Scope of the Variables

Table of Contents

1 Theoretical Definitions

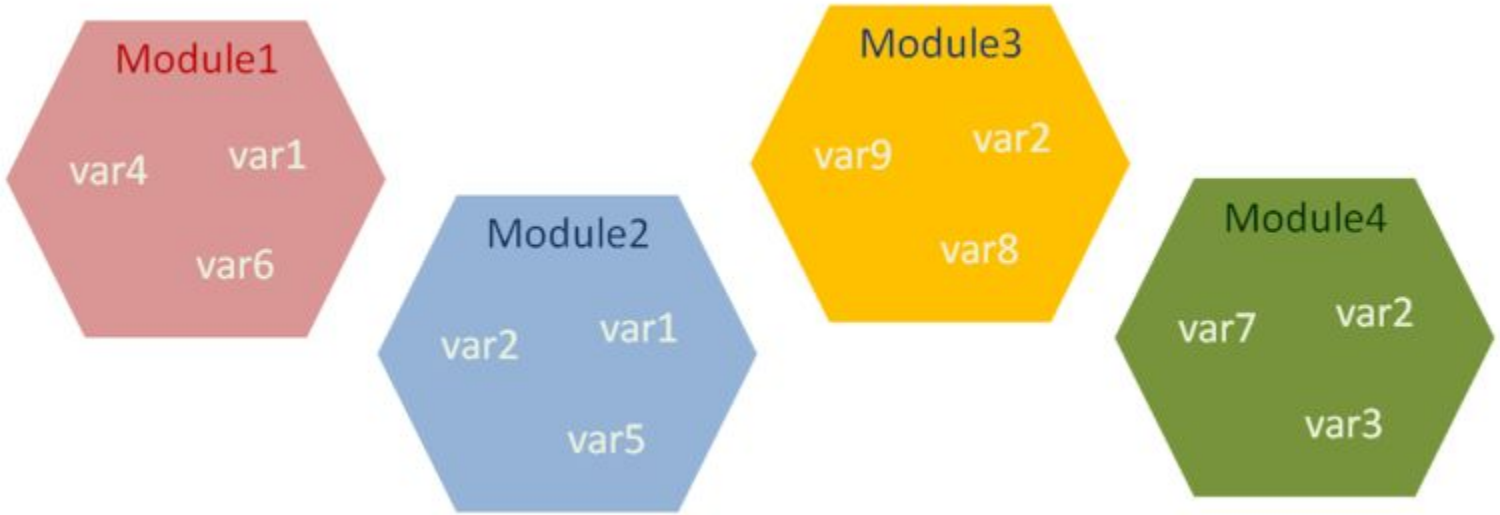
What did you learn from the pre-class contents?

Can you describe “What is Namespace?” and “What is Scope?”

Students, write your response!

What is Namespace? (review)

- In the following figure, you can see some variables which have the same names are in the different modules (**namespaces**) at the same time. You can work with the **variable** that you want using this syntax : `module.variable`. Considering this figure, we can call `var1` in the Module2 as : `module2.var1`



What is Scope? (review)

- The term **scope** is mostly related to nested **functions** and the **main program flow** in accordance with the use of variables. It describes the accessibility and the existence of a variable.
- A **scope** defines the **hierarchical order** in which the names of the variables have to exist in order to match **names** with the **objects**.

What is Scope? (review)

- Now, let's put all these definitions into practice with a simple example :

```
1  my_var = 'outer variable'
2
3  def func_var():
4      my_var= 'inner variable'
5      print(my_var)
6
7  func_var()
8  print(my_var)
```

What is the output? Try to figure out in your mind...

Students, write your response!

REINVENT YOURSELF

Pear Deck Interactive Slide
Do not remove this bar

What is Scope? (review)

- Now, let's put all these definitions into practice with a simple example :

```
1  my_var = 'outer variable'
2
3  def func_var():
4      my_var= 'inner variable'
5      print(my_var)
6
7  func_var()
8  print(my_var)
```

