|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 16-Jan-2017 **Python**  Praveen  • Python is An interpreted high-level programming language  • Similar to Perl, Ruby, Tcl, and other so-called "scripting languages"  • Created by Guido van Rossum around 1990  • Named in honor of Monty Python  • Python programs run inside an interpreter  **Python Versions**  • Version 2.X (most common)  • Version 3.X (bleeding edge, the future)  **Some Uses of Python**  • Simple  • open source(GPL)  • Higher level language  • Portable  • Embedded on C/C++  • Combining C and C++  • Automatic garbage collection  • Scripting language (if we want to convert code into byte code also we can)  • Text processing/data processing  • Application scripting  • Systems administration/programming  • Internet programming  • Graphical user interfaces  • Testing  • Writing quick "throw-away" code  In three different ways we can execute python  1.Interactive mode  2.Script mode  3.IDE  **Python Interpreter :-**  • When you start Python, you get an "interactive" mode where you can experiment  • If you start typing statements, they will run immediately  • No edit/compile/run/debug cycle  • In fact, there is no "compiler" Interactive Mode  • The interpreter runs a "read-eval" loop  >>> print "hello world"  hello world  >>> 37\*42 ( “>>>” is the interpreter prompt for starting a new statement)  >>> 1554  • Executes simple statements typed in directly  • Very useful for debugging, exploration  • To know the python location  $which python  /usr/bin/python  • help(name) command to getting help  **Creating Programs (script file and IDE tools)**  • Programs are put in .py files  # helloworld.py  print "hello world"  • Source files are simple text files  • Create with your favorite editor (e.g., vi)  • Can also edit programs with IDLE or other Python IDE (too many to list)  • In production environments, Python may be run from command line or a script  • Command line (Unix)  bash % python helloworld.py  hello world  bash %  **Unix and Linux Installation**  Here are the simple steps to install Python on Unix/Linux machine.  • Open a Web browser and go to http://www.python.org/download/ .  • Follow the link to download zipped source code available for Unix/Linux.  • Download and extract files.  • Editing the Modules/Setup file if you want to customize some options.  • run ./configure script  • make  • make install  This installs Python at standard location /usr/local/bin and its libraries at /usr/local/lib/pythonXX where XX is the version of Python.  **Statements**  • A Python program is a sequence of statements  • Each statement is terminated by a newline  • Statements are executed one after the other until you reach the end of the file.  • When there are no more statements, the program stops  **Comments**  • Comments are denoted by #  # This is a comment height = 442  # Meters  • It can support for multi line comment to Extend include ” / “ to the end of the line  • First line should be shabang line(#! /usr/bin.python) it doesn't take 1st line as comment  • There are no block comments in Python (e.g., /\* ... \*/).  **Variables**  • A variable is just a name for some value  • Variable names follow same rules as C language [A-Za-z\_][A-Za-z0-9\_]\*  • You do not declare types (int, float, etc.)  >> height = 442 # An integer  >> height = 442.0 # Floating point  >> height = "Really tall" # A string  • Differs from C++/Java where variables have a fixed type that must be declared.  • Python allows you to assign multiple objects to multiple variables.  >>a, b, c = 1, 2, "john" #a=1 /b=2 /c=”john”  • You can delete a single object or multiple objects by using the del statement.  >>del var  >>del var\_a, var\_b  **Keywords**  • Python has a basic set of language keywords  and as assert break class continue def del elif else except exec finally for from global if import in is lambda not or pass print raise return ry while with yield  • Variables can not have one of these names  • These are mostly C-like and have the same meaning in most cases (later)  **Printing**  • The print statement  >>print x  >>print x,y,z  >>print "Your name is", name  >>print x, # Omits newline  • Produces a single line of text  • Items are separated by spaces  • Always prints a newline unless a trailing comma is added after last item.  **Case Sensitivity**  • Python is case sensitive  • These are all different variables:  >>name = "Jake"  >>Name = "Elwood"  >>NAME = "Guido"  • Language statements are always lower-case  >>print "Hello World" # OK  >>PRINT "Hello World" # ERROR  >>while x < 0: # OK  >>WHILE x < 0: # ERROR  **Cleaning up**  • Python has garbage collection  • Values are destroyed when no longer used  >>s = "Guido"  >>s = 42 # Previous value destroyed  • Or you can delete manually  >>del s  **Indentation**  • Indentation used to denote blocks of code  • Indentation must be consistent  >>while num\_bills \* bill\_thickness < sears\_height:  >>print day, num\_bills, num\_bills \* bill\_thickness  >> day = days + 1 (error)  >>num\_bills = num\_bills \* 2  • Colon (:) indicates the start of a block  >>while num\_bills \* bill\_thickness < sears\_height:  pass statement  • Sometimes you will need to specify an empty block of code (like {} in C/Java)  >>if name in namelist:  # Not implemented yet (or nothing)  >> pass  >>else:  >> statements  • pass is a "no-op" statement  • It does nothing, but serves as a placeholder for statements (possibly to be added later)  **Long Lines**  • Sometimes you get long statements that you want to break across multiple lines  • Use the line continuation character (\)  >>if product=="game" and type=="pirate memory" \  >>and age >= 4 and age <= 8:  >>print "I'll take it!"  • However, not needed for code in (), [], or {}  >>if (product=="game" and type=="pirate memory"  >>and age >= 4 and age <= 8):  >>print "I'll take it!"  **Basic Datatypes**  • Python only has a few primitive types of data  • Numbers  • Strings (character text)  • Lists  **Numbers**  • Python has 4 types of numbers  • Booleans  • Integers  • Floating point  • Complex (imaginary numbers)  **Integers**  • Signed values of arbitrary size  >>a = 37  >>b = -299392993727716627377128481812241231  >>c = 0x7fa8 # Hexadecimal  >>d = 0o253 # Octal  >>e = 0b10001111 # Binary  • There are two internal representations  • int : Small values (less than 32-bits in size)  • long : Large values (arbitrary size)  • Sometimes see 'L' shown on end of large values  >>> b  -299392993727716627377128481812241231L  **Floating point (float)**  • Use a decimal or exponential notation  >>a = 37.45  >>b = 4e5  >>c = -1.345e-10  • Represented as double precision using the native CPU representation (IEEE 754) 17 digits of precision Exponent from -308 to 308  • Same as the C double type  **String Representation**  • Strings work like an array : s[n]  a = "Hello world"  print a[0] # it prints 'H'  print a[4] # it prints 'o'  print a[-1] # it prints 'd'  print a[:5] # it prints "Hello"  print a[6:] # it prints "world"  print a[3:8] # it prints "lo wo"  print a[-5:] #it prints "world"  • Slicing/substrings : s[start:end]  • Strings are "immutable" (read only)  • Once created, the value can't be changed  >>> s = "Hello World"  >>> s[1] = 'a'  Traceback (most recent call last):  File "<stdin>", line 1, in <module>  TypeError: 'str' object does not support item assignment  >>>  • All operations and methods that manipulate string data always create new strings.  **Lists**  • lists are similar to arrays in C only the difference is that all the items belonging to a list  can be of different data type.  • list contains items separated by commas and enclosed within square brackets ([]).  • Lists are indexed by integers (starting at 0)  >>names = [ "Elwood", "Jake", "Curtis" ]  >>names[0]  "Elwood"  >>names[1]  "Jake"  >>names[2]  "Curtis"  • Negative indices are from the end  >>names[-1]  "Curtis"  • Changing one of the items  >> names[1] = "Joliet Jake"  • Concatenation (+)  >>a = "Hello" + "World"  >>b = "Say " + a.  **Tuples**  • Tuple consists of a number of values separated by commas like lists.  • Tuples are enclosed within parentheses ( ).  • Tuples elements can not updated and Tuples can be thought of as read- only lists.  Example :-  #!/usr/bin/python  tuple = ( 'abcd', 786 , 2.23, 'john', 70.2 )  tinytuple = (123, 'john')  print tuple # Prints complete list  print tuple[0] # Prints first element of the list  print tuple[1:3] # Prints elements starting from 2nd till 3rd  print tuple[2:] # Prints elements starting from 3rd element  print tinytuple \* 2 # Prints list two times  print tuple + tinytuple # Prints concatenated lists  **Dictionaries**  • A hash table or associative array  • A collection of values indexed by "keys"  • The keys serve as field names  • Example: dict= { 'name' : 'GOOG', 'shares' : 100, 'price' : 490.10 }  • Getting values: Just use the key names  >>> print s['name'],s['shares']  GOOG 100  >>> s['price']  490.10  • Adding/modifying values : Assign to key names  >>> s['shares'] = 75  >>> s['date'] = '6/6/2007'  • Deleting a value  >>> del s['date']  **User Input**  • To read a line of typed user-input  name = raw\_input("Enter your name:") # it will take i/p as string(only string)  name = input(“Enter your name:”) #it will take i/p as based on data(int ,float,string)  • In Python, you must be careful about converting data to an appropriate type  >>x = '37'  >>y = '42' # Strings  >>z = x + y # z = '3742' (concatenation)  >>x = 37  >>y = 42  >>z = x + y # z=79 (integer +)  • Prints a prompt, returns the typed response  • This might be useful for small programs or for simple debugging  • It is not widely used for real programs  **Data Type Conversion**  There are several built-in functions to perform conversion from one data type to another.   |  |  | | --- | --- | | Function | Description | | int(x [,base]) | Converts x to an integer. base specifies the base if x is a string. | | long(x [,base] ) | Converts x to a long integer. base specifies the base if x is a string. | | float(x) | Converts x to a floating-point number. | | complex(real [,imag]) | Creates a complex number. | | str(x) | Converts object x to a string representation. | | repr(x) | Converts object x to an expression string. | | eval(str) | Evaluates a string and returns an object. | | tuple(s) | Converts s to a tuple. | | list(s) | Converts s to a list. | | set(s) | Converts s to a set. | | dict(d) | Creates a dictionary. d must be a sequence of (key,value)  tuples. | | frozenset(s) | Converts s to a frozen set. | | chr(x) | Converts an integer to a character. | | unichr(x) | Converts an integer to a Unicode character. | | oct(x) | Converts an integer to an octal string. | | hex(x) | Converts an integer to a hexadecimal string. |   Example:-  #! /usr/bin/python  val1="123"  cval='a'  ival=65  #repr\_val1=repr(val1) #converts string to expression string  eval\_val1=eval(val1) #evaluate a string and return an object  tup\_val1=tuple(val1) #converts string to tuple  lis\_val1=list(val1) #converts string to list  set\_val1=set(val1) #converts string to set  #dict\_val1=dict(val1) # converts string to dict  froz\_val1=frozenset(val1) #coverts string to frozenset  chr\_val=chr(ival) #converts int to character  unichr\_val=unichr(ival) #converts int to a unicode character (ascii)  ord\_cval=ord(cval) #converts a signle cahracter to its integer value  print"\n\n"  print"string expression of val1",repr(val1)  print"evaluate a string and return an object val1",eval\_val1  print"converts string to tuple",tup\_val1  print"converts string to list val1",lis\_val1  print"converts string to set",set\_val1  #print"converts string to dict val1",dict\_val1  print"coverts string to frozenset val1",froz\_val1  print"converts int to character ival1",chr\_val  print"converts int to a unicode character (ascii) ival",unichr\_val  print"converts a signle cahracter to its integer value cval1",ord\_cval  print"\n\n"  **Types of Operators :-**  Python language supports the following types of operators.  • Arithmetic Operators (  • Comparison (Relational) Operators  • Assignment Operators  • Logical Operators  • Bitwise Operators  • Membership Operators  • Identity Operators  **Arithmetic Operators:-**  example:-  #!/usr/bin/python  val1 = 21  val2= 10  val3 = 0  val3= val1 + val2  print "Value of val3 is ", val3 #Value of val3 is 31  val3= val1 - val2  print "Value of val3 is ", val3 #Value of val3 is 11  val3 = val1 \* val2  print "Value of val3 is ", val3 #Value of val3 is 210  val3 = val1 / val2  print "Value of val3 is ", val3 # Value of val3 is 2  val3 = val1 % val2  print "Value of val3 is ", val3 #Value of val3 is 1  val1 = 2  val2 = 3  val3= val1\*\*val2  print "Value of val3 is ", val3 #Value of val3 is 8  val1 = 10  val2 = 5  val3 = val1// val2  print "Value of val3 is ", val3 #Value of val3 is 2  **Comparison Operators:-**  #!/usr/bin/python  v1 = 11  v2 = 5  v3 = 0  if ( v1 == v2 ):  print "v1 is equal to v2"  else:  print "v1 is not equal to v2" #v1 is not equal to v2  if ( v1 != v2):  print " v1 is not equal to v2" #v1 is not less than v2  else:  print " v1 is equal to v2"  if ( v1<> v2 ):  print " v1 is not equal to v2" #v1 is not equal to v2  else:  print " v1 is equal to b"  if ( v1< v2 ):  print " v1 is less than v2" #v1 is not less than v2  else:  print " v1 is not less than v2"  if ( v1 > v2 ):  print " v1 is greater than v2"  else:  print " v1 is not greater than v2" #v1 is greater than v2  v1 = 1;  v2 = 10;  if ( v1 <= v2 ):  print " v1 is either less than or equal to v2" #v1 is either less than or equal to v2  else:  print " v1 is neither less than nor equal to v2"  if ( v2 >= v1 ):  print " v2 is either greater than or equal to v2"  else:  print "L v2 is neither greater than nor equal to v2" # v2 is either greater than or equal  to v2  **Assignment Operators :-**  example;-  #! /usr/bin/python  n1=input("Enter the number n1 = ")  print n1  v1=input("\nEnter the number v1= ")  print v1  v2=input("\nEnter the number v2= ")  print v2  v3=input("\nEnter the number v3= ")  print v3  v4=input("\nEnter the number v4= ")  print v4  v5=input("\nEnter the number v5= ")  print v5  v6=input("\nEnter the number v6= ")  print v6  v7=input("\nEnter the number v7= ")  print v7    v1+=n1  v2-=n1  v3\*=n1  v4\*\*=n1  v5/=n1  v6//=n1  v7%=n1  print "\n(v1+=n1) = ",v1  print "\n(v2!=n1) = ",v2  print "\n(v3\*=n1) = ",v3  print "\n(v4\*\*=n1) = ",v4  print "\n(v5/=n1) = ",v5  print "\n(v6//=n1) = ",v6  print "\n(v7%=n1) = ",v7  #Example 1  Enter the number n1 = 2  Enter the number v1= 2  Enter the number v2= 2  Enter the number v3= 2  Enter the number v4= 2  Enter the number v5= 2  Enter the number v6= 2  Enter the number v7= 2  (v1+=n1) = 4  (v2!=n1) = 0  (v3\*=n1) = 4  (v4\*\*=n1) = 4  (v5/=n1) = 1  (v6//=n1) = 1  (v7%=n1) = 0  #Example output:-  Enter the number n1 = 2  Enter the number v1= -4  Enter the number v2= -6  Enter the number v3= -8  Enter the number v4= 0  Enter the number v5= 1  Enter the number v6= 3  Enter the number v7= 2  (v1+=n1) = -2  (v2!=n1) = -8  (v3\*=n1) = -16  (v4\*\*=n1) = 0  (v5/=n1) = 0  (v6//=n1) = 1  (v7%=n1) = 0  **Bitwise Operators**  Bitwise operator works on bits and performs bit by bit operation.  example:-  #!/usr/bin/python  val1 = 60 # 60 = 0011 1100  val2= 13 # 13 = 0000 1101  val3 = 0  val3 = val1 & val2; # 12 = 0000 1100  print " Value of val3 is ", val3  val3 = val1 | val2; # 61 = 0011 1101  print " Value of val3 is ", val3  val3= val1 ^ val2; # 49 = 0011 0001  print " Value of val3 is ", val3  val3 = ~ val1; # -61 = 1100 0011  print " Value of val3 is ", val3  val3 = val1 << 2; # 240 = 1111 0000  print " Value of val3 is ", val3  val3 = val1 >> 2; # 15 = 0000 1111  print " Value of val3 is ", val3  **Booleans**  • Two values: True, False  >>a = True  >>b = False  • Evaluated as integers with value 1,0  >>c = 4 + True # c = 5  >>d = False  >>if d == 0:  print "d is False"  • Although doing that in practice would be odd  • Be aware that floating point numbers are inexact when representing decimal values.  >>> a = 2.1 + 4.2  >>> a == 6.3  False  >>> a  6.300000000000001  >>>  • This is not Python, but the underlying floating point hardware on the CPU.  • The result of a calculation may not be quite what you expect (again, not a Python bug)  **Logical Operators**  example:-  #!/usr/bin/python  a = 10  b = 20  list = [1, 2, 3, 4, 5 ];  if ( a in list ):  print "a is available in the given list"  else:  print " a is not available in the given list"  if ( b not in list ):  print " b is not available in the given list"  else:  print " b is available in the given list"  a = 2  if ( a in list ):  print "a is available in the given list"  else:  print "a is not available in the given list"  output:-  a is not available in the given list  b is not available in the given list  a is available in the given list  **Identity Operators**   * Identity operators compare the memory locations of two objects.   Example  #!/usr/bin/python  a = 20  b = 20  if ( a is b ):  print " a and b have same identity"  else:  print " a and b do not have same identity"  if ( id(a) == id(b) ):  print "a and b have same identity"  else:  print " a and b do not have same identity"  b = 30  if ( a is b ):  print " a and b have same identity"  else:  print " a and b do not have same identity"  if ( a is not b ):  print " a and b do not have same identity"  else:  print " a and b have same identity"  output:  a and b have same identity  a and b have same identity  a and b do not have same identity  a and b do not have same identity    **Operators Precedence**   |  |  | | --- | --- | | Operator | Description | | \*\* | Exponentiation (raise to the power) | | ~ + - | Ccomplement, unary plus and minus (method names for the  last two are +@ and -@) | | \* / % // | Multiply, divide, modulo and floor division | | + - | Addition and subtraction | | >> << | Right and left bitwise shift | | & | Bitwise 'AND' | | ^ | | Bitwise exclusive `OR' and regular `OR' | | <= < > >= | Comparison operators | | <> == != | Equality operators | | = %= /= //= -= += \*= \*\*= | Assignment operators | | is is not | Identity operators | | in not in | Membership operators | | not or and | Logical operators | |  |  |   Example:  #!/usr/bin/python  val1 = 20  val2 = 10  c = 15  d = 5  e = 0  e = (a + b) \* c / d #( 30 \* 15 ) / 5 = 90  print "Value of (a + b) \* c / d is ", e  e = ((a + b) \* c) / d # (30 \* 15 ) / 5 = 90  print "Value of ((a + b) \* c) / d is ", e  e = (a + b) \* (c / d); # (30) \* (15/5) = 90  print "Value of (a + b) \* (c / d) is ", e  e = a + (b \* c) / d; #20 + (150/5) = 90  print "Value of a + (b \* c) / d is ", e |

