1.3 定义新的函数

::: details INFO
译者：Mancuoj

来源：1.3 Defining New Functions

对应：Lab 01
:::

我们已经在 Python 中确定了强大的编程语言中一些必须出现的要素：

1. 原始的内置数据和函数：数字和算术运算
2. 组合方式：嵌套函数
3. 受限的抽象方式：将名称与值绑定

现在我们来学习 **函数定义**，这是一种更为强大的抽象技术，通过它可以将名称与复合操作绑定为一个单元。

首先来研究一下 **平方** 的概念。我们可能会说：“平方就是一个数乘以它本身。”这在 Python 中可以表示为：

```
>>> def square(x):
        return mul(x, x)
```

上面的代码定义了一个名为 `square` 的新函数，这个用户定义的函数并不会内置到解释器中，它表示将某值与自身相乘的复合运算。这个定义将 `x` 作为被乘的东西的名称，称为 **形式参数**，同时也将此函数与名称 `square` 绑定。

如何定义函数：函数定义包含 `def` 语句、`<name 函数名>` 和一个以逗号分隔的 `<formal parameters 形式参数>` 列表，然后是一个被称为函数体的 `return` 语句，它指定了调用函数时要计算的表达式，也就是函数的 `<return expression 返回表达式>`：

```
def <name>(<formal parameters>):
    return <return expression>
```

函数的第二行 **必须** 进行缩进，大多数程序员使用四个空格。返回表达式会作为新定义的函数的一部分存储，并且仅在最终调用该函数时才进行求值。

定义了 `square` 之后，我们可以调用它：

```
>>> square(21)
441
>>> square(add(2, 5))
49
>>> square(square(3))
81
```

我们还可以将 `square` 作为一个构建单元来定义其他函数。例如，我们可以很容易地定义一个函数 `sum_squares`，给定任意两个数字作为参数，返回它们的平方和：

```
>>> def sum_squares(x, y):
        return add(square(x), square(y))

>>> sum_squares(3, 4)
25
```

用户定义函数的使用方式与内置函数完全相同。实际上，从 `sum_squares` 的定义中我们并不能判断 `square` 是内置于解释器中，还是从模块中导入的，又或是用户定义的。

`def` 语句和赋值语句都将名称与值绑定，并且绑定后任何之前的绑定都将丢失。例如，下面的 `g` 首先指的是一个没有参数的函数，然后是指一个数字，最后是一个含有两个参数的函数。

```
>>> def g():
        return 1
>>> g()
1
>>> g = 2
>>> g
2
>>> def g(h, i):
        return h + i
>>> g(1, 2)
3
```

帧在调用时创建，返回时销毁

1.3.1 环境

偏向动态　偏向静态

帧(frame)与变量作用域大致是相同的

虽然我们现在的 Python 子集已经足够复杂，但程序的含义并不明显。如果形参与内置函数同名怎么办？两个函数可以共享名称而不会混淆吗？要解决这些问题，我们必须更详细地描述环境。

求解表达式的环境由 **帧** 序列组成，它们可以被描述为一些盒子。每个帧都包含了一些 **绑定**，它们将名称与对应的值相关联。**全局** 帧（global frame）只有一个。赋值和导入语句会将条目添加到当前环境的第一帧。目前，我们的环境仅由全局帧组成。

注：PDF中无法查看

此 **环境图** 显示了当前环境中的绑定，还有名称和值的绑定。本文中的环境图是交互式的：你可以逐步运行左侧程序的每一行，然后在右侧查看环境状态的演变。你还可以单击“Edit this code”以将示例加载到 Online Python Tutor 中，它是由 Philip Guo 创建的用于生成环境图的工具。希望你能够自己去创建示例，研究对应生成的环境图。

函数也会出现在环境图中。`import` 语句将名称与内置函数绑定。`def` 语句将名称与用户自定义的函数绑定。导入 `mul` 并定义 `square` 后的结果环境如下所示：

