

- \* History of python
- \* Need of python programming.
- \* Applications Basics of python programming
- \* Using -the REPL(shell)
- \* Running python scripts
- \* Variables
- \* Assignment
- \* keywords
- \* Input - output
- \* Indentation.

## History of python:

Python is a widely used high-level programming language for general-purpose programming. Created by Guido van Rossum and first released in 1991. An interpreted language, Python has a design philosophy which emphasizes code readability, and a syntax which allows programmers to express concepts in fewer lines of code than possible in languages such as C++ or Java. The language provides constructs intended to enable writing clear programs on both a small and large scale.

Python features a dynamic-type system and automatic memory management and supports multiple programming paradigms, including object-oriented, imperative, functional programming, and procedural styles. It has a large and comprehensive standard library.

CPython, the reference implementation of python, is open source software and has a community-based development model, as do nearly all of its variant implementations. CPython is managed by the non-profit python software Foundation.

\* Python was conceived in the late 1980's, and its implementation began in December 1989 by Guido van Rossum at centrum wiskunde & Informatic (CWI) in the Netherlands as a successor to the ABC language capable of exception.

\* van Rossum is python's principal author, and -his his continuing central role in deciding the direction of python is reflected in the title given to him by the python community, benevolent dictator for life (BDFL)

\* python 2.0 was released on 16 october 2000 with a new feature, including a cycle detecting garbage collector and support for unicode.

\* python 3.0, a major, backwards-incompatible release, was released on 3 dec 2008 after a long period of testing.

\* The End of life date (EOL, sunset date) for python 2.7 was initially set at py 2015, then postponed to 2020 out of concern that a large body of existing code cannot easily be forward-ported to python 3.

\* 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 uses dynamic typing and mix of reference counting and a cycle-detecting garbage collector for memory management

\* The standard library has two modules (itertools and functools) that implement functional tools borrowed from Haskell and standard ML.



**Software quality:** For many, python's focus on readability, and software quality in general sets it apart from other tools in the scripting world. python code is designed to be readable, and hence reusable and maintainable - much more so than traditional scripting languages. The uniformity of python codes 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++, 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. python offers multiple options for coding portable graphical user interfaces, database access programs, web based systems and more.

**Support libraries:**

python comes with a large collection of prebuilt and portable functionality, known as the standard library. The python's third-party



serial port access, game development, and much more (see ahead for a sampling).

### Component integration:

Python scripts can easily communicate with other parts of application, using a variety of integration mechanisms. Such integrations allow python to be used as a product customization and extension tool. .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 easy of use and built-in toolset, it can make the act of programming more pleasure the chore. Although this may be an intangible benefit, its effect on productivity the is an important asset.

### Software Quality:

By design, python implements a deliberately simple and readable syntax and highly coherent programming model. As a slogan at a past python conference attests, the net result is that python seems to "fit your brain" - i.e., features of the language interact python is perhaps best described as an object

During the great Internet boom of the mid-to-late 1990s, it was difficult to find enough programmers to implement software projects; developers were asked to implement systems as fast as the Internet evolved. It is deliberately optimized for speed of development - its simple syntax, dynamic typing, lack of compile steps, and built-in toolset allow programmer to develop programs in a fraction of the time needed when using some other tools.

It's Object-Oriented:

Python is an object oriented language, from the ground up. Its class model supports advanced notations such as polymorphism, operator overloading, Object-Oriented programming (oop) is easy to apply and multiple inheritance; yet in the context of dynamic typing, object-oriented programming (oop)

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.

Database programming:

Python's standard pickle module provides a simple object-persistence system: it allow programs to easily save & restore entire files.



## Python Virtual Machine(PVM):

Once your program has been compiled to byte code for the byte code has been loaded from existing .pyc files) it is shipped off for execution to something generally known as the python Virtual Machine (PVM)

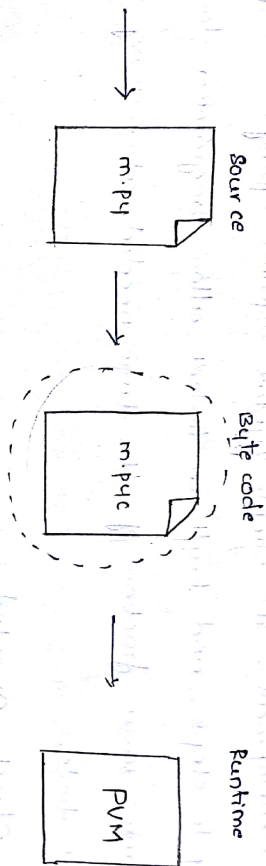


Figure: Python's traditional runtime execution model

Source code you type is translated to byte code, which is then run by the python Virtual Machine. Your code is automatically compiled, but then it is interpreted.

### The Interactive prompt:

Starting an Interactive session:

perhaps the simplest way to run Python programs is to type them at Python's interactive command line, sometimes called the interactive prompt.

```

>>> % python
Python 3.3.0 (v3.3.0:bda6b90ebf2, Sep 29 2012, 10:57:17)
[MSC v.1600 64-bit...]
  
```

Type "help", "copyright", "credits" or "license" for more information.

```

>>> ^Z
  
```

## Introducing the python Interpreter:

\* A Interpreter is a kind of program that executes other programs.

\* Windows users fetch and run a self-installing executable file that puts python on their machines simply double-click and say yes or Next at all prompts.

\* Linux and Mac OS X users probably already have a usable python preinstalled on their computers - it's a standard component on these platforms today.

\* Some Linux & Mac OS X users (and most Unix users) compile python from its full source code distribution package.

## Program Execution:

What is meant to write and run a python script depends on whether you look at these tasks as a programmer or as a python interpreter.

### The programmer's view:

In its simplest form, a python program is just a text file containing python statements. For eg: the following file, named script().py is one of the python scripts I could dream up.

```
print('hello world')    print(2**100)
```

The file contains two python print statements, which simply print a string (text) and a numeric expression result (2 to the power 100) to the output stream.



typing the word "python" at your system shell prompt

this begins an interactive python session; the "." character at the start of this listing stands for a generic system prompt in this book - it's not input that you type yourself.

1. On windows, you can type python in a dos console window - a program named cmd.exe and usually known as command prompt. For more details on starting this program.
2. On linux (and other Unixes), you might type this command in a shell or terminal window.

The system path:

When we typed python in the last section to start an interactive session, we relied on the fact that the system located the python program for us on its program search path.

In linux:

open a terminal and type

```
export PATH = " $ PATH: /usr/local/bin/python"
```

and press Enter.

In Windows:

At the command prompt, type

```
path %path%; c:\python
```

and press Enter.

Python path: Tells the python interpreter where to locate the module files imported in a program, PYTHONPATH is sometime present the python installer.

Variables are nothing but reserved memory locations to store values. This means when you create a variable, you reserve some space in memory. Based on the data type of a variable, the interpreter allocates memory and decides what can be stored in reserved memory.

Rules for variable in python are the same as they are in most other high-level languages inspired by (or more likely, written in) C.

They are simply identifier names with an alphabetic first character "alphabetic" meaning upper- or lowercase letters, including the underscore (`_`). Any additional characters may be alphanumeric or underscore. python is a case-sensitive.

```
>>> counter = 0
```

```
>>> miles = 1000.0
```

```
>>> name = 'Bob'
```

```
>>> counter = counter + 1
```

```
>>> kilometers = 1.609 * miles
```

```
>>> print '%.f miles is the same as %.f km' % (miles, kilometers)
```

```
1000.000000 miles is the same as 1609.000000 km.
```

These few are examples of variable Assignment.

Python also supports augmented assignment, statements that both refer to and assign values to variable.

```
n = n * 10    or    n *= 10
```



unary operators, python allows you to assign a single value to several variables simultaneously.

i.e.,  $a=b=c=1$ .

you can also assign multiple objects to multiple variable.

i.e.,  $a, b, c = 1, 2, "john"$ .

Keywords: these are reserved words and you cannot use them as constant or variable or any other identifier names.

And all python keywords contain lowercase letters only.

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

input function. When the input function is called the program stops running the program, prompts the user to enter something at the keyword by printing a string called the prompt to the screen, 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.

eg: `name = input("please enter name:")`  
`print("The name entered was", name)`

The input function prints prompt message to the screen and waits for the user to enter input in the python shell window. The program does not continue executing until you have provided the input requested. When we enter something, python takes that and stores it in the variable called name in this case.

eg: `age = int(input("enter age:"))`  
`olderAge = age + 1`  
`print("Next year you will be", olderAge)`



## Output (print):

Prints the values to a output window.

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

or

```
print(value 1, value 2, ...)
```

file: a file-like object (stream); defaults to the current `sys.stdout`.

sep: string inserted between values, default a space

end: string appended after the last value, default a newline.

flush: whether to forcibly flush the stream.

Each time a comma appears between items in a print statement, a space appears in the output.

eg: create a file in IDLE and save it as add.py

```
x = input("Enter an integer:")
```

```
y = input("Enter another integer:")
```

```
z = int(x) + int(y)
```

```
print("sum of", x, "and", y, "is", z, ".", sep=" ")
```

output:

Enter an integer: 23

Enter another integer: 12

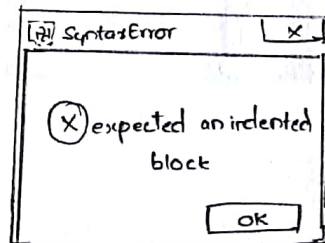
sum of 23 and 12 is 35.

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.

Python does not support braces to indicate blocks of code for class & function definitions or flow control. Blocks of codes are denoted by line indentation.

Eg: 1.   if True:  
          print("True")  
          else  
          print("False")

Eg: 2   if True:  
          print("True")  
          else:  
          print("False")





# UNIT-II

## 2. TYPES , OPERATORS AND EXPRESSIONS

- \* Types - Integers , strings , Booleans
- \* Operators - Arithmetic Operators,
  - comparison (Relational) operators
  - Assignment Operators
  - Logical operators
  - Bitwise operators
  - Membership Operators
  - Identity operators.
- \* Expressions and order of evaluations
- \* control flow - if , if-elif-else , for , while ,  
break , continue , pass.



