Contents

Part I

Part II

Part III

Chapter 1

Chapter 2

元组、文件与其他核心类型

Chapter 3

迭代和推导

3.1 迭代器：初次探索

出于明确性，这里倾向于使用术语可迭代对象来指代有一个支持 iter 调用的对象，使用术语迭代器来指代一个（iter 调用为传入的可迭代对象返回的）支持 next(I) 调用的对象。

3.1.1 迭代协议：文件迭代器

所有带有 __next__ 方法的对象会前进到下一个结果，而在一系列结果的末尾时，则会引发 StopIteration 异常，这种对象在 Python 中也被称为迭代器。任何这类对象也能以 for 循环或其他迭代工具遍历，因为所有迭代工具内部工作起来都是在每次迭代中调用 __next__，并且捕捉 StopIteration 异常来确定何时离开。

while 循环会比基于迭代器的 for 循环运行得更慢，因为迭代器在 Python 中是以 C 语言的速度运行的，而 while 循环版本则是通过 Python 虚拟机运行 Python 字节码的。任何时候，我们把 Python 代码换成 C 程序代码，速度都应该会变快。然而，并非绝对如此。

3.1.2 手动迭代：iter 和 next

迭代协议还有一点值得注意。当 for 循环开始时，会通过它传给 iter 内置函数，以便从可迭代对象中获得一个迭代器，返回的对象含有需要的 next 方法。

完整得迭代协议

作为更正式的定义，Figure 3.1 描绘了这个完整的迭代协议，Python 中的每个迭代工具都使用它，并受到各种对象类型的支持。它实际上基于两个对象，由迭代工具在两个不同的步骤中使用：

- 您请求迭代的可迭代对象，其 __iter__ 由 iter 运行
- 由迭代器返回的迭代器对象，在迭代过程中实际产生值，其 __next__ 由 next 运行，并在完成产生结果时引发 StopIteration

文件对象就是自己的迭代器。由于文件只支持一次迭代（它们不能通过方向查找来支持多重扫描），文件有自己的 __next__ 方法。

列表以及很多其他的内置对象，不是自身的迭代器，因为它们支持多次打开迭代器，例如，嵌套循环中可能在不同位置有多次迭代。对这样的对象，我们必须调用 iter 来启动迭代。

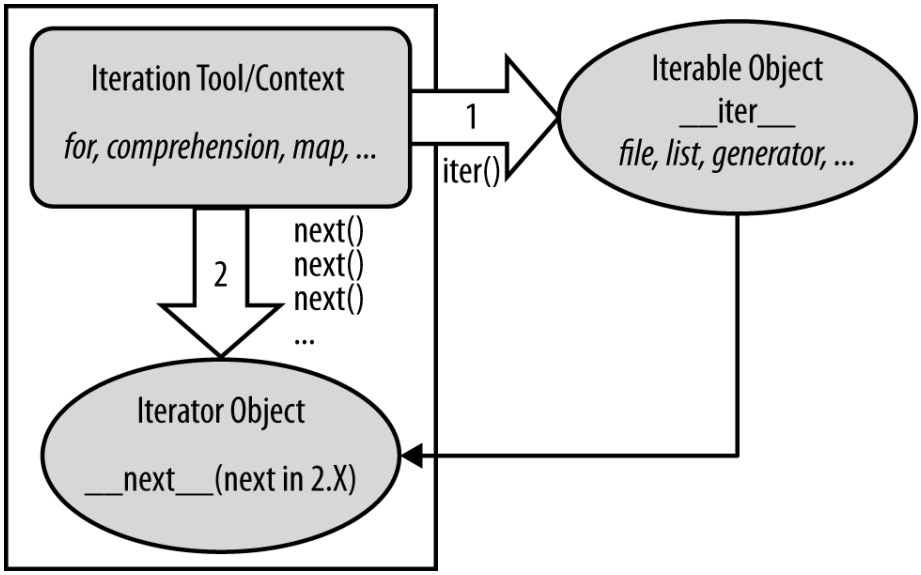

Figure 3.1: Python 迭代协议，由 for 循环、推导式、映射等使用，并受文件、列表、字典、Chapter 7 的生成器等支持。有些对象既是迭代上下文又是可迭代对象，例如生成器表达式和 3.X 风格的某些工具（例如 map 和 zip）。有些对象既是可迭代的又是迭代器，为 iter() 调用返回自身，这就是一个无操作。

3.1.3 其他内置类型迭代器

除了文件以及像列表这样的实际的序列外，其他类型也有其适用的迭代器。例如，遍历字典键的经典方法是明确地获取其键的列表。

不过，在最近的 Python 版本中，字典有一个迭代器，在迭代环境中，会自动一次返回一个键。直接的效果是，我们不再需要调用 keys 方法来遍历字典键——for 循环将使用迭代协议在每次迭代的时候获取一个键。

