Python 程序设计基础

递归

曾昊 博士

浙江传媒学院媒体工程学院

hao.zeng@cuz.edu.cn

递归简介（一）

- 递归函数：一种可以自我调用的函数，例如：

```
1  def main():
2    message()
3
4  def message():
5    print("This is a recursive function.")
6    message()
7
8  main()
```

运行该脚本，结果如下，

```
1  This is a recursive function.
2  This is a recursive function.
3  This is a recursive function.
4  ...
```

递归简介（二）

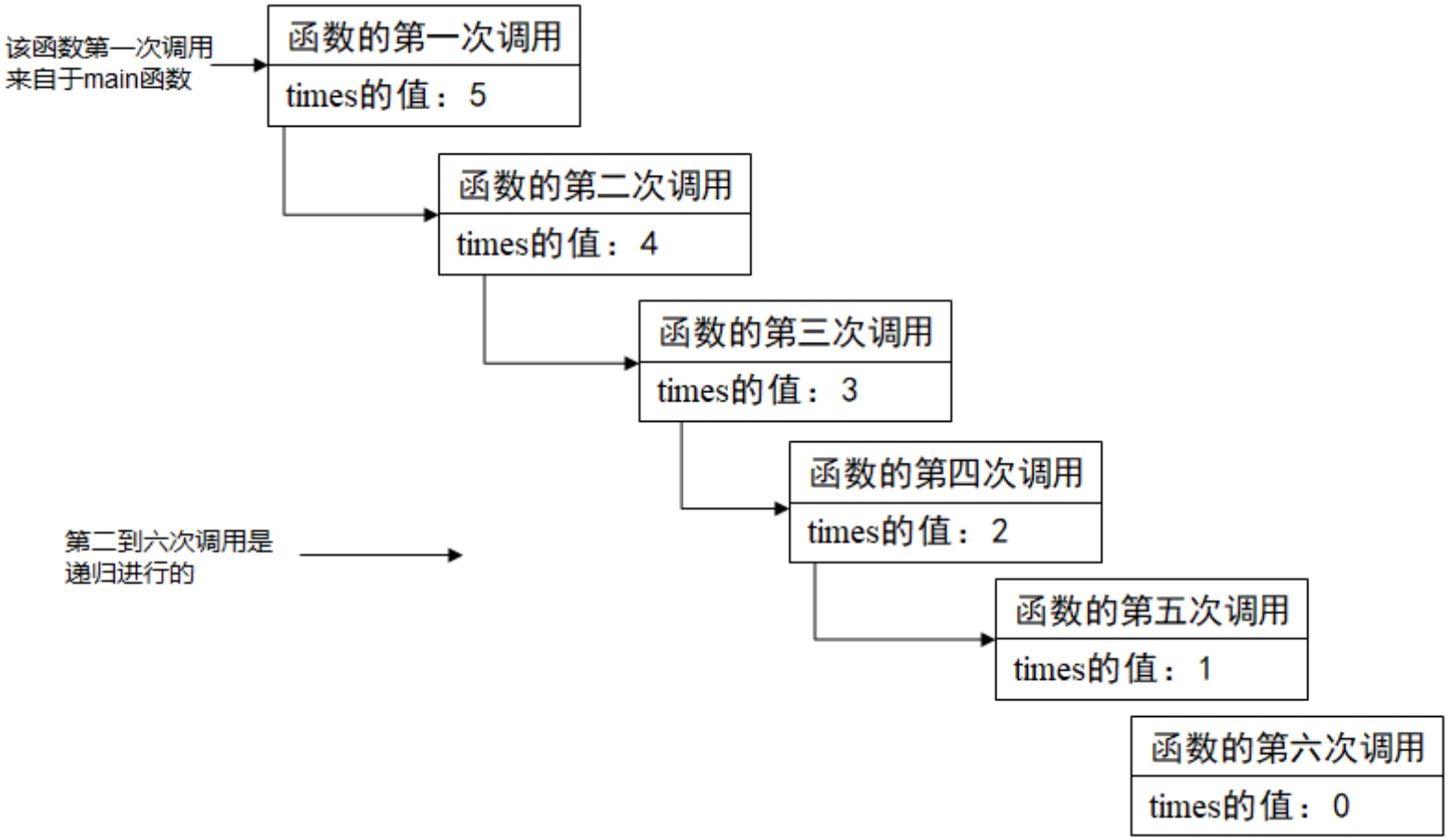
- 如同循环一样，递归函数必须使用特定的方法来控制它重复的次数
  - 通常包含一个 if-else 语句，该语句定义函数何时应该返回值，何时应该调用自身；

```
1  def main():
2    message(5)
3
4  def message(items):
5    if items > 0:
6      print("This is a recursive function.")
7      message(items-1)
8
9  main()
```