每个函数都是一行，以 `func` 开头，后面是函数名称和形式参数。`mul` 等内置函数没有正式的参数名称，所以都是使用 `...` 代替。

函数名称重复两次，一次在环境帧中，另一次是作为函数定义的一部分。函数定义中出现的名称叫做 **内在名称（intrinsic name）**，帧中的名称叫做 **绑定名称（bound name）**。两者之间有一个区别：不同的名称可能指的是同一个函数，但该函数本身只有一个内在名称。

绑定到帧中的函数名称是在求值过程中使用，而内在名称在求值中不起作用。使用 Next 按钮逐步执行下面的示例，可以看到一旦名称 `max` 与数字值 3 绑定，它就不能再用作函数。

错误信息“TypeError: 'int' object is not callable”报告了名称 `max`（当前绑定到数字 3）是一个整数而不是函数，所以它不能用作调用表达式中的运算符。

函数签名：每个函数允许采用的参数数量有所不同。为了跟踪这些要求，我们绘制了每个函数的名称及其形式参数。用户定义的函数 `square` 只需要 `x` 一个参数，提供或多或少的参数都将导致错误。对函数形式参数的描述被称为函数的签名。

函数 `max` 可以接受任意数量的参数，所以它被呈现为 `max(...)`。因为原始函数从未明确定义，所以无论采用多少个参数，所有的内置函数都将呈现为 `<name>(...)`。

1.3.2 调用用户定义的函数

为了求出操作符为用户定义函数的调用表达式，Python 解释器遵循了以下计算过程。与其他任何调用表达式一样，解释器将对操作符和操作数表达式求值，然后用生成的实参调用具名函数。

调用用户定义的函数会引入局部帧（local frame），它只能访问该函数。通过一些实参调用用户定义的函数：

1. 在新的局部帧中，将实参绑定到函数的形参上。
2. 在以此帧开始的环境中执行函数体。

求值函数体的环境由两个帧组成：一是包含形式参数绑定的局部帧，然后是包含其他所有内容的全局帧。函数的每个实例都有自己独立的局部帧。

为了详细说明一个例子，下面将会描述相同示例的环境图中的几个步骤。执行第一个 `import` 语句后，只有名称 `mul` 被绑定在全局帧中。

首先，执行定义函数 `square` 的语句。请注意，整个 `def` 语句是在一个步骤中处理的。直到函数被调用（而不是定义时），函数体才会被执行。

接下来，使用参数 -2 调用 `square` 函数，它会创建一个新的帧，将形式参数 `x` 与 -2 绑定。

然后在当前环境中查找名称 `x`，它由所示的两个帧组成，而在这两种情况下，`x` 的结果都是 -2，因此此 `square` 函数返回 4。

`square()` 帧中的“返回值”不是名称绑定的值，而是调用创建帧的函数返回的值。

即使在这个简单的示例中，也使用了两个不同的环境。顶级表达式 `square(-2)` 在全局环境中求值，而返回表达式 `mul(x, x)` 在调用 `square` 时创建的环境中求值。虽然 `x` 和 `mul` 都在这个环境中，但在不同的帧中。

环境中帧的顺序会影响通过表达式查找名称而返回的值。我们之前说过，名称会求解为当前环境中与该名称关联的值。我们现在可以更准确地说：

名称求解（Name Evaluation）：在环境中寻找该名称，最早找到的含有该名称的帧，其里边绑定的值就是这个名称的计算结果。

环境、名称和函数的概念框架构成了求解模型，虽然一些机械细节仍未指定（例如，如何实现绑定），但我们的模型确实精准地描述了解释器如何求解调用表达式。在第三章中，我们将看到这个模型如何作为蓝图来实现编程语言的工作解释器。

