Python 闭包专题总结

python 函数嵌套 闭包

总第273篇原创

1 Python 函数式编程

python 支持函数式编程，提到函数式编程，大家首先想到的是多个函数内嵌。的确是这样。不过，要想入门函数式编程，里面涉及到的闭包，是不得不掌握的，换句话说，如果不了解闭包就使用函数式编程，那么，函数式编程的功能特性可能不会完全体现出来。

今天用专题的形式，完整的总结下函数式编程中这个非常重要的特性：闭包，并提供PDF下载，如有补充指正，请留言，万分感激。

本资料为 **Python与算法社区** 出品，如需转载，请注明来源。

为什么一直在啰嗦闭包，我们都知道函数式编程中闭包处处存在，Python也支持函数式编程，自然也就存在闭包。

利用闭包的性质，我们可实现一些比较接地气的功能，调用起来比较容易理解的。

下面，从闭包是什么，闭包示例，使用坑点展开。

2 闭包是什么

闭包是由`函数及其相关的引用环境组合而成的实体` ，一句话：闭包 = 函数+引用环境。

函数式编程中，当`内嵌函数体`内引用到`体外的变量`时，将会连同这些变量(引用环境)和内嵌函数体，一块打包成一个整体返回。

3 闭包示例

编写一个能体现闭包特性，使用闭包给我们带来便利的经典例子。

这个例子是这样，实现机器人robot位置的实时更新功能。 定义机器人的初始位置，然后返回一个move函数，它能帮助我们实现机器人x，y 某个或两个方向上的移动，并打印出当前的位置。

构建一个外部函数，传递initx,inity 两个参数，代表robet的初始位置，然后内嵌一个move函数，体内要引用cordx，cordy 两个参数，这就是所谓的环境，它们+move函数组成闭包。

```
def rundist(initx,inity):
 cordx,cordy = initx,inity
```

```
 def move(**kwargs):
  pass
 return move
```

move函数的形参我们使用关键词参数kwargs，我们约定好它的两个参数为x,y，当传递过来x时，更新x方向的距离，如果都传过来，则说明x,y两个方向都有了移动。

下面，写move函数体内实现：

```
def move(**kwargs):
 nonlocal cordx
 nonlocal cordy
 if 'x' in kwargs.keys():
  cordx+=kwargs['x']
 if 'y' in kwargs.keys():
  cordy+=kwargs['y']
  print('current position (%d,%d)' %(cordx,cordy))
```

首先，显示地声明 cordx，cordy为非局部性变量，至于为什么会这样，下面会说到。然后分别判断传入的关键词参数是否包含x,y，有则更新，最后打印。

完整的代码如下：

```
In [21]: def rundist(x,y):
    ...:     cordx,cordy = x,y
    ...:     def move(**kwargs):
    ...:         nonlocal cordx
    ...:         nonlocal cordy
    ...:         if 'x' in kwargs.keys():
    ...:             cordx+=kwargs['x']
    ...:         if 'y' in kwargs.keys():
    ...:             cordy+=kwargs['y']
    ...:         print('current position (%d,%d)' %(cordx,cordy))
    ...:     return move
```

使用rundist函数，过程如下：

```
In [22]: mv = rundist(0,0)

In [23]: mv(x=1,y=3)
current position (1,3)

In [24]: mv(x=5)
current position (6,3)

In [25]: mv(y=3)
current position (6,6)
```

可以看到一次初始化位置，并返回一个move函数，下面不断调用move函数，并且在调用的时候就都能记忆住上一次的位置，比较方便。

这就是函数式编程中利用闭包特性的功能体现。

4 闭包使用坑点

4.1 nonlocal 作用

在上面的示例中，我们使用nonlocal关键词显示声明cordx不是局部变量，如果不这样做，会怎么样?