- 递归深度：函数调用自身的次数；

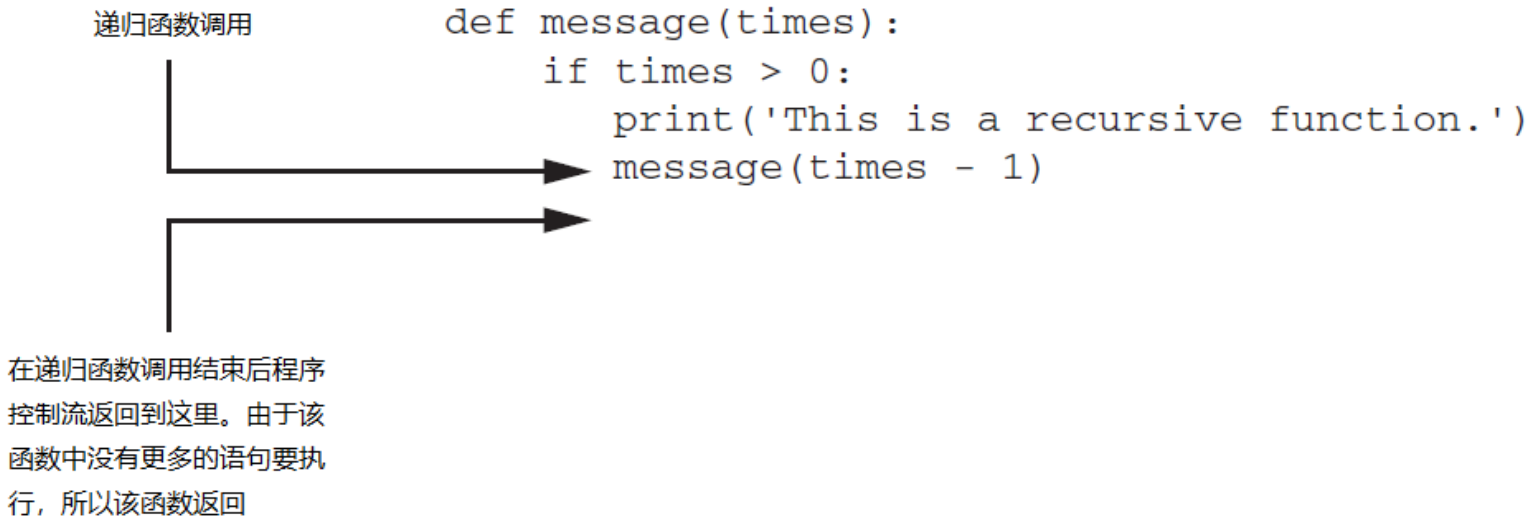
递归简介（三）

- message 函数的 6 次调用：



递归简介（三）

- 程序的控制流将从函数的第 6 个实例直接返回到第 5 个实例的递归调用发生之后的位置；



递归求解问题（一）

- 当一个问题可以被分解为与整体问题结构上一样的更小问题时，则可以使用递归来解决（类似于高中数学的数学归纳法）；
- 递归是解决重复问题的强大工具，但递归并不是解决问题的唯一方法；
  - 任何可以用递归解决的问题都可以通过循环来解决；
  - 递归算法通常比迭代算法效率低；因为调用函数的过程需要计算机执行若干操作（开销），如为参数和局部变量分配内存，以及存储在函数结束后控制流返回的程序位置的地址；

