Contents

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。

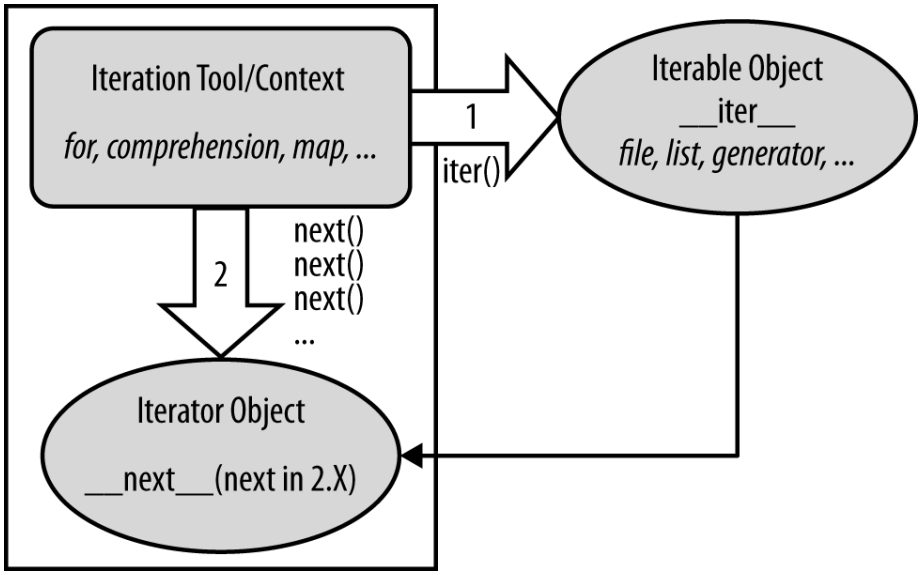

Figure 3.1: Python 迭代协议，由 for 循环、推导式、映射等使用，并受文件、列表、字典、Chapter 4 的生成器等支持。有些对象既是迭代上下文又是可迭代对象，例如生成器表达式和 3.X 风格的某些工具（例如 map 和 zip）。有些对象既是可迭代的又是迭代器，为 iter() 调用返回自身，这就是一个无操作。

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 4 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 5 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 4 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 4 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 4 学习列表解析和迭代器的更多内容，在 Chapter 5 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。

Chapter 4

推导与生成

Chapter 5

运算符重载