迭代协议也是我们必须把某些结果包装到一个list调用中以一次性看到它们的值的原因。可迭代的对象一次返回一个结果，而不是一个实际的列表。

实际上，Python 中可以从左向右扫描的所有对象都已同样的方式实现了迭代协议。

3.2 列表推导：初次深入探索

与 for 循环一起使用的列表推导，是最主要的迭代协议上下文之一。

3.2.1 列表推导基础

从语法上讲，其语法源自于集合理论表示法中的一个结构，该结构对集合中的每个元素应用一个操作。

为了在语法上进行了解，让我们更详细地剖析一个例子：

```
>>> L = [x + 10 for x in L]
```

列表解析写在一个方括号中，因为它们最终是构建一个新的列表的一种方式。它们以我们所组成的一个任意的表达式开始，该表达式使用我们所组成的一个循环变量(x + 10)。这后边跟着我们现在应该看做是一个 for 循环头部的部分，它声明了循环变量，以及一个可迭代对象（for x in L）。

要运行该表达式，Python 在解释器内部执行一个遍历 L 的迭代，按照顺序把 x 赋给每个元素，并且收集对各元素运行左边的表达式的结果。我们得到的结果列表就是列表解析所表达的内容——包含了 x + 10 的一个新列表，针对 L 中的每个 x。

从技术上讲，列表解析并非真的是必需的，因为我们总是可以用一个 for 循环手动地构建一个表达式结果的列表。然而，列表解析编写起来更加精简，并且由于构建结果列表的这种代码样式在 Python 代码中十分常见，因此可以将它们用于多种环境。此外，列表解析比手动的 for 循环语句运行的更快（往往速度会快一倍），因为它们的迭代在解释器内部是以 C 语言的速度执行的，而不是以手动 Python 代码执行的，特别是对于较大的数据集合，这是使用列表解析的一个主要的性能优点。

3.2.2 在文件上使用列表推导

回顾 Chapter 2 中提到的文件对象自身会在垃圾回收的时候自动关闭。因此，文件的列表推导也会自动地在表达式运行结束后，将它们的临时文件对象关闭。然而对于 CPython 以外的 Python 版本，你可能要手动编写代码来关闭循环中的文件对象，以保证资源能够被立即释放。

3.2.3 列表推导语法的拓展

筛选分句：if

作为一个特别有用的扩展，表达式中嵌套的 for 循环可以有一个相关的 if 子句，来过滤那些测试不为真的结果项。

嵌套循环：for

如果需要的话，列表推导甚至可以变得更复杂–例如，我们可以通过编写一些列 for 分句，让推导包含嵌套的循环。实际上，它们的完整语法运行任意数目的 for 分句，并且每个 for 分句都可以带一个可选的关联的 if 分句。（不建议超过两个，否则阅读起来不方便）

3.3 其他迭代上下文

任何利用了迭代协议的工具，都能在遵循了迭代协议的任何内置类型或用户定义的类上自动地工作。

列表解析、in 成员关系测试、map 内置函数以及像 sorted 和 zip 调用这样的内置函数也都使用了迭代协议。当应用于一个文件时，所有这些使用文件对象的迭代器都自动地按行扫描，通过 __iter__ 获取一个迭代器并每次调用 __next__ 方法。

Python 还包含了各种处理迭代的其他内置函数：sorted 排序可迭代对象中的各项，zip 组合可迭代对象中的各项，enumerate 根据相对位置来配对可迭代对象中的项，filter 选择一个函数为真的项，reduce 针对可迭代对象中的成对的项运行一个函数。所有这些都接受一个可迭代的对象，并且在 Python3.0 中，zip、enumerate 和 filter也像 map 一样返回一个可迭代对象。

有趣的是，在当今的 Python 中，迭代协议甚至比我们目前所能展示的示例要更为普遍——Python 的内置工具集中从左到右地扫描一个对象的每项工具，都定义为在主体对象上使用了迭代协议。这甚至包含了更高级的工具，例如 list 和 tuple 内置函数（它们从可迭代对象构建了一个新的对象），字符串 join 方法（它将一个子字符串放置到一个可迭代对象中包含的字符串之间），甚至包括序列赋值。

甚至其他一些工具也出人意料地属于这个类别。例如，序列赋值、in 成员测试、切片赋值和列表的 extend 方法都利用了迭代协议来扫描。

由 Chapter 1 可知，extend 可以自动迭代，但 append 却不能，用 append 给一个列表添加一个可迭代对象并不会对这个对象进行迭代。不过这个对象之后可以从列表中取出来进行迭代。

3.4 Python3.X 中的新的可迭代对象

Python 3.X 中的一个基本的改变是，它比 Python 2.X 更强调迭代。除了与文件和字典这样的内置类型相关的迭代，字典方法 keys、values 和 items 都在 Python 3.X中返回可迭代对象，就像内置函数 range、map、zip 和 filter 所做的那样。