**if..... elif.... else.. statements**

**24 Jan 2017 - Sai Krishna Juturi**

The syntax of the *if...else* statement is

if expression:

statement(s)

else:

statement(s)

### Flow Diagram

![](data:None;base64,)

### Example

#!/usr/bin/python

var1 = 100

if var1:

print "1 - Got a true expression value"

print var1

else:

print "1 - Got a false expression value"

print var1

var2 = 0

if var2:

print "2 - Got a true expression value"

print var2

else:

print "2 - Got a false expression value"

print var2

print "Good bye!"

**Output**

1 - Got a true expression value

100

2 - Got a false expression value

0

Good bye!

## The *elif* Statement

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

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

### syntax

if expression1:

statement(s)

elif expression2:

statement(s)

elif expression3:

statement(s)

else:

statement(s)

Core Python does not provide switch or case statements as in other languages, but we can use if..elif...statements to simulate switch case as follows −

### **Example**

#!/usr/bin/python

var = 100

if var == 200:

print "1 - Got a true expression value"

print var

elif var == 150:

print "2 - Got a true expression value"

print var

elif var == 100:

print "3 - Got a true expression value"

print var

else:

print "4 - Got a false expression value"

print var

print "Good bye!"

**When the above code is executed, it produces the following result −**

3 - Got a true expression value

100

Good bye!

**Loops**

**30/1/2017 -J .Sai Krishna**

In general, statements are executed sequentially. There may be a situation when you need to execute a block of code several number of times.

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

A loop statement allows us to execute a statement or group of statements multiple times.

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

## Loop Control Statements

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

Python supports the following control statements.

|  |  |
| --- | --- |
| **Control Statement** | **Description** |
| break | Terminates the loop statement and transfers execution to the statement immediately following the loop. |
| continue | Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating. |
| pass | The pass statement in Python is used when a statement is required syntactically but you do not want any command or code to execute. |

**while loop**

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

### Syntax

while expression:

statement(s)

Here, **statement(s)** may be a single statement or a block of statements. The **condition** may be any expression, and true is any non-zero value. The loop iterates while the condition is true.

When the condition becomes false, program control passes to the line immediately following the loop.

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

![](data:None;base64,)

### Example for while

#!/usr/bin/python

count = 0

while (count < 9):

print 'The count is:', count

count = count + 1

print "Good bye!"

**Output**

The count is: 0

The count is: 1

The count is: 2

The count is: 3

The count is: 4

The count is: 5

The count is: 6

The count is: 7

The count is: 8

Good bye!

**for loop**

for loop is used to **iterate over a iterable data**. The iterable data may be list, dict, tuple, set......

Syntax

for element in <iterable>:

.

.

. block of statements seperated by same space

else:

.

these statements executes after end of for loop

**for loop flow chart**

![](data:None;base64,)

### Example

#!/usr/bin/python

for letter in **'Python':** # First Example

print 'Current Letter :', letter

fruits = ['banana', 'apple', 'mango']

for fruit in **fruits**: # Second Example

print 'Current fruit :', fruit

else:

print "End of for loop\n”

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

Current Letter : P

Current Letter : y

Current Letter : t

Current Letter : h

Current Letter : o

Current Letter : n

Current fruit : banana

Current fruit : apple

Current fruit : mango

Good bye!

Note : for loop with dictionary will iterate over keys

for loop with string will iterate over characters

**Nested loops:**

Nested loops are loops with loops. When you repeat a loop multiple times with multiple conditions go with nested loops

**Syntax:**

for ele1 in iterable1:

.

.

for ele2 in interable2:

.

.

else:

'''statements after loop\n'''

while condition1:

.

.

while condition2:

.

.

else:

'''statements after while loop'''

**25 Jan 2016**  **FUNCTIONS Sai Krishna J**

Functions are used forreusable code. When a large need to be used for multiple times we can use functions. Functions also increases code modularity.

Cons in functions:

-> Memory : Functions consume extra memory for loading arguments in stack

-> Modification : If you modify a function it effects where ever you have used functions

-> Debugging : Debugging is hard when you use functions

**Syntax for function definition in python** :

def fun(arg1, arg2, arg3....):

statements

.

.

.

return

function definittion starts with def in python. And function code must have

indentation.

**Types of functions:**

1) User defined

2) Built-in

**Types of functions:**

1) No input, No output

2) No input, with output

3) With Input , No output

4)With input, with output

**Call by reference:**

When we pass an argument to function if any change on the argument reflects in actual code it is called call by reference.

eg: passing any mutable data as an argument is call by reference.

**Call by Value:**

When we pass an argument to function if any change on the argument does not reflect in actual code it is called call by value

eg: passing a any immutable data as an argument is call by value.

**Types of Function arguments in Python:**

1) Required arguments

2) Keyword arguments

3) Default arguments

4) Variable arguments

**Required arguments:**

Required arguments are required for sure while calling a function.

def fun\_reqargs(arg1,arg2):

print arg1

print arg2

while function calling we nedd to pass the no.of arguments in the function. If we pass more or less we will encounter an error

**Default arguments:**

def say(msg, tms = 1):

print msg,

print tms

return

say(20) -> output is 20 1

by **default** the argument tms is taken as 1, if you dont pass anything.

**Keyword arguments:**

In python while calling a function there is no need to pass arguments in order. Instead we can specify argument name.

def fun(arg1, arg2, arg3):

print 'arg1',arg1

print 'arg2',arg2

print 'arg3',arg3 # function definition

fun ( arg3 = 10 , arg2 = 20, arg1 = 30) #function calling

**Variable arguments:**

In python we can pass variable arguments and those will be collected as tuple.

def fun2(arg1, \*args ):

print arg1

print args **#function definition**

fun2(10,20,30,40) **#function calling**

**output:**

10

(20, 30, 40)

def fun3 ( arg1 , \*\*kwrags ):

print arg1

print kwargs **#function definition**

fun3(3,key1=1,key2:2) **#function calling**

**output:**

3

{'key1':1,'key2':2}

**Note:**

Kwrags argument in above function is also called as keyword argumet

Try different combinations of keyword arguments and variable length arguments and normal arguments in function definition and see the result.

**Function**

def total(initial = 5,\*number,\*\*keywords):

print 'initial',initial

print 'number',number

print 'keywords',keywords

count =initial

for number in numbers:

count += number

for key in keywords:

count += keywords[key]

return count

print total(10,1,2,3,veg=50,fruits=100)

check the output??

**SCOPE OF VARIABLES**

varg = 20

vars = 90

print varg

def fun( ):

# if you want to access a variable which is outside the function you # need to specify it as global

global varg

varg = 50

# print vars # gives error

fun( ) #function calling

print 'varg after function call',varg

**output**:

20

varg after function call 50

**Iterators:**

->> Iterators gives the values one after other by iterating over data which is iterable.

->> The object of data type which is having method \_\_iter\_\_ is called iterable object.

Str = 'abc'

itr = iter(str)

>>> itr.next()

'a'

>>>itr.next()

'b'

>>>itr.next()

'c'

>>>itr.next()

error.......

StopIteration

**Generators**:

-> Generator generates the data whenever the data is required instead of creating whole chunk of memory required at a time.

**-> yield** is a keyword for generator

def reverse(data):

for index in range(len(data)-1,-1,-1):

yield data[index]

print 'data yielded'

var = reverse('hello')

print type(var)

for ele in var:

print ele

**Execute the above program and check the output**

**Note**: -> range function returns a list on which we can iterate

-> xrange function generates a generator

**Data types in Python in Detail:**

30-JAN-2016  **NUMBERS** Pradeep

Number data types store numeric values. They are immutable data types, means that changing the value of a number data type results in a newly allocated object.

Ex: val1 = 10; val2 = 30

**del:** Is used to delete the reference to a number .

**del** var1 # It will delete the reference

**del** var2,var3,..... #For multiple variables

**Python Supports four different numericals:**

**int :** They are positive or negative whole numbers with no decimal point.

**long :** They are integers of unlimited size, written like integers and followed by an uppercase or lowercase L.But it is recemended to use 'L' to avoid confusion with the number 1.

**float :** They represent real numbers and are written with a decimal point dividing.Floats may also be in scientific notation, with E or e indicating the power of 10 (2.5e2 = 2.5 x 102 = 250).

**complex :** a + bJ,The real part of the number is a, and the imaginary part is b.J (or j) represents the square root of -1.

**Number type Conversion:**

**int(x)** to convert x to a plain integer.

**long(x)** to convert x to a long integer.

**float(x)** to convert x to a floating-point number.

**complex(x)** to convert x to a complex number with real part x and imaginary part zero.

**complex(x, y)** to convert x and y to a complex number with real part x and imaginary part y.

**x** and **y** are numeric expressions.

**Mathematical Functions:**

**abs(x):** The method **abs()** returns absolute value of x - the (positive) distance between x and zero.

**x** **--**is numeric Expressions

**Sample.py** #!/usr/bin/python

print "abs(-45) : ", abs(-45)

print "abs(100.12) : ", abs(100.12)

print "abs(119L) : ", abs(119L)

**O/P:** abs(-45) : 45

abs(100.12) : 100.12

abs(119L) : 119

**ceil(x):**The method **ceil()** returns ceiling value of x - the smallest integer not less than x.

This function is not accessible directly, so we need to import math module.

**x** -- is numeric Expressions

**Sample.py** #!/usr/bin/python

import math # This will import math module

print "math.ceil(-45.17) : ", math.ceil(-45.17)

print "math.ceil(100.12) : ", math.ceil(100.12)

print "math.ceil(100.72) : ", math.ceil(100.72)

print "math.ceil(119L) : ", math.ceil(119L)

print "math.ceil(math.pi) : ", math.ceil(math.pi)

**O/P:** math.ceil(-45.17) : -45.0

math.ceil(100.12) : 101.0

math.ceil(100.72) : 101.0

math.ceil(119L) : 119.0

math.ceil(math.pi) : 4.0

**cmp(x,y):**The method **cmp()** returns the sign of the difference of two numbers : -1 if x < y, 0 if x == y, or 1 if x > y .

**x,y --** are numeric Expressions

**Sample.py** #!/usr/bin/python

print "cmp(80, 100) : ", cmp(80, 100)

print "cmp(180, 100) : ", cmp(180, 100)

print "cmp(-80, 100) : ", cmp(-80, 100)

print "cmp(80, -100) : ", cmp(80, -100)

**O/P:**  cmp(80, 100) : -1

cmp(180, 100) : 1

cmp(-80, 100) : -1

cmp(80, -100) : 1

**exp(x):**The method **exp()** returns returns exponential of x: ex.

This function is not accessible directly, so we need to import math module.

**x** -- This is a numeric expression.

**Sample.py** #!/usr/bin/python

import math # This will import math module

print "math.exp(-45.17) : ", math.exp(-45.17)

print "math.exp(100.12) : ", math.exp(100.12)

print "math.exp(100.72) : ", math.exp(100.72)

print "math.exp(119L) : ", math.exp(119L)

print "math.exp(math.pi) : ", math.exp(math.pi)

**O/P:** math.exp(-45.17) : 2.41500621326e-20

math.exp(100.12) : 3.03084361407e+43

math.exp(100.72) : 5.52255713025e+43

math.exp(119L) : 4.7978133273e+51

math.exp(math.pi) : 23.1406926328

**fabs(x):**The method **fabs()** returns the absolute value of x.

This function is not accessible directly, so we need to import math module.

**x** -- This is a numeric value.

**Sample.py** #!/usr/bin/python

import math # This will import math module

print "math.fabs(-45.17) : ", math.fabs(-45.17)

print "math.fabs(100.12) : ", math.fabs(100.12)

print "math.fabs(100.72) : ", math.fabs(100.72)

print "math.fabs(119L) : ", math.fabs(119L)

print "math.fabs(math.pi) : ", math.fabs(math.pi)

**O/P:** math.fabs(-45.17) : 45.17

math.fabs(100.12) : 100.12

math.fabs(100.72) : 100.72

math.fabs(119L) : 119.0

math.fabs(math.pi) : 3.14159265359

**floor(x):**The method **floor()** returns floor of x - the largest integer not greater than x.

This function is not accessible directly, so we need to import math module.

**x** -- This is a numeric expression.

**Sample.py** #!/usr/bin/python

import math # This will import math module

print "math.floor(-45.17) : ", math.floor(-45.17)

print "math.floor(100.12) : ", math.floor(100.12)

print "math.floor(100.72) : ", math.floor(100.72)

print "math.floor(119L) : ", math.floor(119L)

print "math.floor(math.pi) : ", math.floor(math.pi)

**O/P:** math.floor(-45.17) : -46.0

math.floor(100.12) : 100.0

math.floor(100.72) : 100.0

math.floor(119L) : 119.0

math.floor(math.pi) : 3.0

**log(x):**The method **log()** returns natural logarithm of x, for x > 0.

This function is not accessible directly, so we need to import math module.

**x** -- This is a numeric expression.

**Sample.py** #!/usr/bin/python

import math # This will import math module

print "math.log(100.12) : ", math.log(100.12)

print "math.log(100.72) : ", math.log(100.72)

print "math.log(119L) : ", math.log(119L)

print "math.log(math.pi) : ", math.log(math.pi)

**O/P:** math.log(100.12) : 4.60636946656

math.log(100.72) : 4.61234438974

math.log(119L) : 4.77912349311

math.log(math.pi) : 1.14472988585

**log10(x):**The method **log10()** returns base-10 logarithm of x for x > 0.

This function is not accessible directly, so we need to import math module.

**x** -- This is a numeric expression.

**Sample.py** #!/usr/bin/python

import math # This will import math module

print "math.log10(100.12) : ", math.log10(100.12)

print "math.log10(100.72) : ", math.log10(100.72)

print "math.log10(119L) : ", math.log10(119L)

print "math.log10(math.pi) : ", math.log10(math.pi)

**O/P:** math.log10(100.12) : 2.00052084094

math.log10(100.72) : 2.0031157171

math.log10(119L) : 2.07554696139

math.log10(math.pi) : 0.497149872694

**max(x,y,z....):**The method **max()** returns the largest of its arguments: the value closest to positive infinity.

**x** -- This is a numeric expression.

**y** -- This is also a numeric expression.

**z** -- This is also a numeric expression.

**Sample.py #**!/usr/bin/python

print "max(80, 100, 1000) : ", max(80, 100, 1000)

print "max(-20, 100, 400) : ", max(-20, 100, 400)

print "max(-80, -20, -10) : ", max(-80, -20, -10)

print "max(0, 100, -400) : ", max(0, 100, -400)

**O/P:** max(80, 100, 1000) : 1000

max(-20, 100, 400) : 400

max(-80, -20, -10) : -10

max(0, 100, -400) : 100

**min(x,y,z...):**The method **min(**) returns the smallest of its arguments: the value closest to negative infinity.

**x** -- This is a numeric expression.

**y** -- This is also a numeric expression.

**z** -- This is also a numeric expression.

**Sample.py** #!/usr/bin/python

print "min(80, 100, 1000) : ", min(80, 100, 1000)

print "min(-20, 100, 400) : ", min(-20, 100, 400)

print "min(-80, -20, -10) : ", min(-80, -20, -10)

print "min(0, 100, -400) : ", min(0, 100, -400)

**O/P:** min(80, 100, 1000) : 80

min(-20, 100, 400) : -20

min(-80, -20, -10) : -80

min(0, 100, -400) : -400

**modf(x):**The method **modf()** returns the fractional and integer parts of x in a two-item tuple. Both parts have the same sign as x. The integer part is returned as a float.

