Python中的命名空间与作用域

> *“Namespaces are one honking great idea.”——import this*

一. 标识符：identifier

标识符（identifier）用于命名程序中标识像变量和函数这样的元素（梁勇，python语言程序设计）。例如，在赋值语句x=3中，x即为一个标识符。因此，标识符可以被简单理解为“名称（name）”。Python中一切皆对象，因此标识符就是程序员给各种对象所起的各种名称。

```
x = "global x"

def func():
    x = "local x"
    print(x)

func()
print(x)
```

在上面的代码中，x、func、print都是标识符：x是变量名，func和print是函数名；x和func是我自己起的名称，而print则是python内置的名称；x（def外）这个标识符是字符串对象global x的名称，func这个标识符是其后定义的函数对象的名称，x（def内）这个标识符是字符串对象local x的名称，print这个标识符是python内置的打印函数的名称。

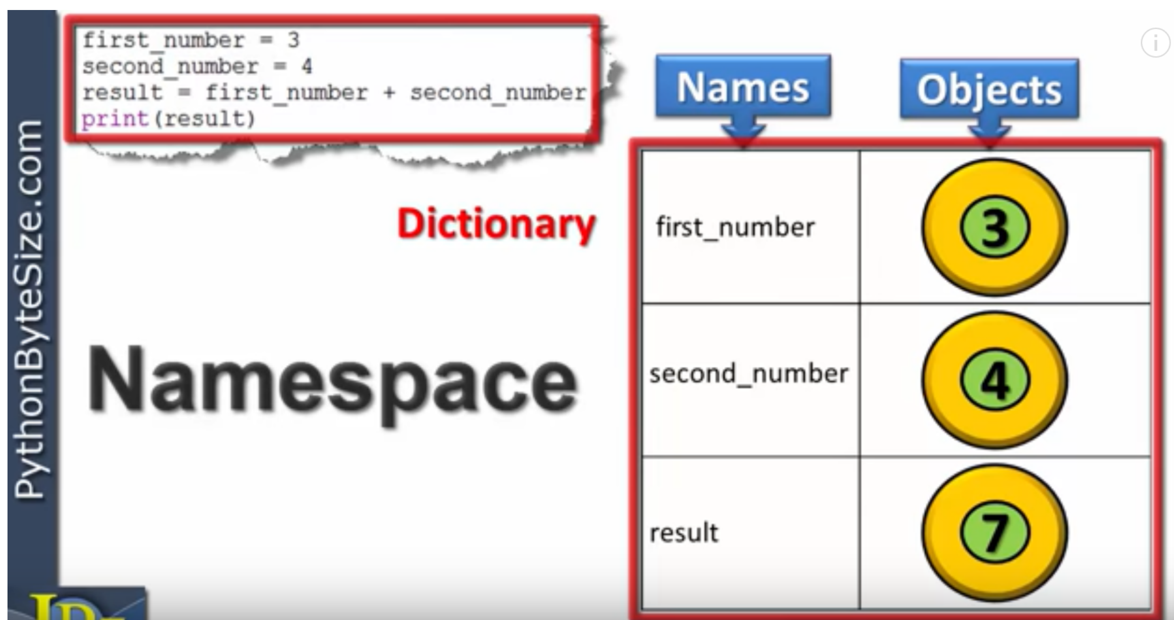
这里大家可能会有疑问：python不会将两个x标识符混淆吗？不会，因为它们虽然名称相同，但是位于不同的**命名空间**中。为什么它们会处于不同的命名空间中？因为它们创建的位置不同——一个在顶层（所有函数外），一个在函数内。

二. 命名空间：namespace

> *Python官方文档对命名空间的描述：*
>
> *A **namespace** is a mapping from names to objects......**Examples of namespaces** are: ①the set of built-in names; ②the global names in a module; ③the local names in a function invocation......The important thing to know about namespaces is that there is **absolutely no relation** between names in different namespaces...... Namespaces are created at different moments and have different **lifetimes**: ①The namespace containing the built-in names is created when the Python interpreter starts up, and is never deleted; ②The global namespace for a module is created when the module definition is read in and module namespaces also last until the interpreter quits; ③The local namespace for a function is created when the function is called, and deleted when the function returns or raises an exception that is not handled within the function——Of course, **recursive invocations each** have their own local namespace.*

