[RSS]

A Beginner's Guide to Python's Namespaces, Scope Resolution, and the LEGB Rule

May 12, 2014
by Sebastian Raschka

This is a short tutorial about Python's namespaces and the scope resolution for variable names using the LEGB-rule. The following sections will provide short example code blocks that should illustrate the problem followed by short explanations. You can simply read this tutorial from start to end, but I'd like to encourage you to execute the code snippets - you can either copy & paste them, or for your convenience, simply download this IPython notebook.

Sections

Objectives

- Namespaces and scopes - where does Python look for variable names?
- Can we define/reuse variable names for multiple objects at the same time?
- In which order does Python search different namespaces for variable names?

Introduction to namespaces and scopes

Namespaces

Roughly speaking, namespaces are just containers for mapping names to objects. As you might have already heard, everything in Python - literals, lists, dictionaries, functions, classes, etc. - is an object.
Such a “name-to-object” mapping allows us to access an object by a name that we’ve assigned to it. E.g., if we make a simple string assignment via `a_string = "Hello string"`, we created a reference to the `"Hello string"` object, and henceforth we can access via its variable name `a_string`.

We can picture a namespace as a Python dictionary structure, where the dictionary keys represent the names and the dictionary values the object itself (and this is also how namespaces are currently implemented in Python), e.g.,

```
a_namespace = {'name_a':object_1, 'name_b':object_2, ...}
```

Now, the tricky part is that we have multiple independent namespaces in Python, and names can be reused for different namespaces (only the objects are unique, for example:

```
a_namespace = {'name_a':object_1, 'name_b':object_2, ...}
```

```
a_namespace = {'name_a':object_1, 'name_b':object_2, ...}
b_namespace = {'name_a':object_3, 'name_b':object_4, ...}
```

For example, everytime we call a `for-loop` or define a function, it will create its own namespace. Namespaces also have different levels of hierarchy (the so-called “scope”), which we will discuss in more detail in the next section.

Scope

In the section above, we have learned that namespaces can exist independently from each other and that they are structured in a certain hierarchy, which brings us to the concept of “scope”. The “scope” in Python defines the “hierarchy level” in which we search namespaces for certain “name-to-object” mappings.
For example, let us consider the following code:

```
i = 1

def foo():
    i = 5
    print(i, 'in foo()')

print(i, 'global')

foo()
```

```
1 global
5 in foo()
```

Here, we just defined the variable name `i` twice, once on the `foo` function.

- `foo_namespace = {'i':object_3, ...}`
- `global_namespace = {'i':object_1, 'name_b':object_2, ...}`

So, how does Python know which namespace it has to search if we want to print the value of the variable `i` ? This is where Python’s LEGB-rule comes into play, which we will discuss in the next section.

Tip:

If we want to print out the dictionary mapping of the global and local variables, we can use the the functions `global()` and `local()`

```
#print(globals()) # prints global namespace
#print(locals()) # prints local namespace

glob = 1

def foo():
    loc = 5
    print('loc in foo():', 'loc' in locals())

foo()
print('loc in global:', 'loc' in globals())
print('glob in global:', 'foo' in globals())
```

```
loc in foo(): True
loc in global: False
glob in global: True
```

Scope resolution for variable names via the LEGB rule.

We have seen that multiple namespaces can exist independently from each other and that they can contain the same variable names on different hierachy levels. The “scope” defines on which hierarchy level Python searches for a particular “variable name” for its associated object. Now, the next question is: “In which order does Python search the different levels of namespaces before it finds the name-to-object’ mapping?”
To answer is: It uses the LEGB-rule, which stands for

Local -> Enclosed -> Global -> Built-in,

where the arrows should denote the direction of the namespace-hierarchy search order.

- *Local* can be inside a function or class method, for example.
- *Enclosed* can be its `enclosing` function, e.g., if a function is wrapped inside another function.
- *Global* refers to the uppermost level of the executing script itself, and
- *Built-in* are special names that Python reserves for itself.

So, if a particular name:object mapping cannot be found in the local namespaces, the namespaces of the enclosed scope are being searched next. If the search in the enclosed scope is unsuccessful, too, Python moves on to the global namespace, and eventually, it will search the built-in namespace (side note: if a name cannot found in any of the namespaces, a *NameError* will is raised).

