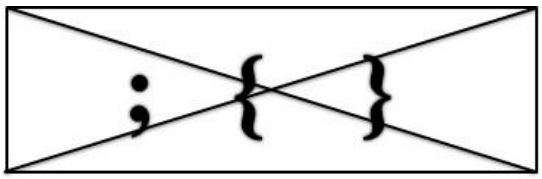


Definition:

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

- **Python is Interpreted:** Python is processed at runtime by the interpreter. You do not need to compile your program before executing it. This is similar to PERL and PHP.
- **Python is Interactive:** You can actually sit at a Python prompt and interact with the interpreter directly to write your programs.
- **Python is Object-Oriented:** Python supports Object-Oriented style or technique of programming that encapsulates code within objects.
- **Python is a Beginner's Language:** Python is a great language for the beginner-level programmers and supports the development of a wide range of applications from simple text processing to WWW browsers to games.

History of Python

- Python was developed by **Guido van Rossum** in the late eighties and early nineties at the National Research Institute for Mathematics and Computer Science in the Netherlands.
- Python is derived from many other languages, including ABC, Modula-3, C, C++, Algol-68, SmallTalk, Unix shell, and other scripting languages.
- At the time when he began implementing Python, Guido van Rossum was also reading the published scripts from "Monty Python's Flying Circus" (a BBC comedy series from the seventies, in the unlikely case you didn't know). It occurred to him that he needed a name that was short, unique, and slightly mysterious, so he decided to call the language Python.
- Python is now maintained by a core development team at the institute, although **Guido van Rossum** still holds a vital role in directing its progress.
- Python 1.0 was released on **20 February, 1991.**
- Python 2.0 was released on **16 October 2000** and had many major new features, including a cycle detecting garbage collector and support for Unicode. With this release the development process was changed and became more transparent and community-backed.
- Python 3.0 (which early in its development was commonly referred to as Python 3000 or py3k), a major, backwards-incompatible release, was released on **3 December 2008** after a long period of testing. Many of its major features have been back ported to the backwards-compatible Python 2.6.x and 2.7.x version series.
- In January 2017 Google announced work on a Python 2.7 to go transcompiler, which The Register speculated was in response to Python 2.7's planned end-of-life.

Python Features:

Python's features include:

- **Easy-to-learn:** Python has few keywords, simple structure, and a clearly defined syntax. This allows the student to pick up the language quickly.
- **Easy-to-read:** Python code is more clearly defined and visible to the eyes.
- **Easy-to-maintain:** Python's source code is fairly easy-to-maintain.
- **A broad standard library:** Python's bulk of the library is very portable and cross-platform compatible on UNIX, Windows, and Macintosh.
- **Interactive Mode:** Python has support for an interactive mode which allows interactive testing and debugging of snippets of code.
- **Portable:** Python can run on a wide variety of hardware platforms and has the same interface on all platforms.
- **Extendable:** You can add low-level modules to the Python interpreter. These modules enable programmers to add to or customize their tools to be more efficient.
- **Databases:** Python provides interfaces to all major commercial databases.
- **GUI Programming:** Python supports GUI applications that can be created and ported to many system calls, libraries, and windows systems, such as Windows MFC, Macintosh, and the X Window system of UNIX.
- **Scalable:** Python provides a better structure and support for large programs than shell scripting.

Need of Python Programming

- **Software quality**

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

- ***Developer productivity***

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

- ***Support libraries***

  Python comes with a large collection of prebuilt and portable functionality, known as the *standard library*. This library supports an array of application-level programming tasks, from text pattern matching to network scripting. In addition, Python can be extended with both home grown libraries and a vast collection of third-party application support software. Python‘s *third-party domain* offers tools for website construction, numeric programming, serial port access, game development, and much more (see ahead for a sampling).

- ***Component integration***

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

- ***Enjoyment***

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

- **It's Object-Oriented**

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

- **It's Free**

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

- **It's Portable**

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

- **It's Powerful**

  From a features perspective, Python is something of a hybrid. Its tool set places it between traditional scripting languages (such as Tcl, Scheme, and Perl), and systems languages (such as C, C++, and Java). Python provides all the simplicity and ease of use of a scripting language, along with more advanced programming tools typically found in systems development languages.

- **Automatic memory management**

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

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

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

- **It's Mixable**

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

- **It's Easy to Use**

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

- **It's Easy to Learn**

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

- **Internet Scripting**

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

- **Database Programming**

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

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

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

Who Uses Python Today?

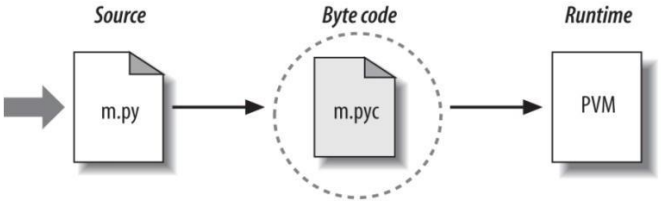
1. *Google* makes extensive use of Python in its web search systems.
2. The popular *YouTube* video sharing service is largely written in Python.
3. The *Dropbox* storage service codes both its server and desktop client software primarily in Python.
4. The *Raspberry Pi* single-board computer promotes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

Byte code Compilation:

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

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:

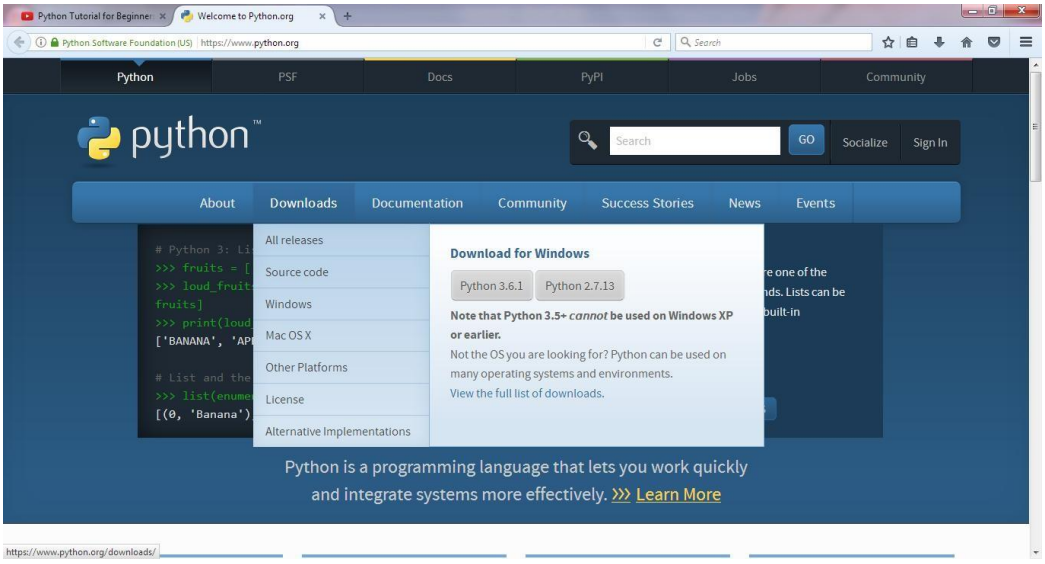
1. Systems Programming
2. GUIs
3. Internet Scripting
4. Component Integration
5. Database Programming
6. Rapid Prototyping
7. Numeric and Scientific Programming

What Are Python's Technical Strengths?

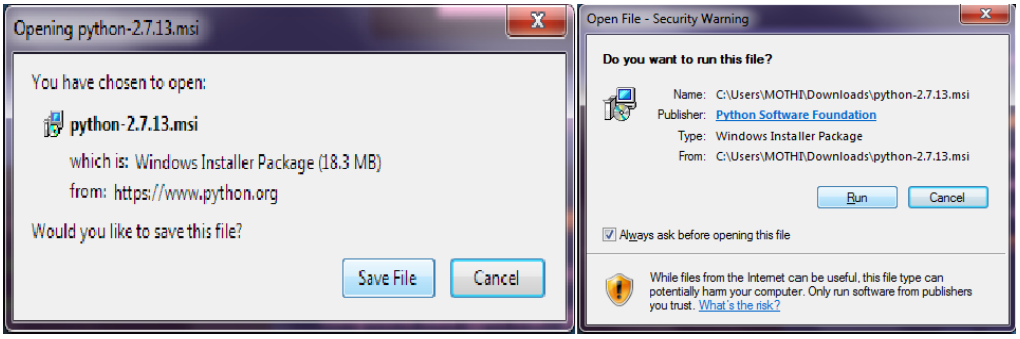
1. It's Object-Oriented and Functional
2. It's Free
3. It's Portable
4. It's Powerful
5. It's Mixable
6. It's Relatively Easy to Use
7. It's Relatively Easy to Learn

Download and installation Python software:

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

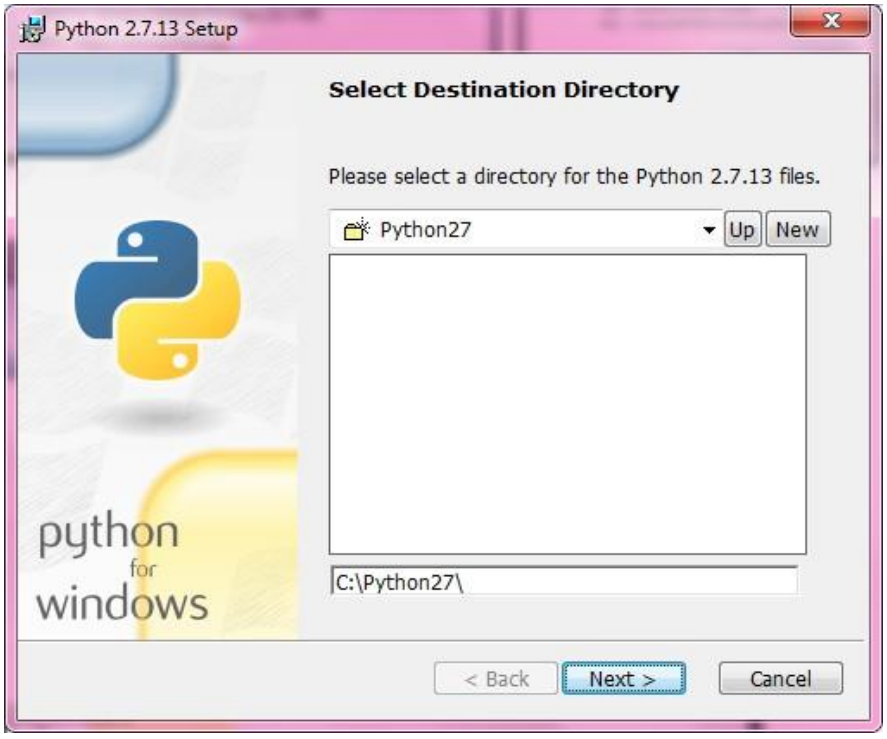


Step 2: Click on **Python 2.7.13** and download. After download open the file.



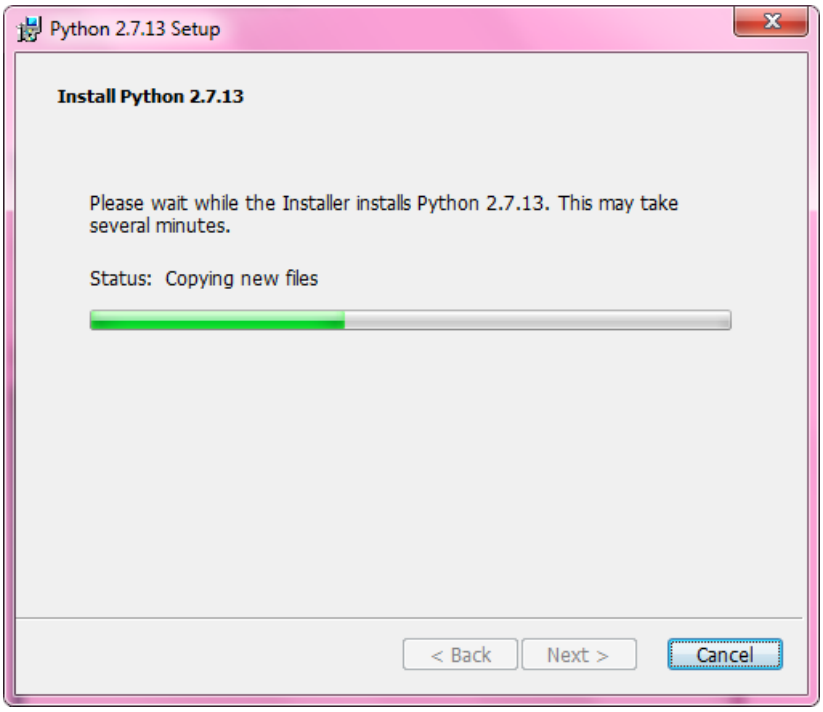
Step 3: Click on **Next** to continue.

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



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

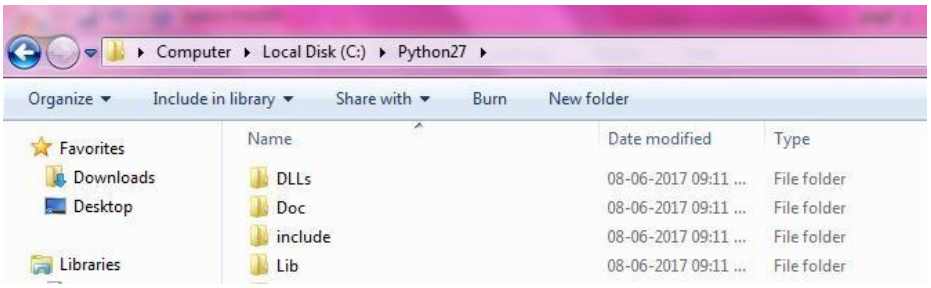
Step 6: The installation has been processed.



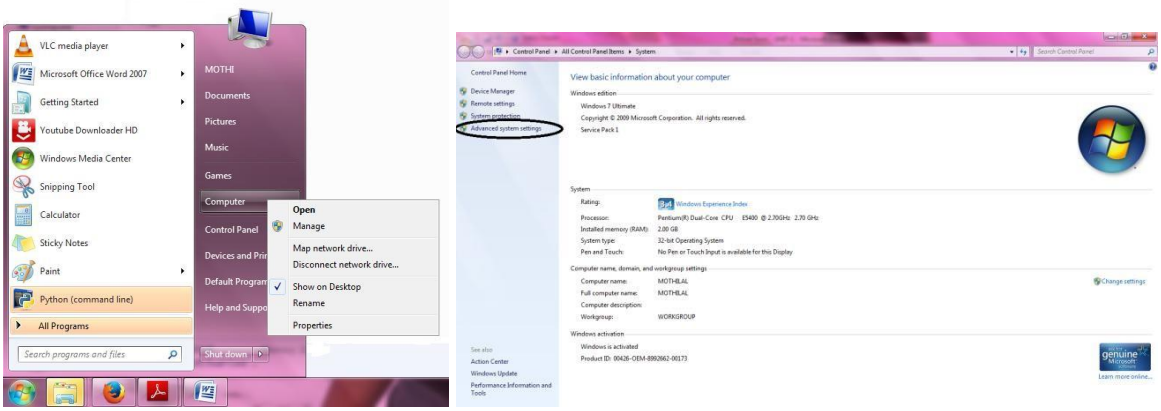
Step 7: Click the **Finish** to complete the installation.

