PYTHON CLASS NOTES

Taken from previous set and modified/updated/added by Sandeep Reddy G and Shravya and Shravani.

Introduction:

- 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 and platform independent
- 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):

• The extension of script file is **.py**,but it is not mandatory,even script will execute without the extension

```
# 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

- Multiline comments are done by using (“”” or ''')

  Eg:”””comment””” or '''comment'''

- 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.

```
>>val1, val2, val3 = 1, 2, "abc"
>>print val1,val2,val3
output: 1 2  abc
```

- 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	Exec	Not
Assert	Finally	Or
Break	For	Pass
Class	From	Print
Continue	Global	Raise
Def	If	Return
Del	Import	Try
Elif	In	While
Else	Is	With
Except	Lambda	yield

- 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

```
>>val=5
>>print val #prints 5
>>name='votary'
>>print "Your name is", name
>>print val,      # 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:
        #statements
```

• Sometimes you will need to specify an empty block of code (like {} in C/Java)

```
if (cndition):
# Not implemented yet (or nothing)
    pass #here pass means nothing executes like ; in C
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 {}

```
f (product=="game" and type=="pirate memory"
and age >= 4 and age <= 8):
        print "I'll take it!"
```

Basic Datatypes :

Python only has a 6 standard data types of data

- Numbers
- Strings (character text)
- Lists
- Dictionary
- Tuple
- Sets

Numbers :

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

- Python has 4 types of numbers
- Booleans
- Integers
- Floating point
- Complex (imaginary numbers)

Constants in python

Numeric character String

Integer Real

Binary octal Decimal Hexadecimal

Integers :

- Signed values of arbitrary size

```
>>val1 = 37
>>val2 = -2993929937277166273771284818122412 31
>>val3 = 0x7fa8              # Hexadecimal
>>val4 = 0o253               # Octal
>>val5 = 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

```
>>> val2
-2993929937277166273771284818122412 31L
```

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

```
Var1,var2,var3=1,2,3
del  var1 # It will delete the reference
del var2,var3,..... #For multiple variables
```

Floating point (float) :

- Use a decimal or exponential notation

```
>>num1 = 37.45
>>num2 = 4e5
>>num3 = -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]

• Slicing/substrings : s[start:end]

```
string = "Hello world"
print string[0]      # it prints  'H'
print string[4]      # it prints  'o'
print string[-1]     # it prints  'd'
print string[:5]     # it prints  "Hello"
print string[6:]     # it prints  "world"
print string[3:8]    # it prints  "lo wo"
print string[-5:]    #it prints  "world"
```

• Strings are "immutable" (read only)

• Once created, the value can't be changed

```
>>> string = "Hello World"
>>> string[1] = 'a'  #error since immutable
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.

String Special Operators:

1. Concatinatin(+)
2. Repetation(*)
3. Slice(:)
4. Rawstring(r or R)
5. Formating

```
String=”hello”
str=”welcome”
print string+”hyd” #prints hellohyd
print string*2       #prints hellohello
print string[1:4]    #prints ell
print string[::2]    #prints hlo
print “string is:%s and str is:%s”%(string,str)    #prints string is:hello and str
```

```
is:welcome
```

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

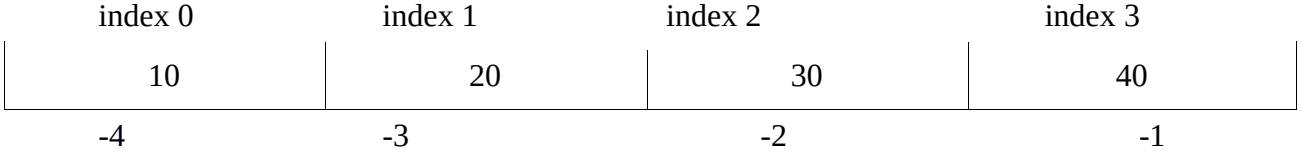
  >> names[1] = "Joliet Jake"

- Concatenation (+)

```
>>string = "Hello" + "World"
>>print string
Helloworld
>>str = "Say " + string.
>>print str
sayHelloworld
```

Eg:

list=[10,20,30,40]



list[1:3]=20,30
list[:3]=10,20
list[2:]=30
list[-4:-2]=10,20

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 i.e tuples are immutable

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
tuple[0]='asdf'                #error
```

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 }

KEY VALUE

- Getting values: Just use the key names

```
>>> print dict['name'],dict['shares']
    GOOG 100
>>> dict['price']
   490.10
```

- Adding/modifying values : Assign to key names

```
>>> dict['shares'] = 75
>>> dict['date'] = '6/6/2007'
```

- Deleting a value

```
>>> del dict['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

```
>>num1 = '37'
>>num2 = '42'                    # Strings
>>res = num1 + num2                    # res = '3742'   (concatenation)
>>num1 = 37
>>num2 = 42
>>res = num1 + num2                    # res=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)

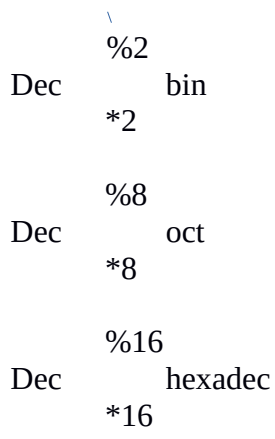
	tuples.
frozenset(s)	Converts s to a frozen set.
chr(x)	Converts an integer to a character.
unichr(x)	Converts an integer to a Unicode character.
oct(x)	Converts an integer to an octal string.
hex(x)	Converts an integer to a hexadecimal string.

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"
```

Table :

	Binary	Octal	Decimal	Hexadecimal
Binary	X	YES	Yes	Yes
octal	Yes	X	Yes	Yes
Decimal	Yes	Yes	X	Yes
hexadecimal	Yes	Yes	Yes	X



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
```

V1	V2	Result=v1/v2
Int	Int	Int
Int	Float	Float
Float	Int	Float
Float	Float	Float

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 O/P:

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
```

Bitwise Operators :

1. Bitwise AND(&)
2. Bitwise OR(|)
3. Bitwise EX-OR(^)
4. Leftshift(<<)
5. Right shift(>>)
6. Compliment(~)

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

```
>>val1 = True
>>val2 = False
```

- Evaluated as integers with value 1,0

```
>>val3 = 4 + True                  # val3 = 5
>>val4 = False
>>if val4 == 0:
      print "val4 is False"
```

- Although doing that in practice would be odd
- Be aware that floating point numbers are inexact when representing decimal values.

```
>>> val1 = 2.1 + 4.2
>>> val1 == 6.3
      False
>>> val1
    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
num1 = 10
num2 = 20
list1 = [1, 2, 3, 4, 5 ];
if ( nm1 in list1 ):
```

```
    print "num1 is available in the given list"
else:
    print " num1 is not available in the given list"
if ( num2 not in list ):
    print " num2 is not available in the given list"
else:
    print " num2 is available in the given list"
num1 = 2
if ( num1 in list ):
    print "num1 is available in the given list"
else:
    print "num1 is not available in the given list"
```

output:-

num1 is not available in the given list
num2 is not available in the given list
num1 is available in the given list

Identity Operators:

Identity operators compare the memory locations of two objects.

1. Id(obj): it gives the memory location of variable
2. is : it compares the memory location
3. is not

Example

```
#!/usr/bin/python
num1 = 20
num2 = 20
if ( num1 is num2 ):
    print " num1 and num2 have same identity"
else:
    print " num1 and num2 do not have same identity"
if ( id(num1) == id(num2) ):
   print " num1 and num2 have same identity"
else:
    print " num1 and num2 do not have same identity"
num2 = 30
if ( num1 is num2 ):
```

```
    print " num1 and num2 have same identity"
else:
    print " num1 and num2 do not have same identity"
if ( num1 is not num2 ):
    print " num1 and num2 do not have same identity"
else:
    print " num1 and num2 have same identity"
```

output:

num1 and num2 have same identity
num1 and num2 have same identity
num1 and num2 do not have same identity
num1 and num2 do not have same identity

Ternery Operators:

Syntax:
[on true] if (expression) else [on flase]
Example:

```
v1,v2=50,25
small=v1 if v1<v2 else v2
print small
```

Output:
25

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 in not in	Identity operators Membership operators
not or and	Logical operators

Example:

```
#!/usr/bin/python
val1 = 20
val2 = 10
val3 = 15
val4 = 5
val5 = 0
val6 = (val1 + val2) * val3 / val4                    #( 30 * 15 ) / 5 = 90
print "Value of (val1 + val2) * val3 / val4 is ",  val6
val6 = ((val1 + val2) * val3) /val4                   # (30 * 15 ) / 5 = 90
print "Value of ((val1 + val2) * val3) /val4  is ", val6
val6=(val1 + val2) * (val3 /val4 )       # (30) * (15/5) = 90

print "Value of (val1 + val2) * (val3 /val4 ) is ", val6
val6 = val1 + (val2 * val3) / val4;                   #20 + (150/5) = 90
print "Value of val1 + (val2 * val3) / val4 is ", val6
```

Conditional Statements:

If Else Statment:

if block will execute whenever if conditaion is True,otherwise else block will execute

syntax:

```
if(condiation):
        #statements
else:
        #statments
```

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!

Elif statement:

The elif statement allows you to check multiple expressions for TRUE and execute a block of code as soon as one of the conditions evaluates to TRUE.Similar to the else, the elif statement is optional. However, unlike else, for which there can be at most one statement,there can be an arbitrary number of elif statements following an if.

Syntax:

```
if condition1:
        #statement(s)
elif condition2:
        #statement(s)
elif condition3:
        #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!"
```

Output:

```
3 - Got a true expression value
100
Good bye!
```

Loops

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.

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 condition:
    #statements
```

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

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

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

Example:

```
#!/usr/bin/python

count = 0
while (count < 4):
   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

Good Bye

While else:

Syntax:

```
while condition:
        #statements
else:
        #statements
```

Here the else block will executes whenever the while condition failes.

Example:

```
val=1
while(val<5):
      print val,
      val+=1
else:
    print “else block”
```

Output:

1 2 3 4

else block

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>:

#statements

else:

#statements

Example:

```
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"
```

Output:

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:
                #statements
else:
        '''statements after loop\n'''
while condition1:
        while condition2:
                #statements
else:
        '''statements after 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.

Example:

```
Val=1
list1=[]
while(val<20):
```

```
    if(val%2!=0)
        continue
    list1.append(val)
    if(val>10):
        break
print list1
```

Output:
[2,4,6,8]

FUNCTIONS

Functions are used for reusable 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.

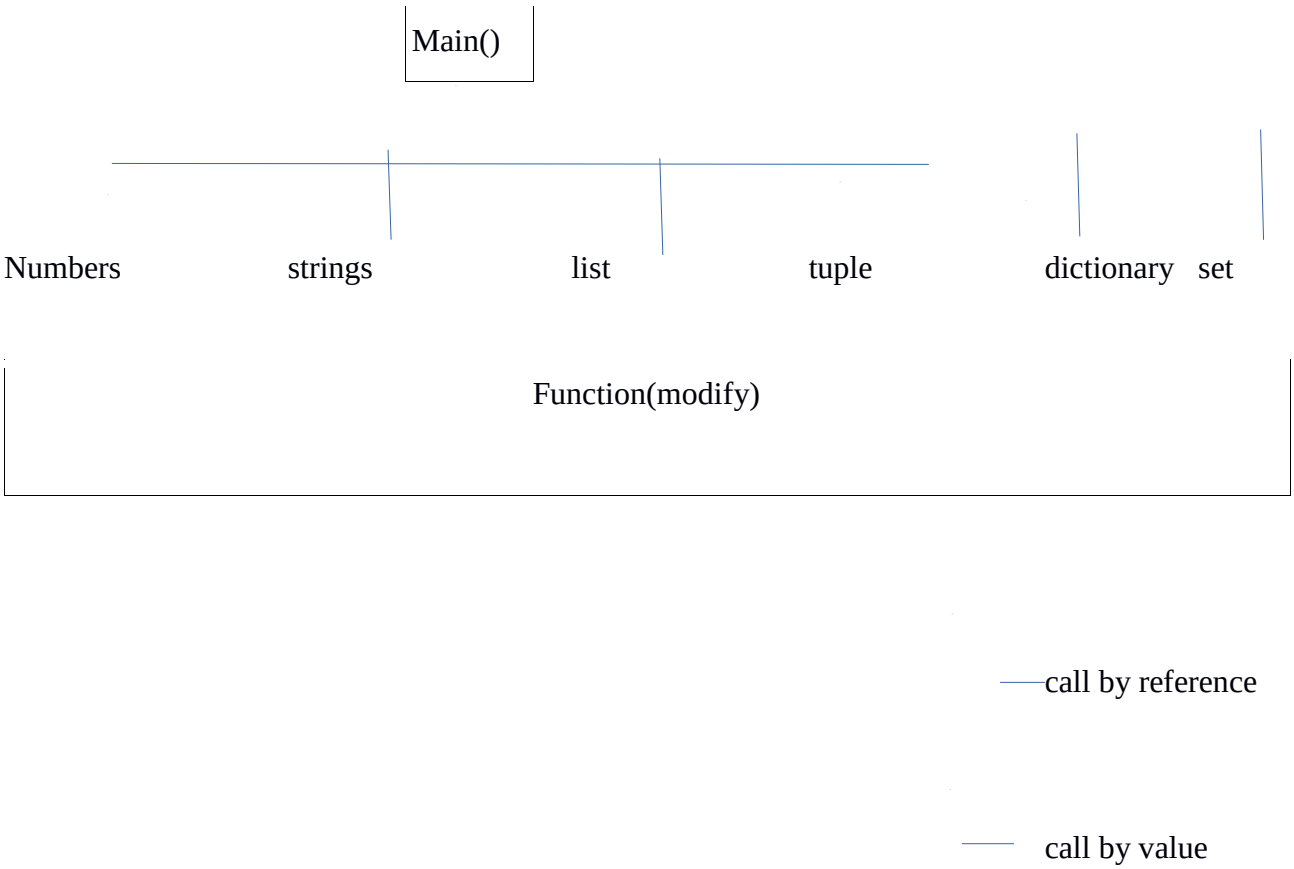
Types of Function arguments in Python:

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:

Assigning default values should start from right side

```
def fun(num,val=2,val2) : #this will give you an error
def say(msg, tms = 1):
        print msg,
        print tms
        return
```

Output:

if we pass 20

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
```

Output:

arg1 10

arg2 20

arg3 30

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 variable:

scope means the availability of variable

```
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
        # accesing vars will give you an error

fun( )  #function calling
print 'varg after function call',varg
```

output:

20

varg after function call 50

Iterators:

1. Iterators gives the values one after other by iterating over data which is iterable.
2. 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

Mathematical Functions:Mathematical Functions:

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

val --is numeric Expressions

```
#!/usr/bin/python
print "abs(-45) : ", abs(-45)

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

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

Output:

abs(-45) : 45

abs(100.12) : 100.12

abs(119L) : 119

ceil(val):

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.

val -- is numeric Expressions

```
#!/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)
```

Output:

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(val1,val2):

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 .

val1,val2 -- are numeric Expressions

```
#!/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)
```

Output:

cmp(80, 100) : -1

cmp(180, 100) : 1

cmp(-80, 100) : -1

cmp(80, -100) : 1

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

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

val -- This is a numeric expression.

```
#!/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)
```

Output:

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

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

val-- This is a numeric value.

```
#!/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)
```

Output:

```
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(val):

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.

val -- This is a numeric expression.

```
#!/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)
```

Output:

```
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(val):

The method **log()** returns natural logarithm of x, for x > 0.This function is not accessible directly, so we need to import math module.

val -- This is a numeric expression.

```
#!/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)
```

Output:

```
math.log(100.12) :  4.60636946656
math.log(100.72) :  4.61234438974
math.log(119L) :  4.77912349311
math.log(math.pi) :  1.14472988585
```

log10(val):

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.

val -- This is a numeric expression.

