INSTITUTE OF AERONAUTICAL ENGINEERING

(Autonomous)

Dundigal, Hyderabad - 500 043

MECHANICAL ENGINEERING

LECTURE NOTES

Course Title	**PYTHON PROGRAMMING**				
Course Code	ACSC01				
Program	B.Tech				
Semester	I Semester: AERO | MECH | CIVIL | CSE | CSE (AI & ML) | CSE (DS) | CSE (CS) | CSIT | ECE | EEE | IT				
Course Type	Core				
Regulation	UG20				
Course Structure	**Theory**			**Practical**	
	Lectures	**Tutorials**	**Credits**	**Laboratory**	**Credits**
	3	0	3	3	1.5
Course Coordinator	Ms. B Padmaja, Assistant Professor				

COURSE OBJECTIVES:

The students will try to learn:	
I	The fundamentals of core Python programming.
II	The in-depth understanding of branching and loop control structures in Python programming.
III	The use of list, tuple, set and dictionary in problem solving.
IV	Develop the ability to write modular programming using object oriented concepts.
V	Enhance programming skills to choose multiple career paths and engage in lifelong learning.

COURSE OUTCOMES:

After successful completion of the course, students will be able to:

	Course Outcomes	**Knowledge Level (Bloom's Taxonomy)**
CO 1	**Understand** operators, precedence of operators, associativity while evaluating expressions in program statements.	Understand
CO 2	**Visualize** the capabilities of procedural as well as object-oriented programming in Python and demonstrate the same in real world scenario.	Understand
CO 3	**Demonstrate** indexing and slicing mechanisms for extracting a portion of data in a sequence.	Apply
CO 4	**Understand** native data types like list, set, tuple, dictionary use them in data processing applications.	Understand
CO 5	**Compare and contrast** mutable and immutable objects and understand the state change of objects during runtime.	Understand
CO 6	**Understand** passing of parameters and arguments in functions to do modular programming.	Understand
CO 7	**Understand** the concepts of inheritance and polymorphism for code reusability and extensibility.	Understand
CO 8	**Make use of** appropriate modules for solving real-time problems.	Apply
CO 9	**Apply** string handling mechanisms to do automated memory management and reduce out-of-bounds accesses.	Apply
CO 10	**Extend** the knowledge of Python programming to build successful career in software development.	Understand

At the end of the unit students are able to:		
Course Outcomes		Knowledge Level (Bloom's Taxonomy)
CO 1	**Understand** operators, precedence of operators, associativity while evaluating expressions in program statements.	Understand
CO 2	**Visualize** the capabilities of procedural as well as object-oriented programming in Python and demonstrate the same in real world scenario.	Understand

MODULE – I
INTRODUCTION TO PYTHON

Introduction to Python:

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 andPHP.
- **Python is Interactive:** You can actually sit at a Python prompt and interact with the interpreter directly to write yourprograms.
- **Python is Object-Oriented:** Python supports Object-Oriented style or technique of programming that encapsulates code withinobjects.
- **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 togames.

Features of Python

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 languagequickly.
- **Easy-to-read:** Python code is more clearly defined and visible to theeyes.
- **Easy-to-maintain:** Python's source code is fairlyeasy-to-maintain.
- **A broad standard library:** Python's bulk of the library is very portable and cross-platform compatible on UNIX, Windows, andMacintosh.
- **Interactive Mode:** Python has support for an interactive mode which allows interactive testing and debugging of snippets ofcode.
- **Portable:** Python can run on a wide variety of hardware platforms and has the same interface on allplatforms.
- **Extendable:** You can add low-level modules to the Python interpreter. These modules enable programmers to add to or customize their tools to be moreefficient.
- **Databases:** Python provides interfaces to all major commercialdatabases.
- **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 ofUNIX.
- **Scalable:** Python provides a better structure and support for large programs than shell scripting.

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 theNetherlands.
- Python is derived from many other languages, including ABC, Modula-3, C, C++, Algol-68, SmallTalk, Unix shell, and other scriptinglanguages.
- 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 itsprogress.
- 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 versionseries.
- 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 plannedend-of-life.

Need of Python Programming

- **Softwarequality**

  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 functionprogramming.
- **Developerproductivity**

  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 asampling).

- **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 standalonetool.

- **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 fullerdescription.

- **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 tonotice.

- **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 theinterpreter.

- **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 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/Javaintegration.

- **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 searchsystems.
2. The popular YouTube video sharing service is largely written inPython.
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 educationallanguage.
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 scriptingAPI.
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.

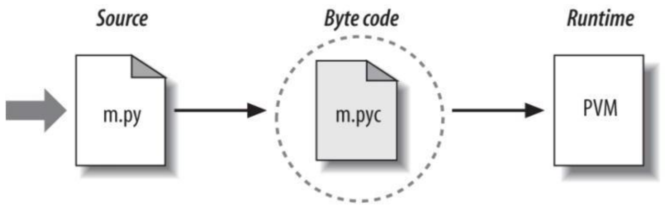
Byte code Compilation:

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 .*pyc*file), it is shipped off for execution to something generally known as the python virtual machine (PVM).



Applications of Python:

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

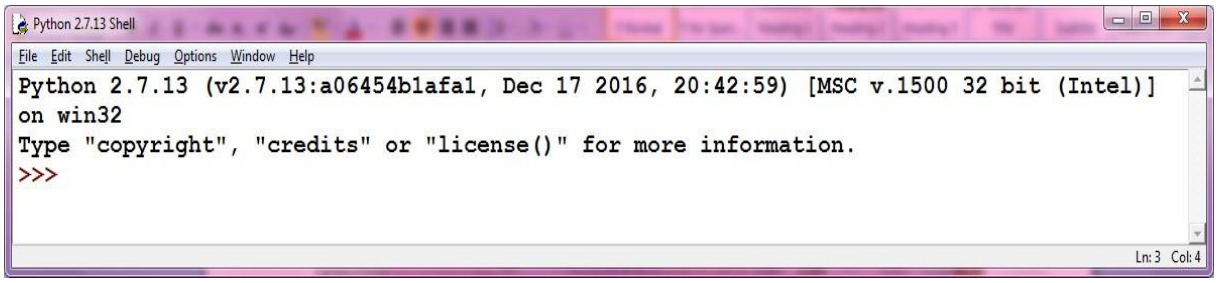
What Are Python's Technical Strengths?

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

Working with Python:

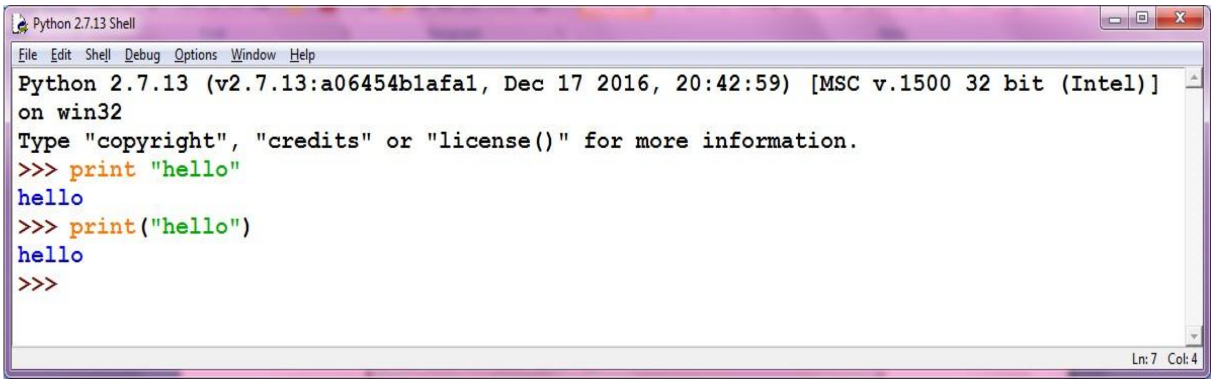
a. Running PythonInterpreter:

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 yourcommands.



```
Python 2.7.13 (v2.7.13:a06454b1afa1, 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.



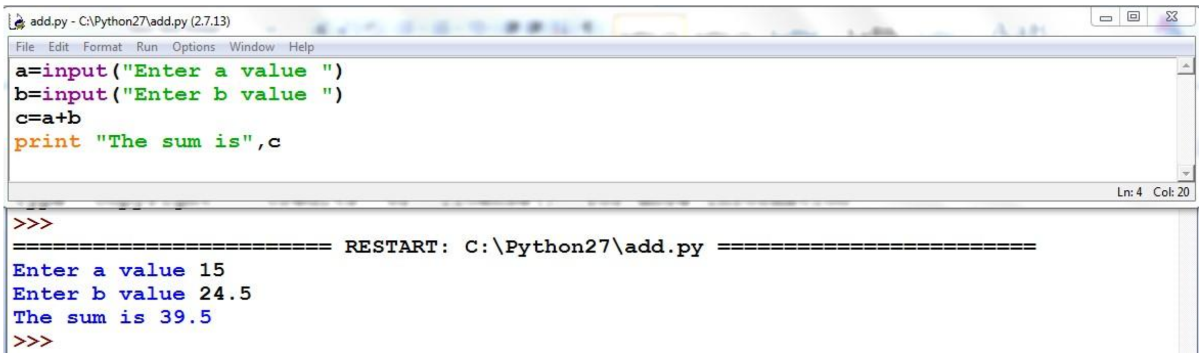
```
Python 2.7.13 (v2.7.13:a06454b1afa1, Dec 17 2016, 20:42:59) [MSC v.1500 32 bit (Intel)]
on win32
Type "copyright", "credits" or "license()" for more information.
>>> print "hello"
hello
>>> print("hello")
hello
>>>
```

b. Running Python Scripts inIDLE:

- Goto**File** 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
Ln: 4 Col: 20

```
a=input("Enter a value ")
b=input("Enter b value ")
c=a+b
print "The sum is",c
```

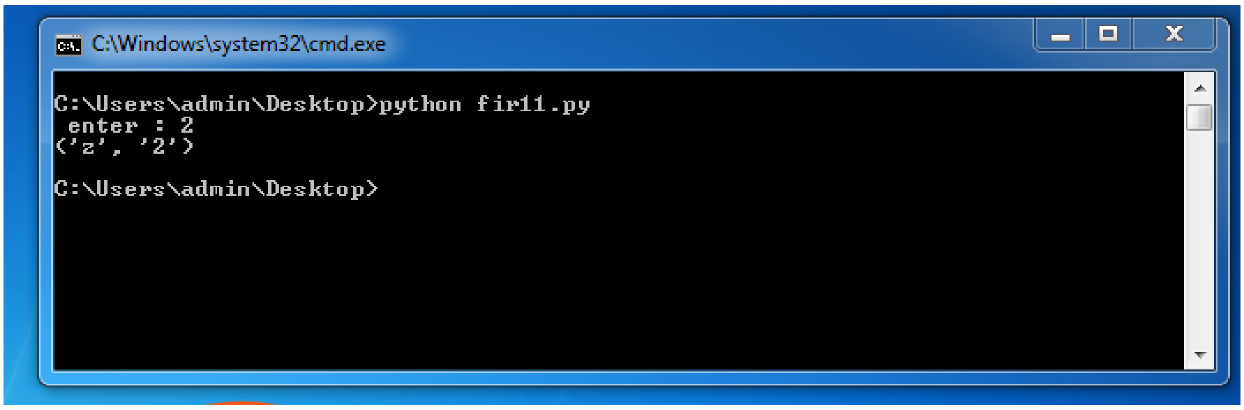
```
>>>
======================== RESTART: C:\Python27\add.py ========================
Enter a value 15
Enter b value 24.5
The sum is 39.5
>>>
```

c. Running python scripts in CommandPrompt:

- Before going to run we have to check the PATH in environmentvariables.
- Open your text editor, type the following text and save it ashello.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 doingit.



```
C:\Users\admin\Desktop>python fir11.py
 enter : 2
('z', '2')