This function is not accessible directly, so we need to import math module.

**x** -- This is a numeric expression.

**Sample.py** #!/usr/bin/python

import math # This will import math module

print "math.modf(100.12) : ", math.modf(100.12)

print "math.modf(100.72) : ", math.modf(100.72)

print "math.modf(119L) : ", math.modf(119L)

print "math.modf(math.pi) : ", math.modf(math.pi)

**O/P:** math.modf(100.12) : (0.12000000000000455, 100.0)

math.modf(100.72) : (0.71999999999999886, 100.0)

math.modf(119L) : (0.0, 119.0)

math.modf(math.pi) : (0.14159265358979312, 3.0)

**pow():**The method **pow()** returns value of x\*\*y.

This function is not accessible directly, so we need to import math module.

**Sample.py** #!/usr/bin/python

import math # This will import math module

print "math.pow(100, 2) : ", math.pow(100, 2)

print "math.pow(100, -2) : ", math.pow(100, -2)

print "math.pow(2, 4) : ", math.pow(2, 4)

print "math.pow(3, 0) : ", math.pow(3, 0)

**O/P:** math.pow(100, 2) : 10000.0

math.pow(100, -2) : 0.0001

math.pow(2, 4) : 16.0

math.pow(3, 0) : 1.0

**round(x,[n]):**The method **round()** returns x rounded to n digits from the decimal point.

**x** -- This is a numeric expression..

**n** -- This is also a numeric expression.

**Sample.py** #!/usr/bin/python

print "round(80.23456, 2) : ", round(80.23456, 2)

print "round(100.000056, 3) : ", round(100.000056, 3)

print "round(-100.000056, 3) : ", round(-100.000056, 3)

**O/P:** round(80.23456, 2) : 80.23

round(100.000056, 3) : 100.0

round(-100.000056, 3) : -100.0

**sqrt(x):**The method **sqrt()** returns the square root of x for x > 0.

**x** -- This is a numeric expression.

**Sample.py** #!/usr/bin/python

import math # This will import math module

print "math.sqrt(100) : ", math.sqrt(100)

print "math.sqrt(7) : ", math.sqrt(7)

print "math.sqrt(math.pi) : ", math.sqrt(math.pi)

**O/P:** math.sqrt(100) : 10.0

math.sqrt(7) : 2.64575131106

math.sqrt(math.pi) : 1.77245385091

**STRINGS**

Strings are amongst the most popular types in Python. We can create them simply by enclosing characters in quotes. Python treats single quotes the same as double quotes. Creating strings is as simple as assigning a value to a variable.Python does not support a character type; these are treated as strings of length one, thus also considered a substring.

var1 = 'Hello World!'

var2 = "Python Programming"

To access substrings, use the square brackets for slicing along with the index or

indices to obtain your substring.

**Sample.py** #!/usr/bin/python

var1 = 'Hello World!'

var2 = "Python Programming"

print "var1[0]: ", var1[0]

print "var2[1:5]: ", var2[1:5]

**O/P:** var1[0]: H

var2[1:5]: ytho

**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 altogether.

**Sample.py** #!/usr/bin/python

var1 = 'Hello World!'

print "Updated String :- ", var1[:6] + 'Python'

**O/P:** Updated String :- Hello Python

**Escape Characters:**

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

**Backslash notation Description**

\a Bell or alert

\b Backspace

\cx Control-x

\C-x Control-x

\e Escape

\f Formfeed

\M-\C-x Meta-Control-x

\n Newline

\nnn Octal notation, where n is in the range 0.7

\r Carriage return

\s Space

\t Tab

\v Vertical tab

\x Character x

\xnn Hexadecimal notation, where n is in the range 0.9, a.f, or A.F

**String Special Operators**

**Operator Description Example**

+ Concatenation - Adds values on either side of the operator a + b HelloPython

\* Repetition Creates new strings, concatenating multiple a\*2 will give-HelloHello copies of the same string

[] Slice - Gives the character from the given index a[1] will give e

[ : ] 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 H in a will give 1

given string

not in Membership - Returns true if a character does not M not in a will give 1

exist in the given string

r/R Raw String - Suppresses actual meaning of Escape print r'\n' prints \n and print R'\n'prints \n characters.

% Format - Performs String formatting See at next section

**String Formatting Operator:**

One of Python's coolest features is the string format operator %.

**Sample.py** #!/usr/bin/python

print "My name is %s and weight is %d kg!" % ('Zara', 21)

**O/P:** My name is Zara and weight is 21 kg!

Here is the list of complete set of symbols which can be used along with %:

**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 (UPPERcase letters)

%e exponential notation (with lowercase 'e')

%E exponential notation (with UPPERcase 'E')

%f floating point real number

%g the shorter of %f and %e

%G the shorter of %f and %E

Other supported symbols and functionality are listed in the following table:

\* 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)

m.n. m is the minimum total width and n is the number of digits to display after the decimal point (if appl.)

**Triple Quotes**

Python's triple quotes comes to the rescue by allowing strings to span multiple lines, including verbatim NEWLINEs, TABs, and any other special characters.

The syntax for triple quotes consists of three consecutive single or double quotes.

**Sample.py** #!/usr/bin/python

para\_str = """this is a long string that is made up of

several lines and non-printable characters such as

TAB ( \t ) and they will show up that way when displayed.

NEWLINEs within the string, whether explicitly given like

this within the brackets [ \n ], or just a NEWLINE within

the variable assignment will also show up. """

print para\_str;

**O/P:** this is a long string that is made up of

several lines and non-printable characters such as

TAB ( \t ) and they will show up that way when displayed.

NEWLINEs within the string, whether explicitly given like

this within the brackets [ \n ], or just a NEWLINE within

the variable assignment will also show up.

**Unicode String :**

Normal strings in Python are stored internally as 8-bit ASCII, while Unicode strings are stored as 16-bit Unicode.

**Sample.py** #!/usr/bin/python

print u'Hello, world!'

**O/P:** Hello, world!

**Built-in String Methods :**

Python includes the following built-in methods to manipulate strings:

**capitalize()** It returns a copy of the string with only its first character capitalized.

**Syntax** str.capitalize()

**Parameters**  NA

**Return Value** string

**Example**  #!/usr/bin/python

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

print "str.capitalize() : ", str.capitalize()

**Result** str.capitalize() : This is string example....wow!!!

**center(width, fillchar)** The method center() returns centered in a string of length width. Padding is done using the specified fillchar. Default filler is a space.

**Syntax**  str.center(width[, fillchar])

**Parameters** width -- This is the total width of the string.

fillchar -- This is the filler character.

**Return Value** This method returns centered in a string of length width.

**Example**  #!/usr/bin/python

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

print "str.center(40, 'a') : ", str.center(40, 'a')

**Result** str.center(40, 'a') : aaaathis is string example....wow!!!aaaa

**count(str, beg= 0,end=len(string))** The method count() returns the number of occurrences of substring sub in the range [start, end]. Optional arguments start and end are interpreted as in slice notation.

**Syntax** str.count(sub, start= 0,end=len(string))

**Parameters** sub -- This is the substring to be searched.

start -- Search starts from this index. First character starts from 0 index. By default search starts from 0 index.

end -- Search ends from this index. First character starts from 0 index. By default search ends at the last index.

**Return Value** Centered in a string of length width.

**Example**  #!/usr/bin/python

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

sub = "i";

print "str.count(sub, 4, 40) : ", str.count(sub, 4, 40)

sub = "wow";

print "str.count(sub) : ", str.count(sub)

**Result**  str.count(sub, 4, 40) : 2

str.count(sub, 4, 40) : 1

**decode(encoding='UTF-8',errors='strict')** The method decode() decodes the string using the codec registered for encoding. It defaults to the default string encoding.

**Syntax** str.decode(encoding='UTF-8',errors='strict')

**Parameters** encoding -- This is the encodings to be used. For a list of all encoding schemes

errors -- This may be given to set a different error handling scheme. The default for errors is 'strict', meaning that encoding errors raise a UnicodeError. Other possible values are 'ignore', 'replace', 'xmlcharrefreplace', 'backslashreplace' and any other name registered via codecs.register\_error().

**Return Value** Decoded string.

**Example** #!/usr/bin/python

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

str = str.encode('base64','strict');

print "Encoded String: " + str;

print "Decoded String: " + str.decode('base64','strict')

**Result** Encoded String: dGhpcyBpcyBzdHJpbmcgZXhhbXBsZS4uLi53b3chISE=

Decoded String: this is string example....wow!!!

**encode(encoding='UTF-8',errors='strict')** The method encode() returns an encoded version of the string. Default encoding is the current default string encoding. The errors may be given to set a different error handling scheme.

**Syntax** str.encode(encoding='UTF-8',errors='strict')

**Parameters**  encoding -- This is the encodings to be used. For a list of all encoding schemes

erors -- This may be given to set a different error handling scheme. The default for errors is 'strict', meaning that encoding errors raise a UnicodeError. Other possible values are 'ignore', 'replace', 'xmlcharrefreplace', 'backslashreplace' and any other name registered via codecs.register\_error().

**Return Value** Encoded string.

**Example** #!/usr/bin/python

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

print "Encoded String: " + str.encode('base64','strict')

**Result** Encoded String: dGhpcyBpcyBzdHJpbmcgZXhhbXBsZS4uLi53b3chISE=

**endswith(suffix, beg=0, end=len(string))** It returns True if the string ends with the specified suffix, otherwise return False optionally restricting the matching with the given indices start and end.

**Syntax** str.endswith(suffix[, start[, end]])

**Parameters** suffix -- This could be a string or could also be a tuple of suffixes to look for.

start -- The slice begins from here.

end -- The slice ends here.

**Return Value** TRUE if the string ends with the specified suffix, otherwise FALSE.

**Example** #!/usr/bin/python

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);

**Result** True

True

True

False

**expandtabs(tabsize=8)** It returns a copy of the string in which tab characters ie. '\t' are expanded using spaces, optionally using the given tabsize (default 8).

**Syntax** str.expandtabs(tabsize=8)

**Parameters** tabsize -- This specifies the number of characters to be replaced for a tab character '\t'.

**Return Value**  This method returns a copy of the string in which tab characters i.e., '\t' have been expanded using spaces.

**Example** #!/usr/bin/python

str = "this is\tstring example....wow!!!";

print "Original string: " + str;

print "Defualt exapanded tab: " + str.expandtabs();

print "Double exapanded tab: " + str.expandtabs(16);

**Result**

Original string: this is

string example....wow!!!

Defualt exapanded tab: this is string example....wow!!!

Double exapanded tab: this is

string example....wow!!!

**find(str, beg=0 end=len(string))** It determines if string str occurs in string, or in a substring of string if starting index beg and ending index end are given.

**Syntax** str.find(str, beg=0 end=len(string))

**Parameters** str -- This specifies the string to be searched.

beg -- This is the starting index, by default its 0.

end -- This is the ending index, by default its equal to the lenght of the string.

**Return Value** Index if found and -1 otherwise.

**Example** #!/usr/bin/python

str1 = "this is string example....wow!!!";

str2 = "exam";

print str1.find(str2);

print str1.find(str2, 10);

print str1.find(str2, 40);

**Result** 15

15

-1

**index(str, beg=0, end=len(string))** It determines if string str occurs in string or in a substring of string if starting index beg and ending index end are given. This method is same as find(), but raises

an exception if sub is not found.

**Syntax** str.index(str, beg=0 end=len(string))

**Parameters** str -- This specifies the string to be searched.

beg -- This is the starting index, by default its 0.

end -- This is the ending index, by default its equal to the length of the string.

**Return Value** Index if found otherwise raises an exception if str is not found.

**Example**  #!/usr/bin/python

str1 = "this is string example....wow!!!";

str2 = "exam";

print str1.index(str2);

print str1.index(str2, 10);

print str1.index(str2, 40);

**Result** 15

15

Traceback (most recent call last):

File "test.py", line 8, in

print str1.index(str2, 40);

ValueError: substring not found shell returned 1

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

**Syntax** str.isa1num()

**Parameters** NA

**Return Value** TRUE if all characters in the string are alphanumeric and there is at least one character, FASLE otherwise.

**Example**  #!/usr/bin/python

str = "this2009"; # No space in this string

print str.isalnum();

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

print str.isalnum();

**Result**  True

False

**isalpha()** The method isalpha() checks whether the string consists of alphabetic characters only.

**Syntax** str.isalpha()

**Parameters** NA

**Return Value** This method returns true if all characters in the string are alphabetic and there is at

least one character, false otherwise.

**Example** #!/usr/bin/python

str = "this"; # No space & digit in this string

print str.isalpha();

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

print str.isalpha();

**Result** True

False

**isdigit()** The method isdigit() checks whether the string consists of digits only.

**Syntax** str.isdigit()

**Parameters** NA

**Return Value** This method returns true if all characters in the string are digits and there is at least

one character, false otherwise.

**Example** #!/usr/bin/python

str = "123456"; # Only digit in this string

int str.isdigit();

r = "this is string example....wow!!!";

int str.isdigit();

**Result** True

False

**islower()** The method islower() checks whether all the case-based characters (letters) of the string are lowercase.

**Syntax** str.islower()

**Parameters** NA

**Return Value** This method returns true if all cased characters in the string are lowercase and there is at least one cased character, false otherwise.

**Example** #!/usr/bin/python

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

print str.islower();

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

print str.islower();

**Result** False

True

**isnumeric()** The method isnumeric() checks whether the string consists of only numeric characters. This method is present only on unicode objects.

Note: To define a string as Unicode, one simply prefixes a 'u' to the opening quotation mark of the assignment.

**Syntax** str.isnumeric()

**Parameters** NA

**Return Value** This method returns true if all characters in the string are numeric, false otherwise.

**Example** #!/usr/bin/python

str = u"this2009";

print str.isnumeric();

str = u"23443434";

print str.isnumeric();

**Result** False

True

**isspace()** The method isspace() checks whether the string consists of whitespace.

**Syntax** str.isspace()

**Parameters** NA

**Return Value** This method returns true if there are only whitespace characters in the string and

there is at least one character, false otherwise.

**Example** #!/usr/bin/python

str = " ";

print str.isspace();

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

print str.isspace();

**Result** True

False

**istitle()** The method istitle() checks whether all the case-based characters in the string following non-casebased letters are uppercase and all other case-based characters are lowercase.

**Syntax** str.istitle()

**Parameters** NA

**Return Value** This method returns true if the string is a titlecased string and there is at least one

character, for example uppercase characters may only follow uncased characters and lowercase characters only cased ones.It returns false otherwise.

**Example** #!/usr/bin/python

str = "This Is String Example...Wow!!!";

print str.istitle();

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

print str.istitle();

**Result** True

False

**isupper()** The method isupper() checks whether all the case-based characters (letters) of the string are uppercase.

**Syntax** str.isupper()

**Parameters** NA

**Return Value** This method returns true if all cased characters in the string are uppercase and there is at least one cased character, false otherwise.

**Example** #!/usr/bin/python

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

print str.isupper();

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

print str.isupper();

**Result** True

False

**join(seq)** The method join() returns a string in which the string elements of sequence have been joined by str separator.

**Syntax** str.join(sequence)

**Parameters** sequence -- This is a sequence of the elements to be joined.

**Return Value** This method returns a string, which is the concatenation of the strings in the

sequence seq. The separator between elements is the string providing this method.

**Example** #!/usr/bin/python

str = "-";

seq = ("a", "b", "c"); # This is sequence of strings.

print str.join( seq );

**Result** a-b-c

**len(string)** The method len() returns the length of the string.

**Syntax** len( str )

**Parameters** NA

**Return Value** This method returns the length of the string.

**Example** #!/usr/bin/python

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

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

**Result** Length of the string: 32

**ljust(width[, fillchar])** The method ljust() returns the string left justified in a string of length width. Padding is done using the specified fillchar (default is a space). The original string is returned if width is less than len(s).

**Syntax** str.ljust(width[, fillchar])

**Parameters** width -- This is string length in total after padding.

fillchar -- This is filler character, default is a space.

**Return Value** This method returns the string left justified in a string of length width. Padding is done using the specified fillchar (default is a space). The original string is returned if width is less than len(s).