命名空间是程序用来组织标识符的地方，就好比“电话簿”：标识符类似于“姓名”，而标识符引用的对象类似于“电话号码”；程序在运行时遇到某个标识符时，就会来这本“电话簿”中查看该标识符对应的对象，再进行此后的操作。因此，命名空间的本质是**映射（mapping）**。在当前最新的python版本中，命名空间是字典：key是标识符，value是标识符所引用的object。

> *When you use a name in a program, Python creates, changes, or looks up the name in what is known as a namespace.*



在上图中：

- 程序执行第一行`first_number=3`时，会在命名空间里添加一个key为`first_number`、value为3的条目；
- 程序在执行第二行`second_number=4`时，会在命名空间中添加一个key为`second_number`、value为4的条目；
- 程序在执行第三行时：
    - 首先执行`first_number+second_number`：程序将在命名空间中找到`first_number`与`second_number`各自相连的对象，即3与4，然后进行相加的操作，得到7；
    - 再将7赋给标识符为`result`的变量：此时程序会在命名空间中添加一个key为`result`、value为7的条目；
- ……

在一段运行中的程序里，**命名空间不止一个。**但重要的是，这些命名空间相互独立，彼此互不干扰：

> *The important thing to know about namespaces is that there is **absolutely no relation** between names in different namespaces.*

Python中有三种类型的命名空间，每一类专门储存某些标识符，且每一类有自己的生命周期（lifetime）：

- **Built-in命名空间**：储存所有的built-in标识符；在python启动时就产生，在程序结束时消失。
- **Global命名空间**：储存模块（一个.py文件）中的global标识符（即不在任何类或函数中创建的名称，位于.py文件的top level，或者被关键字`global`标注）；在该模块被执行时创建，同样在程序结束时才消失。
- **Local命名空间**：储存函数（更严谨的说法：函数调用）中的local标识符（在函数中创建的名称，即位于def下的标识符）；在函数被调用时创建，在函数返回或抛出异常后消失——生命周期较短。

标识符创建后会立刻被分配到其中某一个命名空间。这里，标识符的创建在python中往往是指赋值运算的发生，例如`x=4`的执行，就创建了标识符x。至于被分配到哪一个命名空间中，则取决于该创建发生的位置：一般来说，在.py文件的top level下创建的标识符，会被分配到global命名空间；在函数内创建的标识符，会被分配到该函数的local命名空间。

> *The location of a name's assignment in your source code determines the scope of the name's visibility to your code.*

下面的代码详细解释了一个程序从开始执行到结束的过程中，命名空间的创建与消失：

```
### 1. Start: The built-in namespace and global namespace is created ###

x = "global x"
### 2. Add {key=x, value="global x"} to the global namespace ###

def func():
    ### 3. Add {key=func, value=this function definition} to the global namespace ###
    x = "local x"
    print(x)

func()
### 4. Find the target function through name "func" from the global namespace ###
### 5. Enter in and the local namespace of func is created ###
### 6. x = "local x"
       ### Add {key=x, value="local x"} to the local namespace of func
       ### Do not make any conflict with another x, because the latter is in the global
namespace!
### 7. print(x)
       ### Find the target function through name "print" from the built-in namespace
       ### Find the value of argument through name "x" from the local namespace of func --
LEGB Rule
       ### Do printing operation
       ### After printing operation, the local namespace of func will be deleted because of
the end of this function call.

print(x)
### 8. Find the target function through name "print" from the built-in namespace
### 9. Find the value of argument through name "x" from the global namespace.
### 10. Do printing operation

### 11. End. Delete global namespace and built-in namespace.
```

三. 作用域：scope

> *A scope is a textual region of a Python program where a namespace is directly accessible. “Directly accessible” here means that an unqualified reference to a name attempts to find the name in the namespace.*