C:\Users\admin\Desktop>
```

Identifiers and Keywords

Variables:

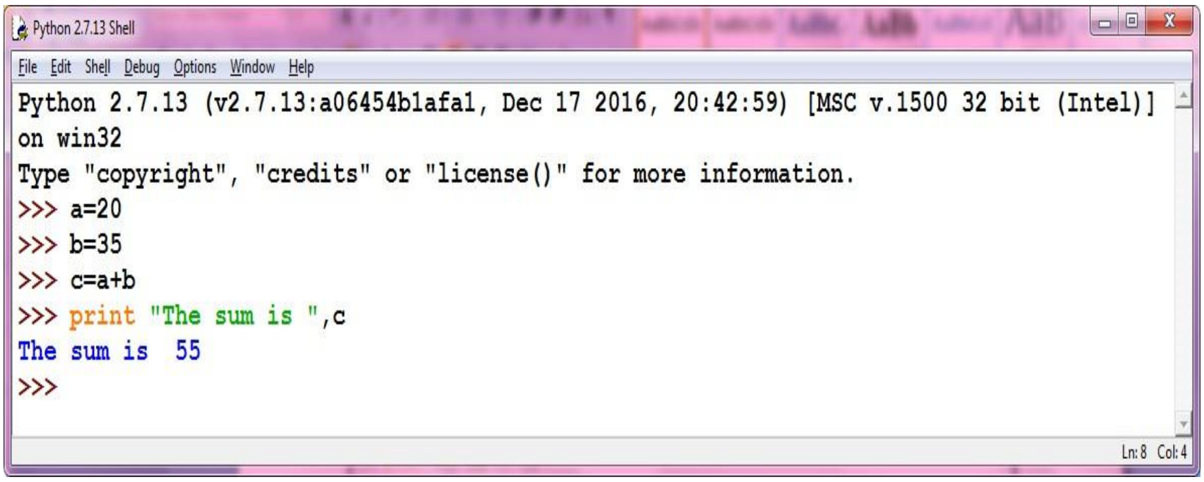
Variables are nothing but reserved memory locations to store values. This means that when you create a variable you reserve some space in memory.

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 tovariables.

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 (v2.7.13:a06454b1afa1, 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 contain lowercase letters only.

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	

COMMENTS

Comments are the non-executable statements explain what the program does. For large programs it often difficult to understand what is does. The comment can be added in the program code with the symbol # and multiple line comment code with symbol ‘’   ‘’ or “”” “”””

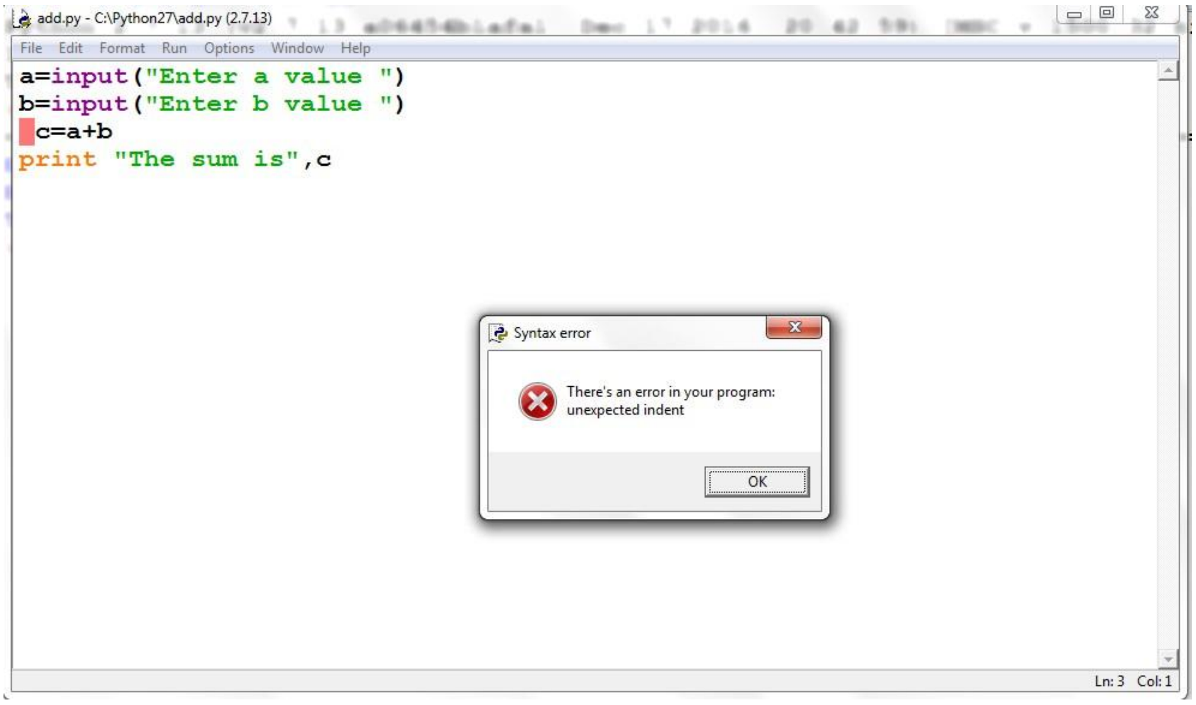
Example:

print ‘Hello, World!’   # print the message Hello, World!; comment
v=5   # creates the variable v and assign the value 5; comment

Indentation and Multi-lining

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 toread.

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 follow indentation rules to indicate the blocks.


add.py - C:\Python27\add.py (2.7.13)
File Edit Format Run Options Window Help
Ln: 3 Col: 1

```
a=input("Enter a value ")
b=input("Enter b value ")
 c=a+b
print "The sum is",c
```

Syntax error

There's an error in your program:
unexpected indent

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 (signedintegers)
- long (long integers, they can also be represented in octal andhexadecimal)
- float (floating point real values)
- complex (complexnumbers)

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 "intis",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 theend.

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

21	lower()	Converts all uppercase letters in string to lowercase.
22	lstrip()	Removes all leading whitespace in string.
23	maketrans()	Returns a translation table to be used in translates 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.

Operators and Expressions

Python language supports Operators and Expressions 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 is2
Modulus is 1
Power value is 8
Floor Division is2
```

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 falseotherwise.	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"
```

Output:

```
Line 1 - a and b have same identity
Line 2 - a and b 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 x b – c	a * b – c
(m + n) (x + y)	(m + n) * (x + y)
(ab / c)	a * b / c
$3x^2$ +2x + 1	3*x*x+2*x+1
(x / y) + c	x / y + c

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:

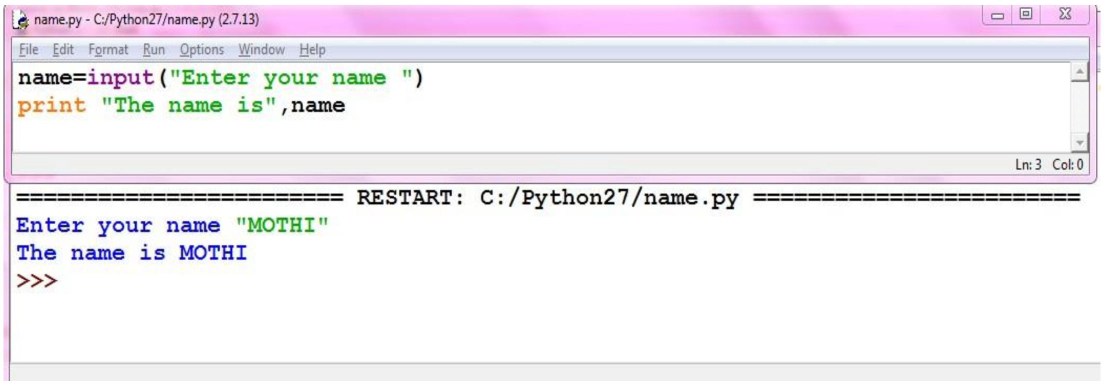
```
x= 254
y=-738
z= 1178
```

Console Input/output

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,


name.py - C:/Python27/name.py (2.7.13)
File Edit Format Run Options Window Help
Ln: 3 Col: 0

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

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

OUTPUT function:

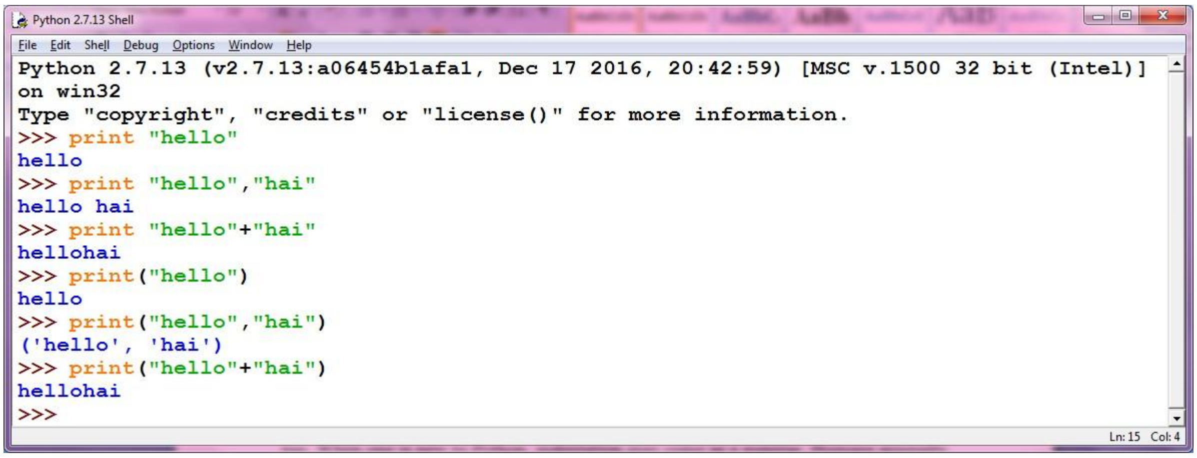
We use the print() function or print keyword to output data to the standard output device (screen). This function prints the object/string written in function.

The actual syntax of the print() function is

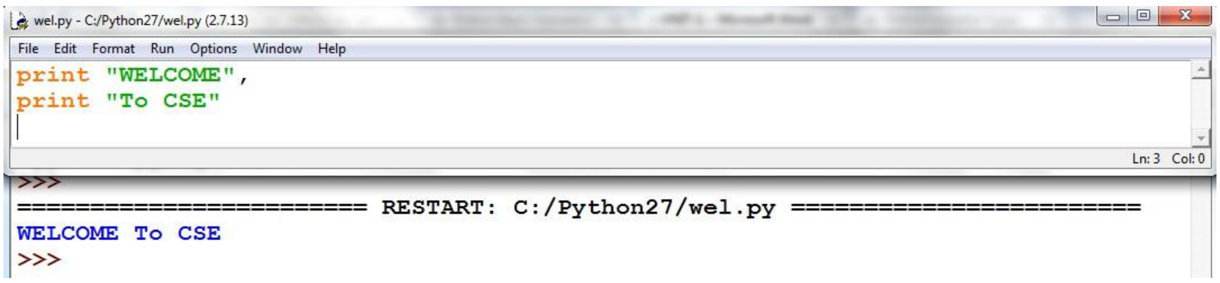
print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)

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 (v2.7.13:a06454b1afa1, 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
Ln: 3 Col: 0

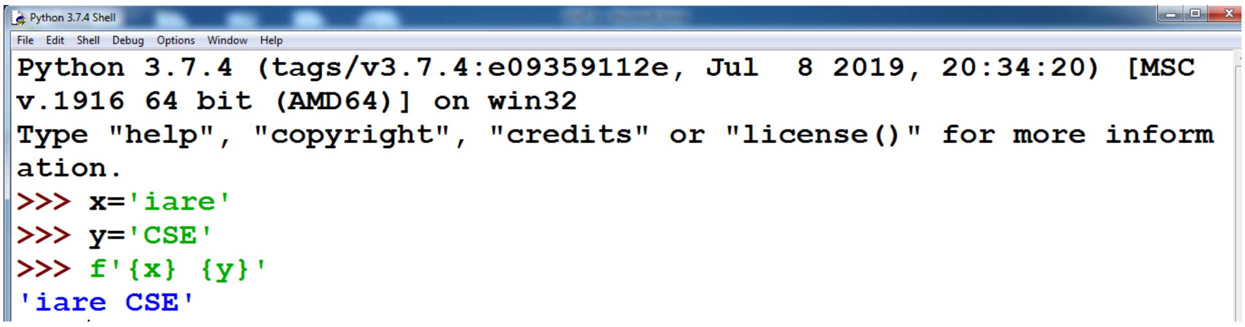
```
print "WELCOME",
print "To CSE"
```

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

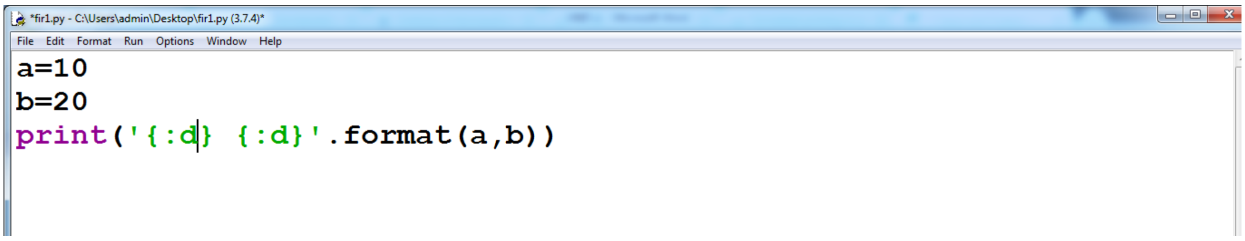
FORMATTING

So far we've encountered two ways of writing values: expression statements and the print() function. Often you'll want more control over the formatting of your output than simply printing space-separated values. There are several ways to format output.

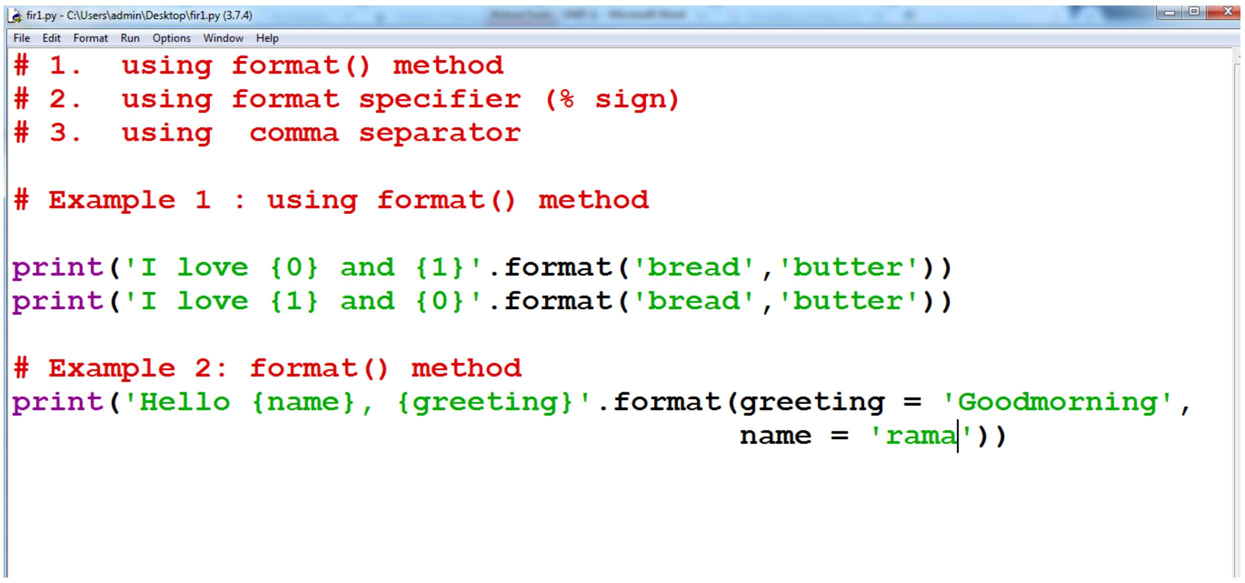
To use formatted string literals, begin a string with f or F before the opening quotation mark or triple quotation mark. Inside this string, you can write a Python expression between { and } characters that can refer to variables or literal values.



```
Python 3.7.4 (tags/v3.7.4:e09359112e, Jul  8 2019, 20:34:20) [MSC
v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more inform
ation.
>>> x='iare'
>>> y='CSE'
>>> f'{x} {y}'
'iare CSE'
```


fir1.py - C:\Users\admin\Desktop\fir1.py (3.7.4)
File Edit Format Run Options Window Help

```
a=10
b=20
print('{:d} {:d}'.format(a,b))
```


fir1.py - C:\Users\admin\Desktop\fir1.py (3.7.4)
File Edit Format Run Options Window Help

```
# 1.  using format() method
# 2.  using format specifier (% sign)
# 3.  using  comma separator

# Example 1 : using format() method

print('I love {0} and {1}'.format('bread','butter'))
print('I love {1} and {0}'.format('bread','butter'))

# Example 2: format() method
print('Hello {name}, {greeting}'.format(greeting = 'Goodmorning',
                                        name = 'rama'))
```

