# 递归函数

廖雪峰
资深软件开发工程师，业余马拉松选手。

在函数内部，可以调用其他函数。如果一个函数在内部调用自身本身，这个函数就是递归函数。

举个例子，我们来计算阶乘 `n! = 1 x 2 x 3 x ... x n` ，用函数 `fact(n)` 表示，可以看出：

$$fact(n) = n! = 1 \times 2 \times 3 \times \cdots \times (n-1) \times n = (n-1)! \times n = fact(n-1) \times n$$

所以， `fact(n)` 可以表示为 `n x fact(n-1)` ，只有n=1时需要特殊处理。

于是， `fact(n)` 用递归的方式写出来就是：

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

上面就是一个递归函数。可以试试：

```
>>> fact(1)
1
>>> fact(5)
120
>>> fact(100)
933262154439441526816992388562667004907159682643816214685929638952175999932299156089414639761565
```

如果我们计算 `fact(5)` ，可以根据函数定义看到计算过程如下：

```
⇒ fact(5)
⇒ 5 * fact(4)
⇒ 5 * (4 * fact(3))
⇒ 5 * (4 * (3 * fact(2)))
⇒ 5 * (4 * (3 * (2 * fact(1))))
⇒ 5 * (4 * (3 * (2 * 1)))
⇒ 5 * (4 * (3 * 2))
⇒ 5 * (4 * 6)
⇒ 5 * 24
⇒ 120
```

递归函数的优点是定义简单，逻辑清晰。理论上，所有的递归函数都可以写成循环的方式，但循环的逻辑不如递归清晰。

使用递归函数需要注意防止栈溢出。在计算机中，函数调用是通过栈（stack）这种数据结构实现的，每当进入一个函数调用，栈就会加一层栈帧，每当函数返回，栈就会减一层栈帧。由于栈的大小不是无限的，所以，递归调用的次数过多，会导致栈溢出。可以试试 `fact(1000)` ：

```
>>> fact(1000)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in fact
  ...
  File "<stdin>", line 4, in fact
RuntimeError: maximum recursion depth exceeded in comparison
```

解决递归调用栈溢出的方法是通过**尾递归**优化，事实上尾递归和循环的效果是一样的，所以，把循环看成是一种特殊的尾递归函数也是可以的。

尾递归是指，在函数返回的时候，调用自身本身，并且，return语句不能包含表达式。这样，编译器或者解释器就可以把尾递归做优化，使递归本身无论调用多少次，都只占用一个栈帧，不会出现栈溢出的情况。

上面的 `fact(n)` 函数由于 `return n * fact(n - 1)` 引入了乘法表达式，所以就不是尾递归了。要改成尾递归方式，需要多一点代码，主要是要把每一步的乘积传入到递归函数中：

```
def fact(n):
    return fact_iter(n, 1)

def fact_iter(num, product):
```

```
    if num == 1:
        return product
    return fact_iter(num - 1, num * product)
```

可以看到，`return fact_iter(num - 1, num * product)` 仅返回递归函数本身，`num - 1` 和 `num * product` 在函数调用前就会被计算，不影响函数调用。

`fact(5)` 对应的 `fact_iter(5, 1)` 的调用如下：

```
⇒ fact_iter(5, 1)
⇒ fact_iter(4, 5)
⇒ fact_iter(3, 20)
⇒ fact_iter(2, 60)
⇒ fact_iter(1, 120)
⇒ 120
```

尾递归调用时，如果做了优化，栈不会增长，因此，无论多少次调用也不会导致栈溢出。

遗憾的是，大多数编程语言没有针对尾递归做优化，Python解释器也没有做优化，所以，即使把上面的 `fact(n)` 函数改成尾递归方式，也会导致栈溢出。

## 练习

汉诺塔的移动可以用递归函数非常简单地实现。

请编写 `move(n, a, b, c)` 函数，它接收参数 `n` ，表示3个柱子A、B、C中第1个柱子A的盘子数量，然后打印出把所有盘子从A借助B移动到C的方法，例如：

```
def move(n, a, b, c):
    if n == 1:
        print(a, '-->', c)

# 期待输出:
# A --> C
# A --> B
# C --> B
# A --> C
# B --> A
# B --> C
# A --> C
move(3, 'A', 'B', 'C')
```