1.3.3 示例：调用用户定义的函数

让我们再次思考两个简单的函数定义，并说明计算用户定义函数的调用表达式的过程。

Python 首先求解名称 `sum_squares`，并将它绑定到全局帧中的用户定义的函数，而原始数值表达式 5 和 12 的计算结果为它们所代表的数字。

接下来 Python 会调用 `sum_squares`，它引入了一个局部帧，将 `x` 绑定到 5，将 `y` 绑定到 12。

`sum_squares` 的主体包含此调用表达式：

```
  add     (  square(x)  ,  square(y)  )
________     _________     _________
operator     operand 0     operand 1
```

所有三个子表达式都在当前环境中计算，且始于标记为 `sum_squares()` 的帧。运算符子表达式 `add` 是在全局帧中找到的名称，它绑定到了内置的加法函数上。在调用加法之前，必须依次求解两个操作数子表达式，两个操作数都在标记为 `sum_squares` 的帧的环境中计算。

在 `operand 0` 中，`square` 在全局帧中命名了一个用户定义的函数，而 `x` 在局部帧中命名了数字 5。Python 通过引入另一个将将 `x` 与 5 绑定的局部帧来调用 `square` 函数。

此环境下表达式 `mul(x, x)` 的计算结果为 25。

继续求解 `operand 1`，其中 y 的值为 12。Python 会再次对 `square` 的函数体进行求解，此时引入另一个将 `x` 与 12 绑定的局部帧，计算结果为 144。

最后，对参数 25 和 144 调用加法得到 `sum_squares` 的最终返回值：169。

这个例子展示了我们到目前为止学到的许多基本思想。将名称绑定到值，而这些值会分布在多个无关的局部帧，以及包含共享名称的单个全局帧中。每次调用函数时都会引入一个新的局部帧，即使是同一个函数被调用两次。

所有这些机制的存在都是为了确保名称在程序执行期间的正确时间解析为正确的值。这个例子说明了为什么我们之前介绍了“模型需要复杂性”。所有三个局部帧都包含名称 `x` 的绑定，但该名称会与不同帧中的不同值进行绑定，局部帧会将这些名称分开。

1.3.4 局部名称

实现函数的一个细节就是，实现者为函数的形参选择的名称不应该影响函数行为。所以，以下函数应该提供相同的行为：

```
>>> def square(x):
        return mul(x, x)
>>> def square(y):
        return mul(y, y)
```

一个函数的含义应该与编写者选择的参数名称无关，这个原则对编程语言有重要的意义。最简单的就是函数的参数名称必须在保持函数体局部范围内。

如果参数不是它们各自函数体的局部参数，那么 `square` 中的参数 `x` 可能会与 `sum_squares` 中的参数 `x` 混淆。但情况并非如此：`x` 在不同局部帧中的绑定是不相关的。计算模型经过精心设计以确保这种无关性。

局部名称的作用域限于定义它的函数的主体，当一个名称不可再访问时，就是超出了作用域。这种界定作用域的行为并不是我们模型的新细节，而是环境工作方式的结果。

1.3.5 选择名称

名称的可修改性并不意味着形式参数名称不重要。相反，精心选择的函数和参数名称对于程序的可解释性至关重要！

以下指南改编自 Python 代码风格指南，它可以作为所有（非叛逆的）Python 程序员的指南。这些共享的约定使开发者社区的成员之间的沟通能够顺利进行。作为遵循这些约定的副作用，你会发现你的代码在内部变得更加一致。

1. 函数名是小写的，单词之间用下划线分隔。鼓励使用描述性名称。
2. 函数名称通常反映解释器应用于参数的操作（例如， `print, add, square` ）或结果（例如， `max, abs, sum` ）。
3. 参数名称是小写的，单词之间用下划线分隔。首选单个词的名称。
4. 参数名称应该反映参数在函数中的作用，而不仅仅是允许的参数类型。
5. 当作用明确时，单字参数名称可以接受，但应避免使用 l（小写的 L）和 O（大写的 o）或 I（大写的 i）以避免与数字混淆。

小心那些用一堆下划线命名的人()

