1.6 高阶函数

::: details INFO
译者：Mancuoj

来源：1.6 Higher-Order Functions

对应：Disc 01、Disc 02、HW 02、Lab 02、Hog
:::

我们已经意识到，函数其实是一种抽象方法，它描述了与特定参数值无关的的复合操作。

```
>>> def square(x):
        return x * x
```

也就是说，在 `square` 中我们并不是在讨论一个特定数字的平方，而是在讨论一种可以获得任何数字平方的方法。当然，我们也可以在不定义这个函数的情况下，通过写下面的表达式来计算平方得到结果，从而不显式地提到 `square`。

```
>>> 3 * 3
9
>>> 5 * 5
25
```

这种做法对于诸如 `square` 之类的简单计算是足够的，但对于诸如 `abs` 或 `fib` 之类的更复杂的样例就会变得很麻烦。一般来说，缺少函数定义会对我们非常不利，它会迫使我们总是工作在特定的原始操作级别（本例中为乘法），而不是更高的操作级别。我们的程序将能够计算平方，但缺少表达平方的概念的能力。

我们对强大的编程语言提出的要求之一就是能够通过将名称分配给通用模板（general patterns）来构建抽象，然后直接使用该名称进行工作。函数提供了这种能力。正如我们将在下面的示例中看到的那样，代码中重复出现了一些常见的编程模板，但它们可以与许多不同的函数一起使用。这些模板也可以通过给它们命名来进行抽象。

为了将某些通用模板表达为具名概念（named concepts），我们需要构造一种“可以接收其他函数作为参数”或“可以把函数当作返回值”的函数。这种可以操作函数的函数就叫做高阶函数（higher-order functions）。本节将会展示：高阶函数可以作为一种强大的抽象机制，来极大地提高我们语言的表达能力。

1.6.1 作为参数的函数

思考以下三个计算求和的函数。第一个 `sum_naturals` 会计算从 1 到 `n` 的自然数之和：

```
>>> def sum_naturals(n):
        total, k = 0, 1
        while k <= n:
            total, k = total + k, k + 1
        return total

>>> sum_naturals(100)
5050
```

第二个 `sum_cubes` 函数会计算 1 到 `n` 的自然数的立方之和。

```
>>> def sum_cubes(n):
        total, k = 0, 1
        while k <= n:
            total, k = total + k*k*k, k + 1
        return total

>>> sum_cubes(100)
25502500
```

第三个 `pi_sum` 会计算下列各项的总和，它的值会非常缓慢地收敛（converge）到 π 。

$$\frac{8}{1 \cdot 3} + \frac{8}{5 \cdot 7} + \frac{8}{9 \cdot 11} + \cdots$$

```
>>> def pi_sum(n):
        total, k = 0, 1
        while k <= n:
            total, k = total + 8 / ((4*k-3) * (4*k-1)), k + 1
        return total

>>> pi_sum(100)
3.1365926848388144
```

这三个函数显然在背后共享着一个通用的模板（pattern）。它们在很大程度上是相同的，仅在名称和用于计算被加项 `k` 的函数上有所不同。我们可以通过在同一模板中填充槽位（slots）来生成每个函数：

```
def <name>(n):
    total, k = 0, 1
    while k <= n:
        total, k = total + <term>(k), k + 1
    return total
```

这种通用模板的存在是一个强有力的证据 --> 表明了有一个实用的抽象手段正在“浮出水面”。这些函数都是用于求出各项的总和。作为程序设计者，我们希望我们的语言足够强大，以便我们可以编写一个表达“求和”概念的函数，而不仅仅是一个计算特定和的函数。在 Python 中，我们可以很轻易地做到这一点，方法就是使用上面展示的通用模板，并将“槽位”转换为形式参数：

