学习笔记：流畅的 Python

Stephen CUI

October 10, 2023

Chapter 1

1.1 Using * to Grab Excess Items

Chapter 2

Unicode 文本和字节序列

2.1 字节概要

bytes 或 bytearray 对象的各个元素是介于 0~255（含）之间的整数。然而，二进制序列的切片始终是同一类型的二进制序列，包括长度为 1 的切片。

2.2 处理解码和编码问题

虽然有个一般性的 `UnicodeError` 异常，但是报告错误时几乎都会指明具体的异常：`UnicodeEncodeError`（把字符串转换成二进制序列时）或 `UnicodeDecodeError`（把二进制序列转换成字符串时）。如果源码的编码与预期不符，加载 Python 模块时还可能抛出 `SyntaxError`。

2.2.1 处理 UnicodeEncodeError

多数非 UTF 编解码器只能处理 Unicode 字符的一小部分子集。把文本转换成字节序列时，如果目标编码中没有定义某个字符，那就会抛出 UnicodeEncodeError 异常，除非把 errors 参数传给编码方法或函数，对错误进行特殊处理。

ASCII is a common subset to all the encodings that I know about, therefore encoding should always work if the text is made exclusively of ASCII characters. Python 3.7 added a new boolean method `str.isascii()` to check whether your Unicode text is 100% pure ASCII. If it is, you should be able to encode it to bytes in any encoding without raising UnicodeEncodeError.

2.2.2 处理UnicodeDecodeError

不是每一个字节都包含有效的 ASCII 字符，也不是每一个字符序列都是有效的 UTF-8 或 UTF-16。因此，把二进制序列转换成文本时，如果假设是这两个编码中的一个，遇到无法转换的字节序列时会抛出 UnicodeDecodeError。

2.3 Unicode 文本排序

Python 比较任何类型的序列时，会一一比较序列里的各个元素。对字符串来说，比较的是码位。可是在比较非 ASCII 字符时，得到的结果不尽如人意。

在 Python 中，非 ASCII 文本的标准排序方式是使用 locale.strxfrm 函数，根据 locale 模块的文档，这个函数会“把字符串转换成适合所在区域进行比较的形式”。使用 locale.strxfrm 函数之前，必须先为应用设定合适的区域设置，还要祈祷操作系统支持这项设置。

2.3.1

2.4 Unicode 数据库

2.4.1 字符的数值意义

Chapter 3

函数是一等对象

在 Python 中，函数是一等对象。编程语言理论家把“一等对象”定义为满足下述条件的程序实体：

- 在运行时创建
- 能赋值给变量或数据结构中的元素
- 能作为参数传给函数
- 能作为函数的返回结果

3.1 把函数视作对象

3.2 高阶函数

接受函数为参数，或者把函数作为结果返回的函数是**高阶函数**（higher-order function）。

3.3 匿名函数

lambda 关键字在 Python 表达式内创建匿名函数。

然而，Python 简单的句法限制了 lambda 函数的定义体只能使用纯表达式。换句话说，lambda 函数的定义体中不能赋值，也不能使用 while 和 try 等 Python 语句。可以有新出现的 := 赋值表达式。But if you need it, your lambda is probably too complicated and hard to read, and it should be refactored into a regular function using def.

3.4 可调用对象

除了用户定义的函数，调用运算符（即 ()）还可以应用到其他对象上。如果想判断对象能否调用，可以使用内置的 callable() 函数。Python 数据模型文档列出了 9 种可调用对象。

用户定义的函数 使用 def 语句或 lambda 表达式创建。
内置函数 使用 C 语言（CPython）实现的函数，如 len 或 time.strftime。
内置方法 使用 C 语言实现的方法，如 dict.get。
方法 在类的定义体中定义的函数。

类 调用类时会运行类的 `__new__` 方法创建一个实例，然后运行 `__init__` 方法，初始化实例，最后把实例返回给调用方。因为 Python 没有 new 运算符，所以调用类相当于调用函数。（通常，调用类会创建那个类的实例，不过覆盖 `__new__` 方法的话，也可能出现其他行为。）