递归求解问题（二）

- 一些重复的问题使用递归比使用循环更容易解决；
  - 递归结构写起来简单，但机器执行起来比较慢；
  - 循环结构写起来困难，但机器执行起来比较快；
- 递归函数的工作原理：
  - 如果问题当前可以直接求解，无需递归，那么该函数直接解决问题并返回结果；
  - 如果问题当前无法直接求解，那么该函数将其简化到较小但类似的问题，并调用自身来解决这个较小的问题。(比如一个问题需要 n 步完成，我们就先解决在第 n-1 步是如何完成的。)

    为了使用这种方法，首先确定至少一种可以直接求解而不递归的情况，称之为基本条件（情况），即解决问题的第 1 步；其次确定一种在所有情况下使用递归来解决问题的方法，称之为递归步骤（情况），即如何从第 n-1 步到第 n 步；

使用递归计算阶乘（一）

- 数学中符号 n! 表示数字 n 的阶乘
  - 如果 n=0，那么 n!=1
  - 如果 n>0，那么 n!=1×2× 3× ...× n
- 若使用循环迭代实现阶乘：

```
1    def factorial(num):
2        if num == 0:
3            return 1
4        else:
5            product = 1
6            for i in num:
7                product = product * i
8            return product
```

使用递归计算阶乘（二）

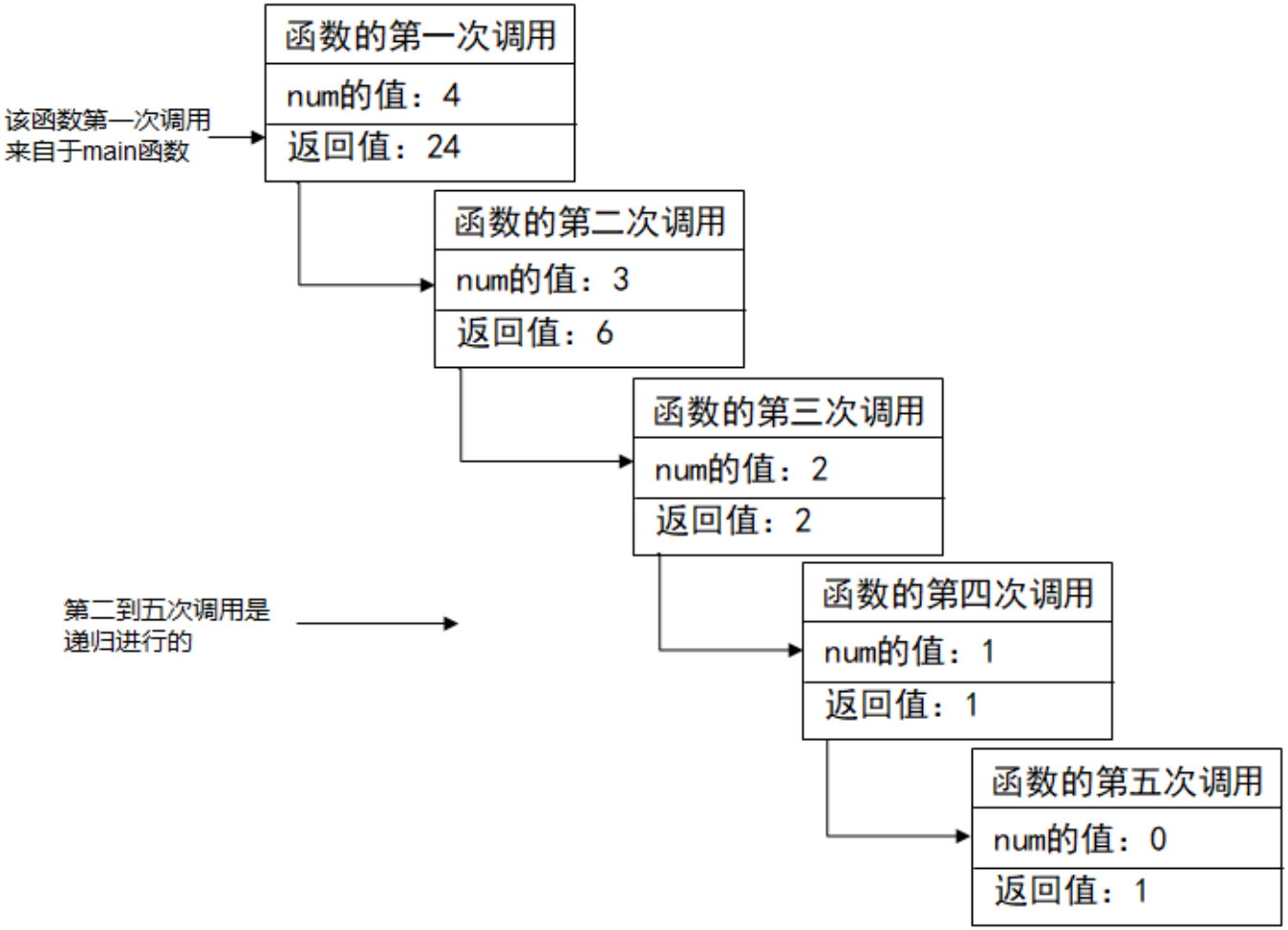
- 数学中符号 n! 表示数字 n 的阶乘
  - 如果 n=0，那么 n!=1
  - 如果 n>0，那么 n!=1×2× 3× ...× n
- 对于求解阶乘，也适合于递归编程：
  - n=0 是基本情况；
  - n>0 是递归情况；

