element „5“ to the list. Since, lists are mutable, adding a new element to the same object is possible. Hence, append() method

modifies the same object.

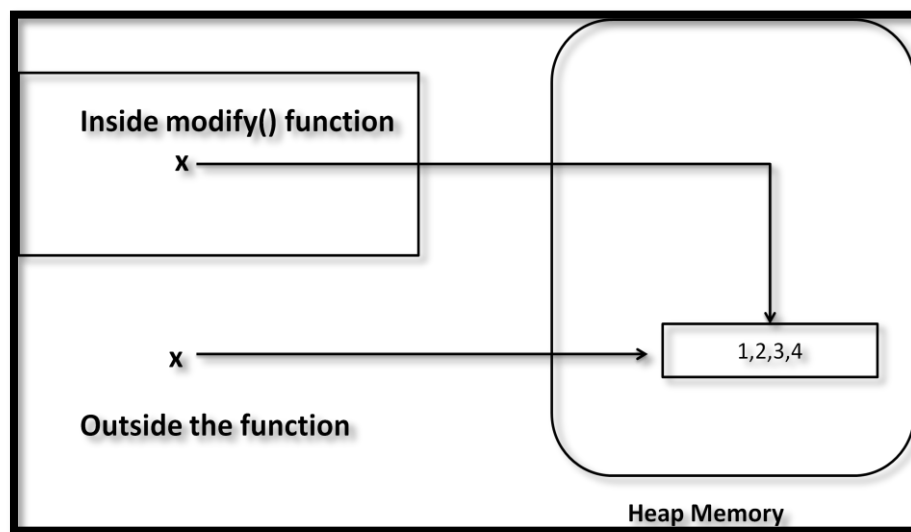


Fig. Passing a list to the function

Formal and Actual Arguments:

When a function is defined, it may have some parameters. These parameters are useful to receive values from outside of the function. They are called „formal arguments“. When we call the function, we should pass data or values to the function. These values are called „actual arguments“. In the following code, „a“ and „b“ are formal arguments and „x“ and „y“ are actual arguments.

Example:

```
def add(a,b): # a, b are formal arguments
    c=a+b
    print c
x,y=10,15
add(x,y) # x, y are actual arguments
```

The actual arguments used in a function call are of 4 types:

- a) Positional arguments
- b) Keyword arguments
- c) Default arguments
- d) Variable length arguments

a) Positional Arguments:

These are the arguments passed to a function in correct positional order. Here, the number of arguments and their position in the function definition should match exactly with the number and position of argument in the function call.

```
def attach(s1,s2):
    s3=s1+s2
    print s3
```

```
attach("New","Delhi") #Positional arguments
```

This function expects two strings that too in that order only. Let's assume that this function attaches the two strings as `s1+s2`. So, while calling this function, we are supposed to pass only two strings as: `attach("New","Delhi")`

The preceding statements displays the following output NewDelhi

Suppose, we passed "Delhi" first and then "New", then the result will be: "DelhiNew". Also, if we try to pass more than or less than 2 strings, there will be an error.

b) KeywordArguments:

Keyword arguments are arguments that identify the parameters by their names. For example, the definition of a function that displays grocery item and its price can be written as:

def grocery(item, price):

At the time of calling this function, we have to pass two values and we can mention which value is for what. For example,

grocery(item='sugar', price=50.75)

Here, we are mentioning a keyword, „item“ and its value and then another keyword, „price“ and its value. Please observe these keywords are nothing but the parameter names which receive these values. We can change the order of the arguments as:

grocery(price=88.00, item='oil')

In this way, even though we change the order of the arguments, there will not be any problem as the parameter names will guide where to store that value.

```
def grocery(item,price):  
    print "item=",item  
    print "price=",price  
grocery(item="sugar",price=50.75) # keyword arguments  
grocery(price=88.00,item="oil") # keyword arguments
```

Output:

```
item= sugar  
price= 50.75  
item= oil price=  
88.0
```

c) DefaultArguments:

We can mention some default value for the function parameters in the definition.

Let's take the definition of grocery() function as:

def grocery(item, price=40.00)

Here, the first argument is „item“ whose default value is not mentioned. But the second argument is „price“ and its default value is mentioned to be 40.00. at the time of calling this function, if we do not pass „price“ value, then the default value of 40.00 is taken. If we mention the „price“ value, then that mentioned value is utilized. So, a default argument is an argument that assumes a default value if a value is not provided in the function call for that argument.

Example:

```
def grocery(item,price=40.00):  
    print "item=",item  
    print "price=",price  
grocery(item="sugar",price=50.75)  
grocery(item="oil")
```

Output:

```
item= sugar
price= 50.75
item= oil
price= 40.0
```

d) Variable LengthArguments:

Sometimes, the programmer does not know how many values a function may receive. In that case, the programmer cannot decide how many arguments to be given in the function definition. for example, if the programmer is writing a function to add two numbers, he/she can write:

add(a,b)

But, the user who is using this function may want to use this function to find sum of three numbers. In that case, there is a chance that the user may provide 3 arguments to this function as:

add(10,15,20)

Then the add() function will fail and error will be displayed. If the programmer want to develop a function that can accept „n“ arguments, that is also possible in python. For this purpose, a variable length argument is used in the function definition. a variable length argument is an argument that can accept any number of values. The variable length argument is written with a „*“ symbol before it in the function definition as:

def add(farg, *args):

here, „farg“ is the formal; argument and „*args“ represents variable length argument. We can pass 1 or more values to this „*args“ and it will store them all in a tuple.

Example:

```
def add(farg, *args):
    sum=0
    for i in args:
        sum=sum+i
    print "sum is",sum+farg
add(5,10)
add(5,10,20)
add(5,10,20,30)
```

Output:

```
sum is15
sum is35
sum is65
```

Local and Global Variables:

When we declare a variable inside a function, it becomes a local variable. A local variable is a variable whose scope is limited only to that function where it is created. That means the local variable value is available only in that function and not outside of that function.

When the variable „a“ is declared inside myfunction() and hence it is available inside that function. Once we come out of the function, the variable „a“ is removed from memory and it is not available.

Example-1:

```
def myfunction():
    a=10
    print "Inside function",a #display 10 myfunction()
print "outside function",a # Error, not available
```

Output:

```
Inside function 10 outside
function
```

NameError: name 'a' is not defined

When a variable is declared above a function, it becomes global variable. Such variables are available to all the functions which are written after it.

Example-2:

```
a=11
def myfunction():
    b=10
    print "Inside function",a #display global var
    print "Inside function",b #display local var
myfunction()
print "outside function",a # available print
"outside function",b # error
```

Output:

```
Inside function11
Inside function10
outside function 11
outsidefunction
```

NameError: name 'b' is not defined The GlobalKeyword:

Sometimes, the global variable and the local variable may have the same name. In that case, the function, by default, refers to the local variable and ignores the global variable. So, the global variable is not accessible inside the function but outside of it, it is accessible.