At the end of the unit students are able to:		
Course Outcomes		Knowledge Level (Bloom's Taxonomy)
CO 2	**Visualize** the capabilities of procedural as well as object-oriented programming in Python and demonstrate the same in real world scenario.	Understand
CO 8	**Make use of** appropriate modules for solving real-time problems.	Apply

MODULE – II: DECISION CONTROL STATEMENTS

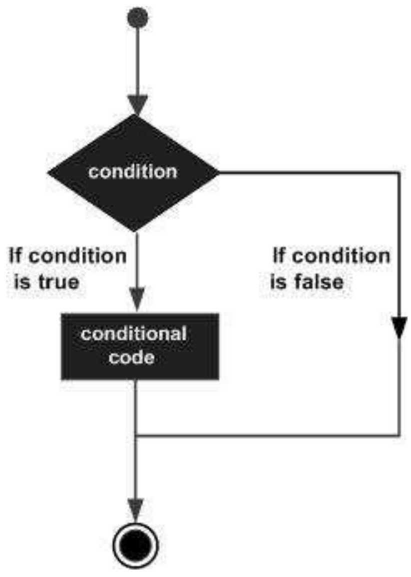
MODULE – II:
DECISION CONTROL STATEMENTS

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:

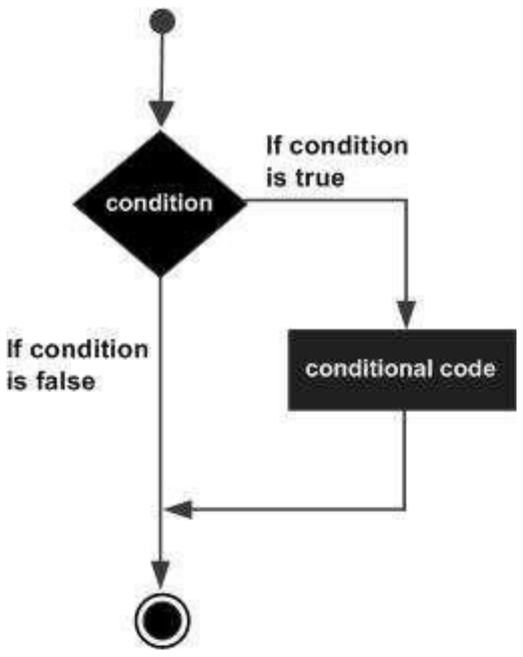


Python programming language assumes any non-zero and non-null values as True, and if it is either zero or null, then it is assumed as Falsevalue.

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
   conditi
```

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
```

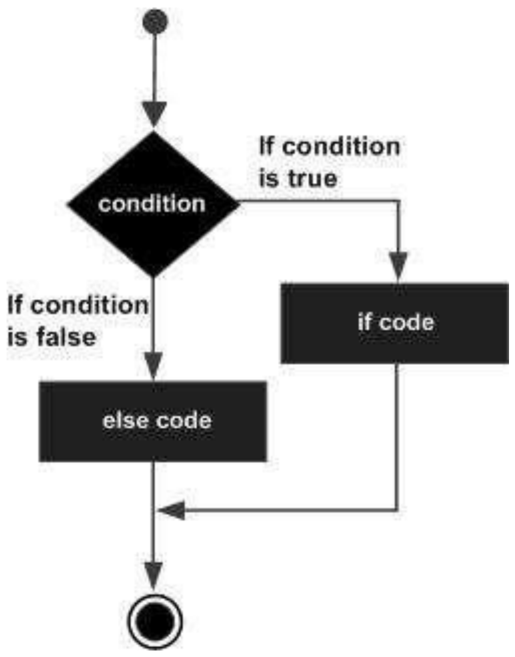
The *if ... else* statement

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 FALSEvalue.

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)
elifcondition2:
  statement(s)
else:
  statement(s)
```

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:

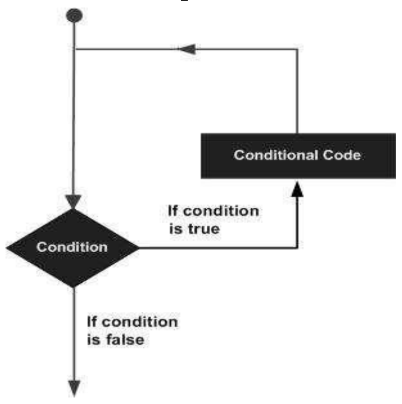
```
c is big
```

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

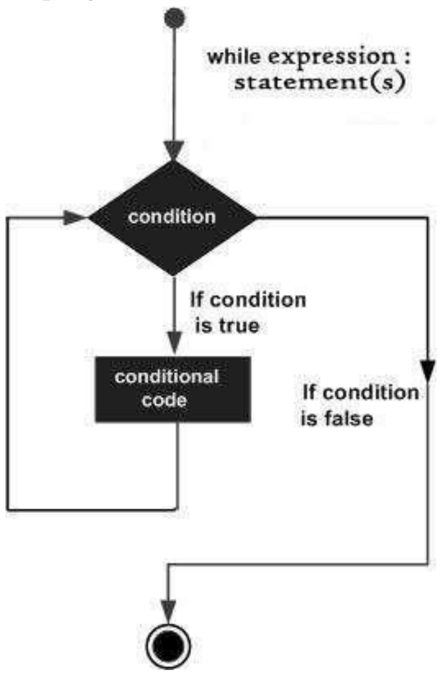
The syntax of a **while** loop in Python programming language is:

```
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 theloop.

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 i
        i+=1
        print "END"
```

Example-2:

```
i=1
while i < 4:
        print i
        i+=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
```

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 i range(1,5):
        print i
        print “END”
```

Output-1:

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

Example-2:

```
for i range(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=1for i in range(1,n+1):

  f=f*i
```

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,
```

```
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
```

Example-2:

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

```
* * * * *
*       *
*       *
*       *
* * * * *
```

```
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:
```

```
* * * * *
* *     *
*   *   *
*     * *
* * * * *
```

Example-4:

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:
```

```
$ * * * *
* $     *
*   $   *
*     $ *
* * * * $
```

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:
```

```
*   *   *
*
*
*
*   *   *
```

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"*",
```

```
*   *   *
*
*   *   *
        *
*   *   *
```

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:
```

```
*   *   *
*
*   *   *
*
*   *   *
```

Example-9:

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

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

```
        *
      *   *
    *   *   *
  *   *   *   *
*   *   *   *   *
```

Example-10:

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

```
1
1  2
1  2  3
1  2  3  4
1  2  3  4  5
```

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"
```

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

Output:

2) WriteaprogramprintFibonacciseriesandsumtheevennumbers.Fibonacciseries

```
n=input("Enter n value ")
f0=1
f1=2
sum=f1
printf0,f1,
for i inrange(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
```

is 1,2,3,5,8,13,21,34,55

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:
```

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 ofnumbers.

```
sum=0
while True:
    n=input("Enter the number: ")
    if n==-1:

        break
    else:
```

Program:

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 followingsequence.

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 DE

Program:

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

Write a program to display the following sequence.

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

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

Program:

8) 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():
```

Output-1:

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

Output-2:

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

9) Write a program to print sum of digits.

Program:

```
n=input("Enter the number: ")

sum=0
while n>0:
    r=n%10

    sum+=r
```

Output:

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

10) 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
```

Output:

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

11) Write a program to take input string from the user and print that string after removingovals.

```
st=input("Enter the
string:") st2=""

for i in st:
    if i not in "aeiouAEIOU":
        st2=st2+i
```

Program:

Output:

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

MODULE – III: CONTAINER DATA TYPES

At the end of the unit students are able to:		
Course Outcomes		**Knowledge Level (Bloom's Taxonomy)**
CO 3	**Demonstrate** indexing and slicing mechanisms for extracting a portion of data in a sequence.	Apply
CO 4	**Understand**native data types like list, set, tuple, dictionary use them in data processing applications.	Understand

MODULE – III:

CONTAINER DATA TYPES

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 samearray.
- 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 inmemory.

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 thebeginning.
- Arrays can grow or shrink in memory dynamically (duringruntime).
- Arrays are useful to handle a collection of elements like a group of numbers orcharacters.
- 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 significantlyimportingthe„array‟classof„array‟module.So,thereisnoneedtomentionthe module name before our array name while creating it. We can create arrayas:

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.

1) Write a program to perform stack operations using array.

Program:

```
import sys

from array import
* a=array('i',[])

while True:
    print "\n1.PUSH 2.POP 3.DISPLAY 4.EXIT"

    ch=input("Enter Your Choice: ")
    if ch==1:

        ele=input("Enter element:
        ") a.append(ele)

        print
    "Inserted"
    elifch==2:

        if len(a)==0:

            print "\t STACK IS EMPTY"
        else:

            print "Deleted element is", a.pop(
    ) elifch==3:

        if len(a)==0:

            print "\t STACK IS EMPTY"
        else:
```

Output:

```
1.PUSH 2.POP 3.DISPLAY4.EXIT
Enter Your Choice: 1
Enter element: 15
Inserted
1.PUSH 2.POP 3.DISPLAY 4.EXIT
Enter Your Choice: 1
Enter element: 18
Inserted
1.PUSH 2.POP 3.DISPLAY 4.EXIT
Enter Your Choice: 3
        The Elements in Stack is 15 18
1.PUSH 2.POP 3.DISPLAY 4.EXIT
Enter Your Choice: 2
Deleted element is 18
```

2) Write a program to perform queue operations using array.
Program:

```
import sys

from array import
* a=array('i',[])

while True:
print "\n1.INSERT 2.DELETE 3.DISPLAY 4.EXIT"
ch=input("Enter Your Choice: ")
if ch==1:

    ele=input("Enter element:
    ") a.append(ele)

elifch==2:

    if len(a)==0:

        print "\t QUEUE IS EMPTY"
    else:

        print "Deleted element is”,
        a[0] a.remove(a[0])

elifch==3:

    if len(a)==0:

        print "\t QUEUE IS EMPTY"
    else:
```

Output:

```
1.INSERT 2.DELETE 3.DISPLAY 4.EXIT
Enter Your Choice: 1
Enter element: 12
1.INSERT 2.DELETE 3.DISPLAY 4.EXIT
Enter Your Choice: 1
Enter element: 13
1.INSERT 2.DELETE 3.DISPLAY 4.EXIT
Enter Your Choice: 1
Enter element: 14
1.INSERT 2.DELETE 3.DISPLAY 4.EXIT
Enter Your Choice: 3
        The Elements in Queue is 12 13 14
1.INSERT 2.DELETE 3.DISPLAY 4.EXIT
```

```
Enter Your Choice: 2
Deleted element is 12
```

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.

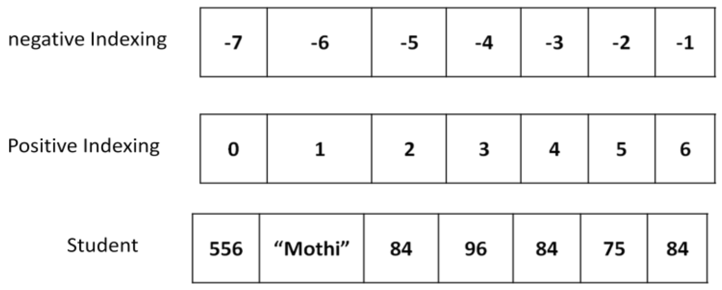
student = [556, “Mothi”, 84, 96, 84, 75, 84]

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.



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 10in 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 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]
dellst[3]                # delete 3rd element from the list i.e., 8
printlst                 # [5,7,1,9,6]
```

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

Concatenation of Two lists:

Wecansimplyuse„+‟operatorontwoliststojointhem.Forexample„„x‟and„y‟are two lists. If we wrte 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:

Wecanrepeattheelementsofalist„n‟numberoftimesusing„*‟operator.

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

Membership in Lists:

Wecancheckifanelementisamemberofalistbyusing„in‟and„notin‟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:

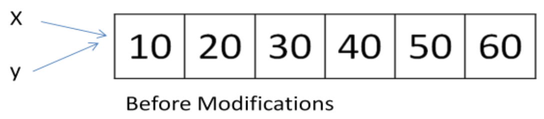
```
x=[10,20,30,45,55,65] a=20
print ain x              # True
a=25
print ain x              # False
a=45
print a not in x # False a=40
print a not in x # True
```

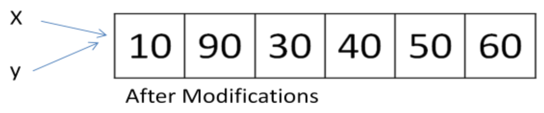
Aliasing and Cloning Lists:

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



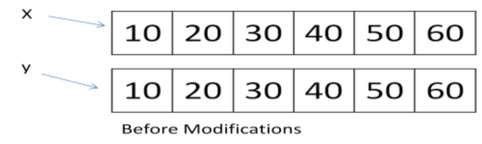


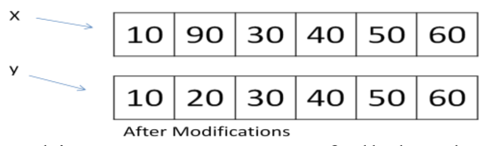
In this case we are having only one list of elements but with two different names „x‟ and „y‟. Here, „x‟ is the original name and „y‟ is the alias name for the same list. Hence, any modifications done to x‟ will also modify „y‟ and vice versa.

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





When we clone a list like this, a separate copy of all the elements is stored into „y‟. Thelists„x‟and„y‟areindependentlists.Hence,anymodificationsto„x‟willnotaffect„y‟ and viceversa.

Methods in Lists:

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

First of all, we should convert the lists into 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]                # display65
printb[2]                # display [ 10, 20, 30 ]
printb[2][0]             # display10
print b[2][1] # display 20 print b[2][2]
# display 30 for x in b[2]:
                    print x,        # display 10 2030
```

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):
```

```
[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):
        c[i][j]= a[i][j]+b[i][j]
for i in range(0,m1):
    for j in range(0,n1):
        print "\t",c[i][j],
    print ""
```

```
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 codeas: 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 rewritten in a compact way as:

```
squares=[x**2 for x inrange(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 fro 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) ifx%2==0]
```

It will display the list even squares aslist.

```
[ 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 ondemand.