$$\texttt{factorial(n)} = \begin{cases} 0 & \texttt{n=0} \\ \texttt{n} \times \texttt{factorial(n-1)} & \texttt{n>=0} \end{cases}$$

```
1  def factorial(num):
2    if num == 0:
3      return 1
4    else:
5      return num * factorial(num-1)
```

使用递归计算阶乘（三）

- factorial 函数的 5 次调用；



使用递归计算阶乘（四）

- 因为每次调用递归函数都能减少问题的规模；
  - 当最终它到达不需要递归的基本情况，递归就会停止；
- 通常，问题规模的缩减可以通过降低每次递归调用中的一个或多个参数的取值来实现。

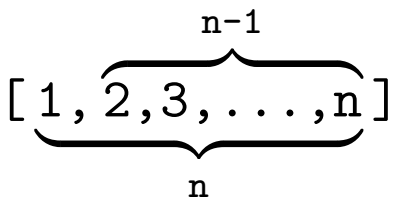
直接递归和间接递归

- 直接递归：直接调用自身的递归函数
  - 到目前为止所展示的所有例子都是直接递归
- 间接递归：当函数 A 调用函数 B，后者又调用 A；在递归中甚至可以涉及多个函数，比如函数 A 可以调用函数 B，函数 B 可以调用函数 C，函数 C 调用函数 A；
  - 交叉递归就是间接递归的一种情况：

```
1     def iseven(num):
2       if num == 0:
3           return True
4       else:
5           return isodd(num)
6
7     def isodd(num):
8       if num == 0:
9           return False
10      else:
11          return iseven(num-1)
```

递归算法示例

- 递归求解列表中元素的和；
  - 函数接收一个列表，该列表包含待求和元素的范围、范围内起始项的索引和范围内结束项的索引
  - 基础条件：当起始索引值大于结束索引值时；
  - 递归步骤：列表元素和等于头元素加上尾列表元素之和；




```
1    def range_sum(mylist,start,end):
2        if start > end:
3            return 0
4        else:
5            return mylist[start] + range_sum(mylist,
                 start+1, end)
```