Example-1:

```
a=11
def myfunction():
    a=10
    print "Inside function",a # display local variable
myfunction()
print "outside function",a # display global variable
```

Output:

```
Inside function 10
outside function 11
```

When the programmer wants to use the global variable inside a function, he can use the keyword „global“ before the variable in the beginning of the function body as:

global Example-2:

```
a=11
def myfunction():
    global a
    a=10
    print "Inside function",a # display global variable myfunction()
print "outside function",a # display global variable
```

Output:

```
Inside function 10
outside function 10
```

Recursive Functions:

A function that calls itself is known as „recursive function“. For example, we can write the factorial of 3 as:

```
factorial(3) = 3 * factorial(2) Here,
factorial(2) = 2 * factorial(1) And,
factorial(1) = 1 * factorial(0)
```

Now, if we know that the factorial(0) value is 1, all the preceding statements will evaluate and give the results:

```
factorial(3) = 3 * factorial(2)
              = 3 * 2 * factorial(1)
              = 3 * 2 * 1 * factorial(0)
              = 3 * 2 * 1 * 1
              = 6
```

From the above statements, we can write the formula to calculate factorial of any number „n“ as: $\text{factorial}(n) = n * \text{factorial}(n-1)$

Example-1:

```
def factorial(n):
    if n==0:
        result=1
    else:
        result=n*factorial(n-1)
    return result
for i in range(1,5):
    print "factorial of ",i,"is",factorial(i)
```

Output:

```
factorial of 1 is 1
factorial of 2 is 2
factorial of 3 is 6
factorial of 4 is 24
```

Anonymous Function or Lambdas:

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.

Let's take a normal function that returns square of given value:

```
def square(x):  
    return x*x
```

the same function can be written as anonymous function as:

```
lambda x: x*x
```

The colon (:) represents the beginning of the function that contains an expression $x*x$. The syntax is:

```
lambda  
argument_list:expression Example:  
f=lambda x:x*x  
value = f(5) print  
value
```

The map() Function

The advantage of the lambda operator can be seen when it is used in combination with the map() function. map() is a function with two arguments:

```
r = map(func, seq)
```

The first argument *func* is the name of a function and the second a sequence (e.g. a list) *seq*. *map()* applies the function *func* to all the elements of the sequence *seq*. It returns a new list with the elements changed by *func*

```
def fahrenheit(T):  
    return ((float(9)/5)*T + 32)  
def celsius(T):  
    return (float(5)/9)*(T-32)  
temp = (36.5, 37, 37.5, 39)  
F = map(fahrenheit, temp)  
C = map(celsius, F)
```

In the example above we haven't used lambda. By using lambda, we wouldn't have had to define and name the functions fahrenheit() and celsius(). You can see this in the following interactive session:

```
>>> Celsius = [39.2, 36.5, 37.3, 37.8]  
>>> Fahrenheit = map(lambda x: (float(9)/5)*x + 32, Celsius)  
>>> print Fahrenheit  
[102.56, 97.700000000000003, 99.140000000000001, 100.03999999999999]  
>>> C = map(lambda x: (float(5)/9)*(x-32), Fahrenheit)
```

```
>>> print C
```

```
[39.200000000000003, 36.5, 37.300000000000004, 37.799999999999997]
```

map() can be applied to more than one list. The lists have to have the same length. map() will apply its lambda function to the elements of the argument lists, i.e. it first applies to the elements with the 0th index, then to the elements with the 1st index until the n-th index is reached:

```
>>> a = [1,2,3,4]
>>> b = [17,12,11,10]
>>> c = [-1,-4,5,9]
>>>map(lambda x,y:x+y, a,b) [18,
14, 14,14]
>>>map(lambda x,y,z:x+y+z,a,b,c) [17,
10, 19,23]
>>>map(lambda x,y,z:x+y-z, a,b,c) [19,
18, 9, 5]
```

We can see in the example above that the parameter x gets its values from the list a, while y gets its values from b and z from list c.

Filtering

The function filter(function, list) offers an elegant way to filter out all the elements of a list, for which the function *function* returns True. The function filter(f,l) needs a function f as its first argument. f returns a Boolean value, i.e. either True or False. This function will be applied to every element of the list l. Only if f returns True will the element of the list be included in the result list.

```
>>> fib = [0,1,1,2,3,5,8,13,21,34,55]
>>> result = filter(lambda x: x % 2, fib)
>>> print result
[1, 1, 3, 5, 13, 21, 55]
>>> result = filter(lambda x: x % 2 == 0, fib)
>>> print result [0, 2,
8, 34]
```

Reducing a List

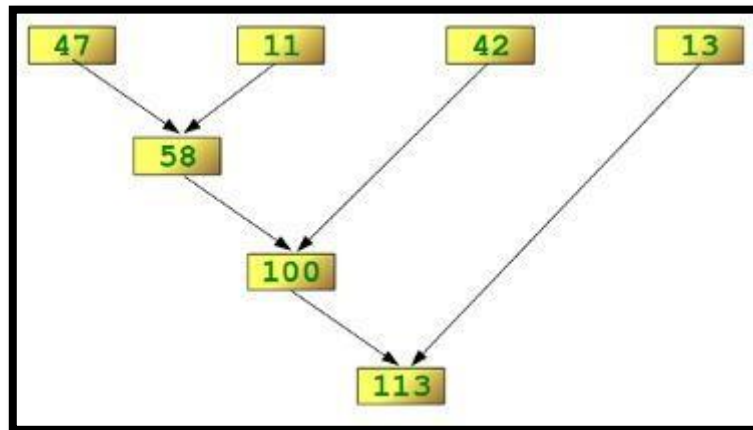
The function reduce(func, seq) continually applies the function func() to the sequence seq. It returns a single value.

If seq = [s₁, s₂, s₃, ... ,s_n], calling reduce(func, seq) works like this:

- At first the first two elements of seq will be applied to func, i.e. func(s₁,s₂) The list on which reduce() works looks now like this: [func(s₁, s₂), s₃, ... , s_n]
- In the next step func will be applied on the previous result and the third element of the list, i.e. func(func(s₁, s₂),s₃). The list looks like this now: [func(func(s₁, s₂),s₃), ... , s_n]
- Continue like this until just one element is left and return this element as the result of reduce()

We illustrate this process in the following example:

```
>>>reduce(lambda x,y: x+y, [47,11,42,13]) 113
```



The following diagram shows the intermediate steps of the calculation:

Examples of reduce ()

Determining the maximum of a list of numerical values by using reduce:

```
>>> f = lambda a,b: a if (a > b) else b
>>> reduce(f, [47,11,42,102,13])
102
>>>
```

Calculating the sum of the numbers from 1 to 100:

```
>>> reduce(lambda x, y: x+y, range(1,101)) 5050
```

Function Decorators:

A decorator is a function that accepts a function as parameter and returns a function. A decorator takes the result of a function, modifies the result and returns it. Thus decorators are useful to perform some additional processing required by a function.

The following steps are generally involved in creation of decorators:

- We should define a decorator function with another function name as parameter.
- We should define a function inside the decorator function. This function actually modifies or decorates the value of the function passed to the decorator function.
- Return the inner function that has processed or decorated the value.

Example-1:

```
def decor(fun):
    def inner():
        value=fun()
        return value+2
    return inner
def num():
    return 10
result=decor(num)
print result()
```

Output:

12

To apply the decorator to any function, we can use '@' symbol and decorator name just

above the function definition.

Example-2: A python program to create two decorators. `def decor1(fun):`

```
def inner():
    value=fun()
    return value+2
return inner
def decor2(fun):
    def inner():
        value=fun()
        return value*2
    return inner
def num():
    return 10

result=decor1(decor2(num))
print result()
```

Output:

22

Example-3: A python program to create two decorators to the same function using „@“ symbol.

```
def decor1(fun):
    def inner():
        value=fun()
        return value+2
    return inner
def decor2(fun):
    def inner():
        value=fun()
        return value*2
    return inner
@decor1
@decor2
def num():
    return 10

print num()
```

Output:

22

Function Generators:

A generator is a function that produces a sequence of results instead of a single value.

„yield“ statement is used to return the value. def

```
mygen(n):
    i = 0
    while i<n:
        yield i
        i +=1
g=mygen(6)
for i in g:
    print i,
```

Output:

0 1 2 3 4 5

Note: „yield“ statement can be used to hold the sequence of results and return it.