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 assign ordinary data type.

```
tup = (10)                                   tup = (10,)
print tup           # display 10             print tup           # display 10
print type(tup)     # display <type „int‟>   print type(tup)     # display<type„tuple‟>
```

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:

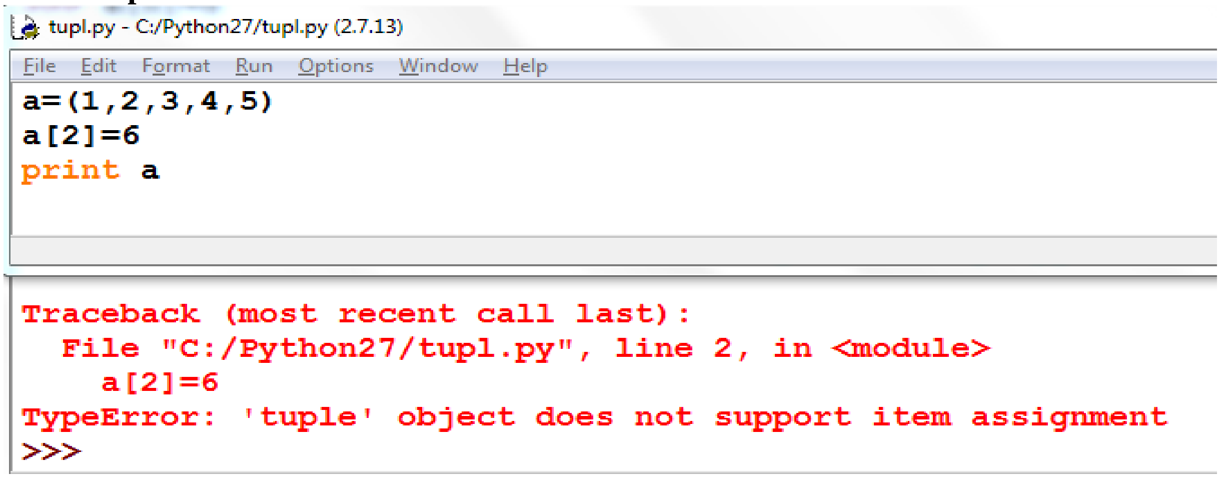
Accessing the elements from a tuple can be done using indexing or slicing. This is same as that of a list. Indexing represents the position number of the element in the tuple. The position starts from 0.

```
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:

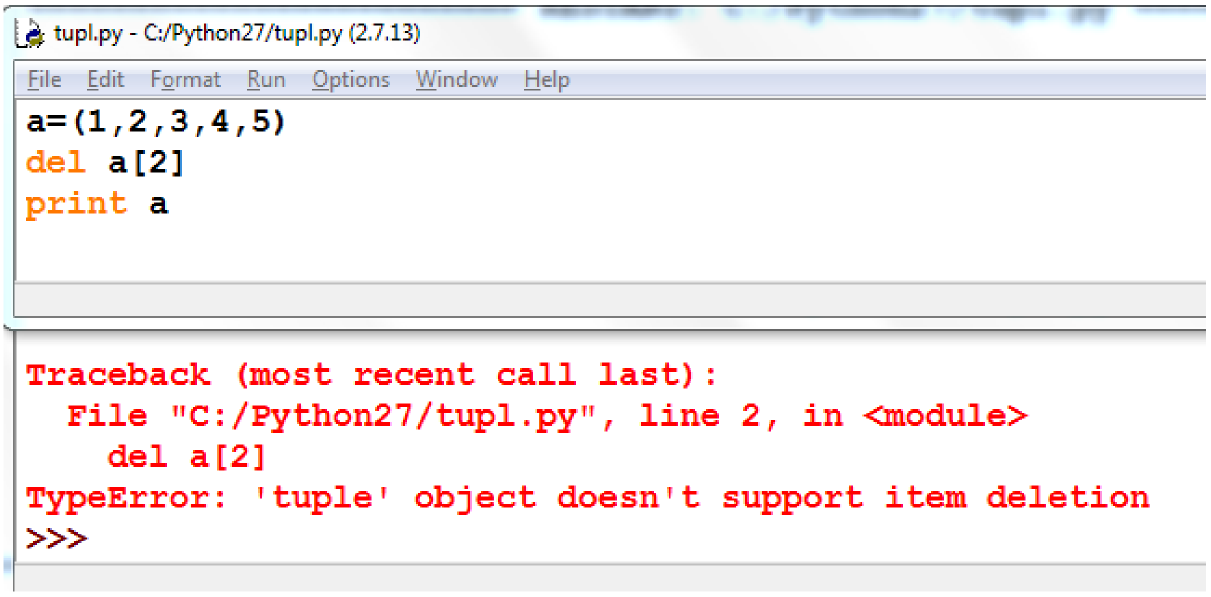
Tuples are immutable which means you cannot update, change or delete the values of tuple elements.

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
>>>
```

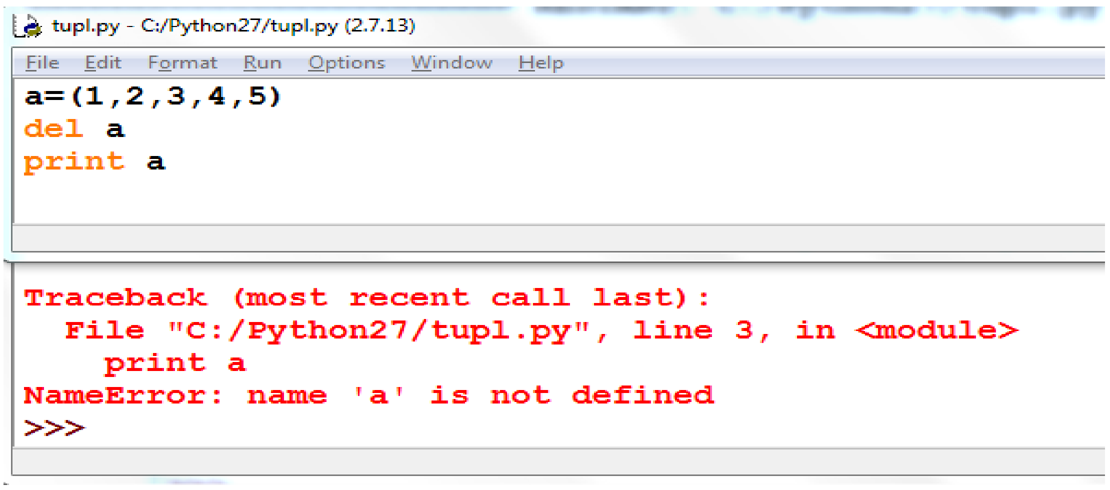

tupl.py - C:/Python27/tupl.py (2.7.13)
File Edit Format Run Options Window Help

```
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
>>>
```

Example-2:

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 andcompare.
- If either element is a number, then the other element is "larger" (numbers are "smallest").
- Otherwise, types are sorted alphabetically byname.

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')                #display-1

print cmp(tuple1, tuple2)           #display-1
```

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 fromit.

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, …….}
```

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

Example:

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” ] )
```

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	len(s)	number of elements in set *s* (cardinality)
2	x in s	test *x* for membership in *s*
3	x not in s	test *x* for non-membership in *s*
4	s.issubset(t) (or) s <= t	test whether every element in *s* is in *t*
5	s.issuperset(t) (or) s >= t	test whether every element in *t* is in *s*
6	s = = t	Returns True if two sets are equivalent and returns False.
7	s ! = t	Returns True if two sets are not equivalent and returns False.
8	s.union(t) (or) s|t	new set with elements from both *s* and *t*

9	s.intersection(t) (or) s & t	new set with elements common to *s* and *t*

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

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.

```
s=set()                                          s={}
printtype(s)    # display<type„set‟>             print type(s) # display <type„dict‟>
```

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
```

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* statementas:

```
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 orFalse.

```
„Name‟ind              #checkif„Name‟isakeyindandreturnsTrue/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 therules:

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

```
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':2
```

Dictionary Methods:

Method	Description
d.clear()	Removes all key-value pairs from dictionary„d‟.
d2=d.copy()	Copies all elements from„d‟ into a new dictionary d2.
d.fromkeys(s [,v])	Create a new dictionary with keys from sequence„s‟ and values all set to „v‟.
d.get(k [,v])	Returns the value associated with key „k‟. If key is not found, it returns „v‟.
d.items()	Returns an object that contains key-value pairs of„d‟. The pairs are stored as tuples in the object.
d.keys()	Returns a sequence of keys from the dictionary„d‟.
d.values()	Returns a sequence of values from the dictionary„d‟.
d.update(x)	Adds all elements from dictionary „x‟ to„d‟.
d.pop(k [,v])	Removesthekey„k‟anditsvaluefrom„d‟andreturnsthe value.Ifkeyisnotfound,thenthevalue„v‟isreturned.If keyisnotfoundand„v‟isnotmentionedthen„KeyError‟ is raised.
d.setdefault(k [,v])	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 asimple dictionary that contains color code and its nameas:

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 incolors:
    print k # displays only keys for k
incolors:
            print colors[k] # keys to 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)
```

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():
```

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:
```

```
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
```

MODULE – IV: STRINGS AND FUNCTIONS

At the end of the unit students are able to:		
Course Outcomes		Knowledge Level (Bloom's Taxonomy)
CO 5	**Compare and contrast** mutable and immutable objects and understand the state change of objects during runtime.	Understand
CO 6	**Understand** passing of parameters and arguments in functions to do modular programming.	Understand

MODULE – IV

What is String in Python?

A string is a sequence of characters.

A character is simply a symbol. For example, the English language has 26 characters.

Computers do not deal with characters, they deal with numbers (binary). Even though you may see characters on your screen, internally it is stored and manipulated as a combination of 0s and 1s.

This conversion of character to a number is called encoding, and the reverse process is decoding. ASCII and Unicode are some of the popular encodings used.

In Python, a string is a sequence of Unicode characters. Unicode was introduced to include every character in all languages and bring uniformity in encoding. You can learn about Unicode from Python Unicode.

How to create a string in Python?
Strings can be created by enclosing characters inside a single quote or double-quotes. Even triple quotes can be used in Python but generally used to represent multiline strings and docstrings.

```
# defining strings in Python
# all of the following are equivalent
my_string = 'Hello'
print(my_string)

my_string = "Hello"
print(my_string)

my_string = '''Hello'''
print(my_string)

# triple quotes string can extend multiple lines
my_string = """Hello, welcome to
           the world of Python"""
print(my_string)
```

When you run the program, the output will be:

```
Hello
Hello
Hello
Hello, welcome to the world of Python
```

How to access characters in a string?
We can access individual characters using indexing and a range of characters using slicing. Index starts from 0. Trying to access a character out of index range will raise an IndexError.

The index must be an integer. We can't use floats or other types, this will result into TypeError.

Python allows negative indexing for its sequences.

The index of -1 refers to the last item, -2 to the second last item and so on. We can access a range of items in a string by using the slicing operator :(colon).

```
#Accessing string characters in Python
str = 'programiz'
print('str = ', str)

#first character
print('str[0] = ', str[0])

#last character
print('str[-1] = ', str[-1])

#slicing 2nd to 5th character
print('str[1:5] = ', str[1:5])

#slicing 6th to 2nd last character
print('str[5:-2] = ', str[5:-2])
```

When we run the above program, we get the following output:

```
str =  programiz
str[0] =  p
str[-1] =  z
str[1:5] =  rogr
str[5:-2] =  am
```

If we try to access an index out of the range or use numbers other than an integer, we will get errors.

```
# index must be in range
>>> my_string[15]
...
IndexError: string index out of range

# index must be an integer
>>> my_string[1.5]
...
TypeError: string indices must be integers
```

Slicing can be best visualized by considering the index to be between the elements as shown below.

If we want to access a range, we need the index that will slice the portion from the string.

Element Slicing in Python
String Slicing in Python
How to change or delete a string?

Strings are immutable. This means that elements of a string cannot be changed once they have been assigned. We can simply reassign different strings to the same name.

```
>>> my_string = 'programiz'
>>> my_string[5] = 'a'
...
TypeError: 'str' object does not support item assignment
>>> my_string = 'Python'
>>> my_string
'Python'
```

We cannot delete or remove characters from a string. But deleting the string entirely is possible using the del keyword.

```
>>> del my_string[1]
...
TypeError: 'str' object doesn't support item deletion
>>> del my_string
>>> my_string
...
NameError: name 'my_string' is not defined
```

Python String Operations

There are many operations that can be performed with strings which makes it one of the most used data types in Python.

To learn more about the data types available in Python visit: Python Data Types

Concatenation of Two or More Strings

Joining of two or more strings into a single one is called concatenation.

The + operator does this in Python. Simply writing two string literals together also concatenates them.

The * operator can be used to repeat the string for a given number of times.

```
# Python String Operations
str1 = 'Hello'
str2 ='World!'

# using +
print('str1 + str2 = ', str1 + str2)

# using *
print('str1 * 3 =', str1 * 3)
```

When we run the above program, we get the following output:

```
str1 + str2 =  Hello World!
str1 * 3 = HelloHelloHello
```

Writing two string literals together also concatenates them like + operator.

If we want to concatenate strings in different lines, we can use parentheses.

```
>>> # two string literals together
>>> 'Hello "World!'
'Hello World!'

>>> # using parentheses
>>> s = ('Hello '
...      'World')
>>> s
'Hello World'
```

Iterating Through a string

We can iterate through a string using a for loop. Here is an example to count the number of 'l's in a string.

```
# Iterating through a string
count = 0
for letter in 'Hello World':
    if(letter == 'l'):
        count += 1
print(count,'letters found')
```

When we run the above program, we get the following output:

```
3 letters found
```

String Membership Test

We can test if a substring exists within a string or not, using the keyword in.

```
>>> 'a' in 'program'
True
>>> 'at' not in 'battle'
False
```

Built-in functions to Work with Python

Various built-in functions that work with sequence work with strings as well.

Some of the commonly used ones are enumerate() and len(). The enumerate() function returns an enumerate object. It contains the index and value of all the items in the string as pairs. This can be useful for iteration.

Similarly, len() returns the length (number of characters) of the string.

```
str = 'cold'

# enumerate()
list_enumerate = list(enumerate(str))
print('list(enumerate(str) = ', list_enumerate)

#character count
print('len(str) = ', len(str))
```

When we run the above program, we get the following output:

```
list(enumerate(str) =  [(0, 'c'), (1, 'o'), (2, 'l'), (3, 'd')]
len(str) =  4
```

Python String Formatting

Escape Sequence

If we want to print a text like He said, "What's there?", we can neither use single quotes nor double quotes. This will result in a SyntaxError as the text itself contains both single and double quotes.

