**Variable Scope**

The existence of multiple, distinct namespaces means several different instances of a particular name can exist simultaneously while a Python program runs. As long as each instance is in a different namespace, they’re all maintained separately and won’t interfere with one another.

But that raises a question: Suppose you refer to the name x in your code, and x exists in several namespaces. How does Python know which one you mean?

The answer lies in the concept of scope. The scope of a name is the region of a program in which that name has meaning. The interpreter determines this at runtime based on where the name definition occurs and where in the code the name is referenced.

Interesting! But how does Python know which **x** you're referring to in your code? This is where the idea of scope comes in. Python has a set of rules it follows to decide what variables (such as **x** in this case) you are referencing in your code. Lets break down the rules:

This idea of scope in your code is very important to understand in order to properly assign and call variable names.

In simple terms, the idea of scope can be described by 3 general rules:

1. Name assignments will create or change local names by default.
2. Name references search (at most) four scopes, these are:
   * local
   * enclosing functions
   * global
   * built-in
3. Names declared in global and nonlocal statements map assigned names to enclosing module and function scopes.

The statement in #2 above can be defined by the LEGB rule.

**LEGB Rule:**

L: Local — Names assigned in any way within a function (def or lambda), and not declared global in that function.

E: Enclosing function locals — Names in the local scope of any and all enclosing functions (def or lambda), from inner to outer.

G: Global (module) — Names assigned at the top-level of a module file, or declared global in a def within the file.

B: Built-in (Python) — Names preassigned in the built-in names module : open, range, SyntaxError,...

**Local**

*# x is local here:*

f **=** **lambda** x:x**\*\***2

**Enclosing function locals**

This occurs when we have a function inside a function (nested functions)

name **=** 'This is a global name'

​

**def** greet():

*# Enclosing function*

*# name = 'Sammy'*

**def** hello():

*# name = "Asha"*

print('Hello '**+**name)

hello()

​

greet()

​

print(name)

Hello Asha

This is a global name

Note how Sammy was used, because the hello() function was enclosed inside of the greet function!

**Global**

The global namespace contains any names defined at the level of the main program. Python creates the global namespace when the main program body starts, and it remains in existence until the interpreter terminates.

Strictly speaking, this may not be the only global namespace that exists. The interpreter also creates a global namespace for any module that your program loads with the import statement.

*# dir(globals()) #globals present in the current module*

*# type(globals())*

*# print(globals())*

print(name)

This is a global name

**Built-in**

These are the built-in function names in Python (don't overwrite these!)

len

​

*# dir(\_\_builtins\_\_) #To find the available builtins we can use dir(\_\_builtins\_\_)*

​

The Python interpreter creates the built-in namespace when it starts up. This namespace remains in existence until the interpreter terminates

**Local Variables**

When you declare variables inside a function definition, they are not related in any way to other variables with the same names used outside the function - i.e. variable names are local to the function. This is called the scope of the variable. All variables have the scope of the block they are declared in starting from the point of definition of the name.

The interpreter creates a new namespace whenever a function executes. That namespace is local to the function and remains in existence until the function terminates.

Example:

**Modify Variables Out of Scope**

Earlier on user-defined Python functions, you learned that argument passing in Python is a bit like pass-by-value and a bit like pass-by-reference. Sometimes a function can modify its argument in the calling environment by making changes to the corresponding parameter, and sometimes it can’t:

An immutable argument can never be modified by a function. A mutable argument can’t be redefined wholesale, but it can be modified in place

​

x **=** 50

​

**def** func(x):

print('x is', x) *#search will happen in local since still it is not created in local scope it will check in global scope*

x **=** 2 *#new object is created and it is local , so when referenced in local scope it will access local variable*

print('Changed local x to', x)

​

func(x)

print('x is still', x) *#still global*

x is 50

Changed local x to 2

x is still 50

A similar situation exists when a function tries to modify a variable outside its local scope. A function can’t modify an immutable object outside its local scope at all

A function can modify an object of mutable type that’s outside its local scope

​

​

​

*# Try modifying global immutable variable within local scope , we will get an error as modifying is not possible*

max**=** 50

​

​

​

**def** func():

print("The value of max",max)

max **=** max**+**1 *#will lead to error if we modify global variable in local scope it will give error*

print("The value of max after modification",max)

