local_vs_global

January 23, 2018

1 Local vs global variables

In Python, you can use two kinds of variables: - **global variables**: defined outside functions. They are accessible everywhere inside the module (module = file with .py extension). - **local variables**: created inside functions. They are accessible only inside the function in which they are created.

WARNING: a local variable can have the same name than a global variable. In that case, the global variable cannot be accessed normally within that function: it is *shadowed* by the local variable.

1.0.1 Python variables

Rule 1: functions have access to global variables:

```
In [1]: global_var = "I'm a global variable"

        def my_function():
            # a function can access variables defined outside of it
            print(global_var)

        my_function()
```

```
I'm a global variable
```

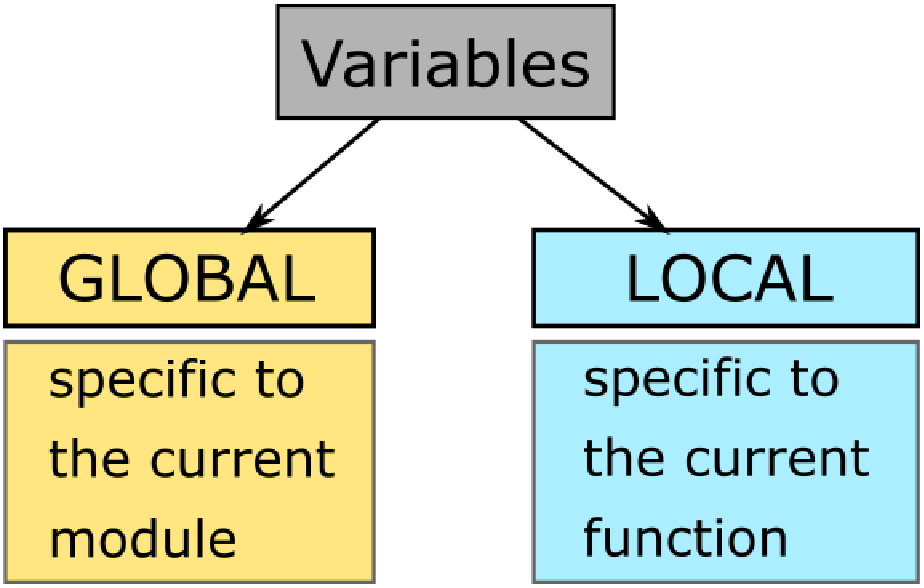
Rule 2: variable assignments (i.e. using operator =) in a function create or act on local variables.

Rule 3: if a local variable has the same name as a global one, using the name will access the **local** variable (*variable shadowing*).

```
In [2]: var = "I'm a global variable"

        def my_function():
            # create a local variable named 'var'
            var = "I'm a local variable"
            # if a local variable has the same name as a global one, Python will access the loca
            print(var)

        my_function()
        # the global variable has not been modified
        print(var)
```


Variables
GLOBAL
specific to the current module
LOCAL
specific to the current function

python_global_local_vars.png

```
I'm a local variable
I'm a global variable
```

Don't want to lose your modifications? Use the **return** statement:

```
In [3]: var = "I'm a global variable"

        def my_function():
            # variable assignments (i.e. using operator =) in a function create or act on local
            var = "I'm a global variable and I have been modified"
            # Don't want to lose your modifications? Use the return statement
            return var

        # set new content to the variable 'var'
        var = my_function()
        # the variable 'var' has been modified
        print(var)
```

```
I'm a global variable and I have been modified
```

Rule 4: Input arguments are local variables. Use the **return** statement If you want to keep back your modifications after the call to the function:

```
In [4]: var = "I'm a variable called 'var'"

        def my_function(var):
            # input arguments are local
            print('local variables:', locals())
            # modifications on input arguments are not kept after function call
            var = "I'm an input argument and I'm local to the function"
            print(var)

        my_function(var)
        # the global variable has not been modified
        print(var)
```

```
local variables: {'var': "I'm a variable called 'var'"}
I'm an input argument and I'm local to the function
I'm a variable called 'var'
```

```
In [5]: var = "I'm a global variable"

        def my_function(var):
            # modifications on input arguments are not kept after function call
            # input arguments are local
            var = "I'm an input argument and I'm local to the function"
            # use return statement to keep back your modifications
            return var

        # set new content to the variable 'var'
        var = my_function(var)
        # the variable 'var' has been modified
        print(var)
```

```
I'm an input argument and I'm local to the function
```

End of story? Nope.

Python can be vicious as a snake...