```
#!/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)
```

Output:

```
math.log10(100.12) :  2.00052084094
math.log10(100.72) :  2.0031157171
math.log10(119L) :  2.07554696139
math.log10(math.pi) :  0.497149872694
```

max(val1,val2,val3 ....):

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

val1 -- This is a numeric expression.

val2-- This is also a numeric expression.

val3 -- This is also a numeric expression.

```
#!/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)
```

Output:

```
max(80, 100, 1000) :  1000
max(-20, 100, 400) :  400
max(-80, -20, -10) :  -10
max(0, 100, -400) :  100
```

min(val1,val2,val3 ....):

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

val1 -- This is a numeric expression.

val2-- This is also a numeric expression.

val3 -- This is also a numeric expression.

```
#!/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)
```

Output:

```
min(80, 100, 1000) :  80
min(-20, 100, 400) :  -20
min(-80, -20, -10) :  -80
min(0, 100, -400) :  -400
```

modf(val):

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.

val -- This is a numeric expression.

```
#!/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)
```

Output:

```
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.

```
#!/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)
```

Output:

```
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(val,[n]):

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

val -- This is a numeric expression..

n -- This is also a numeric expression.

```
#!/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)
```

Output:

round(80.23456, 2) : 80.23

round(100.000056, 3) : 100.0

round(-100.000056, 3) : -100.0

sqrt(val):

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

val -- This is a numeric expression.

```
#!/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)
```

Output:

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.

```
#!/usr/bin/python
var1 = 'Hello World!'

var2 = "Python Programming"

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

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

Output:

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.

```
#!/usr/bin/python
var1 = 'Hello World!'
print "Updated String :- ", var1[:6] + 'Python'
```

Output:

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 copies of the same string	a*2 will give-HelloHello
[]	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 given string	H in a will give 1
Not in	Membership - Returns true if a character does not exist in the given string	M not in a will give 1
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	

String Formatting Operator:

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

```
#!/usr/bin/python
print "My name is %s and weight is %d kg!" % ('Zara', 21)
```

Output:

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	Conversion
%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.

```
#!/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;
```

Output:

```
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.

```
#!/usr/bin/python
print u'Hello, world!'
```

Output:

```
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

```
#!/usr/bin/python
str = "this is string example....wow!!!";
print "str.capitalize() : ", str.capitalize()
```

Output:

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.

```
#!/usr/bin/python
str = "this is string example....wow!!!";
print "str.center(40, 'a') : ", str.center(40, 'a')
```

Output:

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.

```
#!/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)
```

Output:

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.

```
#!/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')
```

Output:

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.

```
#!/usr/bin/python
str = "this is string example....wow!!!";
print "Encoded String: " + str.encode('base64','strict')
```

Output:

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.

```
#!/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);
```

Output:

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.

```
#!/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);
```

Output:

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.

```
#!/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);
```

Output:

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.

```
#!/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);
```

Output:

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.

```
#!/usr/bin/python
str = "this2009";        # No space in this string
print str.isalnum();
str = "this is string example....wow!!!";
print str.isalnum();
```

Output:

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.

```
#!/usr/bin/python
str = "this";     # No space & digit in this string
print str.isalpha();
str = "this is string example....wow!!!";
print str.isalpha();
```

Output:

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.

```
#!/usr/bin/python
str = "123456";          # Only digit in this string

int str.isdigit();

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

int str.isdigit();
```

Output:

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.

```
#!/usr/bin/python
str = "THIS is string example....wow!!!";

print str.islower();

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

print str.islower();
```

Output:

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.

```
#!/usr/bin/python
str = u"this2009";

print str.isnumeric();

str = u"23443434";

print str.isnumeric();
```

Output:

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.

```
#!/usr/bin/python
str = " ";

print str.isspace();

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

print str.isspace();
```

Output:

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.

```
#!/usr/bin/python
str = "This Is String Example...Wow!!!";
print str.istitle();
str = "This is string example....wow!!!";
print str.istitle();
```

Output:

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.

```
#!/usr/bin/python
str = "THIS IS STRING EXAMPLE....WOW!!!";
print str.isupper();
str = "THIS is string example....wow!!!";
print str.isupper();
```

Output:

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.

```
#!/usr/bin/python
str = "-";
```

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

Output:

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.

```
#!/usr/bin/python
str = "this is string example....wow!!!";
print "Length of the string: ", len(str);
```

Output:

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).

```
#!/usr/bin/python
str = "this is string example....wow!!!";
print str.ljust(50, '0');
```

Output:

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.

```
#!/usr/bin/python
str = "THIS IS STRING EXAMPLE....WOW!!!";
print str.lower();
```

Output:

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).

```
#!/usr/bin/python
str = " this is string example....wow!!! ";
print str.lstrip();
str = "88888888this is string example....wow!!!8888888";
print str.lstrip('8');
```

Output:

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.

```
#!/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);
```

Output:

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.

```
#!/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);
```

Output:

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.

```
#!/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);
```

Output:

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.

```
#!/usr/bin/python
str = "this is string example....wow!!! this is really string";
print str.replace("is", "was");
print str.replace("is", "was", 3);
```

Output:

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

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

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.

```
#!/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);
```

Output:

```
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.

```
#!/usr/bin/python
str1 = "this is string example....wow!!!";
```

```
str2 = "is";
print str1.rindex(str2);
print str1.index(str2);
```

Output:

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).

```
#!/usr/bin/python
str = "this is string example....wow!!!";
print str.rjust(50, '0');
```

Output:

00000000000000000this 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).

```
#!/usr/bin/python
str = " this is string example....wow!!! ";
print str.rstrip();
```

```
str = "88888888this is string example....wow!!!8888888";
print str.rstrip('8');
```

Output:

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.

```
#!/usr/bin/python
str = "Line1-abcdef \nLine2-abc \nLine4-abcd";
print str.split( );
print str.split(' ', 1 );
```

Output:

['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.

```
#!/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 );
```

Output:

['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.

```
#!/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 );
```

Output:

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.

```
#!/usr/bin/python
str = "0000000this is string example....wow!!!0000000";
print str.strip( '0' );
```

Output:

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.

```
#!/usr/bin/python
str = "this is string example....wow!!!";
print str.swapcase();
str = "THIS IS STRING EXAMPLE....WOW!!!";
print str.swapcase();
```

Output:

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.

```
#!/usr/bin/python
str = "this is string example....wow!!!";
print str.title();
```

Output:

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.

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);
```

Output:

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');
```

Output:

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.

```
#!/usr/bin/python
str = "this is string example....wow!!!";
print "str.capitalize() : ", str.upper()
```

Output:

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.

```
#!/usr/bin/python
r = "this is string example....wow!!!";
print str.zfill(40);
print str.zfill(50);
```

Output:

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

0000000000000000this 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.

```
#!/usr/bin/python
str = u"this2009";
print str.isdecimal();
str = u"23443434";
print str.isdecimal();
```

Output:

False

True

LISTS

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.

