# PEP: 227 - Statically Nested Scopes

**Louis Bouddhou, Alex Campbell, Josh Fermin**

## What is the problem?

The PEP we chose to do is PEP 227 – Statically Nested Scopes. The main problem that is being addressed is that developers cannot assume lexical scoping within nested functions. Specifically, developers cannot reference variables in higher order functions (the functions that functions are nested within). This is a huge problem while programming with lambdas. Lambdas and other nested functions cannot reference variables defined in the surrounding namespace. To be clear, static scoping was already implemented for the python language before this pep, but PEP 227 introduces static scoping within nested functions.

## Example - Without Statically Nested Scopes

For example, how would the following code without statically nested scoping?

```
def bank_account(initial_balance):
    balance = [initial_balance]
    def deposit(amount):
        balance[0] = balance[0] + amount
        return balance
    def withdraw(amount):
        balance[0] = balance[0] - amount
        return balance
    return deposit, withdraw
```

Without nested scoping, this function would not run, because within the function deposit, balance would not be defined. The inner function deposit will not have access to the outer function bank_account's balance variable. In order for this to work without static nested scoping, we would need to redefine balance in every function following the initialization.

## Introduced changes in this PEP

PEP 227 proposed to give nested functions the scope of outer functions, which allows for variables within the parent function to be inherited by the nested function. In other words, this PEP will give nested functions the scope of parent functions, which will allow for variables within the parent function to be inherited by the nested function. Free variables are not resolved in the nearest enclosing namespace, but in the global namespace, whereas bound variables are resolved in the nearest enclosing namespace or function.

## Problems this PEP addresses: Utility

Another problem that is being addressed by this PEP is the limited utility of nested functions. Because we do not have access to outer scope variables (pre-pep), lambdas and other nested functions have to either set variables to the global scope, or redefine them in their new function.

## Example: Utility

For example, if we look at the code below:

```
from Tkinter import *
root = Tk()
Button(root, text="Click here",
    command=lambda root=root: root.test.configure(text="..."))
```

We can see that root is defined twice. This is cumbersome because we need to explicitly pass any name used in the body of the lambda must be explicitly passed as a default argument to the lambda. With nested static scoping, 'root' will be available to 'Button', therefore making it much more useful.

## Problems this PEP addresses: Non-lexical

Finally, this PEP was introduced to reduce confusion among developers who are already used to coding in a lexical fashion. Most developers are used to nested static scoping (lexical scoping) and introducing this pep will lower the barriers to entry to the python language. This addition will in turn make python a fully statically scoped language, which will effectively reduce confusion among new and old developers.

## Example: Lexical

For example, if we look at the following code:

```python
def make_adder(base):
  def adder(x):
  return base + x
  return adder
add5 = make_adder(5)
add5(6)
```

We can see that with statically nested scoping, this will return 11; we are able to take a functional approach to writing this adder because of static scoping. If we did not have lexical scoping, adder would not know what 'base' was. With the addition of this pep, it will work and return 11. Most developers would expect this to happen. Not surprisingly, this PEP was accepted in 2001, and is a vital aspect of the language.

## Namespaces

Furthermore, in order for lexical scoping to access bindings from everywhere in the code to map names to objects (where everything in Python: literals, functions, lists, classes and so on, is an object), Python will organize these names in containers called namespaces which represent the context that the names are supposed to live in. In order to avoid ambiguity, three namespaces are looked at: a local namespace, a global, and a built-in namespace.

## Local Namespaces

Firstly, the local namespace, specific to the current function or class method, is created when the function is called, and deleted (or forgotten) when the function returns or raises an exception that is not handled by a function. That is, if a function defines a local variable x, or has an argument x, Python will use variable and stop searching. For instance, variable foo defined inside of the function global, is a local or bound variable.

```python
def local(num):
    foo = num * num
    return var

local(2)
```

## Global Namespaces

Secondly, the global namespace, specific to the current module, is created as soon as the interpreter starts. If the current module has defined a variable, function, or class called x, Python will searching inside of that module and stop searching. Looking at the following code, the variable called “bar” defined outside the function classifies as a global or free variable.

```
bar = 10
def global(num)
    foo = bar * num
    return foo
global(2)
```

Thirdly, the built-in namespace is the outermost scope, which is searched last. This scope contains all the built-in names and functions such as dict(), len(), buffer(), property() and much more which are always available and contained in the Python interpreter. Naturally, we want to be able to catch built-in names from scope whether local or global. In the following example, the function builtin() knows that the name “list” refers to the the list() built-in function which naturally returns a list of elements.

```
def builtin(num)
    myList = list()
        for n in list:
            n = n * num
    return list

builtin(2)
```