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__`. For instance, use your favourite text editor to create a file called `fib.py` in the current directory with the following contents:

```
# Fibonacci numbers module
def fib(n): # write Fibonacci series up to n
    a, b = 0,1
    while b < n:
        printb,
        a, b = b, a+b
def fib2(n): # return Fibonacci series up to n
    result =[]
    a, b = 0,1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
```

Now enter the Python interpreter and import this module with the following command:

```
>>>import fibo
```

This does not enter the names of the functions defined in `fib` directly in the current symbol table; it only enters the module name `fibo` there. Using the module name you can access the functions:

```
>>>fibo.fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>>fibo.fib2(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55,89]
>>>fibo.name_____
'fibo'
```

from statement:

- 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 name is encountered in an import statement. (They are also run if the file is executed as ascript.)
- 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, if you know what you are doing you can touch 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. Forexample:

```
>>> from fibo import fib, fib2
```

```
>>> fib(500)
```

```
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This does not introduce the module name from which the imports are taken in the local symbol table (so in the example, `fibo` is not defined).

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

```
>>> from fibo import *
```

```
>>> fib(500)
```

```
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

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 globalvariable.
- Each function has its own local namespace. Class methods follow the same scoping rule as ordinaryfunctions.
- Python makes educated guesses on whether variables are local or global. It assumes that any variable assigned a value in a function islocal.
- 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 thevariable.
- 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.

Unit-IV

A sequence is a datatype that represents a group of elements. The purpose of any sequence is to store and process group elements. In python, strings, lists, tuples and dictionaries are very important sequence datatypes.

LIST:

A list is similar to an array that consists of a group of elements or items. Just like an array, a list can store elements. But, there is one major difference between an array and a list. An array can store only one type of elements whereas a list can store different types of elements. Hence lists are more versatile and useful than an array.

Creating a List:

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

```
student = [556, "Mothi", 84, 96, 84, 75, 84]
```

We can create empty list without any elements by simply writing empty square brackets as: `student=[]`

We can create a list by embedding the elements inside a pair of square braces []. The elements in the list should be separated by a comma (,).

Accessing Values in list:

To access values in lists, use the square brackets for slicing along with the index or indices to obtain value available at that index. To view the elements of a list as a whole, we can simply pass the list name to print function.

negative Indexing	-7	-6	-5	-4	-3	-2	-1
Positive Indexing	0	1	2	3	4	5	6
Student	556	"Mothi"	84	96	84	75	84

Ex:

```
student = [556, "Mothi", 84, 96, 84, 75, 84]
print student
print student[0] # Access 0th element
print student[0:2] # Access 0th to 1st elements
print student[2:] # Access 2nd to end of list elements
print student[:3] # Access starting to 2nd elements
print student[:] # Access starting to ending elements
print student[-1] # Access last index value
print student[-1:-7:-1] # Access elements in reverse order
```

Output:

```
[556, "Mothi", 84, 96, 84, 75, 84]
Mothi
[556, "Mothi"]
[84, 96, 84, 75, 84]
[556, "Mothi", 84]
[556, "Mothi", 84, 96, 84, 75, 84]
```

84

[84, 75, 84, 96, 84, "Mothi"]

Creating lists using range() function:

We can use range() function to generate a sequence of integers which can be stored in a list. To store numbers from 0 to 10 in a list as follows.

```
numbers = list( range(0,11) )  
print numbers # [0,1,2,3,4,5,6,7,8,9,10]
```

To store even numbers from 0 to 10 in a list as follows.

```
numbers = list( range(0,11,2) )  
print numbers # [0,2,4,6,8,10]
```

Looping on lists:

We can also display list by using for loop (or) while loop. The len() function is useful to know the numbers of elements in the list. while loop retrieves starting from 0th to the last element i.e. n-1

Ex-1:

```
numbers = [1,2,3,4,5]  
for i in numbers:  
    print i,
```

Output:

1 2 3 4 5

Updating and deleting lists:

Lists are *mutable*. It means we can modify the contents of a list. We can append, update or delete the elements of a list depending upon our requirements.

Appending an element means adding an element at the end of the list. To, append a new element to the list, we should use the append() method.

Example:

```
lst=[1,2,4,5,8,6]  
printlst                # [1,2,4,5,8,6]  
lst.append(9)  
printlst                # [1,2,4,5,8,6,9]
```

Updating an element means changing the value of the element in the list. This can be done by accessing the specific element using indexing or slicing and assigning a new value.

Example:

```
lst=[4,7,6,8,9,3]  
printlst                #[4,7,6,8,9,3]  
lst[2]=5                # updates 2nd element in the list  
printlst                # [4,7,5,8,9,3]  
lst[2:5]=10,11,12       # update 2nd element to 4th element in the list  
printlst                # [4,7,10,11,12,3]
```

Deleting an element from the list can be done using 'del' statement. The *del* statement takes the position number of the element to be deleted.

Example:

```
lst=[5,7,1,8,9,6]
del lst[3]           # delete 3rd element from the list i.e., 8
print lst           # [5,7,1,9,6]
```

If we want to delete entire list, we can give statement like *del lst*.

Concatenation of Two lists:

We can simply use „+“ operator on two lists to join them. For example, „x“ and „y“ are two lists. If we write x+y, the list „y“ is joined at the end of the list „x“.

Example:

```
x=[10,20,32,15,16]
y=[45,18,78,14,86]
print x+y           # [10,20,32,15,16,45,18,78,14,86]
```

Repetition of Lists:

We can repeat the elements of a list „n“ number of times using „*“ operator.

```
x=[10,54,87,96,45]
print x*2           # [10,54,87,96,45,10,54,87,96,45]
```

Membership in Lists:

We can check if an element is a member of a list by using „in“ and „not in“ operator. If the element is a member of the list, then „in“ operator returns **True** otherwise returns **False**. If the element is not in the list, then „not in“ operator returns **True** otherwise returns **False**.

Example:

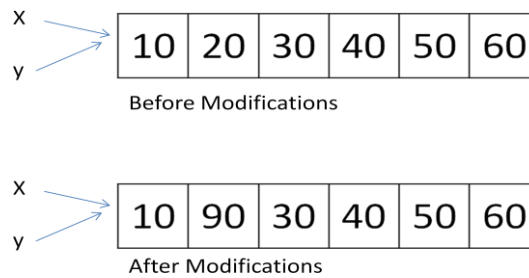
```
x=[10,20,30,45,55,65] a=20
print a in x         # True
a=25
print a in x         # False
a=45
print a not in x # False a=40
print a not in x # True
```

Aliasing and Cloning Lists:

Giving a new name to an existing list is called '*aliasing*'. The new name is called '*alias name*'. To provide a new name to this list, we can simply use assignment operator (=).

Example:

```
x = [10, 20, 30, 40, 50, 60]
y=x                 # x is aliased as y
print x             #[10,20,30,40,50,60]
print y             #[10,20,30,40,50,60]
x[1]=90             # modify 1st element in x
print x             # [10,90,30,40,50,60]
print y             #[10,90,30,40,50,60]
```

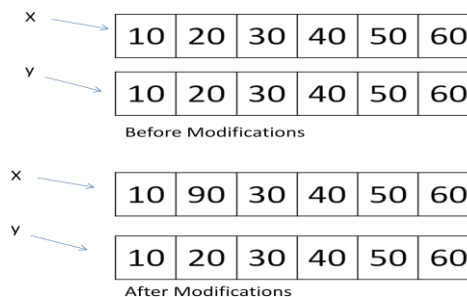


In this case we are having only one list of elements but with two different names „x“ and „y“. Here, „x“ is the original name and „y“ is the alias name for the same list. Hence, any modifications done to x“ will also modify „y“ and vice versa.