斐波那契数列

- 斐波那契数列：有两种基本情况
  - 基础条件：如果 n=0，则 fib(n) = 0
  - 基础条件：如果 n=1，则 fib(n) = 1
  - 递归步骤：如果 n>1，则 fib(n) = fib(n-1) + fib(n-2)

0,1,1,2,3,5,8,13,21,34,55,89,144,244...

```
1  def fib(num):
2      if num == 0:
3          return 0
4      elif num == 1:
5          return 1
6      else:
7          return fib(num-1) + fib(num-2)
```

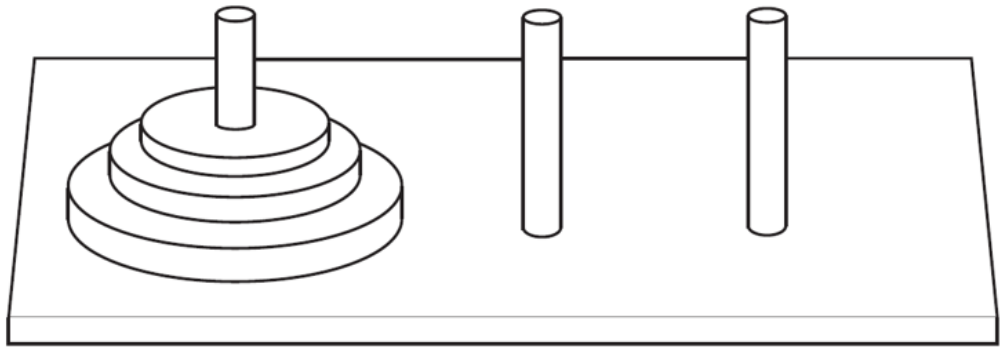
求解最大公约数

- 计算两个数字的最大公约数（通过辗转相除法）：
  - 基础条件：如果 x 可以被 y 整除，则 x 为 x 和 y 的最大公约数，即 gcd(x,y)=y
  - 递归步骤：x 和 y 的最大公约数等于 x%y 和 y 的最大公约数，即 gcd(x,y)=gcd(y,x%y)

```
1     def gcd(x,y):
2         if x > y:
3             if x % y == 0:
4                 return y
5             else:
6                 return gcd(x, x % y)
7         else:
8             if y % x == 0:
9                 return x
10            else:
11                return gcd(y, y % x)
```

汉诺塔（一）

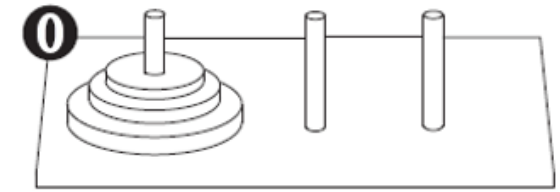
- 汉诺塔是一个数学游戏，常用来说明递归的强大。游戏中使用三个钉子和一组中心有孔的圆盘，圆盘堆叠在其中任意一个钉子上，如下图所示：



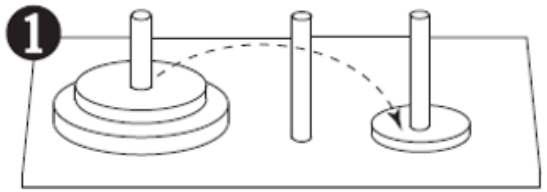
- 游戏的目标是移动圆盘从最左侧的钉子到最右侧的钉子上，必须遵守以下规则：
  - 一次只能移动一个圆盘
  - 大圆盘不能在较小的圆盘上面
  - 除了正在移动的圆盘，所有圆盘必须放在钉子上

汉诺塔（二）

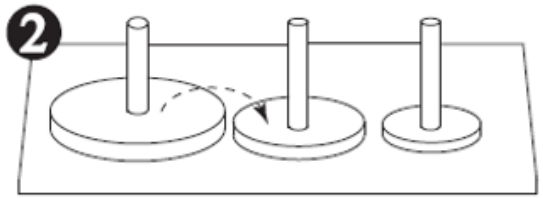
- 移动三个圆盘的步骤：



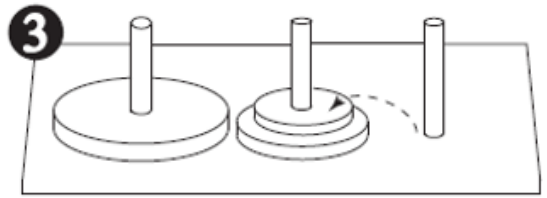
Original setup.