```
#!/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]
```

Output:

```
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.

```
#!/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];
```

Output:

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.

```
#!/usr/bin/python
list1 = ['physics', 'chemistry', 1997, 2000];
print list1;
del list1[2];
print "After deleting value at index 2 : "
print list1;
```

Output:

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

After deleting value at index 2 :

['physics', 'chemistry', 2000]

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.

```
#!/usr/bin/python
list1, list2 = [123, 'xyz'], [456, 'abc']
print cmp(list1, list2);
print cmp(list2, list1);
list3 = list2 + [786];
print cmp(list2, list3)
```

Output:

-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.

```
#!/usr/bin/python
list1, list2 = [123, 'xyz', 'zara'], [456, 'abc']
print "First list length : ", len(list1);
print "Second list length : ", len(list2);
```

Output:

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.

```
#!/usr/bin/python
list1, list2 = [123, 'xyz', 'zara', 'abc'], [456, 700, 200]
print "Max value element : ", max(list1);
print "Max value element : ", max(list2);
```

Output:

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)

```
#!/usr/bin/python
list1 = [123, -5, 1010, 5614,254,[456, 700, 200]]
print "Max value element : ", max(list1)
```

Output:

Max value element : [456, 700, 200]

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

```
#!/usr/bin/python
list1 = [123, 'xyz', 'zar', 'abc', [456, 700, 200]]
print "Max value element : ", max(list1);
```

Output:

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.

```
#!/usr/bin/python
```

```
list1, list2 = [123, 'xyz', 'zara', 'abc'], [456, 700, 200]
print "min value element : ", min(list1);
print "min value element : ", min(list2);
```

Output:

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.

```
#!/usr/bin/python
aList = [123, 'xyz', 'zara', 'abc'];
aList.append( 2009 );
print "Updated List : ", aList;
```

Output:

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

ist.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.

```
#!/usr/bin/python
aList = [123, 'xyz', 'zara', 'abc', 123];
print "Count for 123 : ", aList.count(123);
print "Count for zara : ", aList.count('zara');
```

Output:

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.

```
#!/usr/bin/python
aList = [123, 'xyz', 'zara', 'abc', 123];
bList = [2009, 'manni'];
aList.extend(bList)
print "Extended List : ", aList ;
```

Output:

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

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.

```
#!/usr/bin/python
aList = [123, 'xyz', 'zara', 'abc'];
print "Index for xyz : ", aList.index( 'xyz' ) ;
print "Index for zara : ", aList.index( 'zara' ) ;
```

Output:

Index for xyz : 1

Index for zara : 2

ist.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.

```
#!/usr/bin/python
aList = [123, 'xyz', 'zara', 'abc']
aList.insert( 3, 2009)
print "Final List : ", aList
```

Output:

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.

```
#!/usr/bin/python
aList = [123, 'xyz', 'zara', 'abc'];
print "A List : ", aList.pop();
print "B List : ", aList.pop(2);
```

Output:

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.

```
#!/usr/bin/python
aList = [123, 'xyz', 'zara', 'abc', 'xyz'];
aList.remove('xyz');
print "List : ", aList;
aList.remove('abc');
print "List : ", aList;
```

Output:

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.

```
#!/usr/bin/python
aList = [123, 'xyz', 'zara', 'abc', 'xyz'];
aList.reverse();
print "List : ", aList;
```

Output:

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.

```
#!/usr/bin/python3
list1 = ['physics', 'Biology', 'chemistry', 'maths']
list1.sort()
print ("list now : ", list1)
```

Output:

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

Create a string from a list:

1. join will joines the list elements with str

```
#!/usr/bin/python
```

```
str=''
letters=['v','o','t','a','r','y']
word=str.join(letters)
print word
```

Output:

votary

List Processing- Sort

Selection 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()
```

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()
```

TUPLES

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.

Examples:

```
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.

```
#!/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]
```

Output:

```
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.

```
#!/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;
```

Output:

```
(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.

```
#!/usr/bin/python
tup = ('physics', 'chemistry', 1997, 2000);
```

```
print tup;
del tup;
print "After deleting tup : "
print tup;
```

Output:

('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.

```
#!/usr/bin/python
print 'abc', -4.24e93, 18+6.6j, 'xyz';
x, y = 1, 2;
print "Value of x , y : ", x,y;
```

Output:

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.

```
#!/usr/bin/python
tuple1, tuple2 = (123, 'xyz'), (456, 'abc')
print cmp(tuple1, tuple2);
print cmp(tuple2, tuple1);
tuple3 = tuple2 + (786,);
print cmp(tuple2, tuple3)
```

Output:

-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.

```
#!/usr/bin/python
tuple1, tuple2 = (123, 'xyz', 'zara'), (456, 'abc')
print "First tuple length : ", len(tuple1);
print "Second tuple length : ", len(tuple2);
```

Output:
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.

```
#!/usr/bin/python
tuple1, tuple2 = (123, 'xyz', 'zara', 'abc'), (456, 700, 200)
print "Max value element : ", max(tuple1);
print "Max value element : ", max(tuple2);
```

Output:

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.

```
#!/usr/bin/python
tuple1, tuple2 = (123, 'xyz', 'zara', 'abc'), (456, 700, 200)
print "min value element : ", min(tuple1);
print "min value element : ", min(tuple2);
```

Output:

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.

```
#!/usr/bin/python
aList = (123, 'xyz', 'zara', 'abc');
aTuple = tuple(aList)
print "Tuple elements : ", aTuple
```

Output:

Tuple elements :

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

DICTIONARY

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.

```
#!/usr/bin/python
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'};
```

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

Output:

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:

```
#!/usr/bin/python
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'};
print "dict['Alice']: ", dict['Alice'];
```

Output:

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:

```
#!/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'];
```

Output:

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.

```
#!/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.

```
#!/usr/bin/python
dict = {'Name': 'Zara', 'Age': 7, 'Name': 'Manni'};
print "dict['Name']: ", dict['Name'];
```

Output:

```
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.

```
#!/usr/bin/python
dict = {['Name']: 'Zara', 'Age': 7};
print "dict['Name']: ", dict['Name'];
```

Output:

```
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.

```
#!/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)
```

Output:

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.

```
#!/usr/bin/python
dict = {'Name': 'Zara', 'Age': 7};
print "Length : %d" % len (dict)
```

Output:

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.

```
#!/usr/bin/python
dict = {'Name': 'Zara', 'Age': 7};
print "Equivalent String : %s" % str (dict)
```

Output:

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.

```
#!/usr/bin/python
dict = {'Name': 'Zara', 'Age': 7};
print "Variable Type : %s" %
type (dict)
```

Output:

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.

```
#!/usr/bin/python
dict = {'Name': 'Zara', 'Age': 7};
print "Start Len : %d" %
len(dict)
dict.clear()
print "End Len : %d" %
len(dict)
```

Output:

Start Len : 2

End Len : 0

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.

```
#!/usr/bin/python
dict1 = {'Name': 'Zara', 'Age': 7};
dict2 = dict1.copy()
print "New Dictinary : %s" %
str(dict2)
```

Output:

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.

```
#!/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)
```

Output:

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.

```
#!/usr/bin/python
dict = {'Name': 'Zabra', 'Age': 7}
print "Value : %s" % dict.get('Age')
print "Value : %s" % dict.get('Education', "Never")
```

Output:

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.

```
#!/usr/bin/python
dict = {'Name': 'Zara', 'Age': 7}
print "Value : %s" % dict.has_key('Age')
print "Value : %s" % dict.has_key('Sex')
```

Output:

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.

```
#!/usr/bin/python
dict = {'Name': 'Zara', 'Age': 7}
print "Value : %s" %
dict.items()
```

Output:

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.

```
#!/usr/bin/python
dict = {'Name': 'Zara', 'Age': 7}
print "Value : %s" % dict.keys()
```

Output:

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.

```
#!/usr/bin/python
dict = {'Name': 'Zara', 'Age': 7}
print "Value : %s" % dict.setdefault('Age', None)
print "Value : %s" % dict.setdefault('Sex', None)
```

Output:

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.

```
#!/usr/bin/python
dict = {'Name': 'Zara', 'Age': 7}
dict2 = {'Sex': 'female' }
dict.update(dict2)
print "Value : %s" % dict
```

Output:

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.

```
#!/usr/bin/python
dict = {'Name': 'Zara', 'Age': 7}
print "Value : %s" %
dict.values()
```

Output:

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

Expression oriented functions of Python provides are:

- map(aFunction, aSequence)
- lambda
- filter(aFunction, aSequence)
- reduce(aFunction, aSequence)
- 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)
```

Output:

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))
```

Output:

[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.

