Variable Scope and Binding All Versions

Examples

Nonlocal Variables

Python 3.x ≥3.0

Python 3 added a new keyword called **nonlocal** . The nonlocal keyword adds a scope override to the inner scope. You can read all about it in PEP 3104 . This is best illustrated with a couple of code examples. One of the most common examples is to create function that can increment:

```
def counter():
    num = 0
    def incrementer():
        num += 1
        return num
    return incrementer
```

If you try running this code, you will receive an **UnboundLocalError** because the **num** variable is referenced before it is assigned in the innermost function. Let's add nonlocal to the mix:

```
def counter():
    num = 0
    def incrementer():
        nonlocal num
        num += 1
        return num
    return incrementer

c = counter()
c() # = 1
c() # = 2
c() # = 3
```

Basically `nonlocal` will allow you to assign to variables in an outer scope, but not a global scope. So you can't use `nonlocal` in our `counter` function because then it would try to assign to a global scope. Give it a try and you will quickly get a `SyntaxError` . Instead you must use `nonlocal` in a nested function.

(Note that the functionality presented here is better implemented using generators.)

Local Variables

If a name is *bound* inside a function, it is by default accessible only within the function:

```
def foo():
    a = 5
    print(a) # ok

print(a) #  NameError: name 'a' is not defined
```

Control flow constructs have no impact on the scope (with the exception of `except`), but accessing variable that was not assigned yet is an error:

```
def foo():
    if True:
        a = 5
    print(a) # ok

b = 3
def bar():
    if False:
        b = 5
    print(b) # UnboundLocalError: local variable 'b' referenced before assignment
```

Common binding operations are assignments, `for` loops, and augmented assignments such as `a += 5`

Binding Occurrence

```
x = 5
x += 7
for x in iterable: pass
```

Each of the above statements is a *binding occurrence* - `x` become bound to the object denoted by `5`. If this statement appears inside a function, then `x` will be function-local by default. See the "Syntax" section for a list of binding statements.

Functions skip class scope when looking up names

Classes have a local scope during definition, but functions inside the class do not use that scope when looking up names. Because lambdas are functions, and comprehensions are implemented using function scope, this can lead to some surprising behavior.

```
a = 'global'

class Fred:
    a = 'class'  # class scope
    b = (a for i in range(10))  # function scope
    c = [a for i in range(10)]  # function scope
    d = a  # class scope
    e = lambda: a  # function scope
    f = lambda a=a: a  # default argument uses class scope

    @staticmethod  # or @classmethod, or regular instance method
    def g():  # function scope
        return a

print(Fred.a)  # class
print(next(Fred.b))  # global
print(Fred.c[0])  # class in Python 2, global in Python 3
print(Fred.d)  # class
print(Fred.e())  # global
print(Fred.f())  # class
print(Fred.g()) # global
```

Users unfamiliar with how this scope works might expect `b`, `c`, and `e` to print `class`.

From PEP 227:

> Names in class scope are not accessible. Names are resolved in the innermost enclosing function scope. If a class definition occurs in a chain of nested scopes, the resolution process skips class definitions.

From Python's documentation on naming and binding:

> The scope of names defined in a class block is limited to the class block; it does not extend to the code blocks of methods – this includes comprehensions and generator expressions since they are implemented using a function scope. This means that the following will fail:
>
> ```
> class A:
>     a = 42
>     b = list(a + i for i in range(10))
> ```

This example uses references from this answer by Martijn Pieters, which contains more in depth analysis of this behavior.

Global Variables

In Python, variables that are only referenced inside a function are implicitly global. If a variable is assigned a value anywhere within the function's body, it's assumed to be a local unless explicitly declared as global.

```
x = 'Hi'

def read_x():
    print(x)

read_x()  # prints Hi
```

Normally, an assignment inside a scope will shadow any outer variables of the same name:

```
x = 'Hi'

def change_local_x():
    x = 'Bye'
    print(x)
```

```
change_local_x()  # prints Bye
print(x)  # prints Hi
```

Declaring a name `global` means that, for the rest of the scope, any assignments to the name will happen at the module's top level:

```
x = 'Hi'

def change_global_x():
    global x
    x = 'Bye'
    print(x)

change_global_x()  # prints Bye
print(x)  # prints Bye
```

The `global` keyword means that assignments will happen at the module's top level, not at the program's top level. Other modules will still need the usual dotted access to variables within the module.

Local vs Global Scope

What are local and global scope?

All Python variabes which are accessible at some point in code are either in *local scope* or in *global scope* .

The explanation is that local scope includes all variables defined in the current function and global scope includes variabled defined outside of the current function.

```
foo = 1  # global

def func():
    bar = 2  # local
    print(foo)  # prints variable foo from global scope
    print(bar)  # prints variable bar from local scope
```

One can inspect which variables are in which scope. Built-in functions `locals()` and `globals()` return the whole scopes as dictionaries.

```
foo = 1

def func():
    bar = 2
    print(globals().keys())  # prints all variable names in global scope
    print(locals().keys())  # prints all variable names in local scope
```

What happens with name clashes?

```
foo = 1

def func():
    foo = 2  # creates a new variable foo in local scope, global foo is not affected

    print(foo)  # prints 2

    # global variable foo still exists, unchanged:
    print(globals()['foo'])  # prints 1
    print(locals()['foo'])  # prints 2
```

To modify a global variable, use keyword `global` :

```
foo = 1

def func():
    global foo
    foo = 2  # this modifies the global foo, rather than creating a local variable
```

The scope is defined for the whole body of the function!