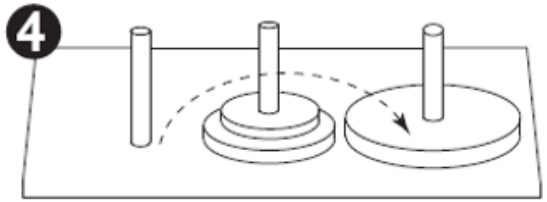
First move: Move disc 1 to peg 3.



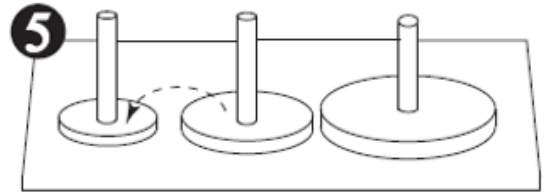
Second move: Move disc 2 to peg 2.



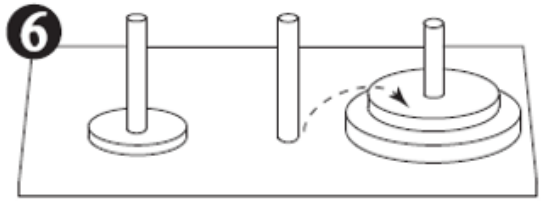
Third move: Move disc 1 to peg 2.



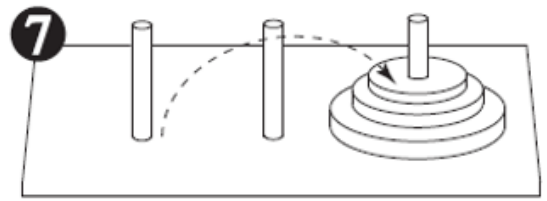
Fourth move: Move disc 3 to peg 3.



Fifth move: Move disc 1 to peg 1.



Sixth move: Move disc 2 to peg 3.



Seventh move: Move disc 1 to peg 3.

汉诺塔（三）

- 问题描述：将第二颗钉子作为中间过渡，将 n 个圆盘从第一颗钉子移动到第三颗钉子上
- 使用递归解决方法：
  - 基本条件：如果 n=1：移动圆盘从第一颗钉子到第三颗钉子；
  - 归纳步骤：
    1. 借助第三颗钉子将第 n-1 个圆盘从第一颗钉子移动到第二颗钉子；
    2. 然后将第 n 个圆盘从第一颗钉子移动到第三颗钉子；
    3. 借助第一颗钉子将第 n-1 个圆盘从第二颗钉子移动到第三颗钉子；

汉诺塔（四）

- 解决汉诺塔的递归函数如下：

```
1  # num:要移动的圆盘数量
2  # from_peg: 存放待移动圆盘的初始钉子
3  # to_peg: 存放代移动圆盘的最终钉子
4  # temp_peg: 可作为临时过渡的钉子
5  def move_dics(num, from_peg, to_peg, temp_peg):
6      if num>0:
7          move_dics(num-1, from_peg, temp_peg, to_peg)
8          print("Move a disc from", from_peg, "to peg",
               to_peg)
9          move_dics(num-1, temp_peg, to_peg, from_peg)
```

- 如果 num 大于 0，则圆盘需要移动，借助 to_peg 将 n-1 个圆盘从 from_peg 移动到 temp_peg；
- 然后显示一条消息，表面将第 n 个圆盘从 from_peg 移动到 to_peg；
- 接着，借助 from_peg 作为临时过渡将 n-1 个圆盘从 temp_peg 移动到 to_peg；