```
>>> print("He said, "What's there?"")
...
SyntaxError: invalid syntax
>>> print('He said, "What's there?"')
...
SyntaxError: invalid syntax
```

One way to get around this problem is to use triple quotes. Alternatively, we can use escape sequences.

An escape sequence starts with a backslash and is interpreted differently. If we use a single quote to represent a string, all the single quotes inside the string must be escaped. Similar is the case with double quotes. Here is how it can be done to represent the above text.

```
# using triple quotes
print('''He said, "What's there?"''')

# escaping single quotes
print('He said, "What\'s there?"')

# escaping double quotes
print("He said, \"What's there?\"")
```

When we run the above program, we get the following output:

```
He said, "What's there?"
He said, "What's there?"
He said, "What's there?"
```

Here is a list of all the escape sequences supported by Python.

Escape Sequence	Description
\newline	Backslash and newline ignored
\\	Backslash
\'	Single quote
\"	Double quote
\a	ASCII Bell
\b	ASCII Backspace
\f	ASCII Formfeed
\n	ASCII Linefeed
\r	ASCII Carriage Return
\t	ASCII Horizontal Tab
\v	ASCII Vertical Tab
\ooo	Character with octal value ooo
\xHH	Character with hexadecimal value HH

Here are some examples

```
>>> print("C:\\Python32\\Lib")
C:\Python32\Lib
```

```
>>> print("This is printed\nin two lines")
This is printed
in two lines

>>> print("This is \x48\x45\x58 representation")
This is HEX representation
```

Raw String to ignore escape sequence

Sometimes we may wish to ignore the escape sequences inside a string. To do this we can place r or R in front of the string. This will imply that it is a raw string and any escape sequence inside it will be ignored.

```
>>> print("This is \x61 \ngood example")
This is a
good example
>>> print(r"This is \x61 \ngood example")
This is \x61 \ngood example
```

The format() Method for Formatting Strings

The format() method that is available with the string object is very versatile and powerful in formatting strings. Format strings contain curly braces {} as placeholders or replacement fields which get replaced.

We can use positional arguments or keyword arguments to specify the order.

```
# Python string format() method

# default(implicit) order
default_order = "{}, {} and {}".format('John','Bill','Sean')
print('\n--- Default Order ---')
print(default_order)

# order using positional argument
positional_order = "{1}, {0} and {2}".format('John','Bill','Sean')
print('\n--- Positional Order ---')
print(positional_order)

# order using keyword argument
keyword_order = "{s}, {b} and {j}".format(j='John',b='Bill',s='Sean')
print('\n--- Keyword Order ---')
print(keyword_order)
```

When we run the above program, we get the following output:

```
--- Default Order ---
John, Bill and Sean

--- Positional Order ---
Bill, John and Sean

--- Keyword Order ---
Sean, Bill and John
```

The format() method can have optional format specifications. They are separated from the field name using colon. For example, we can left-justify <, right-justify > or center ^ a string in the given space.

We can also format integers as binary, hexadecimal, etc. and floats can be rounded or displayed in the exponent format. There are tons of formatting you can use. Visit here for all the string formatting available with the format() method.

```
>>> # formatting integers
>>> "Binary representation of {0} is {0:b}".format(12)
'Binary representation of 12 is 1100'

>>> # formatting floats
>>> "Exponent representation: {0:e}".format(1566.345)
'Exponent representation: 1.566345e+03'

>>> # round off
>>> "One third is: {0:.3f}".format(1/3)
'One third is: 0.333'

>>> # string alignment
>>> "|{:<10}|{:^10}|{:>10}|".format('butter','bread','ham')
'|butter    |  bread   |       ham|'
```

Old style formatting

We can even format strings like the old sprintf() style used in C programming language. We use the % operator to accomplish this.

```
>>> x = 12.3456789
>>> print('The value of x is %3.2f' %x)
The value of x is 12.35
>>> print('The value of x is %3.4f' %x)
The value of x is 12.3457
```

Common Python String Methods

There are numerous methods available with the string object. The format() method that we mentioned above is one of them. Some of the commonly used methods are lower(), upper(), join(), split(), find(), replace() etc. Here is a complete list of all the built-in methods to work with strings in Python.

```
>>> "PrOgRaMiZ".lower()
'programiz'
>>> "PrOgRaMiZ".upper()
'PROGRAMIZ'
>>> "This will split all words into a list".split()
['This', 'will', 'split', 'all', 'words', 'into', 'a', 'list']
>>> ' '.join(['This', 'will', 'join', 'all', 'words', 'into', 'a', 'string'])
'This will join all words into a string'
>>> 'Happy New Year'.find('ew')
7
>>> 'Happy New Year'.replace('Happy','Brilliant')
'Brilliant New Year'STRINGS AND FUNCTIONS
```

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 reusablecode.
- 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 thesoftware.
- The use of functions in a program will reduce the length of theprogram.

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-definedfunctions.*

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 inPython.

- 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 theseparentheses.
- 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 isindented.
- 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:

```
deffunctionname (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
```

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
```

Returning multiple values from a function:

A function can returns 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 bythe function as a tuple. To grab these values, we can three variables at the time of calling the functionas:

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,ex,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 avariable.
- It is possible to define one function inside anotherfunction.
- It is possible to pass a function as parameter to anotherfunction.
- It is possible that a function can return another function.

To understand these points, we will take a few simpleprograms.

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‟isthenameortaggiven 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.

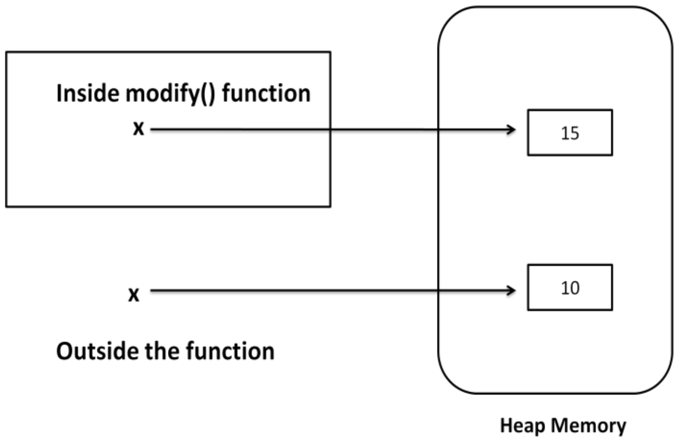


Fig. Passing Integer to a Function

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.

```
defmodify(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.

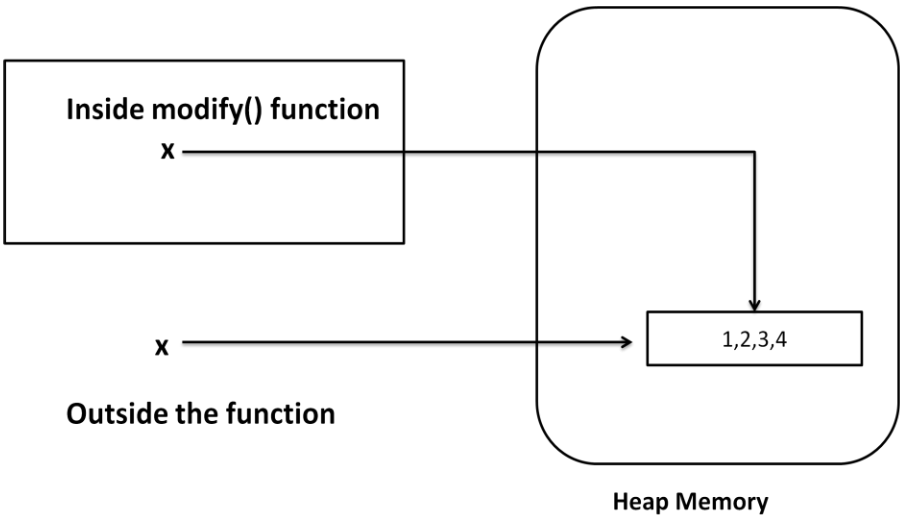


Fig. Passing a list to the function

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„actualarguments‟.Inthefollowingcode„„a‟and„b‟areformalargumentsand„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 actualarguments
```

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

a) Positionalarguments
b) Keywordarguments
c) Defaultarguments
d) Variable lengtharguments

a) PositionalArguments:

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
    prints3
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 anerror.

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,wearementioningakeyword„item‟anditsvalueandthenanotherkeyword„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 argumentsas:

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 defaultvalue 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:

glob

al

aExample-

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 resultas:

```
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: factorial(n) = n *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 is1
factorial of  2 is2
factorial of  3 is6
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 multipleexpressions.
- 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 globalnamespace.
- 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 performancereasons.

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 functionreturns 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.

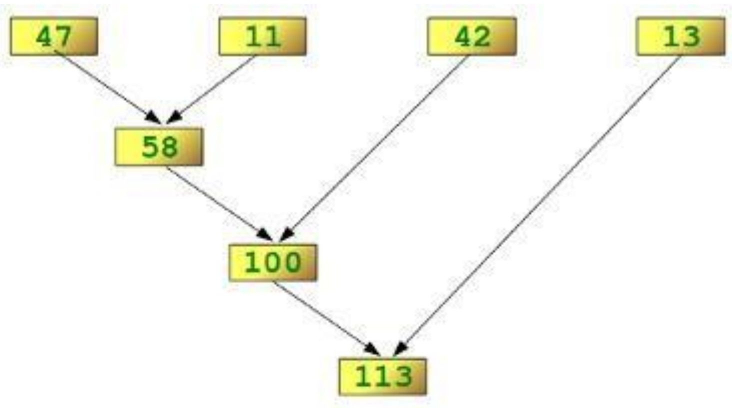
If seq = [s_1, s_2, s_3, ... , s_n], calling reduce(func, seq) works like this:

- At first the first two elements of seq will be applied to func, i.e. func(s_1,s_2) The list on which reduce() works looks now like this: [func(s_1, s_2), s_3, ... , 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(s_1, s_2),s_3). The list looks like this now: [func(func(s_1, s_2),s_3), ... , 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 afunction.

The following steps are generally involved in creation of decorators:

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

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 functiondefinition.

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:
           yieldi
           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 fibo.py in the current directory with the followingcontents:

```
# 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 fibodirectly in the current symbol table; it only enters the module name fibothere. 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 UnboundLocalErroris the result. Uncommenting the global statement fixes the problem.

Packages in Python

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

Third Party Packages:

The Python has got the greatest community for creating great python packages. There are more tha 1,00,000 Packages available at https://pypi.python.org/pypi.

Python Package is a collection of all modules connected properly into one form and distributed PyPI, the Python Package Index maintains the list of Python packages available. Now when you are done with pip setup Go to command prompt / terminal and say

pip install <package_name>

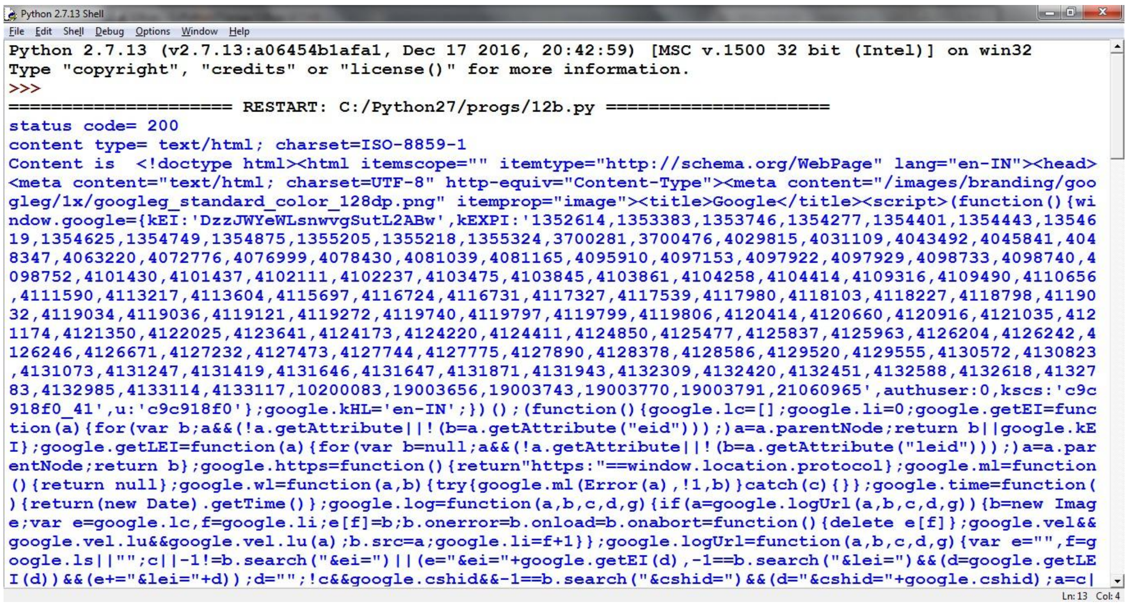
Note: In windows, pip file is in “Python27\Scripts” folder. To install package you have goto the path C:\Python27\Scripts in command prompt and install.