在下面的示例中，`summation` 函数将上界 `n` 和计算第 `k` 项的函数 `term` 作为其两个参数。我们可以像使用任何函数一样使用 `summation` ，它简洁地表达了求和。花点时间单步调试（step through）走完这个示例，注意函数如何将 `cube` 绑定到局部名称 `term` 上以确保 $1\times 1\times 1 + 2\times 2\times 2 + 3\times 3\times 3 = 36$ 计算结果正确。

使用会返回其参数的 `identity` 函数，我们还可以使用完全相同的 `summation` 函数对自然数求和。

```
>>> def summation(n, term):
        total, k = 0, 1
        while k <= n:
            total, k = total + term(k), k + 1
        return total
>>> def identity(x):
        return x
>>> def sum_naturals(n):
        return summation(n, identity)
>>> sum_naturals(10)
55
```

`summation` 函数也可以直接调用，而无需为特定的数列去定义另一个函数。

```
>>> summation(10, square)
385
```

可以定义 `pi_term` 函数来计算每一项的值，从而使用我们对 `summation` 的抽象来定义 `pi_sum` 函数。传入参数 `1e6`（1 * 10 ^{6} = 1000000 的简写）以计算 π 的近似值。

```
>>> def pi_term(x):
        return 8 / ((4*x-3) * (4*x-1))
>>> def pi_sum(n):
        return summation(n, pi_term)
>>> pi_sum(1e6)
3.141592153589902
```

1.6.2 作为通用方法的函数

之前我们引入了“用户定义函数（user-defined functions）”作为一种数值运算的抽象模式，从而使它们与涉及的特定数字无关。有了高阶函数后，我们会看到一种更强大的抽象：用一些函数来表达计算的通用方法（general methods），而且和它们调用的特定函数无关。

尽管我们对函数的意义进行了这些概念上的扩展，但我们用于查看“调用表达式如何求解”的环境模型可以优雅地、无需更改地扩展到高阶函数。当将用户定义的函数应用于某些参数时，形式参数将与局部帧中它们的值（参数可能是函数）绑定。

思考下面的例子，它实现了迭代改进（iterative improvement）的通用方法，并使用它来计算黄金比例。黄金比例 通常被称为“phi”，是一个接近 1.6 的数字，经常出现在自然、艺术和建筑中。

迭代改进算法从方程的 `guess` 解（推测值）开始，重复应用 `update` 函数来改进该猜测，并调用 `close` 比较来检查当前的 `guess` 是否已经“足够接近”正确值。

```
>>> def improve(update, close, guess=1):
        while not close(guess):
            guess = update(guess)
        return guess
```

这个 `improve` 函数是迭代求精（repetitive refinement）的通用表达式。它并不会指定要解决的问题，而是会将这些细节留给作为参数传入的 `update` 和 `close` 函数。

黄金比例的一个著名的特性是它可以通过反复叠加任何正数的倒数加上 1 来计算，而且它比它的平方小 1。我们可以将这些特性表示为与 `improve` 一起使用的函数。

```
>>> def golden_update(guess):
        return 1/guess + 1

>>> def square_close_to_successor(guess):
        return approx_eq(guess * guess, guess + 1)
```

以上，我们调用了 `approx_eq` 函数：如果其参数大致相等，则返回 `True`。为了实现 `approx_eq`，我们可以将两个数字差的绝对值与一个小的公差值（tolerance value）进行比较。

```
>>> def approx_eq(x, y, tolerance=1e-15):
        return abs(x - y) < tolerance
```

使用参数 `golden_update` 和 `square_close_to_successor` 来调用 `improve` 将会计算出黄金比例的有限近似值。

```
>>> improve(golden_update, square_close_to_successor)
1.6180339887498951
```

通过追踪求解的步骤，我们可以看到这个结果是如何计算出来的。首先，将 `update`、`close` 和 `guess` 绑定在构造 `improve` 的局部帧上。然后在 `improve` 的函数体中，将名称 `close` 绑定到 `square_close_to_successor`，它会使用 `guess` 的初始值进行调用。跟踪其余步骤来查看其逐步演变为黄金比例的计算过程。