Setting up PATH to python:

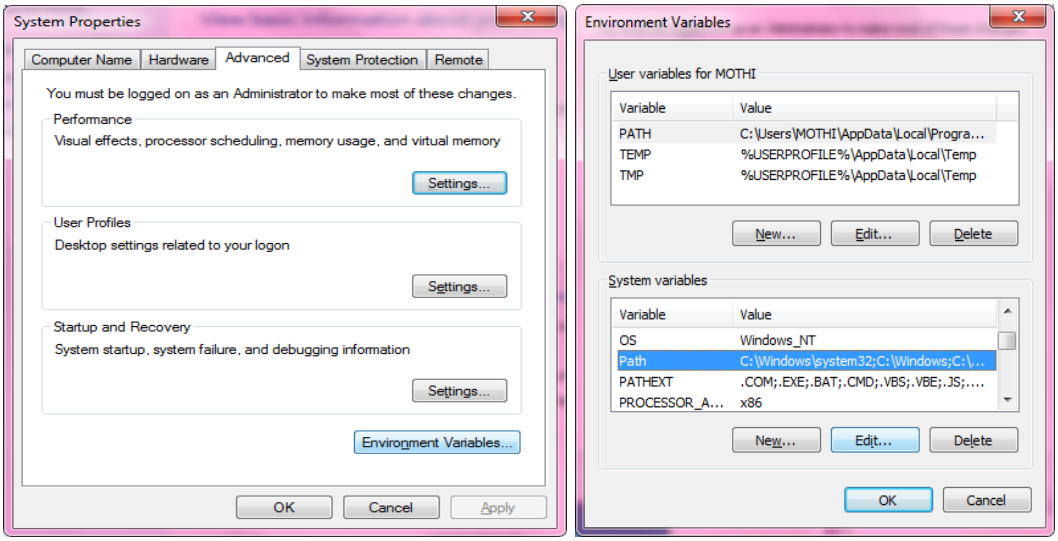
- Programs and other executable files can be in many directories, so operating systems provide a search path that lists the directories that the OS searches for executables.
- The path is stored in an environment variable, which is a named string maintained by the operating system. This variable contains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



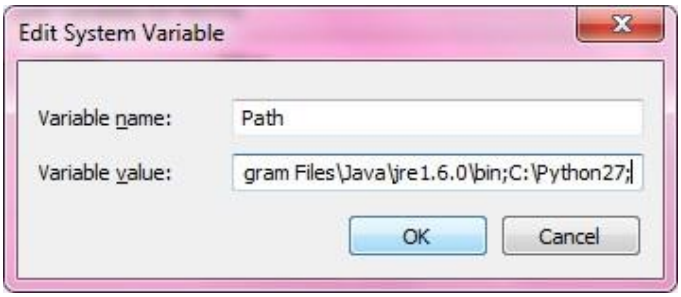
- Right-click the My Computer icon on your desktop and choose **Properties**. And then select **Advanced System properties.**



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



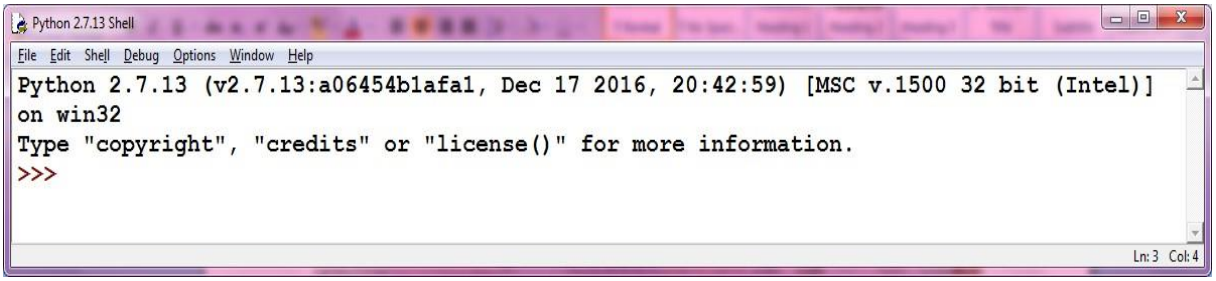
- Add semicolon (;) at end and copy the location **C:\Python27** and give semicolon (;) and click OK.



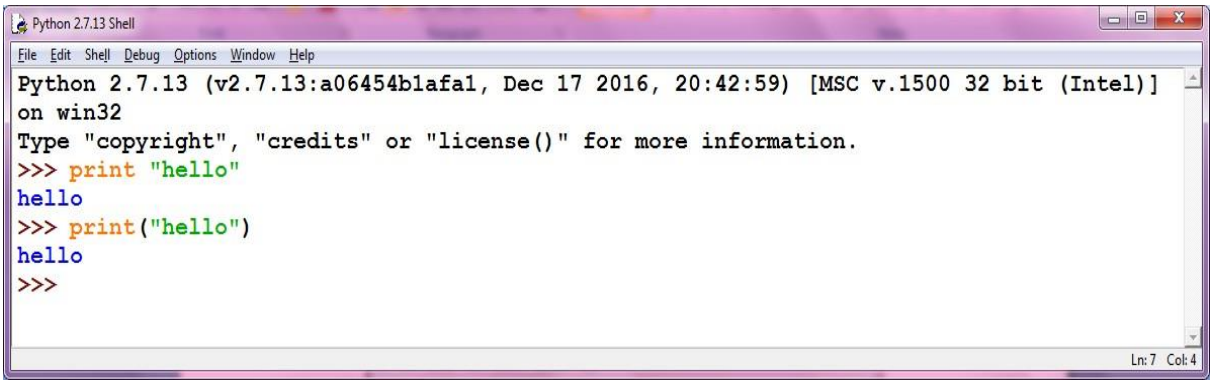
Running Python:

a. Running Python Interpreter:

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



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

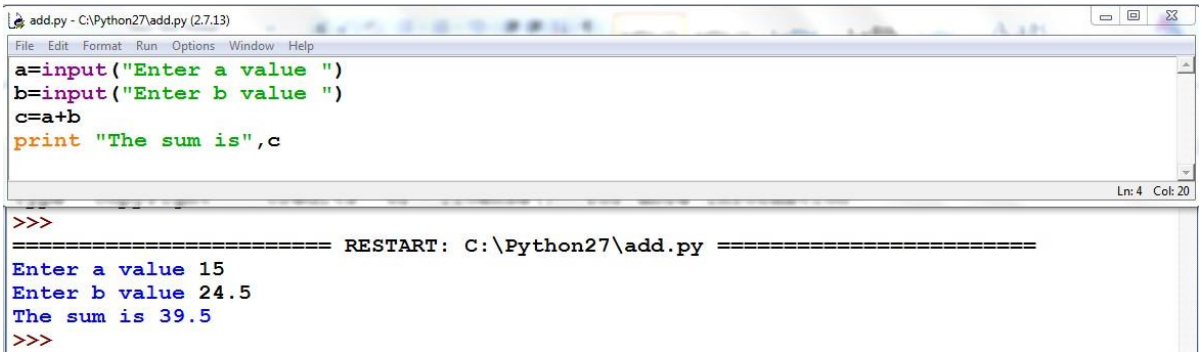


b. Running Python Scripts in IDLE:

- 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→Run Module.



c. Running python scripts in Command Prompt:

- Before going to run we have to check the PATH in environment variables.
- Open your text editor, type the following text and save it as hello.py.

  print "hello"
- And run this program by calling python hello.py. Make sure you change to the directory where you saved the file before doing it.

Variables:

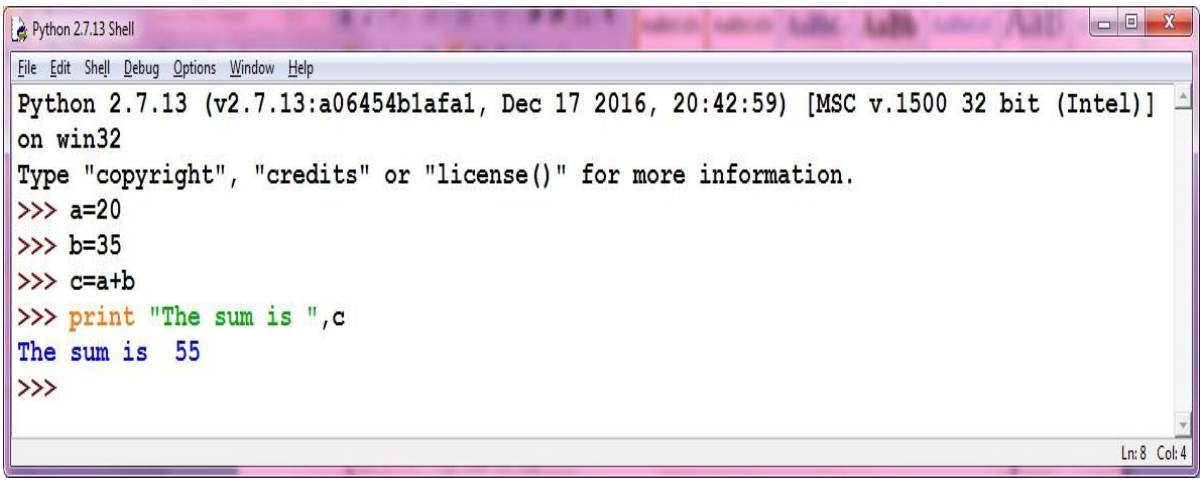
Variables are nothing but reserved memory locations to store values. This means that when you create a variable you reserve some space in memory.

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

Assigning Values to Variables

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

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



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.

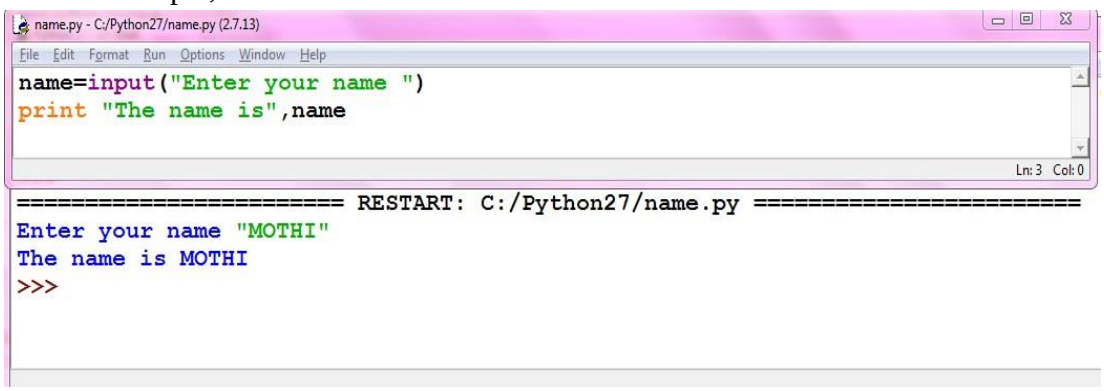
and	elif	if	print
as	else	import	raise
assert	except	in	return
break	exec	is	try
class	finally	lambda	while
continue	for	not	with
def	from	or	yield
del	global	pass	

INPUT Function:

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

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

For example,



OUTPUT function:

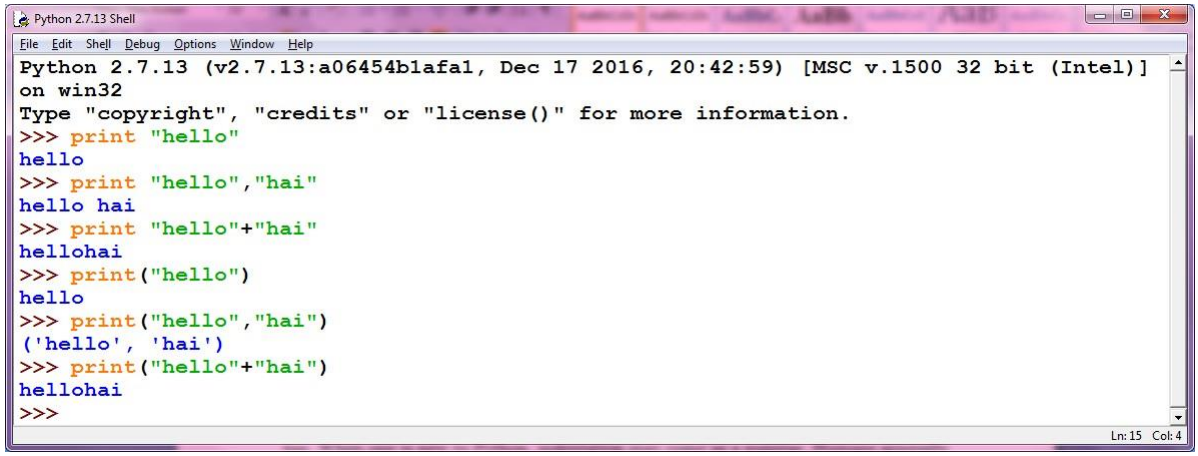
We use the print() function or print keyword to output data to the standard output device (screen). This function prints the object/string written in function.

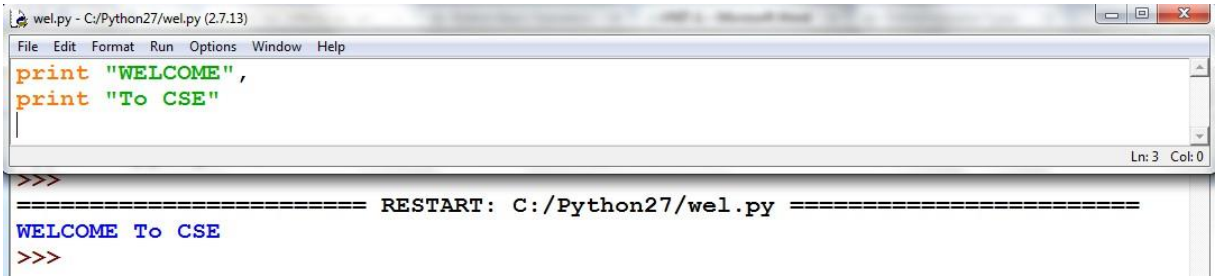
The actual syntax of the print() function is

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).

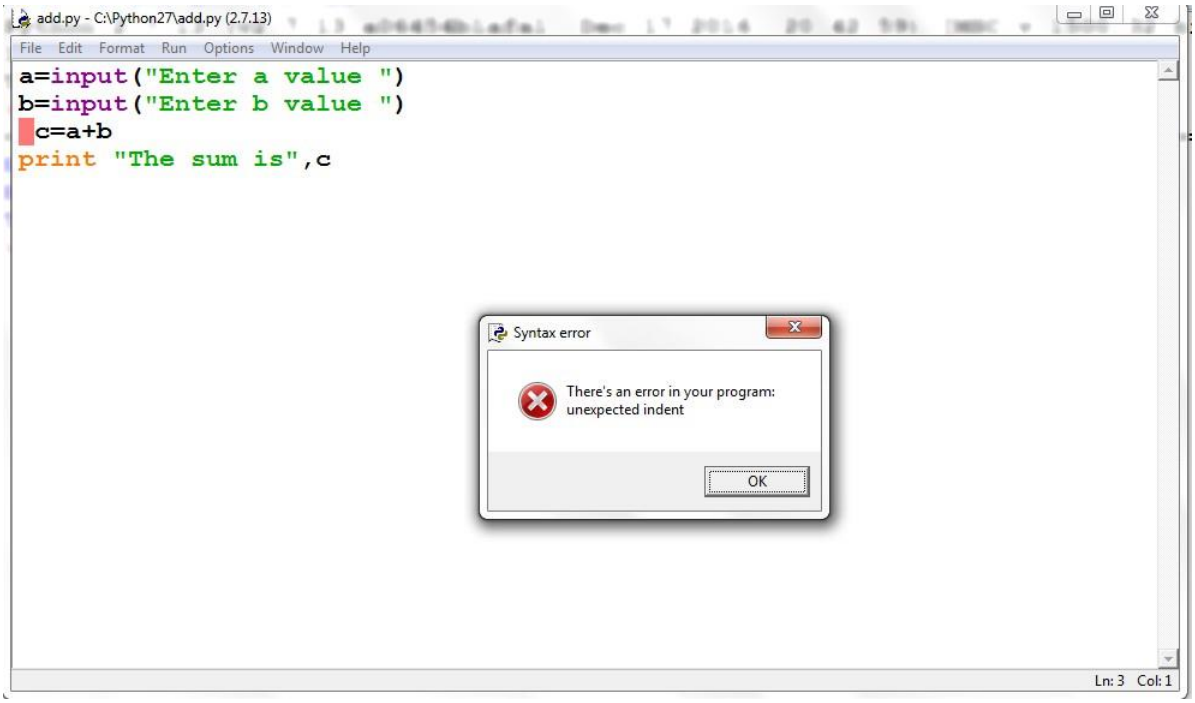




Indentation

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

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



Standard Data Types:

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

Python has five standard data types:

- Numbers
- String
- Boolean
- List
- Tuple
- Set
- Dictionary

Python Numbers:

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

Python supports four different numerical types:

- int (signed integers)
- long (long integers, they can also be represented in octal and hexadecimal)
- float (floating point real values)
- complex (complex numbers)

Python allows you to use a lowercase L with long, but it is recommended that you use only an uppercase L to avoid confusion with the number 1. Python displays long integers with an uppercase L.