## Data types :

Python data types are :

1. Numbers

2. Booleans

3. Strings

### 1. Numbers :

Python has 4 built-in numeric data types :

1. Integers

2. Long integers

3. Floating point numbers

4. Imaginary numbers.

#### Constant

1234, -24, 0

999999999999999L

1.23, 3.14e-10, 4E210,

4.0e+210

0177, 0x9ff

3+4j, 3.0+4.0j, 3J

#### Interpretation

Normal integers (C longs)

Long integers (unlimited size)

Floating-point (C doubles)

Octal and Hex constants.

Complex number constants

### Integer and floating-point constants :-

Integers are written as a string of decimal digits.

Floating-point numbers have an embedded decimal point, and/or an optional signed exponent introduced by an e or E. If you write a number with a decimal point or exponent, python

(not integer) math when it's used in an expression. The rules for writing floating-point numbers are the same as with C.

Python numbers get as much precision as the C compiler used to build the python interpreter gives to longs and doubles. On the other hand, if an integer constant end with an L or l, it becomes a python long integer (not to be confused with a C long) and can grow as large as needed.

### Hexadecimal and octal constants:

The rules for writing hexadecimal (base 16) and octal (base 8) integers are the same as in C: Octal constants start with a leading zero(0), and hexadecimals start with a leading 0x or 0X.

Long integers:

Now for something more exotic: here's a look at long integers in action. When an integer constant ends with an L (or lowercase l), python creates a long integer, which can be arbitrarily big:

>>> 99999999999999999999999999999999 + 1

overflow Error: integer literal too large.

>>> 999999999999999999999999999999+1

1000000000000000000000000000000L

## Complex numbers :

Python complex constants are written as real-part + imaginary-part, and terminated with a j or J. Internally, they are implemented as a pair of floating-point numbers, but



Complex numbers.

```
>>> 1j * 1j
```

```
(-1+0j)
```

```
>>> 2+1j*3
```

```
(2+3j)
```

```
>>> (2+1j)*3
```

```
(6+3j)
```

Attribute	Description
num.real	Real component of complex number num
num.imag	Imaginary component of complex number num
num.conjugate()	Returns complex conjugate of num.

## 2. Boolean:

- \* They have a constant value of either True or False.
- \* Booleans are subclassed from integers but cannot themselves be further derived.
- \* Recall that python object typically have a Boolean False value for any numeric zero or empty set.
- \* Also, if used in an arithmetic context, Boolean values True and False will take on their numeric equivalents of 1 and 0
- \* Most of the standard library and built-in Boolean functions that previously returned integers will now return Booleans.
- \* All python objects have an inherent True or False value.

```
>>> bool(1)
```

```
True
```

```
>>> bool(0)
```

```
False
```

```
>>> bool('1')
```

```
True
```

```
>>> bool('0')
```

```
True
```

```
>>> bool([])
```

```
False
```

#using Booleans numerically

```
>>> foo = 42
```

```
>>> bar = foo < 100
```

```
>>> bar
```

```
True
```

```
>>> print bar + 100
```

```
101
```

```
>>> print '%.s' % bar
```

```
True
```

```
>>> print '%.d' % bar
```

```
1
```



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 & working their way from -1 at the end. The plus (+) sign is the string concatenation operator and the asterisk (\*) is the repetition operator.

```
>>> c = 'aaaaaaaaa bbbb cccc dddd eee ffff'
```

```
>>> c
```

```
'aaaaaaaaa bbbb cccc dddd eee ffff'
```

```
>>> str = "Hello World!"
```

```
print(str)
```

```
print(str[0])
```

```
print(str[2:5])
```

Output:

```
Hello World!
```

```
H
```

```
llo
```

Updating strings:

you can "update" an existing string by (re) assigning a variable to another string. The new value can be related to its previous value or to a completely different string all together.

```
var1 = "Hello World"
print "Updated string : ", var1[:6] + 'python'
```

Output: Updated string : Hello python.

## Escape characters:

An escape character gets interpreted; in a single quoted as well as double quoted strings. Following table is a list of escape or non-printable characters that can be represented with backslash notation.

Backslash notation	Hexadecimal character	Description
\a	0x07	Bell or alert
\b	0x08	Backspace
\e	0x1b	Escape
\f	0x0c	Formfeed
\n	0x0a	Newline
\nnn		Octal notation, where n is in the range 0-7
\r	0x0d	Carriage return
\s	0x20	Space
\t	0x09	Tab
\v	0x0b	Vertical tab
\x	x	character x
\xnn		Hexadecimal notation, where n is in the range 0-9, a-f, or A-F.

Assume string variable a holds 'Hello' and variable b holds 'python', then:

Operator	Description	Example
+	concatention - Adds values on either side of the operator	a+b will give Hello Python
*	Repetition - creates new strings concatenating multiple copies of the same string	a*2 will give Hello Hello.
[ ]	Range slice - Gives the characters from the given range index	a[1] will give c
[ : ]	Range slice - Gives the characters from the given range.	a[1:4] will give ell
in	Membership - Returns true if a character exists in the given string.	H in a will give 1
not in	Membership - Returns true if a character does not exist in the given string.	M not in a will give 1

### String Formatting Operator:

One of python's coolest features is the string format operator %. This operator is unique to strings and makes up for the lack of having function from c's printf() family.



Output: My name is zara and weight is 21 kg!

Format Symbol	Conversion
%c	character
%s	string conversion via str() prior to formatting
%i	signed decimal integer
%d	signed decimal integer
%u	unsigned decimal integer
%o	octal integer
%x	hexadecimal integer (lowercase letters)
%X	hexadecimal integer (UPPER case letters)
%e	exponential notation (with lowercase 'e')
%E	exponential notation (with UPPER case 'E')
%f	floating point real number
%g	the shorter of %f and %e
%G	the shorter of %F and %E.

Symbol	Functionality
*	argument specifies width or precision
-	left justification
+	display the sign
<sp>	leave a blank space before a positive number.
#	add the octal leading zero ('0') or hexadecimal leading '0x' or '0X', depending on whether 'x' or 'X' were used.
0	pad from left with zeros (instead of spaces)
%	'%. %' leaves you with a single literal '%'
(var)	mapping variable (dictionary arguments)

### String Methods :

1. capitalize() : capitalizes first letter of string.

```
>>> st = "hello to all"
```

```
>>> st.capitalize()
```

```
'Hello to all'
```

2. endswith():

```
endswith(suffix, beg=0, end=len(string))
```

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

```
suffix = "wow!!!"
```

```
print str.endswith(suffix)
```

```
print str.endswith(suffix, 20)
```

```
suffix = "is"
```

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

Output: True

True

True

False

3. find():

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

```
str 1 = "this is string example... wow!!!";
```

```
str 2 = "exam";
```

```
print str1.find(str2);
```

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

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

Output: 15 15 -1

4. isalnum(): It checks whether the string consists of alphanumeric characters.

```
str = "this 2009"; #nospace in this string.
```

```
print str.isalnum();
```

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

```
print str.isalnum();
```

Output: True False.

5. isdigit(): checks whether the string consists of digits only.

```
str = "123456";
```

```
print str.isdigit();
```

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



(letters) of the string are uppercase.

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

```
print str.isupper();
```

Output: True False

10. **Join()**: Returns a string in which the string elements of sequence have been joined by strseparator.

```
str = "-";
```

```
seq = ("a", "b", "c");
```

```
print str.join(seq);
```

output: a-b-c

11. **len()**: Returns the length of the string.

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

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

output:

Length of the string: 32

12. **lower()**: Returns a copy of the string in which all case-based characters have been lowercased.

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

```
print str.lower();
```

Output:

this is string example... wow!!!

Output: True

False

6. `islower()`: checks whether all the case-based character (letters) of the string are lowercase.

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

```
print str.islower();
```

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

```
print str.islower();
```

Output: False

True

7. `isspace()`: checks whether the string consists of whitespace.

```
str = " ";
```

```
print str.isspace();
```

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

```
print str.isspace();
```

Output:

True

False

8. `istitle()`:

```
str = "This is string Example... wow!!!";
```

```
print str.istitle();
```

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

```
print str.istitle();
```

output: True

False.

13. `rstrip()`: Returns the string where characters have been stripped from the beginning of the string (default white-space characters).

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

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

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

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

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

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

14. `max()`: Returns the max alphabetical character from the string `str`.

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

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

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

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

output:

Max character: y

Max character: x

15. `min()`: Returns the min alphabetical character from the string `str`.

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

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



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

Output:

Min character: !

Min character: !

16. `replace()`: Returns a copy of the string in which the occurrences of old have been replaced with new, optionally restricting the number of replacement to max.

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

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

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

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

17. `rstrip()`: Return a copy of the string in which all chars have been stripped from the end of the string (default whitespace characters).

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

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

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

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

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

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

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

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

```
str = "line 1-abcdef \n line2-abc \n line 4-abcd";
```

```
print str.split();
```

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

Output: ['line 1-abcdef', 'line 2-abc', 'line 4-abcd']

```
['line 1-abcdef', '\n line2-abc \n line 4-abcd']
```

19. swapcase():

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

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

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

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

# Arithmetic Operators

[www.FirstRanker.com](http://www.FirstRanker.com)
[www.FirstRanker.com](http://www.FirstRanker.com)

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	Multiplication 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 $9.0 // 2.0 = 4.0$



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

Assume Variables a is 10 & b is 20.