The requests and flask Packages are downloaded from internet. To download install the packages follow the commands

- **Installation of requestsPackage:**
  - **Command:** cd C:\Python27\Scripts
  - **Command:** pip install requests
- **Installation of flaskPackage:**
  - **Command:** cd C:\Python27\Scripts
  - **Command:** pip install flask

Example: Write a script that imports requests and fetch content from the page.

```
import requests
r = requests.get('https://www.google.com/')
print r.status_code
print r.headers['content-type']
print r.text
```



```
Python 2.7.13 (v2.7.13:a06454b1afa1, Dec 17 2016, 20:42:59) [MSC v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===================== RESTART: C:/Python27/progs/12b.py =====================
status code= 200
content type= text/html; charset=ISO-8859-1
Content is  <!doctype html><html itemscope="" itemtype="http://schema.org/WebPage" lang="en-IN"><head>
<meta content="text/html; charset=UTF-8" http-equiv="Content-Type"><meta content="/images/branding/goo
gleg/1x/googleg_standard_color_128dp.png" itemprop="image"><title>Google</title><script>(function(){wi
ndow.google={kEI:'DzzJWYeWLsnwvgSutL2ABw',kEXPI:'1352614,1353383,1353746,1354277,1354401,1354443,13546
19,1354625,1354749,1354875,1355205,1355218,1355324,3700281,3700476,4029815,4031109,4043492,4045841,404
8347,4063220,4072776,4076999,4078430,4081039,4081165,4095910,4097153,4097922,4097929,4098733,4098740,4
098752,4101430,4101437,4102111,4102237,4103475,4103845,4103861,4104258,4104414,4109316,4109490,4110656
,4111590,4113217,4113604,4115697,4116724,4116731,4117327,4117539,4117980,4118103,4118227,4118798,41190
32,4119034,4119036,4119121,4119272,4119740,4119797,4119799,4119806,4120414,4120660,4120916,4121035,412
1174,4121350,4122025,4123641,4124173,4124220,4124411,4124850,4125477,4125837,4125963,4126204,4126242,4
126246,4126671,4127232,4127473,4127744,4127775,4127890,4128378,4128586,4129520,4129555,4130572,4130823
,4131073,4131247,4131419,4131646,4131647,4131871,4131943,4132309,4132420,4132451,4132588,4132618,41327
83,4132985,4133114,4133117,10200083,19003656,19003743,19003770,19003791,21060965',authuser:0,kscs:'c9c
918f0_41',u:'c9c918f0'};google.kHL='en-IN';})();(function(){google.lc=[];google.li=0;google.getEI=func
tion(a){for(var b;a&&(!a.getAttribute||!(b=a.getAttribute("eid")));)a=a.parentNode;return b||google.kE
I};google.getLEI=function(a){for(var b=null;a&&(!a.getAttribute||!(b=a.getAttribute("leid")));)a=a.par
entNode;return b};google.https=function(){return"https:"==window.location.protocol};google.ml=function
(){return null};google.wl=function(a,b){try{google.ml(Error(a),!1,b)}catch(c){}};google.time=function(
){return(new Date).getTime()};google.log=function(a,b,c,d,g){if(a=google.logUrl(a,b,c,d,g)){b=new Imag
e;var e=google.lc,f=google.li;e[f]=b;b.onerror=b.onload=b.onabort=function(){delete e[f]};google.vel&&
google.vel.lu&&google.vel.lu(a);b.src=a;google.li=f+1}};google.logUrl=function(a,b,c,d,g){var e="",f=g
oogle.ls||"";c||-1!=b.search("&ei=")||(e="&ei="+google.getEI(d),-1==b.search("&lei=")&&(d=google.getLE
I(d))&&(e+="&lei="+d));d="";!c&&google.cshid&&-1==b.search("&cshid=")&&(d="&cshid="+google.cshid);a=c|
```

There are some libraries in python:

- **Requests**: The most famous HTTP Library. It is a must and an essential criterion for every PythonDeveloper.
- **Scrapy**: If you are involved in webscripting then this is a must have library for you. After using this library you won‟t use anyother.
- **Pillow**: A friendly fork of PIL (Python Imaging Library). It is more user-friendly than PIL and is a must have for anyone who works withimages.
- **SQLAchemy**: It is a databaselibrary.
- **BeautifulSoup**: This xml and html parsinglibrary.
- **Twisted**: The most important tool for any network application developer.
- **NumPy**: It provides some advanced math functionalities topython.
- **SciPy**: It is a library of algorithms and mathematical tools for python and has caused many scientists to switch from ruby topython.
- **Matplotlib**: It is a numerical plotting library. It is very useful for any data scientist or any data analyzer.

At the end of the unit students are able to:		
Course Outcomes		**Knowledge Level (Bloom's Taxonomy)**
CO 2	**Visualize** the capabilities of procedural as well as object-oriented programming in Python and demonstrate the same in real world scenario.	Understand
CO 7	**Understand** the concepts of inheritance and polymorphism for code reusability and extensibility.	Understand
CO 8	**Make use of** appropriate modules for solving real-time problems.	Apply
CO 9	**Apply** string handling mechanisms to do automated memory management and reduce out-of-bounds accesses.	Apply
CO 10	**Extend** the knowledge of Python programming to build successful career in software development.	Understand

MODULE – V: CLASSES AND OBJECTS

MODULE – V

CLASSES AND OBJECTS

Python has been an object-oriented language since it existed. Because of this, creating and using classes and objects are downright easy. This chapter helps you become an expert in using Python's object-oriented programming support.

If you do not have any previous experience with object-oriented (OO) programming, you may want to consult an introductory course on it or at least a tutorial of some sort so that you have a grasp of the basic concepts.

However, here is small introduction of Object-Oriented Programming (OOP) to bring you at speed:

Overview of OOP Terminology

- **Class:** A user-defined prototype for an object that defines a set of attributes that characterize any object of the class. The attributes are data members (class variables and instance variables) and methods, accessed via dotnotation.
- **Class variable:** A variable that is shared by all instances of a class. Class variables are defined within a class but outside any of the class's methods. Class variables are not used as frequently as instance variablesare.
- **Data member:** A class variable or instance variable that holds data associated with a class and itsobjects.
- **Function overloading:** The assignment of more than one behaviourto a particular function. The operation performed varies by the types of objects or argumentsinvolved.
- **Instance variable:** A variable that is defined inside a method and belongs only to the current instance of aclass.
- **Inheritance:** The transfer of the characteristics of a class to other classes that are derived fromit.
- **Instance:** An individual object of a certain class. An object obj that belongs to a class Circle, for example, is an instance of the classCircle.
- **Instantiation:** The creation of an instance of aclass.
- **Method:** A special kind of function that is defined in a classdefinition.
- **Object:** A unique instance of a data structure that's defined by its class. An object comprises both data members (class variables and instance variables) andmethods.
- **Operator overloading:** The assignment of more than one function to a particular operator.

Creation of Class:

A class is created with the keyword *class* and then writing the classname. The simplest form of class definition looks like this:

```
class ClassName:
        <statement-1>
        .
        .
        <statement-N>
```

Class definitions, like function definitions (def statements) must be executed before they have any effect. (You could conceivably place a class definition in a branch of an if statement, or inside a function.)

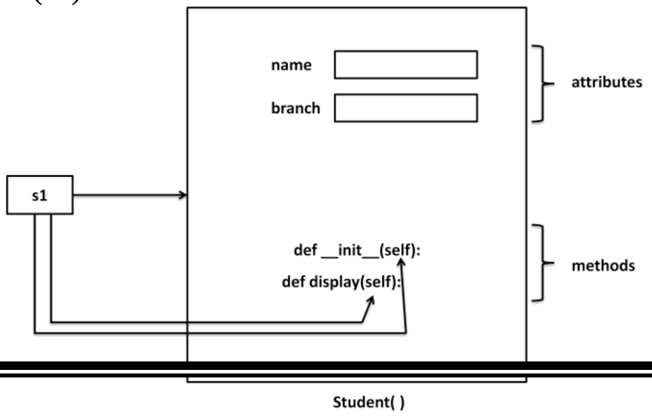
Example: class Student:

```
    definit(self):
        self.name="hari"
        self.branch="CSE"
    def display(self):
        print self.name
        print self.branch
```

- For example, If we „Student‟ class, we can write code in the class that specifies the attributes and actions performed by anystudent.
- Observer that the keyword *class* is used to declare a class. After this, we should write the class name. So, „Student‟ is our class name. Generally, a class name should start with a capital letter, hence „S‟ is a capital in„Student‟.
- In the class, we have written the variables and methods. Since in python, we cannot declare variables, we have written the variables inside a special method, i.e. init (). This method is used to initialize the variables. Hence the name„init‟.
- The method name has two underscores before and after. This indicates that this method is internally defined and we cannot call this methodexplicitly.
- Observe the parameter „self‟ written after the method name in the parentheses. „self‟ is a variable that refers to current classinstance.
- When we create an instance for the Student class, a separate memory block is allocated on the heap and that memory location is default stored in„self‟.
- The instance contains the variables „name‟ and „branch‟ which are called *instance variables*. To refer to instance variables, we can use the dot operator notation along with self as „self.name‟ and„self.branch‟.
- The method display ( ) also takes the „self‟ variable as parameter. This method displays the values of variables by referring them using„self‟.
- The methods that act on instances (or objects) of a class are called instance methods. Instance methods use „self‟ as the first parameter that refers to the location of the instance in thememory.
- Writing a class like this is not sufficient. It should be used. To use a class, we should create an instance to the class. Instance creation represents allotting memory necessary to store the actual data of the variables, i.e., „hari‟,„CSE‟.
- To create an instance, the following syntax is used:

  ```
  instancename = Classname()
  ```
- So, to create an instance to the Student class, we can write as:

  ```
  s1 = Student ()
  ```
- Here „s1‟ represents the instance name. When we create an instance like this, the following steps will take placeinternally:
  1. First of all, a block of memory is allocated on heap. How much memory is to be allocated is decided from the attributes and methods available in the Studentclass.
  2. After allocating the memory block, the special method by the name „init(self)‟ is called internally. This method stores the initial data into the variables. Since this method is useful to construct the instance, it is called„constructor‟.
  3. Finally, the allocated memory location address of the instance is returned into „s1‟ variable. To see this memory location in decimal number format, we can use id( ) function as id(s1).



Self variable:

„self" is a default variable that contains the memory address of the instance of the current class. When an instance to the class is created, the instance name cotains the memory locatin of the instance. This memory location is internally passed to „self".

For example, we create an instance to student class as:

s1 = Student( )

Here, „s1" contains the memory address of the instance. This memory address is internally and by default passed to „self" variable. Since „self" knows the memory address of the instance, it can refer to all the members of the instance.

We use „self" in two eays:

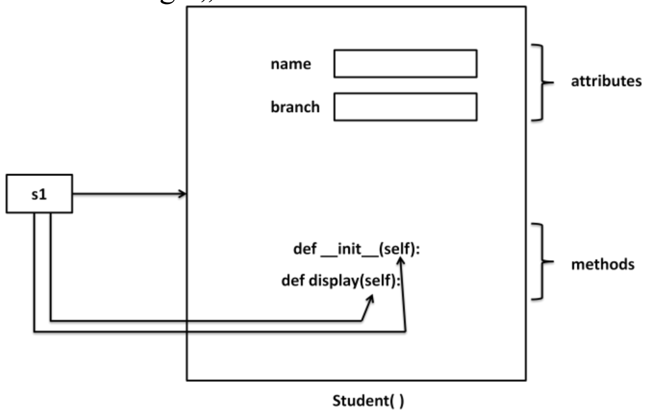
- The self variableis used as first parameter in the constructoras:

  definit(self):

  In this case, „self" can be used to refer to the instance variables inside the constructor.
- „self" can be used as first parameter in the instance methodsas:

  def display(self):

  Here, display( ) is instance method as it acts on the instance variables. If this method wants to act on the instance variables, it should know the memory location of the instance variables. That memory location is by default available to the display( ) method through „self".



Constructor:

A constructor is a special method that is used to initialize the instance variables of a class. In the constructor, we create the instance variables and initialize them with some starting values. The first parameter of the constructor will be „self" variable that contains the memory address of the instance.

```
definit( self ): self.name
        = "hari" self.branch
        ="CSE"
```

Here, the constructor has only one parameter, i.e. „self" using „self.name"and „self.branch", we can access the instance variables of the class. A constructor is called at the time of creating an instance. So, the above constructor will be called when we create an instance as:

s1 = Student()

Let"s take another example, we can write a constructor with some parameters in additionto„self"as:

```
definit( self , n = „ " , b = „ " ):
        self.name = n
        self.branch = b
```

Here, the formal arguments are „n‟ and „b‟ whose default values are given as „‟ (None) and „‟ (None). Hence, if we do not pass any values to constructor at the time of creating an instance, the default values of those formal arguments are stored into name and branch variables. For example,

s1 = Student( )

Since we are not passing any values to the instance, None and None are stored into name and branch. Suppose, we can create an instance as:

s1 = Student(“mothi”, “CSE”)

In this case, we are passing two actual arguments: “mothi” and “CSE” to the Studentinstance.

Example:

```
class Student:
    definit(self,n='',b=''):
        self.name=n
        self.branch=b
    def display(self):
        print "Hi",self.name
        print "Branch", self.branch
s1=Student()
s1.display()
print“ ------------------------------ ”
s2=Student("mothi","CSE")
s2.display()
print“ ------------------------------ ”
```

Output:

```
Hi
Branch
------------------------------
Hi mothi
Branch CSE
------------------------------
```

Types of Variables:

The variables which are written inside a class are of 2 types:

a) Instance Variables
b) Class Variables or StaticVariables

a) Instance Variables

Instance variables are the variables whose separate copy is created in every instance. For example, if „x‟ is an instance variable and if we create 3 instances, there will be 3 copies of „x‟ in these 3 instances. When we modify the copy of „x‟ in any instance, it will not modify the other two copies.

Example: A Python Program to understand instancevariables.

```
classSample:
    definit(self):
        self.x = 10
    def modify(self):
        self.x = self.x + 1
s1=Sample()
s2=Sample()
```

```
print "x in s1=",s1.x
print "x in s2=",s2.x
print" ----------------"
s1.modify()
print "x in s1=",s1.x
print "x in s2=",s2.x
print" ----------------"
```

Output:

```
x in s1= 10
x in s2= 10
----------------
x in s1= 11
x in s2= 10
----------------
```

Instance variables are defined and initialized using a constructor with „self‟ parameter. Also, to access instance variables, we need instance methods with „self‟ as first parameter. It is possible that the instance methods may have other parameters in addition to the „self‟ parameter. To access the instance variables, we can use self.variable as shown in program. It is also possible to access the instance variables from outside the class, as: instancename.variable, e.g. s1.x

b) Class Variables or StaticVariables

Class variables are the variables whose single copy is available to all the instances of the class. If we modify the copy of class variable in an instance, it will modify all the copies in the other instances. For example, if „x‟ is a class variable and if we create 3 instances,thesamecopyof„x‟ispassedtothese3instances.Whenwemodifythecopyof „x‟ in any instance using a class method, the modified copy is sent to the other two instances.

Example: A Python program to understand class variables or staticvariables.

```
classSample:
    x=10
    @classmethod
    def modify(cls):
        cls.x = cls.x + 1
s1=Sample()
s2=Sample()
print "x in s1=",s1.x
print "x in s2=",s2.x
print" ----------------"
s1.modify()
print "x in s1=",s1.x
print "x in s2=",s2.x
print" ----------------"
```

Output:

```
x in s1= 10
x in s2= 10
----------------
x in s1= 11
x in s2= 11
----------------
```

Namespaces:

A *namespace* represents a memory block where names are mapped to objects.