A complex number consists of an ordered pair of real floating-point numbers denoted by x + yj, where x is the real part and b is the imaginary part of the complex number.

For example:

Program:

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

Output:

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

Python Strings:

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

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

Program:

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

Output:

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

Built-in String methods for Strings:

SNO	Method Name	Description
1	capitalize()	Capitalizes first letter of string.
2	center(width, fillchar)	Returns a space-padded string with the original string centered to a total of width columns.
3	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 the original string left-justified to a total of width columns.
21	lower()	Converts all uppercase letters in string to lowercase.
22	lstrip()	Removes all leading whitespace in string.
23	maketrans()	Returns a translation table to be used in 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.

Types of Operators:

Python language supports the following types of operators.

- Arithmetic Operators **+, -, *, /, %, **, //**
- Comparison (Relational) Operators **= =, ! =, < >, <, >, <=, >=**
- Assignment Operators **=, +=, -=, *=, /=, %=, **=, //=**
- Logical Operators **and, or, not**
- Bitwise Operators **&, |, ^, ~,<<, >>**
- Membership Operators **in, not in**
- Identity Operators **is, is not**

Arithmetic Operators:

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

Operator	Description	Example
+ Addition	Adds values on either side of the operator.	a + b = 30
- Subtraction	Subtracts right hand operand from left hand operand.	a – b = -10
* Multiplication	Multiplies values on either side of the operator	a * b = 200
/ Division	Divides left hand operand by right hand operand	b / a = 2
% Modulus	Divides left hand operand by right hand operand and returns remainder	b % a = 0
** Exponent	Performs exponential (power) calculation on operators	a**b =10 to the power 20
// Floor Division	The division of operands where the result is the quotient in which the digits after the decimal point are removed.	9//2 = 4 and 9.0//2.0 = 4.0

Program:

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

Output:

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

Comparison (Relational) Operators

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

Operator	Description	Example
= =	If the values of two operands are equal, then the condition becomes true.	(a == b) is not true.
!=	If values of two operands are not equal, then condition becomes true.	(a != b) is true.
< >	If values of two operands are not equal, then condition becomes true.	(a <> b) is true. This is similar to != operator.
>	If the value of left operand is greater than the value of right operand, then condition becomes true.	(a > b) is not true.
<	If the value of left operand is less than the value of right operand, then condition becomes true.	(a < b) is true.
> =	If the value of left operand is greater than or equal to the value of right operand, then condition becomes true.	(a >= b) is not true.
< =	If the value of left operand is less than or equal to the value of right operand, then condition becomes true.	(a <= b) is true.

Example:

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

Output:

b is big

Assignment Operators

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

Example:

```
a=82
b=27
a += b
print a
a=25
b=12
a -= b
print a
a=24
b=4
a *= b
print a
a=4
b=6
a **= b
print a
```

Output:

```
109
13
96
4096
```

Logical Operators

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

Example:

```
a=20
b=10
c=30
if a >= b and a >= c:
   print "a is big"
elif b >= a and b >= c:
   print "b is big"
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 right operand.	a << 2 = 240 (means 1111 0000)
>> Binary Right Shift	The left operands value is moved right by the number of bits specified by the right operand.	a >> 2 = 15 (means 0000 1111)

Membership Operators

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

Operator	Description	Example
in	Evaluates to true if it finds a variable in the specified sequence and false otherwise.	x in y, here in results in a 1 if x is a member of sequence y.
not in	Evaluates to true if it does not finds a variable in the specified sequence and false otherwise.	x not in y, here not in results in a 1 if x is not a member of sequence y.

Example:

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

Output:

available

Identity Operators

Identity operators compare the memory locations of two objects.

Operator	Description	Example
is	Evaluates to true if the variables on either side of the operator point to the same object and false otherwise.	x is y, here is results in 1 if id(x) equals id(y).
is not	Evaluates to false if the variables on either side of the operator point to the same object and true otherwise.	x is not y, here is not results in 1 if id(x) is not equal to id(y).

Example:

```
a = 20
b = 20
if ( a is b ):
        print "Line 1 - a and b have same identity"
else:
        print "Line 1 - a and b do not have same identity"
if ( id(a) == id(b) ):
        print "Line 2 - a and b have same identity"
else:
        print "Line 2 - a and b do not have same identity"
```

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:

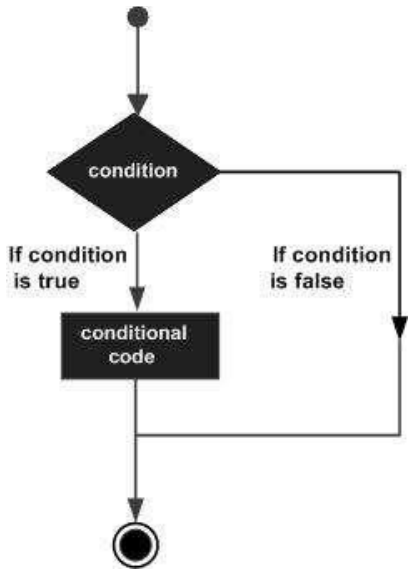
```
x= 254
y= -738
z= 1178
```

Decision Making:

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

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

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

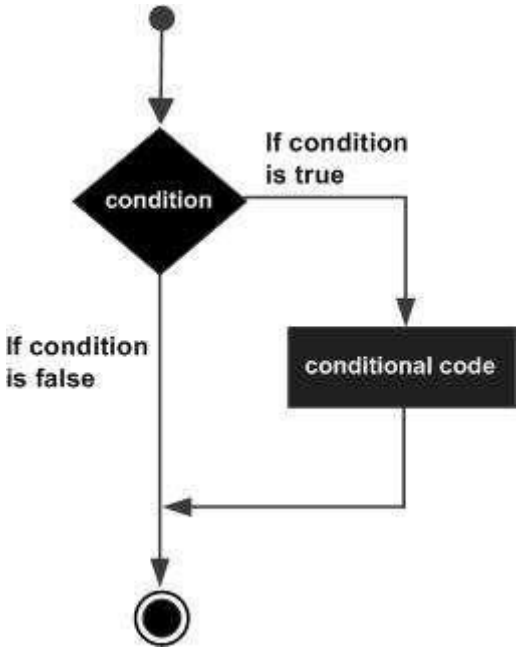


Python programming language assumes any non-zero and non-null values as True, and if it is either zero or null, then it is assumed as False value.

Statement	Description
if statements	**if statement** consists of a boolean expression followed by one or more statements.
if...else statements	**if statement** can be followed by an optional **else statement**, which executes when the boolean expression is FALSE.
nested if statements	You can use one **if** or **else if** statement inside another **if** or **else if** statement(s).

The *if* Statement

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



Syntax:

```
if condition:
   statements
```

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

Example:

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

Output:

B is big
B value is 15

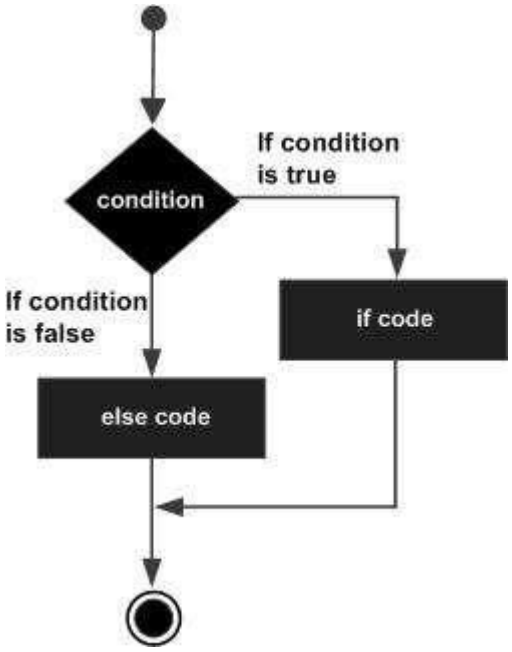
The *if ... else* statement

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

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

Syntax:

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



Example:

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

Output:

A is big
A value is 48
END

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

Program:

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

Output-1:

Enter a value: 56
a is EVEN Number

Output-2:

Enter a value: 27
a is NOT EVEN Number

The *elif* Statement

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

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

Syntax:

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

Example:

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

Output:

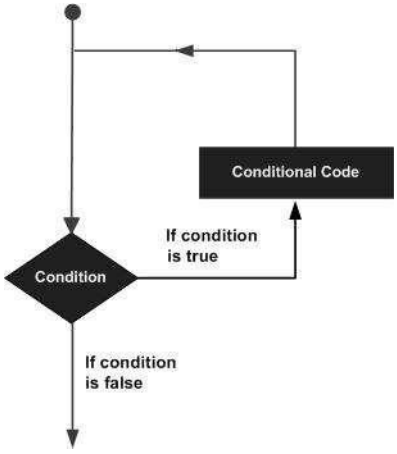
c is big

Decision Loops

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

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

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



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

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

The *while* Loop

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

Syntax

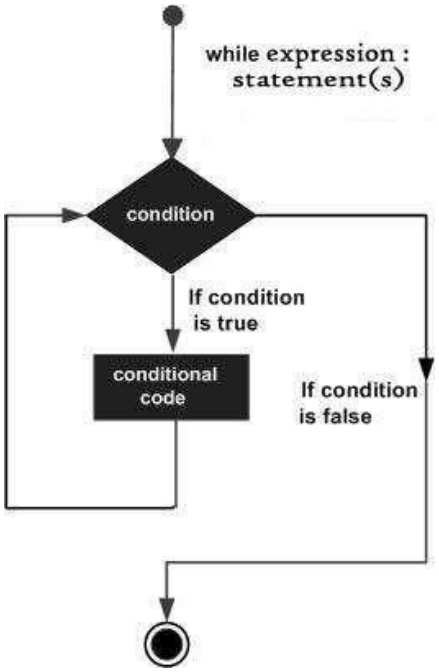
The syntax of a **while** loop in Python programming language is:

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

Here, **statement(s)** may be a single statement or a block of statements.

The **condition** may be any expression, and true is any non-zero value. The loop iterates while the condition is true. When the condition becomes false, program control passes to the line immediately following the loop.

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



Example-1:

```
i=1
while i < 4:
        print 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
print "Factorial is",f
```

Output:

Enter the number: 5
Factorial is 120

The *for* loop:

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

The syntax of the for loop is given below:

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

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

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 in name:
        print letter
```

Output-3:

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

Example-4:

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

Output-4:

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

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

Program:

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

Output:

Enter the number: 5
Factorial is 120

Nested Loop:

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

Example-1:

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

```
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 "*",
    print ""
```

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

Example-3:

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

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

Example-4:

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

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

Example-5:

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

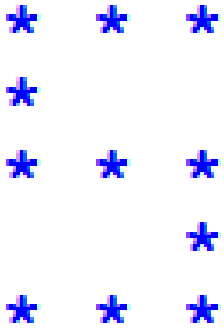
```
$ * * * *
* $     *
*   $   *
*     $ *
* * * * $
```

Example-6:

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

Example-7:

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



Example-8:

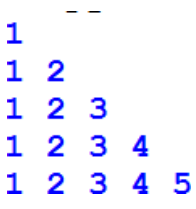
```
for i in range(1,6):
    for j in range(1,4):
        if i==1 or j==1 or i==3 or i==5:
            print "*",
        else:
            print " ",
    print ""
```

Example-9:

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

Example-10:

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



Example-11:

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

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

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

Program:

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

Output:

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

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

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

Output:

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

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

Program:

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

Output:

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

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

Program:

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

Output:

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

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

Program:

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

Output:

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

6) Write a program to display the following sequence.

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

Program:

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

7) Write a program to display the following sequence.

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

Program:

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

8) Write a program to display the following sequence.

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

Program:

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

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

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
    elif pwd[i].islower():
        l=True
    elif pwd[i].isdigit():
        d=True
if u==True and l==True and d==True:
    print pwd.center(20,"*")
else:
    print "Invalid Password"
```

Output-1:

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

Output-2:

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

10) Write a program to print sum of digits.

Program:

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

Output:

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

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

Program:

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

Output:

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

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

Program:

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

Output:

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

Arrays:

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

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

Advantages:

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

Creating an array:

Syntax:

arrayname = array(type code, [elements])

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

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

Example:

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

a = array („i", [4,8,-7,1,2,5,9])

Importing the Array Module:

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

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

import array

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

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

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

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

from array import *

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

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

Example:

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

Output:

```
4 8 -7 1 2 5 9
```

Indexing and slicing of arrays:

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

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

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

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

Example:

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

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

Output:

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

Array Methods:

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

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"
    elif ch==2:
        if len(a)==0:
            print "\t STACK IS EMPTY"
        else:
            print "Deleted element is", a.pop( )
    elif ch==3:
        if len(a)==0:
            print "\t STACK IS EMPTY"
        else:
            print "\tThe Elements in Stack is",
            for i in a:
                print i,
    elif ch==4:
        sys.exit()
    else:
        print "\tINVALID CHOICE"
```

Output:

```
1.PUSH 2.POP 3.DISPLAY 4.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)
    elif ch==2:
        if len(a)==0:
            print "\t QUEUE IS EMPTY"
        else:
            print "Deleted element is", a[0]
            a.remove(a[0])
    elif ch==3:
        if len(a)==0:
            print "\t QUEUE IS EMPTY"
        else:
            print "\tThe Elements in Queue is",
            for i in a:
                print i,
    elif ch==4:
        sys.exit()
    else:
        print "\tINVALID CHOICE"
```

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.