这个例子说明了计算机科学中两个相关的重要思想：首先，命名和函数使我们能将大量的复杂事物进行抽象。虽然每个函数定义都很简单，但我们的求解程序触发的计算过程非常复杂。其次，正是由于我们对 Python 语言有一个极其通用的求解过程，小的组件才能组合成复杂的程序。理解这解释程序的求解过程会有便于我们验证和检查我们创建的程序。

一如既往，我们的新通用方法 `improve` 需要测试来检查其正确性。黄金比例可以提供这样的测试，因为它有一个精确的闭式解，所以我们可以将这个解与迭代结果进行比较。

```
>>> from math import sqrt
>>> phi = 1/2 + sqrt(5)/2
>>> def improve_test():
        approx_phi = improve(golden_update, square_close_to_successor)
        assert approx_eq(phi, approx_phi), 'phi differs from its approximation'
>>> improve_test()
```

对于这个测试，没有消息就是好消息：`improve_test` 函数在 `assert` 语句执行成功后会返回 `None`。

1.6.3 定义函数 III：嵌套定义

上面的示例演示了将函数作为参数传递的能力显著地增强编程语言的表达能力。每个通用概念或方程都能映射到自己的小型函数上，但这种方法的一个负面后果是全局帧会变得混乱，因为小型函数的名称必须都是唯一的。另一个问题是我们受到特定函数签名的限制：`improve` 的 `update` 参数只能接受一个参数。嵌套函数定义（Nested function definition）解决了这两个问题，但需要我们丰富一下环境模型。

让我们思考一个新问题：计算一个数的平方根。在编程语言中，“平方根”通常缩写为 `sqrt`。重复应用以下更新，值会收敛为 `a` 的平方根：

```
>>> def average(x, y):
        return (x + y)/2

>>> def sqrt_update(x, a):
        return average(x, a/x)
```

这个两个参数的更新函数与只有一个参数的 `improve` 函数并不兼容，还有一个问题是它只提供一次更新，但我们真正想要的是通过重复更新来求平方根。这两个问题的解决方案就是嵌套定义函数，将函数定义放在另一个函数定义内。

```
>>> def sqrt(a):
        def sqrt_update(x):
            return average(x, a/x)
        def sqrt_close(x):
            return approx_eq(x * x, a)
        return improve(sqrt_update, sqrt_close)
```

与局部赋值一样，局部的 `def` 语句只影响局部帧。被定义的函数仅在求解 `sqrt` 时在作用域内。而且这些局部 `def` 语句在调用 `sqrt` 之前都不会被求解，与求解过程一致。

词法作用域（Lexical scope）：局部定义的函数也可以访问整个定义作用域内的名称绑定。在此示例中，`sqrt_update` 引用名称 `a`，它是其封闭函数 `sqrt` 的形式参数。这种在嵌套定义之间共享名称的规则称为词法作用域。最重要的是，内部函数可以访问定义它们的环境中的名称（而不是它们被调用的位置）。

我们需要实现两个扩展使环境模型启用词法作用域：

1. 每个用户定义的函数都有一个父环境：定义它的环境。
2. 调用用户定义的函数时，其局部帧会继承其父环境。

在调用 `sqrt` 之前，所有函数都是在全局环境中定义的，因此它们都有相同的父级：全局环境。相比之下，当 Python 计算 `sqrt` 的前两个子句时，它会创建局部环境关联的函数。

```
>>> sqrt(256)
16.0
```

在这次调用中，环境首先为 `sqrt` 添加一个局部帧，然后求解 `sqrt_update` 和 `sqrt_close` 的 `def` 语句。

从现在开始我们将在环境图中的每个函数都增加一个新注释 --> 父级注释（a parent annotation）。函数值的父级是定义该函数的环境的第一帧。没有父级注释的函数是在全局环境中定义的。当调用用户定义的函数时，创建的帧与该函数具有相同的父级。

