Python Essentials 1: Module 4

Functions, Tuples, Dictionaries, and Data Processing

In this module, you will cover the following topics:

code structuring and the concept of function;
function invocation and returning a result from a function;
name scopes and variable shadowing;
tuples and their purpose, constructing and using tuples;
dictionaries and their purpose, constructing and using dictionaries.

(part 4)

Functions and scopes

Let's start with a definition:

The scope of a name (e.g., a variable name) is the part of a code where the name is properly recognizable.

For example, the scope of a function's parameter is the function itself. The parameter is inaccessible outside the function.

Let's check it. Look at the code in the editor. What will happen when you run it?

```
def scope_test():
    x = 123


scope_test()
print(x)
```

The program will fail when run. The error message will read:

```
NameError: name 'x' is not defined
```
output

This is to be expected.

We're going to conduct some experiments with you to show you how Python constructs scopes, and how you can use its habits to your benefit.

Functions and scopes: continued

Let's start by checking whether or not a variable created outside any function is visible inside the functions. In other words, does a variable's name propagate into a function's body?

Look at the code in the editor. Our guinea pig is there.

```
def my_function():
    print("Do I know that variable?", var)


var = 1
my_function()
print(var)
```

The result of the test is positive - the code outputs:

```
Do I know that variable? 1
1
```

output

The answer is: a variable existing outside a function has a scope inside the functions' bodies.

This rule has a very important exception. Let's try to find it.
Let's make a small change to the code:

```
def my_function():
    var = 2
    print("Do I know that variable?", var)

var = 1
my_function()
print(var)
```

The result has changed, too - the code produces a slightly different output now:

```
Do I know that variable? 2
1
```

output

What's happened?

the var variable created inside the function is not the same as when defined outside it - it seems that there two different variables of the same name;
moreover, the function's variable shadows the variable coming from the outside world.
We can make the previous rule more precise and adequate:

A variable existing outside a function has a scope inside the functions' bodies, excluding those of them which define a variable of the same name.

It also means that the scope of a variable existing outside a function is supported only when getting its value (reading). Assigning a value forces the creation of the function's own variable.

Make sure you understand this well and carry out your own experiments.

Functions and scopes: the global keyword

Hopefully, you should now have arrived at the following question: does this mean that a function is not able to modify a variable defined outside it? This would create a lot of discomfort.

Fortunately, the answer is no.

There's a special Python method which can extend a variable's scope in a way which includes the functions' bodies (even if you want not only to read the values, but also to modify them).

Such an effect is caused by a keyword named global:

```
global name
global name1, name2, ...
```

Using this keyword inside a function with the name (or names separated with commas) of a variable(s), forces Python to refrain from creating a new variable inside the function - the one accessible from outside will be used instead.

In other words, this name becomes global (it has a global scope, and it doesn't matter whether it's the subject of read or assign).

Look at the code in the editor.

```
def my_function():
    global var
    var = 2
    print("Do I know that variable?", var)


var = 1
my_function()
print(var)
```

We've added global to the function.

The code now outputs:

```
Do I know that variable? 2
2
```
output

This should be sufficient evidence to show that the global keyword does what it promises.

How the function interacts with its arguments

Now let's find out how the function interacts with its arguments.

The code in the editor should teach you something. As you can see, the function changes the value of its parameter. Does the change affect the argument?

```
def my_function(n):
    print("I got", n)
    n += 1
    print("I have", n)


var = 1
my_function(var)
print(var)
```

Run the program and check.
The code's output is:

```
I got 1
I have 2
1
```
output

The conclusion is obvious - changing the parameter's value doesn't propagate outside the function (in any case, not when the variable is a scalar, like in the example).

This also means that a function receives the argument's value, not the argument itself. This is true for scalars.

Is it worth checking how it works with lists (do you recall the peculiarities of assigning list slices versus assigning lists as a whole?).

The following example will shed some light on the issue:

```
def my_function(my_list_1):
    print("Print #1:", my_list_1)
    print("Print #2:", my_list_2)
    my_list_1 = [0, 1]
    print("Print #3:", my_list_1)
    print("Print #4:", my_list_2)


my_list_2 = [2, 3]
my_function(my_list_2)
print("Print #5:", my_list_2)
```

The code's output is:

```
Print #1: [2, 3]
Print #2: [2, 3]
Print #3: [0, 1]
Print #4: [2, 3]
Print #5: [2, 3]
```
output

It seems that the former rule still works.

Finally, can you see the difference in the example below:

```
def my_function(my_list_1):
    print("Print #1:", my_list_1)
    print("Print #2:", my_list_2)
    del my_list_1[0]  # Pay attention to this line.
    print("Print #3:", my_list_1)
    print("Print #4:", my_list_2)


my_list_2 = [2, 3]
my_function(my_list_2)
print("Print #5:", my_list_2)
```

We don't change the value of the parameter my_list_1 (we already know it will not affect the argument), but instead modify the list identified by it.

The output may be surprising. Run the code and check:

```
Print #1: [2, 3]
Print #2: [2, 3]
Print #3: [3]
Print #4: [3]
Print #5: [3]
```
output

Can you explain it?

Let's try:

if the argument is a list, then changing the value of the corresponding parameter doesn't affect the list (remember: variables containing lists are stored in a different way than scalars),

but if you change a list identified by the parameter (note: the list, not the parameter!), the list will reflect the change.

It's time to write some example functions. You'll do that in the next section.

Key takeaways

1. A variable that exists outside a function has a scope inside the function body (Example 1) unless the function defines a variable of the same name (Example 2, and Example 3), e.g.:

Example 1:
```
var = 2
def mult_by_var(x):
    return x * var
print(mult_by_var(7))    # outputs: 14
```

Example 2:
```
def mult(x):
    var = 5
    return x * var
print(mult(7))    # outputs: 35
```

Example 3:
```
def mult(x):
    var = 7
    return x * var
var = 3
print(mult(7))    # outputs: 49
```

2. A variable that exists inside a function has a scope inside the function body (Example 4), e.g.:

Example 4:
```
def adding(x):
    var = 7
    return x + var
print(adding(4))    # outputs: 11
print(var)    # NameError
```

3. You can use the global keyword followed by a variable name to make the variable's scope global, e.g.:
```
var = 2
print(var)    # outputs: 2
def return_var():
    global var
    var = 5
    return var
print(return_var())    # outputs: 5
print(var)    # outputs: 5
```

Exercise 1

What will happen when you try to run the following code?
```
def message():
    alt = 1
    print("Hello, World!")
print(alt)
```

Exercise 2

What is the output of the following snippet?

```
a = 1
def fun():
    a = 2
    print(a)
fun()
print(a)
```

Exercise 3

What is the output of the following snippet?

```
a = 1
def fun():
    global a
    a = 2
    print(a)
fun()
a = 3
print(a)
```

Exercise 4

What is the output of the following snippet?

```
a = 1
def fun():
    global a
    a = 2
    print(a)
a = 3
fun()
print(a)
```