Note:

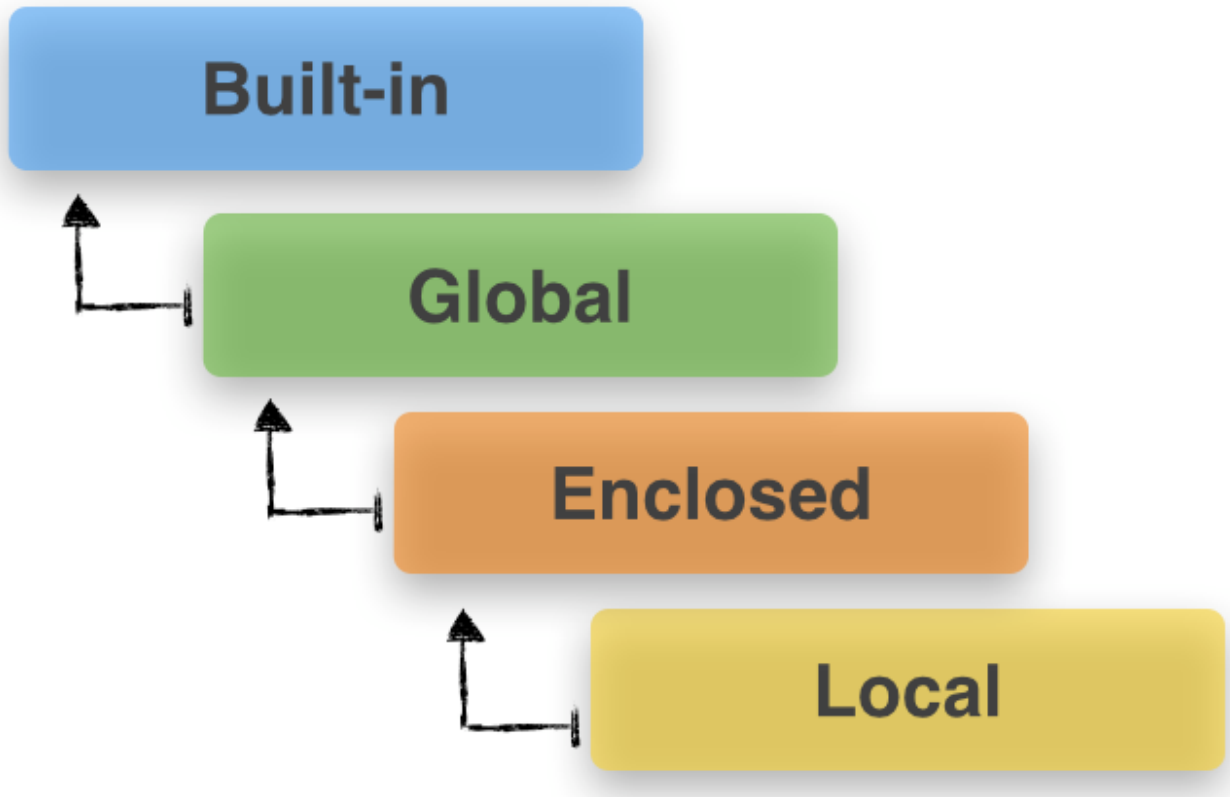
Namespaces can also be further nested, for example if we import modules, or if we are defining new classes. In those cases we have to use prefixes to access those nested namespaces. Let me illustrate this concept in the following code block:

```
import numpy
import math
import scipy

print(math.pi, 'from the math module')
print(numpy.pi, 'from the numpy package')
print(scipy.pi, 'from the scipy package')
```

```
3.141592653589793 from the math module
3.141592653589793 from the numpy package
3.141592653589793 from the scipy package
```

(This is also why we have to be careful if we import modules via " `from a_module import *` ", since it loads the variable names into the global namespace and could potentially overwrite already existing variable names)



1. LG - Local and Global scopes

Example 1.1

As a warm-up exercise, let us first forget about the enclosed (E) and built-in (B) scopes in the LEGB rule and only take a look at LG - the local and global scopes.
What does the following code print?

```
a_var = 'global variable'

def a_func():
    print(a_var, '[ a_var inside a_func() ]')

a_func()
print(a_var, '[ a_var outside a_func() ]')
```

a)

```
raises an error
```

b)

```
global value [ a_var outside a_func() ]
```

c)

```
global value [ a_var inside a_func() ]
global value [ a_var outside a_func() ]
```

Here is why:

We call `a_func()` first, which is supposed to print the value of `a_var`. According to the LEGB rule, the function will first look in its own local scope (L) if `a_var` is defined there. Since `a_func()` does not define its own `a_var`, it will look one-level above in the global scope (G) in which `a_var` has been defined previously.