​

​

​

print(id(max))

func()

max **=** max**+**1 *#in global scope you can modify*

print(id(max))

print(max)

1585473392

**---------------------------------------------------------------------------**

**UnboundLocalError** Traceback (most recent call last)

**<ipython-input-2-ffe12c46644e>** in <module>

14

15 print**(**id**(**max**))**

**---> 16** func**()**

17 max **=** max**+1** **#in global scope you can modify**

18 print**(**id**(**max**))**

**<ipython-input-2-ffe12c46644e>** in func**()**

6 **def** func**():**

7

**----> 8** print**("The value of max",**max**)**

9 max **=** max**+1** **#will lead to error if we modify global variable in local scope it will give error**

10 print**("The value of max after modification",**max**)**

**UnboundLocalError**: local variable 'max' referenced before assignment

​

*#Try modifying Global variable of mutable type with in local scope , there will not be an error and it will get modified*

​

​

lst **=** [1,2,3]

​

**def** func():

​

print("Before modification",lst)

lst.append(10)

print("After modification",lst)

​

func()

print(lst)

​

Before modification [1, 2, 3]

After modification [1, 2, 3, 10]

[1, 2, 3, 10]

The first time that we print the value of the name **x** with the first line in the function’s body, Python uses the value of the parameter declared in the main block, above the function definition.

Next, we assign the value 2 to **x**. The name **x** is local to our function. So, when we change the value of **x** in the function, the **x** defined in the main block remains unaffected.

With the last print statement, we display the value of **x** as defined in the main block, thereby confirming that it is actually unaffected by the local assignment within the previously called function.

to avoid this error we go for using global keyword , which will allow us to modify the value in the global namespace

​

**The global statement**

If you want to assign a value to a name defined at the top level of the program (i.e. not inside any kind of scope such as functions or classes), then you have to tell Python that the name is not local, but it is global. We do this using the global statement. It is impossible to assign a value to a variable defined outside a function without the global statement.

You can use the values of such variables defined outside the function (assuming there is no variable with the same name within the function). However, this is not encouraged and should be avoided since it becomes unclear to the reader of the program as to where that variable’s definition is. Using the global statement makes it amply clear that the variable is defined in an outermost block.

Also it is making us know that we are indeed trying to modify the global varibale only and the local variable

Example:

x **=** 50

​

​

**def** func():

**global** x *#you are indicating that we are using global variable with in local scope*

print('This function is now using the global x!')

print('Because of global x is: ', x)

x **=** x**+**1 *#you can modify the value , but once it is made as global it remains global through out*

*#and any other local variable of same name will also be considered as global only*

print("Ater modification",x)

x **=** 2

print('Ran func(), changed global x to', x)

​

print('Before calling func(), x is: ', x)

func()

print('Value of x (outside of func()) is: ', x) *# is 2 only , to avoid this we use globals()*

Before calling func(), x is: 50

This function is now using the global x!

Because of global x is: 50

Ater modification 51

Ran func(), changed global x to 2

Value of x (outside of func()) is: 2

The global statement is used to declare that **x** is a global variable - hence, when we assign a value to **x** inside the function, that change is reflected when we use the value of **x** in the main block.

You can specify more than one global variable using the same global statement e.g. global x, y, z.

A symbol table is a data structure maintained by a compiler which contains all necessary information about the program.

These include variable names, methods, classes, etc.

There are mainly two kinds of symbol table.

Global symbol table Local symbol table A Global symbol table stores all information related to the global scope of the program, and is accessed in Python using globals() method.

The global scope contains all functions, variables which are not associated to any class or function.

Likewise, Local symbol table stores all information related to the local scope of the program, and is accessed in Python using locals() method.

The local scope could be within a function, within a class, etc.