随后，名称 `sqrt_update` 解析为这个新定义的函数，该函数作为参数传给函数 `improve`。在 `improve` 函数的函数体中，我们必须以 `guess` 的初始值 `x` 为 1 来调用 `update` 函数（绑定到 `sqrt_update`）。这个最后的程序调用为 `sqrt_update` 创建了一个环境，它以一个仅包含 `x` 的局部帧开始，但父帧 `sqrt` 仍然包含 `a` 的绑定。

这个求值过程中最关键的部分是将 `sqrt_update` 的父环境变成了通过调用 `sqrt_update` 创建的（局部）帧。此帧还带有 `[parent=f1]` 的注释。

继承环境（Extended Environments）：一个环境可以由任意长的帧链构成，并且总是以全局帧结束。在举 `sqrt` 这个例子之前，环境最多只包含两种帧：局部帧和全局帧。通过使用嵌套的 `def` 语句来调用在其他函数中定义的函数，我们可以创建更长的（帧）链。调用 `sqrt_update` 的环境由三个帧组成：局部帧 `sqrt_update` 、定义 `sqrt_update` 的 `sqrt` 帧（标记为 `f1`）和全局帧。

`sqrt_update` 函数体中的返回表达式可以通过遵循这一帧链来解析 `a` 的值。查找名称会找到当前环境中绑定到该名称的第一个值。Python 首先在 `sqrt_update` 帧中进行检查 --> 不存在 `a` ，然后又到 `sqrt_update` 的父帧 `f1` 中进行检查，发现 `a` 被绑定到了 256。

因此，我们发现了 Python 中词法作用域的两个关键优势：

- 局部函数的名称不会影响定义它的函数的外部名称，因为局部函数的名称将绑定在定义它的当前局部环境中，而不是全局环境中。
- 局部函数可以访问外层函数的环境，这是因为局部函数的函数体的求值环境会继承定义它的求值环境。

这里的 `sqrt_update` 函数自带了一些数据：`a` 在定义它的环境中引用的值，因为它以这种方式“封装”信息，所以局部定义的函数通常被称为闭包（closures）。

1.6.4 作为返回值的函数

通过创建“返回值就是函数”的函数，我们可以在我们的程序中实现更强大的表达能力。带有词法作用域的编程语言的一个重要特性就是，局部定义函数在它们返回时仍旧持有所关联的环境。下面的例子展示了这一特性的作用。

一旦定义了许多简单的函数，函数组合（composition）就成为编程语言中的一种自然的组合方法。也就是说，给定两个函数 `f(x)` 和 `g(x)`，我们可能想要定义 `h(x) = f(g(x))`。我们可以使用我们现有的工具来定义函数组合：

```
>>> def compose1(f, g):
        def h(x):
            return f(g(x))
        return h
```