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	$(a >= b)$ is 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
----	--	------------------

## Python 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

<p><b>**=</b> Exponent AND</p>	<p>performs exponential (power) calculation on operators and assign value to the left operand.</p>	<p><b>**</b> is equivalent to <math>c = c ** a</math></p>
<p><b>//</b> Floor division</p>	<p>It performs floor division on operators and assign value to the left operand.</p>	<p><math>c // a</math> is equivalent to <math>c = c // a</math>.</p>

## Python Bitwise Operators:

Operator	Description	Example.
<p><b>&amp;</b> Binary AND</p>	Operator copies a bit to the result if it exists in both operands.	<p><math>(a \&amp; b) = 12</math> (means 0000 1100)</p>
<p><b> </b> Binary OR</p>	It copies a bit if it exists in either operand.	<p><math>(a   b) = 61</math> (means 0011 1101)</p>
<p><b>^</b> Binary XOR</p>	It copies the bit if it is set in one operand but not both	<p><math>(a \wedge b) = 49</math> (means 0011 0001)</p>
<p><b>~</b> Binary Ones complement</p>	It is unary and has the effect of 'flipping' bits	<p><math>(\sim a) = -61</math> (means 1100 0011) In 2's complement form due to signed binary number</p>
<p><b>&lt;&lt;</b> Binary left shift</p>	The left operands Value is moved left by the number of bits specified by the right operand	<p><math>a \ll 2 = 240</math> (means 1111 0000)</p>
<p><b>&gt;&gt;</b> Binary Right shift</p>	The left operands Value is moved right by the number of bits specified by the right	<p><math>a \gg 2 = 15</math> (means 0000 1111)</p>



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

### Python Membership Operators :

python's membership operators test for membership in a sequence, such as lists, strings or tuples. There are two membership operators as explained below:

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

Identity operators compare the memory locations of two objects. There are two identity operators as explained below:

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

# Expressions and Order of evaluations :

www.FirstRanker.com

www.FirstRanker.com

## Python operators precedence :

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

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

Operator precedence affects how an expression is evaluated.

Eg:  $x = 7 + 3 * 2$  ; here x is assigned 13, not 20 because Operator \* has highest precedence than +, so it first multiplies  $3 * 2$  and then adds 7. Here, operators with the highest precedence appear at the top of the table, those with the lowest appear bottom.

www.FirstRanker.com



## Control flow:

if: It executes only when the condition is true or else  
syntax: if condition:                      out of the block.

stmt 1

stmt 2

Eg: a=10

if a==10:

print "a is:", a;

output: a is: 10

if-elif-else-

syntax: if condition:

stmt x<sub>1</sub>

elif condition:

stmt x<sub>2</sub>

else:

stmt x<sub>3</sub>

Eg: a=10

b=20

if a>b:

print "a is big";

elif a<b:

print "b is big";

else:

print "both are equal"

output: b is big.

# UNIT-III

### 3. DATA STRUCTURES

\* Lists - Operations

slicing

Methods

\* Tuples

\* sets

\* Dictionaries

\* sequences

\* comprehensions.



## Lists:

Lists are python's most flexible ordered collection object type. Unlike strings, lists can contain any sort of object: numbers, strings, even other lists. python lists do the work of most of the collection datastructures you might have to implement manually in lower-level languages such as C. In terms of some of their main properties, python lists are:

### Ordered collection of arbitrary objects:

From a functional view, lists are just a place to collect other objects, so you can treat them as a group. Lists also define a left-to-right positional ordering of the items in the list.

### Accessed by offset:

Just as with strings, you can fetch a component object out of a list by indexing the list on the object's offset. Since lists are ordered, you can also do such tasks as slicing and concatenation.

### Variable length, heterogeneous, arbitrarily nestable:

Unlike strings, lists can grow and shrink in place (they're variable length), and may contain any sort of object, not just one-character strings (they're heterogeneous). Because lists can contain other complex objects, lists also support arbitrary nesting; you can create lists of lists of lists, and so on.

In fact, sequence operations work the same on lists as on strings. On the other hand, because lists are mutable, they also support other operations strings don't, such as deletion, index assignment, and methods.

### Arrays of object references:

Technically, python lists contain zero or more references to other objects. If you've used a language such as C, lists might remind you of arrays of pointers. Fetching an item from a python list is about as fast as indexing a C array; in fact, lists really are C arrays inside the python interpreter. Use the square brackets for slicing along with the index or indices to obtain value available at that index.

Eg: `list 1 = ['physics', 'chemistry', 1997, 2000];`

`list 2 = [1, 2, 3, 4, 5, 6, 7];`

`print "list [0] :", list1[0]`

`print "list 2[1:5] :", list 2[1:5]`

Output:

`list 1[0] : physics`

`list 2 [1:5] : [2, 3, 4, 5]`

We can update single or multiple elements of lists.

```
list = ['physics', 'chemistry', 1997, 2000];
```

```
print "value available at index 2:"
```

```
print list[2];
```

```
list[2] = 2001;
```

```
print "New value available at index 2:"
```

```
print list[2];
```

Output: value available at index 2:

1997

New value available at index 2:

2001.

**Deleting List Elements:**

Use the del statement to delete elements.

eg: 

```
list1 = ['physics', 'chemistry', 1997, 2000];
```

```
print list1;
```

```
del list1[2];
```

```
print "After deleting value at index 2:"
```

```
print list1;
```

Output: 

```
['physics', 'chemistry', 1997, 2000]
```

After deleting value at index 2:

```
['physics', 'chemistry', 2000]
```



Python Expression	Results
<code>len([1, 2, 3])</code>	3
<code>[1, 2, 3] + [4, 5, 6]</code>	<code>[1, 2, 3, 4, 5, 6]</code>
<code>['Hi!'] * 4</code>	<code>['Hi!', 'Hi!', 'Hi!', 'Hi!']</code>
<code>3 in [1, 2, 3]</code>	True
<code>for x in [1, 2, 3]: print x</code>	1 2 3

### Indexing, slicing:

`L = ['spam', 'Spam', 'SPAM!']`

Python Expression	Results
<code>L[2]</code>	<code>'SPAM!'</code>
<code>L[-2]</code>	<code>'Spam'</code>
<code>L[1:]</code>	<code>['Spam', 'SPAM!']</code>

### Built-in List Functions:

1. `len()`: The method `len()` returns the number of elements in the list.

`List 1, List 2 = [123, 'xyz', 'zara'], ['456', 'abc']`

`print "First list length:", len(List 1);`

`print "Second list length:", len(List 2);`

Output: First list length : 3

Second list length : 2.

list with maximum value.

```
list 1, list 2 = [123, 'xyz', 'zara', 'abc'], [456, 100, 200]
```

```
print "Max value element: ", max(list 1);
```

```
print "Max value element: ", max(list 2);
```

Output: Max value element: zara

Max value element: 100.

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

```
list 1, list 2 = [123, 'xyz', 'abc', 'zara'], [456, 700, 200]
```

```
print "Min value Element: ", min(list 1);
```

```
print "Min value Element: ", min(list 2);
```

Output: min value element: 123

min value element: 200

4. append(): The method append() appends a passed obj into existing list.

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

```
alist.append(2009);
```

```
print "update list: ", a list;
```

output:

update list: [123, 'xyz', 'zara', 'abc', 2009]

items obj. occurs in list.

```
a list = [123, 'xyz', 'zara', 'abc', 123];
```

```
print "count for 123 : ", a list.count(123);
```

Output: count for 123 : 2

6. Extend(): The method extend() appends the contents of seq to list.

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

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

```
alist.extend(blist);
```

```
print ("Extended list:", alist);
```

Output: Extended list : [123, 'xyz', 'zara', 'abc', 123, 2009, 'manni']

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

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

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

Output: Index for xyz : 1

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

```
alist = [123, 'xyz', 'zara', 'abc']
```

```
alist.insert(3, 2009)
```



Output: Final list: [123, 'xyz', 'zara', 'abc']

9. pop(): The method pop() removes and returns last object or Obj from the list.

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

```
print " A list:", alist.pop();
```

Output: A List: abc

10. remove(): This method doesnot return any value but removes the given object from the list.

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

```
alist.remove('xyz');
```

```
print "List:", alist;
```

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

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

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

```
alist.reverse();
```

```
Print "list:", alist;
```

Output:

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

construct simple group of objects. They work exactly like lists, except that tuples can't be changed in place.

Ordered collections of arbitrary objects:

Like strings and lists, tuples are an ordered collection of objects; like lists, they can embed any kind of object.

Accessed by offset:

Like strings & lists, items in a tuple are accessed by offset (not key); they support all the offset-base access operations we've already seen, such as indexing and slicing.

Of the category immutable sequence:

Like strings, tuples are immutable; they don't support any of the in-place change operations we saw applied to list. Like strings and lists, tuples are sequences; they support many of the same operations.

Fixed-length, heterogeneous, and arbitrarily nestable:

Because tuples are immutable, you cannot change the size of a tuple without making a copy. On the other hand, tuples can hold any type of object, including other compound objects (eg: lists, dictionaries, other tuples), and so support arbitrary nesting.

Like lists, tuples are best thought of as object reference arrays; tuples store access points to other objects (reference) and indexing a tuple is relatively quick. Table highlights common tuple operations.

Operation	Interpretation
<code>()</code>	An empty tuple
<code>T = (0,)</code>	A one-item tuple (not an expression)
<code>T = (0, 'Ni', 1.2, 3)</code>	A four-item tuple.
<code>T = 0, 'Ni', 1.2, 3</code>	Another four-item tuple (same as prior line)

Accessing values in Tuples:

```
tup1 = ('physics', 'chemistry', 1997, 2000);
```

```
tup2 = (1, 2, 3, 4, 5, 6, 7);
```

```
print "tup1[0]: ", tup1[0]
```

output:

```
tup1[0]: physics
```

- \* Updating tuple is impossible
- \* Deleting tuple Element is not possible.

Basic tuples Operations:

- \* Tuples respond to the + and \* operators much like strings.
- \* The result is a new tuple.



len((1,2,3))	3
(1,2,3)+(4,5,6)	(1,2,3,4,5,6)
('Hi!')*4	('Hi!', 'Hi!', 'Hi!', 'Hi!')
3 in (1,2,3)	True
for x in (1,2,3):	123
print x,	

### Indexing slicing:

L = ('spam', 'Spam', 'SPAM!')

Python expression	Results
L[2]	'SPAM!'
L[-2]	'Spam'
L[1:]	['Spam', 'SPAM!']

### Built-in Tuple Functions:

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

tuple 1, tuple 2 = (123, 'xyz', 'zara'), (456, 'abc')

print "First tuple length: ", len(tuple 1);

print "Second tuple length: ", len(tuple 2);

### Output:

First tuple length : 3

Second tuple length : 2

max() - the method max() returns the elements from the tuple with maximum value.