作用域（scope）是一个文本区域，表示某个namespace可以被直接“访问”的区域——这里的直接访问是指不使用点运算符的访问。Python中有四种类型的作用域：

1. Built-in Scope：只有built-in命名空间才拥有这种最广的作用域，即built-in命名空间中的任何标识符可以在任何模块中的任何位置直接访问；
2. Global Scope：global作用域，即全局作用域，可以在模块的任何地方直接访问，global命名空间拥有这种较广的作用域，即位于global命名空间中的标识符可以在模块内任何位置被访问到；
3. Enclosing Scope：enclosing作用域，简单来说就是enclosing function（内部还有函数的函数，也可以称为“外围函数”，其内部包裹的函数被称为“嵌套函数”，英文为nested function）的local命名空间所拥有的作用域；enclosing function的local命名空间能够被其内部的函数访问，所以比一般的local命名空间的作用域要大一些；

4. Local Scope：local作用域，即本地作用域，是local命名空间拥有的作用域，表示local命名空间中的标识符只能在该函数内部直接访问，一般也指nested function的local命名空间所拥有的作用域。

附：enclosing function与nested function——

```
def outer():
    x = 3
    def inner():
        print(x)
    inner()
```

上面的代码中，函数outer就是一个enclosing function，函数inner则是一个nested function。调用enclosing function后会立刻创建一个该enclosing function的local命名空间，该命名空间具有enclosing scope，比一般的local scope大，因为能够被它的内部函数所访问到。在outer函数的内部调用inner函数后，会立刻创建一个该nested function的local命名空间，该命名空间的scope即为local scope，而非enclosing scope。

再举个例子：

```
x = "global x"

def outer():
    y = "local outer y"
    def inner():
        z = "local inner z"
        print(z)
    inner()
    print(y)

outer()
print(x)
```

上面的代码中：

- x位于global命名空间，具有global scope；
- y位于local命名空间，但是在一个enclosing函数中，所以具有enclosing scope；
- z位于local命名空间，所以具有local scope。

四. LEGB法则

> *As you saw before, namespaces can exist independently from eachother, and have certain levels of hierarchy, which we refer to as their scope. Depending on where you are in a program, a different namespace will be used. To determine in which order Python should access namespaces, you can use the LEGB rule.*

不同命名空间的作用域可能重合。那么在重合的位置访问一个“同名”的标识符，返回的结果是什么呢？例如下面的代码中，最后调用test函数，其中的`print(x)`将打印"local x"还是"global x"呢？因为位于global命名空间中的x拥有global作用域，能够在函数内被直接访问；同时，函数内部的x拥有local作用域，当然也能在自己的内部被访问。因此，在函数内打印x，打印的是哪一个x？

```python
x = "global x"

def test():
    x = "local x"
    print(x)

test()
```

这里就需要用到LEGB法则，即python按照“local→enclosing→global→built-in”的作用域顺序，依次查找相应大小的命名空间，直到找到目标标识符或没有找到目标标识符——没有找到则会抛出异常。简单来说，LEGB就是一种查找顺序。还是以上面的代码为例：

- 调用test函数，即进入test函数体，此时python会为该函数调用创建一个local命名空间；
- 随着函数体内`x="local x"`的执行，test函数的local命名空间内记录了该x和它的引用对象；
- 随后在函数体内执行`print(x)`，python开始按照LEGB顺序查找该标识符`x`背后的引用对象：直接在test函数的local命名空间找到了x，因此就停止查找，直接返回test函数的local命名空间中x所引用的对象，即字符串`"local x"`，所以打印的是`"local x"`。
- 如果在`test()`下一行增加一行`print(x)`：python依旧按照LEGB顺序查找该标识符，不过此时并没有local命名空间，因为已经退出了test函数，所以test函数的local命名空间被回收了，因此直接在global命名空间中查找——找到了`x="global x"`，因此打印的是`"global x"`。