## 参考源码

recur.py

## 小结

使用递归函数的优点是逻辑简单清晰，缺点是过深的调用会导致栈溢出。

针对尾递归优化的语言可以通过尾递归防止栈溢出。尾递归事实上和循环是等价的，没有循环语句的编程语言只能通过尾递归实现循环。

Python标准的解释器没有针对尾递归做优化，任何递归函数都存在栈溢出的问题。

« 函数的参数　　高级特性 »

## Comments

Comments loaded. To post a comment, please Sign In

勇敢的啊童 @ 2025/12/11 21:37:58

假设n个盘子要从起始柱A，移动到目的柱C，B是中间柱。 那是不是中间一定会出现这个情况：

1. 第n个盘子（最大的）在A柱，n-1个盘子在B柱，C柱上0个盘子；（你必须把所有小的按顺序放到B上，才能把最大的拿出来放到C）

2. 接下来的步骤，是要把A柱上的第n个盘子（最大的盘子）放到C柱上；

3. 再后面的操作，是把B柱上的n-1个盘子借助A柱放到C柱上。

也就是下面这3步

- 1)把n-1个盘子，从A移动到B，此时：A为源，B为目，C为中间
- 2)把第n个盘子，从A移动到C，此时：A为源，C为目
- 3)把n-1个盘子，从B移动到C，此时：B为源，C为目，A为中间

假设我们用一个函数去定义这个操作，可以写成：hano（n,A,B,C） 我们知道这里几个值的意思分别是，盘子数量，起始柱，中间柱，目标柱，这是一个固定参数，按顺序，在哪一位，就对应哪个。 那么这3步就可以写

Read More ▼

勇敢的啊童 @ 2025/12/11 21:46:04

再给一个输出的示例，解释了为啥能挨个把步数打印出来，而且是准确的

```
hanoi(3,A,B,C)
├─ hanoi(2,A,C,B)        # 先执行完整这个调用
│  ├─ hanoi(1,A,B,C)     # 打印: 1 A→C
│  ├─ print(2)           # 打印: 2 A→B
│  └─ hanoi(1,C,A,B)     # 打印: 1 C→B
├─ print(3)              # 打印: 3 A→C
└─ hanoi(2,B,A,C)        # 再执行完整这个调用
   ├─ hanoi(1,B,C,A)     # 打印: 1 B→A
   ├─ print(2)           # 打印: 2 B→C
   └─ hanoi(1,A,B,C)     # 打印: 1 A→C
```

lv @ 2025/12/19 02:13:26

很详细

杰 @ 2025/12/16 03:49:34

终于搞清这个代码逻辑。

```
# !/usr/bin/python3
# -*-coding:UTF-8-*-

"""
汉诺塔的移动用递归函数实现。

请编写move(n, a, b, c)函数，它接收参数n，
表示3个柱子A、B、C中第1个柱子A的盘子数量，
然后打印出把所有盘子从A借助B移动到C的方法
移动规则
1、一次只能移动一个圆盘：每次只能移动柱子最顶部的圆盘
2、不能将较大的圆盘放在较小的圆盘上：任何时候，任何柱子上的圆盘都必须保持从底部到顶部逐渐变小的顺
3、可以使用中间柱子作为辅助：在移动过程中，可以使用柱子B作为中转
```

Read More ▼

杰 @ 2025/12/16 03:04:08

文中这个“fact_iter”写错了，是“fact_tail”（尾递归）。

```
def fact_iter(num, product =1):
    if num == 1:
        return product
    return fact_iter(num - 1, num * product)
```

下面才是用迭代的“fact_iter”。

```
def fact_iter(num):
     product = 1  # 对应尾递归的 product 参数
    while num > 1:
        product *= num
        num = num - 1  # 模拟参数变化
    return product
```

