1.7 递归函数

::: details INFO
译者：Mancuoj

来源：1.7 Recursive Functions

对应：Disc 03、HW 03
:::

如果函数体中直接或间接调用了函数本身，则函数称为递归（recursive）函数。也就是说，执行递归函数主体的过程中可能需要再次调用该函数。在 Python 中，递归函数不需要使用任何特殊语法，但它们确实需要一些努力来理解和创建。

我们将从编写一个对自然数的所有数字位求和的函数的样例开始。在设计递归函数时，我们需要找到可以将问题分解为更简单问题的方法。在这个示例中，可以使用运算符 `%` 和 `//` 将数字分为两部分：最后一位和除最后一位以外的所有数字。

```
>>> 18117 % 10
7
>>> 18117 // 10
1811
```

18117 的数字位之和是 $1+8+1+1+7 = 18$ 。正如同我们可以拆分数字一样，我们可以将这个和分成最后一位数字 7 和除最后一位数字之外的所有数字的和 $1+8+1+1 = 11$ 。这种拆分为我们提供了一种算法：要对数字 n 的数字位求和，就将其最后一位数字 `n % 10` 与 `n // 10` 的所有数字位之和相加。其中有一种特殊情况：如果数字只有一位，那么它的数字位之和就是它本身。该算法可以使用递归函数来实现。

```
>>> def sum_digits(n):
        """返回正整数 n 的所有数字位之和"""
        if n < 10:
            return n
        else:
            all_but_last, last = n // 10, n % 10
            return sum_digits(all_but_last) + last
```

这个 `sum_digits` 函数的定义是完整且准确的。虽然 `sum_digits` 函数在自己的函数体内被调用，但数字求和问题已被细分为两个步骤：先求出除最后一个数字外的所有数字总和，再加上最后一个数字的值。这两个步骤比原问题都更简单。由于第一个步骤与原问题相同，所以该函数被称为递归函数。也就是说，`sum_digits` 函数本身就是我们实现 `sum_digits` 所需要的函数。

```
>>> sum_digits(9)
9
>>> sum_digits(18117)
18
>>> sum_digits(9437184)
36
>>> sum_digits(1140885540205406461347032884838 4)
126
```

我们可以使用我们的计算环境模型来精确理解这个递归函数如何成功应用的，而且不需要添加新的规则。

当执行 `def` 语句时，名称 `sum_digits` 被绑定到一个新函数，但该函数的主体尚未执行。因此，`sum_digits` 的循环特性（circular nature）暂时还不是一个问题。然后，`sum_digits` 被传入参数 738：

1. 创建一个局部帧，将 `n` 绑定到 738，并在该帧作为起点的环境中执行 `sum_digits` 的函数体。
2. 由于 738 不小于 10，会执行第 4 行的赋值语句，将 738 分为 73 和 8。
3. 在下面的返回语句中，会以当前环境中 `all_but_last` 的值 73 调用 `sum_digits`。
4. 创建另一个将 `n` 绑定到 73 的局部帧，并在该帧作为起点的环境中再次执行 `sum_digits` 的函数体。
5. 由于 73 也不小于 10，将 73 分为 7 和 3，并以 7 调用 `sum_digits`，即 `all_but_last` 在此帧中的值。
6. 创建第三个局部帧，其中将 `n` 绑定到 7。
7. 在从这个帧开始的环境中，表达式 `n < 10` 为真，因此返回 7。
8. 在第二个局部帧中，将这个返回值 7 与 `last` 的值 3 相加，返回 10。
9. 在第一个局部帧中，将这个返回值 10 与 `last` 的值 8 相加，返回 18。

尽管这个递归函数具有循环特性，但它使用了不同的参数正确地应用了两次。此外，第二次应用此程序的对象是一个比第一次更简单的数字求和问题。生成调用 `sum_digits(18117)` 的环境图，可以看到每次连续的 `sum_digits` 的调用都使用了比上次更小的参数，直到最后得到个位数的输入。

这个例子还说明了具有简单函数体的函数可以通过使用递归演变成具有复杂计算过程的函数。

1.7.1 递归函数剖析

许多递归函数的函数体中存在着一种常见的模式。函数体会以一个基线条件（base case）开始（这是一种条件语句），它为最容易处理的输入定义了函数的行为。对于 `sum_digits` 函数而言，基线条件是接收到任意一位数的参数，我们只需返回该参数。有些递归函数会有多个基线条件。

然后，在基线条件之后，会有一个或多个递归调用。递归调用总是有一个特点：它们简化了原始问题。递归函数通过逐步简化问题来表达计算。例如，对 7 的数字求和比对 73 的数字求和更简单，而对 73 求和又比对 738 更简单。对于每个后续调用，剩余的计算量都会减少。

递归函数解决问题的方法通常不同于我们之前使用的迭代方法。思考一个计算 n 的阶乘的函数 `fact`，其中 `fact(4)` 会计算为 $4! = 4 \cdot 3 \cdot 2 \cdot 1=24$ 。