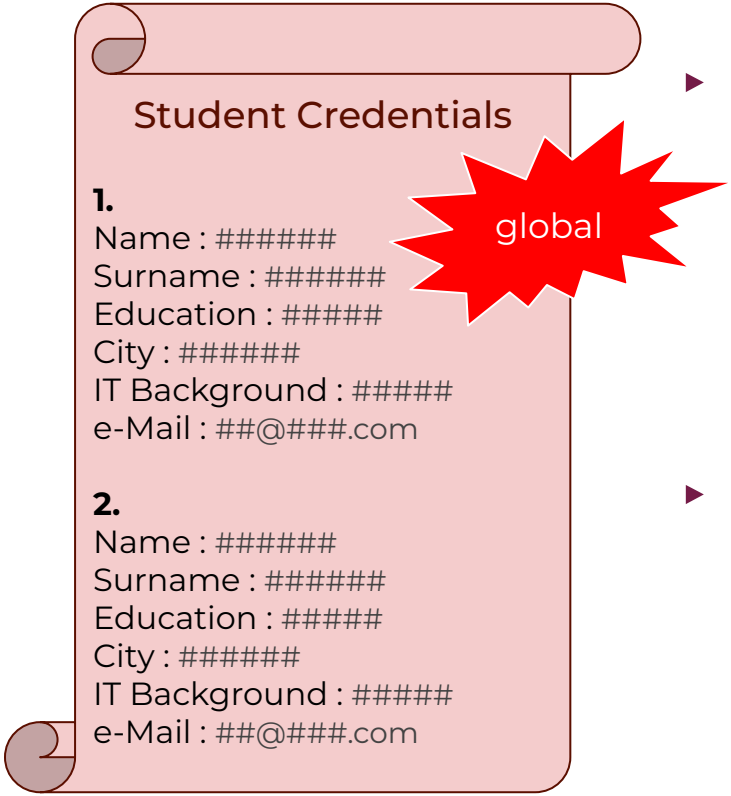
```
1  inner variable
2  outer variable
```

What is Scope? (review)

- As you can see in the example, the name of the variable (`my_var`) has been used both in the function (`func_var`) and at the top of the main program stream. When you call the function (`func_var`) or print directly the variable (`my_var`), you probably noticed that the same variable produces different outputs. This is because of the location (space) of that variable, that is, where or in which space it is defined in the program flow.

2 Global and Local Variables

Global Variable (review)



- If the variable you define is at the highest level of a module, that variable becomes **global**. So you have the freedom to use this **global variable** in a block of code anywhere in your program.
- Global variables allow us to make some interactions between functions. *For example*, suppose we store the credentials of a student who has applied for Clarusway in a **global variable**.

Global Variable (review)

- Let's assume that we use this global variable many times in 3 **different functions** that we have defined regarding course activities. The **global variable** provides us with convenience when the credentials of the person change. Only when we rearrange the information in this global variable will our variables in all functions be rearranged.

Global Variable

- Consider this example.

```
1  x = 5  # at the top of the scirpt, it's a global variable
2
3  def foo():
4      x = x*2  # we've tried to change the value of global variable 'x'
5      print(x)
6
7  foo()
8
```

What is the output? Try to figure out in your mind...

Students, write your response!

Global Variable

- Consider this example.

```
1  x = 5  # at the top of the scirpt, it's a global variable
2
3  def foo():
4      x = x*2  # we've tried to change the value of global variable 'x'
5      print(x)
6
7  foo()
8
```

Output

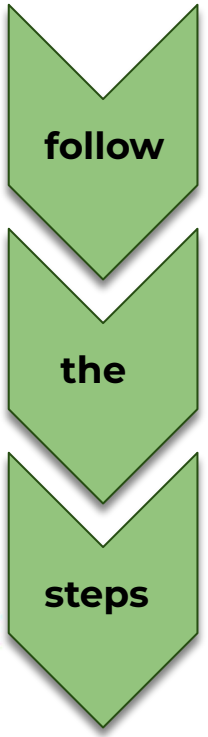
```
Traceback (most recent call last):
  File "code.py", line 7, in <module>
    foo()
  File "code.py", line 4, in foo
    x = x*2
UnboundLocalError: local variable 'x' referenced before assignment
```

Local Variable (review)

- The variables you have defined **in a function** body are **local**. The name of this variable is therefore **only valid** in the function body to which it is located.
- **Local variables** eliminate some of the confusion risks that global variables can cause.