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

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

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

output: Max value: zara

Max value: 700

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

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

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

output: min value: 123

4. tuple(): This method converts an object into tuple.

```
alist = (123, 'xyz', 'zara', 'abc');
```

```
aTuple = tuple(list)
```

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

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

```
>>> a = [2, "abc", 23.7]
```

```
>>> at = tuple(a)
```

```
>>> at
```

```
(2, 'abc', 23.7)
```

Besides decimals, python 2.4 also introduced a new collection type, the set - an unordered collection of unique and immutable objects that supports operation corresponding to mathematical set theory.

However, because sets are unordered and do not map keys to values, they are neither sequence nor mapping types; they are a type category unto themselves. Moreover, because sets are fundamentally mathematical in nature we'll explore the basic utility of python's set objects here.

Built-in set function:

To make a set object, pass in a sequence or other iterable object to the built-in set functions:

```
>>> x = set('abcdef')
```

```
>>> y = set('bdfxyz')
```

You get back a set object, which contains all the items in the object passed in

```
>>> x
```

```
set(['a', 'c', 'b', 'e', 'd', 'f'])
```

#python < 2.6 display format.



Python symbol	Description
in	Is a member of
not in	Is not a member of
==	Is equal to
!=	Is not equal to
<	Is a (strict) subset of
<=	Is a subset of (includes improper subsets)
>	Is a (strict) superset of
>=	Is a superset of (includes improper supersets)
&	Intersection
	Union
- or \	Difference or relative complement
^	Symmetric difference.

Why Sets?

Set operations have a variety of common uses, some more practical than mathematical. For example:

eg: `>>> L = [1, 2, 1, 3, 2, 4, 5]`

`>>> set(L)`

`{1, 2, 3, 4, 5}`

`>>> l = list(set(L))`

`>>> L`

Membership

```
>>> A = {1, 2, 3}
```

```
>>> B = {3, 4, 5, 6}
```

```
>>> 1 in A
```

True.

1. add method : Adds new element to the set.

```
>>> A = {1, 2, 3}
```

```
>>> A.add(4)
```

```
>>> A
```

```
{1, 2, 3, 4}
```

2. remove( ) : Removes a member from the set.

```
>>> A = {2, 3, 4, 5}
```

```
>>> A
```

```
{2, 3, 4, 5}
```

```
>>> A.remove(2)
```

```
>>> A
```

```
{3, 4, 5}
```

3. Union & : Union set of element in either set A or set B.

Operator is |

```
>>> A = {1, 3, 4, 5}
```

```
>>> B = {2, 4, 6, 8}
```

```
>>> A|B
```

```
{1, 2, 3, 4, 5, 6, 8}
```

```
>>> A = {1, 3, 4, 5}
```

```
>>> B = {2, 4, 6, 8}
```

```
>>> A & B
```

```
{4}
```

5. difference ( ): set of elements in set A, but not in set B

```
>>> A = {1, 3, 4, 5}
```

```
>>> B = {2, 4, 6, 8}
```

```
>>> A - B
```

```
{1, 3, 5}
```

6. Symmetric difference: set of elements in set A or set B, but not both

```
>>> A = {1, 3, 4, 5}
```

```
>>> B = {2, 4, 6, 8}
```

```
>>> A ^ B
```

```
{1, 2, 3, 5, 6, 8}
```

7. size ( ): Number of elements in the set

```
>>> A = {1, 3, 4, 6, 7}
```

```
>>> len(A)
```

```
5
```

8. Empty set :

```
>>> subject = {}
```

```
>>> subject
```

```
{}
```



Dictionary: Dictionaries are unordered collections, their chief distinction is that items are stored and fetched in dictionaries by key, instead of offset. As we'll see, built-in dictionaries can replace many of the searching algorithm and data-structures you might have to implement manually in lower-level languages.

Accessed by key, not offset:

Dictionaries are sometimes called associative arrays of hashes. They associate a set of values with keys, so that you can fetch an item out of a dictionary using the key that stores it.

Unordered collections of arbitrary objects:

Unlike lists, items stored in a dictionary aren't kept in any particular order, in fact, python randomizes their order in order to provide quick lookup.

Variable length, heterogeneous, arbitrarily nestable:

Like lists, dictionaries can grow and shrink in place (without making a copy), they can contain objects of any type & support nesting to any depth (lists, other dictionaries & soon) of the category mutable mapping:

Instead, dictionaries are the only built-in representative of the mapping type category - objects that map keys to values.

If lists are arrays of object references, dictionaries are unordered tables of object references. Internally, dictionaries are implemented as hash tables, which start small and grow on demand. Moreover, python employs optimized hashing algorithms to find keys, so retrieval is very fast.

Common dictionary constants and Operations:

Operation

Interpretation.

`d1 = {}`

Empty dictionary

`d2 = {'spam': 2, 'eggs': 3}`

Two-item dictionary

`d3 = {'food': {'ham': 1, 'egg': 2}}`

Nesting

`d2['eggs'], d3['food']['ham']`

Indexing by key.

Each key is separated from its value by a colon (:), the items are separated by commas, and the whole thing is enclosed in curly braces.

An empty dictionary without any items is written with just two curly braces like this {}.

Accessing Values in Dictionary:

Use the square brackets along with the key to obtain its value.

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

```
print "dict['Name']:", dict['Name']
```

```
print "dict['Age']:", dict['Age']
```

Output:

`dict['Name']: Zara``dict['Age']: 7`

Updating Dictionary:

1. adding a new entry or a key-value pair,
2. modifying an existing entry, or
3. deleting an existing entry.

Eg, `dict = {'Name': 'zara', 'Age': 7, 'class': 'First'};``dict['Age'] = 8;``dict['school'] = "DPS School";``print "dict['Age']:", dict['Age'];``print "dict['school']:", dict['school'];`

Output:

`dict['Age']: 8``dict['school']: DPS School.`

Delete Dictionary Elements:

`dict = {'Name': 'zara', 'Age': 7, 'class': 'First'};``del dict['Name'];``dict.clear();``del dict;``print "dict['Age']:", dict['Age'];``print "dict['school']:", dict['school'];`



Output:

```
dict['Age']:
```

Traceback (most recent call last):

File "test.py", line 8, in <module>

```
print "dict['Age']:", dict['Age'];
```

### Built-in Dictionary Functions:

1. `len()`: The method `len()` gives the total length of the dictionary.

This would be equal to the number of items in the dictionary.

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

```
print "Length: %d" % len(dict)
```

Output: Length: 2

2. `str()`: The method `str()` produces a printable string representation of a dictionary.

```
dict = {'Name': 'zara', 'Age': 7};
```

```
print "Equivalent string: %s" % str(dict)
```

Output:

Equivalent string: {'Age': 7, 'Name': 'zara'}

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

```
dict = {'Name': 'zara', 'Age': 7};
```

```
print "variable Type: %s" % type(dict)
```

4. `clear()`: The method `clear()` removes all items from the dictionary.

```
dict = {'Name': 'zara', 'Age': 7};
```

```
print "start Len : %.d" % len(dict)
```

```
dict.clear()
```

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

Output: start len : 2

End len : 0

5. `copy()`: The method `copy()` returns a shallow copy of the dictionary.

```
dict 1 = {'Name': 'zara', 'Age': 7};
```

```
dict 2 = dict 1.copy()
```

```
print "New Dictionary : %.s" % str(dict 2).
```

Output:

New dictionary : { 'Age': 7, 'Name': 'zara' }

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

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

```
dict = dict.fromkeys(seq)
```

```
print "New dictionary : %.s" % str(dict)
```

output: New dictionary : { 'age': None, 'name': None, 'sex': None }

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

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

```
print "value : %s" % dict.get('Age')
```

Output: value : 7

8. `has_key()`: It returns true if a given key is available in the dictionary, otherwise it returns a false.

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

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

```
print "value : %s" % dict.has_key('sex')
```

Output: True  
value: False

9. `items()`: It returns a list of dict's (key, value) tuple pairs.

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

```
print "value : %s" % dict.items()
```

Output:

```
value : [ ('Age', 7), ('Name', 'Zara') ]
```

10. `keys()`: It returns a list of all the available keys in the dictionary

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

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

Output: value : ['Age', 'Name']



This is similar to `get()`, but will set `dict[key] = default` if key is not already in dict.

```
dict = {'name': 'zara', 'Age': 7}
print "value : %s" % dict.setdefault('Age', None)
```

Output:

Value : 7

12. `update()`: It adds dictionary dict2's key-values pairs in to dict. This function does not return anything.

```
dict = {'name': 'zara', 'Age': 7}
dict2 = {'sex': 'female'}
dict.update(dict2)
print "value : %s", %dict
```

Output: value = { 'Age': 7, 'Name': 'zara', 'sex': 'female' }

13. `values()`:

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

```
dict = {'name': 'zara', 'Age': 7}
print "value : %s" % dict.values()
```

Output:

value : [7, 'zara']

1. List comprehensions: In addition to sequence operations and list methods, python includes a more advanced operation known as a list comprehension expression.

[expr for iter\_var in iterable if cond\_expr]

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

```
>>> M
```

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

```
>>> M[1]
```

```
[4, 5, 6]
```

```
>>> M[1][2]
```

```
6
```

```
>>> M
```

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

```
>>> col2 = [row[1] for row in M]
```

```
>>> col2
```

```
[2, 5, 8]
```

```
>>> M
```

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

\* They are a way to build a new list by running an expression on each item in a sequence, one at a time, from left to right.

\* The preceding list comprehension means basically what it says: "Give me row[1] for each row in matrix M, in a new list." It contains a column 2 of the matrix.

```
>>> M
```

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

```
>>> [row[1] + 1 for row in M]
```

```
[3, 6, 9]
```

```
>>> M
```

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

```
>>> [row[1] for row in M if row[1] % 2 == 0]
```

```
[2, 8]
```

```
>>> diag = [M[i][i] for i in [0, 1, 2]]
```

```
>>> diag
```

```
[1, 5, 9]
```

```
>>> doubles = [c * 2 for c in 'spam']
```

```
>>> doubles
```

```
['ss', 'pp', 'aa', 'mm']
```

## 2. set Comprehensions:

\* Set comprehensions are run a loop & collect the result of an expression on each iteration.