类的实例 如果类定义了 `__call__` 方法，那么它的实例可以作为函数调用。

生成器函数 使用 yield 关键字的函数或方法。调用生成器函数返回的是生成器对象。

3.5 用户定义的可调用类型

不仅 Python 函数是真正的对象，任何 Python 对象都可以表现得像函数。为此，只需实现实例方法 `__call__`。

3.6 支持函数式编程的包

3.6.1 operator 模块

3.6.2 使用 functools.partial 冻结参数

partial 它可以根据提供的一个可调用对象产生一个新的可调用对象，为原可调用对象的某些参数绑定预定的值。使用这个函数可以把接受一个或多个参数的函数改造成需要更少参数的回调的 API。

Chapter 4

函数中的类型提示

4.1 关于渐进式类型

A gradual type system:

Is optional By default, the type checker should not emit warnings for code that has no type hints. Instead, the type checker assumes the Any type when it cannot determine the type of an object. The Any type is considered compatible with all other types.

Does not catch type errors at runtime Type hints are used by static type checkers, linters, and IDEs to raise warnings. They do not prevent inconsistent values from being passed to functions or assigned to variables at runtime.

Does not enhance performance Type annotations provide data that could, in theory, allow optimizations in the generated bytecode, but such optimizations are not implemented in any Python runtime that I am aware in of July 2021.

4.2 类型由受支持的操作定义

In a gradual type system, we have the interplay of two different views of types:

Duck typing The view adopted by Smalltalk—the pioneering object-oriented language—as well as Python, JavaScript, and Ruby. Objects have types, but variables (including parameters) are untyped. In practice, it doesn't matter what the declared type of the object is, only what operations it actually supports. If I can invoke birdie.quack(), then birdie is a duck in this context. By definition, duck typing is only enforced at runtime, when operations on objects are attempted. This is more flexible than nominal typing, at the cost of allowing more errors at runtime.

Nominal typing The view adopted by C++, Java, and C#, supported by annotated Python. Objects and variables have types. But objects only exist at runtime, and the type checker only cares about the source code where variables (including parameters) are annotated with type hints. If Duck is a subclass of Bird, you can assign a Duck instance to a parameter annotated as birdie: Bird. But in the body of the function, the type checker considers the call birdie.quack() illegal, because birdie is nominally a Bird,

and that class does not provide the .quack() method. It doesn't matter if the actual argument at runtime is a Duck, because nominal typing is enforced statically. The type checker doesn't run any part of the program, it only reads the source code. This is more rigid than duck typing, with the advantage of catching some bugs earlier in a build pipeline, or even as the code is typed in an IDE.

鸭子类型更容易上手，也更灵活，但是无法主治不受支持的操作在运行时导致错误。名义类型在运行代码之前检测错误，但有时会拒绝实际能运行的代码。（运行实例）

Chapter 5

迭代器、生成器与经典协程

5.1 序列可以迭代的原因：iter 函数

解释器需要迭代对象 x 时，会自动调用 iter(x)。内置的 iter 函数执行以下操作：

1. 检查对象是否实现了 `__iter__` 方法，如果实现了就调用它，获取一个迭代器。
2. 如果没有实现 `__iter__` 方法，但是实现了 `__getitem__` 方法，Python 会创建一个迭代器，尝试按顺序（从索引 0 开始）获取元素。
3. 如果尝试失败，Python 抛出 TypeError 异常，通常会提示“C object is not iterable”（C对象不可迭代），其中 C 是目标对象所属的类。

这是鸭子类型（duck typing）的极端形式：不仅实现了特殊方法的 `__iter__` 的对象被视作可迭代对象，实现了 `__getitem__` 方法的对象也被视作可迭代对象。

If a class provides `__getitem__`, the iter() built-in accepts an instance of that class as iterable and builds an iterator from the instance. Python’s iteration machinery will call `__getitem__` with indexes starting from 0, and will take an IndexError as a signal that there are no more items. Although `__getitem__` could provide items, it is not recognized as such by an isinstance against abc.Iterable.

从 Python 3.10 开始，检查对象 x 能否迭代，最准确的方法是调用 iter() 函数，如果不可以迭代，则处理 TypeError 异常。This is more accurate than using isinstance(x,abc.Iterable), because iter(x) also considers the legacy `__getitem__` method, while the Iterable ABC does not.