**Example** #!/usr/bin/python

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

print str.ljust(50, '0');

**Result** this is string example....wow!!!000000000000000000

**lower()** The method lower() returns a copy of the string in which all case-based characters have been lowercased.

**Syntax** str.lower()

**Parameters** NA

**Return Value** This method returns a copy of the string in which all case-based characters have been

lowercased.

**Example** #!/usr/bin/python

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

print str.lower();

**Result**  this is string example....wow!!!

**lstrip()** The method lstrip() returns a copy of the string in which all chars have been stripped from the beginning of the string (default whitespace characters).

**Syntax** str.lstrip([chars])

**Parameters** chars -- You can supply what chars have to be trimmed.

**Return Value** This method returns a copy of the string in which all chars have been stripped from the beginning of the string (default whitespace characters).

**Example** #!/usr/bin/python

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

print str.lstrip();

str = "88888888this is string example....wow!!!8888888";

print str.lstrip('8');

**Result** this is string example....wow!!!

this is string example....wow!!!8888888

**maketrans()** The method maketrans() returns a translation table that maps each character in

the intabstring into the character at the same position in the outtab string. Then this table is passed to the translate() function.

Note: Both intab and outtab must have the same length.

**Syntax** str.maketrans(intab, outtab]);

**Parameters** intab -- This is the string having actual characters.

outtab -- This is the string having corresponding mapping character.

**Return Value** This method returns a translate table to be used translate() function.

**Example** The following example shows the usage of maketrans() method. Under this, every

vowel in a string is replaced by its vowel position:

#!/usr/bin/python

from string import maketrans

# Required to call maketrans function.

intab = "aeiou"

outtab = "12345"

trantab = maketrans(intab, outtab)

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

print str.translate(trantab);

**Result** th3s 3s str3ng 2x1mpl2....w4w!!!

**max(str)** The method max() returns the max alphabetical character from the string str.

**Syntax** max(str)

**Parameters** str -- This is the string from which max alphabetical character needs to be returned.

**Return Value** This method returns the max alphabetical character from the string str.

**Example** #!/usr/bin/python

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);

**Result** Max character: y

Max character: x

**min(str)** The method min() returns the min alphabetical character from the string str.

**Syntax** min(str)

**Parameters** str -- This is the string from which min alphabetical character needs to be returned.

**Return Value** This method returns the max alphabetical character from the string str.

**Example**  #!/usr/bin/python

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

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

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

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

Result Min character: !

Min character: !

**replace(old, new [, max])** The method replace() returns a copy of the string in which the occurrences of old have been replaced with new, optionally restricting the number of replacements

to max.

**Syntax** str.replace(old, new[, max])

**Parameters** old -- This is old substring to be replaced.

new -- This is new substring, which would replace old substring.

max -- If this optional argument max is given, only the first count occurrences are replaced.

**Return Value** This method returns a copy of the string with all occurrences of substring old replaced by new. If the optional argument max is given, only the first count occurrences are replaced.

**Example** #!/usr/bin/python

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

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

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

**Result** thwas was string example....wow!!! thwas was really string

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

**rfind(str, beg=0,end=len(string))** The method rfind() returns the last index where the substring str is found, or -1 if no such index exists, optionally restricting the search to string[beg:end].

**Syntax** str.rfind(str, beg=0 end=len(string))

**Parameters** str -- This specifies the string to be searched.

beg -- This is the starting index, by default its 0.

end -- This is the ending index, by default its equal to the length of the string.

**Return Value** This method returns last index if found and -1 otherwise.

**Example** #!/usr/bin/python

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

str = "is";

print str.rfind(str);

print str.rfind(str, 0, 10);

print str.rfind(str, 10, 0);

print str.find(str);

print str.find(str, 0, 10);

print str.find(str, 10, 0);

**Result** 5

5

-1

2

2

-1

**rindex(str, beg=0, end=len(string))** The method rindex() returns the last index where the substring str is found, or raises an exception if no such index exists, optionally restricting the search to string[beg:end].

**Syntax** str.rindex(str, beg=0 end=len(string))

**Parameters** str -- This specifies the string to be searched.

beg -- This is the starting index, by default its 0

len -- This is ending index, by default its equal to the length of the string.

**Return Value** This method returns last index if found otherwise raises an exception if str is not found.

**Example** #!/usr/bin/python

str1 = "this is string example....wow!!!";

str2 = "is";

print str1.rindex(str2);

print str1.index(str2);

**Result** 5

2

**rjust(width,[, fillchar])** The method rjust() returns the string right justified in a string of length width. Padding is done using the specified fillchar (default is a space). The original string is returned if width is less than len(s).

**Syntax** str.rjust(width[, fillchar])

**Parameters** width -- This is the string length in total after padding.

fillchar -- This is the filler character, default is a space.

**Return Value** This method returns the string right justified in a string of length width. Padding is

done using the specified fillchar (default is a space). The original string is returned if width is less than len(s).

**Example** #!/usr/bin/python

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

print str.rjust(50, '0');

**Result**  000000000000000000this is string example....wow!!!

**rstrip()** The method rstrip() returns a copy of the string in which all chars have been stripped from the end of the string (default whitespace characters).

**Syntax** str.rstrip([chars])

**Parameters** chars -- You can supply what chars have to be trimmed.

**Return Value** This method returns a copy of the string in which all chars have been stripped from

the end of the string (default whitespace characters).

**Example** #!/usr/bin/python

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

print str.rstrip();

str = "88888888this is string example....wow!!!8888888";

print str.rstrip('8');

**Result** this is string example....wow!!!

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

**split(str="", num=string.count(str))** The method split() returns 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.

**Syntax** str.split(str="", num=string.count(str)).

**Parameters** str -- This is any delimeter, by default it is space.

num -- this is number of lines to be made.

**Return Value** This method returns a list of lines.

**Example** #!/usr/bin/python

str = "Line1-abcdef \nLine2-abc \nLine4-abcd";

print str.split( );

print str.split(' ', 1 );

**Result** ['Line1-abcdef', 'Line2-abc', 'Line4-abcd']

['Line1-abcdef', '\nLine2-abc \nLine4-abcd']

**splitlines(num=string.count('\n')) 103Python** The method splitlines() returns a list with all the lines in string, optionally including the line breaks (if num is supplied and is true)

**Syntax** str.splitlines( num=string.count('\n'))

**Parameters** num -- This is any number, if present then it would be assumed that line breaks need to be included in the lines.

**Return Value** This method returns true if found matching string otherwise false.

**Example #**!/usr/bin/python

str = "Line1-a b c d e f\nLine2- a b c\n\nLine4- a b c d";

print str.splitlines( );

print str.splitlines( 0 );

print str.splitlines( 3 );

print str.splitlines( 4 );

print str.splitlines( 5 );

**Result** ['Line1-a b c d e f', 'Line2- a b c', '', 'Line4- a b c d']

['Line1-a b c d e f', 'Line2- a b c', '', 'Line4- a b c d']

['Line1-a b c d e f\n', 'Line2- a b c\n', '\n', 'Line4- a b c d']

['Line1-a b c d e f\n', 'Line2- a b c\n', '\n', 'Line4- a b c d']

['Line1-a b c d e f\n', 'Line2- a b c\n', '\n', 'Line4- a b c d']

**startswith(str, beg=0,end=len(string))** The method startswith() checks whether string starts with str, optionally restricting the matching with the given indices start and end.

**Syntax** str.startswith(str, beg=0,end=len(string));

**Parameters** str -- This is the string to be checked.

beg -- This is the optional parameter to set start index of the matching boundary.

end -- This is the optional parameter to set start index of the matching boundary.

**Return Value** This method returns true if found matching string otherwise false.

**Example** #!/usr/bin/python

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

print str.startswith( 'this' );

print str.startswith( 'is', 2, 4 );

print str.startswith( 'this', 2, 4 );

**Result** True

True

False

**strip([chars])** The method strip() returns a copy of the string in which all chars have been stripped

from the beginning and the end of the string (default whitespace characters).

**Syntax** str.strip([chars]);

**Parameters** chars -- The characters to be removed from beginning or end of the string.

**Return Value** This method returns a copy of the string in which all chars have been stripped from

the beginning and the end of the string.

**Example** #!/usr/bin/python

str = "0000000this is string example....wow!!!0000000";

print str.strip( '0' );

**Result** this is string example....wow!!!

**swapcase()** The method swapcase() returns a copy of the string in which all the case-based characters have had their case swapped.

**Syntax** str.swapcase();

**Parameters** NA

**Return Value** This method returns a copy of the string in which all the case-based characters have

had their case swapped.

**Example** #!/usr/bin/python

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

print str.swapcase();

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

print str.swapcase();

**Result** THIS IS STRING EXAMPLE....WOW!!!

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

**title()** The method title() returns a copy of the string in which first characters of all the

words are capitalized.

**Syntax** str.title();

**Parameters** NA

**Return Value** This method returns a copy of the string in which first characters of all the words are

capitalized.

**Example** #!/usr/bin/python

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

print str.title();

**Result** This Is String Example....Wow!!!

**translate(table, deletechars="")** The method translate() returns a copy of the string in which all characters have been translated using table (constructed with the maketrans() function in the string

module), optionally deleting all characters found in the string deletechars.

**Syntax** str.translate(table[, deletechars]);

**Parameters** table -- You can use the maketrans() helper function in the string module to create a translation table.

deletechars -- The list of characters to be removed from the source string.

**Return Value** This method returns a translated copy of the string.

**Example** The following example shows the usage of translate() method. Under this every vowel

in a string is replaced by its vowel position:

#!/usr/bin/python

from string import maketrans

# Required to call maketrans function.

intab = "aeiou"

outtab = "12345"

trantab = maketrans(intab, outtab)

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

print str.translate(trantab);

**Result** th3s 3s str3ng 2x1mpl2....w4w!!!

Following is the example to delete 'x' and 'm' characters from the string:

#!/usr/bin/python

from string import maketrans

# Required to call maketrans function.

intab = "aeiou"

outtab = "12345"

trantab = maketrans(intab, outtab)

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

print str.translate(trantab, 'xm');

**Result** th3s 3s str3ng 21pl2....w4w!!!

**upper()** The method upper() returns a copy of the string in which all case-based characters have been uppercased.

**Syntax** str.upper()

**Parameters** NA

**Return Value** This method returns a copy of the string in which all case-based characters have been

uppercased.

**Example** #!/usr/bin/python

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

print "str.capitalize() : ", str.upper()

**Result** THIS IS STRING EXAMPLE....WOW!!!

**zfill (width)** The method zfill() pads string on the left with zeros to fill width.

**Syntax** str.zfill(width)

**Parameters** width -- This is final width of the string. This is the width which we would get after

filling zeros.

**Return Value** This method returns padded string.

**Example** #!/usr/bin/python

r = "this is string example....wow!!!";

print str.zfill(40);

print str.zfill(50);

**Result** 00000000this is string example....wow!!!

000000000000000000this is string example....wow!!!

**isdecimal()** The method isdecimal() checks whether the string consists of only decimal

characters. This method are present only on unicode objects.

Note: To define a string as Unicode, one simply prefixes a 'u' to the opening quotation

mark of the assignment. Below is the example.

**Syntax** str.isdecimal()

**Parameters** NA

**Return Value** This method returns true if all characters in the string are decimal, false otherwise.

**Example** #!/usr/bin/python

str = u"this2009";

print str.isdecimal();

str = u"23443434";

print str.isdecimal();

**Result** False

True

01-Feb-2017 **LISTS**  Pradeep

The list is a most versatile datatype available in Python which can be written as a list of comma-separated values (items) between square brackets. Important thing about a list is that items in a list need not be of the same type.

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

list\_n = [] #Empty List

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

list2 = [1, 2, 3, 4, 5 ];

list3 = ["a", "b", "c", "d"];

list4 = [1,2,5,[4,5,8,6,],5,6,9,10] #List in a list, this inner list will have its own property(list)

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

**Accessing Values in Lists**

To access values in lists, use the square brackets for slicing along with the index or

indices to obtain value available at that index.

**Example** #!/usr/bin/python

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

list2 = [1, 2, 3, 4, 5, 6, 7 ];

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

print "list2[1:5]: ", list2[1:5]

**Result** list1[0]: physics

list2[1:5]: [2, 3, 4, 5]

**Updating Lists**

You can update single or multiple elements of lists by giving the slice on the left-hand side of the assignment operator, and you can add to elements in a list with the append() method.

**Example** #!/usr/bin/python

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];

**Result** Value available at index 2 : 1997

New value available at index 2 : 2001

**Deleting List Elements**

To remove a list element, you can use either the del statement if you know exactly which element(s) you are deleting or the remove() method if you do not know.

**Example** #!/usr/bin/python

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

print list1;

del list1[2];

print "After deleting value at index 2 : "

print list1;

**Result** ['physics', 'chemistry', 1997, 2000]

After deleting value at index 2 :

['physics', 'chemistry', 2000]

**Basic List Operations**

Lists respond to the + and \* operators much like strings; they mean concatenation and repetition here too, except that the result is a new list, not a string.

In fact, lists respond to all of the general sequence operations we used on strings in the prior chapter.

**Python** **Expression Results** **Description**

len([1, 2, 3]) 3 Length

[1, 2, 3] + [4, 5, 6] [1, 2, 3, 4, 5, 6] Concatenation

['Hi!'] \* 4 ['Hi!', 'Hi!', 'Hi!', 'Hi!'] Repetition

3 in [1, 2, 3] True Membership

for x in [1, 2, 3]: print x, 1 2 3 Iteration

**Indexing, Slicing**

Because lists are sequences, indexing and slicing work the same way for lists as they do for strings.

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

L[Start:Stop:Step] .Initially Start =0 and Step =1 if we dont specify any thing(L[3])

Python Expression Results Description

L[2] 'SPAM!' Offsets start at zero

L[-2] 'Spam' Negative: count from the right

L[1:] ['Spam', 'SPAM!'] Slicing fetches sections

Try all these:

L[:]

L[::]

L[1:]

L[:8:]

L[:2]

L[:6:2]

L[::-1]

L[:-1]

L[-6:-1:1]

L[6:1:-1]

**Built-in List Functions and Methods**

**Cmp(list1, list2)** The method cmp() compares elements of two lists.

**Syntax** cmp(list1, list2)

**Parameters** list1 -- This is the first list to be compared.

list2 -- This is the second list to be compared.

**Return Value** If elements are of the same type, perform the compare and return the result. If

elements are different types, check to see if they are numbers.

If numbers, perform numeric coercion if necessary and compare.

If either element is a number, then the other element is "larger" (numbers are "smallest").

Otherwise, types are sorted alphabetically by name.

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

**Example** #!/usr/bin/python

list1, list2 = [123, 'xyz'], [456, 'abc']

print cmp(list1, list2);

print cmp(list2, list1);

list3 = list2 + [786];

print cmp(list2, list3)

**Result** -1

1

-1

**len(List)** The method len() returns the number of elements in the list.

**Syntax** len(list)

**Parameters** list -- This is a list for which number of elements to be counted.

**Return Value** This method returns the number of elements in the list.

**Example** #!/usr/bin/python

list1, list2 = [123, 'xyz', 'zara'], [456, 'abc']

print "First list length : ", len(list1);

print "Second list length : ", len(list2);

**Result** First list length : 3

Second list length : 2

**max(list)**The method max returns the elements from the list with maximum value.

**Syntax** max(list)

**Parameters** list -- This is a list from which max valued element to be returned.

**Return Value** This method returns the elements from the list with maximum value.

**Example** #!/usr/bin/python

list1, list2 = [123, 'xyz', 'zara', 'abc'], [456, 700, 200]

print "Max value element : ", max(list1);

print "Max value element : ", max(list2);

**Result** Max value element : zara

Max value element : 700

**Note:** If the list consists of a inner list along with numbers only then the inner list be will the max element.(Only when elements are integers)

**Example** #!/usr/bin/python

list1 = [123, -5, 1010, 5614,254,[456, 700, 200]]

print "Max value element : ", max(list1)

**Result** Max value element : [456, 700, 200]

Note: If the list consists of a inner list along then the string will be the max element.