\* A loop variable gives access to the current iteration value for use in the collection expression.

```
>>> {x**2 for x in [1, 2, 3, 4]}
```

```
{1, 4, 9, 16}
```



```
{'m', 'a', 's', 'p'}
```

```
>>> S = {c*4 for c in 'spam'}
```

```
{'aaaa', 'ssss', 'mmmm', 'pppp'}
```

### 3. Dictionary Comprehensions:

They run an implied loop, collecting the key/value results of expressions on each iteration and using them to fill out a new dictionary. Like the set comprehensions, dictionary comprehensions are available only in 3.x and 2.7.

```
>>> D = {x: x*2 for x in range(5)}
```

```
>>> D
```

```
{0: 0, 1: 2, 2: 4, 3: 6, 4: 8}
```

```
>>> D = {c: c*4 for c in 'SPAM'}
```

```
>>> D
```

```
{'s': 'ssss', 'p': 'pppp', 'A': 'AAAA', 'M': 'MMMM'}
```

```
>>> D = {c.lower(): c + '!' for c in ['SPAM', 'EGGS', 'HAM']}
```

```
>>> D
```

```
{'ham': 'HAM!', 'spam': 'SPAM!', 'eggs': 'EGGS!'}
```

```
>>> D = dict.fromkeys(['a', 'b', 'c'], 0)
```

```
>>> D
```

```
{'b': 0, 'c': 0, 'a': 0}
```

```
>>> D = {k: 0 for k in ['a', 'b', 'c']}
```

```
>>> D
```

```
{'b': 0, 'c': 0, 'a': 0}
```



>>> D

{ 'm': None, 's': None, 'a': None, 'p': None }

>>> D = { k: None for k in 'spam' }

>>> D

{ 'm': None, 's': None, 'a': None, 'p': None }

## **FREQUENTLY ASKED QUESTIONS**

### **UNIT-I**

1. What are IDLE usability features?
2. Explain about keywords used in Python.
3. Explain output function
4. Give an example of istitle( ) method
5. Describe type( ) method with example
6. Discuss about variables and assignments.
7. Explain about IDLE startup details.
8. What is indentation?
9. What is byte code?
10. Briefly discuss about running Python scripts.
11. Write the history of Python
12. Explain input function

### **UNIT-II**

1. What are 4 built-in numeric data types in Python? Explain
2. Describe Python jump statements with examples
3. What are Python assignment operators? Explain.
4. Explain about iteration statements with examples.
5. Give an example of isalnum( ) method
6. Discuss about IDLE basic usage.
7. Who uses python today? What are Python's technical strengths
8. Give an example of endswith( ) method.
9. Explain Python bitwise operators with example.
10. Discuss about Python operators precedence with example

### **UNIT-III**

1. Explain in detail about dictionaries in Python.
2. Discuss about tuples in Python
3. How to access values in a dictionary
4. Discuss about immutable constraints and frozen sets.
5. What are built-in dictionary functions? Explain.
6. Describe has\_key( ) method with example
7. What are relational operators used in Python? Explain.
8. Explain about string formatting operator with example.
9. What is a set? Why sets
10. What are built-in dictionary functions? Explain.
11. Explain about the importance of lists in Python.



## **FREQUENTLY ASKED QUESTIONS**

### **UNIT-I**

1. What are IDLE usability features?
2. Explain about keywords used in Python.
3. Explain output function
4. Give an example of istitle( ) method
5. Describe type( ) method with example
6. Discuss about variables and assignments.
7. Explain about IDLE startup details.
8. What is indentation?
9. What is byte code?
10. Briefly discuss about running Python scripts.
11. Write the history of Python
12. Explain input function

### **UNIT-II**

1. What are 4 built-in numeric data types in Python? Explain
2. Describe Python jump statements with examples
3. What are Python assignment operators? Explain.
4. Explain about iteration statements with examples.
5. Give an example of isalnum( ) method
6. Discuss about IDLE basic usage.
7. Who uses python today? What are Python's technical strengths
8. Give an example of endswith( ) method.
9. Explain Python bitwise operators with example.
10. Discuss about Python operators precedence with example

### **UNIT-III**

1. Explain in detail about dictionaries in Python.
2. Discuss about tuples in Python
3. How to access values in a dictionary
4. Discuss about immutable constraints and frozen sets.
5. What are built-in dictionary functions? Explain.
6. Describe has\_key( ) method with example
7. What are relational operators used in Python? Explain.
8. Explain about string formatting operator with example.
9. What is a set? Why sets
10. What are built-in dictionary functions? Explain.
11. Explain about the importance of lists in Python.

## 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 parantheses, 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 parantheses 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 form 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 fo 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  
add(1.5,6) #6.5
```



## Returning Multiple values from a function:

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

```
return a, b, c
```

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

```
x, y, z = functionName( )
```

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

### Example:

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

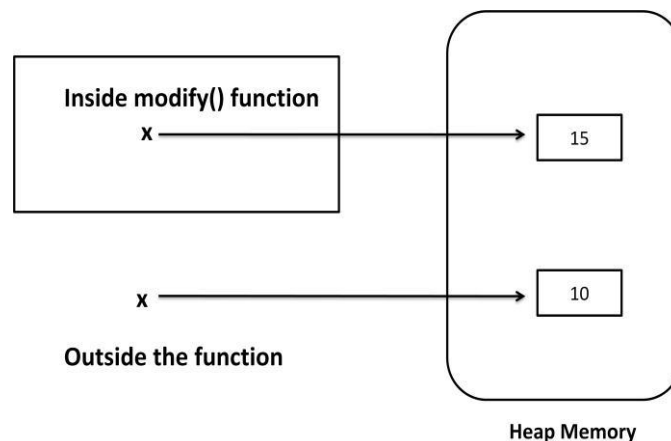
we should remember that we are passing the object references to the modify( ) function. The object is 10 and its references name is „x“. This is being passed to the modify( ) function.

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, 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)
```

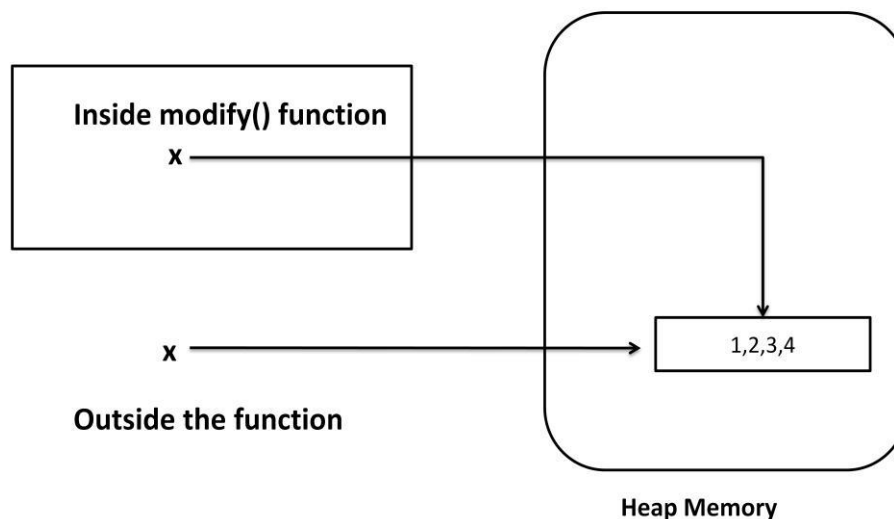
### Output:

```
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:

- Positional arguments
- Keyword arguments
- Default arguments
- 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.

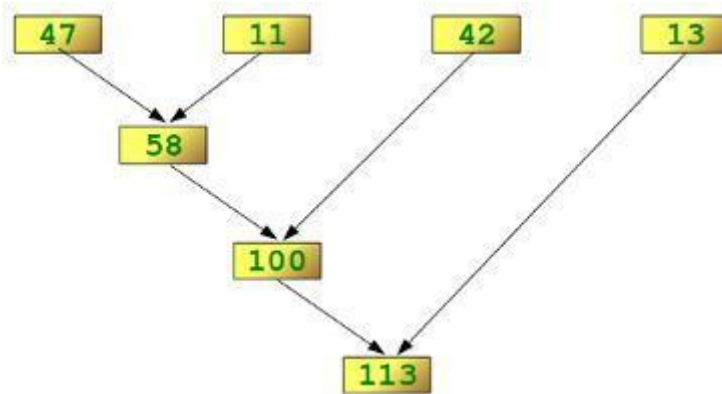
If seq = [ s<sub>1</sub>, s<sub>2</sub>, s<sub>3</sub>, ... , s<sub>n</sub> ], calling reduce(func, seq) works like this:

- At first the first two elements of seq will be applied to func, i.e. func(s<sub>1</sub>,s<sub>2</sub>) The list on which reduce() works looks now like this: [ func(s<sub>1</sub>, s<sub>2</sub>), s<sub>3</sub>, ... , s<sub>n</sub> ]
- In the next step func will be applied on the previous result and the third element of the list, i.e. func(func(s<sub>1</sub>, s<sub>2</sub>),s<sub>3</sub>). The list looks like this now: [ func(func(s<sub>1</sub>, s<sub>2</sub>),s<sub>3</sub>), ... , s<sub>n</sub> ]
- 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 favorite text editor to create a file called `fibonacci.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 than 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 to go to 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:

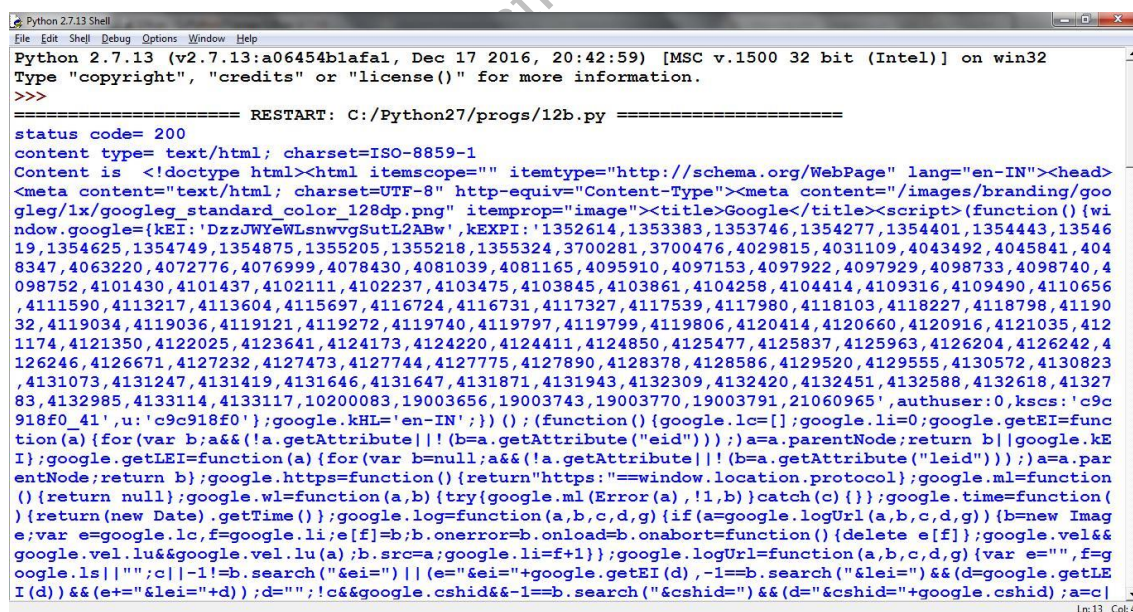
- ∞ **Command:** cd C:\Python27\Scripts
- ∞ **Command:** pip install requests

#### ➤ 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
```



```
Python 2.7.13 Shell
File Edit Shell Debug Options Window Help
Python 2.7.13 (v2.7.13:a06454b1afal, Dec 17 2016, 20:42:59) [MSC v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:/Python27/progs/12b.py =====
status code= 200
content type= text/html; charset=ISO-8859-1
Content is <!doctype html><html itemscope="" itemtype="http://schema.org/WebPage" lang="en-IN"><head>
<meta content="text/html; charset=UTF-8" http-equiv="Content-Type"><meta content="/images/branding/goog
gle/1x/google_standard_color_128dp.png" itemprop="image"><title>Google</title><script>(function() {wi
ndow.google={kEI: 'DzzJWYeWLSnvwgSutL2ABw', kEXPI: '1352614,1353383,1353746,1354277,1354401,1354443,13546
19,1354625,1354749,1354875,1355205,1355218,1355324,3700281,3700476,4029815,4031109,4043492,4045841,404
8347,4063220,4072776,4076999,4078430,4081039,4081165,4095910,4097153,4097922,4097929,4098733,4098740,4
098752,4101430,4101437,4102111,4102237,4103475,4103845,4103861,4104258,4104414,4109316,4109490,4110656
,4111590,4113217,4113604,4115697,4116724,4116731,4117327,4117539,4117980,4118103,4118227,4118798,41190
32,4119034,4119036,4119121,4119272,4119740,4119797,4119799,4119806,4120414,4120660,4120916,4121035,412
1174,4121350,4122025,4123641,4124173,4124220,4124411,4124850,4125477,4125837,4125963,4126204,4126242,4
126246,4126671,4127232,4127473,4127744,4127775,4127890,4128378,4128586,4129520,4129555,4130572,4130823
,4131073,4131247,4131419,4131646,4131647,4131871,4131943,4132309,4132420,4132451,4132588,4132618,41327
83,4132985,4133114,4133117,10200083,19003656,19003743,19003770,19003791,21060965', authuser:0, kscs: 'c9c
918f0_41', u: 'c9c918f0'}; google.kHL='en-IN'; }) (); (function() {google.lc=[]; google.li=0; google.getEI=func
tion(a) {for (var b;a&&(!a.getAttribute)|| (b=a.getAttribute("eid")));) a=a.parentNode; return b} | google.kE
I); google.getLEI=function(a) {for (var b=null;a&&(!a.getAttribute)|| (b=a.getAttribute("leid")));) a=a.par
entNode; return b}; google.https=function() {return "https://" + window.location.protocol}; google.ml=function
() {return null}; google.wl=function(a,b) {try {google.ml(Error(a), !1, b)} catch (c) {} }; google.time=function(
) {return (new Date).getTime()}; google.log=function(a,b,c,d,g) {if (a=google.logUrl(a,b,c,d,g)) {b=new Imag
e; var e=google.lc, f=google.li; e[f]=b; b.onerror=b.onload=b.onabort=function() {delete e[f]}; google.vel&&
google.vel.lu&&google.vel.lu(a); b.src=a; google.li=f+1}; google.logUrl=function(a,b,c,d,g) {var e="", f=g
oogle.ls||""; c|| !1==b.search("&ei=") || (e="&ei="+google.getEI(d), -1==b.search("&lei=") && (d=google.getLE
I(d) && (e+="&lei="+d)); d=""; c&&google.cshid&& -1==b.search("&cshid=") && (d="&cshid="+google.cshid); a=c|
```