Obtaining exact copy of an existing object (or list) is called „cloning“. To Clone a list, we can take help of the slicing operation [:].

Example:

```
x = [10, 20, 30, 40, 50, 60]
y=x[:]
printx      # x is cloned asy
printy      #[10,20,30,40,50,60]
x[1]=90      #[10,20,30,40,50,60]
printx      # modify 1st element in x
printy      # [10,90,30,40,50,60]
            #[10,20,30,40,50,60]
```



When we clone a list like this, a separate copy of all the elements is stored into „y“. The lists „x“ and „y“ are independent lists. Hence, any modification to „x“ will not affect „y“ and vice versa.

Methods in Lists:

Method	Description
<i>lst.index(x)</i>	Returns the first occurrence of x in the list.
<i>lst.append(x)</i>	Appends x at the end of the list.
<i>lst.insert(i,x)</i>	Inserts x to the list in the position specified by i.
<i>lst.copy()</i>	Copies all the list elements into a new list and returns it.
<i>lst.extend(lst2)</i>	Appends lst2 to list.
<i>lst.count(x)</i>	Returns number of occurrences of x in the list.
<i>lst.remove(x)</i>	Removes x from the list.
<i>lst.pop()</i>	Removes the ending element from the list.
<i>lst.sort()</i>	Sorts the elements of list into ascending order.
<i>lst.reverse()</i>	Reverses the sequence of elements in the list.
<i>lst.clear()</i>	Deletes all elements from the list.
<i>max(lst)</i>	Returns biggest element in the list.
<i>min(lst)</i>	Returns smallest element in the list.

Example:

```
lst=[10,25,45,51,45,51,21,65]
lst.insert(1,46)
printlst          # [10,46,25,45,51,45,51,21,65]
printlst.count(45) # 2
```

Finding Common Elements in Lists:

Sometimes, it is useful to know which elements are repeated in two lists. For example, there is a scholarship for which a group of students enrolled in a college. There is another scholarship for which another group of students got enrolled. Now, we want to know the names of the students who enrolled for both the scholarships so that we can restrict them to take only one scholarship. That means, we are supposed to find out the common students (or elements) both the lists.

First of all, we should convert the lists into sets, using `set()` function, as: `set(list)`. Then we should find the common elements in the two sets using `intersection()` method.

Example:

```
scholar1=[ „mothi“, „sudheer“, „vinay“, „narendra“, „ramakoteswararao“ ] scholar2=[
„vinay“, „narendra“, „ramesh“]
s1=set(scholar1) s2=set(scholar2)
s3=s1.intersection(s2) common
=list(s3)
printcommon          # display [ „vinay“, „narendra“]
```

Nested Lists:

A list within another list is called a *nested list*. We know that a list contains several elements. When we take a list as an element in another list, then that list is called a nested list.

Example:

```
a=[10,20,30]
b=[45,65,a]
printb          # display [ 45, 65, [ 10, 20, 30 ] ]
printb[1]       # display 65
printb[2]       # display [ 10, 20, 30 ]
printb[2][0]    # display 10
print b[2][1] # display 20 print b[2][2]
# display 30 for x in b[2]:
print x,       # display 10 20 30
```


Nested Lists as Matrices:

Suppose we want to create a matrix with 3 rows 3 columns, we should create a list with 3 other lists as:

```
mat = [ [ 1, 2, 3 ] , [ 4, 5, 6 ] , [ 7, 8, 9 ] ]
```

Here, „mat“ is a list that contains 3 lists which are rows of the „mat“ list. Each row contains again 3 elements as:

```
[ [ 1, 2, 3],      # first row
  [ 4, 5, 6],      # second row
  [ 7, 8, 9]]      # third row
```

Example:

```
mat=[[1,2,3],[4,5,6],[7,8,9]]
for r in mat:
    print r
print ""
m=len(mat)
n=len(mat[0])
for i in range(0,m):
    for j in range(0,n):
        print mat[i][j],
    print ""
print ""
```

```
[1, 2, 3]
[4, 5, 6]
[7, 8, 9]
```

```
1 2 3
4 5 6
7 8 9
```

One of the main use of nested lists is that they can be used to represent matrices. A matrix represents a group of elements arranged in several rows and columns. In python, matrices are created as 2D arrays or using matrix object in numpy. We can also create a matrix using nested lists.

Q Write a program to perform addition of two matrices.

```
a=[[1,2,3],[4,5,6],[7,8,9]] b=[[4,5,6],[7,8,9],[1,2,3]]
```

```
c=[[0,0,0],[0,0,0],[0,0,0]]
```

```
m1=len(a) n1=len(a[0])
```

```
m2=len(b) n2=len(b[0])
```

```
for i in range(0,m1):
```

```
    for j in range(0,n1):
```

```
        for i in range(0,n1):
```

```
            for j in range(0,n1):
```

```
                c[i][j]= a[i][j]+b[i][j]
```

```
5      7      9
11     13     15
8      10     12
```

Q) Write a program to perform multiplication of two matrices.

```
a=[[1,2,3],[4,5,6]]
b=[[4,5],[7,8],[1,2]]
c=[[0,0],[0,0]]
m1=len(a) n1=len(a[0])
m2=len(b) n2=len(b[0])
    for i in range(0,m1):
        for j in range(0,n2):
            for k in range(0,n1):
                c[i][j] +=a[i][k]*b[k][j]
    for i in range(0,m1):
        for j in range(0,n2):
            print "\t",c[i][j],
        print ""
```

21 27
57 72

List Comprehensions:

List comprehensions represent creation of new lists from an iterable object (like a list, set, tuple, dictionary or range) that satisfy a given condition. List comprehensions contain very compact code usually a single statement that performs the task.

We want to create a list with squares of integers from 1 to 100. We can write code as: squares= []

```
for i in range(1,11):
    squares.append(i**2)
```

The preceding code will create „squares“ list with the elements as shown below:

```
[ 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

The previous code can be rewritten in a compact way as:

```
squares=[x**2 for x in range(1,11)]
```

This is called list comprehension. From this, we can understand that a list comprehension consists of square braces containing an expression (i.e., x^2). After the expression, a for loop and then zero or more if statements can be written.

```
[ expression for item1 in iterable if statement1
    for item1 in iterable if statement2
    for item1 in iterable if statement3.....]
```

Example:

```
Even_squares = [ x**2 for x in range(1,11) if x%2==0]
```

It will display the list even squares as list.

```
[ 4, 16, 36, 64, 100]
```

TUPLE:

A Tuple is a python sequence which stores a group of elements or items. Tuples are similar to lists but the main difference is tuples are immutable whereas lists are mutable. Once we create a tuple we cannot modify its elements. Hence, we cannot perform operations like `append()`, `extend()`, `insert()`, `remove()`, `pop()` and `clear()` on tuples. Tuples are generally used to store data which should not be modified and retrieve that data on demand.

Creating Tuples:

We can create a tuple by writing elements separated by commas inside parentheses (). The elements can be same datatype or different types.

To create an empty tuple, we can simply write empty parenthesis, as: `tup= ()`

To create a tuple with only one element, we can mention that element in parenthesis and after that a comma is needed. In the absence of comma, python treats the element as ordinary data type.

<pre>tup = (10) print tup # display 10 print type(tup) # display <type „int“></pre>	<pre>tup = (10,) print tup # display 10 print type(tup) # display<type„tuple“></pre>
---	--

To create a tuple with different types of elements:

```
tup=(10, 20, 31.5, „Gudivada“)
```

If we do not mention any brackets and write the elements separating them by comma, then they are taken by default as a tuple.

