Python Basic Operators

- Python language supports the following types of operators.
- Arithmetic Operators a + b, a – b
- Comparison (Relational) Operators >=,<=,==,!=
- Assignment Operators =, c **+=** a is equivalent to ***c = c + a***
- Logical Operators and(&) , or(|) , not(~)
- Membership Operators in, not in
  - **x in y**, here in results in a 1 if **x is a member of sequence y**.

Python Decision Making- use of Break

The **break** statement in Python terminates the current loop and resumes execution at the next statement.

```
for letter in 'Python':                                  # First Example
if letter == 'h':
    break
    print 'Current Letter :', letter


    var = 10                                             # Second Example
    while var > 0:
        print 'Current variable value :‘, var
        var = var -1
        if var == 5:
            break
    print "Good bye!"
```

- Current Letter : P
- Current Letter : y
- Current Letter : t

- Current variable value : 10
- Current variable value : 9
- Current variable value : 8
- Current variable value : 7
- Current variable value : 6

- Good bye!

Python Decision Making- use of Continue

- The **continue** statement in Python returns the control to the beginning of the while loop.

- The **continue** statement rejects all the remaining statements in the current iteration of the loop and moves the control back to the top of the loop.

- The **continue** statement can be used in both *while* and *for* loops.

Python Decision Making- use of Continue

```
for letter in 'Python':
   if letter == 'h':
      continue

   print 'Current Letter :', letter
```

- Current Letter : P
- Current Letter : y
- Current Letter : t
- Current Letter : o
- Current Letter : n

Python Decision Making- use of Continue

```
var = 10
while var > 0:
   var = var -1
   if var == 5:
      continue
   print 'Current variable value :', var
```

- Current variable value : 9
- Current variable value : 8
- Current variable value : 7
- Current variable value : 6
- Current variable value : 4
- Current variable value : 3
- Current variable value : 2
- Current variable value : 1
- Current variable value : 0

Mathematical Functions-cmath

- abs(x): The method **abs()** returns absolute value of **x**
  - print "abs(-45) : ", abs(-45) 45
  - print "abs(100.12) : ", abs(100.12) 100.12

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

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

log(): The method **log()** returns natural logarithm.

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

Python Functions

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

- As you already know, Python gives you many built-in functions like print(), etc. but you can also create your own functions.

- These functions are called *user-defined functions.*

Defining a Function

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

Syntax

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

```
def printme( str ):
   "This prints a passed string into this function"
   print str
   return
```

Calling a Function

- Defining a function only gives it a name, specifies the parameters that are to be included in the function and structures the blocks of code.

- Once the basic structure of a function is finalized,

- you can execute it by calling it from another function or directly from the Python prompt.

```
def printme( str ):
   "This prints a passed string into this function"
   print str
   return;

# Now you can call printme function
printme("I'm first call to user defined function!")
printme("Again second call to the same function")
```

Pass by reference vs value

- # Function definition is here
- def changeme(mylist):
- "This changes a passed list into this function"
- mylist.append([1,2,3,4]);
- print "Values inside the function: ", mylist
- return

- # Now you can call changeme function
- mylist = [10,20,30];
- changeme(mylist);
- print "Values outside the function: ", mylist

```
def changeme( mylist ):
   "This changes a passed list into this function"
   mylist = [1,2,3,4]; # This would assig new reference in mylist
   print "Values inside the function: ", mylist
   return

# Now you can call changeme function
mylist = [10,20,30];
changeme( mylist );
print "Values outside the function: ", mylist
```

Function Arguments

- Required arguments
- Keyword arguments
- Default arguments
- Variable-length arguments

Required arguments

- Required arguments are the arguments passed to a function in correct positional order.
- Here, **the number of arguments in the function call should match exactly with the function definition**.
- # Function definition is here

```
def printme( str ):
   Print("This prints a passed string into this function“)
   print str
   return;

# Now you can call printme function
printme(“HELLO”)
```

Keyword arguments

- When you use keyword arguments in a function call, the caller identifies the arguments by the parameter name.
- This allows you to **place them out of order** because the Python interpreter is able to use the keywords provided to match the values with parameters

- # Function definition is here
- def **printme(str):**
- "This prints a passed string into this function"
- print str
- return;