There are some libraries in python:

- **Requests:** The most famous HTTP Library. It is a must and an essential criterion for every 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.
- **SQLAlchemy:** 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>
```

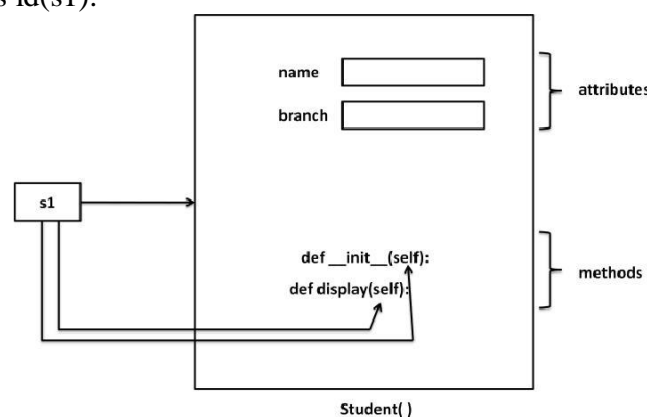
Class definitions, like function definitions (def statements) must be executed before they have any effect. (You could conceivably place a class definition in a branch of an if statement, or inside a function.)

### 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.
- Observe 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 contains the memory location 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 ways:

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

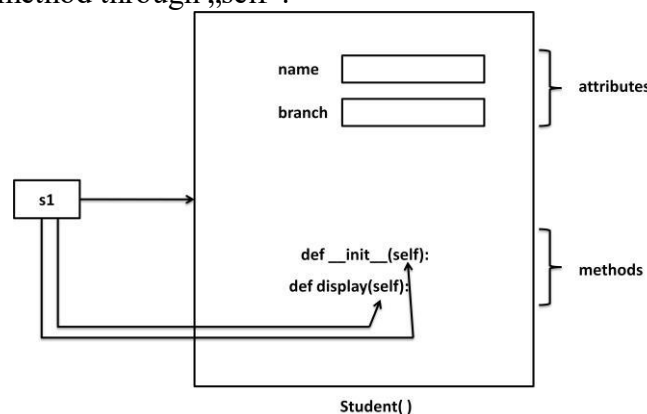
```
def __init__(self):
```

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.

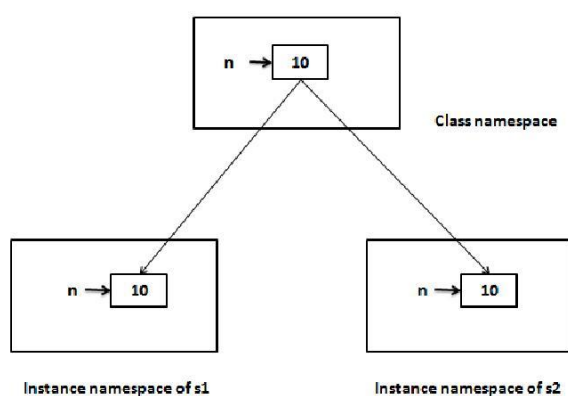
**a) Class Namespace:**

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.

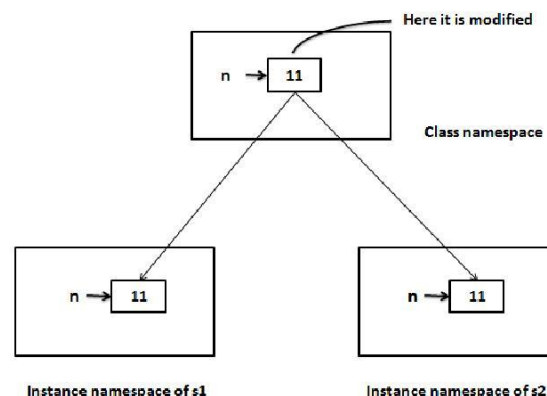
**Example:**

```
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:**

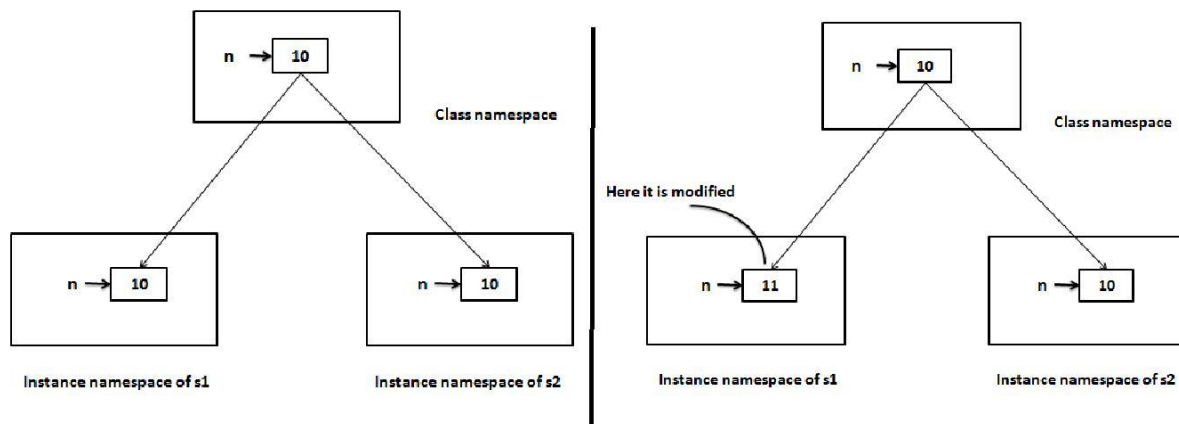
Every instance will have its own name space, called „instance namespace“. In the instance namespace, the names are mapped to instance variables. Every instance will have its own namespace, if the class variable is modified in one instance namespace, it will not affect the variables in the other instance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

