PEP: 227 - Statically Nested Scopes

Louis Bouddhou, Alex Campbell, Josh Fermin

December 8, 2014

What is the problem?

- Nested functions (including lambdas) can reference variables defined in the surrouding namespace.
- Static scoping is already implemented between functions, but not within nested functions.

Example - Without Statically Nested Scopes

Louis Bouddhou, Alex Campbell, Josh Fermin

Why this wouldn't work

```
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
```

Introduced changes in this PEP

- Gives nested functions the scope of parent functions.
- This allows for variables within the parent function to be inherited by the nested function.

Problems this PEP addresses: Utility

- 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

- 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

- Most developers are used to nested static scoping (lexical scoping) and introducing this pep will lower the barriers to entry to the python language.

Example

- Now if we look at a new example, this should work with the pep implemented.
- 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.

Namespaces

- Whenever you run a simple Python 2.1 script, the interpreter treats it as a module called **main**, which gets its own namespace. However, in order to avoid ambiguity, Python stores names in the context they're supposed to live in. Those contexts are called namespaces and there are three of them. A local namespace, a Global namespace and a Builtin namespace.

Local Namespace

- The local namespace for a function 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.

Global Namespace

- The global namespace specific to the current module is created as soon as the interpreter starts. Obviously we want to make sure that any nested subroutine in the function can see names living outside of ther local namespace but in the module.

Builtin Namespace

- The built-in namespace is the outermost scope (which is searched last). This scope scope contains all the built-in names and functions of Python. Naturally, we want to be able to catch buitl-in names from scope whether local or global.
- Looking at the “list” name which will be found in the builtin namespace of Python.

Bounds

- If a name is bound anywhere within a code block, all uses of the name within the block are treated as references to the current block.
- Note: This can lead to errors when a name is used within a block before it is bound.

Name Search

- If a global name happens in a block, all uses of that name refer to the binding of that name in the top-level namespace.
- Names are resolved in the top-level namespace by searching the global namespace, and in the builtin namespace.
- The global namespace is searched first. If the name is not found there, the builtin namespace is searched. The global statement must precede all uses of the name.
- 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.

((((((If a name is used within a code block, but it is not bound there and is not declared global, the use is treated as a reference to the nearest enclosing function region. * Note: If a

Discussion

- The changes introduced by this PEP work under all circumstances except for the following cases:

1. The name in class scope
2. Global statement short-circuits the normal rules

Discussion - Name in Class Scope

- For classes, variable names will be resolved in the innermost nested function
- this is to prevent odd interactions between class attributes and local variable access.

`http://stackoverflow.com/questions/12941748/python-va`

Discussion - Short Circuit

- When the global keyword is used, the name declared by that keyword will bind to the global namespace instead of the local namespace (which it would have been bound to under this PEP's rules).
- This can be seen from the example below which allows x to refer to global namespace ‘‘‘python x = 5 def func(): global x x = 6 #changes global scope print x #prints 6

func() print x #prints 6 ‘‘‘

Problems - Backwards Compatibility

- Two kinds of compatibility problems caused:

1. Differences in code behavior before and afer the change
2. Syntax errors being caused after the change

Example - Code Behavior


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

- before this PEP it would print 1 because the inner scope would not get the scope of the f1 function.
- after this PEP it prints 2, therefore changing the behavior of code.

Example - Syntax Errors


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

- This code will through a syntax error after the change because the function g will not know which y to refer to... Either the global y variable or the local y.

Conclusion

- Changes in the pep are beneficial even though nested scopes aren't used that often.
- Only problems lie in backwards compatibility.