



微信扫一扫
关注该公众号

收录于合集

#CPython

97个 >



楔子

这里再回顾一下函数的local空间，首先我们往global空间添加一个键值对相当于定义一个全局变量，那么如果往函数的local空间里面添加一个键值对，是不是也等价于创建了一个局部变量呢？

```
1 def f1():
2     locals()["name"] = "夏色祭"
3     try:
4         print(name)
5     except Exception as e:
6         print(e)
7
8 f1() # name 'name' is not defined
```

对于全局变量来讲，变量的创建是通过向字典添加键值对的方式实现的。因为全局变量会一直在变，需要使用字典来动态维护。

但对于函数来讲，内部的变量是通过静态方式存储和访问的，因为局部作用域中存在哪些变量在编译的时候就已经确定了，我们通过PyCodeObject的co_varnames即可获取内部都有哪些变量。

所以，虽然我们说查找是按照LGB的方式查找，但是访问函数内部的变量其实是静态访问的，不过完全可以按照LGB的方式理解。关于这方面的细节，后续还会细说。

因此名字空间是Python的灵魂，它规定了Python变量的作用域，使得Python对变量的查找变得非常清晰。



前面说的LGB是针对Python2.2之前的，而从Python2.2开始，由于引入了嵌套函数，所以最好的方式应该是内层函数找不到某个变量时先去外层函数找，而不是直接就跑到global空间里面找。那么此时的规则就是LEGB：

```
1 a = 1
2
3 def foo():
4     a = 2
5
6     def bar():
7         print(a)
8     return bar
9
10 f = foo()
11 f()
12 """
13 2
14 """
```

调用f，实际上调用的是函数bar，最终输出的结果是2。如果按照LGB的规则来查找的话，由于函数bar的作用域没有a、那么应该到全局里面找，打印的结果是1才对。

但是我们之前说了，作用域仅仅是由文本决定的，函数bar位于函数foo之内，所以函数bar定义的作用域内嵌于函数foo的作用域之内。换句话说，函数foo的作用域是函数bar的作用域的直接外围作用域。

所以应该先从foo的作用域里面找，如果没有那么再去全局里面找。而作用域和名字空间是对应的，所以最终打印了2。

另外在执行f = foo()的时候，会执行函数foo中的def bar():语句，这个时候解释器会将a=2与函数bar捆绑在一起，然后返回，这个捆绑起来的整体就叫做闭包。

所以：闭包 = 内层函数 + 引用的外层作用域

这里显示的规则就是LEGB，其中E表示enclosing，代表直接外围作用域。



global表达式

有一个很奇怪的问题，最开始学习Python的时候，笔者也为此困惑了一段时间，下面来看一下。

```
1 a = 1
2
3 def foo():
4     print(a)
5
6 foo()
7 """
8 1
9 """
```

首先这段代码打印1，这显然是没有问题的，不过下面问题来了。

```
1 a = 1
2
3 def foo():
4     print(a)
5     a = 2
```

```

6
7  foo()
8  """
9  UnboundLocalError: local variable 'a' referenced before assignment
10 """

```

仅仅是在print语句后面新建了一个变量a，结果就报错了，提示局部变量a在赋值之前就被引用了，这是怎么一回事，相信肯定有人为此困惑。而想弄明白这个错误的原因，需要深刻理解两点：

- 一个赋值语句所定义的变量，在这个赋值语句所在的整个作用域内都是可见的；
- 函数中的变量是静态存储、静态访问的，内部有哪些变量在编译的时候就已经确定；

在编译的时候，因为a = 2这条语句，所以知道函数中存在一个局部变量a，那么查找的时候就会在当前作用域中查找。但是还没来得及赋值，就print(a)了，所以报错：局部变量a在赋值之前就被引用了。但如果没有a = 2这条语句则不会报错，因为知道局部作用域中不存在a这个变量，所以会找全局变量a，从而打印1。

更有趣的东西隐藏在字节码当中，我们可以通过反汇编来查看一下：

```

1  import dis
2
3  a = 1
4
5  def g():
6      print(a)
7
8  dis.dis(g)
9  """
10  7          0 LOAD_GLOBAL          0 (print)
11             2 LOAD_GLOBAL          1 (a)
12             4 CALL_FUNCTION        1
13             6 POP_TOP
14             8 LOAD_CONST          0 (None)
15             10 RETURN_VALUE
16  """
17
18  def f():
19      print(a)
20      a = 2
21
22  dis.dis(f)
23  """
24  12          0 LOAD_GLOBAL          0 (print)
25             2 LOAD_FAST            0 (a)
26             4 CALL_FUNCTION        1
27             6 POP_TOP
28
29  13          8 LOAD_CONST            1 (2)
30             10 STORE_FAST           0 (a)
31             12 LOAD_CONST            0 (None)
32             14 RETURN_VALUE
33  """
34

```

中间的序号代表字节码的偏移量，我们先看第二条，g的字节码是LOAD_GLOBAL，意思是在global名字空间中查找；而f的字节码是LOAD_FAST，表示在local名字空间中查找。因此结果说明Python采用了静态作用域策略，在编译的时候就已经知道了名字藏身于何处。

而且上面的例子也表明，一旦函数内有了对某个名字的赋值操作，这个名字就会在作用域内可见，就会出现在local名字空间中。换句话说，会遮蔽外层作用域中相同的名字。

当然Python也为我们精心准备了global关键字，让我们在函数内部修改全局变量。比如

函数内部出现了`global a`，就表示我后面的a是全局的，直接到global名字空间里面去找，不要在local空间里面找了。

```
1 a = 1
2
3 def bar():
4     def foo():
5         global a
6         a = 2
7     return foo
8
9 bar()()
10 print(a) # 2
11 # 当然, 也可以通过globals函数拿到名字空间
12 # 然后直接修改里面的键值对
```