递归与循环

- 循环与递归：
  - 循环：从问题的第一步到问题的最后一步；
  - 递归：从问题的最后一步到问题的第一步；
- 不使用递归的原因：
  - 效率较低：循环中不需要函数调用开销；
  - 通常，使用循环比使用递归更清晰；
- 然而有些问题用递归比用循环更容易解决；
  - 例如最大公约数，其中的数学定义适合递归；

递归函数的运行（一）

- 实现阶乘的递归函数；

```
1    def factorial(num):
2      if num == 0:
3        return 1
4      else:
5        return num * factorial(num-1)
```

- 实现阶乘的递归函数的实际运算过程；

$$
\begin{aligned}
\texttt{factorial(4)} &= 4 \times \texttt{factorial(3)} \\
&= 4 \times (3 \times \texttt{factorial(2)}) \\
&= 4 \times (3 \times (2 \times \texttt{factorial(1)})) \\
&= 4 \times (3 \times (2 \times (1 \times \texttt{factorial(0)}))) \\
&= 4 \times (3 \times (2 \times (1 \times 1))) \\
&= 24
\end{aligned}
$$

递归函数的运行（二）

- 在传统的递归中，典型的模式是，你执行第一个递归调用，然后接着调用下一个递归来计算结果。这种方式中途你是得不到计算结果，知道所有的递归调用都返回。这样虽然很大程度上简洁了代码编写，但是让人很难它跟高效联系起来。因为随着递归的深入，之前的一些变量需要分配堆栈来保存。
- 当递归深度足够大时，则会造成内存溢出；

$$
\begin{aligned}
\texttt{factorial(10000)} &= 10000 \times \texttt{factorial(9999)} \\
&= 10000 \times (9999 \times \texttt{factorial(9998)}) \\
&= \ldots\ldots \\
&= 10000 \times (9999 \times (\ldots \times (1 \times 1))) \\
&= \text{很有可能内存溢出}
\end{aligned}
$$

尾递归（一）

- 尾递归相对传统递归，其是一种特例。在尾递归中，先执行某部分的计算，然后开始调用递归，所以你可以得到当前的计算结果，而这个结果也将作为参数传入下一次递归。这也就是说函数调用出现在调用者函数的尾部，因为是尾部，所以其有一个优越于传统递归之处在于无需去保存任何局部变量，从内存消耗上，实现节约特性。
- 尾递归作为一种代码优化，在效率上相当于循环迭代，且不失递归的简洁优雅；

尾递归（二）

- 尾递归实现阶乘函数；

```
1  def tail_fact(n, total = 1):
2    if n == 0:
3      return total
4    else:
5      return tail_fact(n-1, n*total)
```

- 实现阶乘的递归函数的实际运算过程；

$$\begin{aligned}\texttt{tail\_fact(4,1)} &= \texttt{tail\_fact(3, 4}\times\texttt{1)}\\ &= \texttt{tail\_fact(2, 3}\times\texttt{4)}\\ &= \texttt{tail\_fact(1, 2}\times\texttt{12)}\\ &= \texttt{tail\_fact(0, 1}\times\texttt{24)}\\ &= 24\end{aligned}$$

- 当前时刻的计算值作为第二个参数传入下一个递归，使得系统不再需要保留之前计算结果。这就是尾递归的优势；

尾递归（三）

- 但是 python 本身不支持尾递归（没有对尾递归做优化），而且对递归的次数有限制，当递归深度超过 1000 时，会抛出异常；
- 解决办法：重新设置默认递归深度；

```
1  import sys
2
3  sys.setrecursionlimit(10000)
```

总结

- 本章主要包含的内容有：
  - 递归的定义
  - 基本情况的重要性
  - 递归情况用来缩减问题的规模
  - 直接递归和间接递归
  - 递归算法的例子
  - 递归和循环

写在结尾

“读书有三到，谓心到，眼到，口到。”

——朱熹