- Instance methods
  - Accessor methods
  - Mutator methods
- Class methods
- 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 variables 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 or 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 "sparrow 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 easily. 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 create an object, we can access the members of both the super and sub classes.
- `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 sub 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)
s.display()
```

**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

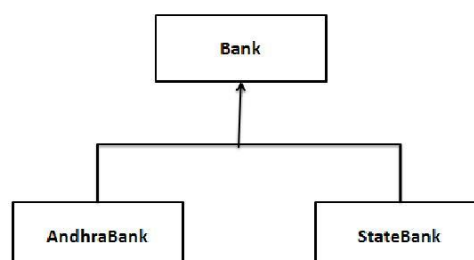
### Types of Inheritance:

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 class from where we derive „AndhraBank“ and „StateBank“ as sub classes. This is called single inheritance.



**Example:**

```

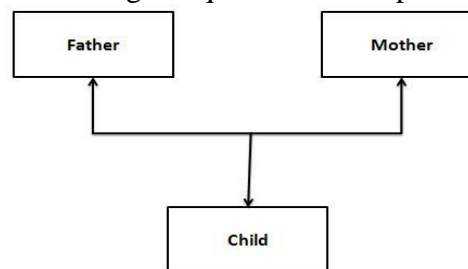
class Bank:
    cash = 100
    @classmethod
    def balance(cls):
        print cls.cash
class AndhraBank(Bank):
    cash = 500
    @classmethod
    def balance(cls):
        print "AndhraBank",cls.cash +
Bank.cash
class StateBank(Bank):
    cash = 300
    @classmethod
    def balance(cls):
        print "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 inchehs"
class Mother:
    def color(self):
        print "Color is brown"
class Child(Father, Mother):
    pass
c=Child()
c.height() # displays Height is 5.8 inchehs
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(self, 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 within 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 or the design of the program itself is wrong. Logical errors are not detected either by Python compiler or PVM. The programmer 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.

➤

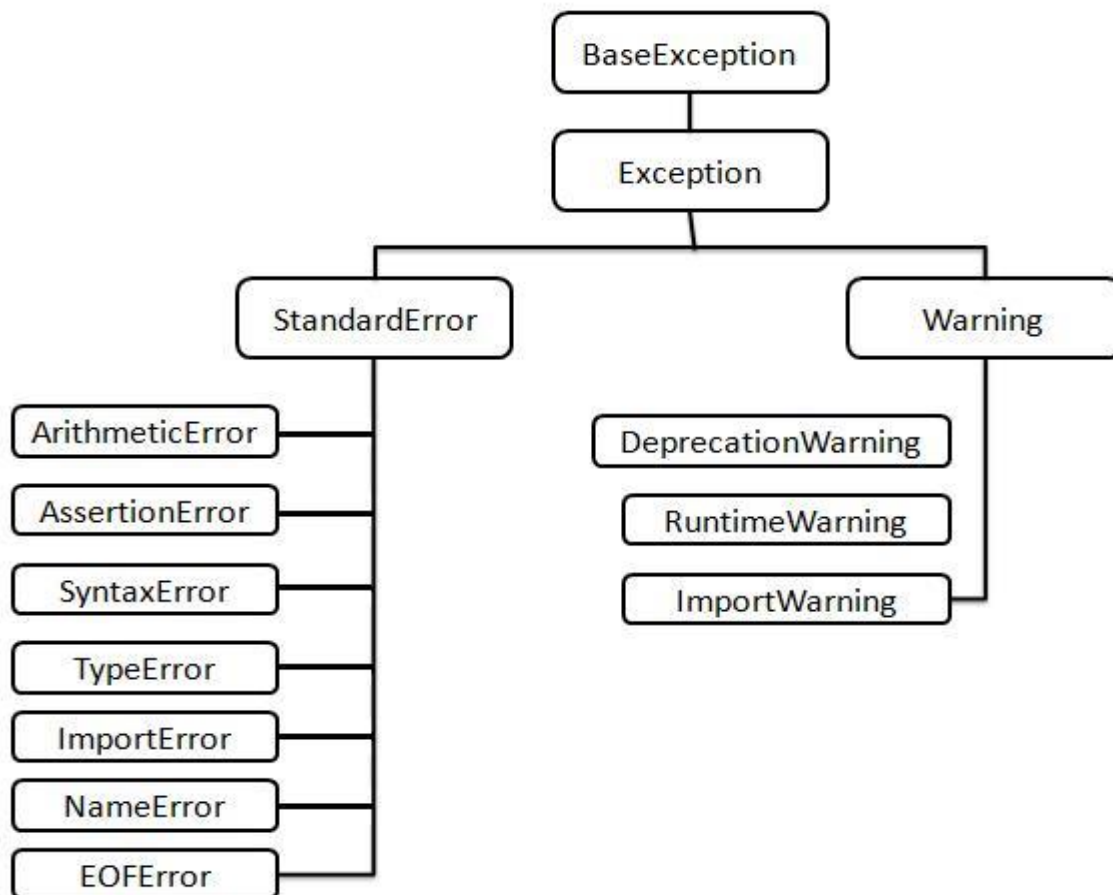
Similarly, all warnings are derived as sub classes from „Warning“ class. A warning represents a caution and even though it is not handled, the program will execute. So, warnings can be neglected but errors cannot neglect.

➤

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
<code>group(num=0)</code>	This method returns entire match (or specific subgroup num)
<code>groups()</code>	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
<code>group(num=0)</code>	This method returns entire match (or specific subgroup num)
<code>groups()</code>	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
<code>^</code>	Matches beginning of line.
<code>\$</code>	Matches end of line.
<code>.</code>	Matches any single character except newline. Using <code>m</code> option allows it to match newline as well.
<code>[...]</code>	Matches any single character in brackets.
<code>[^...]</code>	Matches any single character not in brackets
<code>re*</code>	Matches 0 or more occurrences of preceding expression.
<code>re+</code>	Matches 1 or more occurrence of preceding expression.
<code>re?</code>	Matches 0 or 1 occurrence of preceding expression.
<code>re{ n}</code>	Matches exactly n number of occurrences of preceding expression.
<code>re{ n, }</code>	Matches n or more occurrences of preceding expression.

<code>re{ n, m}</code>	Matches at least n and at most m occurrences of preceding expression.
<code>a  b</code>	Matches either a or b.
<code>(?#...)</code>	Comment.
<code>(?= re)</code>	Specifies position using a pattern. Doesn't have a range.
<code>(?! re)</code>	Specifies position using pattern negation. Doesn't have a range.
<code>(?&gt; re)</code>	Matches independent pattern without backtracking.
<code>\w</code>	Matches word characters.
<code>\W</code>	Matches nonword characters.
<code>\s</code>	Matches whitespace. Equivalent to <code>[\t\n\r\f]</code> .
<code>\S</code>	Matches nonwhitespace.
<code>\d</code>	Matches digits. Equivalent to <code>[0-9]</code> .
<code>\D</code>	Matches nondigits.
<code>\A</code>	Matches beginning of string.
<code>\Z</code>	Matches end of string. If a newline exists, it matches just before newline.
<code>\z</code>	Matches end of string.
<code>\G</code>	Matches point where last match finished.
<code>\B</code>	Matches nonword boundaries.
<code>\n, \t, etc.</code>	Matches newlines, carriage returns, tabs, etc.
<code>\1...\9</code>	Matches nth grouped subexpression.
<code>\10</code>	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
<code>ceil(x)</code>	Returns the smallest integer greater than or equal to x.
<code>copysign(x, y)</code>	Returns x with the sign of y
<code>fabs(x)</code>	Returns the absolute value of x
<code>factorial(x)</code>	Returns the factorial of x
<code>floor(x)</code>	Returns the largest integer less than or equal to x
<code>fmod(x, y)</code>	Returns the remainder when x is divided by y
<code>frexp(x)</code>	Returns the mantissa and exponent of x as the pair (m, e)
<code>fsum(iterable)</code>	Returns an accurate floating point sum of values in the iterable
<code>isfinite(x)</code>	Returns True if x is neither an infinity nor a NaN (Not a Number)
<code>ldexp(x, i)</code>	Returns $x * (2^{**i})$
<code>modf(x)</code>	Returns the fractional and integer parts of x

<code>trunc(x)</code>	Returns the truncated integer value of x
<code>exp(x)</code>	Returns $e^{**}x$
<code>expm1(x)</code>	Returns $e^{**}x - 1$
<code>log(x[, base])</code>	Returns the logarithm of x to the base (defaults to e)
<code>log10(x)</code>	Returns the base-10 logarithm of x
<code>pow(x, y)</code>	Returns x raised to the power y
<code>sqrt(x)</code>	Returns the square root of x
<code>atan2(y, x)</code>	Returns $\text{atan}(y / x)$
<code>cos(x)</code>	Returns the cosine of x
<code>hypot(x, y)</code>	Returns the Euclidean norm, $\text{sqrt}(x^{**}x + y^{**}y)$
<code>sin(x)</code>	Returns the sine of x
<code>tan(x)</code>	Returns the tangent of x
<code>degrees(x)</code>	Converts angle x from radians to degrees
<code>radians(x)</code>	Converts angle x from degrees to radians
<code>acosh(x)</code>	Returns the inverse hyperbolic cosine of x
<code>asinh(x)</code>	Returns the inverse hyperbolic sine of x
<code>atanh(x)</code>	Returns the inverse hyperbolic tangent of x
<code>cosh(x)</code>	Returns the hyperbolic cosine of x
<code>sinh(x)</code>	Returns the hyperbolic cosine of x
<code>tanh(x)</code>	Returns the hyperbolic tangent of x
<code>erf(x)</code>	Returns the error function at x
<code>erfc(x)</code>	Returns the complementary error function at x
<code>gamma(x)</code>	Returns the Gamma function at x
<code>lgamma(x)</code>	Returns the natural logarithm of the absolute value of the Gamma function at x
<code>pi</code>	Mathematical constant, the ratio of circumference of a circle to its diameter (3.14159...)
<code>e</code>	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	<b>time.ctime([secs])</b> Like <code>asctime(localtime(secs))</code> and without arguments is like <code>asctime()</code> ) <b>time.localtime([secs])</b>
2	Accepts an instant expressed in seconds since the epoch and returns a time-tuple <code>t</code> with the local time ( <code>t.tm_isdst</code> is 0 or 1, depending on whether DST applies to instant <code>secs</code> by local rules).
3	<b>time.sleep(secs)</b> Suspends the calling thread for <code>secs</code> seconds.
4	<b>time.time()</b> Returns the current time instant, a floating-point number of seconds since the epoch. <b>time.clock()</b>
5	The method returns the current processor time as a floating point number expressed in seconds on <b>Unix</b> .
6	<b>time.asctime([tupletime])</b> 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	<b>calendar.calendar(year,w=2,l=1,c=6)</b> Returns a multiline string with a calendar for year formatted into three columns separated by <code>c</code> spaces. <code>w</code> is the width in characters of each date; each line has length <code>21*w+18+2*c</code> . <code>l</code> is the number of lines for each week.
2	<b>calendar.isleap(year)</b> Returns True if year is a leap year; otherwise, False.
3	<b>calendar.setfirstweekday(weekday)</b> Sets the first day of each week to weekday code <code>weekday</code> . Weekday codes are 0 (Monday) to 6 (Sunday).