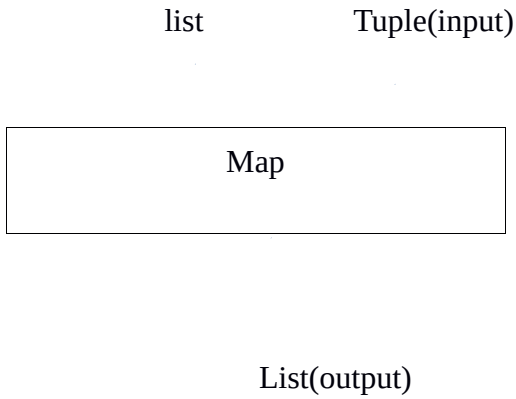
Because map expects a function to be passed in, it also happens to be one of the places where

lambda routinely appears:

```
>>> list(map((lambda x: x **2), items))
```

Output:

[1, 4, 9, 16, 25]



Lambda:

Python supports the creation of anonymous functions (i.e. functions that are not bound to a name) at runtime, using a construct called "lambda". This is not exactly the same as lambda in functional programming languages, but it is a very powerful concept that's well integrated into Python and is often used in conjunction with typical functional concepts like filter(), map() and reduce().

Following is the example of lambda :

```
#!/usr/bin/python
sum = lambda v1,v2:v1+v2
res=sum(10,20)
print "total is ",res              # res=30
res=sum(5,20)
print "total is ",res              # res=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]))
>>> mymap(sqr, [1, 2, 3])
```

Output:

[1, 4, 9]

[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)
>>>pow(2,10)
>>>pow(3,11)
>>>pow(4,12)
>>>list(map(pow,[2, 3, 4], [10, 11, 12]))
```

Output:

```
243
1024
177147
16777216
[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:

```
>>> list1 = [1,2,3]
>>> list2 = [1,4,9]
>>> new_tuple = map(None, list1, list2)
>>> new_tuple
```

Output:

```
[(1, 1), (2, 4), (3, 9)]
```

Filter:

The function filter(function, list) offers an elegant way to filter out all the elements of a list, for which the function function returns True.

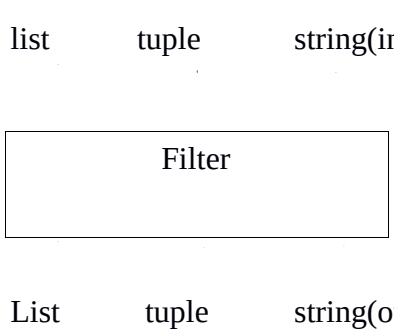
The function filter(funcname,mylist) needs a function funcname as its first argument. funcname returns a Boolean value, i.e. either True or False. This function will be applied to every element of the list mylist. Only if funcname returns True the element of the list be included in the result list.

```
def evennumbers(element):
    rem = (element%2)

    if rem == 0:
        return True

    return False
```

```
numbers=range(1,10)
mylist=filter(evennumbers,numbers)
print mylist
```

Output:

[2,4,6,8]



Reduce:

The reduce(fun,seq) function is used to apply a particular function passed in its argument to all of the list elements mentioned in the sequence passed along.This function is defined in “functools” module.

```
sumfunc=lambda v1,v2:v1+v2
mylist=range(1,101,1)
sumnums=reduce(sumfunc,mylist)
print sumnums

maxvalue=lambda v1,v2:v1 if(v1>v2) else v2

mylist=[10,40,34,67,0,55]
maximum=reduce(maxvalue,mylist)
print maximum
```

Output:

5050

67

List Comprehension:

Python supports a concept called "list comprehensions". It can be used to construct lists in a very natural, easy way, like a mathematician is used to do.

```
mylist=range(10)
myset=[val ** 2 for val in mylist]
```

```
print myset

mylist=range(13)
myvector=[2 ** val for val in mylist]
print myvector

even_multiples=[val for val in mylist]
print even_multiples
```

Output:

[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

[1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096]

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]

Nested list comprehension:

```
res=[]
for val1 in range(2,4):
        for val2 in range(5,8):
                res.append(val1)
print res

mylist=[val1 for val1 in range(2,4)]
```

Output:

[2,2,2,3,3,3]

Tuple Unpacking:

In packing, we place value into a new tuple while in unpacking we extract those values back into variables.

Example:

```
tuple1="one","two"
tuple2=1,2

tuple3=tuple2+tuple1
print tuple3

item1,item2,item3,item4=tuple3
print item1,item2,item3,item4
```

```
mylist=['a','b','c','d','end']
print mylist

mylist[0],mylist[1],mylist[2],mylist[3]=tuple3
print mylist
```

Output:

(1, 2, 'one', 'two')

1 2 one two

['a', 'b', 'c', 'd', 'end']

[1, 2, 'one', 'two', 'end']

Sets:

1. Collection of elements
2. Sets should not have a duplicate elements
3. A set contains the elements in different order
4. To a set we can pass tuple as a element but we should not pass list.

Example:

```
charset=set("A python tutorial")
print "set is",charset
cities=("hyderabad","paris","london","paris","berlin")
cities=set(cities)
print "print unique elements",cities
cities=set((["python","perl"],["paris","berlin","london"]))
print "including lists as elements",cities          #Error because list should not be as a argument
```

Output:
set is set(['A', ' ', 'i', 'h', 'l', 'o', 'n', 'p', 'r', 'u', 't', 'a', 'y'])
print unique elements set(['paris', 'hyderabad', 'berlin', 'london'])

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
```

Ouput :

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
```

Output
sum of two numbers is: 3

Example:

```
step1: as same as above

step2:
#!/usr/bin/python
$from file import sub
$result=sub(1,2)
$print “subraction of two numbers is:”,result
```

Output
$./file1.py
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.

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.

The return type of both these functions is dictionary. Therefore, names can be extracted using the keys() function.

Example:

Let's have a look at the following function:

```
#!/usr/bin/python
v1 =50
def func(v1):
        print 'v1 is:',v1
        v1=2
        print 'changed local v1 to',v1
        return
func(v1)
print 'v1 is still',v1
```

Output:

v1 is 50

v1 is 2

v1 is still 50

The value of v1 is defined as 50 before function(globally). As we are changing the value locally in the function it wont get effected out of the function.so the value will be unchanged after the function also.

```
#!/usr/bin/python
```

```
def func():
      print 'v1 is:',v1
      v1=2
      print 'changed local v1 to',v1
      return
v1 =50
func()
print 'v1 is',v1
```

If we execute the previous script, we get the error message:

UnboundLocalError: local variable 'v1' referenced before assignment

Python "assumes" that we want a local variable due to the assignment to v1 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:

```
#!/usr/bin/python

def func():
      global v1
      print 'v1 is:',v1
      v1=2
      print 'changed local v1 to',v1
      return
v1 =50
func()
print 'v1 is',v1
```

Now there is no ambiguity. The output looks like this:

v1 is: 50
changed local v1 to 2
v1 is 2

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
```

PACKAGES AND MODULES – 15.05.2017

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 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()
```

Output:

```
After executing File.py
 I'm Pots Phone
 I'm ISDN Phone
 I'm 3G Phone
```

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.

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))
```

Output:

```
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)
```

Output:

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

1. The python sys module provides access to any command-line arg via the sys.argv this serves two purpose
2. sys.argv is the list of command line arguments
3. len(sys.argv) is the no of command line arguments
4. 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,"hi: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:])
```

Output:

input file is votary

output file is tech

EXCEPTIONS – 16.05.2017

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 for raising exception:

```
#!/usr/bin/python
import math
anumber=int(input(“enter a number:”))
if anumber<0:
                raise RuntimeError(“you cant use this number”)
else:
                 print(math.sqrt(anumber))
print(“End of program”)
```

Output:

```
Enter an integer: 5
2.2360679775
end of program

Enter an integer: -2
Traceback (most recent call last):
  File "./raising_exception.py", line 10, in <module>
    raise RuntimeError("You can't use a negative number")
RuntimeError: You can't use a negative number
```

Example for exception handling:

```
#!/usr/bin/python
import math,sys
ano=int(input(“enter a number:”))
try:
      print (math.sqrt(ano))
except:
        print (“Exception is:”,sys.exec_type)
        print (“Bad value of sqrt”)
        print (“Using absolute value instead”)
        print(math.sqrt(abs(ano)))
else:
         print (math.sqrt(x))
```

Output:

```
Enter an integer: 4
2.0
end of program

Enter an integer: -2
('Exception is:', <type 'exceptions.ValueError'>)
Bad value of sqrt
Using absolute value instead
1.41421356237
end of program
```

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.