In Python, you have to manipulate two kinds of objects: - **Immutable objects**: int, float, boolean, string, tuple. - **Mutable objects**: list, dict, Axes, LArray, Session, ...

Specific rules applies to **mutable** objects.

1.0.2 Mutable objects (list, dict, Axes, LArray, Session, ...)

Rule 5: Modifying **elements** of a **mutable** variable (list, dictionary, array, session, ...) does not create a new local variable:

```
In [6]: from larray import *

        array_1 = zeros('sex = F,M')
        array_2 = ones('country = be,fr,de')

        def my_function():
            # assigning the whole array creates a new local array
            array_1 = ones('sex = F,M')
            # assigning a subset of an array does not create a local array
            array_2['fr,de'] = 0

        print("array_1:")
        print(array_1)
        print("\narray_2:")
        print(array_2)

        print("\nlet's call 'my_function' and try to modify array_1 and array_2\n")
        my_function()

        print("array_1 has not been modified:")
        print(array_1)
        print("\narray_2 has been modified:")
        print(array_2)
```

```
array_1:
sex    F    M
     0.0  0.0

array_2:
country   be   fr   de
         1.0  1.0  1.0

let's call 'my_function' and try to modify array_1 and array_2

array_1 has not been modified:
sex    F    M
     0.0  0.0

array_2 has been modified:
country   be   fr   de
         1.0  0.0  0.0
```

Why?

Assigning a new value to an object (x = 5, y = [0, 1, 2, 3]) creates a new object.

Instead, modifying elements of a mutable object (y[1:3] = [0, 0]) does not create a new object but modifies the existing object.

What if want to modify the whole content of an array?

Rule 6: To change the whole content of an array without creating a new local one, add **[:]** next to the array:

```
In [7]: from larray import *

        array_1 = zeros('sex = F,M')
        def my_function():
            # trick: to change to whole content of an array, add [:] next to the array
            array_1[:] = ones('sex = F,M')

        print("array_1:")
        print(array_1)

        print("\nlet's call 'my_function' and try to modify the whole content of array_1 using [
        my_function()

        print("array_1 has been modified:")
        print(array_1)
```

```
array_1:
sex    F    M
     0.0  0.0

let's call 'my_function' and try to modify the whole content of array_1 using [:]

array_1 has been modified:
sex    F    M
     1.0  1.0
```

What about input arguments?

Rule 7: Modifying **elements** of a **mutable** input argument (list, dictionary, array, session, ...) modify also the content of the associated variable passed to the function:

```
In [8]: from larray import *

        array_1 = zeros('sex = F,M')
        array_2 = ones('country = be,fr,de')

        def my_function(arr_1, arr_2):
            # assigning the whole array creates a new array
            arr_1 = ones('sex = F,M')
            # assigning a subset of an array does not create a new array
            arr_2['fr,de'] = 0

        print("array_1:")
        print(array_1)
        print("\narray_2:")
        print(array_2)

        print("\nlet's call 'my_function' and try to modify array_1 and array_2\n")
        my_function(array_1, array_2)

        print("array_1 has not been modified:")
```

```
    print(array_1)
    print("\narray_2 has been modified:")
    print(array_2)
```

```
array_1:
sex    F    M
     0.0  0.0

array_2:
country   be   fr   de
         1.0  1.0  1.0

let's call 'my_function' and try to modify array_1 and array_2

array_1 has not been modified:
sex    F    M
     0.0  0.0

array_2 has been modified:
country   be   fr   de
         1.0  0.0  0.0
```

1.0.3 What to remember?

For all objects:

1. Functions have access to global variables.
2. Variable assignments (i.e. using operator =) in a function create or act on local variables.
3. if a local variable has the same name as a global one, using the name will access the **local** variable (*variable shadowing*).
4. Input arguments are local variables. Use the **return** statement If you want to keep back your modifications after the call to the function.

For mutable objects (list, dict, Axes, LArray, Session, ...):

5. Modifying **elements** of a **mutable** variable does not create a new local variable (e.g. pop[10:99] = 0).
6. To change the whole content of an array without creating a new local one, add **[:]** next to the array (e.g. pop[:] = 0).
7. Modifying **elements** of a **mutable** input argument modify also the content of the variable passed to the function (e.g. pop[10:99] = 0).

TIPS:

Global variables may be dangerous. When it is possible, write functions as **independent** blocks of code and pass any external variables you need to work with as input arguments. Use **return** statement to return your modifications.

When you have to deal with many external variables (arrays), passing all of them as arguments may become cumbersome. In that case, remember that modifying **elements** of *mutable* variables does not create a new object. This behavior can be used to simplify function definitions but must be used carefully.