此示例的环境图展示了如何正确解析名称 `f` 和 `g` ，即使它们存在名称冲突。

`compose1` 中的 1 表示这个组合的函数只有一个参数。这个命名惯例不是解释器强制要求的，1 只是函数名的一部分而已。

在这一点上，我们已经能够察觉到到努力去精确定义计算环境模型的好处：不需要修改环境模型就可以解释如何以这种方式来返回函数。

1.6.5 示例：牛顿法

这个扩展的示例展示了函数返回值和局部定义如何协同工作，以简洁地表达一般的思想。我们将实现一种广泛用于机器学习、科学计算、硬件设计和优化的算法。

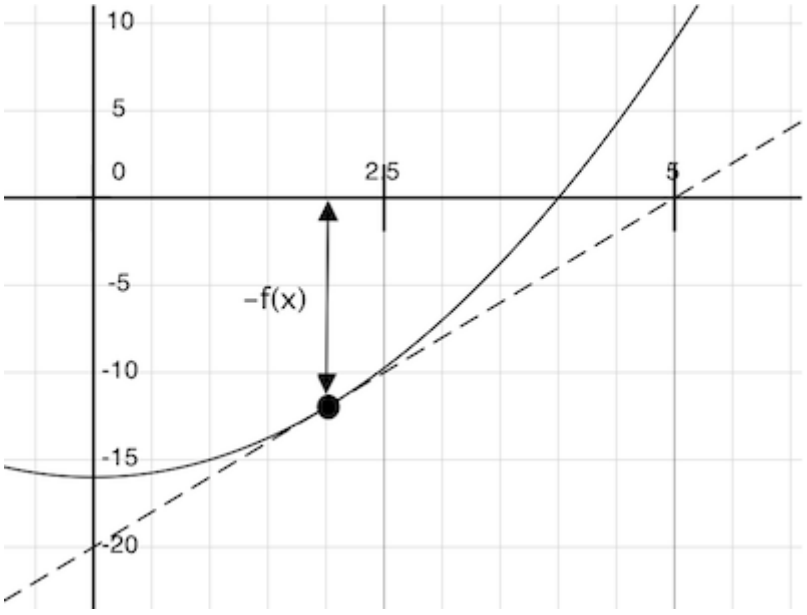
牛顿法（Newton's method）是一种经典的迭代方法，用于查找返回值为 0 的数学函数的参数。这些值（参数）称为函数的零点（Zeros）。找到函数的零点通常等同于解决了其他一些有趣的问题，例如求平方根。

在我们继续之前，有一个激励人心的评论说到：我们很容易想当然地认为我们知道如何计算平方根。不止是 Python，你的手机、网页浏览器或袖珍计算器都可以为你做到这一点。然而，学习计算机科学的一部分是理解这样的量如如何进行计算的，此处介绍的一般方法适用于求解 Python 内置方程之外的一大类方程。

牛顿法是一种迭代改进算法：它会对所有可微（differentiable）函数的零点的猜测值进行改进，这意味着它可以在任意点用直线进行近似处理。牛顿的方法遵循这些线性近似（linear approximations）来找到函数零点。

试想一条穿过点 $(x,f(x))$ 的直线与函数 $f(x)$ 在该点拥有相同的斜率。这样的直线称为切线（tangent），它的斜率我们称为 f 在 x 处的导数（derivative）。

这条直线的斜率是函数值变化量与函数自变量的比值。所以，按照 $f(x)$ 除以这个斜率来平移 x，就会得到切线到达 0 时的自变量的值。



`newton_update` 表示函数 f 及其导数 $\mathrm{d}f$ 沿着这条切线到 0 的计算过程。

```
>>> def newton_update(f, df):
        def update(x):
            return x - f(x) / df(x)
        return update
```

最后，我们可以使用 `newton_update`、`improve` 算法以及比较 $f(x)$ 是否接近 0 来定义 `find_root` 函数。

```
>>> def find_zero(f, df):
        def near_zero(x):
            return approx_eq(f(x), 0)
        return improve(newton_update(f, df), near_zero)
```

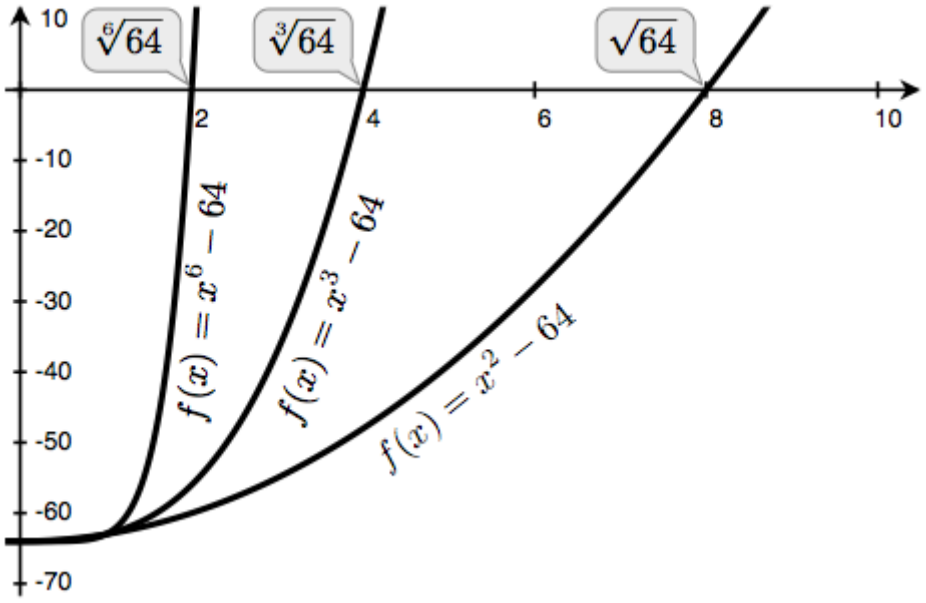
计算根：我们可以使用牛顿法来计算任意次方根，a 的 n 次方根就是使得 $x \cdot x \cdot x \cdots x = a$ 的重复 n 次的 x 的值，例如：