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 foo.txt

```
#!/usr/bin/python

fo = open("foo.txt","w")
print "Name of file: ",fo.name
print "Closed or not: ",fo.closed
print "Opening mode: ",fo.mode
fo.close()
print "Closed or not: ",fo.closed
```

Output:

```
Name of file:  foo.txt
Closed or not:  False
Opening mode:  w
Closed or not:  True
```

Reading a file:

```
#!/usr/bin/python

infile=open("sample2.txt","r")          # Opening an existing file for reading
contents=infile.read()                  # reads entire file as a single string
print contents
infile.close()
```

Output:
This is to read the complete file
as a single string

```
f = open('sample1.txt', 'r')
contents = f.readline()       # reads one line
print contents
f.close()
```

Output:
Python programming is interesting

Example:

```
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
```

Output:
<type 'list'> ['Python programming is interesting\n', 'It is coming with batteries, in built\n', 'It means that almost every operation has a module !\n']

File Positions:

Reading the files based on the file positions

0 - Beginning of file
1 - From current position
2 – End of file

Example:

```
#!/usr/bin/python

fo=open("fileseek.txt","r+")
str=fo.read(13)
print "Read string is: ",str
position=fo.tell()
print "Position is: ",position
position=fo.seek(2,0)
```

```
str=fo.read(12)
print "Read string is: ",str
position=fo.tell()
print "Position is: ",position
position=fo.seek(-10,2)
str=fo.read(10)
print "Read string is: ",str
position=fo.tell()
print "Position is: ",position
fo.close()
```

The data in fileseek.txt is:
Python is a greatest language.
Yeah its great!!

Output:

Read string is: Python is a g
Position is: 13
Read string is: thon is a gr
Position is: 14
Read string is: s great!!
Position is: 48

NETWORKING – 17.05.2017

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:

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.

For communication we need 5-tuple which means 5 sets of data

- socket type (TCP or UDP)
- client ip address
- client port address
- server ip address
- server port address

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
```

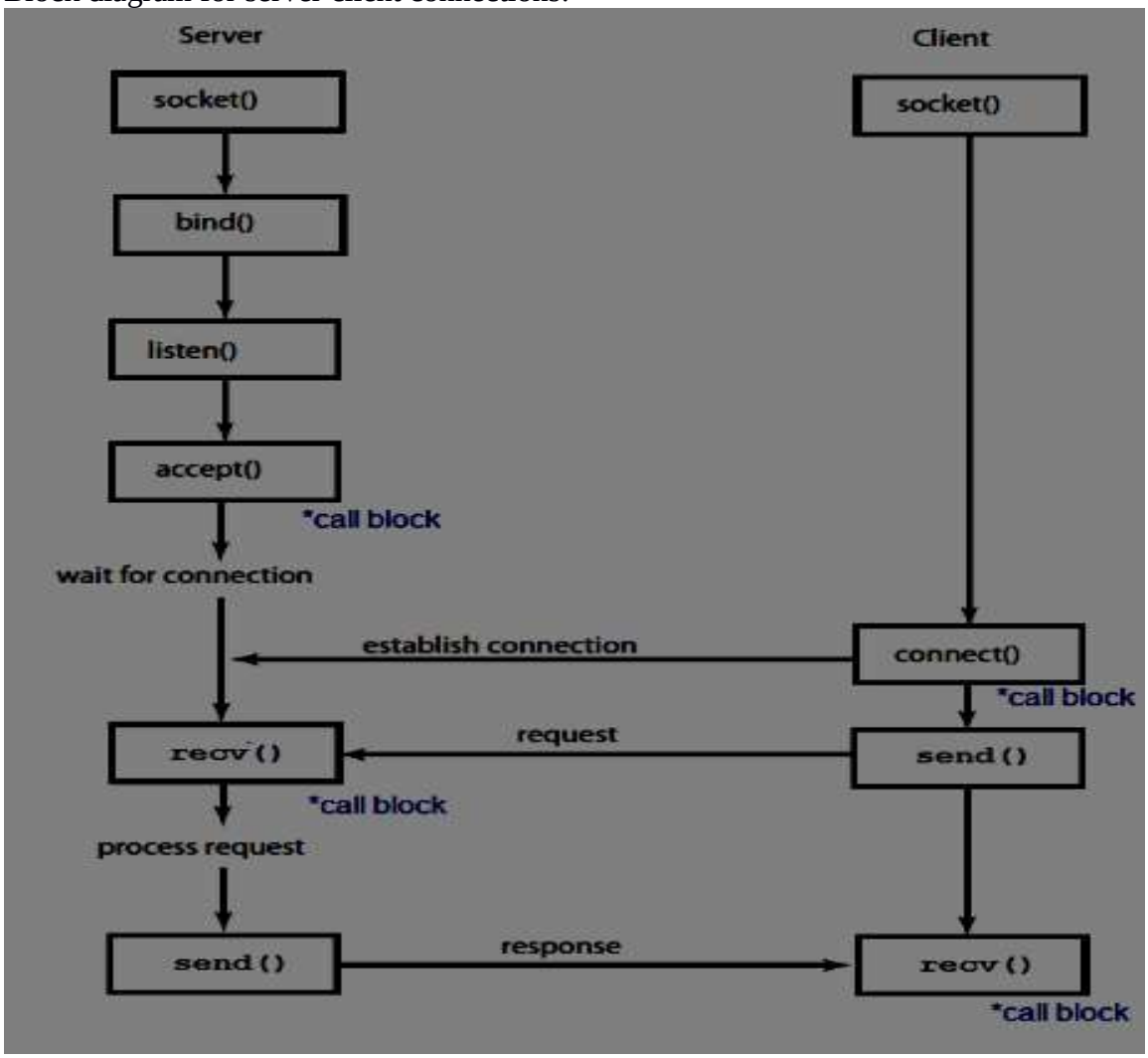
Now run this server.py in background and then run above client.py to see the result.

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:



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.close()**: 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."

OBJECT ORIENTED PROGRAMMING – 18.05.2017

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

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

Creating class and object:

```
Class name:
      #variables
       var=10
       def  display():
            print var
obj=name()
obj.display()
```

Output:

10

```
Class emp:
       empcnt=0
```

```
        def __init__(self,name,salary):
                    self.name=name
                     self.salary=salary
                      emp.empcnt+=1
         def displaycnt(self):
                  print “total emp:”,emp.empcnt
          def displayemp(self):
                  print “emp:”,self.name
obj1=emp(“deepu”,1000000)
obj1.displaycnt()
obj1.displayemp()
obj2=emp(“sandeep”,1100000)
obj2.displaycnt()
obj2.displayemp()
obj1.age=30  #creates new attribute
print obj1.age
print hasattr(obj1,'age') #checks for existance of age attribute prints true
print getattr(obj1,'age') #print age vale
setattr(obj1,'age',55)
print obj1.age
del obj1.age   #deletes the attribute age
print hasattr(obj1,'age') #checks for existance of age attribute prints false
```

Output:

```
total emp:1
emp:deepu
total emp:2
emp:sandeep
30
true
30
55
false
```

Constructor and Destructor:

Constructor is special function which is used to initialise or construct the data members of an object.It will be called only once in a life time of the object and it will be automatically called whenever the object is created

Destructor is used to destroy the constructor

Eg:

```
Class point:
```

```
        def __init__(self,val1=0,val2=0):  #constructor
                print "in init"
                 self.val1=val1
                 self.val2=val2
        def __del__(self):
                class_name=self.__class__.__name__
                print "destroyed ",class_name
obj=point(10,20)
```

Output:

in init
destroyed obj

Inheritance:

The derived class inherits the attributes of its baset class, and you can use those attributes as if they were defined in the derived class. A derived class can also override data members and methods from the base.We can define our own attributes or functions in derived class,and these cant be accessed by baset class

Syntax:

```
class derived (baseClass1[, baseClass2, ...]):
'Optional class documentation string'
        class_suite
```

Eg:

```
Class base:
      baseattr=10
      def __init__(self):
            print "in base constructor"
      def basemethod(self):
            print "in base method"
      def setattr(self,attr):
           base.baseattr=attr
       def getattr(self):
           print "base attr:",base.baseattr
class derived(base):
       def __init__(self):
           print "in derived constructor"
       def derivedmethod(self):
           print "in derived method"
obj1=derived()
obj.basemethod()
obj.setattr(12)
obj.getattr()
```

Output:

in derived constructor
in base method

base attr:12

Overriding Methods:

You can always override your base class methods. One reason for overriding base's methods is because you may want special or different functionality in your subclass.

Eg:

```
Class base:
        def mymethod(self):
                print “in base method”
class derived(base):
        def mymethod(self):
                print “in derived method”
obj1=derived()
obj1.mymethod()
obj2=base()
obj2mymethod()
```

Output:

in derived method
in base method

Operator Overloading:

Suppose you have created a Vector class to represent two-dimensional objects and want to add those two objects we use operator overloading

```
#!/usr/bin/python
class vector:
      def __init__(self,val1,val2):
            self.val1=val1
            self.val2=val2
      def __repr__(self):
            return "vector(%d %d)"%(self.val1,self.val2)
      def __add__(self,other):
            res=vector(self.val1+other.val1,self.val2+other.val2)
            return res
      def __sub__(self,other):
            res=vector(self.val1-other.val1,self.val2-other.val2)
            return res
      def __mul__(self,other):
            res=vector(self.val1*other.val1,self.val2*other.val2)
            return res
obj1=vector(10,20)
obj2=vector(1,2)
```

```
res = obj1+obj2
print res
```

Output:

```
vector(11 22)
vector(9 18)
vector(10 40)
```

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.

Eg;

```
#!/user/bin/python
class base:
      __cnt=0
      def fun(self):
#            cnt+=1
             self.__cnt+=1
             print self.__cnt
c=base()
c.fun()
#print c.__cnt  #cant access outside
```

Output:

```
1
```

```
#!/user/bin/python
##this file name is file.py


class textfile:
      ntfiles=0
      def __init__(self,fname):
            textfile.ntfiles+=1
            self.name=fname
            self.fh=open(fname,"r")
            self.lines=self.fh.readlines()
            self.nlines=len(self.lines)
            print self.nlines
            self.nwords=0
            self.wordcount()
      def wordcount(self):
```

```
            for line in self.lines:
                words=line.split()
                self.nwords+=len(words)
        def grep(self,target):
            for line in self.lines:
                if line.find(target)>=0:
                    print line
#       def totfiles1():
#            print "no.of text file:",textfile.ntfiles
        def totfiles():
                  print "no.of text file:",textfile.ntfiles

        totfiles2=staticmethod(totfiles)            ##static function
obj=textfile("file.txt")
##file.txt have this program as data
obj2=textfile("file.txt")
obj.wordcount()
obj.grep("def")
#obj.totfiles1()
textfile.totfiles2()
```

Output:

22

22

```
        def __init__(self,fname):
        def wordcount(self):
        def grep(self,target):
```

no.of text file: 2

```
#!/user/bin/python
class robot:
        population=0
        def __init__(self,name):
            self.name=name
            print "intialising {}",self.name
            robot.population+=1
        def die(self):
            print "{} is destroyed",(self.name)
            robot.population-=1
            if(robot.population==0):
```

```
                print "{} was last one",self.name
            else:
                print "remaining {} ",robot.population

        def say_hi(self):
            print "greeting",self.name
#       def how_many(cls):
#            print "we have {} robots",cls.population
        def how_many():
            print "we have {} robots",robot.population
        how_many=staticmethod(how_many)
obj1=robot("R2-D2")
obj1.say_hi()
#robot.how_many(obj1)
robot.how_many()

obj2=robot("abc")
obj2.say_hi()
#robot.how_many(obj1)
robot.how_many()

obj1.die()
obj2.die()
#robot.how_many(obj1)
robot.how_many()
```

Output:

```
intialising {} R2-D2
greeting R2-D2
we have {} robots 1
intialising {} abc
greeting abc
we have {} robots 2
{} is destroyed R2-D2
remaining {}  1
{} is destroyed abc
{} was last one abc
we have {} robots 0
```

REGULAR EXPRESSIONS

Regular expressions are text matching patterns described with a formal syntax. The patterns are interpreted as a set of instructions, which are then executed with a string as input to produce a matching subset or modified version of the original. The term "regular expressions" is frequently shortened to as "regex".

Expressions can include literal text matching, repetition, pattern-composition, branching, and other sophisticated rules. A large number of parsing problems are easier to solve with a regular expression than by creating a special-purpose lexer and parser.

Match function:

Syntax: r*e.match(pattern, string, flags=0)*

This function will searches the string with starting word of pattern

Eg:

```
Import re
pattern=" this is my first python program"
matchobj=re.match(r'this',pattern,re.M | re.I)
if matchobj:
    print "matched:",matchobj.group()
else:
    print "no match"
```

Output:

matched:this

```
Import re
pattern=" this is my first python program"
matchobj=re.match(r'is',pattern,re.M | re.I)
if matchobj:
    print "matched:",matchobj.group()
else:
    print "no match"
```

Output:

no match

Search function:

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

this function will searches the string in whole pattern

Eg:

```
import re
pattern=" this is my first python program"
matchobj=re.search(r'is',pattern,re.M | re.I)
if matchobj:
    print "matched:",matchobj.group()
else:
    print "no match"
```

Output:

matched:is

The **start()** and **end()** methods give the integer indexes into the string showing where the text matched by the pattern occurs.

```
Import re
pattern=" this is my first python program"
matchobj=re.search(r'this',pattern,re.M | re.I)
print matchobj.start()
print matchobj.end()
```

Output:

0

4

Findall function:

So far the example patterns have all used search() to look for single instances of literal text strings. The findall() function returns all of the substrings of the input that match the pattern without overlapping.

Eg:

```
Import re
pattern=" this is my first python program"
print re.findall(r'is',pattern,re.M | re.I)
```

Output:

['is','is']

Finditer:

finditer() returns an iterator that produces Match instances instead of the strings returned by findall().

Eg:

```
import re
```

```
pattern=” this is my first python program”
for obj in re.finditer(r’is’,pattern,re.M | re.I):
        st = obj.start()
        end=obj.end()
    print 'Found "%s" at %d:%d' % (pattern[st:end], st, end)
```

Output:

Found “is” at 2:4

Found “is” at 5:7

Escape Codes:

An even more compact representation uses escape codes for several pre-defined character sets. The escape codes recognized by re are:

Code	Meaning
\d	a digit
\D	a non-digit
\s	whitespace (tab, space, newline, etc.)
\S	non-whitespace
\w	alphanumeric
\W	non-alphanumeric
+	1 or more
*	0 or more

```
import re
phone = '123-66-565-54 #this is number'
num=re.sub(r'#.*$',",phone)
print num
#print phone
num =re.sub(r'\D',",phone)
print num
```

Output:

123-66-565-54

1236656554

```
Import re
```

```
cont='voatarytech 123-45,abc,xyz:555-4545 hyd 500001
mat=re.search(r'\w+,\w+:\S+',cont)
print mat.group(0)
```

Output:

'abc,xyz:45-48'