正如 Chapter 7 所述，对于新的迭代工具（例如 zip 和 map），它们只支持单边的扫描，如果要支持多遍扫描，就必须将它们转换成列表—与 2.X 中它们对应的列表形式不同，在 3.X 中的一次遍历会耗尽它们的值。

3.4.1 range 可迭代对象

在 Python 3.X 中，它返回一个迭代器，该迭代器根据需要产生范围中的数字，而不是在内存中构建一个结果列表。这取代了较早的 Python 2.X xrange，如果需要一个范围列表的话，你必须使用 list(range(...)) 来强制一个真正的范围列表。

和在 Python 2.X中返回的列表不同，Python 3.X 中的 range 对象只支持迭代、索引以及len函数。它们不支持任何其他的序列操作（如果你需要更多列表工具的话，使用 list(...)）

3.4.2 map、zip 和 filter 可迭代对象

和 range 类似，map、zip 以及 filter 内置函数在 Python 3.X 中也转变成迭代器以节约内存空间，而不再在内存中一次性生成一个结果列表。所有这 3 个函数不仅像是在 Python 2.X 一样处理可迭代对象，而且在 Python 3.X 中返回可迭代结果。和 range 不同，它们都是自己的迭代器——在遍历其结果一次之后，它们就用尽了。换句话说，不能在它们的结果上拥有在那些结果中保持不同位置的多个迭代器。

和其他迭代器一样，如果确实需要一个列表的话，可以用 list(...) 来强制一个列表，但是，对于较大的结果集来说，默认的行为可以节省不少内存空间。

zip 内置函数，返回以同样方式工作的迭代器。filter 内置函数，也是类似的。对于传入的函数返回 True 的可迭代对象中的每一项，它都会返回该项。

3.4.3 多遍迭代器 VS 单遍迭代器

对比 range 对象与本小节介绍的内置函数的不同之处十分钟重要，它支持 len 和索引，它不是自己的迭代器（手动迭代时，我们使用 iter 产生一个迭代器），并且，它支持在其结果上的多个迭代器，这些迭代器会记住它们各自的位置。

相反，3.X 中的 zip，map 和 filter **不支持同一结果上的多个活跃迭代器**；因此，iter 调用对遍历这类对象的结果是可选的—它们的 iter 结果就是它们自身。

当我们在 Chapter 8 使用类来编写自己的可迭代对象的时候，将会看到通常通过针对 iter 调用返回一个新的对象，来支持多个迭代器；单个的迭代器一般意味着一个对象返回其自身。在 Chapter 7 中，生成器函数和表达式的行为就像 map 和 zip 一样支持单个的活跃迭代器，而不是像 range 一样。在 Chapter 7 中，我们将会看到一些微妙的例子：位于循环中的一个单个的迭代器试图多次扫描—之前天真地将它们当作列表的代码，会因为没有手动进行列表转换而失效。

3.4.4 字典试图可迭代对象

在 Python 3.X 中，字典的 keys、values 和 items 方法返回可迭代的视图对象，它们一次产生一个结果项，而不是在内存中一次产生全部结果列表。视图项保持和字典中的那些项相同的物理顺序，并且反映对底层的字典做出的修改。

Python 3.X 字典仍然有自己的迭代器，它返回连续的键。因此，无需直接在此环境中调用 keys。

最后，再次提醒，由于 keys 不再返回一个列表，按照排序的键来扫描一个字典的传统编码模式在 Python 3.X 中不再有效。相反，首先用一个 list 调用来转换 keys 视图，或者在一个键视图或字典自身上使用 sorted 调用。我不确定这是否正确，截止 2024-01-25，Python 最新的版本中，字典已经是有序的了。

3.5 其他迭代器主题

我们还将在 Chapter 7 学习列表解析和迭代器的更多内容，在 Chapter 8 学习类的时候，我们还将再次遇到它们。在后面，我们将会看到：

- 使用 yield 语句，用户定义的函数可以转换为可迭代的生成器函数。
- 当编写在圆括号中的时候，列表解析转变为可迭代的生成器表达式。
- 用户定义的类通过 __iter__ 或 __getitem__ 运算符重载变得可迭代。

Part IV

函数和生成器

Chapter 4

函数基础

简而言之，一个函数就是将一些语句集合在一起的部件，它们能够不止一次地在程序中运行。函数还能够计算出一个返回值，并能够改变作为函数输入的参数，而这些参数在代码运行时也许每次都不相同。

更具体地说，函数是在编程过程中剪剪贴贴的替代——我们不再有一个操作的代码的多个冗余副本，而是将代码包含到一个单独的函数中。通过这样做，我们可以大大减少今后的工作：如果这个操作之后必须要修改，我们只需要修改其中的一份拷贝，而不是所有代码。