一种自然的实现方式是使用 `while` 语句将每个小于等于 n 的正整数都相乘得到结果。

```
>>> def fact_iter(n):
        total, k = 1, 1
        while k <= n:
            total, k = total * k, k + 1
        return total

>>> fact_iter(4)
24
```

另一方面，阶乘的递归实现可以用 `fact(n-1)` 来表达 `fact(n)`，就是一个更简单的问题了。这个递归的基线条件是问题最简单的形式：`fact(1)` 是 1。

这两个阶乘函数在概念上有所不同。迭代函数通过在每一项中连续相乘，来构造从基线条件 1 到最终总数 n 的结果。另一方面，递归函数直接从最终项 n 和更简单的问题 `fact(n-1)` 的结果来构造出最终结果。

递归会通过 `fact` 函数的连续应用，逐步"展开 unwinds"为越来越简单的问题实例，最后从基线条件开始构造出结果。它通过将参数 1 传递给 `fact` 来结束递归；每次调用的结果取决于下一次调用，直到达到基线条件。

从阶乘的标准数学函数定义中，很容易验证这个递归函数的正确性：

$$
\begin{aligned}
(n-1)! &= (n-1)\cdot(n-2)\cdot\cdots\cdot 1 \\
n! &= n\cdot(n-1)\cdot(n-2)\cdot\cdots\cdot 1 \\
n! &= n\cdot(n-1)!
\end{aligned}
$$

虽然我们可以使用计算模型来展开递归，但将递归调用（recursive calls）视为函数抽象会更容易理解一点。也就是说，我们不用在意 `fact(n-1)` 在 `fact` 的函数体中是怎么实现的；我们只需要相信它能计算 n-1 的阶乘就好了。将递归调用看作一种函数抽象这一思想，就是所谓"递归的信仰之跃（recursive leap of faith）"。我们根据函数自身来定义一个函数，但在验证函数的正确性时，我们只需相信在更简单的情况下，函数同样能正确工作。在这个示例中，我们假设 `fact(n-1)` 能够正确计算 $(n-1)!$；如果假设成立，我们只需要检查 $n!$ 是否被正确计算即可。这样，验证递归函数的正确性实际上变成了一种归纳法（induction）的证明形式。

函数 `fact_iter` 和 `fact` 也有所不同，因为前者必须引入两个额外的名称 `total` 和 `k`，这在递归实现中是不需要的。一般来说，迭代函数必须在计算过程中维护一些会变化的局部状态。在迭代中的任何时刻，该状态都可以表示已完成计算的结果和剩余的待计算的量。例如，当 `k = 3`，`total = 2` 时，仍然有两项需要处理，即 3 和 4。另一方面，`fact` 的特征是它的单一参数 `n`。计算的状态完全嵌入在环境的结构中，它的返回值扮演 total 的角色，并将 `n` 绑定到不同帧中的不同值，而不是显式地跟踪 `k`。

递归函数利用调用表达式求值的规则将名称绑定到值，通常避免了在迭代期间正确分配局部名称的麻烦。由于这个原因，我们可能更容易正确地定义递归函数。但是，学着识别由递归函数演化而来的计算过程需要一定的实践练习。

1.7.2 互递归

当一个递归过程被划分到两个相互调用的函数中时，这两个函数被称为是互递归的（mutually recursive）。例如，思考以下非负整数的偶数和奇数定义：

- 如果一个数比一个奇数大 1，那它就是偶数
- 如果一个数比一个偶数大 1，那它就是奇数
- 0 是偶数

使用这个定义，我们可以实现一个互递归函数来确定一个数字是偶数还是奇数：

通过打破两个函数之间的抽象边界，可以将互递归函数转换为单个递归函数。在这个例子中，可以将 `is_odd` 的函数体合并到 `is_even` 的函数体中，确保将 `is_odd` 函数体中的 n 替换为 n-1 以反映传递给它的参数：

```
>>> def is_even(n):
        if n == 0:
            return True
        else:
            if (n-1) == 0:
                return False
            else:
                return is_even((n-1)-1)
```

因此，互递归并不比简单递归更神秘或更强大，它只是提供了一种在复杂递归程序中维护抽象的机制。

1.7.3 递归函数中的打印

通过对 `print` 函数的调用，递归函数的计算过程通常可以可视化。作为示例，我们将实现一个 `cascade` 函数，该函数按从大到小再到大的顺序，打印一个数字的所有前缀。

```
>>> def cascade(n):
        """打印数字 n 的前缀的级联"""
        if n < 10:
            print(n)
        else:
            print(n)
            cascade(n//10)
            print(n)

>>> cascade(2013)
2013
201
20
2
20
201
2013
```

在这个递归函数中，基线条件是打印出来个位数。否则，就在两个 `print` 调用之间使用递归调用。

在递归调用之前写出基线条件表达式并不是一个严格的要求。实际上，通过观察可以看到 `print(n)` 在条件语句的两个子句中重复出现，我们可以将其前置从而更简洁地表达这个函数。

```
>>> def cascade(n):
        """Print a cascade of prefixes of n."""
        print(n)
        if n >= 10:
            cascade(n//10)
            print(n)
```