**Example** #!/usr/bin/python

list1 = [123, 'xyz', 'zar', 'abc', [456, 700, 200]]

print "Max value element : ", max(list1);

**Result** Max value element : zara

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

**Syntax** min(list)

**Parameters** list -- This is a list from which min valued element to be returned.

**Return Value** This method returns the elements from the list with minimum value.

**Example** #!/usr/bin/python

list1, list2 = [123, 'xyz', 'zara', 'abc'], [456, 700, 200]

print "min value element : ", min(list1);

print "min value element : ", min(list2);

**Result** min value element : 123

min value element : 200

**Python includes following list methods**

**List.append(obj)** The method append() appends a passed obj into the existing list.

**Syntax** list.append(obj)

**Parameters** obj -- This is the object to be appended in the list.

**Return Value** This method does not return any value but updates existing list.

**Example** #!/usr/bin/python

aList = [123, 'xyz', 'zara', 'abc'];

aList.append( 2009 );

print "Updated List : ", aList;

**Result** Updated List : [123, 'xyz', 'zara', 'abc', 2009]

**list.count(obj)** The method count() returns count of how many times obj occurs in list.

**Syntax** list.count(obj)

**Parameters** obj -- This is the object to be counted in the list.

**Return Value** This method returns count of how many times obj occurs in list.

**Example** #!/usr/bin/python

aList = [123, 'xyz', 'zara', 'abc', 123];

print "Count for 123 : ", aList.count(123);

print "Count for zara : ", aList.count('zara');

**Result** Count for 123 : 2

Count for zara :1

**list.extend(seq)** The method extend() appends the contents of seq to list.

**Syntax** list.extend(seq)

**Parameters** seq -- This is the list of elements

**Return Value** This method does not return any value but add the content to existing list.

**Example** #!/usr/bin/python

aList = [123, 'xyz', 'zara', 'abc', 123];

bList = [2009, 'manni'];

aList.extend(bList)

print "Extended List : ", aList ;

**Result** Extended List : [123, 'xyz', 'zara', 'abc', 123, 2009, 'manni']

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

**Syntax** list.index(obj)

**Parameters** obj -- This is the object to be find out.

**Return Value** This method returns index of the found object otherwise raise an exception indicating

that value does not find.

**Example** #!/usr/bin/python

aList = [123, 'xyz', 'zara', 'abc'];

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

print "Index for zara : ", aList.index( 'zara' ) ;

**Result** Index for xyz : 1

Index for zara : 2

**list.insert(index,obj)** The method insert() inserts object obj into list at offset index.

**Syntax** list.insert(index, obj)

**Parameters** index -- This is the Index where the object obj need to be inserted.

obj -- This is the Object to be inserted into the given list.

**Return Value** This method does not return any value but it inserts the given element at the given index.

**Example** #!/usr/bin/python

aList = [123, 'xyz', 'zara', 'abc']

aList.insert( 3, 2009)

print "Final List : ", aList

**Result** Final List : [123, 'xyz', 'zara', 2009, 'abc']

**list.pop(obj=list[-1])** The method pop() removes and returns last object or obj from the list.

**Syntax** list.pop(obj=list[-1])

**Parameters** obj -- This is an optional parameter, index of the object to be removed from the list.

**Return Value** This method returns the removed object from the list.

**Example** #!/usr/bin/python

aList = [123, 'xyz', 'zara', 'abc'];

print "A List : ", aList.pop();

print "B List : ", aList.pop(2);

**Result** A List : abc

B List : zara

**list.remove(obj)** The method remove() removes object or obj from the list.

**Parameters** obj -- This is the object to be removed from the list.

**Return Value** This method does not return any value but removes the given object from the list.

**Example** #!/usr/bin/python

aList = [123, 'xyz', 'zara', 'abc', 'xyz'];

aList.remove('xyz');

print "List : ", aList;

aList.remove('abc');

print "List : ", aList;

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

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

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

**Syntax** list.reverse()

**Parameters** NA

**Return Value** This method does not return any value but reverse the given object from the list.

**Example** #!/usr/bin/python

aList = [123, 'xyz', 'zara', 'abc', 'xyz'];

aList.reverse();

print "List : ", aList;

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

**list.sort([func])** The sort() method sorts objects of list, use compare function if given.

**Syntax** list.sort([func])

**Parameters** NA

**Return Value** This method does not return any value but reverses the given object from the list.

**Example** #!/usr/bin/python3

list1 = ['physics', 'Biology', 'chemistry', 'maths']

list1.sort()

print ("list now : ", list1)

**Result**  list now : ['Biology', 'chemistry', 'maths', 'physics']

**List Processing- Sort**

#!/usr/bin/python

from random import randrange # It will only import randrange

def random\_list():

result = []

count = randrange(3,20)

for index1 in range(count):

result+=[randrange(-50,50)]

return result

def selection\_sort(lst):

nitems = len(lst)

for index2 in range(nitems-1):

small = index2

for index3 in range(index2 + 1,nitems):

if lst[index3]<lst[small]:

small = index3

if index2 != small:

lst[index2],lst[small]=lst[small],lst[index2]

return

def main():

for n in range(10):

col = random\_list()

print col

selection\_sort(col)

print col

print"============================================================="

main()

02-Feb-2017 **TUPLES** Pradeep

Tuples are sequences, just like lists. The differences between tuples and lists are, the tuples cannot be changed unlike lists and tuples use parentheses(), whereas lists use square brackets[].

Creating a tuple is as simple as putting different comma-separated values.

Optionally you can put these comma-separated values between parentheses also.

**Example**

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

tup2 = (1, 2, 3, 4, 5 );

tup3 = "a", "b", "c", "d";

tup1 = (); # Empty tuple

tup1 = (50,); # tuple with single value

Like string indices, tuple indices start at 0, and they can be sliced, concatenated, and so on.

**Accessing Values in Tuples**

To access values in tuple, use the square brackets for slicing along with the index or indices to obtain value available at that index.

**Example** #!/usr/bin/python

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

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

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

print "tup2[1:5]: ", tup2[1:5]

**Result** tup1[0]: physics

tup2[1:5]: [2, 3, 4, 5]

**Updating Tuples**

Tuples are immutable which means you cannot update or change the values of tuple elements. You are able to take portions of existing tuples to create new tuples.

**Example** #!/usr/bin/python

tup1 = (12, 34.56);

tup2 = ('abc', 'xyz');

# Following action is not valid for tuples

# tup1[0] = 100;

# So let's create a new tuple as follows

tup3 = tup1 + tup2;

print tup3;

**Result**

(12, 34.56, 'abc', 'xyz')

**Deleting Tuple Elements**

Removing individual tuple elements is not possible. There is, of course, nothing wrong with putting together another tuple with the undesired elements discarded.

To explicitly remove an entire tuple, just use the del statement.

**Example** #!/usr/bin/python

tup = ('physics', 'chemistry', 1997, 2000);

print tup;

del tup;

print "After deleting tup : "

print tup;

**Result**  ('physics', 'chemistry', 1997, 2000)

After deleting tup :

Traceback (most recent call last):

File "test.py", line 9, in <module>

print tup;

NameError: name 'tup' is not defined

**Basic Tuples Operations**

Tuples respond to the + and \* operators much like strings; they mean concatenation and repetition here too, except that the result is a new tuple, not a string.

**Python Expression Results Description**

len((1, 2, 3)) 3 Length

(1, 2, 3) + (4, 5, 6) (1, 2, 3, 4, 5, 6) Concatenation

('Hi!',) \* 4 ('Hi!', 'Hi!', 'Hi!', 'Hi!') Repetition

3 in (1, 2, 3) True Membership

for x in (1, 2, 3): print x, 1 2 3 Iteration

**Indexing, Slicing** Because tuples are sequences, indexing and slicing work the same way for tuples as they do for strings.

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

**Python Expression Results Description**

L[2] 'SPAM!' Offsets start at zero

L[-2] 'Spam' Negative: count from the right

L[1:] ['Spam', 'SPAM!'] Slicing fetches sections

Note: Try the examples mentionjes in Lists

**No Enclosing Delimiters**

Any set of multiple objects, comma-separated, written without identifying symbols, i.e., brackets for lists, parentheses for tuples, etc., default to tuples.

**Examples** #!/usr/bin/python

print 'abc', -4.24e93, 18+6.6j, 'xyz';

x, y = 1, 2;

print "Value of x , y : ", x,y;

**Result** abc -4.24e+93 (18+6.6j) xyz

Value of x , y : 1 2

**Cmp(tuple1, tuple2)** The method cmp() compares elements of two tuples.

**Syntax** cmp(tuple1, tuple2)

**Parameters** tuple1 -- This is the first tuple to be compared

tuple2 -- This is the second tuple to be compared

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

If numbers, perform numeric coercion if necessary and compare.

If either element is a number, then the other element is "larger" (numbers are "smallest").

Otherwise, types are sorted alphabetically by name.

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

**Example** #!/usr/bin/python

tuple1, tuple2 = (123, 'xyz'), (456, 'abc')

print cmp(tuple1, tuple2);

print cmp(tuple2, tuple1);

tuple3 = tuple2 + (786,);

print cmp(tuple2, tuple3)

**Result** -1

1

-1

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

**Syntax** len(tuple)

**Parameters** tuple -- This is a tuple for which number of elements to be counted.

**Return Value** This method returns the number of elements in the tuple.

**Example** #!/usr/bin/python

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

print "First tuple length : ", len(tuple1);

print "Second tuple length : ", len(tuple2);

**Result**  First tuple length : 3

Second tuple length : 2

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

**Syntax** max(tuple)

**Parameters** tuple -- This is a tuple from which max valued element to be returned.

**Return Value** This method returns the elements from the tuple with maximum value.

**Example** #!/usr/bin/python

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

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

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

**Result** Max value element : zara

Max value element : 700

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

**Syntax** min(tuple)

**Parameters** tuple -- This is a tuple from which min valued element to be returned.

**Return Value** This method returns the elements from the tuple with minimum value.

**Example** #!/usr/bin/python

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

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

print "min value element : ", min(tuple2);

**Result** min value element : 123

min value element : 200

**tuple(seg)** The **tuple()** method converts a list of items into tuples.

**Syntax** tuple( seq )

**Parameters** seq -- This is a tuple to be converted into tuple.

**Return Value** This method returns the tuple.

**Example** #!/usr/bin/python

aList = (123, 'xyz', 'zara', 'abc');

aTuple = tuple(aList)

print "Tuple elements : ", aTuple

**Result** Tuple elements :

(123, 'xyz', 'zara', 'abc')

**LINEAR SEARCH**

#!/usr/bin/python

def locate(lst,seek):

for index in range(len(lst)):

if (lst[index]==seek):

return index

return None

def display(lst,value):

position = locate(lst,value)

if position!= None:

print value,"is found at position",position

else:

print value,"is not found"

return

def main():

lst = [10,20,3,5,10,6,9,4]

seek = input("Enter Search Element: ")

display(lst,seek)

main()

**BINARY SEARCH**

#!/usr/bin/python

def binary\_search(lst,seek):

first = 0 ;last = len(lst)-1

while first<=last:

mid = first + (last - first + 1)//2

if lst[mid] == seek:

return mid

elif lst[mid]>seek:

last = mid - 1

else:

first = mid +1

return None

def display(lst,value):

pos = binary\_search(lst,value)

if (pos != None):

print value,"value found at ",pos

else:

print value,"not found"

return

def main():

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

value = input("Enter Search Element: ")

display(lst,value)

main()

03-Feb-2017  **DICTIONARY** Pradeep

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: {}.

Keys are unique within a dictionary while values may not be. The values of a dictionary can be of any type, but the keys must be of an immutable data type such as strings, numbers, or tuples.

**Accessing Values in Dictionary**

To access dictionary elements, you can use the familiar square brackets along with the key to obtain its value.

**Example** #!/usr/bin/python

dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'};

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

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

**Result** dict['Name']: Zara

dict['Age']: 7

If we attempt to access a data item with a key, which is not part of the dictionary, we get an error as follows:

**Example** #!/usr/bin/python

dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'};

print "dict['Alice']: ", dict['Alice'];

**Result** dict['Zara']:

Traceback (most recent call last):

File "test.py", line 4, in <module>

print "dict['Alice']: ", dict['Alice'];

KeyError: 'Alice'

**Updating Dictionary** You can update a dictionary by adding a new entry or a key-value pair, modifying an existing entry, or deleting an existing entry as shown below in the simple example:

**Example** #!/usr/bin/python

dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'};

dict['Age'] = 8; # update existing entry

dict['School'] = "DPS School"; # Add new entry

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

print "dict['School']: ", dict['School'];

**Result**  dict['Age']: 8

dict['School']: DPS School

**Delete Dictionary Elements** You can either remove individual dictionary elements or clear the entire contents of a dictionary. You can also delete entire dictionary in a single operation.

**Example** #!/usr/bin/python

dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'};

del dict['Name']; # remove entry with key 'Name'

dict.clear(); # remove all entries in dict

del dict ; # delete entire dictionary

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

print "dict['School']: ", dict['School'];

This produces the following result. Note that an exception is raised because after del dict, dictionary does not exist anymore:

dict['Age']:

Traceback (most recent call last):

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

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

TypeError: 'type' object is unsubscriptable

**Properties of Dictionary Keys**

Dictionary values have no restrictions. They can be any arbitrary Python object, either standard objects or user-defined objects. However, same is not true for the keys. There are two important points to remember about dictionary keys:

**(a)** More than one entry per key not allowed. Which means no duplicate key is allowed. When duplicate keys encountered during assignment, the last assignment wins.

**Example** #!/usr/bin/python

dict = {'Name': 'Zara', 'Age': 7, 'Name': 'Manni'};

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

**Result** dict['Name']: Manni

**(b)** Keys must be immutable. Which means you can use strings, numbers or tuples

as dictionary keys but something like ['key'] is not allowed.

**Example** #!/usr/bin/python

dict = {['Name']: 'Zara', 'Age': 7};

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

Result Traceback (most recent call last):

File "test.py", line 3, in <module>

dict = {['Name']: 'Zara', 'Age': 7};

TypeError: list objects are unhashable

**Built-in Dictionary Functions and Methods**

**Cmp(dict1, dict2)** The method cmp() compares two dictionaries based on key and values.

**Syntax** cmp(dict1, dict2)

**Parameters** dict1 -- This is the first dictionary to be compared with dict2.

dict2 -- This is the second dictionary to be compared with dict1.

**Return Value** This method returns 0 if both dictionaries are equal, -1 if dict1 < dict2, and 1 if dict1

> dic2.

**Example** #!/usr/bin/python

dict1 = {'Name': 'Zara', 'Age': 7};

dict2 = {'Name': 'Mahnaz', 'Age': 27};

dict3 = {'Name': 'Abid', 'Age': 27};

dict4 = {'Name': 'Zara', 'Age': 7};

print "Return Value : %d" % cmp (dict1, dict2)

print "Return Value : %d" % cmp (dict2, dict3)

print "Return Value : %d" % cmp (dict1, dict4)

**Result** Return Value : -1

Return Value : 1

Return Value : 0

**len(dict)** The method len() gives the total length of the dictionary. This would be equal to the

number of items in the dictionary.

**Syntax** len(dict)

**Parameters** dict -- This is the dictionary, whose length needs to be calculated.

**Return Value** This method returns the length.

**Example** #!/usr/bin/python

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

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

**Result** Length : 2

**str(dict)** The method str() produces a printable string representation of a dictionary.

**Syntax** str(dict)

**Parameters** dict -- This is the dictionary.

**Return Value** This method returns string representation.

**Example** #!/usr/bin/python

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

print "Equivalent String : %s" % str (dict)

**Result** Equivalent String : {'Age': 7, 'Name': 'Zara'}

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

**Syntax** type(dict)

**Parameters** dict -- This is the dictionary.

**Return Value** This method returns the type of the passed variable.

**Example** #!/usr/bin/python

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

print "Variable Type : %s" %

type (dict)

**Result** Variable Type : <type 'dict'>

**dict.clear()** The method clear() removes all items from the dictionary.

**Syntax** dict.clear()

**Parameters** NA

**Return Value** This method does not return any value.

**Example** #!/usr/bin/python

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

print "Start Len : %d" %

len(dict)

dict.clear()

print "End Len : %d" %

len(dict)

**Result** Start Len : 2

End Len : 0

**dict.copy()** The method copy() returns a shallow copy of the dictionary.

**Syntax** dict.copy()

**Parameters** NA

**Return Value** This method returns a shallow copy of the dictionary.

**Example** #!/usr/bin/python

dict1 = {'Name': 'Zara', 'Age': 7};

dict2 = dict1.copy()

print "New Dictinary : %s" %

str(dict2)

**Result** New Dictinary : {'Age': 7, 'Name': 'Zara'}

**dict.fromkeys()** The method fromkeys() creates a new dictionary with keys from seq and values set

to value.

**Syntax**  dict.fromkeys(seq[, value]))

**Parameters** seq -- This is the list of values which would be used for dictionary keys preparation.

value -- This is optional, if provided then value would be set to this value

**Return Value** This method returns the list.

**Example** #!/usr/bin/python

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

dict = dict.fromkeys(seq)

print "New Dictionary : %s" %

str(dict)

dict = dict.fromkeys(seq, 10)

print "New Dictionary : %s" %

str(dict)

**Result** New Dictionary : {'age': None, 'name': None, 'sex': None}

New Dictionary : {'age': 10, 'name': 10, 'sex': 10}

**dict.get(key,default=none)** The method get() returns a value for the given key. If key is not available then returns default value None.