函数是 Python 为了代码最大程度的重用和最小化代码冗余而提供的最基本的程序结构。

我们将学习与函数相关的主要语句和表达式，其中包含了函数调用语句、两种声明函数的方式（def 和 lambda）、两种管理作用域的方式（global 和 nonlocal），以及两种传回返回值的方式（return 和 yield）。

4.1 编写函数

下面是一个关于Python函数背后的一些主要概念的简要介绍：

- def 是可执行的代码。Python 的函数是由一个新的语句编写的，即 def。不像 C 这样的编译语言，def 是一个可执行的语句——函数并不存在，直到 Python 运行了 def 后才存在。事实上，在 if 语句、while 循环甚至是其他的 def 中嵌套是合法的（甚至在某些场合还很有效）。在典型的操作中，def语句在模块文件中编写，并自然而然地在模块文件第一次被导入的时候生成定义的函数。
- def 创建了一个对象并将其赋值给某一变量名。当 Python 运行到 def 语句时，它将会生成一个新的函数对象并将其赋值给这个函数名。就像所有的赋值一样，函数名变成了某一个函数的引用。函数名其实并没有什么神奇——就像你将看到的那样，函数对象可以赋值给其他的变量名，保存在列表之中。
- lambda 创建一个对象但将其作为结果返回。也可以用 lambda 表达式创建函数，这一功能允许我们把函数定义内联到语法上一条def语句不能工作的地方（这是一个更加高级的概念，我们推迟到 Chapter 6 介绍）。

- return 将一个结果对象发送给调用者。当函数被调用时，其调用者停止运行直到这个函数完成了它的工作，之后函数才将控制权返回调用者。函数是通过 return 语句将计算得到的值传递给调用者的，返回值成为函数调用的结果。
- yield 向调用者发回一个结果对象，但是记住它离开的地方。像生成器这样的函数也可以通过 yield 语句来返回值，并挂起它们的状态以便稍后能够恢复状态。这是本书稍后要介绍的另一个高级话题。
- global 声明了一个模块级的变量并被赋值。在默认情况下，所有在一个函数中被赋值的对象，是这个函数的本地变量，并且仅在这个函数运行的过程中存在。为了分配一个可以在整个模块中都可以使用的变量名，函数需要在 global 语句中将它列举出来。通常情况下，变量名往往需要关注它的作用域（也就是说变量存储的地方），并且是通过实赋值语句将变量名绑定至作用域的。
- nonlocal 声明了将要赋值的一个封闭的函数变量。类似的，Python 3.X 中添加的 nonlocal 语句允许一个函数来赋值一条语法封闭的 def 语句的作用域中已有的名称。这就允许封闭的函数作为保留状态的一个地方——当一个函数调用的时候，信息被记住了——而不必使用共享的全局名称。
- 函数是通过赋值（对象引用）传递的。在 Python 中，参数通过赋值传递给了函数（也就是说，就像我们所学过的，使用对象引用）。正如你将看到的那样，Python 的模式中，调用者以及函数通过引用共享对象，但是不需要别名。改变函数中的参数名并不会改变调用者中的变量名，但是改变传递的可变对象可以改变调用者共享的那个对象。
- 除非你显式指明形式参数与实际参数的对应，否则实际参数按位置赋值给形式参数。
- 参数、返回值以及变量不需要被声明。

def 语句将创建一个函数对象并将其赋值给一个变量名。def 语句一般的格式如下所示：

```
def name(arg1, arg2,... argN):
    statements
```

函数体通常包含一条 return 语句：

```
def name(arg1, arg2,... argN):
    ...
    return value
```

Python 的 return 语句可以在函数主体中的任何地方出现。它表示函数调用的结束，并将结果返回至函数调用处。return 语句包含一个对象表达式，这个对象给出的函数的结果。return 语句是可选的。如果它没有出现，那么函数将会在控制流执行完函数主体时结束。从技术角度来讲，一个没有返回值的函数自动返回了 None 对象，但是这个值是往往被忽略掉的。

函数也许会有 yield 语句，这在每次都会产生一系列值时被用到，这在 Chapter 7 我们研究函数的高级话题时才会讨论到。

4.1.1 def 语句执行于运行时

Python 的 def 语句实际上是一个可执行的语句：当它运行的时候，它创建一个新的函数对象并将其赋值给一个变量名。（请记住，Python 中所有的语句都是实时运行的，没有像独立的编译时间这样的流程）。

因为函数定义是实时发生的，所以对于函数名来说并没有什么特别之处。关键之处在于函数名所引用的那个对象。

4.2 第一个例子：定义和调用

4.2.1 Python 中的多态

Python 将对某一对象在某种语法的合理性交由对象自身来判断。这种依赖类型的行为称为多态，其含义就是一个操作的意义取决于被操作对象的类型。因为 Python 是动态类型语言，所以多态在 Python 中随处可见。