4	<b>calendar.leapdays(y1,y2)</b> Returns the total number of leap days in the years within range(y1,y2).
5	<b>calendar.month(year,month,w=2,l=1)</b> 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”

```
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 are 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 *args*. 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

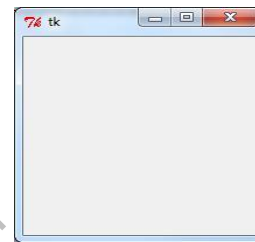
Tkinter is the standard GUI library for Python. Python when combined with Tkinter provides a fast and easy way to create GUI applications. Tkinter provides a powerful object-oriented interface to the Tk GUI toolkit.

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.



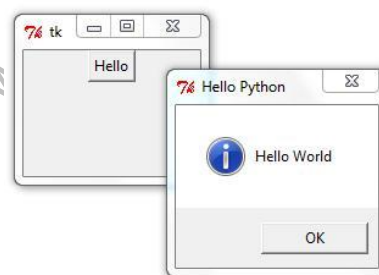
Message	The Message widget is used to display multiline text fields for accepting values from a user.
Radiobutton	The Radiobutton widget is used to display a number of options as radio buttons. The user can select only one option at a time. Scale The Scale widget is used to provide a slider widget.
Scrollbar	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**

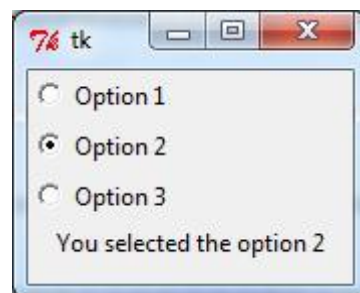
- This widget implements a multiple-choice button, which is a way to offer many possible selections to the user and lets user choose only one of them.
- In order to implement this functionality, each group of radiobuttons must be associated to the same variable and each one of the buttons must symbolize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

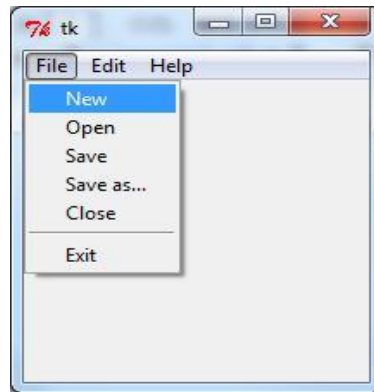
filemenu = Menu(menubar, tearoff=0)
filemenu.add_command(label="New", command=donothing)
filemenu.add_command(label="Open", command=donothing)
filemenu.add_command(label="Save", command=donothing)
filemenu.add_command(label="Save as...",
command=donothing) filemenu.add_command(label="Close",
command=donothing) 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)

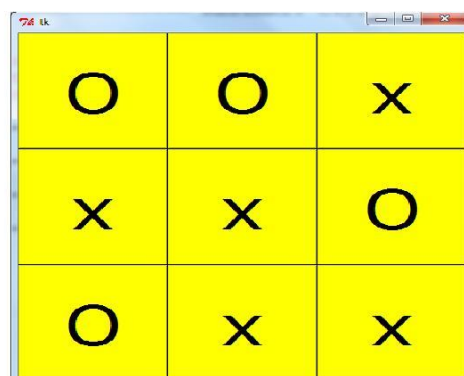
root.config(menu=menubar)
root.mainloop()
```

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 clear():  
    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)

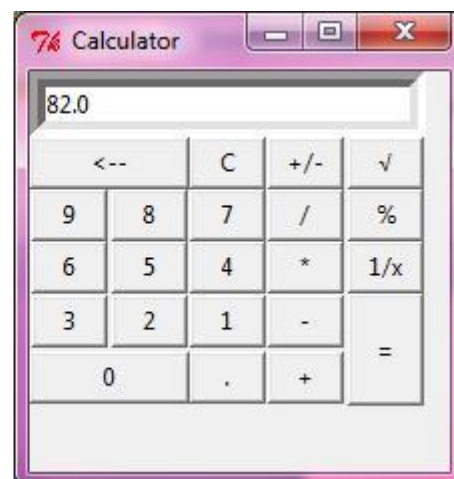
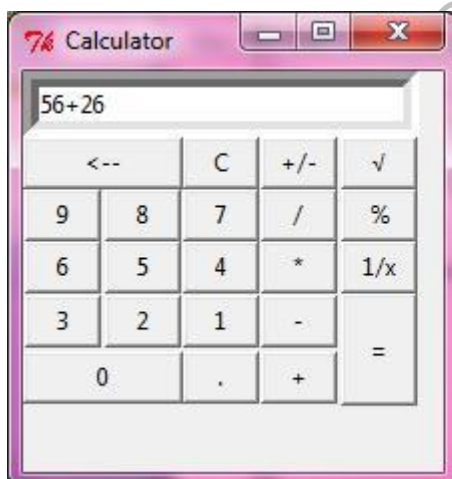
plus=Button(root,text="+",command=lambda:setText("+"),width=4)
plus.grid(row=5,column=4)

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 <sup>0</sup> .
seth	Sets the orientation of the turtle to to_angle.

## Turtle Object:

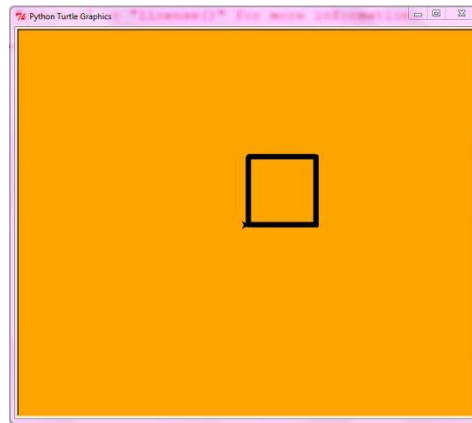
**t=Turtle( )** creates a new turtle object and open sits window. The window's drawing area is 200 pixels wide and 200 pixels high.

**t=Turtle( width, height )** creates a new turtle object and open sits window. The window's drawing area has given width and height.



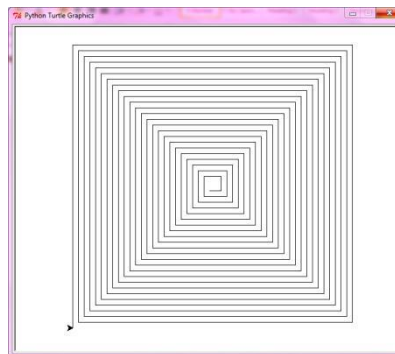
**Example-1:** Write a program to draw

```
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)
```



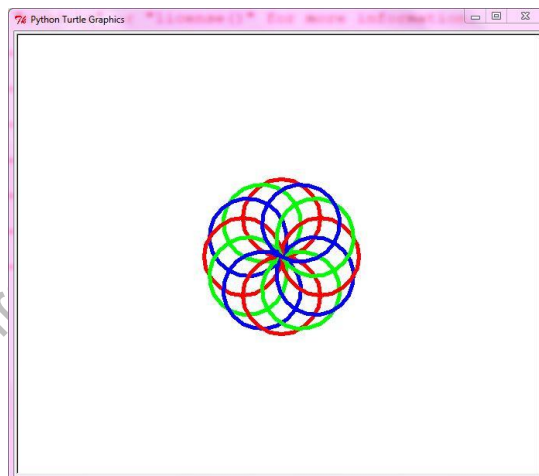
**Example-2:**

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



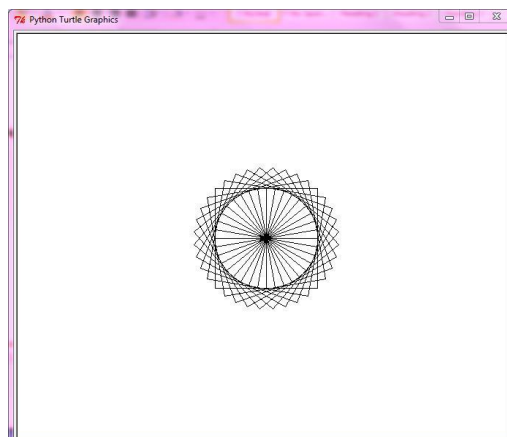
**Example-3:**

```
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
<code>assertEqual(a, b)</code>	<code>a == b</code>
<code>assertNotEqual(a, b)</code>	<code>a != b</code>
<code>assertTrue(x)</code>	<code>bool(x) is True</code>
<code>assertFalse(x)</code>	<code>bool(x) is False</code>
<code>assertIs(a, b)</code>	<code>a is b</code>
<code>assertIsNot(a, b)</code>	<code>a is not b</code>
<code>assertIsNone(x)</code>	<code>x is None</code>
<code>assertIsNotNone(x)</code>	<code>x is not None</code>
<code>assertIn(a, b)</code>	<code>a in b</code>
<code>assertNotIn(a, b)</code>	<code>a not in b</code>
<code>assertIsInstance(a, b)</code>	<code>isinstance(a, b)</code>
<code>assertNotIsInstance(a, b)</code>	<code>not isinstance(a, b)</code>

**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
```

[www.FirstRanker.com](http://www.FirstRanker.com)