当然这些准则也有许多例外，即使在 Python 标准库中也是如此。像英语的词汇一样，Python 继承了各种贡献者的词汇，所以结果并不总是一致的。

1.3.6 抽象函数

虽然 `sum_squares` 这个函数非常简单，但它体现了用户自定义函数最强大的特性。函数 `sum_squares` 是根据函数 `square` 定义的，但仅依赖于 `square` 在其输入参数和输出值之间定义的关系。

我们可以编写 `sum_squares` 而不用关心如何对一个数求平方。如何计算平方的细节可以被隐藏到之后再考虑。确实，对于 `sum_squares` 而言， `square` 并不是一个特定的函数体，而是一个函数的抽象，也就是所谓的函数抽象（functional abstraction）。在这个抽象层次上，任何计算平方的函数都是等价的。

所以在只考虑返回值的情况下，以下两个计算平方数的函数是无法区分的：它们都接受数值参数并返回该数的平方值。

```
>>> def square(x):
        return mul(x, x)
>>> def square(x):
        return mul(x, x-1) + x
```

换句话说，函数定义能够隐藏细节。用户可能不会自己去编写函数，而是从另一个程序员那里获得它，然后将它作为一个“黑盒”，用户只需要调用，而不需要知道实现该功能的细节。Python 库就具有此属性，许多开发人员使用这里定义的函数，但很少有人去探究它们的实现。

抽象函数的各个方面：思考抽象函数的三个核心属性通常对掌握其使用很有帮助。**函数的域 domain** 是它可以接受的参数集合；**范围 range** 是它可以返回的值的集合；**意图 intent** 是计算输入和输出之间的关系（以及它可能产生的任何副作用）。通过域、范围和意图来理解函数抽象对之后能在复杂程序中正确使用它们至关重要。

例如，我们用于实现 `sum_squares` 的任何平方函数应具有以下属性：

- **域** 是任意单个实数。
- **范围** 是任意非负实数。
- **意图** 是计算输入的平方作为输出。

这些属性不会描述函数是如何执行的，这个细节已经被抽象掉了。

1.3.7 运算符

数学运算符（例如 + 和 -）为我们提供了组合方法的第一个示例，但我们尚未给包含这些运算符的表达式定义求值过程。

带有中缀运算符的 Python 表达式都有自己的求值过程，但你通常可以将它们视为调用表达式的简写形式。当你看到

```
>>> 2 + 3
5
```

可以认为简单地将它理解为以下代码

```
>>> add(2, 3)
5
```

中缀表示法可以嵌套，就像调用表达式一样。Python 运算符优先级采用了正常数学规则，它规定了如何求解具有多个运算符的复合表达式。

```
>>> 2 + 3 * 4 + 5
19
```

它和以下表达式的求解结果完全相同

```
>>> add(add(2, mul(3, 4)), 5)
19
```

调用表达式中的嵌套比运算符版本更加明显，但也更难以阅读。Python 还允许使用括号对子表达式进行分组，用以覆盖正常的优先级规则，或使表达式的嵌套结构更加明显。

```
>>> (2 + 3) * (4 + 5)
45
```

它和以下表达式的求解结果完全相同

```
>>> mul(add(2, 3), add(4, 5))
45
```

对于除法，Python 提供了两个中缀运算符：`/` 和 `//`。前者是常规除法，因此即使除数可以整除被除数，它也会产生 **浮点数**（十进制小数）：

```
>>> 5 / 4
1.25
>>> 8 / 4
2.0
```

而后一个运算符 `//` 会将结果向下舍入到一个整数：

```
>>> 5 // 4
1
>>> -5 // 4
-2
```

这两个运算符算是 `truediv` 和 `floordiv` 函数的简写。

```
>>> from operator import truediv, floordiv
>>> truediv(5, 4)
1.25
>>> floordiv(5, 4)
1
```

你可以在程序中随意使用中缀运算符和圆括号。对于简单的数学运算，Python 惯例上更喜欢使用运算符而不是调用表达式。 谁没事用这个……