浙江理工大学
ZHEJIANG SCI-TECH UNIVERSITY

7. Function (2)

《Python programming》 / Lecturer：Zhiyi Luo (罗志一)

School of Computer Science and Technology
计算机科学与技术学院

Scope and Lifetime of Variables

- When we define a function with variables, then those variables' scope is limited to that function. In Python, the scope of a variable is an area where a variable is declared. It is called the variable's local scope.
- We can not access the local variables from outside of the function. Because the scope is local, those variables are not visible from the outside of the function.

Scope and Lifetime of Variables

- When we are executing a function, the life of the variables is up to running time. Once we return from the function, those variables get destroyed. So function does no need to remember the value of a variable from its previous call.

Scope and Lifetime of Variables

- The following code shows the scope of a variable inside a function.

```python
global_lang = 'DataScience'

def var_scope_test():
    local_lang = 'Python'
    print(local_lang)

var_scope_test()  # Output 'Python'

# outside of function
print(global_lang)  # Output 'DataScience'

# NameError: name 'local_lang' is not defined
print(local_lang)
```

Local Variable in function

- A local variable is a variable declared inside the function that is not accessible from outside of the function. The scope of the local variable is limited to that function only where it is declared.
- If we try to access the local variable from the outside of the function, we will get the error as NameError.

Local Variable in function

```
def function1():
    # local variable
    loc_var = 888
    print("Value is :", loc_var)

def function2():

    print("Value is :", loc_var)

function1()
function2()
```

Value is : 888 print("Value is :", loc_var) # gives error, NameError: name 'loc_var' is not defined

Global Variable in function

- A global variable is a variable that declares outside of the function. The scope of a global variable is board. It is accessible in all functions of the same module.

```
global_var = 999

def function1():
    print("Value in 1nd function :", global_var)

def function2():
    print("Value in 2nd function :", global_var)

function1()
function2()
```

Global Variable in function

- In Python, global is the keyword used to access the actual global variable from outside the function. We use the global keyword for two purposes:
    - To declare a global variable inside the function
    - Declaring a variable as global, which makes it available to function to perform the modification

Global Variable in function

- Let's see what happens when we don't use global keyword to access the global variable

```
global_var = 5  # Global variable
def function1():
    print("Value in 1st function :", global_var)

def function2():
    # function will treat `global_var` as a local variable
    global_var = 555
    print("Value in 2nd function :", global_var)

def function3():
    print("Value in 3rd function :", global_var)

function1()
function2()
function3()
```

Global Variable in function

```
x = 5  # Global variable

# defining 1st function
def function1():
    print("Value in 1st function :", x)

# defining 2nd function
def function2():
    # Modify global variable using global keyword
    global x
    x = 555
    print("Value in 2nd function :", x)

# defining 3rd function
def function3():
    print("Value in 3rd function :", x)

function1()
function2()
function3()
```

Nonlocal Variable in function

- In Python, nonlocal is the keyword used to declare a variable that acts as a global variable for a nested function (i.e., function within another function).
- We can use a nonlocal keyword when we want to declare a variable in the local scope but act as a global scope.

Nonlocal Variable in function

```
def outer_func():
    x = 777

    def inner_func():
        # local variable now acts as global variable
        nonlocal x
        x = 700
        print("value of x inside inner function is :", x)

    inner_func()
    print("value of x inside outer function is :", x)

outer_func()
```

绑定、解析与作用域

函数内部绑定的名称，作用域局限于函数范围；它与外部的相同名称可以具有不同的值，互不影响；函数的参数也看作函数内部绑定的名称。相同函数每次调用的作用域也是互相独立的。

```
a = 1
def func():
    a = 2
    return a
b = func()   # 2
a            # 1
```

在绑定之前就使用了相应名称的变量会报错

```
def func():
    b = a      # 报错
    a = 1
```

绑定、解析与作用域

- 由于相同名称在不同作用域中可以有不同的值，从名称获取值的过程称为解析
- 函数内部名称的解析服从以下规则：
  - 如果这一名称在函数内的任意语句中被绑定（包括普通绑定、特殊绑定以及参数绑定），则该名称解析的作用域是当前作用域
  - 否则，该名称将被解析到外部作用域

```
a = 1
def func():
    print a    # 1
def func2():
    print a    # 报错
    a = 2
```

绑定、解析与作用域

- 函数定义时，每个名称是解析到当前作用域还是外部作用域就已经决定
  - 即使绑定语句从未执行，也会在解析时使用当前作用域

```
a = 2
def func():
    print a # 报错
    if False:
        a = 1
```

- 名称解析到具体的值是在运行时进行的

```
def func():
    return a
a = 1
func() # 1
a = 2
func() # 2
del a
func() # 报错
```

嵌套调用与递归

- 函数内调用其他函数的过程，是首先将要调用的函数名称解析到对应的函数对象，然后执行函数对象，解析函数名称这一步也是在运行时进行的
- 因此，函数内部可以调用尚未定义的函数，包括自己

```
def a(n):
    return b(n-1) if n else 0
def b(n):
    return a(n-1) if n else 1
def c(n):
    return c(n-1) + c(n-2) if n > 2 else 1
```

作用域嵌套

- 定义函数是单纯的名称绑定，因此在函数内部也可以定义新的函数
- 新定义的函数有自己的作用域，同时也继承外层函数的作用域
- 如果某个名称在最内层作用域中没有被绑定，则会依次寻找外层作用域，如果外层作用域中被绑定，则绑定到外层作用域，否则再寻找更外层作用域
- 作用域嵌套只与定义时的嵌套关系有关，与调用无关；名称不会解析到调用时的作用域中

```
def func1(a, b):
    def func2():
        return a+b
    return func2
```