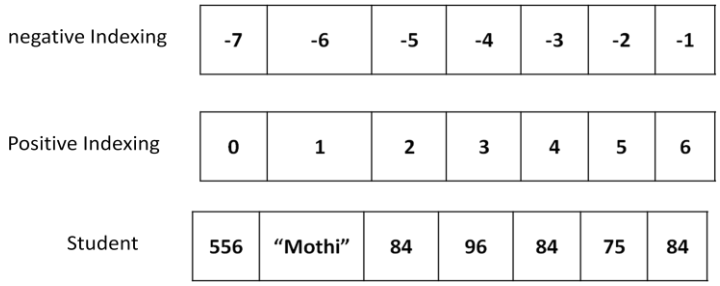
student = [556, “Mothi”, 84, 96, 84, 75, 84]

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 0^{th} 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]
print lst        # [1,2,4,5,8,6]
lst.append(9)
print lst        # [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]
print lst        # [4,7,6,8,9,3]
lst[2]=5         # updates 2nd element in the list
print lst        # [4,7,5,8,9,3]
lst[2:5]=10,11,12      # update 2nd element to 4th element in the list
print lst        # [4,7,10,11,12,3]
```

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

Example:

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

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

Concatenation of Two lists:

We can simply use „+" operator on two lists to join them. For example, „x" and „y" are two lists. If we 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:

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

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

Membership in Lists:

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

Example:

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

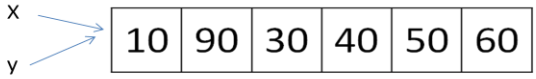
Aliasing and Cloning Lists:

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

Example:

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

Before Modifications



After Modifications

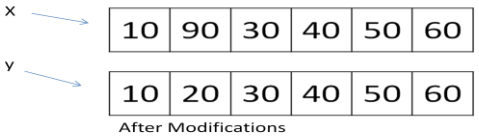
In this case we are having only one list of elements but with two different names „x" and „y". Here, „x" is the original name and „y" is the alias name for the same list. Hence, any modifications done to x" will also modify „y" and vice versa.

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

Before Modifications



After Modifications

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

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)
print lst          # [10,46,25,45,51,45,51,21,65]
print lst.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 the lists.

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

Nested Lists as Matrices:

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

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

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

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

Example:

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

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

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

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

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

```
a=[[1,2,3],[4,5,6],[7,8,9]]
b=[[4,5,6],[7,8,9],[1,2,3]]
c=[[0,0,0],[0,0,0],[0,0,0]]
m1=len(a)
n1=len(a[0])
m2=len(b)
n2=len(b[0])
for i in range(0,m1):
    for j in range(0,n1):
        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 code as:

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

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

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

The previous code can rewritten in a compact way as:

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

This is called list comprehension. From this, we can understand that a list comprehension consists of square braces containing an expression (i.e., x**2). After the expression, a 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) if x%2==0 ]
```

It will display the list even squares as list.

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

TUPLE:

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

Creating Tuples:

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

To create an empty tuple, we can simply write empty parenthesis, as:

```
tup=( )
```

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

```
tup = (10)
print tup          # display 10
print type(tup)    # display <type „int“>
```

```
tup = (10,)
print tup          # display 10
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)
print tp          # 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))
print t           # display (2,4,6,8,10)
```

Accessing the tuple elements:

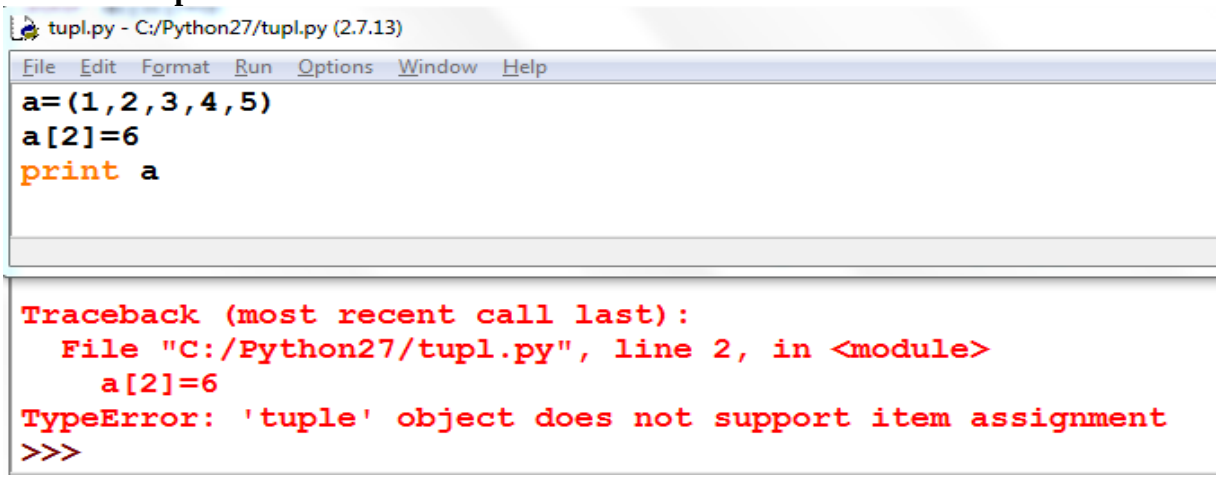
Accessing the elements from a tuple can be done using indexing or slicing. This is same as that of a list. Indexing represents the position number of the element in the tuple. The position starts from 0.

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

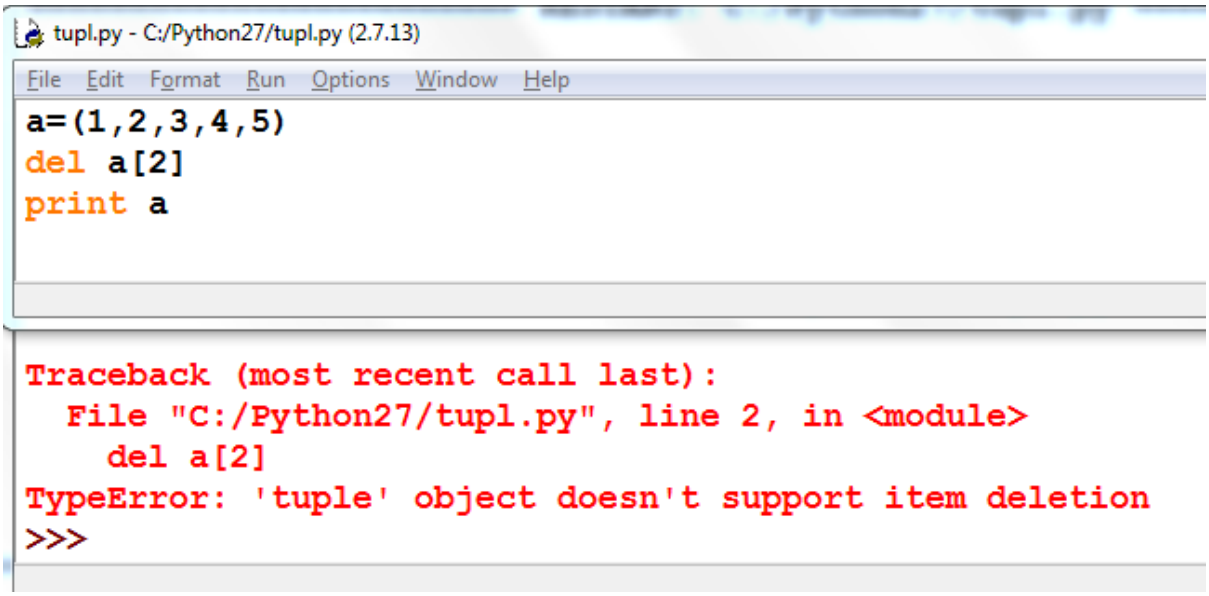
Updating and deleting elements:

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

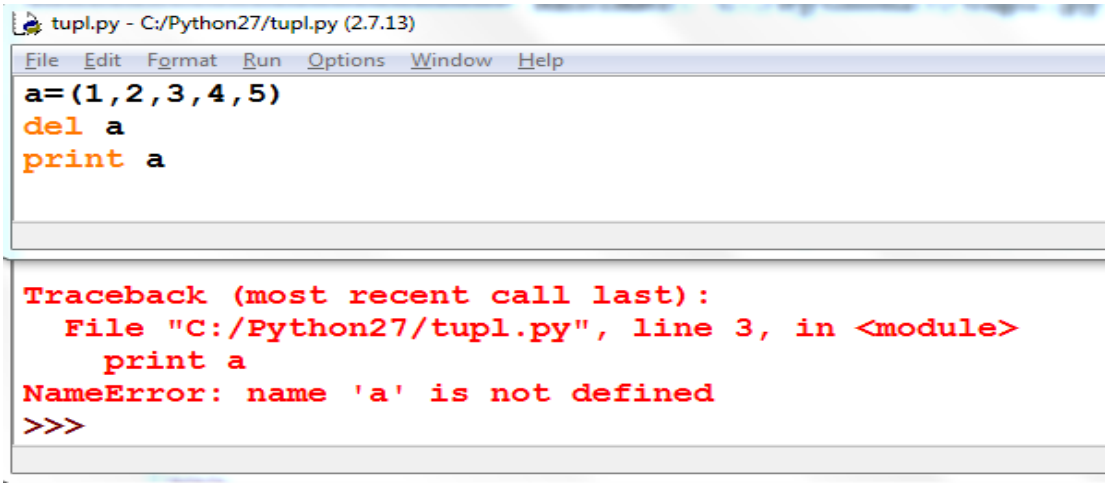
Example-1:



Example-2:



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



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

Operations on tuple:

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

cmp(tuple1, tuple2)

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

Syntax

cmp(tuple1, tuple2)

Parameters

tuple1 -- This is the first tuple to be compared

tuple2 -- This is the second tuple to be compared

Return Value

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

- If numbers, perform numeric coercion if necessary and compare.
- If either element is a number, then the other element is "larger" (numbers are "smallest").
- Otherwise, types are sorted alphabetically by name.

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

Example:

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

Creating a Set:

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

Syntax:

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

Example:

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

Converting a list into set:

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

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

We can also convert tuple or string into set.

```
tup= ( 1, 2, 3, 4, 5 )
print set(tup) # set( [ 1, 2, 3, 4, 5 ] )
str= “MOTHILAL”
print str        # 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 the set.

Note:

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

```
s=set( )                                    s={}
print type(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‟}
print d[„Regd.No‟]          # 556
print d[„Name‟]             # Mothi
print d[„Branch‟]           # CSE
```

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

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

```
d={„Regd.No‟:556,„Name‟:‟Mothi‟,„Branch‟:„CSE‟}
print len(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'}
print d    #{'Branch': 'CSE', 'Name': 'Mothi', 'Regd.No': 556}
d['Gender']="Male"
print d    # {'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* statement as:

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

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

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

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

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

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

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

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

Dictionary Methods:

Method	Description
`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] )`	Removes the key „k“ and its value from,„d“ and returns the value. If key is not found, then the value „v“ is returned. If key is not found and „v“ is not mentioned then „KeyError“ is raised.
`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 a simple dictionary that contains color code and its name as:

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

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

```
colors = { 'r':"RED", 'g':"GREEN", 'b':"BLUE", 'w':"WHITE" }
for k in colors:
        print k # displays only keys
for k in colors:
        print colors[k] # keys to to dictionary and display the values
```

Converting Lists into Dictionary:

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

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

Example:

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

Output:

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

Converting Strings into Dictionary:

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

```
s="Vijay=23,Ganesh=20,Lakshmi=19,Nikhil=22"
s1=s.split(',')
s2=[]
d={}
for i in s1:
   s2.append(i.split('='))
print d
```

{'Ganesh': '20', 'Lakshmi': '19', 'Nikhil': '22', 'Vijay': '23'}

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

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

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

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

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

FUNCTIONS:

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

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

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

Difference between a function and a method:

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

Objectname.methodname()

Classname.methodname()

Defining a Function

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

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

Syntax:

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

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

Example:

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

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

Calling Function:

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

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

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

Example:

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

Returning Results from a function:

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

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

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

Returning multiple values from a function:

A function can 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 function as:

x, y, z = functionName( )

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

Example:

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

Functions are First Class Objects:

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

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

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

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

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

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")
print x
```

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())
print fun
```

Output: haihow r u?

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

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

Output: how r u?

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 memory for which a name „x" is attached. So, 10 is the object and „x" is the name or tag given 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 modify it.

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

Output:

```
inside 15 6356456
outside 10 6356516
```

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

modify(x)

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

x=15

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

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

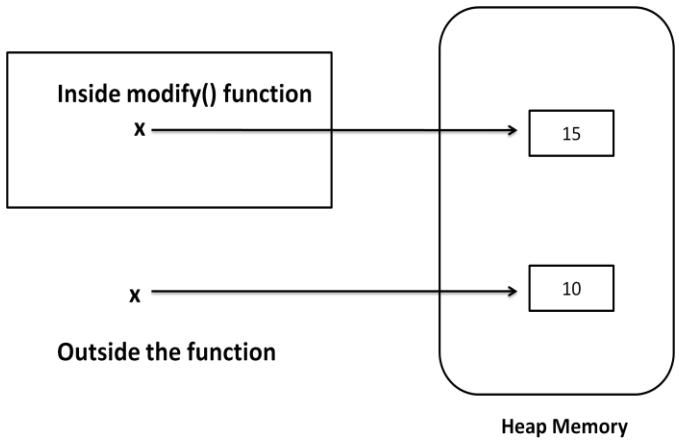


Fig. Passing Integer to a Function

Pass by Reference:

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

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

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

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

Output:

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

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

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

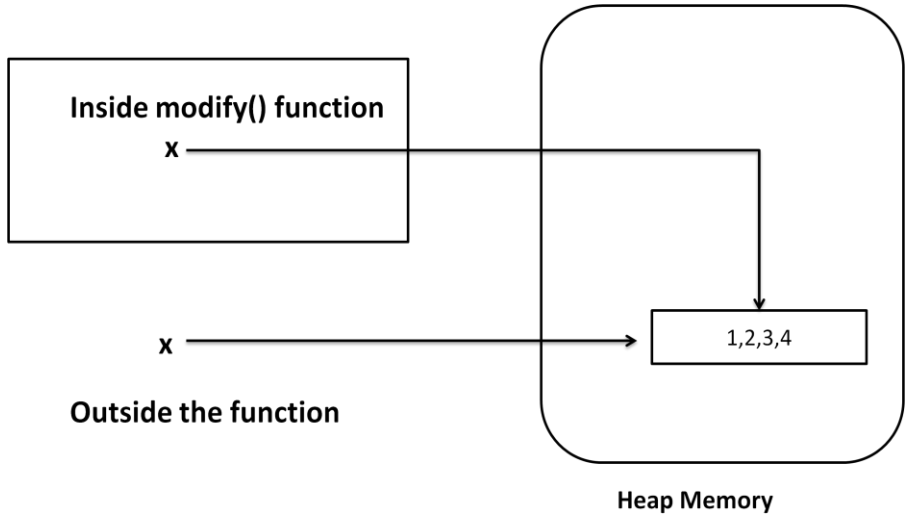


Fig. Passing a list to the function

Formal and Actual Arguments:

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

Example:

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

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

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

a) Positional Arguments:

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

```
def attach(s1,s2):
   s3=s1+s2
   print s3
attach("New","Delhi") #Positional arguments
```

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

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

b) Keyword Arguments:

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

def grocery(item, price):

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

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

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

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

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

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

Output:

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

c) Default Arguments:

We can mention some default value for the function parameters in the definition. Let"s take the definition of grocery( ) function as:

def grocery(item, price=40.00)

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

Example:

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

Output:

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

d) Variable Length Arguments:

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 is 15
sum is 35
sum is 65
```

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 function 11
Inside function 10
outside function 11
outside function
NameError: name 'b' is not defined
```

The Global Keyword:

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

Example-1:

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

Output:

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

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

global a

Example-2:

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

Output:

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

Recursive Functions:

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

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

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

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

Anonymous Function or Lambdas:

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

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

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

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

the same function can be written as anonymous function as:

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

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

```
lambda argument_list: expression
```

Example:

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

The map() Function

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

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

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

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

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

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

```
>>> print C
[39.200000000000003, 36.5, 37.300000000000004, 37.799999999999997]
```

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

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

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

Filtering

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

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

Reducing a List

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

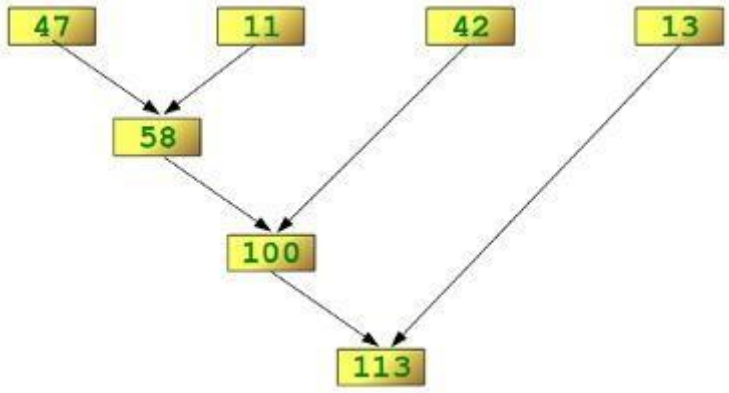
If seq = [s_1, s_2, s_3, ... , s_n], calling reduce(func, seq) works like this:

- 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 a function.

The following steps are generally involved in creation of decorators:

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

Example-1:

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

Output:

```
12
```

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

Example-2: A python program to create two decorators.

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

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

Output:

```
22
```

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

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

print num()
```

Output:

```
22
```

Function Generators:

A generator is a function that produces a sequence of results instead of a single value. „yield" statement is used to return the value.

