Scoping

Scope

The scope of a variable means the area of the code where that variable can be accessed and modified.

The scope can be categorized into 4 groups.

- Global scope
- Local scope
- Enclosed scope
- Built-in scope

① Local can't be accessed from Global.
② but global can be accessed from Local + inclosed
③
scope
- The region that a variable is recognized
- A variable is only available from inside the region it is created.
- A global and locally scoped versions of a variable can be created

Local Scope & Local Variable

When a variable is created within a function body, then the scope of that variable is the function body and the variable is called "Local variable". Local variables can be modified and accessed within the function body where it was created (its scope). Outside that function, local variables do not exist.

1. ~~Example1~~: Printing the local variable inside a function

Code	Output

`def printHi():`
`  local_variable = "Hi local"`
`  print(local_variable)`
 	
`printHi()`
	`Hi local`
	
	Here, local_variable is a local variable of the function, so it can be accessed within that function

2. Example2: trying to access a local variable from outside the function.

Code	Output

def printHi():
  local_variable = "Hi local"
  print(local_variable)
 	
print(local_variable)
	NameError: name 'local_variable' is not defined
	
	Since local variables cannot be accessed from outside the function, it raises an error.

3. Example3: Using Global and Local variables in the same code. Global variables can be accessed from the inside of a function. But the local variables cannot be accessed outside its local scope.

Code	Output

a = "Global Hi"
def printHi():
  b = "Local Hi"
  print("a inside: " + a)
  print("b inside: " + b)
 	
printHi()
	a inside: Global Hi
	b inside: Local Hi

4. Example4: When the global and the local variables with the same name are used, Python creates a local version of the variable, instead of modifying the global variable with the same name. And this local version of the variable is called "Local variable".

Code	Output

a = "Global Hi"
def printHi():
  a = "Local Hi"
  print("Local a: " + a)
 	
printHi()
print("Global a: " + a)
	Local a: Local Hi
	Global a: Global Hi

Global Scope & Global variable

When a variable is created within the main body of a Python code, then the scope of that variable is the main body of the Python code, and that variable is called "Global variable". Global variables can be modified and accessed by any functions and from any scope (global, local & enclosed).

1. Example1: Global variables can be accessed from anywhere.

Code	Output
a = "Global Hi" def printHi(): print("a inside: " + a) printHi() print("a outside: " + a)	a inside: Global Hi a outside: Global Hi

2. Example 2: When we try to change the global variable inside a function, Python treats it as a local variable.

Code	Output

a = "Global Hi"
def printHi():
  a = a + "Bye"
  print("a inside: " + a)
 	
printHi()
print("a outside: " + a)
	UnboundLocalError: local variable 'a' referenced before assignment
	
	Here, Python is considering 'a' to be a local variable. Since 'a' has not been initialized with value yet, its value cannot be updated.

Global Keyword

When a Global variable is modified inside a function. It creates a local copy of that global variable and updates that local variable. So, the value of the global variable is never updated.

Code	Output

a = "Global Hi"
def printHi():
  a = "Local Hi" # local variable a
  print("a inside: " + a)
 	
printHi()
print("a outside: " + a)
	a inside: Local Hi
	a outside: Global Hi

For solving this problem, before updating the global variable, the **global** keyword is used inside the

function. The **global** keyword makes the local variable work as a global variable or we can say, it binds the local variable with the global variable. Then, instead of creating a local copy like the previous code, the global variable's value is updated.

Code	Output

a = "Global Hi"
def printHi():
  global a #(problem solved)
  a = "Local Hi"
  print("a inside: " + a)
 	