**Syntax** dict.get(key, default=None)

**Parameters** key -- This is the Key to be searched in the dictionary.

default -- This is the Value to be returned in case key does not exist.

**Return Value** This method return a value for the given key. If key is not available, then returns

default value None.

**Example** #!/usr/bin/python

dict = {'Name': 'Zabra', 'Age': 7}

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

print "Value : %s" % dict.get('Education', "Never")

**Result** Value : 7

Value : Never

**dict.has\_key(key)** The method has\_key() returns true if a given key is available in the dictionary,

otherwise it returns a false.

**Syntax** dict.has\_key(key)

**Parameters** key -- This is the Key to be searched in the dictionary.

**Return Value** This method return true if a given key is available in the dictionary, otherwise it

returns a false.

**Example** #!/usr/bin/python

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

print "Value : %s" % dict.has\_key('Age')

print "Value : %s" % dict.has\_key('Sex')

**Result** Value : True

Value : False

**dict.items()** The method items() returns a list of dict's (key, value) tuple pairs

**Syntax** dict.items()

**Parameters** NA

**Return Value** This method returns a list of tuple pairs.

**Example** #!/usr/bin/python

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

print "Value : %s" %

dict.items()

**Result** Value : [('Age', 7), ('Name', 'Zara')]

**dict.keys()** The method keys() returns a list of all the available keys in the dictionary.

**Syntax** dict.keys()

**Parameters** NA

**Return Value** This method returns a list of all the available keys in the dictionary.

**Example** #!/usr/bin/python

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

print "Value : %s" % dict.keys()

**Result** Value : ['Age', 'Name']

**dict.setdefault(key, default=None)** The method setdefault() is similar to get(), but will set dict[key]=default if key is not already in dict.

**Syntax** dict.setdefault(key, default=None)

**Parameters** key -- This is the key to be searched.

default -- This is the Value to be returned in case key is not found.

**Return Value** This method returns the key value available in the dictionary and if given key is not

available then it will return provided default value.

**Example** #!/usr/bin/python

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

print "Value : %s" % dict.setdefault('Age', None)

print "Value : %s" % dict.setdefault('Sex', None)

**Result** Value : 7

Value : None

**dict.update(dict2)** The method update() adds dictionary dict2's key-values pairs in to dict. This function does not return anything.

**Syntax** dict.update(dict2)

**Parameters** dict2 -- This is the dictionary to be added into dict.

**Return Value** This method does not return any value.

**Example** #!/usr/bin/python

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

dict2 = {'Sex': 'female' }

dict.update(dict2)

print "Value : %s" % dict

**Result** Value : {'Age': 7, 'Name': 'Zara', 'Sex': 'female'}

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

**Syntax** dict.values()

**Parameters** NA

**Return Value** This method returns a list of all the values available in a given dictionary.

**Example** #!/usr/bin/python

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

print "Value : %s" %

dict.values()

**Result** Value : [7, 'Zara']

**map, filter, and reduce**

Python provides several functions which enable a **functional approach** to programming. These functions are all convenience features in that they can be written in Python fairly easily.

Functional programming is all about expressions. We may say that the Functional programming is an expression oriented programming.

Expression oriented functions of Python provides are:

1. map(aFunction, aSequence)
2. filter(aFunction, aSequence)
3. reduce(aFunction, aSequence)
4. lambda
5. list comprehension

map

One of the common things we do with list and other sequences is applying an operation to each item and collect the result.

For example, updating all the items in a list can be done easily with a for loop:

>>> items = [1, 2, 3, 4, 5]

>>> squared = []

>>> for x in items:

>>> squared.append(x \*\* 2)

>>> squared[1, 4, 9, 16, 25]

Since this is such a common operation, actually, we have a built-in feature that does most of the work for us.

The map(aFunction, aSequence) function applies a passed-in function to each item in an iterable object and returns a list containing all the function call results.

>>> items = [1, 2, 3, 4, 5]

>>> def sqr(x): return x \*\* 2

>>> list(map(sqr, items))

>>>[1, 4, 9, 16, 25]

We passed in a user-defined function applied to each item in the list. map calls sqr on each list item and collects all the return values into a new list.

Because map expects a function to be passed in, it also happens to be one of the places where lambda routinely appears:

>>> list(map((lambda x: x \*\*2), items))

>>>[1, 4, 9, 16, 25]

In the short example above, the lambda function squares each item in the items list.

As shown earlier, map is defined like this:map(aFunction, aSequence)

While we still use lamda as a aFunction, we can have a list of functions as aSequence:

def square(x):

return (x\*\*2)

def cube(x):

return (x\*\*3)

funcs = [square, cube]

for r in range(5):

value = map(lambda x: x(r), funcs)

print value

Output:

[0, 0]

[1, 1]

[4, 8]

[9, 27]

[16, 64]

Because using map is equivalent to for loops, with an extra code we can always write a general mapping utility:

>>> def mymap(aFunc, aSeq):

result = []

for x in aSeq: result.append(aFunc(x))

return result

>>> list(map(sqr, [1, 2, 3]))

>>>[1, 4, 9]

>>> mymap(sqr, [1, 2, 3])

>>>[1, 4, 9]

Since it's a built-in, map is always available and always works the same way. It also has some performance benefit because it is usually faster than a manually coded for loop. On top of those, map can be used in more advance way. For example, given multiple sequence arguments, it sends items taken form sequences in parallel as distinct arguments to the function:

>>> pow(3,5)

>>>243

>>> pow(2,10)

>>>1024

>>> pow(3,11)

>>>177147

>>> pow(4,12)

>>>16777216

>>> list(map(pow,[2, 3, 4], [10, 11, 12]))

>>>[1024, 177147, 16777216]

As in the example above, with multiple sequences, map() expects an N-argument function for N sequences. In the example, pow function takes two arguments on each call.

The map call is similar to the list comprehension expression. But map applies a function call to each item instead of an arbitrary expression. Because of this limitation, it is somewhat less general tool. In some cases, however, map may be faster to run than a list comprehension such as when mapping a built-in function. And map requires less coding.

If function is None, the identity function is assumed; if there are multiple arguments, map() returns a list consisting of tuples containing the corresponding items from all iterables (a kind of transpose operation). The iterable arguments may be a sequence or any iterable object; the result is always a list:

>>> m = [1,2,3]

>>> n = [1,4,9]

>>> new\_tuple = map(None, m, n)

>>> new\_tuple

[(1, 1), (2, 4), (3, 9)]

**07-02-2017** **J Vinay kumar**

**MODULES**

A module is a file consisting of Python code. A module can define functions,classes and variables. A module can also include runnable code.

There are two types of modules:

**1.Userdefined modules**

**2.Basic essential Inbuilt modules – sys, math, time, os**

**import statement:**

You can use any Python source file as a module by executing an import statement in

some other Python source file.

Syntax:

import modulename1

or

import modulename1,modulename2,..........modulenameN

**Example:**

**$ #!/usr/bin/python**

**$ import math**

This statement does import the entire module math into the current code.

**math module** contains different mathematical functions like sqrt,ceil,floor..etc

**Different ways of importing:**

**1.import module**

**2.from module import <function or class > #(import specific function or class)**

**3.from package import <module> #(import specific module)**

**4.from package import \* #(\*-->all modules)**

**5.from module import \*#(\*--> all classes and functions)**

**from import statement:**

Python's from statement lets you import specific attributes from a module into the

current code.

Syntax: **from modulename import functionname**

**Example:**

**$ #!/usr/bin/python**

**$ from math import sqrt**

**$ num=25**

**$ result=math.sqrt(num)**

**$ print 'square root of the number is:',result**

**o/p: square root of the number is:5**

**Example for userdefined modules:**

Step1:

save below code in file.py

**$#!/usr/bin/python**

**$def add(a,b):**

**$ result = a+b**

**$ return result**

**$def sub(a,b):**

**$ result=a-b**

**$ return result**

Step2:

create other file with file1.py

**#!/usr/bin/python**

**$import file**

**$result=file.add(1,2)**

**$print “sum of two numbers is:”,result**

**O/P: sum of two numbers is: 3**

**Example2:**

**step1: as same as above**

**step2:**

**#!/usr/bin/python**

**$from file import sub**

**$result=sub(1,2)**

**$print “subraction of two numbers is:”,result**

**$./file1.py**

**O/P:subraction of two numbers is:-1**

**Locating Modules:**

When you import a module, the Python interpreter searches for the module in the

following sequences:

1.current directory

2.If the module isn't found, Python then searches each directory in the shell

variable PYTHONPATH.

3.If all else fails, Python checks the default path. On UNIX, this default path is

normally /usr/local/lib/python/.

The PYTHONPATH is an environment variable, consisting of a list of directories. The

syntax of PYTHONPATH is the same as that of the shell variable PATH.

**07-02-2017** **J.vinaykumar**

**PACKAGES**

A package is a hierarchical file directory structure that defines a single Python

application environment that consists of modules and subpackages and sub-

subpackages, and so on.

Simply, package is a collection of modules and subpackeges and sub-subpackages.

Example:

NOTE1:create directory.In that directory create different modules.

NOTE2:move out of directory and import directory in other file

NOTE3:execute the file.

Follow below steps:

Step1:

Consider a file Pots.py ,isdn.py,G3.py available in Phone directory. This file has following line of

source code:

$ #!/usr/bin/python

$ def Pots():

'''Posts.py file'''

$ print “I'm Pots Phone”

$ #!/usr/bin/python

$ def isdn():

'''isdn.py file'''

$ print “I'm isdn Phone”

$ #!/usr/bin/python

$ def G3():

'''G3.py file'''

$ print “I'm G3 Phone”

Now, create one more file \_\_init\_\_.py in Phone directory.

$ #!/usr/bin/python

$ from pots import pots

$ from isdn import isdn

$ from g3 import g3

Note:To make all of your functions available when you've imported Phone, you need to put

explicit import statements in \_\_init\_\_.py .

Step2:

After you create the \_\_init\_\_.py, you have all of these classes available when

you import the Phone package in other file outside the phone directory.

-->File.py

#!/usr/bin/python

# Now import your Phone Package.

import Phone

Phone.Pots()

Phone.Isdn()

Phone.G3()

after executing File.py

O/P: I'm Pots Phone

I'm ISDN Phone

I'm 3G Phone

**07-02-2017 J.Vinaykumar**

**Globals and Locals**

If locals() is called from within a function, it will return all the names that can

be accessed locally from that function.

If globals() is called from within a function, it will return all the names that can

be accessed globally from that function.

Example:

Let's have a look at the following function:

#!/usr/bin/python

def func():

print str

str = "I hate spam"

func()

The variable str is defined as the string "I hate spam", before we call the function func(). The only statement in func() is the "print str" statement. As there is no local str, the value from the global str will be used. So the output will be the string "I hate spam". The question is, what will happen, if we change the value of str inside of the function func()? Will it affect the global str as well? We test it in the following piece of code:

def func():

str = "Me too."

print str

str = "I hate spam."

func()

print str

The output looks like this:

Me too.

I hate spam.

What if we combine the first example with the second one, i.e. first access str and then assigning a value to it? It will throw an error, as we can see in the following example:

def func():

print str

str = "Me too."

print str

str = "I hate spam."

func()

print str

If we execute the previous script, we get the error message:

UnboundLocalError: local variable 'str' referenced before assignment

Python "assumes" that we want a local variable due to the assignment to str inside of func(), so the first print statement throws this error message. Any variable which is changed or created inside of a function is local, if it hasn't been declared as a global variable. To tell Python, that we want to use the global variable, we have to use the keyword "global", as can be seen in the following example:

def func():

global str

print str

str = "That's clear."

print str

str = "Python is great!"

func()

print str

Now there is no ambiguity. The output is as follows:

Python is great!

That's clear.

That's clear.

Local variables of functions can't be accessed from outside, when the function call has finished:

def func():

str = "I am globally not known"

print str

func()

print str

If you start this script, you get an output with the following error message:

I am globally not known

Traceback (most recent call last):

File "global\_local3.py", line 6, in <module>

print s

NameError: name 'str' is not defined

Example2:

def foo(x, y):

global num

num = 42

x,y = y,x

num1 = 33

num1 = 17

num3 = 100

print num,num1,x,y

num,num1,x,y = 1,15,3,4

foo(17,4)

print num,num1,x,y

The output of the script above looks like this:

42 17 4 17

42 15 3 4

**08-07-2017 J.Vinaykumar**

**COMMAND LINE ARGUMENTS**

Python programs can be started using command line arguments.

For example:

$ python program.py num1 num2

where num1,num2 is an argument. You can choose any argument you want in your program.

(You only want to do this with command line programs)

Command line arguments in Python  
You can get access to the command line parameters using the sys module.  len(sys.argv) contains the number of arguments.  To print all of the arguments simply execute str(sys.argv)

Example:

**#!/usr/bin/python**

**import sys**

**print('Arguments:', len(sys.argv))**

**print('List:', str(sys.argv))**

$ **python example.py num1 num2**

**Arguments: 3  
List: [‘example.py’, ‘num1’, ‘num2’]**

Storing command line arguments  
You can store the arguments given at the start of the program in variables.  
For example, an program may start like this:

**#!/usr/bin/python**

**import sys**

**print('Arguments:', len(sys.argv))**

**print('List:', str(sys.argv))**

**if sys.argv < 2:**

**print('To few arguments, please specify a filename')**

**filename = sys.argv[1]**

**print('Filename:', filename)**

**O/P:**

**$ Python example.py num1**

**('Arguments:', 2)  
('List:', "['example.py', 'num1']")  
('Filename:', 'num1')**

-python provides a getopt module that helps you parse command-line options & arguments

-The python sys module provides access to any command-line arg via the sys.argv this serves two purpose

-sys.argv is the list of command line arguments

-len(sys.argv) is the no of command line arguments

-Here sys.argv[0] is the script name.

**PARSING COMMAND LINE ARGUMENTS:**

**syntax:getopt.getopt(args,options[,long-options])**

**Example:**

**#!/usr/bin/python**

**import sys,getopt**

**def main(argv):**

**inputfile='votary'**

**outputfile='tech'**

**try:**

**opts,args=getopt.getopt(argv,"hki:o:",["ifile=","ofile="])**

**except getopt.GetoptError:**

**print 'getopts\_test.py -i <inputfile> -o <outputfile>'**

**sys.exit(2)**

**for opt,arg in opts:**

**if opt == '-h':**

**print "usage getopts\_test.py -i <inputfile> -o <outputfile>"**

**sys.exit()**

**elif opt in ("-i","--ifile"):**

**inputfile = arg**

**elif opt in ("-o","--ofile"):**

**outputfile = arg**

**print "input file is",inputfile**

**print "output file is",outputfile**

**return**

**if \_\_name\_\_=="\_\_main\_\_":**

**main(sys.argv[1:])**

**O/P:**

**input file is votary**

**output file is tech**

**08-02-2017 J.Vinaykumar**

**Exceptions**

Almost all programming languages, except shell scripting and some scripting languages, possess exception handling capabilities.

There are two kinds of errors in Python.

1. syntax errors- If something went wrong, the resulting error code is ­1 to indicate the failure of a call.

2. Exception(Runtime error) - Used to handle exceptional cases.

In Python, the errors are handled by the interpreter by raising an exception and allowing that exception to be handled.

Exceptions indicate errors and break out of the normal control flow of a program. An exception is raised using **raise** statement.

Syntax:

try:

logic

...

except <ExceptionName1>, <alias identifier>:

logic to handle that exception

...

except <ExceptionName2> as <alias identifier>:

logic to handle that exception.

This logic gets executed, if error is not covered

in ExceptionName1 exception

...

else:

logic to execute if

there is no exception

...

finally:

logic to execute either

if exception occurs are not

...

Note : try and except are mandatory blocks. And, else and finally are optional blocks.

**Example**

Result = 21/0 Is ZeroDivisionError

ZeroDivisionError: integer division or modulo by zero

**#!/usr/bin/python**

**try:**

**result = 21/0**

**except:**

**print 'An error occurred!'**

NOTE:This code handles all exceptions. But, we should know the error to do corresponding action

**O/P:An error occurred!**

**Import sys**

**try:**

**result = 21/0**

**except Exception as ex:**

**print sys.exc\_type**

**print 'An error occurred'**

**print 'The error is %s'%(ex)**

**O/P:**

**<type 'exceptions.ZeroDivisionError'>**

**An error occurred**

**The error is integer division or modulo by zero**

**NameError Traceback (most recent call last)**

**<ipython‐input‐34‐e878884284c3> in <module>()**

**3 except:**

**4 print 'An error occurred'**

**‐‐‐‐> 5 print 'The error is %s'%(ex)**

**NameError: name 'ex' is not defined**

Note:Here, the exception was resulted in the exception block

**try:**

**result = (1+2.3)/(2\*4\*0)**

**except:**

**print 'An error occurred'**

**try:**

**print 'The error is %s'%(ex)**

**except:**

**print 'variable "ex" is not defined'**

**O/P:**

**An error occurred**

**variable "ex" is not defined**

**Example for raising exceptions:**

**#!/usr/bin/python**

**import math**

**num=input(“enter a number:”)**

**if(num<0):**

**raise RuntimeError(“entered negitive number please enter positive numbers”)**

**else:**

**print (math.sqrt(x))**

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.