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

Output:

```
0 1 2 3 4 5
```

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

Modules:

A module is a file containing Python definitions and statements. The file name is the module name with the suffix.py appended. Within a module, the module"s name (as a string) is available as the value of the global variable __name__. For instance, use your favourite text editor to create a file called fibo.py in the current directory with the following contents:

```
# Fibonacci numbers module
def fib(n): # write Fibonacci series up to n
    a, b = 0, 1
    while b < n:
        print b,
        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 fibo directly in the current symbol table; it only enters the module name fibo there. Using the module name you can access the functions:

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

from statement:

- A module can contain executable statements as well as function definitions. These statements are intended to initialize the module. They are executed only the first time the module name is encountered in an import statement. (They are also run if the file is executed as a script.)
- 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. For example:

```
>>> 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 global variable.
- Each function has its own local namespace. Class methods follow the same scoping rule as ordinary functions.
- Python makes educated guesses on whether variables are local or global. It assumes that any variable assigned a value in a function is local.
- Therefore, in order to assign a value to a global variable within a function, you must first use the global statement.
- The statement *global VarName* tells Python that *VarName* is a global variable. Python stops searching the local namespace for the variable.
- For example, we define a variable *Money* in the global namespace. Within the function*Money*, we assign *Money* a value, therefore Python assumes *Money* as a local variable. However, we accessed the value of the local variable *Money* before setting it, so an UnboundLocalError is the result. Uncommenting the global statement fixes the problem.

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 requests Package:**
  - ∝ **Command:** cd C:\Python27\Scripts
  - ∝ **Command:** pip install requests

- **Installation of flask Package:**
  - ∝ **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
```

There are some libraries in python:

- **Requests**: The most famous HTTP Library. It is a must and an essential criterion for every Python Developer.
- **Scrapy**: If you are involved in webscripting then this is a must have library for you. After using this library you won‟t use any other.
- **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 with images.
- **SQLAchemy**: It is a database library.
- **BeautifulSoup**: This xml and html parsing library.
- **Twisted**: The most important tool for any network application developer.
- **NumPy**: It provides some advanced math functionalities to python.
- **SciPy**: It is a library of algorithms and mathematical tools for python and has caused many scientists to switch from ruby to python.
- **Matplotlib**: It is a numerical plotting library. It is very useful for any data scientist or any data analyzer.

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

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

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

Overview of OOP Terminology

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

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.)

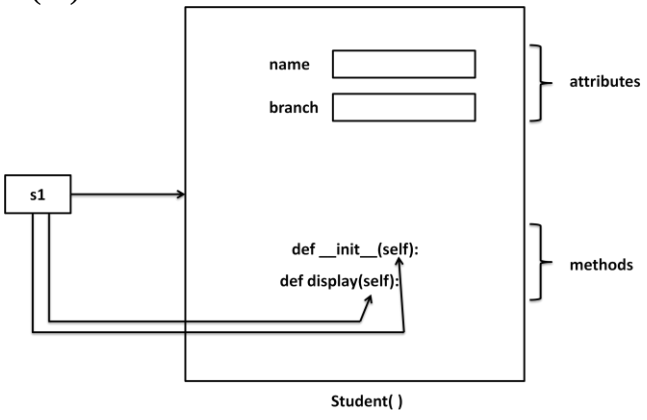
Example:

```
class Student:
    def __init__(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 any student.
- 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 method explicitly.
- Observe the parameter „self‟ written after the method name in the parentheses. „self‟ is a variable that refers to current class instance.
- 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 the memory.
- 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 place internally:
  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 Student class.
  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:

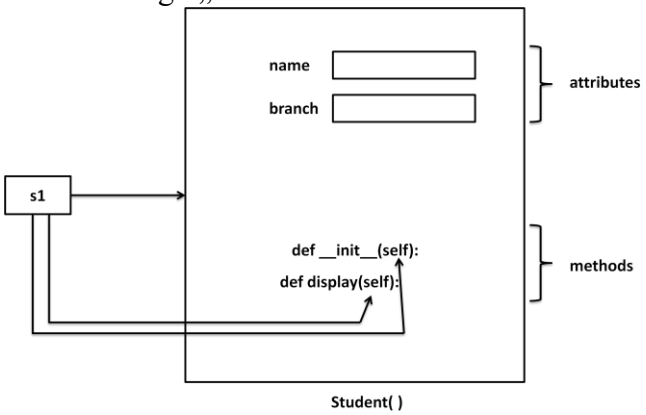
- The self variable is used as first parameter in the constructor as:

  def_init_(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 methods as:

  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.

```
def_init_( 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 addition to „self" as:

```
def_init_( 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 Student instance.

Example:

```
class Student:
   def init (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 Static Variables

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 instance variables.

```
class Sample:
   def init (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 Static Variables

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, the same copy of „x‟ is passed to these 3 instances. When we modify the copy of „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 static variables.

```
class Sample:
   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.

Suppose we write: 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.

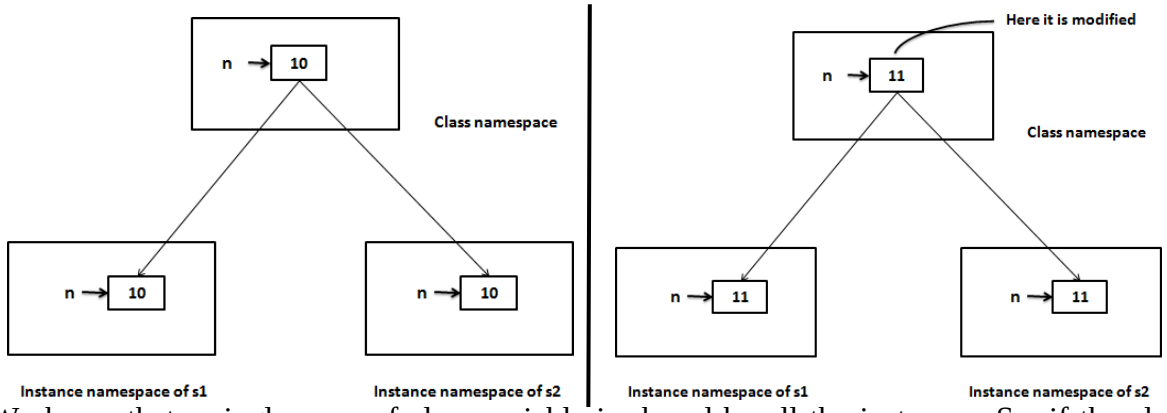
a) Class Namespace:

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 gives 10.

Example:

```
class Student:
        n = 10
print Student.n          # displays 10
Student.n += 1
print Student.n          # displays 11
s1 = Student( )
print s1.n               # displays 11
s2 = Student( )
print s2.n               # displays 11
```

Before modifying the class variable „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:

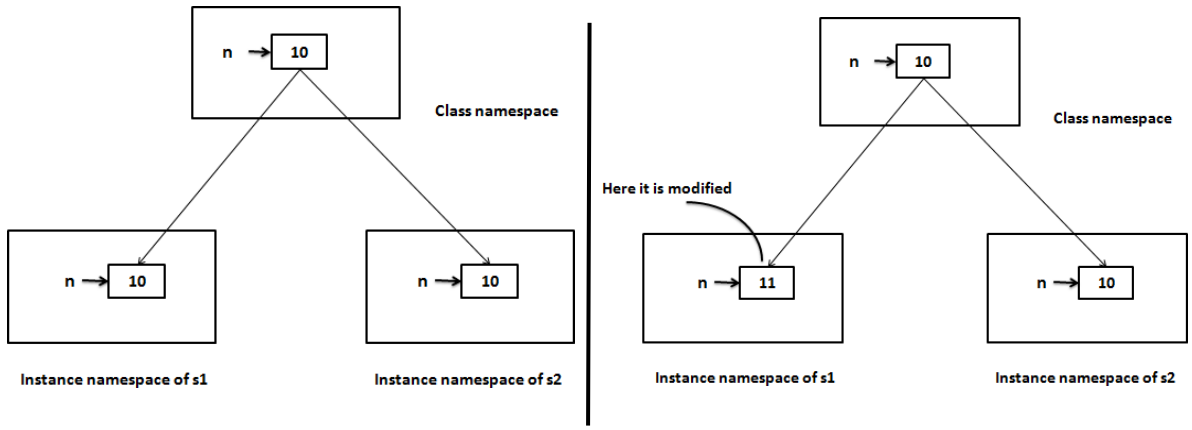
Every instance will have its own name space, called „instance namespace‟. In the instance namespace, the names are mapped to instance variables. Every instance will have its own namespace, if the class variable is modified in one instance namespace, it will not affect the variables in the other instance 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:

```
class Student:
        n = 10
s1 = Student( )
print s1.n               # displays 10
s1.n += 1
print s1.n               # displays 11
```

```
s2 = Student( )
print s2.n                # displays 11
```

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



Types of methods:

We can classify the methods in the following 3 types:

a) Instance methods
   - Accessor methods
   - Mutator methods

b) Class methods

c) Static methods

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:
    def_init_(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 mutator methods.
- 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) Class methods:

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,"flies with",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 a class.

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
   def init (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 develop software.
- 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 own class.
- 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 child class.
- A child class can also override data members and methods from the parent.

Syntax:

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

- When an object is to SubClass is created, it contains a copy of BaseClass within it. This means there is a relation between the BaseClass and SubClass objects.
- We do not create BaseClass object,but still a copy of it is available to SubClass object.
- 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 of time.
- If the programmer used inheritance, he will be able to develop more code in less time.
- 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 sub classes.

The super( ) method:

- super( ) is a built-in method which is useful to call the super class constructor or methods from the sub class.
- 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 class constructor.
- super( ) is a built-in method which contains the history of super class methods.
- Hence, we can use super( ) to refer to super class constructor and methods from a aub class. So, super( ) can be used as:

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

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

```
class Father:
    def_init_(self, p = 0):
        self.property = p
    def display(self):
        print "Father Property",self.property
class Son(Father):
    def_init_(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:
    def_init_(self, x = 0):
        self.x = x
    def area(self):
        print "Area of square", self.x * self.x
class Rectangle(Square):
    def_init_(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

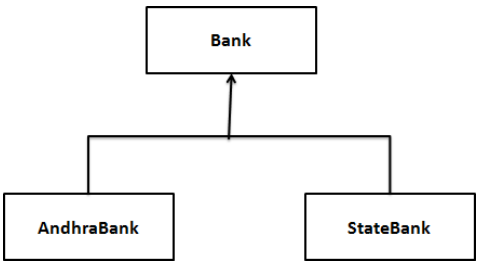
Types of Inheritance:

There are mainly 2 types of inheritance.

a) Single inheritance
b) Multiple inheritance

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.



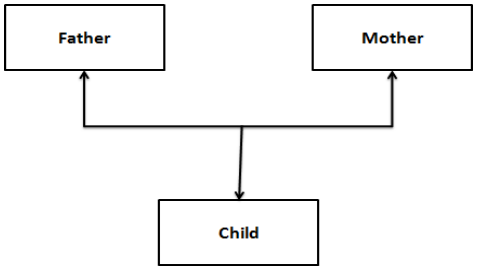
Example:

```
class Bank:
    cash = 100
    @classmethod
    def balance(cls):
        print cls.cash
class AndhraBank(Bank):
    cash = 500
    @classmethod
    def balance(cls):
        print "AndhraBank",cls.cash + Bank.cash
class StateBank(Bank):
    cash = 300
    @classmethod
    def balance(cls):
        print "StateBank",cls.cash +
Bank.cash a=AndhraBank()
a.balance()                    # displays AndhraBank 600
s=StateBank()
s.balance()                    #displays StateBank 400
```

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:
    def color(self):
        print "Color is brown"
class Child(Father, Mother):
    pass
c=Child()
c.height() # displays Height is 5.8 incehs
c.color() # displays Color is brown
```

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 sub class.
- But writing constructor is very common to initialize the instance variables.
- 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 its constructor.

Example-1:

```
class A(object):
    def_init_(self):
        print "Class A"
class B(object):
    def___init__(self):
        print "Class B"
class C(A,B,object):
    def_init_(self):
        super()._init_()
        print "Class C"
c1= C()
```

Output:

```
Class A
Class C
```

Example-2:

```
class A(object):
    def_init_(self):
        super()._init_()
        print "Class A"
class B(object):
    def___init__(self):
        super().__init__()
        print "Class B"
class C(A,B,object):
    def_init_(self):
        super()._init_()
        print "Class C"
c1= C()
```

Output:

```
Class B
Class A
Class 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 to outsiders.

Example:

```
class JustCounter:
    __secretCount = 0
    def count(self):
        self._secretCount += 1
        print self._secretCount
counter = JustCounter()
counter.count()
counter.count()
print counter._secretCount
```

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

```
1
2

Traceback (most recent call last):
  File "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._className_attrName*. If you would replace your last line as following, then it works for you:

```
.....................
print counter._JustCounter_secretCount
```

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

```
1
2
2
```

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. The errors in the software are called „bugs" and the process of removing them are called „debugging". In general, we can classify errors in a program into one of these three types:

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

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 at runtime.

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) Logical errors

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
```

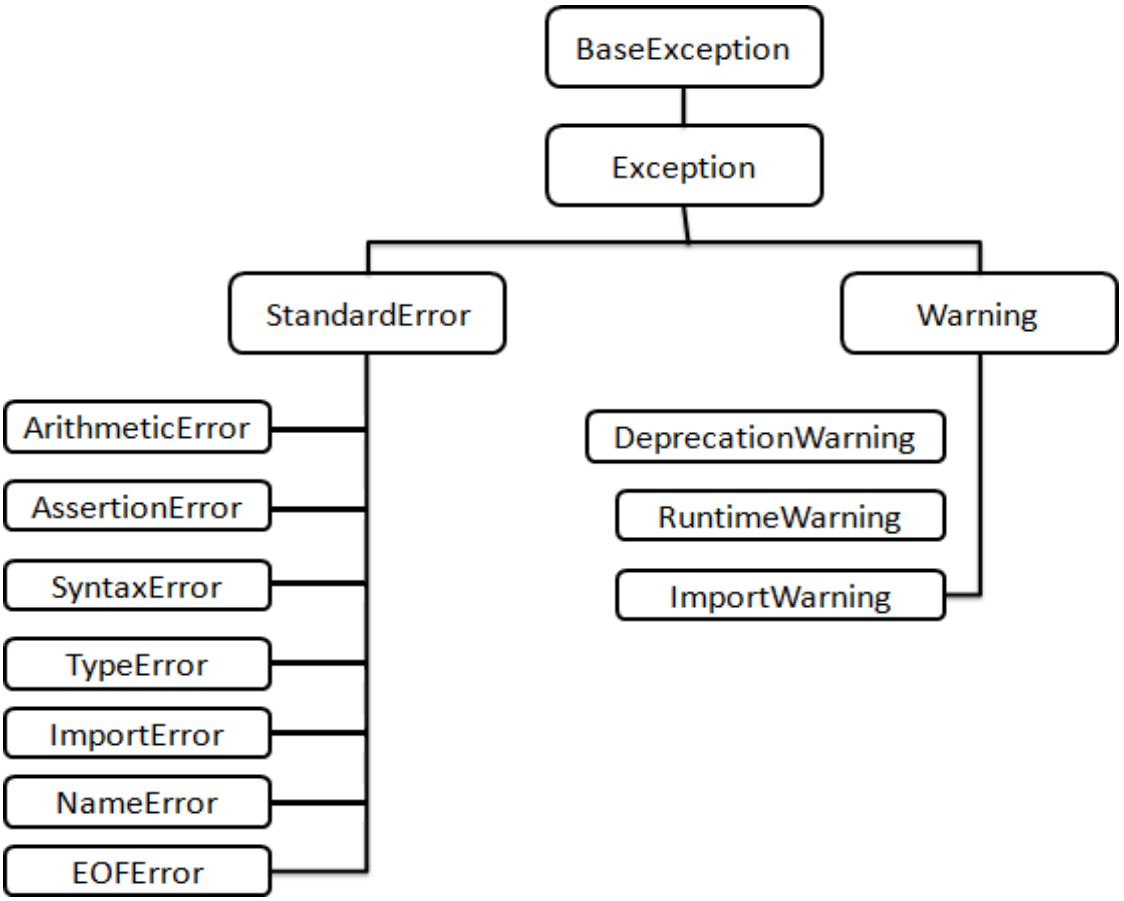
From the above program the formula for salary is wrong, because only the increment but it is not adding it to the original salary. So, the correct formula would be:

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

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

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 something to eliminate the harm caused by that error, then it is called an „exception".
- If the programmer cannot do anything in case of an error, then it is called an „error" and not an exception.
- 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" are derived.
- All errors (or exceptions) are defined as sub classes of StandardError. An error should be compulsory handled otherwise the program will not execute.
- 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 cannot neglect.
- Just like the exceptions which are already available in python language, a programmer can also create his own exceptions, called „user-defined" exceptions.
- When the programmer wants to create his own exception class, he should derive his class from Exception class and not from „BaseException" class.