呢喃 @ 2025/10/12 23:19:22

呢喃 @ 2025/10/13 11:13:44 Delete！！首先理解递归思想：“递” 是 “递推拆解”，“归” 是 “回归合并”。核心：大问题要能拆解成小问题，拆解后的小问题与大问题是相似结构。 举个例子：你下班需要1000步到家，是个很复杂的过程，那就拆解，看怎么走完1000步，先“递推拆解”：到家需要1000步=999步+1步，999步=998步+1步，998步=997步+1步，一直拆到2步=1步+1步（这里涉及递归核心思想之一：最小单元，我不必再将1步拆解，它是拆解的终止值）；“回归合并”：1步+999步，2步+998步.....999步+1步，回到家。你可以通俗理解为你就会走1步，让你走1000步你不会，当我们把问题拆到1步+1步时，你不就会了吗。递归的精髓就在于拆完后问题和总的问题结构是相似的，那么就可以由拆解的最小单元不断重复直到完成大单元。 ！！ 回到盘子问题（前提：盘子从上到下依次变大，每次只能移动一个盘子，就算在移动过程中也是从上到下依次变大，3个柱子）：假设10个盘子堆在A柱上，拆解问题：怎么把10个盘子从A移动到C上，让你移动10个不会，那就拆为1个+9个，就是拿走A柱上的上面9个，放到B柱上（先别管怎么过来的），此时A柱剩下了全场最大的一个盘子，B柱从上到下依次排列9个，C柱空，这个时候就将问题完美拆为1个+9个，1个盘子你会了，接着将A柱一个盘子直接移动到C，完成第一步。此时状态：A柱空，B柱从上到下依次排列9个，C柱一个最大的盘子（这里需要想通：我们的目的是将所有盘子移动到C柱，那么一旦C柱上摆上了这个最大的盘子，在以后所有的移动步骤中，我们都不必再动到这个盘子，为了方便理解，我们默认为只要全场最大的盘子放到C柱上就自动消失，我

Read More ▼

Moon @ 2025/12/4 02:21:48

终于懂了！谢谢大神

Hypersomnia @ 2025/11/24 02:12:12

```
def move(n, a, b, c):
    if n == 1:
        print(a, '-->', c)
    else:
        move(n-1, a, c, b)
        print(a, '-->', c)
        move(n-1, b, a, c)
```

杨志强 @ 2025/11/18 11:49:36

```
def move(n, a, b, c):
    if n == 1:
        print(a, '-->', c)

    else:
           move(n - 1, a, c, b)
           move(1, a, b, c)
           move(n - 1, b, a, c)
```

Bard @ 2025/11/14 03:19:40

def hannoi(n,a,b,c,count): if n==1: print(str.format("{0:1}->{1:1}",a,c)) count[0]+=1 else: hannoi(n-1,a,c,b,count) print(str.format("{0:1}->{1:1}",a,c)) count[0]+=1 hannoi(n-1,b,a,c,count) move_count=[0] hannoi(6,"A","B","C",move_count) print({move_count[0]})

待绝笔墨痕干 @ 2025/11/3 03:32:42

```
def move(n,a,b,c):
    if n ==1:
        print(f'{a}-->{c}')
    else:
        move(n-1,a,c,b)
        print(f'{a}-->{c}')
        move(n-1,b,a,c)

move(4,'A','B','C')
```

Sfz @ 2025/10/24 04:54:41

def move(n, a, b, c): if n == 1: print(a, '-->', c) else: move(n - 1, a ,c ,b) print(a, '-->', c) move(n - 1 , b , a ,c)

灰色.渲染 @ 2025/10/22 11:41:45

```
def move(num ,a ,b ,c):
    if num == 1:
        print(f'{a} ---> {c}')
    else:
        move(num - 1, a ,c ,b)
        print(f'{a} ---> {c}')
        move(num - 1 , b , a ,c)

move(4 , 'a','b','c')
```

江叶舟 @ 2025/10/17 06:55:58

def move(n, x, y, z): if n == 1: print(x, '-->', z) else: move(n-1, x, z, y) print(x, '-->', z) move(n-1, y, x, z)

## 期待输出:

## A --> C

## A --> B

## C --> B

## A --> C

## B --> A

Read More ▼

骄阳 @ 2025/10/12 23:51:57

```
def move(n,a,b,c):
    if n == 1:
        print(a, '-->', c)
    else:
        move(n-1,a,c,b)
        move(1,a,b,c)
        move(n-1,b,a,c)
move(3,'A','B','C')
```

bot吃宵夜 @ 2025/10/11 04:46:38

def move(n, a, b, c): if n == 1: print(a, '-->', c) else : # 将n-1个盘子从a借助c移动到b move(n - 1, a, c, b) # 将最大的盘子从a移动到c print(a, '-->', c) # 将n-1个盘子从b借助a移动到c move(n - 1, b, a, c)

