4.2 隐式序列

::: details INFO
译者：DuKle3

来源：4.2Implicit Sequences

对应：HW04、Disc05、Lab07
:::

一个序列（sequence）不一定要把每个元素显式存储在计算机的内存中。换句话说，我们可以建立一个对象（object），它提供对某个序列的访问，而无需事先计算每个元素的值。
取而代之，我们只在有需要的时候才计算元素。

这个想法的一个例子为在第 2 章所介绍的 `range` 容器（container）类型，`range` 表示连续、有范围的整数序列，然而，该序列的每个元素并不是显式储存在内存中的。
相反，当从 `range` 访问某个元素时，才会进行计算来求得其值。因此，我们可以在不需要很多内存来表示非常大范围的整数。只有端点会被储存为 `range` 对象的一部份。

```
>>> r = range(10000,1000000000)
>>> r[45006230]
45016230
```

在这个例子中，当创建 `range` 实例时，并没有存储此范围内的 999,990,000 整数。反之，`range` 对像将第一个元素 10,000 添加到索引 45,006,230 来得出元素 45,016,230。按需求来计算值，而不是从现有的表示中去检索他们，
这是惰性计算（Lazy computation）的一个范例。在计算机科学中，惰性计算指任何延迟计算，直到需要该值的程序。

> 译者注：惰性计算是这一整篇的重点

4.2.1 迭代器

Python 和许多其他编程语言都提供了一种统一的方法来按照顺序地处理容器内的元素，称为迭代器（iterators）迭代器是一种对象，提供对值逐一顺序访问的功能。
迭代器抽象有两个组件：

- 检索下一个元素的机制
- 到达序列末尾并且没有剩余元素，发出信号的机制

对于任何容器，例如 `list` 或 `range`，都可以通过调用内置的 `iter` 函数来获取迭代器。使用内置的 `next` 函数来访问迭代器的内容。

```
>>> primes = [2, 3, 5, 7]
>>> type(primes)
<class 'list'>
>>> iterator = iter(primes)
>>> type(iterator)
<class 'list-iterator'>
>>> next(iterator)
2
>>> next(iterator)
3
>>> next(iterator)
5
```

在 Python 中表示没有更多可用值的方式是在调用 `next` 时引发 `StopIteration` 异常。可以使用 `try` 语句来处理此错误。

```
>>> next(iterator)
7
>>> next(iterator)
Traceback (most recent call las):
  File "<stdin>", line 1, in <module>
StopIteration

>>> try:
        next(iterator)
    except StopIteration:
        print('No more values')
No more values
```

迭代器保持本地状态（local state）来表示其在序列中的位置。每次调用 `next` 时，该位置都会前进。两个单独的迭代器可以跟踪同一序列中的两个不同位置。但是，两个变量如果指向同一迭代器，它们会共享相同的值。

```
>>> r = range(3, 13)
>>> s = iter(r)  # r 的第一个迭代器
>>> next(s)
3
>>> next(s)
4
>>> t = iter(r)  # r 的第二个迭代器
>>> next(t)
3
>>> next(t)
4
>>> u = t        # u 绑定到 r 的第二个迭代器
>>> next(u)
5
>>> next(u)
6
```

推进第二个迭代器不会影响第一个迭代器。由于第一个迭代器最后返回的值是 4，下一次返回 5。另一方面，第二个迭代器的位置会下一次返回 7。

```
>>> next(s)
5
>>> next(t)
7
```

在迭代器上调用 `iter` 将返回该迭代器，而不是其副本。
Python 中包含此行为，以便程序员可以对某个值调用 `iter` 来获取迭代器，而不必担心它是迭代器还是容器。

> 译者注：和 `list` 不同！

```
>>> v = iter(t)  # v 绑定到 r 的第二个迭代器
>>> next(v)      # u, v, t 都为 r 的第二个迭代器
8
>>> next(u)
9
>>> next(t)
10
```

迭代器的有用性源于迭代器的基础数据序列可能不会在内存中以显式方式表示。

迭代器提供了一种机制，可以逐个考虑一系列的值，但不必同时存储所有这些元素。

相反，当从迭代器请求下一个元素时，该元素可以按需求计算，而不是从现有的内存中检索。

范围（range）能够懒惰地计算序列的元素，因为所代表的序列是统一的，并且任何元素都可以从范围的起始和结束端点轻松计算得出。
迭代器允许对更广泛的基础连续数据集进行懒惰生成，因为它们不需要提供对基础系列的任意元素的访问。