Supposewewrite: n =10

Here, " n" is the name given to the integer object 10. Please recollect that numbers, strings, lists etc. Are all considered as objects in python. The name "n" is linked to 10 in the namespace.

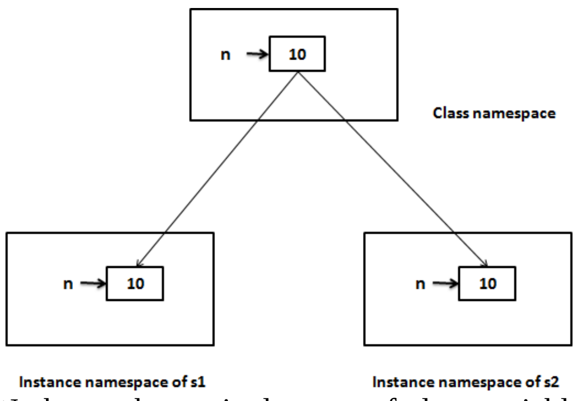
a) ClassNamespace:

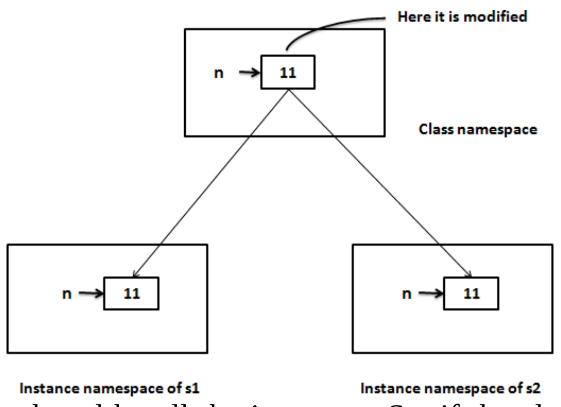
A class maintains its own namespace, called „class namespace". In the class namespace, the names are mapped to class variables. In the following code, „n" is a class variable in the student class. So, in the class namespace, the name „n" is mapped or linked to 10 as shown in figure. We can access it in the class namespace, using classname.variable, as: Student.n which gives10.

Example:

```
 classStudent:
           n =10
printStudent.n          # displays 10
Student.n +=1
printStudent.n          # displays 11
s1 = Student()
prints1.n               # displays 11
s2 = Student()
prints2.n               # displays11
```

Before modifying the classvariable„n"   After modifying the class variable„n"





We know that a single copy of class variable is shared by all the instances. So, if the class variable is modified in the class namespace, since same copy of the variable is modified, the modified copy is available to all the instances.

b) Instance namespace:

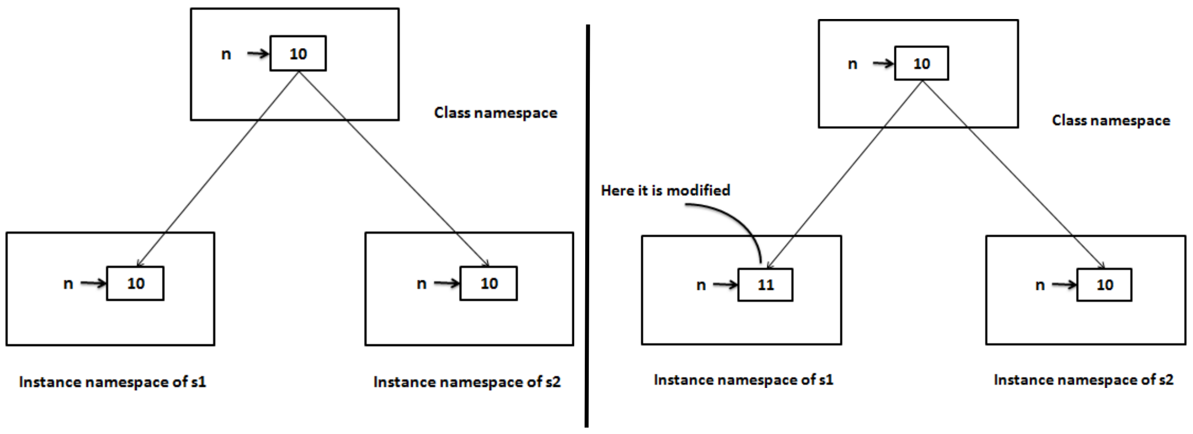
Every instance will have its own name space, called „instance namespace". In the instance namespace, the names are mapped to instance variables. Every instance will have its own namespace, if the class variable is modified in one instance namespace, it will not affect the variables in the other instance namespaces. To access the class variable at the instance level, we have to create instance first and then refer to the variable as instancename.variable.

Example:

```
 classStudent:
           n =10
s1 = Student( )
prints1.n               # displays 10
s1.n +=1
prints1.n               # displays11
```

```
s2 = Student( )
prints2.n                    # displays11
```

Before modifying the classvariable„n‟ After modifying the class variable„n‟



Types of methods:

We can classify the methods in the following 3 types:

a) Instancemethods
   - Accessormethods
   - Mutatormethods

b) Classmethods

c) Staticmethods

a) Instance Methods:

Instance methods are the methods which act upon the instance variables of the class.instance methods are bound to instances and hence called as: instancename.method(). Since instance variable**s** are available in the instance, instance methods need to know the memory address of instance. This is provided through „self‟ variable by default as first parameter for the instance method. While calling the instance methods, we need not pass any value to the „self‟ variable.

Example:

```
class Student:
   definit(self,n='',b=''):
      self.name=n
      self.branch=b
   def display(self):
      print "Hi",self.name
      print "Branch", self.branch
s1=Student()
s1.display()
print“ ------------------------------ ”
s2=Student("mothi","CSE")
s2.display()
print“ ------------------------------ ”
```

- Instance methods are of two types: accessor methods and mutatormethods.
- Accessor methods simply access of read data of the variables. They do not modify the data in the variables. Accessor methods are generally written in the form of getXXXX( ) and hence they are also called *getter*methods.
- Mutator methods are the methods which not only read the data but also modify them. They are written in the form of setXXXX( ) and hence they are also called *setter*methods.

Example:

```
class Student:
    def setName(self,n):
        self.name = n
    def setBranch(self,b):
        self.branch = b
    def getName(self):
        return self.name
    def getBranch(self):
        return self.branch
s=Student()
name=input("Enter Name: ")
branch=input("Enter Branch:")
s.setName(name)
s.setBranch(branch)
print s.getName()
print s.getBranch()
```

b) Classmethods:

These methods act on class level. Class methods are the methods which act on the class variables or static variables. These methods are written using *@classmethod*decorator above them. By default, the first parameter for class methods is „cls‟ which refers to the class itself.

For example, „cls.var‟ is the format to the class variable. These methods are generally called using classname.method( ). The processing which is commonly needed by all the instances of class is handled by the class methods.

Example:

```
class Bird:
    wings = 2

    @classmethod
    def fly(cls,name):
        print name,"flieswith",cls.wings,"wings"

Bird.fly("parrot") #display "parrot flies with 2 wings"
Bird.fly("sparrow") #display "sparow flies with 2 wings"
```

c) Static methods:

We need static methods when the processing is at the class level but we need not involve the class or instances. Static methods are used when some processing is related to the class but does not need the class or its instances to perform any work.

For example, setting environmental variables, counting the number of instances of the class or changing an attribute in another class, etc. are the tasks related to aclass.

Such tasks are handled by static methods. Static methods are written with decorator @staticmethod above them. Static methods are called in the form of classname.method ( ).

Example:

```
class MyClass: n = 0
    definit(self):
        MyClass.n = Myclass.n + 1
    def noObjects():
        print "No. of instances created: ", MyClass.n
m1=MyClass()
m2=MyClass()
m3=MyClass()
MyClass.noObjects()
```

Inheritance:

- Software development is a team effort. Several programmers will work as a team to developsoftware.
- When a programmer develops a class, he will use its features by creating an instance to it. When another programmer wants to create another class which is similar to the class already created, then he need not create the class from the scratch. He can simply use the features of the existing class in creating his ownclass.
- Deriving new class from the super class is called*inheritance*.
- The child class inherits the attributes of its parent class, and you can use those attributes as if they were defined in the childclass.
- A child class can also override data members and methods from theparent.

Syntax:

```
class Subclass(BaseClass):
        <class body>
```

- When an object is to SubClassis created, it contains a copy of BaseClass within it. This means there is a relation between the BaseClass and SubClassobjects.
- We do not create BaseClassobject,but still a copy of it is available to SubClassobject.
- By using inheritance, a programmer can develop classes very easilt. Hence programmer‟s productivity is increased. Productivity is a term that refers to the code developed by the programmer in a given span oftime.
- If the programmer used inheritance, he will be able to develop more code in lesstime.
- In inheritance, we always create only the sub class object. Generally, we do not create super class object. The reason is clear. Since all the members of the super class are available to sub class, when we crate an object, we can access the members of both the super and subclasses.

The super( ) method:

- super( ) is a built-in method which is useful to call the super class constructor or methods from the subclass.
- Any constructor written in the super class is not available to the sub class if the sub class has a constructor.
- Then how can we initialize the super class instance variables and use them in the sub class? This is done by calling super class constructor using super( ) method from inside the sub classconstructor.
- super( ) is a built-in method which contains the history of super classmethods.
- Hence, we can use super( ) to refer to super class constructor and methods from a aubclass. So, super( ) can be usedas:

```
super().init()  # call super classconstructor

super().init(arguments) # call super class constructor and pass arguments super().method() #
call super classmethod
```

Example: Write a python program to call the super class constructor in the sub class using super( ).

```
class Father:

    definit(self, p = 0): self.property =
        p

    def display(self):

        print "Father Property",self.property
class Son(Father):

    definit(self,p1 = 0, p = 0):
        super().init(p1) self.property1
        = p

    def display(self):

        print "Son Property",self.property+self.property1 s=Son(200000,500000)

    isplay()
```

Output:

```
Son Property 700000
```

Example: Write a python program to access base class constructor and method in the sub class using super( ).

```
class Square:

    definit(self, x = 0):
        self.x = x

    def area(self):

        print "Area of square", self.x * self.x
classRectangle(Square):

    definit(self, x = 0, y = 0):
        super().init(x)

        self.y = y def
    area(self):

        super().area()

        print "Area of Rectangle", self.x * self.y r =
Rectangle(5,16)

r.area()
```

Output:

```
Area of square 25 Area
of Rectangle 80
```

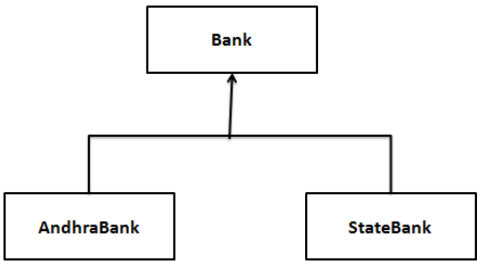
Types of Inheritance:

There are mainly 2 types of inheritance.

a) Singleinheritance
b) Multipleinheritance

a) Single inheritance

Deriving one or more sub classes from a single base class is called „single inheritance‟. In single inheritance, we always have only one base class, but there can be n number of sub classes derived from it. For example, „Bank‟ is a single base clas from where we derive „AndhraBank‟ and „StateBank‟ as sub classes. This is called single inheritance.



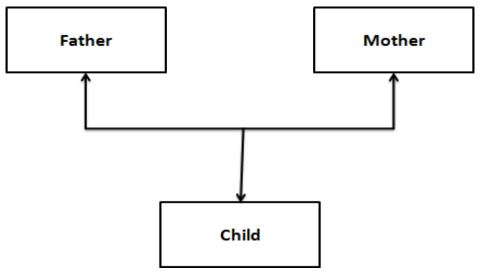
Example:

```
class Bank:
    cash = 100

    @classmethoddefb
    alance(cls):

        printcls.cash

class AndhraBank(Bank):
    cash = 500
    @classmethod

    def balance(cls):

        print "AndhraBank",cls.cash + Bank.cash class
StateBank(Bank):

    cash = 300
    @classmethoddef
    balance(cls):

        print "StateBank",cls.cash + Bank.cash
a=AndhraBank()

a.balance()                    # displays AndhraBank 600
s=StateBank()

s.balance()                    #displays StateBank400
```

b) Multiple inheritance

Deriving sub classes from multiple (or more than one)base classes is called „multiple inheritance‟. All the members of super classes are by default available to sub classes and the sub classes in turn can have their own members.

The best example for multiple inheritance is that parents are producing the children and the children inheriting the qualities of the parents.



Example:

```
class Father:

    def height(self):

        print "Height is 5.8 incehs" class
Mother:

    defcolor(self):

        print "Color is brown" class
Child(Father, Mother):
  p                    ()
a
s                      c.height() # displays Height is 5.8 incehsc.color() #
s                      displays Color is brown

c
=
C
h
i
l
d
```

Problem in Multiple inheritance:

- If the sub class has a constructor, it overrides the super class constructor and hence the super class constructor is not available to the subclass.
- But writing constructor is very common to initialize the instancevariables.
- In multiple inheritance, let"s assume that a sub class „C" is derived from two super classes „A" and „B" having their own constructors. Even the sub class „C" also has itsconstructor.

Example-1:

```
class A(object):

     definit(self):

          print"ClassA"                              Output:

  classB(object):                                                    Class A

        definit(self):                                                ClassC

          print "Class B"
classC(A,B,object):

     definit(self): super().init()
          print "ClassC"

c1= C()
```

Example-2:

```
class A(object):

     definit(self): super().init()

          print"ClassA"                              Output:

  classB(object):                                                    Class B

       definit(self):                                                Class A

             super().init()                                           ClassC

          print "Class B"
classC(A,B,object):

     definit(self): super().init()
          print "ClassC"

c1= C()
```

Method Overriding:

When there is a method in the super class, writing the same method in the sub class so that it replaces the super class method is called „method overriding". The programmer overrides the super class methods when he does not want to use them in sub class.

Example:

```
import math
class square:

     def area(slef, r):

          print "Square area=",r * r class
Circle(Square):

     def area(self, r):

          print "Circle area=", math.pi * r * r c=Circle()

c.area(15) # displays Circle area= 706.85834
```

Data hiding:

An object's attributes may or may not be visible outside the class definition. You need to name attributes with a double underscore prefix, and those attributes then are not be directly visible tooutsiders.