Example 1.2

Now, let us define the variable `a_var` in the global and the local scope.
Can you guess what the following code will produce?

```
a_var = 'global value'

def a_func():
    a_var = 'local value'
    print(a_var, '[ a_var inside a_func() ]')

a_func()
print(a_var, '[ a_var outside a_func() ]')
```

a)

```
raises an error
```

b)

```
local value [ a_var inside a_func() ]
global value [ a_var outside a_func() ]
```

c)

```
global value [ a_var inside a_func() ]
global value [ a_var outside a_func() ]
```

Here is why:

When we call `a_func()`, it will first look in its local scope (L) for `a_var`, since `a_var` is defined in the local scope of `a_func`, its assigned value `local variable` is printed. Note that this doesn't affect the global variable, which is in a different scope.

However, it is also possible to modify the global by, e.g., re-assigning a new value to it if we use the global keyword as the following example will illustrate:

```
a_var = 'global value'

def a_func():
    global a_var
    a_var = 'local value'
    print(a_var, '[ a_var inside a_func() ]')

print(a_var, '[ a_var outside a_func() ]')
a_func()
print(a_var, '[ a_var outside a_func() ]')
```

```
global value [ a_var outside a_func() ]
local value [ a_var inside a_func() ]
local value [ a_var outside a_func() ]
```

But we have to be careful about the order: it is easy to raise an `UnboundLocalError` if we don't explicitly tell Python that we want to use the global scope and try to modify a variable's value (remember, the right side of an assignment operation is executed first):

```
a_var = 1

def a_func():
    a_var = a_var + 1
    print(a_var, '[ a_var inside a_func() ]')

print(a_var, '[ a_var outside a_func() ]')
a_func()
```

```
---------------------------------------------------------------------------
UnboundLocalError                         Traceback (most recent call last)

<ipython-input-4-a6cdd0ee9a55> in <module>()
      6
      7 print(a_var, '[ a_var outside a_func() ]')
----> 8 a_func()


<ipython-input-4-a6cdd0ee9a55> in a_func()
      2
      3 def a_func():
----> 4     a_var = a_var + 1
      5     print(a_var, '[ a_var inside a_func() ]')
      6


UnboundLocalError: local variable 'a_var' referenced before assignment


1 [ a_var outside a_func() ]
```

2. LEG - Local, Enclosed, and Global scope

Now, let us introduce the concept of the enclosed (E) scope. Following the order “Local -> Enclosed -> Global”, can you guess what the following code will print?

Example 2.1

```
a_var = 'global value'

def outer():
    a_var = 'enclosed value'

    def inner():
        a_var = 'local value'
        print(a_var)

    inner()

outer()
```

a)

```
global value
```

b)

```
enclosed value
```

c)

```
local value
```

Here is why:

Let us quickly recapitulate what we just did: We called `outer()`, which defined the variable `a_var` locally (next to an existing `a_var` in the global scope). Next, the `outer()` function called `inner()`, which in turn defined a variable with of name `a_var` as well. The `print()` function inside `inner()` searched in the local scope first (L->E) before it went up in the scope hierarchy, and therefore it printed the value that was assigned in the local scope.

Similar to the concept of the `global` keyword, which we have seen in the section above, we can use the keyword `nonlocal` inside the inner function to explicitly access a variable from the outer (enclosed) scope in order to modify its value.

Note that the `nonlocal` keyword was added in Python 3.x and is not implemented in Python 2.x (yet).

```
a_var = 'global value'

def outer():
       a_var = 'local value'
       print('outer before:', a_var)
       def inner():
           nonlocal a_var
           a_var = 'inner value'
           print('in inner():', a_var)
       inner()
       print("outer after:", a_var)
outer()
```

```
outer before: local value
in inner(): inner value
outer after: inner value
```

3. LEGB - Local, Enclosed, Global, Built-in

To wrap up the LEGB rule, let us come to the built-in scope. Here, we will define our "own" length-funcion, which happens to bear the same name as the in-built `len()` function. What outcome do you excpect if we'd execute the following code?