- 64 的平方根是 8, 因为 $8 \cdot 8 = 64$
- 64 的三次方根是 4, 因为 $4 \cdot 4 \cdot 4 = 64$
- 64 的六次方根是 2，因为 $2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 = 64$

我们可以使用牛顿法根据以下观察结果来计算根：

- 64 的平方根 (写作 $\sqrt{64}$) 是使得 $x^{2}- 64 = 0$ 的 x 的值
- 推广来说，a 的 n 次方根 (写作 $\sqrt [n]{a}$) 是使得 $x^{n}- a = 0$ 的 x 的值

如果我们可以找到最后一个方程的零点，那么我们就可以计算出 n 次方根。通过绘制 n 等于 2、3 和 6，a 等于 64 的曲线，我们可以将这种关系可视化。



我们首先通过定义 f 和它的导数 $\mathrm{d}f$ 来实现 `square_root` 函数，使用微积分中的知识，$f(x)=x^{2}-a$ 的导数是线性方程 $\mathrm{d}f(x)=2x$。

```
>>> def square_root_newton(a):
        def f(x):
            return x * x - a
        def df(x):
            return 2 * x
        return find_zero(f, df)

>>> square_root_newton(64)
8.0
```

推广到 n 次方根，我们可以得到 $f(x)=x^{n}-a$ 和它的导数 $\mathrm{d}f(x)=nx^{n-1}$。

```
>>> def power(x, n):
        """返回 x * x * x * ... * x，n 个 x 相乘"""
        product, k = 1, 0
        while k < n:
            product, k = product * x, k + 1
        return product

>>> def nth_root_of_a(n, a):
        def f(x):
            return power(x, n) - a
        def df(x):
            return n * power(x, n-1)
        return find_zero(f, df)

>>> nth_root_of_a(2, 64)
8.0
>>> nth_root_of_a(3, 64)
4.0
>>> nth_root_of_a(6, 64)
```

```
2.0
```

所有这些计算中的近似误差都可以通过将 `approx_eq` 中的公差 tolerance 改为更小的数字来减小。

当你使用牛顿法时，要注意它并不总是收敛（converge）的。`improve` 的初始猜测必须足够接近零，并且必须满足有关函数的各种条件。尽管有这个缺点，牛顿法仍是一种用于求解微分方程的强大的通用计算方法。现代计算机技术中的对数和大整数除法的快速算法，都采用了该方法的变体。

1.6.6 柯里化

我们可以使用高阶函数将一个接受多个参数的函数转换为一个函数链，每个函数接受一个参数。更具体地说，给定一个函数 `f(x, y)`，我们可以定义另一个函数 `g` 使得 `g(x)(y)` 等价于 `f(x, y)`。在这里，`g` 是一个高阶函数，它接受单个参数 `x` 并返回另一个接受单个参数 `y` 的函数。这种转换称为柯里化（Curring）。

例如，我们可以定义 `pow` 函数的柯里化版本：

```
>>> def curried_pow(x):
        def h(y):
            return pow(x, y)
        return h
>>> curried_pow(2)(3)
8
```