ZeroDivisonError : 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.

**08-02-2017 J.vinaykumar**

**Files I/O**

**File operation modes**

**r** read only

**w** write only

**a** appending the data

**Note:** If you open an existing file with 'w' mode, it's existing data get vanished.

**r+** both for read and write

**a+** both for read and append

In windows, the data is stored in binary format. Placing this 'b' doesn't effect in unix and linux.

**rb** read only

**wb** write only

**ab** append only

**ab+** Both reading and appending data

Default file operation is read only.

**Accessing a file**

Taking a sample file, named test.txt

**#!/usr/bin/python**

**string = "Python programming is interesting\n\ It is coming with batteries, in built\n\It means that almost every operation has a module !\n\ "**

**fileHandler = open('test.txt', 'w') # creating a new file**

**fileHandler.write(string)**

**fileHandler.close()**

**O/P:**

**vi test.txt**

Python programming is interesting

It is coming with batteries, in built

It means that almost every operation has a module !

**f = open('test.txt', 'r') # Opening an exi sting file for reading**

**data1 = f.readline() # reads one line**

**f.close()**

**print type(data1), data1**

**O/P:**

**<type 'str'> Python programming is interesting**

**f = open('test.txt', 'r')**

**data2 = f.readlines() # reads all lines, but results list of each line, a s a string**

**f.close()**

**print type(data2), data2**

**O/P:**

**<type 'list'> ['Python programming is interesting\n', 'It is coming with batt**

**eries, in built\n', 'It means that almost every operation has a module !\n']**

**f = open('test.txt', 'r')**

**data3 = f.read() # reads entire file as a single string**

**f.close()**

**print type(data3), data3**

**O/P:**

**<type 'str'> Python programming is interesting**

**It is coming with batteries, in built**

**It means that almost every operation has a module !**

**with open('test.txt', 'r') as f: # file operation with context manager**

**data4 = f.read()**

**f.close()**

**print type(data4), data4**

**O/P:**

**<type 'str'> Python programming is interesting**

**It is coming with batteries, in built**

**It means that almost every operation has a module !**

**#!/usr/bin/python**

**fo=open("tes.txt",'a+')**

**fo.seek(0)**

**fo.write("jdhshfd")**

**fo.close()**

**O/p:**

**jdhshfdPython programming is interesting**

**It is coming with batteries, in built**

**It means that almost every operation has a module !**

**09-02-2017 J.Vinaykumar**

**NETWORKING**

Python provides two levels of access to network services. At a low level, you can

access the basic socket support in the underlying operating system, which allows you

to implement clients and servers for both connection-oriented and connectionless

protocols.

Python also has libraries that provide higher-level access to specific application-level

network protocols, such as FTP, HTTP, and so on.

This chapter gives you understanding on most famous concept in Networking – Socket

**SOCKET MODULE**

To create a socket, you must use the socket.socket() function in socket module, which has the general syntax:

available

**s = socket.socket (socket\_family, socket\_type, protocol=0)**

Here is the description of the parameters:

->socket\_family: This is either AF\_UNIX or AF\_INET, as explained earlier.

->socket\_type: This is either SOCK\_STREAM or SOCK\_DGRAM.

->protocol: This is usually left out, defaulting to 0.

Once you have socket object, then you can use required functions to create your

client or server program. Following is the list of functions required:

Server Socket Methods:

-> s.bind() : This method binds address (hostname, port number pair) to socket.

-> s.listen() :This method sets up and start TCP listener.

-> s.accept() :This passively accept TCP client connection, waiting until

connection arrives (blocking).

Client Socket Methods:

->s.connect() :This method actively initiates TCP server connection.

General Socket Methods :

s.recv() : This method receives TCP message

s.send() :This method transmits TCP message

s.recvfrom() : This method receives UDP message

s.sendto() :This method transmits UDP message

s.close() :This method closes socket

socket.gethostname(): Returns the hostname.

Sample program:

#!/usr/bin/python # This is client.py file

import socket # Import socket module

s = socket.socket() # Create a socket object

host = socket.gethostname() # Get local machine name

port = 12345 # Reserve a port for your service.

s.connect((host, port))

print s.recv(1024)

s.close # Close the socket when done

#!/usr/bin/python # This is server.py file

import socket # Import socket module

s = socket.socket() # Create a socket object

host = socket.gethostname() # Get local machine name

port = 12345 # Reserve a port for your service.

s.bind((host, port)) # Bind to the port

s.listen(5) # Now wait for client connection.

while True:

c, addr = s.accept() # Establish connection with client.

print 'Got connection from', addr

c.send('Thank you for connecting')

c.close() # Close the connection

Now run this server.py in background and then run above client.py to see the result.

# Following would start a server in background.

$ python server.py &

# Once server is started run client as follows:

$ python client.py

This would produce following result:

Got connection from ('127.0.0.1', 48437)

Thank you for connecting

Block diagram for server client connections:![](data:None;base64,)

Simple Socket Program:

In the following code, the server sends the current time string to the client:

# server.py

import socket

import time

# create a socket object

serversocket = socket.socket(

socket.AF\_INET, socket.SOCK\_STREAM)

# get local machine name

host = socket.gethostname()

port = 9999

# bind to the port

serversocket.bind((host, port))

# queue up to 5 requests

serversocket.listen(5)

while True:

# establish a connection

clientsocket,addr = serversocket.accept()

print("Got a connection from %s" % str(addr))

currentTime = time.ctime(time.time()) + "\r\n"

clientsocket.send(currentTime.encode('ascii'))

clientsocket.close()

Here is the summary of the key functions from ***socket:***

1. **socket.socket()**: Create a new socket using the given address family, socket type and protocol number.
2. **socket.bind(address)**: Bind the socket to **address**.
3. **socket.listen(backlog)**: Listen for connections made to the socket. The **backlog** argument specifies the maximum number of queued connections and should be at least 0; the maximum value is system-dependent (usually 5), the minimum value is forced to 0.
4. **socket.accept()**: The return value is a pair **(conn, address)** where **conn** is a new socket object usable to send and receive data on the connection, and **address** is the address bound to the socket on the other end of the connection.  
   At **accept()**, a new socket is created that is distinct from the named socket. This new socket is used solely for communication with this particular client.  
   For TCP servers, the socket object used to receive connections is not the same socket used to perform subsequent communication with the client. In particular, the **accept()** system call returns a new socket object that's actually used for the connection. This allows a server to manage connections from a large number of clients simultaneously.
5. **socket.send(bytes[, flags])**: Send data to the socket. The socket must be connected to a remote socket. Returns the number of **bytes** sent. Applications are responsible for checking that all data has been sent; if only some of the data was transmitted, the application needs to attempt delivery of the remaining data.
6. **socket.colse()**: Mark the socket closed. all future operations on the socket object will fail. The remote end will receive no more data (after queued data is flushed). Sockets are automatically closed when they are garbage-collected, but it is recommended to close() them explicitly.

Note that the **server** socket doesn't receive any data. It just produces **client** sockets. Each **clientsocket** is created in response to some other **client** socket doing a **connect()** to the host and port we're bound to. As soon as we've created that **clientsocket**, we go back to listening for more connections.

# client.py

import socket

# create a socket object

s = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

# get local machine name

host = socket.gethostname()

port = 9999

# connection to hostname on the port.

s.connect((host, port))

# Receive no more than 1024 bytes

tm = s.recv(1024)

s.close()

print("The time got from the server is %s" % tm.decode('ascii'))

The output from the run should look like this:

$ python server.py &

Got a connection from ('127.0.0.1', 54597)

$ python client.py

The time got from the server is THURSDAY 08 19:14:15 2017

"If you need fast **IPC** between two processes on one machine, you should look into whatever form of shared memory the platform offers. A simple protocol based around shared memory and locks or semaphores is by far the fastest technique."

Name:SHAIKH ISHAQUE

Date :08/02/2017

CLASES/OBJECT -OVERVIEW OF OOPS

Class :

it is a blue print of an object

just like structur class is a user defined data type.

Class is having a logical existence.it doesnot have physical existence.

Object:

it is an instance of a class.

it is having a physicsal existence

Instance Varible:

a variable that is defined inside the method and belong only to the

current instance of the class

Class variable:

a variable defind inside a class is called class variable

Function Overloading:

n number of function having same name but different signature is

called Function Overloading

signature:every functions having different number of argument

Inheritance:

“deriving a new class from existing one is called inheritance”

“acquiring the property of parent by child class is called Inheritance”

Operator Overloading:

it comes under the compiletime polymorphisam

it is nothing but different operator have different implementation

depending on their argument

Attribute Access:

emp1.age=7

emp1.age=8

del.emp1.age

OR we can use the following function

1. hasattr(emp1,'age') #return value of age attribute

2.setattr(emp1,'age',8) #get attribute at age 8

3.delattr(emp1,'age') #deleting attribute age

Built in class Attribute:

every python class keeps following built in attribute they can be access using dot operator

1. \_\_dict\_\_ : dictionary contain the class name space

2.\_\_doc\_\_ : class documentation string or none if undefined

3.\_\_name\_\_ : class name

4.\_\_module\_\_: module name in which the class is defined

5.\_\_bases\_\_ : a possible empty tupple containing the base class in the order of their occurance

in the base class

Example:

print “Employe.\_\_doc\_\_”,Employe.\_\_doc\_\_ #Employe is class name

print Employee.\_\_name\_\_

print Employee.\_\_doc\_\_

print Employee.\_\_module\_\_

print Employee.\_\_bases\_\_

print Employee.\_\_dict\_\_

Creating a class in python

class Classname:

Example:

class Employe:

#common base class to all employe

empcnt=0

def \_\_init\_\_(self,name,salary):

self.name=name

self.salary=salary

Employe.emp.cnt+=1

def displaycnt(self):

print 'Total employe%d”%Employe,empcnt

def displayEmplye(self):

print 'Name:',self.name,'salary:',self.salary

#creating a object

emp1=Employe('ishaque',2000) #this would creat a first object

emp2=Employe('shaikh',5000) #this would creat a second object

emp1.displaycnt()

emp1.displayEmploye(self)

emp2.displaycnt()

emp2.displayEmploye(self)

Destroying the object (garbage collection)

a=40 #creat a object

b=a #increase the reference count

c=[b] #decress the reference count

del a #decrease the reference count

b=100

c[0]=-1 #increase the reference count

\_\_del\_\_() it is called a destructor

Example:

class point:

X = 0

Y = 0

def \_\_init\_\_(self,n=0,z=0):

print 'in init'

self.x = n

self.y = z

def display(self):

print 'x ===',self.x

print 'y ===',self.y

def \_\_del\_\_(self):

self.display()

class\_name = self.\_\_class\_\_.\_\_name\_\_

print 'CLASS WITH NAME ::: ',class\_name,' ::: destroyed'

pt1 = point()

pt2 = pt1

pt3 = pt2

pt4 = point(10,20)

pt5 = point(10,20)

print id(pt1),id(pt2),id(pt3)

print id(pt4),id(pt5)

File based Example

#!/usr/bin/python

class textfile:

ntfiles = 0# count of number of textfile objects

def \_\_init\_\_(self,fname):

textfile.ntfiles+=1

self.name=fname#name

self.fh = open(fname)#handle for the file

self.lines = self.fh.readlines()

self.nlines = len(self.lines)#number of lines

self.nwords = 0#number of words

self.wordcount()

def wordcount(self):

"finds the number of words in the file"

for l in self.lines:

w = l.split()

self.nwords+=len(w)

def grep(self,target):

"prints out all lines containing target"

for i in self.lines:

if i.find(target)>=0:

print i

file1 = textfile('file1.txt')

file2 = textfile('file2.txt')

print "The number of textfles open",file1.ntfiles

print "Some info about them(name,lines,words):"

for myfile in [file1,file2]:

print myfile.name,myfile.nlines,myfile.nwords

file1.grep('example')

File based Example with static method

#!/usr/bin/python

class textfile:

\_ntfiles = 0# count of number of textfile objects

@staticmethod

def totfiles1():

print 'no.of text files (way1) is' , textfile.ntfiles

return

def totfiles():

print 'number of textfiles (way2) are',textfile.ntfiles

return

totfiles2 = staticmethod(totfiles)

def \_\_init\_\_(self,fname):

textfile.ntfiles+=1

self.name=fname#name

self.fh = open(fname)#handle for the file

self.lines = self.fh.readlines()

self.nlines = len(self.lines)#number of lines

self.nwords = 0#number of words

self.wordcount()

def wordcount(self):

"finds the number of words in the file"

for l in self.lines:

w = l.split()

self.nwords+=len(w)

def grep(self,target):

"prints out all lines containing target"

for i in self.lines:

if i.find(target)>=0:

print i

file1 = textfile('file1.txt')

file2 = textfile('file2.txt')

print "The number of textfles open",file1.ntfiles

print "Some info about them(name,lines,words):"

for myfile in [file1,file2]:

print myfile.name,myfile.nlines,myfile.nwords

file1.grep('example')

print "'''''",textfile.\_\_ntfiles

textfile.totfiles1()

textfile.totfiles2()

**Name: Yasir NL**

**Emp ID : 830**

**Date: 13/02/2017**

**CLASSES AND OBJECTS**

**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 behavior 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.The class statement creates a new class definition. The name of the class immediately follows the keyword class followed by a colon as follows:

Following is the example of a simple Python class:

#!/usr/bin/python

class Employee:

'Common base class for all employees'

empCount = 0

def \_\_init\_\_(self, name, salary):

self.name = name

self.salary = salary

Employee.empCount += 1

def displayCount(self):

print "Total Employee %d" % Employee.empCount

def displayEmployee(self):

print "Name : ", self.name, ", Salary: ", self.salary

"This would create first object of Employee class"

emp1 = Employee("Zara", 2000)

"This would create second object of Employee class"

emp2 = Employee("Manni", 5000)

emp1.displayEmployee()

emp2.displayEmployee()

print "Total Employee %d" % Employee.empCount

**output:**

Name : Zara ,Salary: 2000

Name : Manni ,Salary: 5000

Total Employee 2

**Program explaination**

The variable **empCount** is a class variable whose value is shared among all instances of a this class. This can be accessed as Employee.empCount from inside the class or outside the class.

The first method **\_\_init\_\_()** is a special method, which is called class constructor or initialization method that Python calls when you create a new

instance of this class.

You declare other class methods like normal functions with the exception that the first argument to each method is self. Python adds the self argument to the list for you; you do not need to include it when you call the methods.

Instead of using the normal statements to access attributes, you can use the following

functions:

**The getattr(obj, name[, default]) :** to access the attribute of object.

**The hasattr(obj,name) :** to check if an attribute exists or not.

**The setattr(obj,name,value) :** to set an attribute. If attribute does not exist, then it would be created.

**The delattr(obj, name) :** to delete an attribute.

Example:

**hasattr(emp1, 'age')** # Returns true if 'age' attribute exists

**getattr(emp1, 'age')** # Returns value of 'age' attribute

**setattr(emp1, 'age', 8)** # Set attribute 'age' at 8

**delattr(empl, 'age')** # Delete attribute 'age'

**Destroying Objects (Garbage Collection):**

Python deletes unneeded objects (built-in types or class instances) automatically to free the memory space. The process by which Python periodically reclaims blocks of memory that no longer are in use is termed Garbage Collection.

#!/usr/bin/python

class Point:

def \_\_init( self, x=0, y=0):

self.x = x

self.y = y

def \_\_del\_\_(self):

class\_name = self.\_\_class\_\_.\_\_name\_\_

print class\_name, "destroyed"

pt1 = Point()

pt2 = pt1

pt3 = pt1

print id(pt1), id(pt2), id(pt3) # prints the ids of the obejcts

del pt1

del pt2

del pt3

**output:**

3083401324 3083401324 3083401324

Point destroyed

**Class 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 SubClassName (ParentClass1[, ParentClass2, ...]):

'Optional class documentation string'

class\_suite

**Example**

#!/usr/bin/python

class Parent: # define parent class

parentAttr = 100

def \_\_init\_\_(self):

print "Calling parent constructor"

def parentMethod(self):

print 'Calling parent method'

def setAttr(self, attr):

Parent.parentAttr = attr

def getAttr(self):

print "Parent attribute :", Parent.parentAttr

class Child(Parent): # define child class

def \_\_init\_\_(self):

print "Calling child constructor"

def childMethod(self):

print 'Calling child method'

c = Child() # instance of child

c.childMethod() # child calls its method

c.parentMethod() # calls parent's method

c.setAttr(200) # again call parent's method

c.getAttr() # again call parent's method

**output:**

Calling child constructor

Calling child method

Calling parent method

Parent attribute : 200

Similar way, you can drive a class from multiple parent classes as follows:

class A: # define your class A

.....

class B: # define your calss B

.....

class C(A, B): # subclass of A and B

.....