Example 3

```
a_var = 'global variable'

def len(in_var):
    print('called my len() function')
    l = 0
    for i in in_var:
        l += 1
    return l

def a_func(in_var):
    len_in_var = len(in_var)
    print('Input variable is of length', len_in_var)

a_func('Hello, World!')
```

a)

```
raises an error (conflict with in-built `len()` function)
```

b)

```
called my len() function
Input variable is of length 13
```

c)

```
Input variable is of length 13
```

Here is why:

Since the exact same names can be used to map names to different objects - as long as the names are in different name spaces - there is no problem of reusing the name `len` to define our own length function (this is just for demonstration pruposes, it is NOT recommended). As we go up in Python's L -> E -> G -> B hierarchy, the function `a_func()` finds `len()` already in the global scope (G) first before it attempts to search the built-in (B) namespace.

Self-assessment exercise

Now, after we went through a couple of exercises, let us quickly check where we are. So, one more time: What would the following code print out?

```
a = 'global'

def outer():

    def len(in_var):
        print('called my len() function: ', end="")
        l = 0
        for i in in_var:
            l += 1
        return l

    a = 'local'
```

```
    def inner():
        global len
        nonlocal a
        a += ' variable'
    inner()
    print('a is', a)
    print(len(a))


outer()

print(len(a))
print('a is', a)
```

Conclusion

I hope this short tutorial was helpful to understand the basic concept of Python's scope resolution order using the LEGB rule. I want to encourage you (as a little self-assessment exercise) to look at the code snippets again tomorrow and check if you can correctly predict all their outcomes.

A rule of thumb

In practice, **it is usually a bad idea to modify global variables inside the function scope**, since it often be the cause of confusion and weird errors that are hard to debug.
If you want to modify a global variable via a function, it is recommended to pass it as an argument and reassign the return-value.
For example:

```
a_var = 2

def a_func(some_var):
    return 2**3

a_var = a_func(a_var)
print(a_var)
```

```
8
```

Solutions

In order to prevent you from unintentional spoilers, I have written the solutions in binary format. In order to display the character representation, you just need to execute the following lines of code:

```
print('Example 1.1:', chr(int('01100011',2)))
```

```
print('Example 1.2:', chr(int('01100010',2)))
```

```
print('Example 2.1:', chr(int('01100011',2)))
```

```
print('Example 3.1:', chr(int('01100010',2)))
```

```
# Execute to run the self-assessment solution

sol = "0000101001101111011101010111010001100101011100100010100"\
"0000101001001110100000101000001010011000010010000001101001011100110"\
"0100000011011000110111101100011011000010110110000010000001110110011 0"\
"0001011100100110100101100000101100010011011000110010100001010011000 1"\
"1011000010110110001101100011001010110010000100000011011010111100100"\
"1000000110110001100101011011100010100000101001001000000110011001110"\
"1010110111001100011011101000110100101101111011011100011101000100000"\
"0011000100110100000010100000101001100111011011000110111110110001001 1"\
"0000101101100001110100000101000001010001101100000010100110000100100 0"\
"0001101001011100110010000001100111011011000110111101100010011000010"\
"1101100"

sol_str =''.join(chr(int(sol[i:i+8], 2)) for i in range(0, len(sol), 8))
for line in sol_str.split('\n'):
    print(line)
```

Warning: For-loop variables “leaking” into the global namespace

In contrast to some other programming languages, `for-loops` will use the scope they exist in and leave their defined loop-variable behind.

```
for a in range(5):
    if a == 4:
        print(a, '-> a in for-loop')
print(a, '-> a in global')
```

```
4 -> a in for-loop
4 -> a in global
```

This also applies if we explicitly defined the `for-loop` variable in the global namespace before! In this case it will rebind the existing variable:

```
b = 1
for b in range(5):
    if b == 4:
        print(b, '-> b in for-loop')
print(b, '-> b in global')
```

```
4 -> b in for-loop
4 -> b in global
```

However, in **Python 3.x**, we can use closures to prevent the for-loop variable to cut into the global namespace. Here is an example (exectuted in Python 3.4):

```
i = 1
print([i for i in range(5)])
print(i, '-> i in global')
```

```
[0, 1, 2, 3, 4]
1 -> i in global
```

Why did I mention “Python 3.x”? Well, as it happens, the same code executed in Python 2.x would print:

```
4 -> i in global
```

```
This goes back to a change that was made in Python 3.x and is described in [What's

"List comprehensions no longer support the syntactic form `[... for var in item1, i
```

Have feedback on this post? I would love to hear it. Let me know and send me a tweet or email.