

master ▾

...

the-craft-of-selfteaching / markdown / Part.2.D.4-recursion.md



xiaolai From wangpeiran 15 selfteaching#708



2 contributors



Raw

Blame



367 lines (257 sloc) 14.8 KB

递归函数

递归 (Recursion)

在函数中有个理解门槛比较高的概念：**递归函数** (Recursive Functions) —— 那些在自身内部调用自身的函数。说起来都比较拗口。

先看一个例子，我们想要有个能够计算 n 的**阶乘** (factorial) $n!$ 的函数， $f()$ ，规则如下：

- $n! = n \times (n-1) \times (n-2) \dots \times 1$
- 即, $n! = n \times (n-1)!$
- 且, $n \geq 1$

注意：以上是数学表达，不是程序，所以， $=$ 在这一小段中是“等于”的意思，不是程序语言中的赋值符号。

于是，计算 $f(n)$ 的 Python 程序如下：

```
def f(n):  
    if n == 1:  
        return 1  
    else:
```

```
        return n * f(n-1)

print(f(5))
```

120

递归函数的执行过程

以 factorial(5) 为例，让我们看看程序的流程：

f(5) 递归参数调用和值的返回过程					
	f(5): 5>1...				
		f(4): 4>1...			
			f(3): 3>1...		
				f(2): 2>1...	
					f(1): 1==1...
					return 1
				return 2 * f(1)	
			return 3 * f(2)		
		return 4 * f(3)			
	return 5 * f(4)				

当 f(5) 被调用之后，函数开始运行.....

- 因为 $5 > 1$ ，所以，在计算 $n * f(n-1)$ 的时候要再次调用自己 $f(4)$ ；所以必须等待 $f(4)$ 的值返回；
- 因为 $4 > 1$ ，所以，在计算 $n * f(n-1)$ 的时候要再次调用自己 $f(3)$ ；所以必须等待 $f(3)$ 的值返回；
- 因为 $3 > 1$ ，所以，在计算 $n * f(n-1)$ 的时候要再次调用自己 $f(2)$ ；所以必须等待 $f(2)$ 的值返回；
- 因为 $2 > 1$ ，所以，在计算 $n * f(n-1)$ 的时候要再次调用自己 $f(1)$ ；所以必须等待 $f(1)$ 的值返回；
- 因为 $1 == 1$ ，所以，这时候不会再次调用 $f()$ 了，于是递归结束，开始返回，这次返回的是 1 ；
- 下一步返回的是 $2 * 1$ ；
- 下一步返回的是 $3 * 2$ ；
- 下一步返回的是 $4 * 6$ ；
- 下一步返回的是 $5 * 24$ —— 至此，外部调用 $f(5)$ 的最终返回值是 120

加上一些输出语句之后，能更清楚地看到大概的执行流程：

```
def f(n):
    print('\tn =', n)
    if n == 1:
```

```

        print('Returning...')
        print('\tn =', n, 'return:', 1)
        return 1
    else:
        r = n * f(n-1)
        print('\tn =', n, 'return:', r)
        return r

print('Call f(5)...')
print('Get out of f(n), and f(5) =', f(5))

```

```

Call f(5)...
    n = 5
    n = 4
    n = 3
    n = 2
    n = 1
Returning...
    n = 1 return: 1
    n = 2 return: 2
    n = 3 return: 6
    n = 4 return: 24
    n = 5 return: 120
Get out of f(n), and f(5) = 120

```

有点烧脑..... 不过，分为几个层面去逐个突破，你会发现它真的很好玩。

递归的终点

递归函数在内部必须有一个能够让自己停止调用自己的方式，否则永远循环下去了.....

其实，我们所有人很小就见过递归应用，只不过，那时候不知道那就是递归而已。听过那个无聊的故事罢？

山上有座庙，庙里有个和尚，和尚讲故事，说.....

山上有座庙，庙里有个和尚，和尚讲故事，说.....

山上有座庙，庙里有个和尚，和尚讲故事，说.....

写成 Python 程序大概是这样：

```

def a_monk_telling_story():
    print('山上有座庙，庙里有个和尚，和尚讲故事，他说..... ')
    return a_monk_telling_story()

a_monk_telling_story()

```

这是个**无限循环**的递归，因为这个函数里**没有设置中止自我调用的条件**。无限循环还有个不好听的名字，叫做“死循环”。

在著名的电影**盗梦空间**（2010）里，从整体结构上来看，“入梦”也是个“递归函数”。只不过，这个函数和 `a_monk_telling_story()` 不一样，它并不是死循环——因为它设定了**中止自我调用的条件**：

在电影里，醒过来的条件有两个

- 一个是在梦里死掉；
- 一个是在梦里被 kicked 到.....

如果这两个条件一直不被满足，那就进入 limbo 状态——其实就跟死循环一样，出不来了.....