在 Python 中，代码不应该关心特定的数据类型。如果不是这样，那么代码将只对编写时你所关心的那些类型有效，对以后的那些可能会编写的兼容对象类型并不支持，这样做会打乱代码的灵活性。大体上来说，我们在 Python 中为对象编写接口，而不是数据类型[1]。

4.2.2 局部变量

所有的在函数内部进行赋值的变量名都默认为局部变量。所有的局部变量都会在函数调用时出现，并在函数退出时消失。正因如此，一个函数的变量不会在两次调用期间记忆值；尽管一个函数返回的对象将继续存在，保存其他状态的信息则需要另外的技术。

[1] 鸭子类型，其核心思想是，你的代码不必在意一个对象是不是一只鸭子，只需要关心它能像鸭子那样叫。不管是不是鸭子，只要能叫就行，同时鸭子叫的实现留给对象自己来完成。

Chapter 5

作用域

5.1 Python 作用域基础

在代码中变量名被赋值的位置决定了这个变量名能被访问到的范围。

由于变量名最初没有声明，Python 将一个变量名被赋值的地点关联为（绑定给）一个特定的命名空间。换句话说，在代码中给一个变量赋值的地方决定了这个变量将存在于哪个命名空间，也就是它可见的范围。

除打包代码之外，函数还为程序增加了一个额外的命名空间层：在默认的情况下，一个函数的所有变量名都是与函数的命名空间相关联的。这意味着：

- 一个在 def 内定义的变量名能够被 def 内的代码使用。不能在函数的外部引用这样的变量名。
- def 之中的变量名与 def 之外的变量名并不冲突，即使是使用在别处的相同的变量名。

分别对应3种不同的作用域：

- 如果一个变量在 def 内赋值，它被定位在这个函数之内。
- 如果一个变量在一个外层的 def 中赋值，对于内层的函数来说，它是非局部的。
- 如果在 def 之外赋值，它就是整个文件全局的。

5.1.1 作用域细节

从技术上讲，交互式命令行是一个名为 __main__ 的模块，它可以打印结果，但不会保存其代码；不过在其他方面，它都与模块文件的顶层相同。

函数定义了局部作用域，而模块定义的是全局作用域：

外围模块是全局作用域 每个模块都是一个全局作用域（也就是说，一个创建于模块文件顶层的变量的命名空间）。在模块被导入后，相对于外部世界而言全局变量就成为一个模块对象的属性，但是在一个模块中能够像简单的变量一样使用。

全局作用域的作用范围仅限于单个文件 别被这里的“全局”所迷惑，这里的全局指的是在一个文件的顶层的变量名仅对于这个文件内部的代码而言是全局的。在 Python 中是没有基于一个单个的、无所不包的情景文件的全局作用域的。替代这种方法的是，变量名由模块文件隔开，并

且必须精确地导入一个模块文件才能够使用这个文件中定义的变量名。当你在 Python 中听到“全局的”，你就应该想到“模块”。

赋值的变量名除非被声明为 global 或 nonlocal，否则均为局部变量。 在默认情况下，所有函数定义内的变量是位于局部作用域（与函数调用相关的）内的。如果您需要分配一个位于包含该函数的模块顶层的名称，您可以通过在函数内部的全局语句中声明它来实现。如果您需要分配一个位于封闭 def 中的名称，从 Python 3.X 开始，您可以通过在非局部语句中声明它来实现。

所有其他的变量名都可以归纳为局部、全局或者内置的。 在函数定义内部的尚未赋值的变量名是一个在一定范围内（在这个 def 内部）的局部变量、全局（在一个模块的命名空间内部）或者内置（由 Python 的预定义模块提供的）变量。我有点没看太明白。

每次对函数的调用都创建了一个新的局部作用域。 每次调用函数，都创建了一个新的局部作用域。也就是说，将会存在由那个函数创建的变量的命名空间。可以认为每一个 def 语句（以及 lambda 表达式）都定义了一个新的局部作用域，但是因为 Python 允许函数在循环中调用自身（一种叫做递归的高级技术），所以从技术上讲，局部作用域实际上对应的是函数的调用。换句话说，每一个函数调用都创建了一个新的局部命名空间。递归在处理不能提前预知的流程结构时是一个有用工具。

还要注意，一个函数内部的任何类型的赋值都会把一个名称划定为局部的。这包括 = 语句、import 中的模块名称、def 中的函数名称、函数参数名称等。如果在一个def 中以任何方式赋值一个名称，它都将对于该函数成为局部的。

此外，注意就地改变对象（in-place changes）并不会把变量划分为局部变量，实际上只有对变量名赋值才可以。例如，如果变量名 L 在模块的顶层被赋值为一个列表，在函数内部的像 L.append(X) 这样的语句并不会将 L 划分为局部变量，而 L = X 却可以。通常，记住名称和对象之间的清楚的区分是有帮助的：修改一个对象并不是对一个名称赋值。