**期待输出:**

**A --> C**

**A --> B**

**C --> B**

**A --> C**

**B _ _ \ A**

Read More ▼

---

Nostalgia @ 2025/10/7 22:19:42

其实意思就是说，无论盘子的个数是多少个，底层的逻辑都是，需要把除了最底下的一个盘子，其余的盘子(n-1)需要按照大小顺序挪到一个盘子（也就是B盘），这样能让最底下的盘子直接挪到C盘。那么其他(n-1)盘子后面怎么挪到C盘呢？问题就变成了B盘的(n-1)盘子如何通过A盘挪到C盘，那是不是可以理解成有(n-1)的盘子在A盘，如何通过B盘把盘子挪到C盘，这样一看问题其实有规律的进行循环了，n个盘子的问题是不是就拆解成了n-1的盘子+最底下的一个盘子两个部分了。

---

學不懂Fourier @ 2025/10/4 04:47:40

def move(n, a, b, c): if n == 1: print(a, '-->', c) else: move(n-1,a,c,b) print(a, '-->', c) move(n-1,b,a,c) move(3,'A','B','C')

---

一2三4 @ 2025/9/18 04:15:27

不懂原理的可以去这个在线的汉诺塔游戏玩一下
https://gallery.selfboot.cn/zh/algorithms/hanoitower#google_vignette

```
def move(n, a, b, c):
    """
    汉诺塔问题的递归解决方案

    参数:
        n: 盘子的数量
        a: 起始柱子
        b: 辅助柱子
        c: 目标柱子
    """
    if n == 1:
        print(a, '-->', c)
    else:
```

Read More ▼

---

2333 @ 2025/9/26 09:30:41

膜拜大佬，想移动第n个盘子需要把n-1个盘子先移开再移回去我是想到了，但是一直没想明白怎么用递归实现，ABC的关系好绕，大佬好强

---

理直气壮 @ 2025/9/20 05:05:51

好难，完全没想还能把函数里的参数顺序给改一下

---

长路漫漫 @ 2025/9/19 04:59:12

def move(n, a, b, c): if n == 1: print(a, '-->', c) else: move(n-1, a, c, b) print(a, '-->', c) move(n-1, b, a, c)

move(3, 'A', 'B', 'C')

Salvatore @ 2025/9/19 03:00:55

def move(n, a, b, c): if n == 1: print(a, '-->', c) else: move(n-1, a, c, b) move(1, a, b, c) move(n-1, b, a, c)

move(3, 'A', 'B', 'C')

---

Salvatore @ 2025/9/19 03:08:02

我自己的理解是：首先A上面的三个盘子从小到大顺序排列，那么第一个盘子是直接放到C柱上，所以需要考虑的是第二个盘子，也就是n-1，先放把第一个放到C然后把第二个放到B，此时，A上面就只剩下第三个盘子，这是需要考虑B柱上的盘子，B柱上的盘子，由于不能大盘子在小盘子上，所以需将C柱上的第一个盘子放在B柱上，再将A柱的第三个盘子放在C柱上，B柱上盘子的恰好满足n-1A柱上的移动（B-A-C）,最后就都是满足单个盘子的移动，就很直接了。

---

Doing_Alright @ 2025/9/16 00:45:57

#a是起始地 b是借助地 c是目标地 def move(n,a,b,c): if n==1: print(f'{a}-->{c}') else: move(n-1,a,c,b) print(f'{a}-->{c}') move(n-1,b,a,c)

move(10,'A','B','c')

可以去哔哩哔哩看这个up讲解，，，看作一个数理题来解答就很好理解了 https://www.bilibili.com/video/BV12741157Zp/?spm_id_from=333.337.search-card.all.click&vd_source=5f33ec9481cfe9039fd76b6c45545056 函数中的a,b,c这三参数是位置参数，，，a是起始位置，b是借助位置，c是目的地址。一定记住。 f(1,a,b,c) 是已知的，a-->c 然后用f(1,a,b,c)去求未知函数f(3,a,b,c) f(2,a,b,c)也是已知条件，，，然后去推出其中的变化规律f(n,a,b,c) f(2,a,b,c)=a->b , a->c ,b->c f(3,a,b,c)=f(2,a,c,b)+ a->c + f(2,b,a,c) f(4,a,b,c) =f(3,a,c,b) + a->c + f(3,b,a,c)

---