一些编程语言，例如 Haskell，只允许使用单个参数的函数，因此程序员必须对所有多参数过程进行柯里化。在 Python 等更通用的语言中，当我们需要一个只接受单个参数的函数时，柯里化就很有用。例如，map 模式（map pattern）就可以将单参数函数应用于一串值。在下个章节中，我们将看到更通用的 map 模式的示例，但现在，我们可以在函数中实现该模式：

```
>>> def map_to_range(start, end, f):
        while start < end:
            print(f(start))
            start = start + 1
```

我们可以使用 `map_to_range` 和 `curried_pow` 来计算 2 的前十次方，而不是专门编写一个函数来这样做：

```
>>> map_to_range(0, 10, curried_pow(2))
1
2
4
8
16
32
64
128
256
512
```

我们可以类似地使用相同的两个函数来计算其他数字的幂。柯里化允许我们这样做，而无需为每个我们希望计算其幂的数字编写特定的函数。

在上面的例子中，我们手动对 `pow` 函数进行了柯里化变换，得到了 `curried_pow`。相反，我们可以定义函数来自动进行柯里化，以及逆柯里化变换（uncurrying transformation）：

```
>>> def curry2(f):
        """返回给定的双参数函数的柯里化版本"""
        def g(x):
            def h(y):
                return f(x, y)
            return h
        return g
>>> def uncurry2(g):
        """返回给定的柯里化函数的双参数版本"""
        def f(x, y):
            return g(x)(y)
        return f
>>> pow_curried = curry2(pow)
>>> pow_curried(2)(5)
32
>>> map_to_range(0, 10, pow_curried(2))
1
2
4
8
16
32
64
128
256
512
```

`curry2` 函数接受一个双参数函数 `f` 并返回一个单参数函数 `g`。当 `g` 应用于参数 `x` 时，它返回一个单参数函数 `h`。当 `h` 应用于参数 `y` 时，它调用 `f(x, y)`。因此，`curry2(f)(x)(y)` 等价于 `f(x, y)`。`uncurry2` 函数反转了柯里化变换，因此 `uncurry2(curry2(f))` 等价于 `f`。

```
>>> uncurry2(pow_curried)(2, 5)
32
```

1.6.7 Lambda 表达式

到目前为止，每当我们想要定义一个新函数时，都需要给它取一个名字。但是对于其他类型的表达式，我们不需要将中间值与名称相关联。也就是说，我们可以计算 `a * b + c * d` 而不必命名子表达式 `a*b` 或 `c*d` 或完整的表达式。在 Python 中，我们可以使用 lambda 表达式临时创建函数，这些表达式会计算为未命名的函数。一个 lambda 表达式的计算结果是一个函数，它仅有一个返回表达式作为主体。不允许使用赋值和控制语句。

```
>>> def compose1(f, g):
        return lambda x: f(g(x))
```

我们可以通过构造相应的英文句子来理解 lambda 表达式的结构：

```
lambda              x         :              f(g(x))
"A function that    takes x   and returns    f(g(x))"
```

lambda 表达式的结果称为 lambda 函数（匿名函数）。它没有固有名称（因此 Python 打印 `<lambda>` 作为名称），但除此之外它的行为与任何其他函数都相同。

```
>>> s = lambda x: x * x
>>> s
<function <lambda> at 0xf3f490>
>>> s(12)
144
```

在环境图中，lambda 表达式的结果也是一个函数，以希腊字母 λ（lambda）命名。我们的 `compose` 示例可以用 lambda 表达式非常简洁地表示出来：

一些程序员认为，使用 lambda 表达式中的匿名函数可以让代码更简洁、更直观。然而，尽管复合 lambda 表达式很简洁，但它是出了名的难以辨认。以下的代码虽然没有错误，但很多程序员却很难快速理解它。

```
>>> compose1 = lambda f,g: lambda x: f(g(x))
```