printHi()
print("a outside: " + a)
	a inside: Local Hi
	a outside: Local Hi

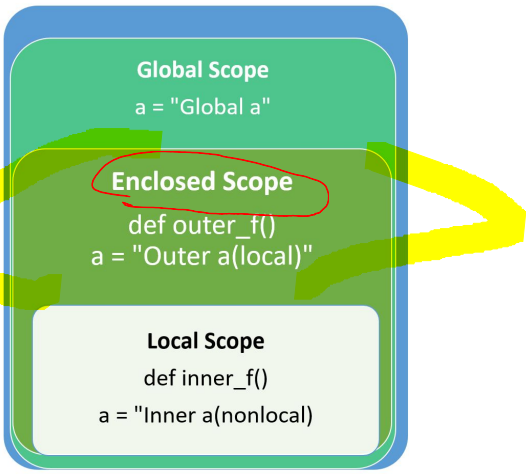
Enclosing Scope & nonlocal variables

Enclosing scopes are associated with nested functions. The variables declared in the enclosed scope are known as "nonlocal variables". They have a special scope which neither falls in the local nor the global scope. So, the enclosing scope is between the local and global scope. The keyword "nonlocal" is used to declare this kind of variable.

Code	Output

a = "Global a"
 	
def outer_f():
#==Scope of outer a starts======
  a = "Outer a(local)"
 	
  def inner_f():
  #==Scope of inner a starts======
  nonlocal a
  a = "Inner a(nonlocal)"
  print("Inside inner_f(): ", a)
  #==Scope of inner a ends======
 	
  inner_f()
  print("Inside outer_f(): ", a)
 	
##==Scope of outer a ends====
 	
outer_f()
print("In the main: ", a)
	Inside inner_f(): Inner a(nonlocal)
	
	Inside outer_f(): Inner a(nonlocal)
	
	In the main: Global a

```
global_scope_var = 5
OUTER SCOPE
def main():
  x = "main"
  INNER SCOPE
  def innerfunc():
    nonlocal x
    x = "modified"
  innerfunc()
  return x
print(main())
```



In the code above code, a function named `outer_f()` has been created with a nested inner function named `inner_f()`. The `inner_f()` has been defined within the scope of `outer_f()`. Thus, any modifications made to the nonlocal variable(a) will be reflected in the `outer_f()`'s variable a.

Non-local Keyword

It is mainly associated with nested functions. It declares in an inner nested function that the variable declared is non(not) local and permits us to make modifications to the outer function variable from the inner function scope.

When variables with the same name are used in nested functions, instead of using the variables declared in the outer function, it creates an inner function copy of the same variable with a scope of that created function. So any modifications made to the inner function variable will be reflected neither in the outer function variable nor the global variable.

For making modifications in the global variable from the inner function, we have to use the "global" keyword and for making changes in the outer functions from the inner function, we have to use the "nonlocal" keyword.

1. Example1: Nested function with same variable name.

Code	Output

`a = "Global a"`
 	
`def outer_f():`
`#==Scope of outer a starts======`
`  a = "Outer a"`
 	
`  def inner_f():`
`  #==Scope of inner a starts======`
	`Inside inner_f():`
	`Inner a`
	
	`Inside outer_f():`
	`Outer a`
	
	`In the main: Global a`

```
    a = "Inner a"
    print("Inside inner_f(): ", a)
    #==Scope of inner a ends======

    inner_f()
    print("Inside outer_f(): ", a)

##==Scope of outer a ends====

outer_f()
print("In the main: ", a)
```

2. Example 2: When we try to change the outer function variable from an inner function without using a "nonlocal" keyword, Python is unable to bind it with any variable (outer variable/global variable) and generates an error.

Code	Output

`a = "Global a"`
 	
`def outer_f():`
`#==Scope of outer a starts======`
`  a = "Outer a"`
 	
`  def inner_f():`
`  #==Scope of inner a starts======`
`  a = a + "Inner a" #(ERROR)`
`  print("Inside inner_f(): ", a)`
`  #==Scope of inner a ends======`
 	
`  inner_f()`
`  print("Inside outer_f(): ", a)`
 	
`##==Scope of outer a ends====`
 	
`outer_f()`
`print("In the main: ", a)`
	`UnboundLocalError: local variable 'a' referenced before assignment`

3. Example 3: When we try to change the outer function variable from an inner function using the "nonlocal" keyword, Python binds the inner function variable with the outer function variable. So any modifications made to the variable from the local function are reflected in the outer function variable.

