

5.7. 使用dict和set

6. 函数

6.1. 调用函数

6.2. 定义函数

6.3. 函数的参数

6.4. 递归函数

7. 高级特性

7.1. 切片

7.2. 迭代

7.3. 列表生成式

7.4. 生成器

7.5. 迭代器

8. 函数式编程

8.1. 高阶函数

8.1.1. map/reduce

8.1.2. filter

8.1.3. sorted

8.2. 返回函数

8.3. 匿名函数

8.4. 装饰器

8.5. 偏函数

9. 模块

9.1. 使用模块

9.2. 安装第三方模块

10. 面向对象编程

10.1. 类和实例

10.2. 访问限制

10.3. 继承和多态

10.4. 获取对象信息

10.5. 实例属性和类属性

11. 面向对象高级编程

11.1. 使用_slots_

11.2. 使用@property

11.3. 多重继承

11.4. 定制类

11.5. 使用枚举类

11.6. 使用元类

12. 错误、调试和测试

12.1. 错误处理

12.2. 调试

12.3. 单元测试

12.4. 文档测试

13. IO编程

14. 进程和线程

15. 正则表达式

16. 常用内建模块

17. 常用第三方模块

返回函数



廖雪峰



资深软件开发工程师，业余马拉松选手。

高阶函数除了可以接受函数作为参数外，还可以把函数作为结果值返回。

我们来实现一个可变参数的求和。通常情况下，求和的函数是这样定义的：

```
def calc_sum(*args):
    ax = 0
    for n in args:
        ax = ax + n
    return ax
```

但是，如果不需要立刻求和，而是在后面的代码中，根据需要再计算怎么办？可以不返回求和的结果，而是返回求和的函数：

```
def lazy_sum(*args):
    def sum():
        ax = 0
        for n in args:
            ax = ax + n
        return ax
    return sum
```

当我们调用 `lazy_sum()` 时，返回的并不是求和结果，而是求和函数：

```
>>> f = lazy_sum(1, 3, 5, 7, 9)
>>> f
<function lazy_sum.<locals>.sum at 0x101c6ed90>
```

调用函数 `f` 时，才真正计算求和的结果：

```
>>> f()
25
```

在这个例子中，我们在函数 `lazy_sum` 中又定义了函数 `sum`，并且，内部函数 `sum` 可以引用外部函数 `lazy_sum` 的参数和局部变量，当 `lazy_sum` 返回函数 `sum` 时，相关参数和变量都保存在返回的函数中，这种称为“闭包（Closure）”的程序结构拥有极大的威力。

请再注意一点，当我们调用 `lazy_sum()` 时，每次调用都会返回一个新的函数，即使传入相同的参数：

```
>>> f1 = lazy_sum(1, 3, 5, 7, 9)
>>> f2 = lazy_sum(1, 3, 5, 7, 9)
>>> f1==f2
False
```

`f1()` 和 `f2()` 的调用结果互不影响。

闭包

注意到返回的函数在其定义内部引用了局部变量 `args`，所以，当一个函数返回了一个函数后，其内部的局部变量还被新函数引用，所以，闭包用起来简单，实现起来可不容易。

另一个需要注意的问题是，返回的函数并没有立刻执行，而是直到调用了 `f()` 才执行。我们来看一个例子：

```
def count():
    fs = []
    for i in range(1, 4):
        def f():
            return i*i
        fs.append(f)
    return fs
```

```
f1, f2, f3 = count()
```

5.7. 使用dict和set

6. 函数

- 6.1. 调用函数
- 6.2. 定义函数
- 6.3. 函数的参数
- 6.4. 递归函数

7. 高级特性

- 7.1. 切片
- 7.2. 迭代
- 7.3. 列表生成式
- 7.4. 生成器
- 7.5. 迭代器

8. 函数式编程

- 8.1. 高阶函数
 - 8.1.1. map/reduce
 - 8.1.2. filter
 - 8.1.3. sorted

8.2. 返回函数

- 8.3. 匿名函数
- 8.4. 装饰器
- 8.5. 偏函数

9. 模块

- 9.1. 使用模块
- 9.2. 安装第三方模块

10. 面向对象编程

- 10.1. 类和实例
- 10.2. 访问限制
- 10.3. 继承和多态
- 10.4. 获取对象信息
- 10.5. 实例属性和类属性

11. 面向对象高级编程

- 11.1. 使用__slots__
- 11.2. 使用@property
- 11.3. 多重继承
- 11.4. 定制类
- 11.5. 使用枚举类
- 11.6. 使用元类

