GLOBAL, LOCAL AND NONLOCAL VARIABLES

The way Python uses global and local variables is maverick. While in many or most other programming languages variables are treated as global if not otherwise declared, Python deals with variables the other way around. They are local, if not otherwise declared. The driving reason behind this approach is that global variables are generally bad practice and should be avoided. In most cases where you are tempted to use a global variable, it is better to utilize a parameter for getting a value into a function or return a value to get it out. Like in many other program structures, Python also imposes good programming habit by design.

So when you define variables inside a function definition, they are local to this function by default. This means that anything you will do to such a variable in the body of the function will have no effect on other variables outside of the function, even if they have the same name. This means that the function body is the scope of such a variable, i.e. the enclosing context where this name with its values is associated.

All variables have the scope of the block, where they are declared and defined in. They can only be used after the point of their declaration.

Just to make things clear: Variables don't have to be and can't be declared in the way they are declared in programming languages like Java or C. Variables in Python are implicitly declared by defining them, i.e. the first time you assign a value to a variable, this variable is declared and has automatically the data type of the object which has to be assigned to it. If you have problems understanding this, please consult our chapter about data types and variables, see links on the left side.

GLOBAL AND LOCAL VARIABLES IN FUNCTIONS

In the following example, we want to demonstrate, how global values can be used inside the body of a function:

```
def f():
    print(s)
s = "I love Paris in the summer!"
f()
```

The variable s is defined as the string "I love Paris in the summer!", before calling the function f(). The body of f() consists solely of the "print(s)" statement. As there is no local variable s, i.e. no assignment to s, the value from the global variable s will be used. So the output will be the string "I love Paris in the summer!". The question is, what will happen, if we change the value of s inside of the function f()? Will it affect the global variable as well? We test this in the following piece of code:

```
def f():
    s = "I love London!"
    print(s)

s = "I love Paris!"
f()
print(s)
```

The output looks like this:

```
I love London!
I love Paris!
```

What if we combine the first example with the second one, i.e. we first access s with a print() function, hoping to get the global value, and then assigning a new value to it? Assigning a value to it, means - as we have previously stated - creating a local variable s. So, we would have s both as a global and a local variable in the same scope, i.e. the body of the function. Python fortunately doesn't allow this ambiguity. So, it will throw an error, as we can see in the following example:

```
>>> def f():
...   print(s)
...   s = "I love London!"
...   print(s)
...
>>> s = "I love Paris!"
>>> f()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in f
UnboundLocalError: local variable 's' referenced before assignment
>>>
```

A variable can't be both local and global inside of a function. So Python decides that we want a local variable due to the assignment to s inside of f(), so the first print statement before the definition of s throws the error message above. Any variable which is changed or created inside of a function is local, if it hasn't been declared as a global variable. To tell Python, that we want to use the global variable, we have to explicitly state this by using the keyword "global", as can be seen in the following example:

```
def f():
    global s
    print(s)
    s = "Only in spring, but London is great as well!"
    print(s)


s = "I am looking for a course in Paris!"
f()
print(s)
```

We have solved our problem. There is no ambiguity left. The output of this small script looks like this:

```
I am looking for a course in Paris!
Only in spring, but London is great as well!
Only in spring, but London is great as well!
```

Local variables of functions can't be accessed from outside, when the function call has finished:

```
def f():
    s = "I am globally not known"
    print(s)

f()
print(s)
```

Starting this script gives us the following output with an error message:

```
monty@python:~$ python3 ex.py
I am globally not known
Traceback (most recent call last):
  File "ex.py", line 6, in <module>
    print(s)
NameError: name 's' is not defined
monty@python:~$
```

The following example shows a wild combination of local and global variables and function parameters:

```
def foo(x, y):
    global a
    a = 42
    x,y = y,x
    b = 33
    b = 17
    c = 100
    print(a,b,x,y)

a, b, x, y = 1, 15, 3,4
foo(17, 4)
print(a, b, x, y)
```

The output looks like this:

```
42 17 4 17
42 15 3 4
```

GLOBAL VARIABLES IN NESTED FUNCTIONS

We will examine now what will happen, if we use the global keyword inside of nested functions. The following example shows a situation where a variable x is used in various scopes:

```
def f():
    x = 42
    def g():
        global x
        x = 43
    print("Before calling g: " + str(x))
    print("Calling g now:")
    g()
    print("After calling g: " + str(x))

f()
print("x in main: " + str(x))
```

This program returns the following results:

```
Before calling g: 42
Calling g now:
After calling g: 42
x in main: 43
```

We can see that the global statement inside of the nested function g does not affect the variable x of the function f, i.e. it keeps its value 42. We can also deduce from this example that after calling f() a variable x exists in the module namespace and has the value 43. This means that the global keyword in nested functions does not affect the namespace of their enclosing namespace! This is consistent to what we have found out in the previous subchapter: A variable defined inside of a function is local unless it is explicitly marked as global. In other words, we can refer a variable name in any enclosing scope, but we can only rebind variable names in the local scope by assigning to it or in the module-global scope by using a global declaration. We need a way to access variables of other scopes as well. The way to do this are nonlocal definitions, which we will explain in the next chapter.

NONLOCAL VARIABLES

Python3 introduced nonlocal variables as a new kind of variables. nonlocal variables have a lot in common with global variables. One difference to global variables lies in the fact that it is not possible to change variables from the module scope, i.e. variables which are not defined inside of a function, by using the nonlocal statement. We show this in the two following examples:

```
def f():
    global x
    print(x)
```

```
x = 3
f()
```

This program is correct and returns the number 3 as the output. We will change "global" to "nonlocal" in the following program:

```
def f():
    nonlocal x
    print(x)

x = 3
f()
```

The program, which we have saved as example1.py, cannot be executed anymore now. We get the following error:

```
  File "example1.py", line 2
    nonlocal x
SyntaxError: no binding for nonlocal 'x' found
```

This means that nonlocal bindings can only be used inside of nested functions. A nonlocal variable has to be defined in the enclosing function scope. If the variable is not defined in the enclosing function scope, the variable cannot be defined in the nested scope. This is another difference to the "global" semantics.

```
def f():
    x = 42
    def g():
        nonlocal x
        x = 43
    print("Before calling g: " + str(x))
    print("Calling g now:")
    g()
    print("After calling g: " + str(x))

x = 3
f()
print("x in main: " + str(x))
```

Calling the previous program results in the following output:

```
Before calling g: 42
Calling g now:
After calling g: 43
x in main: 3
```

In the previous example the variable x was defined prior to the call of g. We get an error if it isn't defined:

```
def f():
    #x = 42
    def g():
        nonlocal x
        x = 43
    print("Before calling g: " + str(x))
    print("Calling g now:")
    g()
    print("After calling g: " + str(x))

x = 3
f()
print("x in main: " + str(x))
```

We get the following error message:

```
  File "example3.py", line 4
    nonlocal x
SyntaxError: no binding for nonlocal 'x' found
```

The program works find - with or without the line "x = 42" inside of f - , when we change "nonlocal" to "global":

```
def f():
    #x = 42
    def g():
        global x
        x = 43
    print("Before calling g: " + str(x))
    print("Calling g now:")
    g()
    print("After calling g: " + str(x))

x = 3
f()
print("x in main: " + str(x))
```

This leads to the following output:

```
Calling g now:
The value of x after the call to g: 43

Before calling g: 3
Calling g now:
After calling g: 43
x in main: 43
```

Yet there is a huge difference: The value of the global x is changed now!