Code	Output

a = "Global a"
#=======================
def outer_f():
  a = "Outer a "
  #=======================
  def inner_f():
  nonlocal a #**(Problem solved)**
  a = a + "Inner a"
  print("Inside inner_f(): ", a)
  #=======================
  inner_f()
  print("Inside outer_f(): ", a)
#=======================
outer_f()
print("In the main: ", a)
	Inside inner_f():
	Outer a Inner a
	
	Inside outer_f():
	Outer a Inner a
	
	In the main: Global a

4. Example 4: Using 'nonlocal' in the outer function will not bind it with the global variable, because nonlocal only works with nested functions. Thus it will generate an error.

Code	Output

a = "Global a "
#=======================
def outer_f():
  nonlocal a
  a = a + "Outer a " #**(ERROR)**
  #=======================
  def inner_f():
  a = "Inner a"
  print("Inside inner_f(): ", a)
  #=======================
  inner_f()
  print("Inside outer_f(): ", a)
#=======================
outer_f()
print("In the main: ", a)
	no binding for
	nonlocal 'a' found

For solving this problem, we need to use the "global" keyword in the outer function to bind it with the global variable.

Code	Output

```
a = "Global a "
#=======================
def outer_f():
     global a #(Problem solved)
     a = a + "Outer a "
     #=======================
     def inner_f():
          a = "Inner a"
          print("Inside inner_f(): ", a)
     #=======================
     inner_f()
     print("Inside outer_f(): ", a)
#=======================
outer_f()
print("In the main: ", a)
```

```
Inside inner_f():
Inner a

Inside outer_f():
Global a Outer a

In the main:  Global a
Outer a
```

5. Example 5: If none of the outer scope variables have the same name as the nonlocal variable of the inner function, it will generate an error. It will not bind with the global variable.

Code	Output

```
a = "Global a"

def outer_f():

     def inner_f1():

     def inner_f2():
          #==Scope of inner2 a starts======
          nonlocal a #(ERROR)
          a = "Inner2 a"
          print("Inside inner_f2(): ", a)
          #==Scope of inner2 a ends======

     inner_f2()
     print("Inside inner_f1(): ", a)

     inner_f1()
     print("Inside outer_f(): ", a)

outer_f()
print("In the main: ", a)
```

```
no binding for
nonlocal 'a' found
```

6. Example 6: if the immediate outer function does not have the same variable present, it will bind with the same variable present in the next outer function.

Code	Output

a = "Global a"
 	
def outer_f():
#==Scope of outer a starts======
  a = "Outer a "
 	
  def inner_f1():
  #missing variable a
 	
  def inner_f2():
  #==Scope of inner2 a starts======
  nonlocal a
  a = a + "Inner2 a"
  print("Inside inner_f2(): ", a)
  #==Scope of inner2 a ends======
 	
  inner_f2()
  print("Inside inner_f1(): ", a)
 	
  inner_f1()
  print("Inside outer_f(): ", a)
#==Scope of outer a ends======
 	
outer_f()
print("In the main: ", a)
	Inside inner_f2():
	Outer a Inner2 a
	Inside inner_f1():
	Outer a Inner2 a
	Inside outer_f():
	Outer a Inner2 a
	In the main:
	Global a

Built-in Scope

If the variable is not found in the local scope, Python by default starts searching for the variable in the immediate next outer scope. If the variable is not found in the local, enclosed, and global scope, then Python tries to find it in the built-in scope. If found then shows the value.

Example: In the following example, `pi_value` is directly imported from the math module and this variable has not been defined in local, global, or enclosed scope. So, Python is getting this value by looking for it in the built-in scope.

Code	Output
# Built-in Scope from math import pi as pi_value # missing pi_value variable	3.141592653589793

```
def outer_f():
     # missing pi_value variable

     def inner_f():
         # missing pi_value variable
         print(pi_value)

     inner_f()

outer_f()
```

Style Guide for Python Code

For every programming language, there are few coding conventions followed by the coding community of that language. All those conventions or rules are stored in a collected document manner for the convenience of the coders, and it is called the “Style Guide” of that particular programming language. The provided link gives the style guidance for Python code comprising the standard library in the main Python distribution.

Python style guide link: https://www.python.org/dev/peps/pep-0008/