```
tup= 10, 20, 34, 47
```

It is possible to create a tuple from a list. This is done by converting a list into a tuple using tuple function.

```
n=[1,2,3,4]
tp=tuple(n)
printtp          # display(1,2,3,4)
```

Another way to create a tuple by using `range()` function that returns a sequence.

```
t=tuple(range(2,11,2))
printt           # display(2,4,6,8,10)
```

Accessing the tuple elements:

Accessing the elements from a tuple can be done using indexing or slicing. This is same as that of a list. Indexing represents the position number of the element in the tuple. The position starts from 0.

```
tup=(50,60,70,80,90)
printtup[0]          # display50
printtup[1:4]        # display(60,70,80)
print tup[-1]        # display90
printtup[-1:-4:-1]   # display (90,80,70)
printtup[-4:-1]      # display(60,70,80)
```

Updating and deleting elements:

Tuples are immutable which means you cannot update, change or delete the values of tuple elements.

Example-1:

```
tupl.py - C:/Python27/tupl.py (2.7.13)
File Edit Format Run Options Window Help
a=(1,2,3,4,5)
a[2]=6
print a

Traceback (most recent call last):
  File "C:/Python27/tupl.py", line 2, in <module>
    a[2]=6
TypeError: 'tuple' object does not support item assignment
>>>
```

Example-2:

```
tupl.py - C:/Python27/tupl.py (2.7.13)
File Edit Format Run Options Window Help
a=(1,2,3,4,5)
del a[2]
print a

Traceback (most recent call last):
  File "C:/Python27/tupl.py", line 2, in <module>
    del a[2]
TypeError: 'tuple' object doesn't support item deletion
>>>
```

However, you can always delete the entire tuple by using the statement.

```
tupl.py - C:/Python27/tupl.py (2.7.13)
File Edit Format Run Options Window Help
a=(1,2,3,4,5)
del a
print a

Traceback (most recent call last):
  File "C:/Python27/tupl.py", line 3, in <module>
    print a
NameError: name 'a' is not defined
>>>
```

Note that this exception is raised because you are trying print the deleted element.

Operations on tuple:

Operation	Description
len(t)	Return the length of tuple.
tup1+tup2	Concatenation of two tuples.
Tup*n	Repetition of tuple values in n number of times.
x in tup	Return True if x is found in tuple otherwise returns False.
cmp(tup1,tup2)	Compare elements of both tuples
max(tup)	Returns the maximum value in tuple.
min(tup)	Returns the minimum value in tuple.
tuple(list)	Convert list into tuple.
tup.count(x)	Returns how many times the element „x“ is found in tuple.
tup.index(x)	Returns the first occurrence of the element „x“ in tuple. Raises ValueError if „x“ is not found in the tuple.
sorted(tup)	Sorts the elements of tuple into ascending order. sorted(tup,reverse=True) will sort in reverse order.

cmp(tuple1, tuple2)

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

Syntax

```
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:

```
tuple1 = (123,'xyz')
tuple2 = (456,'abc')
print cmp(tuple1, tuple2)    #display-1
print cmp(tuple2, tuple1)    #display1
```

Nested Tuples:

Python allows you to define a tuple inside another tuple. This is called a *nested tuple*.

```
students=((("RAVI", "CSE", 92.00), ("RAMU", "ECE", 93.00), ("RAJA", "EEE", 87.00))
for i in students:
    print i
```

Output: ("RAVI", "CSE", 92.00)
("RAMU", "ECE", 93.00)
("RAJA", "EEE", 87.00)

SET:

Set is another data structure supported by python. Basically, sets are same as lists but with a difference that sets are lists with no duplicate entries. Technically a set is a mutable and an unordered collection of items. This means that we can easily add or remove items from it.

Creating a Set:

A set is created by placing all the elements inside curly brackets `{ }`. Separated by comma or by using the built-in function `set()`.

Syntax:

```
Set_variable_name={var1, var2, var3, var4, .....}
```

Example:

```
s={1, 2.5, "abc" }  
print s          # display set( [ 1, 2.5, "abc" ] )
```

Converting a list into set:

A set can have any number of items and they may be of different data types. `set()` function is used to converting list into set.

```
s=set( [ 1, 2.5, "abc" ] )  
prints          # display set( [ 1, 2.5, "abc" ] )
```

We can also convert tuple or string into set.

```
tup= ( 1, 2, 3, 4, 5)  
print set(tup) # set( [ 1, 2, 3, 4, 5 ] )  
str="MOTHILAL"  
printstr      # set( [ 'i', 'h', 'm', 't', 'o' ] )
```

Operations on set:

Sno	Operation	Result
1	<code>len(s)</code>	number of elements in set <i>s</i> (cardinality)
2	<code>x in s</code>	test <i>x</i> for membership in <i>s</i>
3	<code>x not in s</code>	test <i>x</i> for non-membership in <i>s</i>
4	<code>s.issubset(t)</code> (or) <code>s <= t</code>	test whether every element in <i>s</i> is in <i>t</i>
5	<code>s.issuperset(t)</code> (or) <code>s >= t</code>	test whether every element in <i>t</i> is in <i>s</i>
6	<code>s == t</code>	Returns True if two sets are equivalent and returns False.
7	<code>s != t</code>	Returns True if two sets are not equivalent and returns False.
8	<code>s.union(t)</code> (or) <code>s t</code>	new set with elements from both <i>s</i> and <i>t</i>
9	<code>s.intersection(t)</code> (or) <code>s & t</code>	new set with elements common to <i>s</i> and <i>t</i>

Sno	Operation	Result
10	<code>s.difference(t)</code> (or) <code>s-t</code>	new set with elements in <i>s</i> but not in <i>t</i>
11	<code>s.symmetric_difference(t)</code> (or) <code>s ^ t</code>	new set with elements in either <i>s</i> or <i>t</i> but not both
12	<code>s.copy()</code>	new set with a shallow copy of <i>s</i>
13	<code>s.update(t)</code>	return set <i>s</i> with elements added from <i>t</i>
14	<code>s.intersection_update(t)</code>	return set <i>s</i> keeping only elements also found in <i>t</i>
15	<code>s.difference_update(t)</code>	return set <i>s</i> after removing elements found in <i>t</i>
16	<code>s.symmetric_difference_update(t)</code>	return set <i>s</i> with elements from <i>s</i> or <i>t</i> but not both
17	<code>s.add(x)</code>	add element <i>x</i> to set <i>s</i>
18	<code>s.remove(x)</code>	remove <i>x</i> from set <i>s</i> ; raises <u>KeyError</u> if not present
19	<code>s.discard(x)</code>	removes <i>x</i> from set <i>s</i> if present
20	<code>s.pop()</code>	remove and return an arbitrary element from <i>s</i> ; raises <u>KeyError</u> if empty
21	<code>s.clear()</code>	remove all elements from set <i>s</i>
22	<code>max(s)</code>	Returns Maximum value in a set
23	<code>min(s)</code>	Returns Minimum value in a set
24	<code>sorted(s)</code>	Return a new sorted list from the elements in the set.

Note:

To create an empty set you cannot write `s={ }`, because python will make this as a directory. Therefore, to create an empty set use `set()` function.

<pre>s=set() printtype(s) # display<type,,set"></pre>	<pre>s={ } print type(s) # display <type,,dict"></pre>
---	--

Updating a set:

Since sets are unordered, indexing has no meaning. Set operations do not allow users to access or change an element using indexing or slicing.

