学习笔记：流畅的 Python

Stephen CUI

October 9, 2023

Chapter 1

函数中的类型提示

1.1 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.

1.2 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 2

迭代器、生成器与经典协程

2.1 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.

2.2 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。

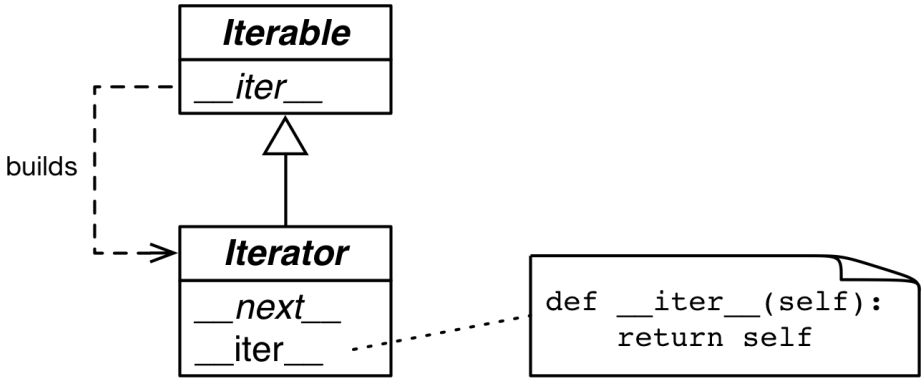

Figure 2.1: Iterable 和 Iterator 抽象基类。以斜体显示的是抽象方法。具体的 `Iterable.__iter__` 方法应该返回一个 Iterator 实例。具体的 Iterator 类必须实现 `__next__` 方法。`Iterator.__iter__` 方法直接返回实例本身

2.2.1 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。

2.2.2 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。

我觉得，使用准确的词语描述从生成器中获取结果的过程，有助于理解生成器。注意，我说的是产出或生成值。如果说生成器“返回”值，就会让人难以理解。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。