一般来说，Python style 更喜欢使用明确的 `def` 语句而不是 lambda 表达式，但在需要简单函数作为参数或返回值的情况下可以使用它们。

这样的风格规则只是指导方针，你可以按照自己的方式编程。但是，在编写程序时，请考虑今后可能阅读你的程序的受众。如果你能够使程序更易于理解，那么你就是在帮助那些人。

术语 lambda 是历史上的偶然事件，它源于书面数学符号与早期排版系统的不兼容性。

> It may seem perverse to use lambda to introduce a procedure/function.
>
> The notation goes back to Alonzo Church, who in the 1930's started with a "hat" symbol, he wrote the square function as "ŷ . y × y".
>
> But frustrated typographers moved the hat to the left of the parameter and changed it to a capital lambda: "Λ y . y × y";
>
> From there the capital lambda was changed to lowercase, and now we see "λ y . y × y" in math books and `(lambda (y) (* y y))` in Lisp.
>
> — Peter Norvig (norvig.com/lispy2.html)

尽管其词源不寻常，但是 lambda 表达式和对应的函数应用的形式语言 --> λ 演算（lambda calculus）是计算机科学中的基本概念，不仅是被 Python 编程社区广泛使用。在第三章学习解释器的设计时，我们将重新讨论这个问题。

1.6.8 抽象和一等函数

我们在本节的开头提到了用户定义函数是一种至关重要的抽象机制，因为它们使我们能够将计算的一般方法表达为编程语言中的显式元素。现在，我们已经学习了如何使用高阶函数来操作这些一般方法，以创建更进一步的抽象。

作为程序员，我们应该警觉地寻找发现我们程序中的基本抽象，然后对其进行扩展，并加以推广去创建更强大的抽象。这并不是说我们应该总是以最抽象的方式编写程序，有经验的程序员知道如何选择符合任务要求的抽象级别。不过，重要的是我们能够思考这些抽象的概念，这样我们就能准备好将其应用到新的环境中。高阶函数的重要性在于，它们使我们能够将这些抽象显式地表示为我们编程语言中的元素，以便可以像其他计算元素一样进行处理。

一般而言，编程语言会对计算元素的操作方式施加限制。拥有最少限制的元素可以获得一等地位（first-class status）。这些一等元素的"权利和特权"包括：

1. 可以与名称绑定
2. 可以作为参数传递给函数
3. 可以作为函数的结果返回
4. 可以包含在数据结构中

Python 授予函数完全的一等地位，由此带来的表达能力的提升是巨大的。

1.6.9 函数装饰器

Python 提供了一种特殊的语法来使用高阶函数作为执行 `def` 语句的一部分，称为装饰器（decorator）。最常见的例子也许就是 `trace`：

```
>>> def trace(fn):
        def wrapped(x):
            print('-> ', fn, '(', x, ')')
            return fn(x)
        return wrapped

>>> @trace
    def triple(x):
        return 3 * x

>>> triple(12)
->  <function triple at 0x102a39848> ( 12 )
36
```

在这个例子中，定义了一个高阶函数 `trace`，它返回一个函数，该函数在调用其参数前先输出一个打印语句来显示该参数。`triple` 的 `def` 语句有一个注解（annotation） `@trace`，它会影响 `def` 执行的规则。和往常一样，函数 `triple` 被创建了。但是，名称 triple 不会绑定到这个函数上。相反，这个名称会被绑定到在新定义的 `triple` 函数调用 `trace` 后返回的函数值上。代码中，这个装饰器等价于：

```
>>> def triple(x):
        return 3 * x
>>> triple = trace(triple)
```

在本教材相关的项目中，装饰器被用于追踪，以及在从命令行运行程序时选择要调用哪些函数。

对于专家的额外内容：装饰器符号 `@` 也可以后跟一个调用表达式。跟在 `@` 后面的表达式会先被解析（就像上面的 'trace' 名称一样），然后是 `def` 语句，最后将装饰器表达式的运算结果应用到新定义的函数上，并将其结果绑定到 `def` 语句中的名称上。