Dictionary:

A dictionary represents a group of elements arranged in the form of key-value pairs. The first element is considered as „key“ and the immediate next element is taken as its „value“. The key and its value are separated by a colon (:). All the key-value pairs in a dictionary are inserted in curly braces {}.

```
d= { „Regd.No“: 556, „Name“:“Mothi“, „Branch“: „CSE“ }
```

Here, the name of dictionary is „dict“. The first element in the dictionary is a string „Regd.No“. So, this is called „key“. The second element is 556 which is taken as its „value“.

Example:

```
d={„Regd.No“:556,„Name“:“Mothi“,„Branch“:„CSE“}  
printd[„Regd.No“]          # 556  
printd[„Name“]              # Mothi  
printd[„Branch“]            # CSE
```

To access the elements of a dictionary, we should not use indexing or slicing. For example, dict[0] or dict[1:3] etc. expressions will give error. To access the value associated with a key, we can mention the key name inside the square braces, as: dict[„Name“].

If we want to know how many key-value pairs are there in a dictionary, we can use the len() function, as shown

```
d={„Regd.No“:556,„Name“:“Mothi“,„Branch“:„CSE“} printlen(d)  
# 3
```

We can also insert a new key-value pair into an existing dictionary. This is done by mentioning the key and assigning a value to it.

```
d={'Regd.No':556,'Name':'Mothi','Branch':'CSE'}  
printd          #{'Branch': 'CSE', 'Name': 'Mothi', 'Regd.No': 556}  
d['Gender']="Male"  
printd          # {'Gender': 'Male', 'Branch': 'CSE', 'Name': 'Mothi', 'Regd.No': 556}
```

Suppose, we want to delete a key-value pair from the dictionary, we can use *del* statements as:

```
del dict[„Regd.No“] #{'Gender': 'Male', 'Branch': 'CSE', 'Name': 'Mothi'}
```

To Test whether a „key“ is available in a dictionary or not, we can use „in“ and „not in“ operators. These operators return either True or False.

```
„Name“ in d          #check if „Name“ is a key in d and returns True/False
```

We can use any datatypes for value. For example, a value can be a number, string, list, tuple or another dictionary. But keys should obey the rules:

- Keys should be unique. It means, duplicate keys are not allowed. If we enter same key again, the old key will be overwritten and only the new key will be available.

```
emp={'nag':10,'vishnu':20,'nag':20}  
print emp # {'nag': 20, 'vishnu': 20}
```

- Keys should be immutable type. For example, we can use a number, string or tuples as keys since they are immutable. We cannot use lists or dictionaries as keys. If they are used as keys, we will get „TypeError“.

```
emp=[ 'nag':10,'vishnu':20,'nag':20}  
Traceback (most recent call last):  
  File "<pyshell#2>", line 1, in <module>  
    emp=[ 'nag':10,'vishnu':20,'nag':20}  
TypeError: unhashable type: 'list'
```


Dictionary Methods:

Method	Description
<code>d.clear()</code>	Removes all key-value pairs from dictionary „d“.
<code>d2=d.copy()</code>	Copies all elements from „d“ into a new dictionary d2.
<code>d.fromkeys(s [,v])</code>	Create a new dictionary with keys from sequence „s“ and values all set to „v“.
<code>d.get(k [,v])</code>	Returns the value associated with key „k“. If key is not found, it returns „v“.
<code>d.items()</code>	Returns an object that contains key-value pairs of „d“. The pairs are stored as tuples in the object.
<code>d.keys()</code>	Returns a sequence of keys from the dictionary „d“.
<code>d.values()</code>	Returns a sequence of values from the dictionary „d“.
<code>d.update(x)</code>	Adds all elements from dictionary „x“ to „d“.
<code>d.pop(k [,v])</code>	Removes the key „k“ and its value from „d“ and returns the value. If key is not found, then the value „v“ is returned. If key is not found and „v“ is not mentioned then „KeyError“ is raised.
<code>d.setdefault(k [,v])</code>	If key „k“ is found, its value is returned. If key is not found, then the k, v pair is stored into the dictionary „d“.

Using for loop with Dictionaries:

for loop is very convenient to retrieve the elements of a dictionary. Let's take a simple dictionary that contains color code and its names as:

```
colors = { 'r':"RED", 'g':"GREEN", 'b':"BLUE", 'w':"WHITE" }
```

Here, „r“, „g“, „b“ represents keys and „RED“, „GREEN“, „BLUE“ and „WHITE“ indicate values.

```
colors = { 'r':"RED", 'g':"GREEN", 'b':"BLUE", 'w':"WHITE" }
```

```
for k in colors:
```

```
    print k # displays only keys for k
```

```
in colors:
```

```
    print colors[k] # keys to dictionary and display the values
```

Converting Lists into Dictionary:

When we have two lists, it is possible to convert them into a dictionary. For example, we have two lists containing names of countries and names of their capital cities.

There are two steps involved to convert the lists into a dictionary. The first step is to create a „zip“ class object by passing the two lists to `zip()` function. The `zip()` function is useful to convert the sequences into a zip class object. The second step is to convert the zip object into a dictionary by using `dict()` function.

Example:

```
countries = [ 'USA', 'INDIA', 'GERMANY', 'FRANCE' ]  
cities = [ 'Washington', 'New Delhi', 'Berlin', 'Paris' ]  
z=zip(countries, cities)  
d=dict(z)  
print d
```

Output:

```
{'GERMANY': 'Berlin', 'INDIA': 'New Delhi', 'USA': 'Washington', 'FRANCE': 'Paris'}
```

Converting Strings into Dictionary:

When a string is given with key and value pairs separated by some delimiter like a comma (,) we can convert the string into a dictionary and use it as dictionary.

```
s="Vijay=23,Ganesh=20,Lakshmi=19,Nikhi  
l=22" s1=s.split(',')  
s2=[]  
d={ }  
for i in s1:  
    s2.append(i.spli  
t('='))  
print d  
  
{'Ganesh': '20', 'Lakshmi': '19', 'Nikhil': '22', 'Vijay': '23'}
```

Q) A Python program to create a dictionary and find the sum of values.

```
d={'m1':85,'m3':84,'eng':86,'c':91}  
sum=0  
for i in d.values():  
    sum+=i  
printsum      # 346
```

Q) A Python program to create a dictionary with cricket player's names and scores in a match. Also we are retrieving runs by entering the player's name.

```
n=input("Enter How many players? ")  
d={ }  
for i in range(0,n):  
    k=input("Enter Player name: ")  
    v=input("Enter score: ")  
    d[k]=v  
print d  
name=input("Enter name of player for score: ")  
print "The Score is",d[name]
```

```
Enter   How   many  
players? 3 Enter Player  
name: "Sachin" Enter  
score:98  
Enter   Player   name:  
"Sehwag"      Enter  
score:91  
Enter Player name:  
"Dhoni" Enter score:95  
{'Sehwag': 91, 'Sachin': 98, 'Dhoni':  
95} Enter name of player for score:  
"Sehwag" The Score is 91
```

Unit-V

Sorting:

Sorting refers to arranging data in a particular format. Sorting algorithm specifies the way to arrange data in a particular order. Most common orders are in numerical or lexicographical order.

The importance of sorting lies in the fact that data searching can be optimized to a very high level, if data is stored in a sorted manner. Sorting is also used to represent data in more readable formats.