相反，迭代器只需要按顺序计算系列的下一个元素，每次请求另一个元素时进行计算。
虽然不如访问序列的任意元素（称为随机访问）灵活，但对于数据处理应用程序来说，对连续数据的顺序访问通常是足够的。

4.2.2 可迭代性

任何可以产生迭代器的值都称为可迭代值（iterable value）。在 Python 中，可迭代值是任何可以传递给内置 `iter` 函数的值。

可迭代对象包括：

- **序列值**：例如字符串（string）和元组（tuples）
- **容器**：例如集合（sets）和字典（dictionaries）

迭代器也是可迭代的，因为它们可以传递给 `iter` 函数。

即使是无序集合（例如字典），在生成迭代器时也必须定义其内容的顺序。
字典和集合是无序的，因为程序员无法控制迭代的顺序，但 Python 确实在其规范中保证了有关其顺序的某些属性。

> 译者注：现在 Python 3.6+ 版本，字典的顺序是键值对（key-value pair）加入字典时的顺序；在 Python 3.5 和之前的版本，字典是无顺序的

```
>>> d = {'one': 1, 'two': 2, 'three': 3}
>>> d
{'one': 1, 'three': 3, 'two': 2}
>>> k = iter(d)
>>> next(k)
'one'
>>> next(k)
'three'
>>> v = iter(d.values())
>>> next(v)
1
>>> next(v)
3
```

如果字典由于添加或删除键而导致其结构发生变化，则所有迭代器都会失效，
并且未来的迭代器可能会对其内容顺序进行任意更改。另一方面，更改现有键的值不会更改内容的顺序或使迭代器无效。

4.2.3 内置迭代器

有几个内置函数将可迭代值作为参数，并返回迭代器。这些函数广泛用于惰性序列处理

`map` 函数是惰性的：调用它时并不会执行计算，直到返回的迭代器被 `next` 调用

相反，会创建一个迭代器对象，如果使用 `next` 查询，
该迭代器对象可以返回结果。我们可以在下面的示例中观察到这一事实，其中对 `print` 的调用被延迟，直到从 `doubled` 迭代器请求相应的元素为止。

```
>>> def double_and_print(x):
        print('***', x, '=>', 2*x, '***')
        return 2*x
>>> s = range(3, 7)
>>> doubled = map(double_and_print, s)  # double_and_print 未被调用
>>> next(doubled)                       # double_and_print 调用一次
*** 3 => 6 ***
6
>>> next(doubled)                       # double_and_print 再次调用
*** 4 => 8 ***
8
>>> list(doubled)                       # double_and_print 再次调用两次
*** 5 => 10 ***                         # list() 会把剩余的值都计算出来并生成一个列表
*** 6 => 12 ***
[10, 12]
```

`filter` 函数返回一个迭代器， `zip` 和 `reversed` 函数也返回迭代器。

4.2.4 For 语句

Python 中的 `for` 语句是对迭代器进行操作。
如果对象具有返回迭代器的 `__iter__` 方法（method），则表示对象是可迭代的。

可迭代对象可以是 `for` 语句标题中 `<expression>` 的值：

```
for <name> in <expression>:
    <suite>
```

执行 `for` 语句，Python 会评估标头（header）`<expression>`，它必须为可迭代的值。然后，对该值调用 `__iter__` 方法。

在触发 `StopIteration` 异常之前，Python 会重复调用该迭代器上的 `__next__` 方法，并将结果绑定到 `for` 语句中的 `<name>`。然后，执行 `<suite>`。

```
>>> counts = [1, 2, 3]
>>> for item in counts:
        print(item)
1
2
3
```

在上面的例子中， `counts` 列表在对其调用 `__iter__()` 方法之后返回一个迭代器。 `for` 语句接着重复地调用迭代器的 `__next__()` 方法，并将其值和 `item` 绑定。
这个过程重复至迭代器触发 `StopIteration` ，此时 for 语句的执行结束。

有了上述迭代器的知识，现在我们就可以用 `while` 、赋值、 `try` 语句来实现 `for` 语句的执行规则。

```
>>> items = counts.__iter__()
>>> try:
        while True:
            item = items.__next__()
            print(item)
    except StopIteration:
        pass
1
2
3
```