作为另一个互递归的例子，请思考一个两人博弈的情景，桌子上最初有 n 个石子，玩家轮流从桌面上拿走一个或两个石子，拿走最后一个石子的玩家获胜。假设 Alice 和 Bob 在玩这个游戏，两个人都使用一个简单的策略：

- Alice 总是取走一个石子
- 如果桌子上有偶数个石子，Bob 就拿走两个石子，否则就拿走一个石子

给定 n 个初始石子且 Alice 先开始拿，谁会赢得游戏？

该问题的一个自然分解是将每个策略封装在其自己的函数中。这使我们可以修改一个策略而不会影响其他策略，保持两者之间的抽象界限（abstraction barrier）。为了融入游戏的回合制性质，这两个函数在每个回合结束时互相调用。

```
>>> def play_alice(n):
        if n == 0:
            print("Bob wins!")
        else:
            play_bob(n-1)

>>> def play_bob(n):
        if n == 0:
            print("Alice wins!")
        elif is_even(n):
            play_alice(n-2)
        else:
            play_alice(n-1)

>>> play_alice(20)
Bob wins!
```

在函数 `play_bob` 中，我们看到多个递归调用可能会出现一个函数体中。虽然在这个例子中，每次调用 `play_bob` 最多只会调用一次 `play_alice`。在下个小节中，我们将会思考当单个函数调用同时直接进行多个递归函数调用时会发生什么。

1.7.4 树递归

另一种常见的计算模式称为树递归（tree recursion），在这种模式中，函数会多次调用自己。例如计算斐波那契数列，其中的每个数都是前两个数的和。

相对于我们之前的尝试，这个递归定义非常吸引人：它完全反映了我们熟悉的斐波那契数的定义。具有多个递归调用的函数称为树递归，因为每个调用都会分成多个较小的调用，每个较小的调用又会分成更小的调用，就像树枝从树干伸出一样，变得更小但数量更多。

我们已经能够不用树递归定义一个函数来计算斐波那契数列。事实上，我们以前的方法更加高效，这是本文后面讨论的主题。接下来，我们会思考一个问题，而对于这个问题，树递归的解决方案比任何迭代方案都要简单得多。

1.7.5 示例：分割数

求正整数 n 的分割数，最大部分为 m，即 n 可以分割为不大于 m 的正整数的和，并且按递增顺序排列。例如，使用 4 作为最大数对 6 进行分割的方式有 9 种。

```
1.  6 = 2 + 4
2.  6 = 1 + 1 + 4
3.  6 = 3 + 3
4.  6 = 1 + 2 + 3
5.  6 = 1 + 1 + 1 + 3
6.  6 = 2 + 2 + 2
7.  6 = 1 + 1 + 2 + 2
8.  6 = 1 + 1 + 1 + 1 + 2
9.  6 = 1 + 1 + 1 + 1 + 1 + 1
```

我们将定义一个名为 `count_partitions(n, m)` 的函数，该函数返回使用 `m` 作为最大部分对 n 进行分割的方式的数量。这个函数有一个使用树递归的简单的解法，它基于以下的观察结果：

使用最大数为 m 的整数分割 n 的方式的数量等于

1. 使用最大数为 m 的整数分割 n-m 的方式的数量，加上
2. 使用最大数为 m-1 的整数分割 n 的方式的数量

要理解为什么上面的方法是正确的，我们可以将 n 的所有分割方式分为两组：至少包含一个 m 的和不包含 m 的。此外，第一组中的每次分割都是对 n-m 的分割，然后在最后加上 m。在上面的实例中，前两种拆分包含 4，而其余的不包含。

因此，我们可以递归地将使用最大数为 m 的整数分割 n 的问题转化为两个较简单的问题：① 使用最大数为 m 的整数分割更小的数字 n-m，以及 ② 使用最大数为 m-1 的整数分割 n。

为了实现它，我们需要指定以下的基线情况：

1. 整数 0 只有一种分割方式
2. 负整数 n 无法分割，即 0 种方式
3. 任何大于 0 的正整数 n 使用 0 或更小的部分进行分割的方式数量为 0

```
>>> def count_partitions(n, m):
        """计算使用最大数 m 的整数分割 n 的方式的数量"""
        if n == 0:
            return 1
        elif n < 0:
            return 0
        elif m == 0:
            return 0
        else:
            return count_partitions(n-m, m) + count_partitions(n, m-1)

>>> count_partitions(6, 4)
9
>>> count_partitions(5, 5)
7
>>> count_partitions(10, 10)
42
>>> count_partitions(15, 15)
176
>>> count_partitions(20, 20)
627
```

我们可以将树递归函数视为探索不同的可能性。在这种情况下，我们探讨了使用大小为 m 的部分以及不使用这部分的可能性。第一次和第二次递归调用即对应着这些可能性。

如果不使用递归，则需要投入更多的精力来实现这个函数。有兴趣的读者可以尝试一下。