```
Import re
cont='voatarytech 123-45,abc,xyz:555-4545 hyd 500001
mat=re.search(r'(\w+),(\w+):(\S+)',cont)
print mat.group(1)
print mat.group(2)
print mat.group(3)
```

Output:

abc

xyz

555-4545

MULTITHREADING – 19.05.2017

This module constructs higher-level threading interfaces on top of the lower level thread module.

Threads are usually contained in processes. More than one thread can exist within the same process. These threads share the memory and the state of the process. In other words: They share the code or instructions and the values of its variables.

Advantages:

- Execution of the program is faster as they are sharing same address space
- A program can remain responsive to input. This is true both on single and on multiple CPU

Disadvantage:

- Data curruption as they share same global variables

There are two modules which support the usage of threads in Python:

- thread
- threading

It's possible to execute functions in a separate thread with the module Thread. To do this, we can use the function thread.start_new_thread:

thread.start_new_thread(function, args[, kwargs])

This method starts a new thread and return its identifier. The thread executes the function "function" (function is a reference to a function) with the argument list args (which must be a list or a tuple).

The optional kwargs argument specifies a dictionary of keyword arguments. When the function returns, the thread silently exits. When the function terminates with an unhandled exception, a stack trace is printed and then the thread exits (but other threads continue to run).

Eg:

```
import thread
import time
def delay_loop(task,secs):
	print "in delay loop"
	print "%s:sleep %d secs"%(task,secs)
	for counter in range(secs):
		time.sleep(counter)
	print "ending delay loop"
def print_time(task,delay):
	cnt=0
	while cnt<5:
		print "in print"
		time.sleep(delay)
		cnt+=1
		secs=time.time()
		print secs
		cal_time=time.ctime(secs)
		print "%s:%s"%(task,cal_time)
try:
	thread.start_new_thread(delay_loop,('thread1',5,))
	thread.start_new_thread(print_time,('thread2',10,))
except:
	 print "error"
while True:
	pass
```

Output:

in delay loop

thread1:sleep 5 secs

in print

1493016337.96 ending delay loop

thread2:Mon Apr 24 12:15:37 2017

in print

1493016347.97

thread2:Mon Apr 24 12:15:47 2017

in print

1493016357.99

thread2:Mon Apr 24 12:15:57 2017

in print

1493016367.99

thread2:Mon Apr 24 12:16:07 2017

in print

1493016378.0

thread2:Mon Apr 24 12:16:18 2017 #here while loop will executes

```
import thread
import time
def print_time(task,delay):
        cnt=0
        while cnt<5:
#               print "in print"
                time.sleep(delay)
                cnt+=1
                #secs=time.time()
                #cal_time=time.ctime(secs)
                #print "%s:%s"%(task,cal_time)
                print "%s:%d"%(task,cnt)
        print "%s exited"%(task)
try:
        thread.start_new_thread(print_time,('thread1',5,))
        thread.start_new_thread(print_time,('thread2',10,))
except:
        print "error"
while True:
        pass
```

Output:

thread1,1

thread2,1 thread1,2

thread1,3

thread2,2

thread1,4

thread1,5

thread1 exited

thread2,3

thread2,4

thread2,5

thread2 exited

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.

```
import threading
import time
exitflag=0
class mythread(threading.Thread):
    def __init__(self,threadid,name,counter):
        threading.Thread.__init__(self)
        self.threadid=threadid
        self.counter=counter
        self.name=name
    def run(self):
        print "starting"+self.name
        print_time(self.name,self.counter,5)
        print "exiting"+self.name

def print_time(threadname,delay,cnt):
```

```
    while cnt:
        if exitflag:
            thread.exit()
        time.sleep(delay)
        cal_time=time.ctime(time.time())
        print "%s:%s"%(threadname,cal_time)
        cnt-=1
thread1=mythread(1,"thread1",1)
thread2=mythread(2,"thread2",2)
thread1.start()
thread2.start()
print "exiting"
```

Output:

thread1:Mon Apr 24 12:32:00 2017

thread2:Mon Apr 24 12:32:01 2017

thread1:Mon Apr 24 12:32:01 2017

thread1:Mon Apr 24 12:32:02 2017

thread2:Mon Apr 24 12:32:03 2017

thread1:Mon Apr 24 12:32:03 2017

thread1:Mon Apr 24 12:32:04 2017

exitingthread1

thread2:Mon Apr 24 12:32:05 2017

thread2:Mon Apr 24 12:32:07 2017

thread2:Mon Apr 24 12:32:09 2017

exitingthread2

Data corruption:

```
import threading
import time,sys
class global_value(threading.Thread):
    globval=0
    def __init__(self,threadid,name,loopcnt):
        threading.Thread.__init__(self)
        self.threadid=threadid
        self.loopcnt=loopcnt
        self.name=name
    def run(self):
        print "starting"+self.name
```

```
            global_value.globval+=1
            print “%s : %d”%(self.name,self.globval)
            time.sleep(3)
            print "exiting"+self.name
thread1=global_value(1,"thread1",loop)
thread2=global_value(2,"thread2",loop)
thread1.start()
thread2.start()
thread1.join()
thread2.join()
print "exiting"
```

Output:

startingthread1

thread1:1

startingthread2

thread2:2

exitingthread1

exitingthread2

exiting

Sync Data:

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.

```
import threading
import time
exitflag=0
v=0
class mythread(threading.Thread):
    def __init__(self,threadid,name,counter):
        threading.Thread.__init__(self)
        self.threadid=threadid
        self.counter=counter
        self.name=name
```

```
    def run(self):
        print "starting"+self.name
        threadLock.acquire()
        print_time(self.name,self.counter,5)
        print "exiting"+self.name
        threadLock.release()

def print_time(threadname,delay,cnt):
    while cnt:
        if exitflag:
            thread.exit()
#         print v
#         v+=1
         time.sleep(delay)
        cal_time=time.ctime(time.time())
        print "%s:%s"%(threadname,cal_time)
        cnt-=1
threadLock=threading.Lock()
threads=[]
thread1=mythread(1,"thread1",1)
thread2=mythread(2,"thread2",2)
thread1.start()
thread2.start()
threads.append(thread1)
threads.append(thread2)
#print threads
for t in threads:
    t.join()

print "exiting"
```

Output:

startingthread1

startingthread2

thread1:Mon Apr 24 14:15:52 2017

thread1:Mon Apr 24 14:15:53 2017

thread1:Mon Apr 24 14:15:54 2017

thread1:Mon Apr 24 14:15:55 2017

thread1:Mon Apr 24 14:15:56 2017

exitingthread1

thread2:Mon Apr 24 14:15:58 2017

thread2:Mon Apr 24 14:16:00 2017

thread2:Mon Apr 24 14:16:02 2017

thread2:Mon Apr 24 14:16:04 2017

thread2:Mon Apr 24 14:16:06 2017

exitingthread2

exiting

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.

```
import os,time
print "before"
val=0
print os.fork()
print "after1"
os.fork()
os.fork()
print "after2"
```

Output:

befor

4962

after1

after1

after2

after2

after2

after2

after2

after2

after2

after2

```
import os,time,sys
def print_time(processname,delay):
        cnt=0
        while cnt<5:
                time.sleep(delay)
                cnt+=1
                print "%s cnt:%d"%(processname,cnt)
#               print "%s :%s"%(processname,time.ctime(time.time()))
processid=os.fork()
if processid:
        print "parent"
        print_time("parent",2)
        print processid
#       time.sleep(2)
        print "parent exiting"
else:
        print "child"
        print_time("child",2)
        print processid
        time.sleep(2)
        print "child exiting"
        sys.exit(0)
time.sleep(10)
```

Output:

parent

child

parent cnt:1

child cnt:1

parent cnt:2

child cnt:2

parent cnt:3

child cnt:3

parent cnt:4

child cnt:4

parent cnt:5

child cnt:5

5024

parent exiting

0

child exiting

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.

```
Import subprocess,os
print os.system('ls -l') #prints ls -l cmd output
print subprocess.call('ls -l')#prints ls -l cmd output
print subprocess.call('echo $HOME',shell=True)#print home path
```

Output:

prints output of ls -l command

/home/dir

```
Import subprocess,os
buf=subprocess.check_output('ls -l')
print buf   ##prints ls -l output
```

```
import os,subprocess
print os.popen("echo hello").read()
print subprocess.Popen("echo hello",stdout=subprocess.PIPE,shell=True).stdout.read()
proc= subprocess.Popen(["echo"," hello"],stdout=subprocess.PIPE)
stddata=proc.communicate()
```

```
print stddata
```

Output:

hello

hello

(' hello\n', None)