Local Variable (review)

```
1  text = "I am the global one"
2
3  def global_func():
4      print(text)  # we can use 'text' in a function
5                   # because it's a global variable
6
7  global_func()  # 'I am the global one' will be printed
8  print(text)  # it can also be printed outside of the function
9
10 text = "The globals are valid everywhere "
11
12 global_func()  # we changed the value of 'text'
13 # 'The globals are valid everywhere' will be printed
14
15 def local_func():
16     local_text = "I am the local one"
17     print(local_text)  # local_text is a local variable
18
19 local_func()  # 'I am the local one' will be printed as expected
20
21 print(local_text)  # NameError will be raised
22 # because we can't use local variable outside of its function
```



Local Variable (review)

```
1  text = "I am the global one"
2
3  def global_func():
4      print(text)  # we can use 'text' in a function
5                   # because it's a global variable
6
7  global_func()  # 'I am the global one' will be printed
8  print(text)  # it can also be printed outside of the function
9
10 text = "The globals are valid everywhere "
11
12 global_func()  # we changed the value of 'text'
13 # 'The globals are valid everywhere' will be printed
14
15 def local_func():
16     local_text = "I am the local one"
17     print(local_text)  # local_text is a local variable
18
19 local_func()  # 'I am the local one' will be printed as expected
20
21 print(local_text)  # NameError will be raised
```

```
1  I am the global one
2  I am the global one
3  The globals are valid everywhere
4  I am the local one
5  ---------------------------------------------------------------------------
6  NameError: name 'local_text' is not defined
```

Read the descriptions on the next slide

Local Variable (review)

- In the above example, we have seen that a *global variable* can be accessed not only from the top-level of the module but also from the body of the function. On the other hand, a *local variable* is valid only in the function's body it is defined. So, it is accessible from inside the nearest scope level and can not be accessed from the outside.

Tips:

- You might have a question about where you will need to use these issues. But, if you are writing a relatively long algorithm, you will eventually need to work with the nested functions and modules.

Local Variable

- Consider another example.

```
1  def foo():
2      y = "local"  # 'y' is a local variable
3
4  foo()
5
6  print(y)  # we've tried to use local variable 'y' in the global scope
7
```

What is the output? Try to figure out in your mind...

Students, write your response!

Local Variable

- Consider another example.

```
1  def foo():
2      y = "local"  # 'y' is a local variable
3
4  foo()
5
6  print(y)  # we've tried to use local variable 'y' in the global scope
7
```

Output

```
Traceback (most recent call last):
  File "code.py", line 6, in <module>
    print(y)  # we've tried to use local variable 'y' in the global scope
NameError: name 'y' is not defined
```

3 LEGB Ranking Rule

LEGB Ranking Rule (review)

- When you call an object (method or variable), the interpreter looks for its name in the following order:

LEGB Ranking Rule (review)

- When you call an object (method or variable), the interpreter looks for its name in the following order:

Locals. The space which is searched first, contains the local names defined in a function body.

LEGB Ranking Rule (review)

- When you call an object (method or variable), the interpreter looks for its name in the following order:

Locals. The space which is searched first, contains the local names defined in a function body.

Enclosing. The scopes of any enclosing functions, which are searched starting with the nearest enclosing scope (from inner to outer), contains non-local, but also non-global names.

LEGB Ranking Rule (review)

- When you call an object (method or variable), the interpreter looks for its name in the following order:

Locals. The space which is searched first, contains the local names defined in a function body.

Enclosing. The scopes of any enclosing functions, which are searched starting with the nearest enclosing scope (from inner to outer), contains non-local, but also non-global names.

Globals. It contains the current module's global names. The variables defined at the top-level of its module.

LEGB Ranking Rule (review)

- When you call an object (method or variable), the interpreter looks for its name in the following order:

Locals. The space which is searched first, contains the local names defined in a function body.