Bubble Sort:

Bubble sort, sometimes referred to as sinking sort, is a simple sorting algorithm that repeatedly steps through the list to be sorted, compares each pair of adjacent items and swaps them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted.

```
def bubbleSort(arr):
    n = len(arr)

    # Traverse through all array elements
    for i in range(n):
        # Last i elements are already in place
        for j in range(0, n-i-1):
            # traverse the array from 0 to n-i-1
            # Swap if the element found is greater
            # than the next element
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
    # Driver code to test above
    arr = [64, 34, 25, 12, 22, 11, 90]
    bubbleSort(arr)
    print("Sorted array is:")
    for i in range(len(arr)):
        print("%d" % arr[i])
```

Selection Sort:

The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array.

- 1) The subarray which is already sorted.
- 2) Remaining subarray which is unsorted.

In every iteration of selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the sorted subarray.

```
def selection_sort(alist):
    for i in range(0, len(alist) - 1):
        smallest = i
```

```

        for j in range(i + 1, len(alist)):
            if alist[j] < alist[smallest]:
                smallest = j
    alist[i], alist[smallest] = alist[smallest], alist[i]
    alist = input('Enter the list of numbers: ').split()
    alist = [int(x) for x in alist]
    selection_sort(alist)
    print('Sorted list: ', end=")
    print(alist)

```

Insertion Sort:

This is an in-place comparison-based sorting algorithm. Here, a sub-list is maintained which is always sorted. For example, the lower part of an array is maintained to be sorted. An element which is to be 'insert'ed in this sorted sub-list, has to find its appropriate place and then it has to be inserted there. The array is searched sequentially and unsorted items are moved and inserted into the sorted sub-list (in the same array). This algorithm is not suitable for large data sets as its average and worst case complexity are of $O(n^2)$, where n is the number of items.

```

def insertion_sort(alist):
    for i in range(1, len(alist)):
        temp = alist[i]
        j = i - 1
        while (j >= 0 and temp < alist[j]):
            alist[j + 1] = alist[j]
            j = j - 1
        alist[j + 1] = temp
    alist = input('Enter the list of numbers: ').split()
    alist = [int(x) for x in alist]
    insertion_sort(alist)
    print('Sorted list: ', end=")
    print(alist)

```

Merge sort:

Merge sort is a sorting technique based on divide and conquer technique. With worst-case time complexity being $O(n \log n)$, it is one of the most respected algorithms. Merge sort first divides the array into equal halves and then combines them in a sorted manner.

```

def merge_sort(alist, start, end):
    """Sorts the list from indexes start to end - 1 inclusive."""
    if end - start > 1:
        mid = (start + end)//2
        merge_sort(alist, start, mid)
        merge_sort(alist, mid, end)
        merge_list(alist, start, mid, end)
    def merge_list(alist, start, mid, end):
        left = alist[start:mid]
        right = alist[mid:end]
        k = start
        i = 0
        j = 0
        while (start + i < mid and mid + j < end):

```

```

        if (left[i]<= right[j]):
alist[k] = left[i]
i = i + 1
        else:
alist[k] = right[j]
        j = j + 1
        k = k + 1
        if start + i< mid:
            while k < end:
alist[k] = left[i]
i = i + 1
            k = k + 1
        else:
            while k < end:
alist[k] = right[j]
            j = j + 1
            k = k + 1
alist = input('Enter the list of numbers: ').split()
alist = [int(x) for x in alist]
merge_sort(alist, 0, len(alist))
print('Sorted list: ', end=")
print(alist)

```

Quick sort:

Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays. A large array is partitioned into two arrays one of which holds values smaller than the specified value, say pivot, based on which the partition is made and another array holds values greater than the pivot value.

Quick sort partitions an array and then calls itself recursively twice to sort the two resulting subarrays. This algorithm is quite efficient for large-sized data sets as its average and worst case complexity are of $O(n^2)$, where n is the number of items. The algorithm is given below:

- Step 1 – Choose the highest index value has pivot
- Step 2 – Take two variables to point left and right of the list excluding pivot
- Step 3 – left points to the low index
- Step 4 – right points to the high
- Step 5 – while value at left is less than pivot move right
- Step 6 – while value at right is greater than pivot move left
- Step 7 – if both step 5 and step 6 does not match swap left and right
- Step 8 – if $\text{left} \geq \text{right}$, the point where they met is new pivot

```

def quicksort(alist, start, end):
    """Sorts the list from indexes start to end - 1 inclusive."""
    if end - start > 1:
        p = partition(alist, start, end)
        quicksort(alist, start, p)
        quicksort(alist, p + 1, end)
    def partition(alist, start, end):
        pivot = alist[start]

```

```

i = start + 1
j = end - 1
while True:
    while (i <= j and alist[i] <= pivot):
        i = i + 1
    while (i <= j and alist[j] >= pivot):
        j = j - 1
    if i <= j:
        alist[i], alist[j] = alist[j], alist[i]
    else:
        alist[start], alist[j] = alist[j], alist[start]
    return j
alist = input('Enter the list of numbers: ').split()
alist = [int(x) for x in alist]
quicksort(alist, 0, len(alist))
print('Sorted list: ', end='')
print(alist)

```

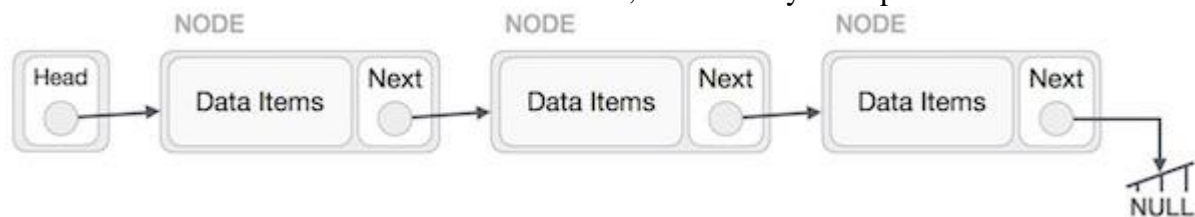
Linked List:

A linked list is a sequence of data structures, which are connected together via links. Linked List is a sequence of links which contains items. Each link contains a connection to another link. Linked list is the second most-used data structure after array. Following are the important terms to understand the concept of Linked List.

- Link – Each link of a linked list can store a data called an element.
- Next – Each link of a linked list contains a link to the next link called Next.
- LinkedList – A Linked List contains the connection link to the first link called First.

Linked List Representation:

Linked list can be visualized as a chain of nodes, where every node points to the next node.



As per the above illustration, following are the important points to be considered.

- Linked List contains a link element called first.
- Each link carries a data field(s) and a link field called next.
- Each link is linked with its next link using its next link.
- Last link carries a link as null to mark the end of the list.

Types of Linked List:

Following are the various types of linked list.

- Simple Linked List – Item navigation is forward only.
- Doubly Linked List – Items can be navigated forward and backward.

- Circular Linked List – Last item contains link of the first element as next and the first element has a link to the last element as previous.

Creation of Linked list

A linked list is created by using the node class we studied in the last chapter. We create a Node object and create another class to use this node object. We pass the appropriate values through the node object to point to the next data elements. The below program creates the linked list with three data elements. In the next section we will see how to traverse the linked list.

```
class Node:
    def __init__(self, dataval=None):
self.dataval = dataval
self.nextval = None
class SLinkedList:
    def __init__(self):
self.headval = None
list1 = SLinkedList()
list1.headval = Node("Mon")
e2 = Node("Tue")
e3 = Node("Wed")
# Link first Node to second node
list1.headval.nextval = e2
# Link second Node to third node
e2.nextval = e3
```

Traversing a Linked List

Singly linked lists can be traversed in only forward direction starting from the first data element. We simply print the value of the next data element by assigning the pointer of the next node to the current data element.

```
class Node:
    def __init__(self, dataval=None):
self.dataval = dataval
self.nextval = None
class SLinkedList:
    def __init__(self):
self.headval = None
    def listprint(self):
printval = self.headval
    while printval is not None:
        print (printval.dataval)
printval = printval.nextval
```

```

list = SLinkedList()
list.headval = Node("Mon")
e2 = Node("Tue")
e3 = Node("Wed")
# Link first Node to second node
list.headval.nextval = e2
# Link second Node to third node
e2.nextval = e3
list.listprint()

```

Insertion in a Linked List

Inserting element in the linked list involves reassigning the pointers from the existing nodes to the newly inserted node. Depending on whether the new data element is getting inserted at the beginning or at the middle or at the end of the linked list, we have the below scenarios.

