nonlocal keyword in Python 2.x

I'm trying to implement a closure in Python 2.6 and I need to access a nonlocal variable but it seems like this keyword is not available in python 2.x. How should one access nonlocal variables in closures in these versions of python?

python closures python-2.x python-nonlocal

edited Jul 1 '13 at 13:39
Niels Bom
2,522 3 22 37

asked Jul 6 '10 at 22:31
adinsa
351 1 3 3

10 Answers

Python can *read* nonlocal variables in 2.x, just not *change* them. This is annoying, but you can work around it. Just declare a dictionary, and store your variables as elements therein.

To use the example from Wikipedia:

```
def outer():
    d = {'y' : 0}
    def inner():
        d['y'] += 1
        return d['y']
    return inner

f = outer()
print(f(), f(), f()) #prints 1 2 3
```

answered Jul 6 '10 at 22:50
Chris B.
20.2k 7 53 92

3 why is it possible to modify the value from dictionary? – coelhudo Apr 13 '12 at 2:29

6 @coelhudo Because you **can** modify nonlocal variables. But you cannot do assignment to nonlocal variables. E.g., this will raise UnboundLocalError: `def inner(): print d; d = {'y': 1}` . Here, `print d` reads outer `d` thus creating nonlocal variable `d` in inner scope. – suzanshakya May 1 '12 at 17:52

11 Thanks for this answer. I think you could improve the terminology though: instead of "can read, cannot change", maybe "can reference, cannot assign to". You can change the contents of an object in a nonlocal scope, but you can't change which object is referred to. – metamatt Feb 4 '13 at 6:46

1 Alternatively, you can use arbitrary class and instantiate object instead of the dictionary. I find it much more elegant and readable since it does not flood the code with literals (dictionary keys), plus you can make use of methods. – Alois Mahdal Aug 15 '13 at 13:19

3 For Python I think it is best to use the verb "bind" or "rebind" and to use the noun "name" rather than "variable". In Python 2.x, you can close over an object but you can't rebind the name in the original scope. In a language like C, declaring a variable reserves some storage (either static storage or temporary on the stack). In Python, the expression `x = 1` simply binds the name `x` with a particular object (an `int` with the value `1`). `x = 1; y = x` binds two names to the same exact object. Anyway, some objects are *mutable* and you can change their values. – steveha Dec 20 '13 at 7:08

The following solution is inspired by the answer by mikez302, but contrary to that answer does handle multiple calls of the outer function correctly. The "variable" `inner.y` is local to the current call of `outer` . Only it isn't a variable, since that is forbidden, but an object attribute (the object being the function `inner` itself). This is very ugly (note that the attribute can only be created after the `inner` function is defined) but seems effective.

```
def outer():
    def inner():
        inner.y += 1
        return inner.y
    inner.y = 0
    return inner

f = outer()
g = outer()
print(f(), f(), g(), f(), g()) #prints (1, 2, 1, 3, 2)
```

answered Apr 16 '13 at 8:51
Marc van Leeuwen
1,423 7 19

It has not to be uggly. Instead of using inner.y, use outer.y. You can define outer.y = 0 before the def inner. – jgomo3 Dec 27 '13 at 14:33

Ok, I see my comment is wrong. Elias Zamaria implemented the outer.y solution also. But as is commented by Nathaniel[1], you should beware. I think this answer should be promoted as the solution, but note about the outer.y solution and cavets. [1] stackoverflow.com/questions/3190706/… – jgomo3 Dec 27 '13 at 14:44

I think the key here is what you mean by "access". There should be no issue with reading a variable outside of the closure scope, e.g.,

```
x = 3
def outer():
    def inner():
        print x
    inner()
outer()
```

should work as expected (printing 3). However, overriding the value of x does not work, e.g.,

```
x = 3
def outer():
    def inner():
        x = 5
    inner()
outer()
print x
```

will still print 3. From my understanding of PEP-3104 this is what the nonlocal keyword is meant to cover. As mentioned in the PEP, you can use a class to accomplish the same thing (kind of messy):

```
class Namespace(object): pass
ns = Namespace()
ns.x = 3
def outer():
    def inner():
        ns.x = 5
    inner()
outer()
print ns.x
```

edited Feb 7 '14 at 22:55
Scott Ritchie
4,617 1 6 35

answered Jul 6 '10 at 22:51
ig0774
12.5k 2 28 37

Instead of creating a class and instantiating it, one can simply create a function: `def ns(): pass` followed by `ns.x = 3` . It ain't pretty, but it's slightly less ugly to my eye. – davidchambers Jan 25 '11 at 11:36

2 Any particular reason for the downvote? I admit mine is not the most elegant solution, but it works... – ig0774 Mar 13 '13 at 12:25

The second code-sample does not work simply because assignment to a name (variable) that does not exist in the current scope will result in the creation of a local variable with the assigned value. Right? I would not find other behavior logical. – Niels Bom Jul 1 '13 at 13:48

2 What about `class Namespace: x = 3` ? – Feuermurmel Feb 18 '14 at 20:28

1 I don't think this way simulates nonlocal, since it creates a global reference instead of a reference in a closure. – CarmeloS Nov 24 '14 at 2:43

There is another way to implement nonlocal variables in Python 2, in case any of the answers here are undesirable for whatever reason:

```
def outer():
    outer.y = 0
    def inner():
        outer.y += 1
        return outer.y
    return inner
```

```
f = outer()
print(f(), f(), f()) #prints 1 2 3
```

It is redundant to use the name of the function in the assignment statement of the variable, but it looks simpler and cleaner to me than putting the variable in a dictionary. The value is remembered from one call to another, just like in Chris B.'s answer.