以上例子，通过调用 `counts` 的 `__iter__` 方法返回的迭代器绑定到一个名称项（`items`），
以便可以依次查询每个元素。 `StopIteration` 异常的处理子句不执行任何操作，但处理异常提供了退出 while 循环的控制机制。

要在 `for` 循环中使用迭代器，迭代器还必须具有 `__iter__` 方法。

迭代器类型 Python 文档 的部分，建议迭代器有一个返回迭代器本身的 `__iter__` 方法，这样所有的迭代器都是可迭代的。

4.2.5 生成器和 Yield 语句

上面的 Letters 和 Positives（字母和正整数）对象要求我们引入一个新的字段（field）`self.current` 到我们的对象中，以跟踪通过序列的进度。
对于像上面所示的简单序列，这可以很容易地完成。然而，对于复杂的序列来说， `__next__` 方法在计算中保存其位置可能相当困难。
生成器（Generators）使我们能够通过利用 Python 解析器（Interpreter）的功能来定义更复杂的迭代。

生成器是由一种特殊类型的函数 **生成器函数** 返回的迭代器。
生成器函数与常规函数不同之处在于，它们在其主体内不包含 `return` 语句，而是使用 `yield` 语句来返回一系列元素。

生成器不使用对象的属性来跟踪它们在序列中的进度。
相反，它们控制生成器函数的执行，在每次调用生成器的 `__next__` 方法时执行，直到下一个 `yield` 语句被执行为止。使用生成器函数可以更简洁地实现 `Letters` 迭代器。

```
>>> def letters_generator():
        current = 'a'
        while current <= 'd':
            yield current
            current = chr(ord(current) + 1)

>>> for letter in letters_generator():
        print(letter)
a
b
c
d
```

即使我们从未明确定义过 `__iter__` 或 `__next__` 方法， `yield` 语句表明我们正在定义一个生成器函数。
当调用时，生成器函数不返回特定的返回值，而是返回一个生成器（一种迭代器类型），该生成器本身可以返回所产出（yields）的值。

生成器对象具有 `__iter__` 和 `__next__` 方法，每次调用 `__next__` 方法都会从之前离开的地方继续执行生成器函数，直到另一个 `yield` 语句被执行为止。

当第一次调用 `__next__` 时，程序会执行 `letters_generator` 函数的语句，直到遇到 `yield` 语句。然后，它暂停并返回 `current` 的值。
`yield` 语句不会销毁新建的环境，而是保留它供以后使用。

当再次调用 `__next__` 时，执行会从上次离开的地方继续。 `current` 的值以及 `letters_generator` 作用域内的任何其他绑定名称的值在多次调用 `__next__` 之下都会保留。

- 我们可以通过手动调用 `__next__()` 来遍历生成器：

```
>>> letters = letters_generator()
>>> type(letters)
<class 'generator'>
>>> letters.__next__()
'a'
>>> letters.__next__()
'b'
>>> letters.__next__()
'c'
>>> letters.__next__()
'd'
>>> letters.__next__()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

生成器在第一次调用 `__next__` 之前，不会执行任何在生成器函数内的语句。当生成器函数执行完毕返回时会触发 `StopIteration` 异常。

4.2.6 可迭代接口

如果对一个对象调用其 `__iter__` 方法会返回一个迭代器，则称为可迭代的（iterable），
可迭代值表示数据的集合，它们提供了一种可能产生多个迭代器的固定表示。

例如，下面的 `Letters` 类的实例表示连续字母的序列。每次调用 `__iter__` 方法时，都会构造一个新的 `LetterIter` 实例，该实例允许顺序访问序列的内容。

> 译者注：这里没有 `LetterIter` 的实现，就把他当作可迭代的值。

```
>>> class Letters:
        def __init__(self, start='a', end='e'):
            self.start = start
            self.end = end
        def __iter__(self):
            return LetterIter(self.start, self.end)
```

内置 `iter` 函数在其参数上调用 `__iter__` 方法。在下面的表达式序列中，从同一可迭代序列所衍生的两个迭代器独立地按顺序产生字母。