5.2 可迭代对象与迭代器

使用 iter 内置函数可以获取迭代器的对象。如果对象实现了能返回迭代器的 `__iter__` 方法，那么对象就是可迭代的。序列都可以迭代；实现了 `__getitem__` 方法，而且其参数是从 0 开始的索引，这种对象也可以迭代。

我们要明确可迭代的对象和迭代器之间的关系：**Python 从可迭代的对象中获取迭代器**。

Python 标准的迭代器接口有以下两个方法：

1. `__next__` 返回序列中的下一项，如果没有项了，则抛出 StopIteration 。
2. `__iter__` 放回 self，以便在预期内可迭代对象的地方使用迭代器。

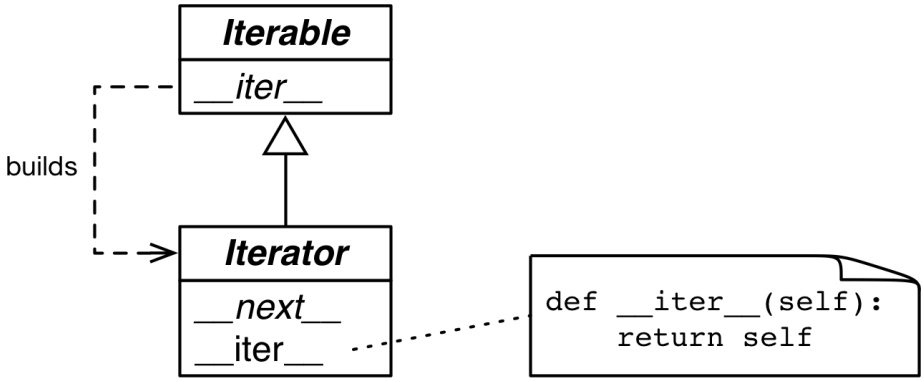

Figure 5.1: Iterable 和 Iterator 抽象基类。以斜体显示的是抽象方法。具体的 `Iterable.__iter__` 方法应该返回一个 Iterator 实例。具体的 Iterator 类必须实现 `__next__` 方法。`Iterator.__iter__` 方法直接返回实例本身

5.2.1 不要把可迭代对象变成迭代器

构建可迭代的对象和迭代器时经常会出现错误，原因是混淆了二者。要知道，可迭代的对象有个 `__iter__` 方法，每次都实例化一个新的迭代器；而迭代器要实现 `__next__` 方法，返回单个元素，此外还要实现 `__iter__` 方法，返回迭代器本身。

因此，迭代器可以迭代，但是可迭代的对象不是迭代器。

5.2.2 生成器的工作原理

只要 Python 函数的定义体中有 yield 关键字，该函数就是生成器函数。调用生成器函数时，会返回一个生成器对象。也就是说，生成器函数是生成器工厂。

生成器函数会创建一个生成器对象，包装生成器函数的定义体。把生成器传给 next(...) 函数时，生成器函数会向前，执行函数定义体中的下一个 yield 语句，返回产出的值，并在函数定义体的当前位置暂停。最终，函数的定义体返回时，外层的生成器对象会抛出 StopIteration 异常——这一点与迭代器协议一致。

应该这样说：函数返回值；调用生成器函数返回生成器；生成器产出或生成值。生成器不会以常规的方式“返回”值：生成器函数定义体中的 return 语句会触发生成器对象抛出 StopIteration 异常。如果生成器中由 return x 语句，则调用方能从 StopIteration 异常中获取 x 的值，但是我们往往把这个操作交给 yield from 句法。

5.3 惰性实现版本

5.4 何时使用生成器表达式

如果生成器表达式要分成多行编写，那么倾向于定义生成器函数，以便提高可读性

对比迭代器和生成器

迭代器 泛指实现了 `__verb__` 方法的对象。迭代器用于生成供客户代码使用的数据，即客户代码通过 for 循环或其他迭代方式，或者直接在迭代器上调用 next(it) 驱动迭代器。不过显示调用 next() 并不常见。实际上，我们在 Python 中使用的迭代器多数都是生成器。

生成器 由 Python 编译器构建的迭代器。为了创建生成器，我们不实现 `__next__` 方法，而是使用 yield 关键字得到生成器函数（创建生成器对象的工厂）。生成器表达式时构建生成器对象的另一种方式。生成器对象提供了 `__next__` 方法，因此生成器对象是迭代器，Python3.5 之后，还可以使用 async def 声明异步生成器。

Python 术语表最近添加了术语“生成器迭代器”（generator iterator），指代由生成器函数构建的生成器对象。根据“生成器表达式”词条，生成器表达式返回“迭代器”。