Inserting at the Beginning of the Linked List

This involves pointing the next pointer of the new data node to the current head of the linked list. So the current head of the linked list becomes the second data element and the new node becomes the head of the linked list.

```

class Node:
    def __init__(self, dataval=None):
self.dataval = dataval
self.nextval = None
class SLinkedList:
    def __init__(self):
self.headval = None
# Print the linked list
    def listprint(self):
printval = self.headval
        while printval is not None:
            print (printval.dataval)
printval = printval.nextval
    def AtBegining(self,newdata):
NewNode = Node(newdata)
# Update the new nodes next val to existing node
NewNode.nextval = self.headval
self.headval = NewNode
list = SLinkedList()
list.headval = Node("Mon")
e2 = Node("Tue")
e3 = Node("Wed")
list.headval.nextval = e2
e2.nextval = e3

```



```
list.AtBegining("Sun")
list.listprint()
```

Inserting at the End of the Linked List

This involves pointing the next pointer of the the current last node of the linked list to the new data node. So the current last node of the linked list becomes the second last data node and the new node becomes the last node of the linked list.

```
class Node:
    def __init__(self, dataval=None):
self.dataval = dataval
self.nextval = None
class SLinkedList:
    def __init__(self):
self.headval = None
# Function to add newnode
    def AtEnd(self, newdata):
NewNode = Node(newdata)
    if self.headval is None:
self.headval = NewNode
        return
laste = self.headval
        while(laste.nextval):
laste = laste.nextval
laste.nextval=NewNode
# Print the linked list
    def listprint(self):
printval = self.headval
        while printval is not None:
            print (printval.dataval)
printval = printval.nextval
list = SLinkedList()
list.headval = Node("Mon")
e2 = Node("Tue")
e3 = Node("Wed")
list.headval.nextval = e2
e2.nextval = e3
list.AtEnd("Thu")
list.listprint()
```

Inserting in between two Data Nodes

This involves chaging the pointer of a specific node to point to the new node. That is possible by passing in both the new node and the existing node after which the new node will be inserted. So we define an additional class which will change the next pointer of the new node to

the next pointer of middle node. Then assign the new node to next pointer of the middle node.

```
class Node:
    def __init__(self, dataval=None):
self.dataval = dataval
self.nextval = None
class SLinkedList:
    def __init__(self):
self.headval = None
# Function to add node
    def Inbetween(self,middle_node,newdata):
        if middle_node is None:
print("The mentioned node is absent")
            return
        NewNode = Node(newdata)
        NewNode.nextval = middle_node.nextval
        middle_node.nextval = NewNode
# Print the linked list
    def listprint(self):
printval = self.headval
        while printval is not None:
            print (printval.dataval)
            printval = printval.nextval
list = SLinkedList()
list.headval = Node("Mon")
e2 = Node("Tue")
e3 = Node("Thu")
list.headval.nextval = e2
e2.nextval = e3
list.Inbetween(list.headval.nextval,"Fri")
list.listprint()
```

Removing an Item form a Liked List

We can remove an existing node using the key for that node. In the below program we locate the previous node of the node which is to be deleted. Then point the next pointer of this node to the next node of the node to be deleted.

```
class Node:
    def __init__(self, data=None):
self.data = data
self.next = None
class SLinkedList:
    def __init__(self):
self.head = None
    def Atbegining(self, data_in):
        NewNode = Node(data_in)
        NewNode.next = self.head
        self.head = NewNode
```

```

# Function to remove node
def RemoveNode(self, Removekey):
HeadVal = self.head
    if (HeadVal is not None):
        if (HeadVal.data == Removekey):
self.head = HeadVal.next
HeadVal = None
        return
    while (HeadVal is not None):
        if HeadVal.data == Removekey:
            break
prev = HeadVal
HeadVal = HeadVal.next
    if (HeadVal == None):
        return
prev.next = HeadVal.next
HeadVal = None
    def LListprint(self):
printval = self.head
    while (printval):
        print(printval.data),
printval = printval.next
l1 = SLinkedList()
l1.Atbegining("Mon")
l1.Atbegining("Tue")
l1.Atbegining("Wed")
l1.Atbegining("Thu")
l1.RemoveNode("Tue")
l1.LListprint()

```

Stack:

Stack is a linear data structure which follows a particular order in which the operations are performed. The order may be LIFO(Last In First Out) or FILO(First In Last Out).

Mainly the following three basic operations are performed in the stack:

Push: Adds an item in the stack. If the stack is full, then it is said to be an Overflow condition.

Pop: Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an Underflow condition.

Peek or Top: Returns top element of stack.

isEmpty: Returns true if stack is empty, else false

PUSH into a Stack

```

class Stack:
    def __init__(self):

```

```

self.stack = []
def add(self, dataval):
# Use list append method to add element
    if dataval not in self.stack:
self.stack.append(dataval)
        return True
    else:
        return False
# Use peek to look at the top of the stack
def peek(self):
    return self.stack[-1]
AStack = Stack()
AStack.add("Mon")
AStack.add("Tue")
AStack.peek()
print(AStack.peek())
AStack.add("Wed")
AStack.add("Thu")
print(AStack.peek())

```

POP from a Stack

```

class Stack:
    def __init__(self):
self.stack = []
        def add(self, dataval):
# Use list append method to add element
            if dataval not in self.stack:
self.stack.append(dataval)
                return True
            else:
                return False
# Use list pop method to remove element
def remove(self):
    if len(self.stack) <= 0:
        return ("No element in the Stack")
    else:
        return self.stack.pop()
AStack = Stack()
AStack.add("Mon")
AStack.add("Tue")
AStack.add("Wed")
AStack.add("Thu")
print(AStack.remove())
print(AStack.remove())

```

Queue:

Queue is an abstract data structure, somewhat similar to Stacks. Unlike stacks, a queue is open at both its ends. One end is always used to insert data (enqueue) and the other is used to remove data (dequeue). Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first.

Adding Elements to a Queue

```
class Queue:
    def __init__(self):
self.queue = list()
    def addtoq(self,dataval):
# Insert method to add element
    if dataval not in self.queue:
self.queue.insert(0,dataval)
        return True
    return False
    def size(self):
        return len(self.queue)
TheQueue = Queue()
TheQueue.addtoq("Mon")
TheQueue.addtoq("Tue")
TheQueue.addtoq("Wed")
print(TheQueue.size())
```

Removing Element from a Queue

```
class Queue:
    def __init__(self):
self.queue = list()
    def addtoq(self,dataval):
# Insert method to add element
    if dataval not in self.queue:
self.queue.insert(0,dataval)
        return True
    return False
# Pop method to remove element
    def removefromq(self):
    if len(self.queue)>0:
        return self.queue.pop()
    return ("No elements in Queue!")
TheQueue = Queue()
TheQueue.addtoq("Mon")
TheQueue.addtoq("Tue")
TheQueue.addtoq("Wed")
```

```
print(TheQueue.removefromq())  
print(TheQueue.removefromq())
```