## Builtin Namespaces

Moreover, these three namespaces are organized in such a way that the interpreter will search for a name depending on the scope that it lives in, in a specific order. On one hand, if a name is bound anywhere within a code block e.g. a function, all uses of the name within the block are treated as references to the current function. However, This can lead to errors when a name is used within a block before it is bound. For example, in the subsequent piece of code, the variable bar referenced within the bound() function is defined only after the bound(), outside of its scope. Consequently we will observe a “NameError: global name ‘bar’ is not defined” error.

```
def bound(num)
```

```
    foo = bar * num
    return foo
function(2)

bar = 10
```

## Name Search

On the other hand, if a global name happens in a block, e.g. outside of a function, all uses of that name will refer to the binding of that name in the top-level namespace. Thus names are resolved in the top-level namespace by searching through the global namespace, and in the builtin namespace. Effectively, the global namespace will be searched first, and if it is not found there, the builtin namespace will be searched. In the example case shown below, looking at the code, we can see that the name list will be looked up first in the local namespace of the function and then looked from outside, in the global namespace, then the builtin namespace.

```
myList = list([1,2,3])
print myList
def function(num):
  list = myList
  for x in range(0,len(list)):
    list[x] = list[x] * num
  return list # [2,4,6]
function(2)
```

We should note that if a name is used within a code block, but it is not bound there and it is not declared global, the use of that name is treated as a reference to parent scopes.

## Discussion

As described by the python enhancement proposal, “the specified rules [from the PEP] allow names defined in a function to be referenced in any nested function defined with that function” (https://www.python.org/dev/peps/pep-0227/). The python language follows these rules except for the following cases: Names in class scope, the use of the global statement, and variables not declared. Only the first two cases will be described in this paper.

The names in a class scope are omitted from the rules to prevent inconsistent behaviors when accessing class attributes and local variables. Take for example the following code:

```
my_var = 'global'
class MyClass(object):
   my_var = 'class'
   def __init__(self):
      print my_var #global
      print MyClass.my_var #class
      print self.my_var #class -- Only since we haven't set this attribute on the instance
      self.my_var = 'instance' #set instance attribute.
      print self.my_var #instance
      print MyClass.my_var #class
```

In this code it can be inferred that there is a naming conflict of my_var between the global scope and the class scope MyClass. To solve this "you can get access to it via MyClass.my_var or as self.my_var from within the class (provided you don't create an instance variable with the same name)" (http://stackoverflow.com/questions/12941748/python-variable-scope-and-classes). This PEP ignores the statically nested scope and instead will try to resolve the name in the innermost nested function. The names in the class scope are not directly accessible without the use of the self or MyClass keyword. The main reason for not allowing name bindings in class scope is because this would allow class attributes be referenced by either their simple name or the attribute reference. Allowing the simple names would make the code difficult to read and understand which variable is which.

The other exception to the rules of the statically nested scopes is the global statement. This global statement retains "the same effect that it does for Python 2.0" (https://www.python.org/dev/peps/pep-0227/). To describe the reasoning behind this see the example below:

```
myvariable = 5
def func():
    global myvariable
    myvariable = 6   #changes global scope
    print myvariable #prints 6

func()
print myvariable  #prints 6
```

Example from: http://stackoverflow.com/questions/13881395/in-python-what-is-a-global-statement

When using the declaration "global" the name that was declared now only refers to the binding in the global module namespace instead of the local binding namespace. Therefore in statically nested functions, anything that is declared with the global keyword does not refer to the whole function scope but instead to the global namespace.

## Backwards Compatibility

In terms of backwards compatibility there are two obvious problems caused by this new change. The first being different code behaviors (as seen in examples before) and then code giving syntax errors upon compilation from before and after the change. To reinforce the idea of the code behavior problem, take the following example:

```
x = 1
   def f1():
       x = 2
       def inner():
           print x
       inner()
```

Example from: https://www.python.org/dev/peps/pep-0227/

Before the changes in this PEP, this function would print 1 because the inner scope of function inner() would not inherit the scope the function f1(). After this change the function now prints 2 because it will inherit the scope of the f1() function thus changing the behavior of the code.

In terms of syntax errors being caused by the changes made in this PEP observe the following chunk of code:

```
y = 1
def f():
    exec "y = 'gotcha'" # or from module import *
    def g():
        return y
```

Example from: https://www.python.org/dev/peps/pep-0227/

After this change is implemented, during compile-time, the compiler will not be able to distinguish in g() if "y" refers to the global scoped "y" or the local scope "y" in the function f(). Therefore this will throw a compile time error because the compiler will not know which "y" to use in the function g.

## Conclusion

The changes introduced in this PEP are very beneficial to the languages as best seen in the bank account example. Without this change, to implement the bank account nested function would require a lot of workarounds such as implementing a class to allow for the scope to work. And even though nested scopes aren't as common and multi level nested scopes even less common, allowing these scopes to inherit the parent functions helps with intuitively understanding how variables are transferred.

## Sources

https://www.python.org/dev/peps/pep-0227/

http://stackoverflow.com/questions/13881395/in-python-what-is-a-global-statement

https://hg.python.org/peps/file/tip/pep-0227.txt

http://bytebaker.com/2008/07/30/python-namespaces/