What it means is that a variable will never be global for a half of the function and local afterwards, or vice-versa.

```
foo = 1

def func():
    # This function has a local variable foo, because it is defined down below.
    # So, foo is local from this point. Global foo is hidden.

    print(foo) # raises UnboundLocalError, because local foo is not yet initialized
    foo = 7
    print(foo)
```

Likewise, the oposite:

```
foo = 1

def func():
    # In this function, foo is a global variable from the begining

    foo = 7  # global foo is modified

    print(foo)  # 7
    print(globals()['foo'])  # 7

    global foo  # this could be anywhere within the function
    print(foo)  # 7
```

Functions within functions

There may be many levels of functions nested within functions, but within any one function there is only one local scope for that function and the global scope. There are no intermediate scopes.

```
foo = 1

def f1():
    bar = 1

    def f2():
        baz = 2
        # here, foo is a global variable, baz is a local variable
        # bar is not in either scope
        print(locals().keys())  # ['baz']
        print('bar' in locals())  # False
        print('bar' in globals())  # False

    def f3():
        baz = 3
        print(bar)  # bar from f1 is referenced so it enters local scope of f3 (closure)
        print(locals().keys())  # ['bar', 'baz']
        print('bar' in locals())  # True
        print('bar' in globals())  # False

    def f4():
        bar = 4  # a new local bar which hides bar from local scope of f1
        baz = 4
        print(bar)
        print(locals().keys())  # ['bar', 'baz']
        print('bar' in locals())  # True
        print('bar' in globals())  # False
```

`global` vs `nonlocal` (Python 3 only)

Both these keywords are used to gain write access to variables which are not local to the current functions.

The `global` keyword declares that a name should be treated as a global variable.

```
foo = 0  # global foo

def f1():
    foo = 1  # a new foo local in f1

    def f2():
        foo = 2  # a new foo local in f2

        def f3():
            foo = 3  # a new foo local in f3
            print(foo)  # 3
            foo = 30  # modifies local foo in f3 only

        def f4():
            global foo
            print(foo)  # 0
            foo = 100  # modifies global foo
```

On the other hand, `nonlocal` (see Nonlocal Variables), available in Python 3, takes a *local* variable from an enclosing scope into the local scope of current function.

From the Python documentation on `nonlocal` :

> The nonlocal statement causes the listed identifiers to refer to previously bound variables in the nearest enclosing scope excluding globals.

Python 3.x ≥3.0

```
def f1():

    def f2():
        foo = 2  # a new foo local in f2

        def f3():
```

```
        nonlocal foo  # foo from f2, which is the nearest enclosing scope
        print(foo)  # 2
        foo = 20  # modifies foo from f2!
```

The del command

This command has several related yet distinct forms.

del v

If `v` is a variable, the command `del v` removes the variable from its scope. For example:

```
x = 5
print(x) # out: 5
del x
print(x) # NameError: name 'f' is not defined
```

> Note that `del` is a *binding occurence* , which means that unless explicitly stated otherwise (using `nonlocal` or `global`), `del v` will make `v` local to the current scope. If you intend to delete `v` in an outer scope, use `nonlocal v` or `global v` in the same scope of the `del v` statement.

In all the following, the intention of a command is a default behavior but is not enforced by the language. A class might be written in a way that invalidates this intention.

del v.name

This command triggers a call to `v.__delattr__(name)` .

The intention is to make the attribute `name` unavailable. For example:

```
class A:
    pass

a = A()
a.x = 7
print(a.x) # out: 7
del a.x
print(a.x) # error: AttributeError: 'A' object has no attribute 'x'
```

del v[item]

This command triggers a call to `v.__delitem__(item)` .

The intention is that `item` will not belong in the mapping implemented by the object `v` . For example:

```
x = {'a': 1, 'b': 2}
del x['a']
print(x) #  out: {'b': 2}
print(x['a']) # error: KeyError: 'a'
```

del v[a:b]

This actually calls `v.__delslice__(a, b)` .

The intention is similar to the one described above, but with slices - ranges of items instead of a single item. For example:

```
x = [0, 1, 2, 3, 4]
del x[1:3]
print(x) #  out: [0, 3, 4]
```

See also Garbage Collection#The del command .

Syntax

```
global a, b, c
```

```
nonlocal a, b
```

```
x = something # binds x
```

```
(x, y) = something # binds x and y
```

```
x += something # binds x. Similarly for all other "op="
del x # binds x
for x in something: # binds x
with something as x: # binds x
except Exception as ex: # binds ex inside block
```

Parameters

Remarks