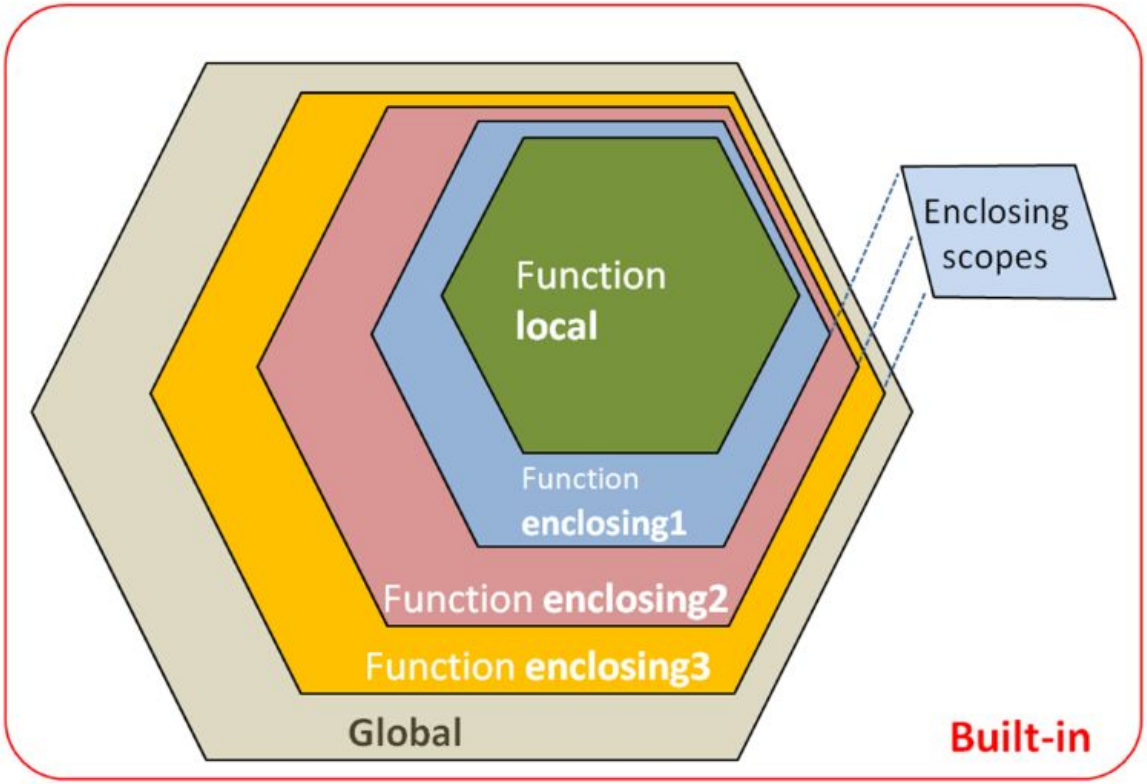
Enclosing. The scopes of any enclosing functions, which are searched starting with the nearest enclosing scope (from inner to outer), contains non-local, but also non-global names.

Globals. It contains the current module's global names. The variables defined at the top-level of its module.

Built-in. The outermost scope (searched last) is the namespace containing built-in names.

LEGB Ranking Rule (review)

- You can examine LEGB Rule in the following figure.



LEGB Ranking Rule (review)

- Let's see how it works in an example :

```
1  variable = "global"
2
3  def func_outer():
4      variable = "enclosing outer local"
5      def func_inner():
6          variable = "enclosing inner local"
7          def func_local():
8              variable = "local"
9              print(variable)
10         func_local()
11     func_inner()
12
13 func_outer()  # prints 'local' defined in the innermost function
14 print(variable)  # 'global' level variable holds its value
```

What is the output? Try to figure out in your mind...

Students, write your response!

LEGB Ranking Rule (review)

- Let's see how it works in an example :

```
variable = "global"

def func_outer():
    variable = "enclosing outer local"
    def func_inner():
        variable = "enclosing inner local"
        def func_local():
            variable = "local"
            print(variable)
        func_local()
    func_inner()

func_outer()  # prints 'local' defined in the innermost functio
print(variable)  # 'global' level variable holds its value
```

```
local
global
```

Read the descriptions on the next slide

LEGB Ranking Rule

- In this example, during the execution of the code lines, the interpreter has to resolve the name '*variable*'.
- The searching order of the variable names will be as follows :
  - 'local' in `func_local`
  - 'enclosing inner local' in `func_inner`
  - 'enclosing outer local' in `func_outer`
  - globals
  - built-in names

4 'global' and 'nonlocal'

'global' and 'nonlocal' (review)

- You know from the previous lesson that a variable defined in a function body becomes **local**. In some cases, we want to work with the variables defined as a *global scope in the function body*. Normally they are perceived globally and processed accordingly.
- Or we may need to work with the nonlocal variables in the function body. The keywords `global` and `nonlocal` save us from these restrictions.