5.1.2 变量名解析：LEGB 原则

对于一个 def 语句：

- 变量名引用分为三个作用域进行查找：首先是局部，之后是函数内（如果有的话），之后全局，最后是内置。
- 在默认情况下，变量名赋值会创建或者改变局部变量。
- 全局声明和非局部声明将赋值的变量名映射到模块文件内部的作用域。换句话说，所有在函数 def 语句（或者 lambda，我们稍后会学习的一个表达式）内赋值的变量名默认均为局部变量。函数能够在函数内部以及全局作用域(也就是物理上)直接使用变量名，但是必须声明为非局部变量和全局变量去改变其属性。

Python 的变量名解析机制有时称为 LEGB 法则，这也是由作用域的命令而来的。

- 当在函数中使用未认证的变量名时，Python 搜索 4 个作用域[本地作用域（L），之后是上一层结构中 def 或 lambda 的本地作用域（E），之后是全局作用域（G），最后是内置作用域（B）]并且在第一处能够找到这个变量名的地方停下来。如果变量名在这次搜索中没有找到，Python 会报错。变量名在使用前首先必须赋值过。

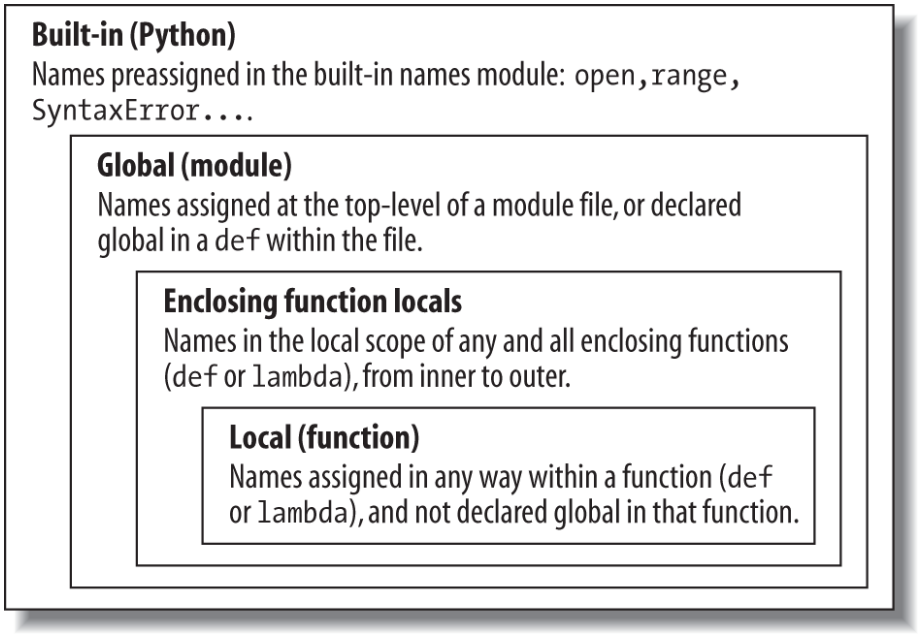

Figure 5.1: LEGB 作用域查找原则。当引用一个变量时，Python 按以下顺序依次进行查找：从局部变量中，在任意上层函数的作用域，在全局作用域，最后在内置作用域中查找。第一个能够完成查找的就算成功。变量在代码中被赋值的位置通常就决定了它的作用域。在 Python 3.X 中，nonlocal 声明也可以迫使名称映射到函数内部的作用域中，而不管是否对其赋值。

- 当在函数中给一个变量名赋值时（而不是在一个表达式中对其进行引用），Python 总是创建或改变本地作用域的变量名，除非它已经在那个函数中声明为全局变量。
- 当在函数之外给一个变量名赋值时（也就是，在一个模块文件的顶层，或者是在交互提示模式下），本地作用域与全局作用域（这个模块的命名空间）是相同的。

Figure 5.1 描述了 Python 的四个作用域的关系。注意到第二个 E 作用域的查找层次（上层 def 和 lambda 的作用域）从技术上来说可能不仅是一层查找的层次。当你在函数中嵌套函数时这个层次才需要考虑。

此外，记住这些规则仅对简单的变量名有效（例如，spam）。在 Part V 和 Part VI 中，我们将会看到被验证的属性变量名（例如，object.spam）会存在于特定的对象中，并遵循一种完全不同的查找规则，而不止我们这里提到的作用域的概念。属性引用（变量名跟着点号）搜索一个或多个对象，而不是作用域，并且有可能涉及所谓的“继承”的概念（将在 Part VI 讨论）。