如下，我们去掉那两行代码:

```
In [21]: def rundist(x,y):
    ...:     cordx,cordy = x,y
    ...:     def move(**kwargs):
    ...:         if 'x' in kwargs.keys():
    ...:             cordx+=kwargs['x']
    ...:         if 'y' in kwargs.keys():
    ...:             cordy+=kwargs['y']
    ...:         print('current position (%d,%d)' %(cordx,cordy))
    ...:     return move
```

再次执行，

```
In [8]: mv = rundist(0,0)

In [9]: mv(x=1)
```

在执行 mv(x=1)时，报错 UnboundLocalError:

```
UnboundLocalError                         Traceback (most recent call last)
<ipython-input-9-3060599821a7> in <module>
----> 1 mv(x=1)

<ipython-input-6-a5d76037d2c4> in move(**kwargs)
      3     def move(**kwargs):
      4         if 'x' in kwargs.keys():
----> 5             cordx+=kwargs['x']
      6         if 'y' in kwargs.keys():
      7             cordy+=kwargs['y']

UnboundLocalError: local variable 'cordx' referenced before assignment
```

你可能会疑惑为什么?

这是因为，python 规则指定所有在赋值语句左面的变量都是局部变量，则在闭包 move() 中，变量 cordx 在赋值符号"="的左面，被 python 认为是 move() 中的局部变量。

再接下来执行 move() 时，程序运行至 cordx += x 时，因为之前已经把 cordx 归为 move() 中的局部变量了，因此，python 会在 move() 中去找在赋值语句右面的 cordx 的值，结果找不到，就会报错。

通过使用语句 `nonloacal cordx' 显式的指定 cordx 不是闭包的局部变量，避免出现 UnboundLocalError.

4.2 容易犯错

函数式编程新手，包括我自己，经常会犯一个错误，就是在内嵌函数内总是想修改体外的变量，如下:

```
In [10]: def run(x):
    ...:     cordx = x
    ...:     def move(x):
    ...:         cordx+=x
    ...:     return move
```

这和上面说道的cordx嵌入到move体内，且位于等号左侧时，自动被调整为move函数的局部变量，是一样的。 不过，对于我们刚入门函数式编程，这个错误是最容易犯的，使用注意就是声明cordx为非局部变量。

4.3 面试必考

有一道关于函数式编程考闭包的面试题，可以说是被各大公司都考过了，在网上一查就能找到这道题。

今天，我试着帮大家透彻解释清楚，希望未来参加面试的小伙伴，可以轻松拿下，不光知道答案，还知道为啥，最后叫面试官对你刮目相看。

先从一种比较好理解的方式入手，我们不使用 lambda，那样貌似把闭包隐蔽的太厉害了，不容易辨识出是闭包。

不过，下面这种方式，结合前几章节，还是比较容易就能看出来吧。

```
In [19]: def exfun():
    ...:     funli = []
    ...:     for i in range(3):
    ...:         def intfun(x):
    ...:            print( x*i)
    ...:         funli.append(intfun)
    ...:     return funli
    ...:
    ...:
```

就是生成了一个list,里面的3个元素，元素类型是intfun()函数，它是一个闭包，引用了外部变量i

下面，我们调用函数：

```
In [20]: funli = exfun()

In [21]: funli[0](5)
10

In [22]: funli[1](5)
10

In [23]: funli[2](5)
10
```

有些意外，返回的都是10，按照我们的期望应该是0,5,10

这是为什么？

原因： i 是闭包函数引用的外部作用域的变量，只有在内部函数被调用的时候， 才会搜索变量i的值。

由于循环已结束，i指向最终值2，所以各函数调用都得到了相同的结果。

如何解决这个问题？ 我们可以在生成闭包函数的时候，立即绑定变量 i，如下：

```
In [32]: def exfun():
    ...:     funli = []
    ...:     for i in range(3):
    ...:      def intfun(x,i=i):
    ...:       print( x*i)
    ...:        funli.append(intfun)
    ...:     return funli
    ...:
```

再次调用：

```
In [34]: funli[0](5)
0

In [35]: funli[1](5)
```

```
5
```

OK

真正在面试中，这道题会使用 lambda 进一步隐蔽闭包，呵呵，这回你可以识别出来了。

这是带陷阱的版本

```
In [38]: def exfun():
    ...:     return [lambda x: i*x for i in range(3)]
```

这是OK版本：

```
In [38]: def exfun():
    ...:     return [lambda x,i=i: i*x for i in range(3)]
```

5 总结

以上就是Python函数式编程，闭包的完整入门总结，希望能帮助到有这方面困惑的小伙伴。

本专题为 Python与算法社区 公众号出品，转载请注明出处。

Python与算法社区