12. 错误、调试和测试

- 12.1. 错误处理
- 12.2. 调试
- 12.3. 单元测试
- 12.4. 文档测试

13. IO编程

- 14. 进程和线程
- 15. 正则表达式
- 16. 常用内建模块
- 17. 常用第三方模块

在上面的例子中，每次循环，都创建了一个新的函数，然后，把创建的3个函数都返回了。

你可能认为调用 `f1()` , `f2()` 和 `f3()` 结果应该是 `1` , `4` , `9` , 但实际结果是:

```
>>> f1()
9
>>> f2()
9
>>> f3()
9
```

全部都是 `9` ! 原因就在于返回的函数引用了变量 `i` , 但它并非立刻执行。等到3个函数都返回时，它们所引用的变量 `i` 已经变成了 `3` , 因此最终结果为 `9` 。

△ 注意

返回闭包时牢记一点：返回函数不要引用任何循环变量，或者后续会发生变化的变量。

如果一定要引用循环变量怎么办？方法是再创建一个函数，用该函数的参数绑定循环变量当前的值，无论该循环变量后续如何更改，已绑定到函数参数的值不变：

```
def count():
    def f(j):
        def g():
            return j*j
        return g
    fs = []
    for i in range(1, 4):
        fs.append(f(i)) # f(i)立刻被执行，因此i的当前值被传入f()
    return fs
```

再看看结果：

```
>>> f1, f2, f3 = count()
>>> f1()
1
>>> f2()
4
>>> f3()
9
```

缺点是代码较长，可利用lambda函数缩短代码。

nonlocal

使用闭包，就是内层函数引用了外层函数的局部变量。如果只是读外层变量的值，我们会发现返回的闭包函数调用一切正常：

```
def inc():
    x = 0
    def fn():
        # 仅读取x的值:
        return x + 1
    return fn

f = inc()
print(f()) # 1
print(f()) # 1
```

但是，如果对外层变量赋值，由于Python解释器会把 `x` 当作函数 `fn()` 的局部变量，它会报错：

```
def inc():
    x = 0
    def fn():
        # nonlocal x
        x = x + 1
        return x
```

5.7. 使用dict和set

6. 函数

- 6.1. 调用函数
- 6.2. 定义函数
- 6.3. 函数的参数
- 6.4. 递归函数

7. 高级特性

- 7.1. 切片
- 7.2. 迭代
- 7.3. 列表生成式
- 7.4. 生成器
- 7.5. 迭代器

8. 函数式编程

- 8.1. 高阶函数
 - 8.1.1. map/reduce
 - 8.1.2. filter
 - 8.1.3. sorted
- 8.2. 返回函数
- 8.3. 匿名函数
- 8.4. 装饰器
- 8.5. 偏函数

9. 模块

- 9.1. 使用模块
- 9.2. 安装第三方模块

10. 面向对象编程

- 10.1. 类和实例
- 10.2. 访问限制
- 10.3. 继承和多态
- 10.4. 获取对象信息
- 10.5. 实例属性和类属性

11. 面向对象高级编程

- 11.1. 使用_slots_
- 11.2. 使用@property
- 11.3. 多重继承
- 11.4. 定制类
- 11.5. 使用枚举类
- 11.6. 使用元类

12. 错误、调试和测试

- 12.1. 错误处理
- 12.2. 调试
- 12.3. 单元测试
- 12.4. 文档测试

13. IO编程

14. 进程和线程

15. 正则表达式

16. 常用内建模块

17. 常用第三方模块

```
return fn

f = inc()
print(f()) # 1
print(f()) # 2
```

原因是 `x` 作为局部变量并没有初始化，直接计算 `x+1` 是不行的。但我们其实是想引用 `inc()` 函数内部的 `x`，所以需要在 `fn()` 函数内部加一个 `nonlocal x` 的声明。加上这个声明后，解释器把 `fn()` 的 `x` 看作外层函数的局部变量，它已经被初始化了，可以正确计算 `x+1`。

💡 提示

使用闭包时，对外层变量赋值前，需要先使用`nonlocal`声明该变量不是当前函数的局部变量。

练习

利用闭包返回一个计数器函数，每次调用它返回递增整数：

```
def createCounter():
    def counter():
        return 1
    return counter

# 测试:
counterA = createCounter()
print(counterA(), counterA(), counterA(), counterA()) # 1 2 3 4 5
counterB = createCounter()
if [counterB(), counterB(), counterB(), counterB()] == [1, 2, 3, 4]:
    print('测试通过!')
else:
    print('测试失败!')
```

参考源码

`return_func.py`

小结

一个函数可以返回一个计算结果，也可以返回一个函数。

返回一个函数时，牢记该函数并未执行，返回函数中不要引用任何可能会变化的变量。

« sorted

匿名函数 »

An advertisement for Gitee's DevOps platform. It features the Gitee logo and the text "一站式 DevOps 研发效能平台". Below it says "灵活选择部署方式 | 支持 SaaS 在线使用 | 私有化部署" and has a button labeled "进入 Gitee 官网".

Comments

>Loading comments...

©liaoxuefeng.com - 微博 - GitHub - License