Recommended Reading: Namespace and scope of in Python

​

​

Type *Markdown* and LaTeX: 𝛼2α2

*#globals() builtin function which return table of current global variables in the form of a dictionary*

*# hence we can refer to the global variable 'a' as: globals()['a'] . now this value can be assigned to another variable*

a**=**1

**def** myfunction():

a**=**2 *# local var*

x **=** globals()['a'] *# get global var into x , you are using global variable with some other name*

print(id(x))

x **=** x**+**3 *#now you can modify the global variable , since it is modified to create new object the address changes and change to x will not change global a*

print(id(x))

print('global var a=' ,x)

print('local var a =',a)

*# globals()['a'] = 4 #a+1 #we can even assign values to the global variable using the globals symbol table*

print(id(a))

myfunction()

print(id(a))

print('global var a', a)

1626039392

1626039392

1626039440

global var a= 4

local var a = 2

1626039392

global var a 1

*#similar to globals() we have locals()*

title **=** "Outer title"

**def** innerfunc():

title **=** '2'

title1 **=** locals()['title']

title1 **=** title1 **+** "Inner Title"

print("Inner Title",title1)

print("Outer Title",title)

print("Outer Title",title)

innerfunc()

*# outerfunc()*

print("Outer Title",title)

Outer Title Outer title

Inner Title 2Inner Title

Outer Title 2

Outer Title Outer title

*# locals() #check in empty file default builtin locals will be displayed*

*# globals()*

**Nonlocal Variables**

It is possible to define functions inside other functions, and this can be very useful when we are working with collections of data and operations However, local variables are local to a specific function; even functions defined within another function cannot modify the outer functions local variables (as the inner function is a separate function). They can reference it, just as we could reference the global variable earlier; the issue is again modification. The global keyword is no help here as the outer function’s variables are not global, they are local to a function. For example, if we define a nested function (inner) inside the parent outer function (outer) and want the inner function to modify the local field we have a problem:

​

**def** outer():

title **=** 2

**def** inner():

​

title **=** title**+**1 *# can be accessed but modification will give error*

print('inner:', title)

​

inner()

print('outer:', title)

outer()

**---------------------------------------------------------------------------**

**UnboundLocalError** Traceback (most recent call last)

**<ipython-input-3-96e058fede00>** in <module>

10 print**('outer:',** title**)**

11

**---> 12** outer**()**

**<ipython-input-3-96e058fede00>** in outer**()**

7

8

**----> 9** inner**()**

10 print**('outer:',** title**)**

11

**<ipython-input-3-96e058fede00>** in inner**()**

3 **def** inner**():**

4

**----> 5** title **=** title**+1** **# can be accessed but modification will give error**

6 print**('inner:',** title**)**

7

**UnboundLocalError**: local variable 'title' referenced before assignment

​

In this example both outer() and inner() functions modify the title variable. However, they are not the same title variable and as long as this is what we need then that is fine; both functions have their own version of a title local variable. This can be seen in the output where the outer function maintains its own value for title: However, if what we want is for the inner() function to modify the outer() function’s title variable then we have a problem. This problem can be solved using the nonlocal keyword. This indicates that a variable is not global but is also not local to the current function and Python should look within the scope in which the function is defined to fund a local variable with the same name: If we now declare title as nonlocal in the inner() function, then it will use the outer() functions version of title (it will be shared between them) and thus when the inner() function changes the title it will change the it for both functions

**def** outer():

title **=** 3

**def** inner():

**nonlocal** title

​

title **=** title**+**1 *#can be modified*

print('inner:', title)

x **=** locals()['title']

x**=** 10

print(locals()['title'],x)

inner()

print('outer:', title)

print(locals())

​

outer()

inner: 4

4 10

outer: 4

{'inner': <function outer.<locals>.inner at 0x00B07930>, 'title': 4}

**Best Practices**

Even though Python provides the global and nonlocal keywords, it’s not always advisable to use them.

When a function modifies data outside the local scope, either with the global or nonlocal keyword or by directly modifying a mutable type in place, it’s a kind of side effect similar to when a function modifies one of its arguments. Widespread modification of global variables is generally considered unwise, not only in Python but also in other programming languages.

As with many things, this is somewhat a matter of style and preference. There are times when judicious use of global variable modification can reduce program complexity.

In Python, using the global keyword at least makes it explicit that the function is modifying a global variable. In many languages, a function can modify a global variable just by assignment, without announcing it in any way. This can make it very difficult to track down where global data is being modified.

All in all, modifying variables outside the local scope usually isn’t necessary. There’s almost always a better way, usually with function return values.