Keyword 'global' (review)

- You can not change the value assigned to a globally defined variable within a function. To do this we use the keyword `global`. If you examine the example below you will understand better.

```
1  count = 1
2
3  def print_global():
4      print(count)
5
6  print_global()
7
8  def counter():
9      print(count)
10     count += 1  # we're trying to change its value
11
12 print()  # just empty line
13 counter()
```

What is the output? Try to figure out in your mind...

Students, write your response!

Keyword 'global' (review)

- The output :

```
1  count = 1
2
3  def print_global():
4      print(count)
5
6  print_global()
7
8  def counter():
9      print(count)
10     count += 1  # we're trying to change its value
11
12 print()  # just empty line
13 counter()
```

```
1  1
2
3  Traceback (most recent call last):
4    File "code.py", line 11, in <module>
5      counter()
6    File "code.py", line 8, in counter
7      print(count)
8  UnboundLocalError: local variable 'count' referenced before assignment
```

Read the descriptions on the next slide

Keyword '`global`' (review)

- As you can see in the example, if you try to assign a value **contains local variable expressions** to a **global variable** within a function, *UnboundLocalError* will raise.
- We've tried to assign a value to the `count` variable using an expression contains the `count` variable.
- This is because the interpreter can't find this variable in the **local scope**.
- So, let's use the keyword `global` to solve this problem.

Keyword 'global' (review)

```
1  count = 1
2
3  def counter():
4      global count  # we've changed its scope
5      print(count)  # it's global anymore
6      count += 1
7
8  counter()
9  counter()
10 counter()
```

Keyword 'global' (review)

```
1  count = 1
2
3  def counter():
4      global count  # we've changed its scope
5      print(count)  # it's global anymore
6      count += 1
7
8  counter()
9  counter()
10 counter()
```

```
1  1
2  2
3  3
```

Keyword 'nonlocal' (review)

- On the other hand, you can use the keyword `nonlocal` to extend the scope of the local variable to an upper scope. Consider the examples of non localization :

```
1  def func_enclosing1():
2      x = 'outer variable'
3      def func_enclosing2():
4          x = 'inner variable'
5          print("inner:", x)
6      func_enclosing2()
7      print("outer:", x)
8
9  func_enclosing1()
10
```

What is the output? Try to figure out in your mind...

Keyword 'nonlocal' (review)

- The output :

```
def func_enclosing1():
    x = 'outer variable'
    def func_enclosing2():
        x = 'inner variable'
        print("inner:", x)
    func_enclosing2()
    print("outer:", x)

func_enclosing1()

```

```
inner: inner variable
outer: outer variable
```

Keyword 'nonlocal' (review)

- We will make the variable `x` nonlocal so we can use its inner-value in the outer function (scope). Let's see.

```
1  def enclosing_func1():
2      x = 'outer variable'
3      def enclosing_func2():
4          nonlocal x  # its inner-value can be used in the outer scope
5          x = 'inner variable'
6          print("inner:", x)
7      enclosing_func2()
8      print("outer:", x)
9
10 enclosing_func1()
11
```

What is the output? Try to figure out in your mind...

Keyword 'nonlocal' (review)

- We will make the variable `x` nonlocal so we can use its inner-value in the outer function (scope). Let's see.

```
def enclosing_func1():
    x = 'outer variable'
    def enclosing_func2():
        nonlocal x  # its inner-value can be used in the outer scope
        x = 'inner variable'
        print("inner:", x)
    enclosing_func2()
    print("outer:", x)

enclosing_func1()

```

```
inner: inner variable
outer: inner variable
```

Keyword 'nonlocal' (review)

```
1  def enclosing_func1():
```

Tips:

- Frankly, these keywords are not widely used in programming but are worth discussing.

```
9
10 enclosing_func1()
11
```

```
1  inner: inner variable
2  outer: inner variable
```

Keyword 'global'

- **Task** :
  - Define a function named `assigner` to assign a new value that passed into it.
  - **Call** the function and print the result.

```
1  var = 1
2
3  def assigner(a):
4      ...
5      ...
6      ...
7  assigner("one")  # we change value of 'var'
8  print(var)
```

Output

```
one
```

Keyword 'global'

- **The defining of that function can be as:**

```
1  var = 1
2
3  def assigner(a):
4      global var
5      var = a
6
7  assigner("one")
8  print(var)
9
```

THANKS!

Any questions?

You can find me at:

- @andy
- andy@clarusway.com