Exception Handling:

- The purpose of handling errors is to make the program *robust*. The word „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 continue execution.
- Designing such programs is needed in any software development.
- For that purpose, the programmer should handle the errors. When the errors can be handled, they are called exceptions.

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 as follows:

```
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
except Exception1:
    statements
except Exception2:
    statements
else:
    statements
finally:
    statements
```

The following points are followed in exception handling:

- ✓ A single try block can be followed by several except blocks.
- ✓ Multiple except blocks can be used to handle multiple exceptions.
- ✓ We cannot write except blocks without a try block.
- ✓ We can write a try block without any except blocks.
- ✓ Else block and finally blocks are not compulsory.
- ✓ When there is no exception, else block is executed after try block.
- ✓ Finally block is always executed.

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 IOError is 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 name as:
   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 parantheses as:
   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 exceptions which are called „User-defined exceptions‟ or „Custom exceptions‟. We know 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 raise it.
- For example, let‟s take a bank where customers have accounts. Each account is characterized should by customer name and balance amount.
- The rule of the bank is that every customer should keep minimum Rs. 2000.00 as balance amount in his account.
- The programmer now is given a task to check the accounts to know every customer is maintaining minimum balance of Rs. 2000.00 or not.
- 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 the customer.
- 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 own exception.
- For this purpose, he/she has to follow these steps:

1. Since all exceptions are classes, the programme is supposed to create his own exception as a class. Also, he should make his class as a sub class to the in-built „Exception‟ class.

```
class MyException(Exception):
def init (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‟ statement as:

```
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 exception using „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):
    def __init__(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
```

Brief Tour of the Standard Library:

Python‟s standard library is very extensive, offering a wide range of facilities. The library contains built-in modules that provide access to system functionality such as I/O that would otherwise be inaccessible to the python programmers.

Operating system interface:

- The OS module in Python provides a way of using operating system dependent functionality.
- The functions that the OS module provides allows you to interface with the underlying operating system that Python is running on – be that Windows, Mac or Linux.
- You can find important information about your location or about the process.

OS functions

1. Executing a shell command
   os.system()
2. Returns the current working directory.
   os.getcwd()
3. Return the real group id of the current process.
   os.getgid()
4. Return the current process‟s user id.
   os.getuid()
5. Returns the real process ID of the current process.
   os.getpid()
6. Set the current numeric umask and return the previous umask.
   os.umask(mask)
7. Return information identifying the current operating system.
   os.uname()
8. Change the root directory of the current process to path.
   os.chroot(path)
9. Return a list of the entries in the directory given by path.
   os.listdir(path)
10. Create a directory named path with numeric mode mode.
    os.mkdir(path)
11. Remove (delete) the file path.
    os.remove(path)
12. Remove directories recursively.
    os.removedirs(path)
13. Rename the file or directory src to dst.
    os.rename(src, dst)

String Pattern Matching:

The **re** module provides regular expression tools for advanced string processing. For complex matching and manipulation, the regular expressions offer succinct, optimized solutions.

re Functions:

1. match Function

re.match(pattern, string, flags=0)

Here is the description of the parameters:

Parameter	Description
Pattern	This is the regular expression to be matched.
String	This is the string, which would be searched to match the pattern at the beginning of string.
Flags	You can specify different flags using bitwise OR (|). These are modifiers, which are listed in the table below.

The *re.match* function returns a **match** object on success, **None** on failure. We use*group(num)* or *groups()* function of **match** object to get matched expression.

Match Object Methods	Description
group(num=0)	This method returns entire match (or specific subgroup num)
groups()	This method returns all matching subgroups in a tuple (empty if there weren't any)

```
import re

line = "Cats are smarter than dogs"

matchObj = re.match( r'(.*) are (.*?) .*', line, re.M|re.I)

if matchObj:
   print "matchObj.group() : ", matchObj.group()
   print "matchObj.group(1) : ", matchObj.group(1)
   print "matchObj.group(2) : ", matchObj.group(2)
else:
   print "No match!!"
```

Output:

```
matchObj.group() : Cats are smarter than dogs
matchObj.group(1) :  Cats
matchObj.group(2) :  smarter
```

2. *search* Function

This function searches for first occurrence of RE *pattern* within *string* with optional *flags*.

Here is the syntax for this function:

re.search(pattern, string, flags=0)

Here is the description of the parameters:

Parameter	Description
pattern	This is the regular expression to be matched.
string	This is the string, which would be searched to match the pattern anywhere in the string.
flags	You can specify different flags using bitwise OR (|). These are modifiers, which are listed in the table below.

The *re.search* function returns a **match** object on success, **none** on failure. We use *group(num)* or *groups()* function of **match** object to get matched expression.

Match Object Methods	Description
group(num=0)	This method returns entire match (or specific subgroup num)
groups()	This method returns all matching subgroups in a tuple (empty if there weren't any)

3. sub function:

One of the most important **re** methods that use regular expressions is sub.

re.sub(pattern, repl, string, max=0)

This method replaces all occurrences of the RE pattern in string with repl, substituting all occurrences unless max provided. This method returns modified string.

Example:

```
import re

phone = "2004-959-559 # This is Phone Number"

# Delete Python-style comments
num = re.sub(r'#.*$', "", phone)
print "Phone Num : ", num

# Remove anything other than digits
num = re.sub(r'\D', "", phone)
print "Phone Num : ", num
```

Output: Phone Num : 2004-959-559
Phone Num : 2004959559

Regular Expression Patterns

Except for control characters, **(+ ? . * ^ $ ( ) [ ] { } | \)**, all characters match themselves. You can escape a control character by preceding it with a backslash.

Following table lists the regular expression syntax that is available in Python −

Pattern	Description
^	Matches beginning of line.
$	Matches end of line.
.	Matches any single character except newline. Using m option allows it to match newline as well.
[...]	Matches any single character in brackets.
[^...]	Matches any single character not in brackets
re*	Matches 0 or more occurrences of preceding expression.
re+	Matches 1 or more occurrence of preceding expression.
re?	Matches 0 or 1 occurrence of preceding expression.
re{ n}	Matches exactly n number of occurrences of preceding expression.
re{ n,}	Matches n or more occurrences of preceding expression.

re{ n, m}	Matches at least n and at most m occurrences of preceding expression.
a| b	Matches either a or b.
(?#...)	Comment.
(?= re)	Specifies position using a pattern. Doesn't have a range.
(?! re)	Specifies position using pattern negation. Doesn't have a range.
(?> re)	Matches independent pattern without backtracking.
\w	Matches word characters.
\W	Matches nonword characters.
\s	Matches whitespace. Equivalent to [\t\n\r\f].
\S	Matches nonwhitespace.
\d	Matches digits. Equivalent to [0-9].
\D	Matches nondigits.
\A	Matches beginning of string.
\Z	Matches end of string. If a newline exists, it matches just before newline.
\z	Matches end of string.
\G	Matches point where last match finished.
\B	Matches nonword boundaries.
\n, \t, etc.	Matches newlines, carriage returns, tabs, etc.
\1...\9	Matches nth grouped subexpression.
\10	Matches nth grouped subexpression if it matched already. Otherwise refers to the octal representation of a character code.

Mathematical Functions:

The **math** module is a standard module in Python and is always available. To use mathematical functions under this module, you have to import the module using **import math**.

Function	Description
ceil(x)	Returns the smallest integer greater than or equal to x.
copysign(x, y)	Returns x with the sign of y
fabs(x)	Returns the absolute value of x
factorial(x)	Returns the factorial of x
floor(x)	Returns the largest integer less than or equal to x
fmod(x, y)	Returns the remainder when x is divided by y
frexp(x)	Returns the mantissa and exponent of x as the pair (m, e)
fsum(iterable)	Returns an accurate floating point sum of values in the iterable
isfinite(x)	Returns True if x is neither an infinity nor a NaN (Not a Number)
ldexp(x, i)	Returns x * (2**i)
modf(x)	Returns the fractional and integer parts of x

trunc(x)	Returns the truncated integer value of x
exp(x)	Returns e**x
expm1(x)	Returns e**x - 1
log(x[, base])	Returns the logarithm of x to the base (defaults to e)
log10(x)	Returns the base-10 logarithm of x
pow(x, y)	Returns x raised to the power y
sqrt(x)	Returns the square root of x
atan2(y, x)	Returns atan(y / x)
cos(x)	Returns the cosine of x
hypot(x, y)	Returns the Euclidean norm, sqrt(x*x + y*y)
sin(x)	Returns the sine of x
tan(x)	Returns the tangent of x
degrees(x)	Converts angle x from radians to degrees
radians(x)	Converts angle x from degrees to radians
acosh(x)	Returns the inverse hyperbolic cosine of x
asinh(x)	Returns the inverse hyperbolic sine of x
atanh(x)	Returns the inverse hyperbolic tangent of x
cosh(x)	Returns the hyperbolic cosine of x
sinh(x)	Returns the hyperbolic cosine of x
tanh(x)	Returns the hyperbolic tangent of x
erf(x)	Returns the error function at x
erfc(x)	Returns the complementary error function at x
gamma(x)	Returns the Gamma function at x
lgamma(x)	Returns the natural logarithm of the absolute value of the Gamma function at x
pi	Mathematical constant, the ratio of circumference of a circle to it's diameter (3.14159...)
e	mathematical constant e (2.71828...)

Internet Access:

- Simple Mail Transfer Protocol (SMTP) is a protocol, which handles sending e-mail and routing e-mail between mail servers.
- Python provides **smtplib** module, which defines an SMTP client session object that can be used to send mail to any Internet machine with an SMTP or ESMTP listener daemon.
- Here is a simple syntax to create one SMTP object, which can later be used to send an e-mail:

```
import smtplib
smtpObj = smtplib.SMTP( [host [, port [, local_hostname]]] )
```

- Here is the detail of the parameters:
  - **host:** This is the host running your SMTP server. You can specify IP address of the host or a domain name like tutorialspoint.com. This is optional argument.
  - **port:** If you are providing *host* argument, then you need to specify a port, where SMTP server is listening. Usually this port would be 25.
  - **local_hostname**: If your SMTP server is running on your local machine, then you can specify just *localhost* as of this option.

An SMTP object has an instance method called **sendmail**, which is typically used to do the work of mailing a message. It takes three parameters:

- The *sender* - A string with the address of the sender.
- The *receivers* - A list of strings, one for each recipient.
- The *message* - A message as a string formatted as specified in the various RFCs.

Example: Write a program to send email to any mail address.

```
import smtplib
from email.mime.text import MIMEText
body="The message you want to send..... "
msg=MIMEText(body)
fromaddr="fromaddress@gmail.com"
toaddr="toaddress@gmail.com"
msg['From']=fromaddr
msg['To']=toaddr
msg['Subject']="Subject of mail"
server=smtplib.SMTP('smtp.gmail.com',587)
server.starttls()
server.login(fromaddr,"fromAddressPassword")
server.sendmail(fromaddr,toaddr,msg.as_string())
print "Mail Sent..... "
server.quit()
```

Output:

```
Mail Sent.....
```

Note: To send a mail to others you have to change "**Allow less secure apps: ON**" in from address mail. Because Google has providing security for vulnerable attacks

Dates and Times:

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

The *time* Module

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

Sr. No.	Function with Description
1	**time.ctime([secs])** Like asctime(localtime(secs)) and without arguments is like asctime( )
2	**time.localtime([secs])** Accepts an instant expressed in seconds since the epoch and returns a time-tuple t with the local time (t.tm_isdst is 0 or 1, depending on whether DST applies to instant secs by local rules).
3	**time.sleep(secs)** Suspends the calling thread for secs seconds.
4	**time.time( )** Returns the current time instant, a floating-point number of seconds since the epoch.
5	**time.clock( )** The method returns the current processor time as a floating point numberexpressed in seconds on **Unix**.
6	**time.asctime([tupletime])** Accepts a time-tuple and returns a readable 24-character string such as 'Tue Dec 11 18:07:14 2008'.

Example:

```
import time
print "time: ",time.time()
print "ctime: ",time.ctime()
time.sleep(5)
print "ctime: ",time.ctime()
print "localtime: ",time.localtime()
print "asctime: ",time.asctime( time.localtime(time.time()) )
print "clock: ",time.clock()
```

Output:

time: 1506843198.01
ctime: Sun Oct 01 13:03:18 2017
ctime: Sun Oct 01 13:03:23 2017
localtime: time.struct_time(tm_year=2017, tm_mon=10, tm_mday=1, tm_hour=13, tm_min=3, tm_sec=23, tm_wday=6, tm_yday=274, tm_isdst=0)
asctime: Sun Oct 01 13:03:23 2017
clock: 1.14090912202e-06

The *calendar* Module:

- The calendar module supplies calendar-related functions, including functions to print a text calendar for a given month or year.
- By default, calendar takes Monday as the first day of the week and Sunday as the last one. To change this, call calendar.setfirstweekday() function.

Sr. No.	Function with Description
1	**calendar.calendar(year,w=2,l=1,c=6)** Returns a multiline string with a calendar for year formatted into three columns separated by c spaces. w is the width in characters of each date; each line has length 21*w+18+2*c. l is the number of lines for each week.
2	**calendar.isleap(year)** Returns True if year is a leap year; otherwise, False.
3	**calendar.setfirstweekday(weekday)** Sets the first day of each week to weekday code weekday. Weekday codes are 0 (Monday) to 6 (Sunday).

4	**calendar.leapdays(y1,y2)** Returns the total number of leap days in the years within range(y1,y2).
5	**calendar.month(year,month,w=2,l=1)** Returns a multiline string with a calendar for month of year, one line per week plus two header lines. w is the width in characters of each date; each line has length 7*w+6. l is the number of lines for each week.

Example:

```
import calendar
print "Here it is the calendar:"
print calendar.month(2017,10)
calendar.setfirstweekday(6)
print calendar.month(2017,10)
print "Is 2017 is leap year?",calendar.isleap(2017)
print "No.of Leap days",calendar.leapdays(2000,2013)
print "1990-November-12 is",calendar.weekday(1990,11,12)
```

Output:

```
Here it is the calendar:
    October 2017
Mo Tu We Th Fr Sa Su
                   1
 2   3    4    5 6    7  8
 9  10   11 12 13  14 15
16  17   18 19 20  21 22
23  24   25 26 27  28 29
30  31

    October 2017
Su Mo Tu We Th Fr Sa
 1 2     3    4    5 6  7
 8 9    10 11 12 13 14
15 16  17 18 19 20 21
22 23  24 25 26 27 28
29 30  31

Is 2017 is leap year? False
No.of Leap days 4
1990-November-12 is 0
```

Data Compression

Common data archiving and compression formats are directly supported by the modules including: zlib, gzip, bz2, lzma, zipfile and tarfile.

Example: write a program to zip the three files into one single ".zip" file

```
import zipfile
FileNames=['README.txt','NEWS.txt','LICENSE.txt']
with zipfile.ZipFile('reportDir1.zip', 'w') as myzip:
    for f in FileNames:
        myzip.write(f)
```

Multithreading

Running several threads is similar to running several different programs concurrently, but with the following benefits:

- Multiple threads within a process share the same data space with the main thread and can therefore share information or communicate with each other more easily than if they were separate processes.
- Threads sometimes called light-weight processes and they do not require much memory overhead; they care cheaper than processes.

A thread has a beginning, an execution sequence, and a conclusion. It has an instruction pointer that keeps track of where within its context it is currently running.

- It can be pre-empted (interrupted).
- It can temporarily be put on hold (also known as sleeping) while other threads are running - this is called yielding.

Starting a New Thread

- To spawn another thread, you need to call following method available in *thread* module:
  thread.start_new_thread (function, args[, kwargs])
- This method call enables a fast and efficient way to create new threads in both Linux and Windows.
- The method call returns immediately and the child thread starts and calls function with the passed list of *agrs*. When function returns, the thread terminates.
- Here, *args* is a tuple of arguments; use an empty tuple to call function without passing any arguments. *kwargs* is an optional dictionary of keyword arguments.

Example:

```
import thread
import time
def print_time(tname,delay):
   count=0
   while count<5:
      count+=1
      time.sleep(delay)
      print tname,time.ctime(time.time())

thread.start_new_thread( print_time, ("Thread-1", 2 ) )
thread.start_new_thread( print_time, ("Thread-2", 5 ) )
```

Output:

```
Thread-1 Sun Oct 01 22:15:08 2017
Thread-1 Sun Oct 01 22:15:10 2017
Thread-2 Sun Oct 01 22:15:11 2017
Thread-1 Sun Oct 01 22:15:12 2017
Thread-1 Sun Oct 01 22:15:14 2017
Thread-1Thread-2 Sun Oct 01 22:15:16 2017Sun Oct 01 22:15:16 2017
Thread-2 Sun Oct 01 22:15:21 2017
Thread-2 Sun Oct 01 22:15:26 2017
Thread-2 Sun Oct 01 22:15:31 2017
```

The Threading Module:

The *threading* module exposes all the methods of the *thread* module and provides some additional methods:

- **threading.activeCount():** Returns the number of thread objects that are active.
- **threading.currentThread():** Returns the number of thread objects in the caller's thread control.
- **threading.enumerate():** Returns a list of all thread objects that are currently active.

In addition to the methods, the threading module has the *Thread* class that implements threading. The methods provided by the *Thread* class are as follows:

- **run():** The run() method is the entry point for a thread.
- **start():** The start() method starts a thread by calling the run method.
- **join([time]):** The join() waits for threads to terminate.
- **isAlive():** The isAlive() method checks whether a thread is still executing.
- **getName():** The getName() method returns the name of a thread.
- **setName():** The setName() method sets the name of a thread.

Creating Thread Using *Threading* Module:

To implement a new thread using the threading module, you have to do the following:

- Define a new subclass of the *Thread* class.
- Override the *init_(self [,args])* method to add additional arguments.
- Then, override the run(self [,args]) method to implement what the thread should do when started.

Once you have created the new *Thread* subclass, you can create an instance of it and then start a new thread by invoking the *start()*, which in turn calls *run()* method.

Example:

```
import threading
import time
exitFlag = 0
class myThread (threading.Thread):
   def init (self, threadID, name, counter):
      threading.Thread. init (self)
      self.threadID = threadID
      self.name = name
      self.counter = counter
   def run(self):
      print "Starting " + self.name
      print_time(self.name, self.counter, 5)
      print "Exiting " + self.name
def print_time(threadName, delay, counter):
   while counter:
      if exitFlag:
         thread.exit()
      time.sleep(delay)
      print threadName, time.ctime(time.time())
```

```
            counter -= 1
    # Create new threads
    thread1 = myThread(1, "Thread-1", 1)
    thread2 = myThread(2, "Thread-2", 2)

    # Start new Threads
    thread1.start()
    thread2.start()
    print "Exiting Main Thread"
```

Output:

```
Starting Thread-1Starting Thread-2Exiting Main Thread
Thread-1 Sun Oct 01 22:26:17 2017
Thread-1 Sun Oct 01 22:26:18 2017
Thread-2 Sun Oct 01 22:26:18 2017
Thread-1 Sun Oct 01 22:26:19 2017
Thread-1Thread-2 Sun Oct 01 22:26:20 2017Sun Oct 01 22:26:20 2017

Thread-1 Sun Oct 01 22:26:21 2017
Exiting Thread-1
Thread-2 Sun Oct 01 22:26:22 2017
Thread-2 Sun Oct 01 22:26:24 2017
Thread-2 Sun Oct 01 22:26:26 2017
Exiting Thread-2
```

Synchronizing Threads

- The threading module provided with Python includes a simple-to-implement locking mechanism that allows you to synchronize threads. A new lock is created by calling the *Lock()* method, which returns the new lock.
- The *acquire(blocking)* method of the new lock object is used to force threads to run synchronously. The optional *blocking* parameter enables you to control whether the thread waits to acquire the lock.
- If *blocking* is set to 0, the thread returns immediately with a 0 value if the lock cannot be acquired and with a 1 if the lock was acquired. If blocking is set to 1, the thread blocks and waits for the lock to be released.
- The *release()* method of the new lock object is used to release the lock when it is no longer required.

Example:

```
    import threading
    import time
    exitFlag = 0
    class myThread (threading.Thread):
       def init (self, threadID, name, counter):
          threading.Thread. init (self)
          self.threadID = threadID
          self.name = name
          self.counter = counter
```

```
    def run(self):
        print "Starting " + self.name
        threadLock.acquire()
        print_time(self.name, self.counter, 5)
        threadLock.release()
        print "Exiting " + self.name
def print_time(threadName, delay, counter):
    while counter:
        if exitFlag:
            thread.exit()
        time.sleep(delay)
        print threadName, time.ctime(time.time())
        counter -= 1

threadLock = threading.Lock()
threads = []
# Create new threads
thread1 = myThread(1, "Thread-1", 1)
thread2 = myThread(2, "Thread-2", 2)
# Start new Threads
thread1.start()
thread2.start()

threads.append(thread1)
threads.append(thread2)
# wait for all threads to complete
for t in threads:
    t.join()
print "Exiting Main Thread"
```

Output:

```
Starting Thread-1Starting Thread-2
Thread-1 Sun Oct 01 22:32:54 2017
Thread-1 Sun Oct 01 22:32:55 2017
Thread-1 Sun Oct 01 22:32:56 2017
Thread-1 Sun Oct 01 22:32:57 2017
Thread-1 Sun Oct 01 22:32:58 2017
Exiting Thread-1
Thread-2 Sun Oct 01 22:33:00 2017
Thread-2 Sun Oct 01 22:33:02 2017
Thread-2 Sun Oct 01 22:33:04 2017
Thread-2 Sun Oct 01 22:33:06 2017
Thread-2 Sun Oct 01 22:33:08 2017
Exiting Thread-2
Exiting Main Thread
```

GUI Programming

Python provides various options for developing graphical user interfaces (GUIs). Most important are listed below:

- **Tkinter:** Tkinter is the Python interface to the Tk GUI toolkit shipped with Python..
- **wxPython:** This is an open-source Python interface for wxWindows **http://wxpython.org**.
- **JPython:** JPython is a Python port for Java which gives Python scripts seamless access to Java class libraries on the local machine **http://www.jython.org**.

There are many other interfaces available, which you can find them on the net.

Tkinter Programming

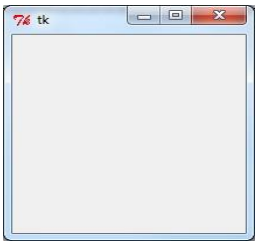
Tkinter is the standard GUI library for Python. Python when combined with Tkinter provides a fast and easy way to create GUI applications. Tkinter provides a powerful object-oriented interface to the Tk GUI toolkit.

Creating a GUI application using Tkinter is an easy task. All you need to do is perform the following steps:

- ✓ Import the *Tkinter* module.
- ✓ Create the GUI application main window.
- ✓ Add one or more of the above-mentioned widgets to the GUI application.
- ✓ Enter the main event loop to take action against each event triggered by the user.

Example:

```
import  Tkinter
top = Tkinter.Tk()
# Code to add widgets will go here...
top.mainloop()
```



Tkinter Widgets

- Tkinter provides various controls, such as buttons, labels and text boxes used in a GUI application. These controls are commonly called widgets.
- There are currently 15 types of widgets in Tkinter. We present these widgets as well as a brief description in the following table:

Operator	Description
Button	The Button widget is used to display buttons in your application.
Canvas	The Canvas widget is used to draw shapes, such as lines, ovals, polygons and rectangles, in your application.
Checkbutton	The Checkbutton widget is used to display a number of options as checkboxes. The user can select multiple options at a time.
Entry	The Entry widget is used to display a single-line text field for accepting values from a user.
Frame	The Frame widget is used as a container widget to organize other widgets.
Label	The Label widget is used to provide a single-line caption for other widgets. It can also contain images.
Listbox	The Listbox widget is used to provide a list of options to a user.
Menubutton	The Menubutton widget is used to display menus in your application.
Menu	The Menu widget is used to provide various commands to a user. These commands are contained inside Menubutton.

Message	The Message widget is used to display multiline text fields for accepting values from a user.
Radiobutton	The Radiobutton widget is used to display a number of options as radio buttons. The user can select only one option at a time. Scale The Scale widget is used to provide a slider widget.
Scrollbar	The Scrollbar widget is used to add scrolling capability to various widgets, such as list boxes.
Text	The Text widget is used to display text in multiple lines.
Toplevel	The Toplevel widget is used to provide a separate window container.
Spinbox	The Spinbox widget is a variant of the standard Tkinter Entry widget, which can be used to select from a fixed number of values.
PanedWindow	A PanedWindow is a container widget that may contain any number of panes, arranged horizontally or vertically.
LabelFrame	A labelframe is a simple container widget. Its primary purpose is to act as a spacer or container for complex window layouts.
tkMessageBox	This module is used to display message boxes in your applications.

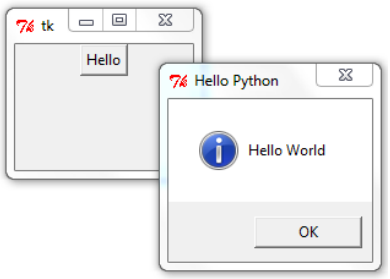
Button:

The Button widget is used to add buttons in a Python application. These buttons can display text or images that convey the purpose of the buttons. You can attach a function or a method to a button which is called automatically when you click the button.

Example:

```
import Tkinter
import tkMessageBox
top = Tkinter.Tk()
def helloCallBack():
    tkMessageBox.showinfo( "Hello Python", "Hello World")
B = Tkinter.Button(top, text ="Hello", command = helloCallBack)
B.pack()
top.mainloop()
```

Output:



Entry

The Entry widget is used to accept single-line text strings from a user.

- If you want to display multiple lines of text that can be edited, then you should use the *Text* widget.
- If you want to display one or more lines of text that cannot be modified by the user, then you should use the *Label* widget.

Example:

```
from Tkinter import *
top = Tk()
L1 = Label(top, text="User Name")
L1.pack( side = LEFT)
E1 = Entry(top, bd =5)
E1.pack(side = RIGHT)
top.mainloop()
```

Output:

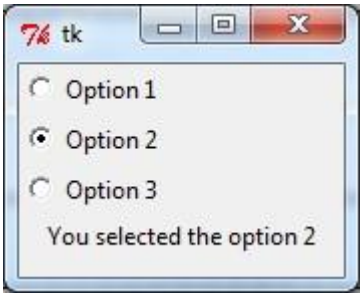
Radiobutton

- This widget implements a multiple-choice button, which is a way to offer many possible selections to the user and lets user choose only one of them.
- In order to implement this functionality, each group of radiobuttons must be associated to the same variable and each one of the buttons must symbolize a single value. You can use the Tab key to switch from one radionbutton to another.

Example:

```
from Tkinter import *
def sel():
   selection = "You selected the option " + str(var.get())
   label.config(text = selection)
root = Tk()
var = IntVar()
R1 = Radiobutton(root,text="Option 1",variable=var,value=1,command=sel)
R1.pack( anchor = W )
R2 = Radiobutton(root,text="Option 2",variable=var,value=2,command=sel)
R2.pack( anchor = W )
R3 = Radiobutton(root,text="Option 3",variable=var,value=3,command=sel)
R3.pack( anchor = W)
label = Label(root)
label.pack()
root.mainloop()
```

Output:



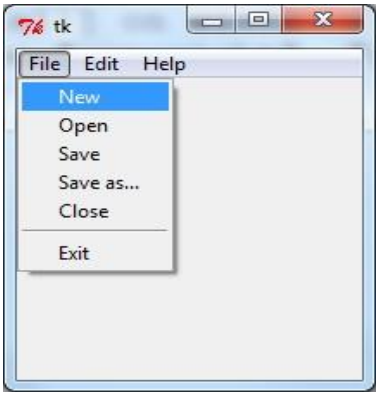
Menu

- The goal of this widget is to allow us to create all kinds of menus that can be used by our applications. The core functionality provides ways to create three menu types: pop-up, toplevel and pull-down.
- It is also possible to use other extended widgets to implement new types of menus, such as the *OptionMenu* widget, which implements a special type that generates a pop-up list of items within a selection.

Example:

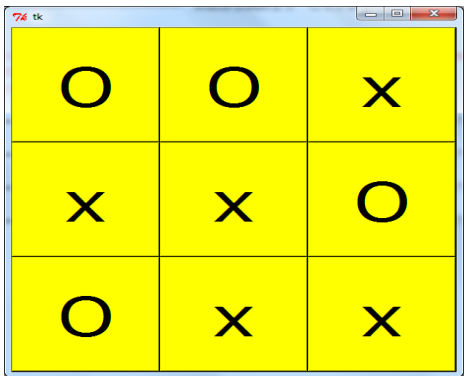
```
from Tkinter import *
def donothing():
   filewin = Toplevel(root)
   button = Button(filewin, text="Do nothing button")
   button.pack()

root = Tk()
menubar = Menu(root)

filemenu = Menu(menubar, tearoff=0)
filemenu.add_command(label="New", command=donothing)
filemenu.add_command(label="Open", command=donothing)
filemenu.add_command(label="Save", command=donothing)
filemenu.add_command(label="Save as...", command=donothing)
filemenu.add_command(label="Close", command=donothing)
filemenu.add_separator()
filemenu.add_command(label="Exit", command=root.quit)
menubar.add_cascade(label="File", menu=filemenu)

editmenu = Menu(menubar, tearoff=0)
editmenu.add_command(label="Undo", command=donothing)
editmenu.add_separator()
editmenu.add_command(label="Cut", command=donothing)
editmenu.add_command(label="Copy", command=donothing)
editmenu.add_command(label="Paste", command=donothing)
editmenu.add_command(label="Delete", command=donothing)
editmenu.add_command(label="Select All", command=donothing)
menubar.add_cascade(label="Edit", menu=editmenu)

helpmenu = Menu(menubar, tearoff=0)
helpmenu.add_command(label="Help Index", command=donothing)
helpmenu.add_command(label="About...", command=donothing)
menubar.add_cascade(label="Help", menu=helpmenu)

root.config(menu=menubar)
root.mainloop()
```

Output:



Example: Write a program for Tic-Tac-Toe Game

```
from Tkinter import *
def callback(r,c):
    global player
    if player=='X' and states[r][c]==0:
        b[r][c].configure(text='x')
        states[r][c]='X'
        player='O'
    if player=='O' and states[r][c]==0:
        b[r][c].configure(text='O')
        states[r][c]='O'
        player='X'
root=Tk()
states=[[0,0,0],[0,0,0],[0,0,0]]
b=[[0,0,0],[0,0,0],[0,0,0]]
for i in range(3):
    for j in range(3):

b[i][j]=Button(font=('verdana',56),width=3,bg='yellow',command=lambda
r=i,c=j:callback(r,c))
        b[i][j].grid(row=i,column=j)
player='X'
root.mainloop()
```

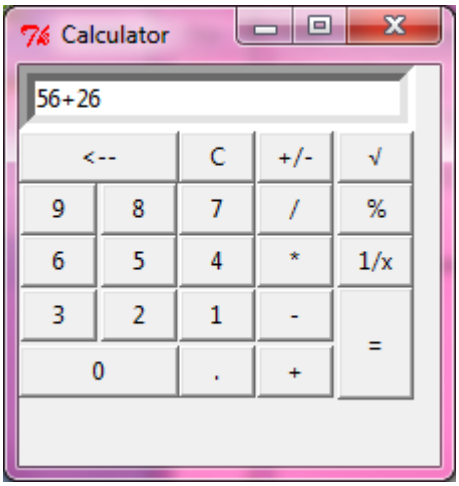


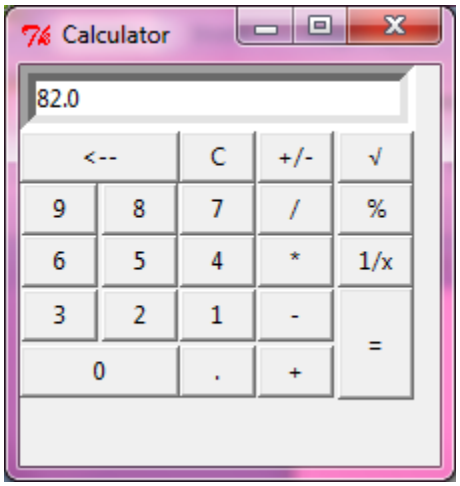
Example: Write a GUI for an Expression Calculator using tk