Now you can call printme function

printme(str = "My string")

```
# Function definition is here
def printinfo( name, age ):
   "This prints a passed info into this function"
   print "Name: ", name
   print "Age ", age
   return;

#Now you can call printinfo function
printinfo( age=50, name="miki" )
```

Default arguments

A default argument is an argument that assumes a default value if a value is not provided in the function call for that argument.

```
# Function definition is here
def printinfo( name, age = 35 ):
   "This prints a passed info into this function"
   print "Name: ", name
   print "Age ", age
   return;

# Now you can call printinfo function
printinfo( age=50, name="miki" )
printinfo( name="miki" )
```

- Name: miki
- Age 50
- Name: miki
- Age 35

Variable-length arguments

- You may need to process a function for more arguments than you specified while defining the function.
- These arguments are called *variable-length* arguments and **are not named in the function definition.**

```
def printinfo( arg1, *vartuple ):
   "This prints a variable passed arguments"
   print "Output is: "
   print arg1
   for var in vartuple:
      print var
   return;

# Now you can call printinfo function
printinfo( 10 )
printinfo( 70, 60, 50 )
```

```
10
70
60
50
```

The *return* Statement

- The statement return [expression] exits a function, optionally passing back an expression to the caller.
- A return statement with no arguments is the same as return None.

```
# Function definition is here
def sum( arg1, arg2 ):
   # Add both the parameters and return them."
   total = arg1 + arg2
   print "Inside the function : ", total
   return total;

# Now you can call sum function
total = sum( 10, 20 );
print "Outside the function : ", total
```

Inside the function : 30

Outside the function : 30

Scope of Variables

- All variables in a program may not be accessible at all locations in that program.
- This depends on where you have declared a variable.
- The scope of a variable determines the portion of the program where you can access a particular identifier.

- There are two basic scopes of variables in Python –
- Global variables
- Local variables

Global vs. Local variables

- Variables that are defined inside a function body have a local scope, and those defined outside have a global scope.

- This means that local variables can be accessed only inside the function in which they are declared, whereas global variables can be accessed throughout the program body by all functions.

Inside the function local total : 30
Outside the function global total : 0

- total = 0; # This is global variable.
- # Function definition is here
- def sum(arg1, arg2):
- # Add both the parameters and return them."
- total = arg1 + arg2; # Here total is local variable.
- print "Inside the function local total : ", total
- return total;

- # Now you can call sum function
- sum(10, 20);
- print "Outside the function global total : ", total

Python Tuples

- A tuple is a sequence of immutable Python objects.
- 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.

- tup1 = ('physics', 'chemistry', 1997, 2000)
- tup2 = (1, 2, 3, 4, 5, 6, 7)
- print (tup2[1:5])
- (2, 3, 4, 5)
- ----------
- >>> tup1 = (12, 34.56);
- >>> tup2 = ('abc', 'xyz')
- >>> tup3 = tup1 + tup2
- >>> print(tup3)
- (12, 34.56, 'abc', 'xyz')
- >>>

Delete Tuple Elements

- Removing individual tuple elements is not possible.
- To explicitly remove an entire tuple, just use the **del** statement.

- >>> tup = ('physics', 'chemistry', 1997, 2000);
- >>> print(tup)
- ('physics', 'chemistry', 1997, 2000)
- >>> del tup;

Basic Tuples Operations

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

Built-in Tuple Functions

- **Max() ,Min() Method:**
- tuple2 = (123, 200),
- >>> max(tuple2)
- 200
- min(tuple2)
- 123

- **tuple() Method:**
- aList = [123, 'xyz', 'zara', 'abc']
- >>> aTuple = tuple(aList)
- >>> print(aTuple)
- (123, 'xyz', 'zara', 'abc')

Delete List Elements

- >>> list1 = ['physics', 'chemistry', **1997**, 2000];
- >>> del list1[2];
- >>> print (list1)
- ['physics', 'chemistry', 2000]

Python Lists

- >>> list = ['physics', 'chemistry', 1997, 2000];
- >>> print (list[2])
- 1997

- >>> list[2] = 2001
- >>> print(list)
- ['physics', 'chemistry', 2001, 2000]