Attention：LEGB法则是需要查找某标识符背后的对象时使用的查找顺序规则——**Name references** search at most four scopes: local, then enclosing functions (if any), then global, then built-in，与赋值时改变哪个命名空间中的标识符无关——**Name assignments** create or change local names by default，即赋值只于赋值发生的位置有关：赋值发生在global层（模块的top level），则只于global命名空间有关；赋值发生在一个函数中，则只于该函数的local命名空间有关。

五. Closure & Enclosing function

我们将Enclosing function译为外围函数，表示该函数内部还有嵌套的函数。嵌套函数的标识符（名字）位于外围函数的local命名空间中，具有enclosing scope，因此：

```python
def outer():
    def inner():
        print(inner)
    inner()

outer()
# 返回<function outer.<locals>.inner at 0x000001E38FE62378>
```

Attention：除非要实现“闭包(Closure)”，否则不必非要使用外围-嵌套结构的函数。即，Python中闭包的实现，enclosing function是必要条件。

什么是闭包？

> *Factory functions (a.k.a. closures) are sometimes used by programs that need to generate event handlers on the fly in response to conditions at runtime. For instance, imagine a GUI that must define actions according to user inputs that cannot be anticipated when the GUI is built. In such cases, we need a function that creates and returns another function, with information that may vary per function made.——《Python学习手册(第5版)》，p501*

闭包的运用场景：需要在程序运行时根据一些临时响应构造新的处理模块。例如，想象一个图形界面，该图形界面需要根据用户的输入定义一个动作，因此该动作的定义一定发生在运行时，而不是程序运行前；因此，我们需要在程序中编写这样一个函数，该函数能够在运行时根据不同的场景构造并返回另一个“定制版”的函数。这样的函数就叫做闭包，也叫做工厂函数。

一个简单的例子：

```
def outer(n):
    def inner(a):
        return a ** n
    return inner

f = outer(2)
print(f(5))     # 25
g = outer(3)
print(g(5))     # 125
```

上面的代码中，outer函数的功能仅仅是构造并返回一个名为inner的nested函数，而没有去调用它。更精确地，我们调用outer函数后，得到的返回值是对“生成”的inner函数的引用——如果我们调用它，就相当于调用了inner函数。这里有两点需要注意：

1. 闭包使我们可以在外围函数的外部调用外围函数内部的嵌套函数——按理说，嵌套函数的标识符仅存在于enclosing scope，而非global scope；但闭包巧妙地将函数的引用（因为函数也是对象）作为返回值，使得在外围函数外通过引用访问嵌套函数成为可能；
2. 闭包返回的嵌套函数的引用，“记住”了其使用的enclosing scope中的值。上例中outer(2)记住了2，outer(3)记住了3——好像有悖常理，因为外围函数在返回嵌套函数的引用后就结束了，那么它的local命名空间也就释放掉了，参数n的值也应该随之消失才对，怎么会被嵌套函数的引用“记住”呢？解释是："*n* from the enclosing local scope is retained as state information attached to the generated *inner*."

最后，the following are the conditions that are required to be met in order to create a closure in Python:

1. There must be a nested function.
2. The inner function has to refer to a value that is defined in the enclosing scope.
3. The enclosing function has to return the nested function.

六. Global & Nonlocal

`global`是一个“命名空间声明(namespace declaration)”：The global statement tell Python that a function plans to change one or more global names(Python学习手册). 也就是说，被global标记的标识符，将被储存于global命名空间，而不是默认的local命名空间。下例：

```
X = 88

def func():
    global X
    X = 99

func()
print(X)
```

这段代码的输出是99，而不是88——一般情况下，func函数若被执行，其内部的X则储存于local命名空间，具有local scope，那么在函数内对X进行赋值等修改，根据LEGB规则，应该是对其local命名空间中的X进行操作，怎么会影响到global命名空间中的X呢？这是因为在函数内部，X已经被声明是global的……

- Names declared in global and nonlocal statements map assigned names to enclosing module and function scopes, respectively.

- If a variable is assigned in an enclosing def, it is nonlocal to nested functions.