```
from Tkinter import *
from math import *
root=Tk()
root.title("Calculator")
root.geometry("210x200")
e=Entry(root,bd=8,width=30)
e.grid(row=0,column=1,columnspan=5)
def setText(txt):
   l=len(e.get())
   e.insert(l,txt)
def clear1():
   txt=e.get()
   e.delete(0,END)
   e.insert(0,txt[:-1])
def clear():
   e.delete(0,END)
def sqroot():
   txt=sqrt(float(e.get()))
   e.delete(0,END)
   e.insert(0,txt)
def negation():
   txt=e.get()
   if txt[0]=="-":
      e.delete(0,END)
      e.insert(0,txt[1:])
   elif txt[0]=="+":
      e.delete(0,END)
      e.insert(0,"-"+txt[1:])
   else:
      e.insert(0,"-")

def equals():
   try:
      s=e.get()
      for i in range(0,len(s)):
         if s[i]=="+" or s[i]=="-" or s[i]=="*" or s[i]=="/" or s[i]=="%":
            expr=str(float(s[:i]))+s[i:]
            break
         elif s[i]==".":
            expr=s
            break
      e.delete(0,END)
      e.insert(0,eval(expr))
```

```
    except Exception:
        e.delete(0,END)
        e.insert(0,"INVALID EXPRESSION")

back1=Button(root,text="<--",command=lambda:clear1(),width=10)
back1.grid(row=1,column=1,columnspan=2)

sqr=Button(root,text=u'\u221A',command=lambda:sqroot(),width=4)
sqr.grid(row=1,column=5)

can=Button(root,text="C",command=lambda:clear(),width=4)
can.grid(row=1,column=3)

neg=Button(root,text="+/-",command=lambda:negation(),width=4)
neg.grid(row=1,column=4)

nine=Button(root,text="9",command=lambda:setText("9"),width=4)
nine.grid(row=2,column=1)

eight=Button(root,text="8",command=lambda:setText("8"),width=4)
eight.grid(row=2,column=2)

seven=Button(root,text="7",command=lambda:setText("7"),width=4)
seven.grid(row=2,column=3)

six=Button(root,text="6",command=lambda:setText("6"),width=4)
six.grid(row=3,column=1)

five=Button(root,text="5",command=lambda:setText("5"),width=4)
five.grid(row=3,column=2)

four=Button(root,text="4",command=lambda:setText("4"),width=4)
four.grid(row=3,column=3)

three=Button(root,text="3",command=lambda:setText("3"),width=4)
three.grid(row=4,column=1)

two=Button(root,text="2",command=lambda:setText("2"),width=4)
two.grid(row=4,column=2)

one=Button(root,text="1",command=lambda:setText("1"),width=4)
one.grid(row=4,column=3)
zero=Button(root,text="0",command=lambda:setText("0"),width=10)
zero.grid(row=5,column=1,columnspan=2)
```

```
dot=Button(root,text=".",command=lambda:setText("."),width=4)
dot.grid(row=5,column=3)

div=Button(root,text="/",command=lambda:setText("/"),width=4)
div.grid(row=2,column=4)

mul=Button(root,text="*",command=lambda:setText("*"),width=4)
mul.grid(row=3,column=4)

minus=Button(root,text="-",command=lambda:setText("-"),width=4)
minus.grid(row=4,column=4)

plus=Button(root,text="+",command=lambda:setText("+"),width=4)
plus.grid(row=5,column=4)

mod=Button(root,text="%",command=lambda:setText("%"),width=4)
mod.grid(row=2,column=5)

byx=Button(root,text="1/x",command=lambda:setText("%"),width=4)
byx.grid(row=3,column=5)

equal=Button(root,text="=",command=lambda:equals(),width=4,height=3)
equal.grid(row=4,column=5,rowspan=2)

root.mainloop()
```





Turtle Graphics

- Graphics is the discipline that underlies the representation and display of geometric shapes in two and three-dimensional space.
- A Turtle graphics library provides an enjoyable and easy way to draw shapes in a window and gives you an opportunity to run several functions with an object.
- Turtle graphics were originally developed as part of the children‟s programming language called Logo, created by Seymour Papert and his colleagues at MIT in the late 1960s.
- Imagine a turtle crawling on a piece of paper with a pen tied to its tail.
- Commands direct the turtle as it moves across the paper and tells it to lift or lower its tail, turn some number of degrees left or right and move a specified distance.
- Whenever the tail is down, the pen drags along the paper, leaving a trail.
- In the context of computer, of course, the sheet of paper is a window on a display screen and the turtle is an invisible pen point.
- At any given moment of time, the turtle coordinates. The position is specified with (x, y) coordinates.
- The coordinate system for turtle graphics is the standard Cartesian system, with the origin (0, 0) at the centre of a window. The turtle‟s initial position is the origin, which is also called the home.

Turtle Operations:

Turtle is an object; its operations are also defined as methods. In the below table the list of methods of Turtle class.

Turtle Methods	WHAT IT DOES
home	Moves the turtle to the origin – coordinates (0, 0) – and set its heading to its start-orientation.
fd | forward	Moves the turtle forward for a specified distance, in the direction where the turtle is headed.
bk | backward	Moves the turtle backward for a specified distance, in the direction where the turtle is headed. Do not change the turtle‟s heading.
right | rt	Turns the turtle right by angle units. Units are by default degrees, but can be set via the degrees ( ) and radians ( ) functions.
left | lt	Turns the turtle left by angle units. Units are by default degrees, but can be set via the degrees ( ) and radians ( ) functions.
setx	Set the turtle‟s first coordinate to x, leaves the second coordinate unchanged.
sety	Set the turtle‟s second coordinate to y, leaves the first coordinate unchanged.
goto	Moves the turtle to an absolute position. If the pen is down, draws a line. Do not change the turtle‟s orientation.
degrees	Set the angle measurement unit to radians. Equivalent to degrees (2 * math.pi)
radians	Set the angle measurement unit, i.e., set the number of degrees for a full circle. The default value is 360^0.
seth	Sets the orientation of the turtle to to_angle.

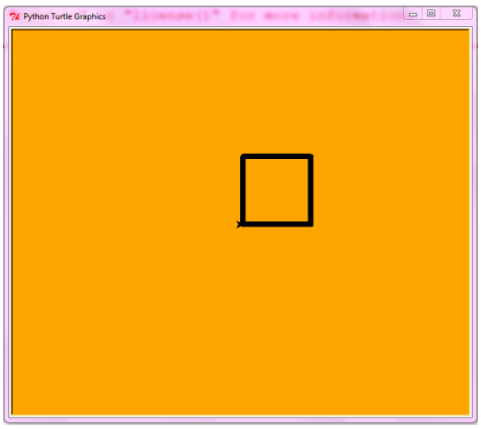
Turtle Object:

t=Turtle( ) creates a new turtle object and open sits window. The window‟s drawing area is 200 pixels wide and 200 pixels high.

t=Turtle(width, height) creates a new turtle object and open sits window. The window‟s drawing area has given width and height.

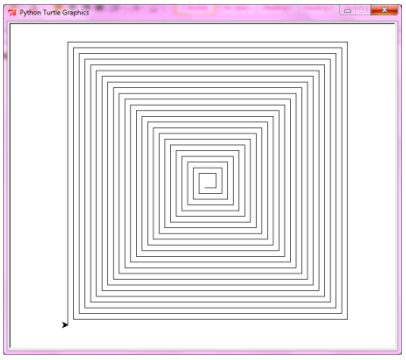
Example-1: Write a program to draw square.

```
import turtle
turtle.bgcolor('orange')
turtle.pensize(8)
turtle.forward(100)
turtle.left(90)
turtle.forward(100)
turtle.left(90)
turtle.forward(100)
turtle.left(90)
turtle.forward(100)
turtle.left(90)
```



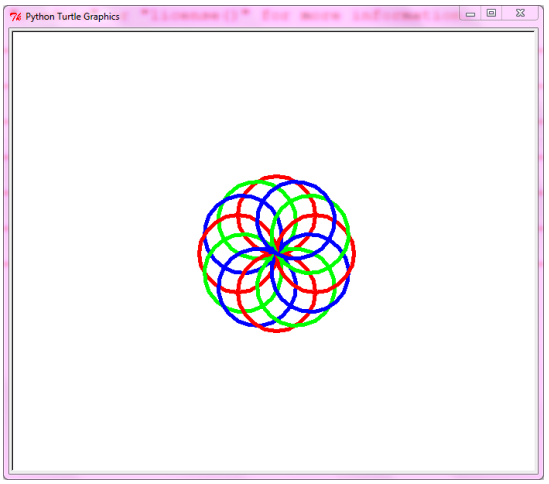
Example-2:

```
import turtle
for i in range(20,500,5):
    turtle.forward(i)
    turtle.left(90)
```



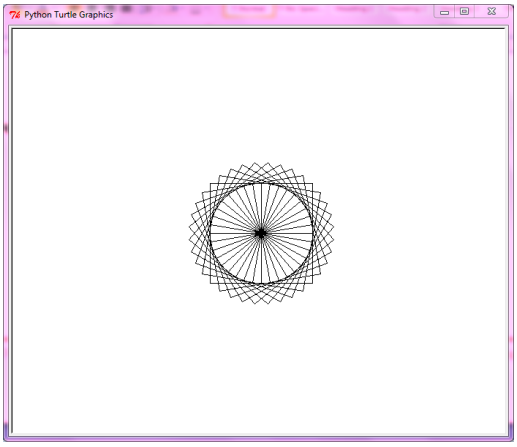
Example-3:

```
import turtle
c=["red","green","blue"]
i=0
turtle.pensize(5)
for angle in range(0,360,30):
    if i>2:
        i=0
    turtle.color(c[i])
    turtle.seth(angle)
    turtle.circle(50)
    i=i+1
```



Example-4:

```
import turtle
for i in range(36):
    for j in range(4):
        turtle.forward(70)
        turtle.left(90)
    turtle.left(10)
```



Testing: Why testing is required?

Software testing is necessary because we all make mistakes. Some of those mistakes are unimportant, but some of them are expensive or dangerous. We need to check everything and anything we produce because things can always go wrong-humans make mistakes all the time.

Software testing is very important because of the following reasons:

1. Software testing is really required to point out the defects and errors that were made during the development phases.
2. It‟s essential since it makes sure of the customer‟s reliability and their satisfaction in the application.
3. It is very important to ensure the quality of the product. Quality product delivered to the customers helps in gaining their confidence.
4. Testing is necessary in order to provide the facilities to the customers like the delivery of high quality product or software application which requires lower maintenance cost and hence results into more accurate, consistent and reliable results.
5. Testing is required for an effective performance of software application or product.
6. It‟s important to ensure that the application should not result into any failures because it can be very expensive in the future or in the later stages of the development.
7. It‟s required to stay in the business.

Basic concepts of testing:

Basics	**Summary**
Software Quality	Learn how software quality is defined and what it means. Software quality is the degree of conformance to explicit or implicit requirements and expectations.
Dimensions of Quality	Learn the dimensions of quality. Software quality has dimensions such as Accessibility, Compatibility, Concurrency, Efficiency …
Software Quality Assurance	Learn what it means and what its relationship is with Software Quality Control. Software Quality Assurance is a set of activities for ensuring quality in software engineering processes.
Software Quality Control	Learn what it means and what its relationship is with Software Quality Assurance. Software Quality Control is a set of activities for ensuring quality in software products.
SQA and SQC Differences	Learn the differences between Software Quality Assurance and Software Quality Control. SQA is process-focused and prevention-oriented but SQC is product-focused and detection-oriented.
Software Development Life Cycle	Learn what SDLC means and what activities a typical SDLC model comprises of. Software Development Life Cycle defines the steps/stages/phases in the building of software.

Software Testing Life Cycle	Learn what STLC means and what activities a typical STLC model comprises of. Software Testing Life Cycle (STLC) defines the steps/ stages/ phases in testing of software.
Definition of Test	Learn the various definitions of the term „test". Merriam Webster defines Test as "a critical examination, observation, or evaluation".
Software Testing Myths	Just as every field has its myths, so does the field of Software Testing. We explain some of the myths along with their related facts.

Unit testing in Python:

The first unit testing framework, JUnit was invented by Kent Back and Erich Gamma in 1997, for testing Java programs. It was so successful that the framework has been implemented again in every major programming language. Here we discuss the python version, unit test.

Unit testing is nothing but testing individual „units", or functions of a program. It does not have a lot to say about system integration, whether the various parts of a program fit together. That"s a separate issue.

The goals of unit testing framework are:

- To make it easy to write tests. All a „test" needs to do is to say that, for this input, the function should give that result. The framework takes care of running the tests.
- To make it easy to run tests. Usually this is done by clicking a single button or typing a single keystroke (F5 in IDLE). Ideally, you should be comfortable running tests after every change in the program, however minor.
- To make it easy to tell if the tests passed. The framework takes care of reporting results; it either simply indicates that all tests passed, or it provides a detailed list of failures.

Example:

```
import unittest

class TestStringMethods(unittest.TestCase):

    def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')

    def test_isupper(self):
        self.assertTrue('FOO'.isupper())
        self.assertFalse('Foo'.isupper())
if__name__== '_main_':
    unittest.main()
```

Output:

```
..
----------------------------------------------------------------------
Ran 2 tests in 0.016s

OK
```

Writing and Running Test cases

- Your object is to write test and not to prove that your program works, it‟s to try to find out where it doesn‟t! Test every „extreme‟ case you can think of.
- For example, if you were to write and test a function to sort a list, then the first and last elements get moved to correct position? Can you sort a 1-element list without getting an error? How about an empty list?
- While you can put as many tests as you like into one test method that you shouldn‟t test methods should be short and single-purpose. If you are testing different aspects of a function, they should be in separate tests.
- Here are the rules for writing test methods:
  - The name of a test method must start with the letters „test‟, otherwise it will be ignored.
  - This is so that you can write „helper‟ methods you can call from your tests, but are not directly called by the test framework.
  - Every test method must have exactly one parameter, which is nothing but „self‟. You must put self in front of every built-in assertion method you call.
  - The tests must be independent of one another, because they may be run in any order.
  - Do not assume they will execute in the order they occur in the program.
- Here are some of the built-in test methods you can call. Each has an optional message parameter, to be printed if the test fails.

Method	Checks that
assertEqual(a, b)	a == b
assertNotEqual(a, b)	a != b
assertTrue(x)	bool(x) is True
assertFalse(x)	bool(x) is False
assertIs(a, b)	a is b
assertIsNot(a, b)	a is not b
assertIsNone(x)	x is None
assertIsNotNone(x)	x is not None
assertIn(a, b)	a in b
assertNotIn(a, b)	a not in b
assertIsInstance(a, b)	isinstance(a, b)
assertNotIsInstance(a, b)	not isinstance(a, b)

Example: Unittest for addition of two numbers.

```
import unittest
def add(a,b):
    if isinstance(a,int) and isinstance(b,int):
        return a+b
    elif isinstance(a,str) and isinstance(b,str):
        return int(a)+int(b)
    else:
        raise Exception('Invalid arguments')
```

```
class TestAdd(unittest.TestCase):
    def test_add(self):
        self.assertEqual(5,add(2,3))
        self.assertEqual(15,add(-6,21))
        self.assertRaises(Exception,add,4.0,5.0)
unittest.main()
```

Output:

```
.
----------------------------------------------------------------------
Ran 1 test in 0.008s

OK
```