5.1.3 内置作用域

实际上，内置作用域仅仅是一个名为 builtins 的内置模块，但是必须要导入 builtins 之后才能使用内置作用域，因为变量名 builtins 本身并没有预先内置。

3.X 内置作用域是通过一个名为 builtins 的标准库模块来实现的，但是这个变量名自身并没有放入内置作用域内，所以必须导入这个文件才能够使用它。

两种方法引用一个内置函数：通过 LEBG 法则带来的好处，或者手动导入 builtins 模块。

重定义内置名称：有好有坏

有时在高级编程中你可能真的想要在自己的代码中重定义内置名称来替换原有的名称。然而，重定义一个内置名称往往是个错误。并且让人头疼的是，Python 对于这个问题不会处理为警告信息。

5.2 global 语句

global 语句和它的 3.X 近亲 nonlocal 语句是 Python 中唯一看起来有些像声明语句的语句。但是，它并不是一个类型或大小的声明，它是一个命名空间的声明。它告诉 Python 函数打算生成一个或多个全局变量名。也就是说，存在于整个模块内部作用域（命名空间）的变量名。

- 全局变量是在外层模块文件的顶层被赋值的变量名。
- **全局变量如果是在函数内被赋值的话，必须经过声明。**
- **全局变量名在函数的内部不经过声明也可以被引用。**

换句话说，global 允许我们修改一个 def 之外的模块文件的顶层的名称。global 语句包含了关键字 global，其后跟着一个或多个由逗号分开的变量名。当在函数主体被赋值或引用时，所有列出来的变量名将被映射到整个模块的作用域内。

5.2.1 程序设计：最小化全局变量

一般而言，函数应该依赖形参与返回值而不是全局变量。

5.2.2 程序设计：最小化跨文件的修改

尽管能直接修改另一个文件中的变量，但是往往我们都不这样做。在文件间进行通信最好的办法就是通过调用函数，传递参数，然后得到其返回值。

5.3 作用域和嵌套函数

5.3.1 嵌套作用域的细节

在增加了嵌套的函数作用域后，变量的查找法则变得稍微复杂了一些。对于一个函数：

- 一个引用（X）首先在本地（函数内）作用域查找变量名 X；之后会在代码的语法上嵌套了的函数中的本地作用域，从内到外查找；之后查找当前的全局作用域（模块文件）；最后再内置作用域内。全局声明将会直接从全局（模块文件）作用域进行搜索。
- 在默认情况下，一个赋值（X = value）创建或改变了变量名 X 的当前作用域。如果 X 在函数内部声明为全局变量，它将会创建或改变变量名 X 为整个模块的作用域。另一方面，如果 X 在函数内声明为 nonlocal，赋值会修改最近的嵌套函数的本地作用域中的名称 X。

注意：全局声明将会将变量映射至整个模块。当嵌套函数存在时，嵌套函数中的变量也许仅仅是引用，但它们需要 nonlocal 声明才能修改。

5.3.2 工厂函数：闭包

根据要求的对象，这种行为有时也叫做闭合（closure）或者工厂函数（factory function）——一个能够记住嵌套作用域的变量值的函数，尽管那个作用域或许已经不存在了。

一个简单的函数工厂

工厂函数（也称为闭包）有时被需要动态生成事件处理程序以响应运行时条件的程序使用。例如，想象一个 GUI 必须根据用户输入定义操作，而这些操作在 GUI 构建时是无法预料的。在这种情况下，我们需要一个函数来创建并返回另一个函数，其中的信息可能因每个函数而异。

闭包 vs 类：回合 1

对于某些人来说，Part VI 中完整描述的类可能看起来更擅长像这样的状态保留，因为它们通过属性分配使它们的记忆更加明确。类还直接支持闭包函数不支持的附加工具，例如通过继承和运算符重载进行定制，并且更自然地以方法的形式实现多种行为。由于这些区别，类可能更擅长实现更完整的对象。

尽管如此，当保留状态是唯一目标时，闭包函数通常提供更轻量级且可行的替代方案。它们为单个嵌套函数所需的数据提供每次调用的本地化存储。当我们添加前面描述的 3.X 非局部语句以允许封闭作用域状态更改时尤其如此（在 2.X 中，封闭作用域是只读的，因此用途更有限）。

从更广泛的角度来看，Python 函数有多种方法可以在调用之间保留状态。虽然普通局部变量的值在函数返回时消失，但全局变量的值可以在调用过程中保留；在类实例属性中；在我们在这里遇到的封闭范围参考中；以及参数默认值和函数属性。有些可能也在此列表中包含可变的默认参数（尽管其他人可能希望他们不这样做）。

5.3.3 使用默认参数来保留嵌套作用域的状态

最好的处方就是简单地避免在 def 中嵌套 def，这会让程序更加得简单—在 Python 的世界观里，扁平通常胜于嵌套。

嵌套作用域，默认值参数和 lambda

lambda，简短地说，它就是一个表达式，将会生成后面调用的一个新的函数，与 def 语句很相似。由于它是一个表达式，尽管能够使用在 def 中不能使用的地方，例如，在一个列表或是字典常量之中。

