# Theory: Scopes



20 minutes 9 / 11 problems solved

Start practicing

A **scope** is a part of the program where a certain variable can be reached by its name. The scope is a very important concept in programming because it defines the visibility of a name within the code block.

## §1. Global vs. Local

When you define a variable it becomes either **global** or **local**. If a variable is defined at the top-level of the module it is considered global. That means that you can refer to this variable from every code block in your program. Global variables can be useful when you need to share state information or some configuration between different functions. For example, you can store the name of a current user in a global variable and then use it where needed. It makes your code easier to change: in order to set a new user name you will only have to change a single variable.

Local variables are created when you define them in the body of a function. So its name can only be resolved inside the current function's scope. It lets you avoid issues with side-effects that may happen when using global variables.

Consider the example to see the difference between global and local variables:

```
phrase = "Let it be"

def global_printer():
    print(phrase)  # we can use phrase because it's a global variable

global_printer()  # Let it be is printed
print(phrase)  # we can also print it directly

phrase = "Hey Jude"



global_printer()  # Hey Jude is now printed because we changed the value of phrase



def printer():

    local_phrase = "Yesterday"

    print(local_phrase)  # local_phrase is a local variable



printer()  # Yesterday is printed as expected



print(local_phrase)  # NameError is raised
```

Thus, a global variable can be accessed both from the top-level of the module and the function's body. On the other hand, a local variable is only visible inside the nearest scope and cannot be accessed from the outside.

## §2. LEGB rule

A variable resolution in Python follows the **LEGB rule**. That means that the interpreter looks for a name in the following order:

1. **Locals**. Variables defined within the function body and not declared global.

Current topic:

✓ Scopes 5★ (Stage 1)

Topic depends on:

✓ Declaring a function 10★ (Stage 1)

Topic is required for:

✓ Class instances 3★ (Stage 1)

Table of contents:

2. **Enclosing**. Names of the local scope in all enclosing functions from inner to outer.
3. **Globals**. Names defined at the top-level of a module or declared global with a `global` keyword.
4. **Built-in**. Any built-in name in Python.

Let's consider an example to illustrate the LEGB rule:

```
x = "global"
def outer():
    x = "outer local"
    def inner():
        x = "inner local"
        def func():
            x = "func local"
            print(x)
        func()

    inner()



outer()  # "func local"
```

When the `print()` function inside the `func()` is called the interpreter needs to resolve the name `x`. It'll first look at the innermost variables and will search for the local definition of `x` in `func()` function. In the case of the code above, the interpreter will find the local `x` in `func()` successfully and print its value, `'func local'`. But what if there isn't a definition of `x` in `func()`? Then, the interpreter will move outward and turn to `inner()` function. Check out the following example:

```
x = "global"
def outer():
    x = "outer local"
    def inner():
        x = "inner local"
        def func():
            print(x)
        func()
    inner()



outer()  # "inner local"
```

As you see, the name `x` was resolved in `inner()` function, since the value `"inner local"` was printed.

If we remove the definition of `x` from the `inner()` function as well and run the code again, the interpreter will continue the search among the `outer()` locals in the same fashion. If we keep deleting the lines of code defining `x`, the interpreter will move on to `outer()` locals, then globals, and then built-in names. In case there is no matching built-in name, an error will be raised. Look at the example where the global definition of `x` is reached by the interpreter:

```
x = "global"
def outer():
    def inner():
        def func():
            print(x)
        func()
    inner()

outer()  # "global"
```

Don't forget about LEGB rule if you plan on using enclosing functions.

## §3. Keywords "nonlocal" and "global"

We already mentioned one way to assign a global variable: make a definition at the top-level of a module. But there is also a special keyword `global` that allows us to declare a variable global inside a function's body.

You can't change the value of a global variable inside the function without using the `global` keyword:

```
x = 1
def print_global():
    print(x)

print_global()  # 1

def modify_global():
    print(x)
    x = x + 1



modify_global()  # UnboundLocalError
```

An error is raised because we are trying to assign to a local variable `x` the expression that contains `x` and the interpreter can't find this variable in a local scope. To fix this error, we need to declare `x` global:

```
x = 1
def global_func():
    global x
    print(x)
    x = x + 1

global_func()  # 1
global_func()  # 2
global_func()  # 3
```

When `x` is global you can increment its value inside the function.

`nonlocal` keyword lets us assign to variables in the outer (but not global) scope:

```
def func():
    x = 1
    def inner():
        x = 2
        print("inner:", x)
    inner()
    print("outer:", x)

def nonlocal_func():

    x = 1

    def inner():

        nonlocal x

        x = 2

        print("inner:", x)

    inner()

    print("outer:", x)


func()  # inner: 2

        # outer: 1


nonlocal_func()  # inner: 2

                 # outer: 2
```

Though `global` and `nonlocal` are present in the language, they are not often used in practice, because these keywords make programs less predictable and harder to understand.

## §4. Why do we need scopes?

First of all, why does Python need the distinction between global and local scope? Well, from the experience of some other programming languages that do not have *local* scopes it became clear, that using only global scope is highly inconvenient: when every variable is accessible from every part of the code, a whole bunch of bugs is inevitable. The longer the code, the more difficult it becomes to remember all the variables' names and not accidentally change the value of the variable that you were supposed to keep untouched. Therefore, Python saves you the trouble by allowing you to "isolate" some variables from the rest of the code when you split it into functions.

On the other hand, why do we need *global* scope then? Well, as was already mentioned above, global scope is one of the easiest ways to retain information between function calls: while local variables disappear the moment the function returns, global variables remain and help functions to transfer the necessary data between each other. Similarly, global variables can enable the communication between more complex processes, such as threads in multithreaded applications.