为了演示，我把故事情节改变成这样：

- 入梦，`in_dream()`，是个递归函数；
- 入梦之后醒过来的条件有两个：
 - 一个是在梦里死掉，`dead is True`；
 - 一个是在梦里被 kicked，`kicked is True`.....

以上两个条件中任意一个被满足，就苏醒.....

至于为什么会死掉，如何被 kick，我偷懒了一下：管它怎样，管它如何，反正，每个条件被满足的概率是 1/10.....（也只有这样，我才能写出一个简短的，能够运行的“**盗梦空间程序**”。）

把这个很抽象的故事写成 Python 程序，看看一次入梦之后能睡多少天，大概是这样：

```
import random

def in_dream(day=0, dead=False, kicked=False):
    dead = not random.randrange(0,10) # 1/10 probability to be dead
    kicked = not random.randrange(0,10) # 1/10 probability to be kicked
    day += 1
    print('dead:', dead, 'kicked:', kicked)

    if dead:
        print((f"I slept {day} days, and was dead to wake up..."))
        return day
    elif kicked:
        print(f"I slept {day} days, and was kicked to wake up...")
        return day

    return in_dream(day)

print('The in_dream() function returns:', in_dream())
```

```
dead: False kicked: False
dead: False kicked: False
dead: False kicked: False
dead: False kicked: False
dead: False kicked: False
dead: False kicked: False
dead: False kicked: False
dead: True kicked: True
I slept 8 days, and was dead to wake up...
The in_dream() function returns: 8
```

如果疑惑为什么 `random.randrange(0,10)` 能表示 1/10 的概率，请返回去重新阅读[第一部分中关于布尔值的内容](#)。

另外，在 Python 中，若是需要将某个值与 True 或者 False 进行比较，尤其是在条件语句中，推荐写法是（参见 [PEP8](#)）：

```
if condition:
    pass
```

就好像上面代码中的 `if dead:` 一样。

而不是（虽然这么写通常也并不妨碍程序正常运行^[1]）：

```
if condition is True:
    pass
```

抑或：

```
if condition == True:
    pass
```

让我们再返回来接着讲递归函数。正常的递归函数一定有个退出条件。否则的话，就无限循环下去了……下面的程序在执行一会儿之后就会告诉你：`RecursionError: maximum recursion depth exceeded`（上面那个“山上庙里讲故事的和尚说”的程序，真要跑起来，也是这样）：

```
def x(n):
    return n * x(n-1)
x(5)
```

```

RecursionError                                Traceback (most recent call
last)

<ipython-input-3-daa4d33fb39b> in <module>
      1 def x(n):
      2     return n * x(n-1)
----> 3 x(5)

<ipython-input-3-daa4d33fb39b> in x(n)
      1 def x(n):
----> 2     return n * x(n-1)
      3 x(5)

... last 1 frames repeated, from the frame below ...

<ipython-input-3-daa4d33fb39b> in x(n)
      1 def x(n):
----> 2     return n * x(n-1)
      3 x(5)

RecursionError: maximum recursion depth exceeded

```

不用深究上面盗梦空间这个程序的其它细节，不过，通过以上三个递归程序 —— 两个很扯淡的例子，一个正经例子 —— 你已经看到了递归函数的共同特征：

1. 在 `return` 语句中返回的是**自身的调用**（或者是**含有自身的表达式**）
2. 为了避免死循环，**一定要有至少一个条件**下返回的不再是自身调用.....

变量的作用域

再回来看计算阶乘的程序 —— 这是正经程序。这次我们把程序名写完整，`factorial()`：

```

def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n-1)

print(factorial(5))

```

120

最初的时候，这个函数的执行流程之所以令人迷惑，是因为初学者对**变量**的**作用域**把握得不够充分。

变量根据作用域，可以分为两种：全局变量（Global Variables）和局部变量（Local Variables）。

可以这样简化管理解：

- 在函数内部被赋值而后使用的，都是**局部变量**，它们的作用域是**局部**，无法被函数外的代码调用；
- 在所有函数之外被赋值而后开始使用的，是**全局变量**，它们的作用域是**全局**，在函数内外都可以被调用。

定义如此，但通常程序员们会严格地遵守一条原则：

在函数内部绝对不调用全局变量。即便是必须改变全局变量，也只能通过函数的返回值在函数外改变全局变量。

你也必须遵守同样的原则。而这个原则同样可以在日常的工作生活中“调用”：

做事的原则：自己的事自己做，别人的事，最多通过自己的产出让他们自己去搞.....