像 def 一样，lambda 表达式引入了新的本地作用域。多亏了嵌套作用域查找层，lambda 能够看到所有在其所编写的函数中可用的变量。

由于 lambda 是表达式，所以它们自然而然地（或者更一般的）嵌套在了 def 中。因此，它们也就成为了后来在查找原则中增补嵌套函数作用域的最大受益者。在大多数情况下，给 lambda 函数通过默认参数传递值也就没有什么必要了。

循环变量可能需要默认值参数，而不是作用域

如果在函数中定义的 lambda 或者 def 嵌套在一个循环之中，而这个内嵌函数友引用了一个外层作用域的变量，该变量被循环所改变，那么所有在这个循环中产生的函数会有相同的值—也就是在最后一次循环中完成时被引用变量的值。

> 因为嵌套作用域中的变量在嵌套的函数被调用时才进行查找，所以它们实际上记住的是同样的值（在最后一次循环迭代中循环变量的值）。

这是在嵌套作用域的值和默认参数方面遗留的一种仍需要解释清楚的情况，而不是引用所在的嵌套作用域的值。也就是说，为了让这类代码能够工作，必须使用默认参数把当前的值传递给嵌套作用域的变量。因为默认参数是在嵌套函数创建时评估的（而不是在其稍后调用时）。

任意的作用域嵌套

我们要主要作用域可以做任意的嵌套，但是当名称被引用时指挥查找外层函数 def 语句（而不是类）。

5.4 Python 3.X 中的 nonlocal 语句

我们们介绍了嵌套函数可以引用一个嵌套的函数作用域中的变量的方法，即便这个函数已经返回了。事实上，在 Python 3.X 中，我们也可以修改这样的嵌套作用域变量，只要我们在一条 nonlocal 语句中声明它们。使用这条语句，嵌套的 def 可以对嵌套函数中的名称进行读取和写入访问。

nonlocal 语句是 global 的近亲，前面已经介绍过 global。nonlocal 和 global 一样，声明了将要在一个嵌套的作用域中修改的名称。和 global 的不同之处在于，nonlocal 应用于一个嵌套的函数的作用域中的一个名称，而不是所有 def 之外的全局模块作用域；而且在声明 nonlocal 名称的时候，它必须已经存在于该嵌套函数的作用域中——它们可能只存在于一个嵌套的函数中，并且不能由一个嵌套的 def 中的第一次赋值创建。

换句话说，nonlocal 即允许对嵌套的函数作用域中的名称赋值，并且把这样的名称的作用域查找限制在嵌套的 def。直接效果是更加直接和可靠地实现了可更改的作用域信息，对于那些不想要或不需要带有属性的类的程序而言。

5.4.1 nonlocal 基础

Python 3.X 引入了一条新的 nonlocal 语句，它只在一个函数内有意义：

```
def func():
    nonlocal name1, name2
```

nonlocal 语句除了允许修改外层 def 中的名称外，还会强制发起引用，这点很想 global 语句，nonlocal 是的对该语句中列出的名称的查找从外层的 def 作用域开始，而不是该函数本身的作用域，也就是说，nonlocal 意味着完全略过我的局部作用域。

实际上，当执行到 nonlocal 语句的时候，nonlocal 中列出的名称必须在一个嵌套的 def 中提前声明过，否则，将会产生一个错误。

当在一个函数中使用的时候，global 和 nonlocal 语句都在某种程度上限制了查找规则：

- global 使得作用域查找从嵌套的模块的作用域开始，并且允许对那里的名称赋值。如果名称不存在于该模块中，作用域查找继续到内置作用域，但是，对全局名称的赋值总是在模块的作用域中创建或修改它们。
- nonlocal 限制作用域查找只是嵌套的 def，要求名称已经存在于那里，并且允许对它们赋值。作用域查找不会继续到全局或内置作用域。

边界情况

有几件事情需要注意。首先，和 global 语句不同，当执行一条 nonlocal 语句时，nonlocal 名称必须已经在一个嵌套的 def 作用域中赋值过，否则将会得到一个错误——不能通过在嵌套的作用域中赋给它们一个新值来创建它们。事实上，nonlocal 名称在外层或内嵌函数被调用之前就已经在函数定义的时候被检查了。其次，nonlocal 限制作用域查找仅为嵌套的 def，nonlocal 不会在嵌套的模块的全局作用域或所有 def 之外的内置作用域中查找，即便已经有了这些作用域。

5.5 为什么使用 nonlocal？状态保持备选项

Chapter 6

Chapter 7

推导与生成

Part V

Part VI

Chapter 8

运算符重载