answered Dec 10 '12 at 3:25

Elias Zamaria
19.6k 20 66 106

The most clear. – Jimmy Kane Jan 17 '13 at 17:52

12 Please beware: when implemented this way, if you do `f = outer()` and then later do `g = outer()`, then `f`'s counter will be reset. This is because they both share a *single* `outer.y` variable, rather than each having their own independent one. Although this code looks more aesthetically pleasing than Chris B's answer, his way seems to be the only way to emulate lexical scoping if you want to call `outer` more than once. – Nathaniel Mar 14 '13 at 3:57

@Nathaniel: Let me see if I understand this correctly. The assignment to `outer.y` does not involve anything local to the function call (instance) `outer()`, but assigns to an attribute of the function object that is bound to the name `outer` in its *enclosing* scope. And therefore one could equally well have used, in writing `outer.y`, *any other* name instead of `outer`, provided it is known to be bound in that scope. Is this correct? – Marc van Leeuwen Apr 16 '13 at 8:13

1 Correction, I should have said, after "bound in that scope": to an object whose type allows setting attributes (like a function, or any class instance). Also, since this scope is actually further out than the one we want, wouldn't this suggest the following extremely ugly solution: instead of `outer.y` use the name `inner.y` (since `inner` is bound inside the call `outer()`, which is exactly the scope we want), but putting the initialisation `inner.y = 0` *after* the definition of *inner* (as the object must exist when its attribute is created), but of course before `return inner` ? – Marc van Leeuwen Apr 16 '13 at 8:37

@MarcvanLeeuwen Looks like your comment inspired the solution with inner.y by Elias. Good thought by you. – abc Jul 20 at 5:21

There is a wart in python's scoping rules - assignment makes a variable local to its immediately enclosing function scope. For a global variable, you would solve this with the `global` keyword.

The solution is to introduce an object which is shared between the two scopes, which contains mutable variables, but is itself referenced through a variable which is not assigned.

```
def outer(v):
    def inner(container = [v]):
        container[0] += 1
        return container[0]
    return inner
```

An alternative is some scopes hackery:

```
def outer(v):
    def inner(varname = 'v', scope = locals()):
        scope[varname] += 1
        return scope[varname]
    return inner
```

You might be able to figure out some trickery to get the name of the parameter to `outer`, and then pass it as varname, but without relying on the name `outer` you would like need to use a Y combinator.

answered May 1 '12 at 17:08

Marcin
28.1k 9 56 115

Here's something inspired by a suggestion Alois Mahdal made in a comment regarding another answer:

```
class Nonlocals(object):
    """ Helper to implement nonlocal names in Python 2.x """
    def __init__(self, **kwargs):
        self.__dict__.update(kwargs)

def outer():
    nl = Nonlocals(y=0)
    def inner():
        nl.y += 1
        return nl.y
    return inner

f = outer()
print(f(), f(), f()) # -> (1 2 3)
```

edited Feb 16 '14 at 14:15 answered Feb 16 '14 at 14:09


martineau
30.8k 6 38 71

This one seems to be the best in terms of readability and preserving lexical scoping. – Aaron S. Kurland Oct 8 '14 at 3:17

Rather than a dictionary, there's less clutter to a **nonlocal class**. Modifying @ChrisB's example:

```
def outer():
    class context:
        y = 0
    def inner():
        context.y += 1
        return context.y
    return inner
```

then

```
f = outer()
assert f() == 1
assert f() == 2
assert f() == 3
```

answered Feb 10 at 14:02


BobStein-VisiBone
868 1 12 21

Another way to do it (although it's too verbose):

```
import ctypes

def outer():
    y = 0
    def inner():
        ctypes.pythonapi.PyCell_Set(id(inner.func_closure[0]), id(y + 1))
        return y
    return inner

x = outer()
x()
>> 1
x()
>> 2
y = outer()
y()
>> 1
x()
>> 3
```

answered Jan 10 '14 at 16:37


Ezer Fernandes
38 6

1 actually I prefer the solution using a dictionary, but this one is cool :) – Ezer Fernandes Jan 10 '14 at 16:39

Extending Martineau elegant solution above to a practical and somewhat less elegant use case I get:

```
class nonlocals(object):
""" Helper to implement nonlocal names in Python 2.x.
Usage example:
def outer():
     nl = nonlocals( n=0, m=1 )
     def inner():
         nl.n += 1
     inner() # will increment nl.n

or...
    sums = nonlocals( { k:v for k,v in locals().iteritems() if k.startswith('tot_') } )
"""
def __init__(self, **kwargs):
    self.__dict__.update(kwargs)

def __init__(self, a_dict):
    self.__dict__.update(a_dict)
```

answered Jan 12 at 10:57

Amnon Harel
12 1

Use a global variable

```
def outer():
    global y # import1
    y = 0
    def inner():
        global y # import2 - requires import1
        y += 1
        return y
    return inner

f = outer()
print(f(), f(), f()) #prints 1 2 3
```

Personally, I do not like the global variables. But, my proposal is based on http://stackoverflow.com/a/19877437/1083704 answer

```
def report():
        class Rank:
            def __init__(self):
                report.ranks += 1
        rank = Rank()
report.ranks = 0
report()
```

where user needs to declare a global variable `ranks`, every time you need to call the `report`.
My improvement eliminates the need to initialize the function variables from the user.

answered Nov 10 '13 at 12:59

Val
1

It is much better to use a dictionary. You can reference the instance in `inner`, but can't assign to it, but you can modify it's keys and values. This avoids use of global variables. – johannestaas Feb 7 '14 at 22:43