再仔细观察一下以下代码。当一个变量被当做参数传递给一个函数的时候，这个变量本身并不会被函数所改变。比如，`a = 5`，而后，再把 `a` 当作参数传递给 `f(a)` 的时候，这个函数当然应该返回它内部任务完成之后应该传递回来的值，但 `a` 本身不会被改变。

```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n-1)

a = 5
b = factorial(a)    # a 并不会因此改变；
print(a, b)
a = factorial(a)    # 这是你主动为 a 再一次赋值.....
print(a, b)
```

```
5 120
120 120
```

理解了这一点之后，再看 `factorial()` 这个递归函数的递归执行过程，你就能明白这个事实：

在每一次 `factorial(n)` 被调用的时候，它都会形成一个作用域，`n` 这个变量作为参数把它的值传递给了函数，**但是**，`n` 这个变量本身并不会被改变。

我们再修改一下上面的代码：

```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n-1)

n = 5          # 这一次，这个变量名称是 n
m = factorial(n) # n 并不会因此改变；
print(n, m)
```

5 120

在 `m = factorial(n)` 这一句中，`n` 被 `factorial()` 当做参数调用了，但无论函数内部如何操作，并不会改变变量 `n` 的值。

关键的地方在这里：在函数内部出现的变量 `n`，和函数外部的变量 `n` 不是一回事——它们只是名称恰好相同而已，函数参数定义的时候，用别的名称也没什么区别：

```
def factorial(x): # 在这个语句块中出现的变量，都是局部变量
    if x == 1:
        return 1
    else:
        return x * factorial(x-1)

n = 5          # 这一次，这个变量名称是 n
m = factorial(n) # n 并不会因此改变；
print(n, m)
# 这个例子和之前再之前的示例代码有什么区别吗？
# 本质上没区别，就是变量名称换了而已.....
```

5 120

函数开始执行的时候，`x` 的值，是由外部代码（即，函数被调用的那一句）传递进来的。即便函数内部的变量名称与外部的变量名称相同，它们也不是同一个变量。

递归函数三原则

现在可以小小总结一下了。

一个递归函数，之所以是一个有用、有效的递归函数，是因为它要遵守递归三原则。正如，一个机器人之所以是个合格的机器人，是因为它遵循[阿西莫夫三铁律](#)（Three Laws of Robotics）一样^[2]。

1. 根据定义，递归函数必须在内部调用自己；
2. 必须设定一个退出条件；

3. 递归过程中必须能够逐步达到退出条件.....

从这个三原则望过去，`factorial()` 是个合格有效的递归函数，满足第一条，满足第二条，尤其还满足第三条中的“逐步达到”！

而那个扯淡的盗梦空间递归程序，说实话，不太合格，虽然它满足第一条，也满足第二条，第三条差点蒙混过关：它不是逐步达到，而是不管怎样肯定能达到——这明显是两回事..... 原谅它罢，它的作用就是当例子，一次正面的，一次负面的，作为例子算是功成圆满了！

刚开始的时候，初学者好不容易搞明白递归函数究竟是怎么回事之后，就不由自主地想“我如何才能学会递归式思考呢？”——其实吧，这种想法本身可能并不是太正确或者准确。

准确地讲，递归是一种解决问题的方式。当我们需要解决的问题，可以被逐步拆分成很多越来越小的模块，然后每个小模块还都能用同一种算法处理的时候，用递归函数最简洁有效。所以，只不过是在遇到可以用递归函数解决问题的时候，才需要去写递归函数。

从这个意义上来看，递归函数是程序员为了自己方便而使用的，并不是为了计算机方便而使用——计算机么，你给它的任务多一点或者少一点，对它来讲无所谓，反正有电就能运转，它自己又不付电费.....

理论上讲，所有用递归函数能完成的任务，不用递归函数也能完成，只不过代码多一点，啰嗦一点，看起来没有那么优美而已。

还有，递归，不像“序列类型”那样，是某个编程语言的特有属性。它其实是一种特殊算法，也是一种编程技巧，任何编程语言，都可以使用递归算法，都可以通过编写递归函数巧妙地解决问题。

但是，学习递归函数本身就很烧脑啊！这才是最大的好事。从迷惑，到不太迷惑，到清楚，到很清楚，再到特别清楚——这是个非常有趣，非常有成就感的过程。

这种过程锻炼的是脑力——在此之后，再遇到大多数人难以理解的东西，你就可以使用这一次积累的经验，应用你已经磨炼过的脑力。有意思。

至此，封面上的那个“伪代码”应该很好理解了：

```
def teach_yourself(anything):  
    while not create():  
        learn()  
        practice()  
    return teach_yourself(another)  
  
teach_yourself(coding)
```

自学还真的就是递归函数呢.....

思考与练习

普林斯顿大学的一个网页，有很多递归的例子

<https://introcs.cs.princeton.edu/java/23recursion/>

脚注

[1]: 参见 Stackoverflow 上的讨论: [Boolean identity == True vs is True](#)

[↑ Back to Content ↑](#)

[2]: 关于[阿西莫夫三铁律](#) (Three Laws of Robotics) 的类比, 来自著名的 Python 教程, [Think Python: How to Think Like a Computer Scientist](#)

[↑ Back to Content ↑](#)