If I'm understanding your situation correctly, what you're seeing is the result of how Python handles local (function) and global (module) namespaces.

Say you've got a module like this:

# sample.py

myGlobal = 5

def func1():

myGlobal = 42

def func2():

print myGlobal

func1()

func2()

You might expecting this to print 42, but instead it prints 5. As has already been mentioned, if you add a 'global' declaration to func1(), then func2() will print 42.

def func1():

global myGlobal

myGlobal = 42

What's going on here is that Python assumes that any name that is *assigned to*, anywhere within a function, is local to that function unless explicitly told otherwise. If it is only *reading* from a name, and the name doesn't exist locally, it will try to look up the name in any containing scopes (e.g. the module's global scope).

When you assign 42 to the name myGlobal, therefore, Python creates a local variable that shadows the global variable of the same name. That local goes out of scope and is [garbage-collected](http://www.digi.com/wiki/developer/index.php/Python_Garbage_Collection) when func1() returns; meanwhile, func2() can never see anything other than the (unmodified) global name. Note that this namespace decision happens at compile time, not at runtime -- if you were to read the value of myGlobal inside func1() before you assign to it, you'd get an UnboundLocalError, because Python has already decided that it must be a local variable but it has not had any value associated with it yet. But by using the 'global' statement, you tell Python that it should look elsewhere for the name instead of assigning to it locally.

(I believe that this behavior originated largely through an optimization of local namespaces -- without this behavior, Python's VM would need to perform at least three name lookups each time a new name is assigned to inside a function (to ensure that the name didn't already exist at module/builtin level), which would significantly slow down a very common operation.)