**Overriding Methods**

You can always override your parent class methods. One reason for overriding parent's methods is because you may want special or different functionality in your subclass.

**Example**

#!/usr/bin/python

class Parent: # define parent class

def myMethod(self):

print 'Calling parent method'

class Child(Parent): # define child class

def myMethod(self):

print 'Calling child method'

c = Child() # instance of child

c.myMethod() # child calls overridden method

**output:**

Calling child method

**Base Overloading Methods**

**Sr. No.**  **Method, Description, and Sample Call**

1 \_\_init\_\_ ( self [,args...] )

Constructor (with any optional arguments)

Sample Call : obj = className(args)

2 \_\_del\_\_( self )

Destructor, deletes an object

Sample Call : del obj

3 \_\_repr\_\_( self )

Evaluatable string representation

Sample Call : repr(obj)

4 \_\_str\_\_( self )

Printable string representation

Sample Call : str(obj)

5 \_\_cmp\_\_ ( self, x )

Object comparison

Sample Call : cmp(obj, x)

**Overloading Operators**

Suppose you have created a Vector class to represent two-dimensional vectors, what

happens when you use the plus operator to add them? Most likely Python will yell at

you.

You could, however, define the \_\_add\_\_ method in your class to perform vector

addition and then the plus operator would behave as per expectation:

**Example**

#!/usr/bin/python

class Vector:

def \_\_init\_\_(self, a, b):

self.a = a

self.b = b

def \_\_str\_\_(self):

return 'Vector (%d, %d)' % (self.a, self.b)

def \_\_add\_\_(self,other):

return Vector(self.a + other.a, self.b + other.b)

v1 = Vector(2,10)

v2 = Vector(5,-2)

print v1 + v2

output:

Vector(7,8)

**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**

#!/usr/bin/python

class JustCounter:

\_\_secretCount = 0

def count(self):

self.\_\_secretCount += 1

print self.\_\_secretCount

counter = JustCounter()

counter.count()

counter.count()

print counter.\_\_secretCount

**output:**

1

2

Traceback (most recent call last):

File "test.py", line 12, 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:

#!/usr/bin/python

class JustCounter:

\_\_secretCount = 0

def count(self):

self.\_\_secretCount += 1

print self.\_\_secretCount

counter = JustCounter()

counter.count()

counter.count()

print counter.\_JustCounter\_\_secretCount

**output:**

1

2

2

**REGULAR EXPRESSIONS**

A regular expression is a special sequence of characters that helps you match or find other strings or sets of strings, using a specialized syntax held in a pattern. Regular expressions are widely used in UNIX world.

The module re provides full support for Perl-like regular expressions in Python. The re module raises the exception re.error if an error occurs while compiling or using a regular expression.

**Match Function**

This function attempts to match RE pattern to string with optional flags.

**syntax**

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

**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 usegroup(num) or groups() function of match object to get matched expression.

**Example:**

#!/usr/bin/python

import re

line = "Cats are smarter than dogs";

searchObj = re.search( r'(.\*) are (.\*?) .\*', line, re.M|re.I)

if searchObj:

print "searchObj.group() : ", searchObj.group()

print "searchObj.group(1) : ", searchObj.group(1)

print "searchObj.group(2) : ", searchObj.group(2)

else:

print "Nothing found!!"

**output:**

matchObj.group() :

Cats are smarter than dogs

matchObj.group(1) : Cats

matchObj.group(2) : smarter

**Matching Versus Searching**

Python offers two different primitive operations based on regular expressions: match checks for a match only at the beginning of the string, while search checks for a match anywhere in the string.

#!/usr/bin/python

import re

line = "Cats are smarter than dogs";

matchObj = re.match( r'dogs', line, re.M|re.I)

if matchObj:

print "match --> matchObj.group() : ", matchObj.group()

else:

print "No match!!"

searchObj = re.search( r'dogs', line, re.M|re.I)

if searchObj:

print "search --> searchObj.group() : ", searchObj.group()

else:

print "Nothing found!!"

**Output:**

No match!!

search --> matchObj.group() :

dogs

**Search and Replace**

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

**Syntax**

**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**

#!/usr/bin/python

import re

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

# Delete Python-style comments

num = re.sub(r'#.\*$', "", phone)

print "Phone Num : ", num

# Remove anything other than digits

num = re.sub(r'\D', "", phone)

print "Phone Num : ", num

**output:**

Phone Num : 2004-959-559

Phone Num : 2004959559

**Regular-Expression Patterns**

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

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

**Pattern Description**

^ Matches beginning of line.

$ Matches end of line.

. Matches any single character except newline. Using moption allows it to match newline as well.

[...] Matches any single character in brackets.

[^...] Matches any single character not in brackets

re\* Matches 0 or more occurrences of preceding expression.

re+ Matches 1 or more occurrence of preceding expression.

re? Matches 0 or 1 occurrence of preceding expression.

re{ n} Matches exactly expression.

re{ n,} Matches n or more occurrences of preceding expression.

re{n,m} Matches at least n and at most m occurrences of preceding

expression.

a| b Matches either a or b.

(re) Groups regular expressions and remembers matched text.

(?imx) Temporarily toggles on i, m, or x options within a regular expression. If in parentheses, only that area is affected.

(?-imx) Temporarily toggles off i, m, or x options within a regular expression. If in parentheses, only that area is affected.

(?: re) Groups regular expressions without remembering matchedtext.

(?imx: re) Temporarily toggles on i, m, or x options within parentheses.

(?-imx: re) Temporarily toggles off i, m, or x options within parentheses.

(?#...) Comment.

(?= re) Specifies position using a pattern. Doesn't have a range.

(?! re) Specifies position using pattern negation. Does not have a range.

(?> re) Matches independent pattern without backtracking.

\w Matches word characters.

\W Matches non-word characters.

\s Matches whitespace. Equivalent to [\t\n\r\f].

\S Matches non-whitespace.

\d Matches digits. Equivalent to [0-9].

\D Matches non-digits.

\A Matches beginning of string.

\Z Matches end of string. If a newline exists, it matches just before newline.

\z Matches end of string.

\G Matches point where last match finished.

\b Matches word boundaries when outside brackets. Matches backspace (0x08) when inside brackets.

\B Matches non-word boundaries.

\n, \t, etc. Matches newlines, carriage returns, tabs, etc.

\1...\9 Matches nth grouped subexpression.

\10 Matches nth grouped subexpression if it matched already.

MULTITHREADING

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

1. 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.

2. Threads sometimes called light-weight processes and they do not require much memory overhead; they care cheaper than processes.

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

1.It can be pre-empted (interrupted)

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

**Starting a New Thread**

**syntax**

**thread.start\_new\_thread ( function, args[, kwargs] )**

This method call enables a fast and efficient way to create new threads in both Linux and Windows.

The method call returns immediately and the child thread starts and calls function with the passed list of agrs. When function returns, the thread terminates.

Here, args is a tuple of arguments; use an empty tuple to call function without passing any arguments. kwargs is an optional dictionary of keyword arguments.

**Example**

#!/usr/bin/python

import thread

import time

# Define a function for the thread

def print\_time( threadName, delay):

count = 0

while count < 5:

time.sleep(delay)

count += 1

print "%s: %s" % ( threadName, time.ctime(time.time()) )

# Create two threads as follows

try:

thread.start\_new\_thread( print\_time, ("Thread-1", 2, ) )

thread.start\_new\_thread( print\_time, ("Thread-2", 4, ) )

except:

print "Error: unable to start thread"

while 1:

pass

**output:**

Thread-1: Thu Jan 22 15:42:17 2009

Thread-1: Thu Jan 22 15:42:19 2009

Thread-2: Thu Jan 22 15:42:19 2009

Thread-1: Thu Jan 22 15:42:21 2009

Thread-2: Thu Jan 22 15:42:23 2009

Thread-1: Thu Jan 22 15:42:23 2009

Thread-1: Thu Jan 22 15:42:25 2009

Thread-2: Thu Jan 22 15:42:27 2009

Thread-2: Thu Jan 22 15:42:31 2009

Thread-2: Thu Jan 22 15:42:35 2009

**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**

#!/usr/bin/python

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 "%s: %s" % (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-1

Starting Thread-2

Exiting Main Thread

Thread-1: Thu Mar 21 09:10:03 2013

Thread-1: Thu Mar 21 09:10:04 2013

Thread-2: Thu Mar 21 09:10:04 2013

Thread-1: Thu Mar 21 09:10:05 2013

Thread-1: Thu Mar 21 09:10:06 2013

Thread-2: Thu Mar 21 09:10:06 2013

Thread-1: Thu Mar 21 09:10:07 2013

Exiting Thread-1

Thread-2: Thu Mar 21 09:10:08 2013

Thread-2: Thu Mar 21 09:10:10 2013

Thread-2: Thu Mar 21 09:10:12 2013

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 theLock() 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 wait 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**

#!/usr/bin/python

import threading

import time

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

# Get lock to synchronize threads

threadLock.acquire()

print\_time(self.name, self.counter, 3)

# Free lock to release next thread

threadLock.release()

def print\_time(threadName, delay, counter):

while counter:

time.sleep(delay)

print "%s: %s" % (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()

# Add threads to thread list

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-1

Starting Thread-2

Thread-1: Thu Mar 21 09:11:28 2013

Thread-1: Thu Mar 21 09:11:29 2013

Thread-1: Thu Mar 21 09:11:30 2013

Thread-2: Thu Mar 21 09:11:32 2013

Thread-2: Thu Mar 21 09:11:34 2013

Thread-2: Thu Mar 21 09:11:36 2013

Exiting Main Thread

**Multithreaded Priority Queue**

The Queue module allows you to create a new queue object that can hold a specific number of items. There are following methods to control the Queue:

**get():** The get() removes and returns an item from the queue.

**put():** The put adds item to a queue.

**qsize():** The qsize() returns the number of items that are currently in the queue.

**empty():** The empty( ) returns True if queue is empty; otherwise, False.

**full():** The full() returns True if queue is full; otherwise, False.

**Example**

#!/usr/bin/python

import Queue

import threading

import time

exitFlag = 0

class myThread (threading.Thread):

def \_\_init\_\_(self, threadID, name, q):

threading.Thread.\_\_init\_\_(self)

self.threadID = threadID

self.name = name

self.q = q

def run(self):

print "Starting " + self.name

process\_data(self.name, self.q)

print "Exiting " + self.name

def process\_data(threadName, q):

while not exitFlag:

queueLock.acquire()

if not workQueue.empty():

data = q.get()

queueLock.release()

print "%s processing %s" % (threadName, data)

else:

queueLock.release()

time.sleep(1)

threadList = ["Thread-1", "Thread-2", "Thread-3"]

nameList = ["One", "Two", "Three", "Four", "Five"]

queueLock = threading.Lock()

workQueue = Queue.Queue(10)

threads = []

threadID = 1

# Create new threads

for tName in threadList:

thread = myThread(threadID, tName, workQueue)

thread.start()

threads.append(thread)

threadID += 1

# Fill the queue

queueLock.acquire()

for word in nameList:

workQueue.put(word)

queueLock.release()

# Wait for queue to empty

while not workQueue.empty():

pass

# Notify threads it's time to exit

exitFlag = 1

# Wait for all threads to complete

for t in threads:

t.join()

print "Exiting Main Thread"

**output:**

Starting Thread-1

Starting Thread-2

Starting Thread-3

Thread-1 processing One

Thread-2 processing Two

Thread-3 processing Three

Thread-1 processing Four

Thread-2 processing Five

Exiting Thread-3

Exiting Thread-1

Exiting Thread-2

Exiting Main Thread

Date: 15/02/2017

Name: Meena.P

ID: 847

**CHILD PROCESS CREATION**

The system function call fork() creates a copy of the process, which has called it. This copy runs as a child process of the calling process. The child process gets the data and the code of the parent process. The child process receives a process number (PID, Process IDentifier) of its own from the operating system.

The child process runs as an independent instance, this means independent of a parent process. With the return value of fork() we can decide in which process we are: 0 means that we are in the child process while a positive return value means that we are in the parent process. A negative return value means that an error occurred while trying to fork.

os.fork() is used to start another process in parallel to the current one.

os.fork() creates a copy of the previous Python session and opens it in parallel.

os.fork() returns the id of the new process.

**Example**:

"forks child processes until you type 'q'"

import os

def child():

print('Hello from child', os.getpid())

os.\_exit(0) # else goes back to parent loop

def parent():

while True:

if input() == 'q': break

parent()

newpid = os.fork()

if newpid == 0:

child()

else:

print('Hello from parent', os.getpid(), newpid)

**Output:**

Hello from parent 2057 2062

Hello from child 2062

Hello from parent 2057 2068

Hello from child 2068

Hello from parent 2057 2069

Hello from child 2069

Hello from parent 2057 2070

Hello from child 2070

q

**Subprocess Module:**

The subprocess module provides a consistent interface to creating and working with additional processes. It offers a higher-level interface than some of the other available modules, and is intended to replace functions such as os.system(), os.spawn\*(), os.popen\*(), popen2.\*() and commands.\*(). To make it easier to compare subprocess with those other modules, many of the examples here re-create the ones used for os and popen.

The Subprocess module defines one class, Popen and a few wrapper functions that use that class.

The constructor for Popen takes arguments to set up the new process so the parent can communicate with it via pipes.

It provides all of the functionality of the other modules and functions it replaces, and more.

The API is consistent for all uses, and many of the extra steps of overhead needed (such as closing extra file descriptors and ensuring the pipes are closed) are “built in” instead of being handled by the application code separately.

## Running External Command

To run an external command without interacting with it, such as one would do with os.system(), Use the call() function.

**Example**

import subprocess

# Simple command

subprocess.call(['ls', '-1'], shell=True)

**output:**

$ python subprocess\_os\_system.py

\_\_init\_\_.py

index.rst

interaction.py

repeater.py

signal\_child.py

signal\_parent.py

subprocess\_check\_call.py

subprocess\_check\_output.py

subprocess\_check\_output\_error.py

subprocess\_check\_output\_error\_trap\_output.py

subprocess\_os\_system.py

subprocess\_pipes.py

subprocess\_popen2.py

subprocess\_popen3.py

subprocess\_popen4.py

subprocess\_popen\_read.py

subprocess\_popen\_write.py

subprocess\_shell\_variables.py

subprocess\_signal\_parent\_shell.py

subprocess\_signal\_setsid.py

Setting the shell argument to a true value causes subprocess to spawn an intermediate shell process, and tell it to run the command. The default is to run the command directly.

**Example:**

import subprocess

# Command with shell expansion

subprocess.call('echo $HOME', shell=True)

**Output:**

$ python subprocess\_shell\_variables.py

/Users/dhellmann

**Error Handling**

The return value from call() is the exit code of the program. The caller is responsible for interpreting it to detect errors. The check\_call() function works like call() except that the exit code is checked, and if it indicates an error happened then a CalledProcessError exception is raised.

**Example**

import subprocess

subprocess.check\_call(['false'])

The false command always exits with a non-zero status code, which check\_call() interprets as an error.

**Ouput:**

$ python subprocess\_check\_call.py

Traceback (most recent call last):

File "subprocess\_check\_call.py", line 11, in <module>

subprocess.check\_call(['false'])

File "/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.

7/subprocess.py", line 511, in check\_call

raise CalledProcessError(retcode, cmd)

subprocess.CalledProcessError: Command '['false']' returned non-zero e

xit status 1

**Capturing output**

The standard input and output channels for the process started by call() are bound to the parent’s input and output. That means the calling programm cannot capture the output of the command. Use check\_output() to capture the output for later processing.

**Example**

import subprocess

output = subprocess.check\_output(['ls', '-1'])

print 'Have %d bytes in output' % len(output)

print output

The ls -1 command runs successfully, so the text it prints to standard output is captured and returned.

**Output:**

$ python subprocess\_check\_output.py

Have 462 bytes in output

\_\_init\_\_.py

index.rst

interaction.py

repeater.py

signal\_child.py

signal\_parent.py

subprocess\_check\_call.py

subprocess\_check\_output.py

subprocess\_check\_output\_error.py

subprocess\_check\_output\_error\_trap\_output.py

subprocess\_os\_system.py

subprocess\_pipes.py

subprocess\_popen2.py

subprocess\_popen3.py

subprocess\_popen4.py

subprocess\_popen\_read.py

subprocess\_popen\_write.py

subprocess\_shell\_variables.py

subprocess\_signal\_parent\_shell.py

subprocess\_signal\_setsid.py