但如果外层函数里面也出现了变量a，而我们想修改的也是外层函数的a、不是全局的a，这时该怎么办呢？Python同样为我们准备了关键字: nonlocal，但是使用nonlocal的时候，必须是在内层函数里面。

```
1 a = 1
2
3 def bar():
4     a = 2
5     def foo():
6         nonlocal a
7         a = "xxx"
8     return foo
9
10 bar()()
11 print(a) # 1
12 # 外界依旧是1, 但是bar里面的a已经被修改了
```

属性引用与名字引用

属性引用实质上也是一种名字引用，其本质都是到名字空间中去查找一个名字所引用的对象。这个就比较简单了，比如`a.xxx`，就是到a里面去找属性xxx，这个规则是不受LEGB作用域限制的，就是到a里面查找，有就是有、没有就是没有。

但是有一点需要注意，我们说查找会按照LEGB规则，但这必须限制在自身所在的模块内，如果是多个模块就不行了。举个栗子：

```
1 # a.py
2 print(name)
```

```
1 # b.py
2 name = "夏色祭"
3 import a
```

关于模块的导入我们后续会详细说，总之目前在b.py里面执行的`import a`，你可以简单认为就是把a.py里面的内容拿过来执行一遍即可，所以这里相当于`print(name)`。

但是执行b.py的时候会提示变量name没有被定义，可把a导进来的话，就相当于`print(name)`，而我们上面也定义name这个变量了呀。

显然，即使我们把a导入了进来，但是a.py里面的内容依旧是处于一个模块里面。而我们也说了，名称引用虽然是LEGB规则，但是无论如何都无法越过自身所在的模块。

print(name)在a.py里面，而变量name被定义在b.py里面，所以不可能跨过模块a的作用域去访问模块b里面的name，因此在执行 `import a` 的时候会抛出 `NameError`。

所以我们发现，虽然每个模块内部的作用域规则有点复杂，因为要遵循LEGB；但模块与模块之间的作用域还是划分的很清晰的，就是相互独立。

关于模块，我们后续会详细说。总之通过 `.` 的方式，本质上都是去指定的名字空间中查找对应的属性。



属性空间

我们知道，自定义的类里面如果没有 `__slots__`，那么这个类的实例对象都会有一个属性字典。

```
1 class Girl:
2
3     def __init__(self):
4         self.name = "古明地觉"
5         self.age = 16
6
7
8 g = Girl()
9 print(g.__dict__) # {'name': '古明地觉', 'age': 16}
10
11 # 对于查找属性而言，也是去属性字典中查找
12 print(g.name, g.__dict__["name"]) # 古明地觉 古明地觉
13
14 # 同理设置属性，也是更改对应的属性字典
15 g.__dict__["gender"] = "female"
16 print(g.gender) # female
```

当然模块也有属性字典，本质上和类的实例对象是一致的。

```
1 import builtins
2
3 print(builtins.str) # <class 'str'>
4 print(builtins.__dict__["str"]) # <class 'str'>
5
6 # 另外，有一个内置的变量 __builtins__，和导入的 builtins 等价
7 print(__builtins__ is builtins) # True
```

另外这个 `__builtins__` 位于 `global` 名字空间里面，然后获取 `global` 名字空间的 `globals` 又是一个内置函数，于是神奇的事情就出现了。

```
1 print(globals()["__builtins__"].globals()["__builtins__"].
2       globals()["__builtins__"].globals()["__builtins__"].
3       globals()["__builtins__"].globals()["__builtins__"]
4       ) # <module 'builtins' (built-in)>
5
6 print(globals()["__builtins__"].globals()["__builtins__"].
7       globals()["__builtins__"].globals()["__builtins__"].
8       globals()["__builtins__"].globals()["__builtins__"].list("abc")
9       ) # ['a', 'b', 'c']
```

所以 `global` 名字空间和 `builtin` 名字空间，都保存了指向彼此的指针，不管套娃多少次，都是可以的。

在 Python 中，一个名字(变量)的可见范围由[作用域](#)决定，而作用域由语法静态划分，划分规则提炼如下：

- .py文件(模块)最外层为全局作用域；
- 遇到函数定义，函数体形成子作用域；
- 遇到类定义，类定义体形成子作用域；
- 名字仅在其作用域以内可见；
- 全局作用域对其他所有作用域可见；
- 函数作用域对其直接子作用域可见，并且可以传递(闭包)；

与[作用域](#)相对应，Python在运行时借助PyDictObject对象保存作用域中的名字，构成动态的[名字空间](#)。这样的名字空间总共有 4 个：

- 局部名字空间(local)：不同的函数，局部名字空间不同，可以通过调用 locals 获取；
- 全局名字空间(global)：全局唯一，可以通过调用 globals 获取；
- 闭包名字空间(enclosing)；
- 内置名字空间(builtin)：可以通过调用 `__builtins__.__dict__` 获取；

查找名字时会按照LEGB规则查找，但是注意：无法跨越文件本身，也就是按照自身文件的LEGB。如果属性查找都找到builtin空间了，那么证明这已经是最后的倔强。如果builtin空间再找不到，那么就只能报错了，不可能跑到其它文件中找。

收录于合集 [#CPython 97](#)

[< 上一篇](#)

《源码探秘 CPython》49. 虚拟机是怎么执行字节码的？

[下一篇 >](#)

《源码探秘 CPython》47. 名字、作用域、名字空间（上）

喜欢此内容的人还喜欢

MySQL全面瓦解28：分库分表
架构与思维



Ramda 哪些让人困惑的函数签名规则
Tecvan



React 比较类和函数的两种创建组件方式（1） - 概述
老李物语