Example:

```
classJustCounter:

    secretCount = 0
    defcount(self):

        self.secretCount += 1
        print self.secretCount

counter = JustCounter()
```

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

```
1
2

Traceback (most recent call last):
  File "C:/Python27/JustCounter.py", line 9, in <module>
    print counter.secretCount
AttributeError: JustCounter instance has no attribute 'secretCount'
```

Python protects those members by internally changing the name to include the class name. You can access such attributes as *object._classNameattrName*. If you would replace your last line as following, then it works for you:

```
.........................
print counter._JustCountersecretCount
```

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

```
1
2
2
```

Errors and Exceptions:

As human beings, we commit several errors. A software developer is also a human being and hence prone to commit errors wither in the design of the software or in writing the code.Theerrorsinthesoftwarearecalled„bugs‟andtheprocessofremovingthemarecalled „debugging‟. In general, we can classify errors in a program into one of these three types:

a) Compile-time errors
b) Runtime errors
c) Logicalerrors

a) Compile-time errors

These are syntactical errors found in the code, due to which a program fails to compile. For example, forgetting a colon in the statements like if, while, for, def, etc. will result in compile-time error. Such errors are detected by python compiler and the line number along with error description is displayed by the python compiler.

Example: A Python program to understand the compile-time error.

```
a = 1
if a == 1
    print “hello”
```

Output:

```
File ex.py, line 3
   If a == 1
            ^
SyntaxError: invalid syntax
```

b) Runtime errors

When PVM cannot execute the byte code, it flags runtime error. For example, insufficient memory to store something or inability of PVM to execute some statement come under runtime errors. Runtime errors are not detected by the python compiler. They are detected by the PVM, Only atruntime.

Example: A Python program to understand the compile-time error.

```
print "hai"+25
```

Output:

```
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    print "hai"+25
TypeError: cannot concatenate 'str' and 'int' objects
```

c) Logicalerrors

These errors depict flaws in the logic of the program. The programmer might be using a wrong formula of the design of the program itself is wrong. Logical errors are not detected either by Python compiler of PVM. The programme is solely responsible for them. In the following program, the programmer wants to calculate incremented salary of an employee, but he gets wrong output, since he uses wrong formula.

Example: A Python program to increment the salary of an employee by 15%.

```
def increment(sal):

        sal = sal * 15/100
        return sal

sal = increment(5000)

print "Salary after Increment is", sal
```

Output:

```
Salary after Increment is 750
```

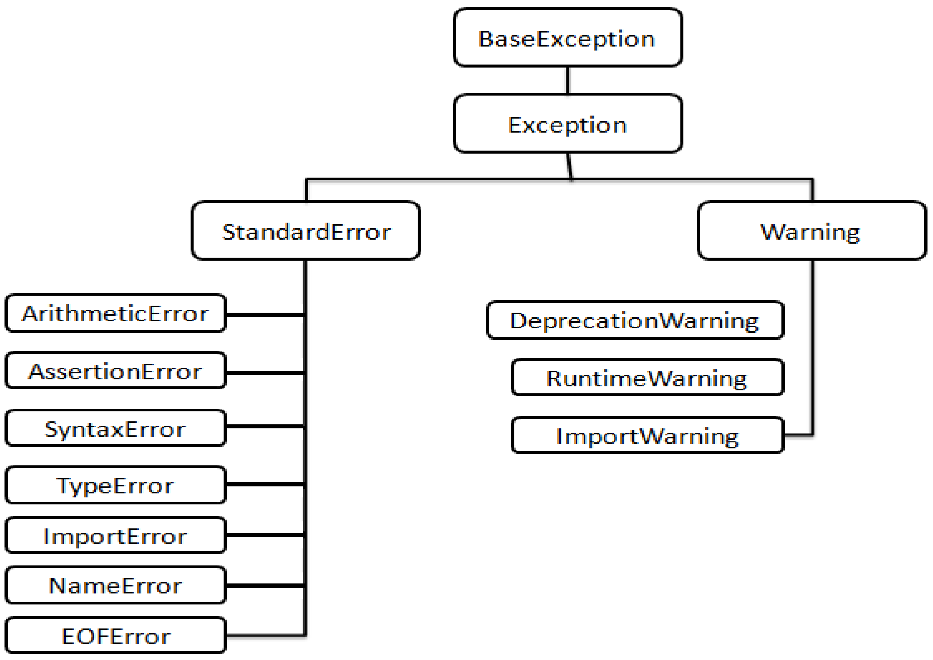
From the above program the formula for salary is wrong, because only the increment but it is not adding it to the original salary. So, the correct formula would be:

```
sal = sal + sal * 15/100
```

- ✓ Compile time errors and runtime errors can be eliminated by the programmer by modifying the program sourcecode.
- ✓ In case of runtime errors, when the programmer knows which type of error occurs, he has to handle them using exception handlingmechanism.

Exceptions:

- ➢ An exception is a runtime error which can be handled by the programmer.
- ➢ That means if the programmer can guess an error in the program and he can do somethingtoeliminatetheharmcaused bythaterror,thenitiscalledan„exception‟.
- ➢ If the programmer cannot do anything in case of an error, then it is called an „error‟ and not anexception.
- ➢ All exceptions are represented as classes in python. The exceptions which are already available in python are called „built-in‟ exceptions. The base class for all built-in exceptions is „BaseException‟class.
- ➢ From BaseException class, the sub class „Exception‟ is derived. From Exception class, the sub classes „StandardError‟ and „Warning‟ arederived.
- ➢ All errors (or exceptions) are defined as sub classes of StandardError. An error should be compulsory handled otherwise the program will notexecute.
- ➢ Similarly, all warnings are derived as sub classes from „Warning‟ class. A warning represents a caution and even though it is not handled, the program will execute. So, warnings can be neglected but errors cannotneglect.
- ➢ Just like the exceptions which are already available in python language, a programmer can also create his own exceptions, called „user-defined‟exceptions.



Exception Handling:

- Thepurposeofhandlingerrorsistomaketheprogram*robust*.Theword„robust‟means „strong‟. A robust program does not terminate in the middle.
- Also, when there is an error in the program, it will display an appropriate message to the user and continueexecution.
- Designing such programs is needed in any softwaredevelopment.
- For that purpose, the programmer should handle the errors. When the errors can be handled, they are calledexceptions.

To handle exceptions, the programmer should perform the following four steps:

Step 1: The programmer should observe the statements in his program where there may be a possibility of exceptions. Such statements should be written inside a „try‟ block. A try block looks like as follows:

try:
statements

The greatness of try block is that even if some exception arises inside it, the program will not be terminated. When PVM understands that there is an exception, it jumps into an „except‟ block.

Step 2: The programmer should write the „except‟ block where he should display the exception details to the user. This helps the user to understand that there is some error in the program. The programmer should also display a message regarding what can be done to avoid this error. Except block looks like asfollows:

except exceptionname:
statements

The statements written inside an except block are called „handlers‟ since they handle the situation when the exception occurs.

Step 3: If no exception is raised, the statements inside the „else‟ block is executed. Else block looks like as follows:

else:
statements

Step 4: Lastly, the programmer should perform clean up actions like closing the files and terminating any other processes which are running. The programmer should write this code in the finally block. Finally block looks like as follows:

finally:

statements

The speciality of finally block is that the statements inside the finally block are executed irrespective of whether there is an exception or not. This ensures that all the opened files are properly closed and all the running processes are properly terminated. So, the data in the files will not be corrupted and the user is at the safe-side.

Here, the complete exception handling syntax will be in the following format:

```
try:

    statements
exceptException1:

    statements
exceptException2:

    statements
else:

    statements
finally:

    statements
```

The following points are followed in exception handling:

- ✓ A single try block can be followed by several exceptblocks.
- ✓ Multiple except blocks can be used to handle multipleexceptions.
- ✓ We cannot write except blocks without a tryblock.
- ✓ We can write a try block without any exceptblocks.
- ✓ Else block and finally blocks are notcompulsory.
- ✓ When there is no exception, else block is executed after tryblock.
- ✓ Finally block is alwaysexecuted.

Example: A python program to handle IOError produced by open() function.

```
import sys
try:

   f = open('myfile.txt','r') s =
   f.readline()

   print s
   f.close()

except IOError as e:

   print "I/O error", e.strerror
except:

   print "Unexpected error:"
```

Output:I/O error No such file or directory

In the if the file is not found, then IOErroris raised. Then „except‟ block will display a message: „I/O error‟. if the file is found, then all the lines of the file are read using readline()method.

List of Standard Exceptions

Exception Name	Description
Exception	Base class for all exceptions
StopIteration	Raised when the next() method of an iterator does not point to any object.
SystemExit	Raised by the sys.exit() function.
StandardError	Base class for all built-in exceptions except StopIteration and SystemExit.
ArithmeticError	Base class for all errors that occur for numeric calculation.
OverflowError	Raised when a calculation exceeds maximum limit for a numeric type.
FloatingPointError	Raised when a floating point calculation fails.
ZeroDivisionError	Raised when division or modulo by zero takes place for all numeric types.
AssertionError	Raised in case of failure of the Assert statement.
AttributeError	Raised in case of failure of attribute reference or assignment.
EOFError	Raised when there is no input from either the raw_input() or input() function and the end of file is reached.
ImportError	Raised when an import statement fails.
KeyboardInterrupt	Raised when the user interrupts program execution, usually by pressing Ctrl+c.
LookupError	Base class for all lookup errors.
IndexError	Raised when an index is not found in a sequence.
KeyError	Raised when the specified key is not found in the dictionary.
NameError	Raised when an identifier is not found in the local or global namespace.
UnboundLocalError	Raised when trying to access a local variable in a function or method but no value has been assigned to it.
EnvironmentError	Base class for all exceptions that occur outside the Python environment.
IOError	Raised when an input/ output operation fails, such as the print statement or the open() function when trying to open a file that does not exist.
OSError	Raised for operating system-related errors.
SyntaxError	Raised when there is an error in Python syntax.
IndentationError	Raised when indentation is not specified properly.
SystemError	Raised when the interpreter finds an internal problem, but when this error is encountered the Python interpreter does not exit.
SystemExit	Raised when Python interpreter is quit by using the sys.exit() function. If not handled in the code, causes the interpreter to exit.
TypeError	Raised when an operation or function is attempted that is invalid for the specified data type.
ValueError	Raised when the built-in function for a data type has the valid type of arguments, but the arguments have invalid values specified.
RuntimeError	Raised when a generated error does not fall into any category.
NotImplementedError	Raised when an abstract method that needs to be implemented in an inherited class is not actually implemented.

The Except Block:

The „except‟ block is useful to catch an exception that is raised in the try block. When there is an exception in the try block, then only the except block is executed. it is written in various formats.

1. To catch the exception which is raised in the try block, we can write except block with the Exceptionclass nameas:

except Exceptionclass:

2. We can catch the exception as an object that contains some description about the exception.

except Exceptionclass as obj:

3. To catch multiple exceptions, we can write multiple catch blocks. The other way is to use a single except block and write all the exceptions as a tuple inside paranthesesas:

except (Exceptionclass1, Exceptionclass2,):

4. To catch any type of exception where we are not bothered about which type of exception it is, we can write except block without mentioning any Exceptionclass name as:

except:

Example:

```
try:

    f = open('myfile.txt','w')
    a=input("Enter a value ")
    b=input("Enter a value ")
    c=a/float(b)

    s = f.write(str(c))

    print “Result is stored”
except ZeroDivisionError:

    print "Division is not possible" except:

    print "Unexpected error:"
finally:

    f.close()
```

Output:

```
Enter  a  value  1
Enter  a  value  5
Result is stored
```

Raising an Exception

You can raise exceptions in several ways by using the raise statement. The general syntax for the **raise** statement is as follows.

```
raise [Exception [, args [, traceback]]]
```

Here, *Exception* is the type of exception (For example, NameError) and *argument* is a value for the exception argument. The argument is optional; if not supplied, the exception argument is None.

For Example, If you need to determine whether an exception was raised but don"t intend to handle it, a simpler form of the raise statement allows you to re-raise the exception:

```
try:
     raise NameError('HiThere')
   except NameError:

     print 'An exception flew by!'
     raise
```

User-Defined Exceptions:

- Like the built-in exceptions of python, the programmer can also create his own exceptionswhicharecalled„User-definedexceptions"or„Customexceptions". Weknow Python offers many exceptions which will raise in different contexts.
- But, there may be some situations where none of the exceptions in Python are useful for the programmer. In that case, the programme has to create his/her own exception and raiseit.
- For example, let"s take a bank where customers have accounts. Each account is characterized should by customer name and balanceamount.
- The rule of the bank is that every customer should keep minimum Rs. 2000.00 as balance amount in hisaccount.
- The programmer now is given a task to check the accounts to know every customer is maintaining minimum balance of Rs. 2000.00 ornot.
- If the balance amount is below Rs. 2000.00, then the programmer wants to raise an exception saying „Balance amount is less in the account of so and so person." This will be helpful to the bank authorities to find out thecustomer.
- So, the programmer wants an exception that is raised when the balance amount in an account is less than Rs. 2000.00. Since there is no such exception available in python, the programme has to create his/her ownexception.
- For this purpose, he/she has to follow thesesteps:

1. Since all exceptions are classes, the programmeis supposed to create his own exceptionasaclass.Also,heshouldmakehisclassasasubclasstothein-built „Exception" class.

   ```
   class MyException(Exception):
       definit(self, arg):

           self.msg = arg
   ```

   Here, MyException class is the sub class for „Exception" class. This class has a constructor where a variable „msg" is defined. This „msg" receives a message passed from outside through „arg".

2. The programmer can write his code; maybe it represents a group of statements or a

function. When the programmer suspects the possibility of exception, he should raise his own exception using „raise" statementas:

raise MyException('message')

Here, raise statement is raising MyException class object that contains the given „message".

3. The programmer can insert the code inside a „try" block and catch the exceptionusing „except" block as:

```
try:
    code

  except MyException as me:
      print me
```

Here, the object „me" contains the message given in the raise statement. All these steps are shown in below program.

Example:

```
class MyException(Exception): definit(self, arg):
      self.msg = arg def check(dict):
   for k,v in dict.items():
      print "Name=",k,"Balance=",v if v<2000.00:
         raise MyException("Balance amount is less in the account of "+k)

bank={"ravi":5000.00,"ramu":8500.00,"raju":1990.00} try:
   check(bank)
except MyException as me: print me.msg
```

Output:

```
Name= ramu Balance= 8500.0 Name= ravi
Balance= 5000.0 Name= raju Balance= 1990.0
Balance amount is less in the account of raju
```