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,
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.
When you call a function, the variables declared inside it are brought into scope.

Example (try):

```
#!/usr/bin/python

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
```

Output calling the script:

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

Pass function arguments by reference vs. value

All parameters (arguments) of functions in Python are passed by reference:
if you change what a parameter refers to within a function,
then the change also reflects back in the calling function.

Example:

```
#!/usr/bin/python

# 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:\n ", mylist
   return

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

Output:

```
Values inside the function:
[10, 20, 30, [1, 2, 3, 4]]

Values outside the function:
[10, 20, 30, [1, 2, 3, 4]]
```

Be careful duplicating variable names

```
!/usr/bin/python
# Function definition is here
def changelist( mylist ):
   "This changes a passed list into this function"
   mylist = [1,2,3,4];      # This assig new reference in mylist
   print "Values inside the function: ", mylist
   return

# Function call
mylist = [10,20,30];
changelist( mylist );
print "Values outside the function: ", mylist
```

The parameter mylist is local to the function changelist. Changing mylist within the function does not affect mylist outside the function.

Output:

```
Values inside the function:  [1, 2, 3, 4]
Values outside the function:  [10, 20, 30]
```

The Anonymous Functions

Anonymous functions are not declared in the standard manner by using the def keyword. The lambda keyword is used to create small anonymous functions.

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

Syntax

The syntax of lambda functions contains only a single statement, which is as follows:

lambda [arg1 [,arg2,.....argn]]:expression

Example: The Anonymous Functions

```
#!/usr/bin/python

# Function definition is here
sum = lambda arg1, arg2: arg1 + arg2;

# Now you can call sum as a function
print "Value of total : ", sum( 10, 20 )
print "Value of total : ", sum( 20, 20 )
```

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

```
Value of total :  30
Value of total :  40
```

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.

Example of return value

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

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

```
Inside the function :  30
Outside the function :  30
```