```
>>> b_to_k = Letters('b', 'k')
>>> first_iterator = b_to_k.__iter__()
>>> next(first_iterator)
'b'
>>> next(first_iterator)
'c'
>>> second_iterator = iter(b_to_k)
>>> second_iterator.__next__()
'b'
>>> first_iterator.__next__()
'd'
>>> first_iterator.__next__()
'e'
>>> second_iterator.__next__()
'c'
>>> second_iterator.__next__()
'd'
```

- `Letters` 实例 `b_to_k`
- `LetterIter` 迭代器实例 `first_iterator` 和 `secondary_iterator`

他们的不同之处在于，`Letters` 实例不会更改，而迭代器实例会随着每次调用 `next`（或等效地，每次调用 `__next__`）而更改。迭代器通过顺序数据跟踪进度，而可迭代则代表数据本身。

Python 中的许多内置函数都采用可迭代参数并返回迭代器。
例如，前文所提到过的 `map` 函数接受一个函数和一个可迭代对象。它返回一个迭代器，该迭代器将函数参数应用于可迭代参数中的每个元素的结果。

```
>>> caps = map(lambda x: x.upper(), b_to_k)
>>> next(caps)
'B'
>>> next(caps)
'C'
```

4.2.7 使用 Yield 创建可迭代对象

在 Python 中，迭代器只对底层数据系列进行一次遍历。在遍历一次之后，当调用 `__next__()` 时，迭代器将引发 `StopIteration` 异常。许多应用程序需要多次对元素进行迭代。例如，我们必须多次遍历列表，以列举所有元素对。

```
>>> def all_pairs(s):
        for item1 in s:
            for item2 in s:
                yield (item1, item2)
>>> list(all_pairs([1, 2, 3]))
[(1, 1), (1, 2), (1, 3), (2, 1), (2, 2), (2, 3), (3, 1), (3, 2), ( 3, 3)]
```

序列本身并不是迭代器，而是可迭代的对象。在 Python 中，可迭代的接口由一个单独的方法 `__iter__` 组成，该方法返回一个迭代器。
Python 中的内置序列类型在调用它们的 `__iter__` 方法时会返回新的迭代器实例。如果可迭代对象每次调用 `__iter__` 时返回一个新的迭代器实例，那么它可以被多次迭代。

可以通过实现可迭代接口来定义新的可迭代类。例如，下面的可迭代 `LettersWithYield` 类，当每次调用 `__iter__` 时，都会返回一个新的字母迭代器。

```
>>> class LettersWithYield:
        def __init__(self, start='a', end='e'):
            self.start = start
            self.end = end
        def __iter__(self):
            next_letter = self.start
            while next_letter < self.end:
                yield next_letter
                next_letter = chr(ord(next_letter) + 1)
```

`__iter__` 方法是一个生成器函数；它返回一个生成器对象，该对像生成字母“a”到“d”，然后停止。每次我们调用此方法时，新的生成器都会开始重新遍历顺序数据。

```
>>> letters = LettersWithYield()
>>> list(all_pairs(letters))[:5]
[('a', 'a'), ('a', 'b'), ('a', 'c'), ('a', 'd'), ('b', 'a') ]
```

> 译者注：在 Python 3.3 中，加入了 `yield from` 语句，可以在 Lecture 或是 Python 3.3 更新文档 取得更多相关资讯。

4.2.8 迭代器接口

Python 的迭代器接口是使用一个名为 `__next__` 的方法来定义的，该方法返回某个底层连续序列的下一个元素。
在调用 `__next__` 时，迭代器可以执行检索或计算下一个元素。对 `__next__` 的调用会对迭代器进行变异性（mutating）更改：它们会推进迭代器的位置。
因此，对 `__next__` 的多次调用会返回底层序列的连续元素。当在调用 `__next__` 时引发 `StopIteration` 异常，表示已经达到了序列的末端。

下面的 `LetterIter` 类，可以从 `start` 字符迭代至 `end` （不包括 `end` ）。

其中， `self.next_letter` 储存下一个字符，在调用 `__next__` 时会返回并计算新的 `next_letter`。

```
>>> class LetterIter:
        """依照 ASCII 码值顺序迭代字符的迭代器。"""
        def __init__(self, start='a', end='e'):
            self.next_letter = start
            self.end = end
        def __next__(self):
            if self.next_letter == self.end:
                raise StopIteration
            letter = self.next_letter
            self.next_letter = chr(ord(letter)+1)
            return letter
```

我们可以利用这个 `LetterIter` 类，以下面两种方式来获取在序列中的字符。

- `__next__` 方法
- `next()` 内置函数（其实也是调用 `__next__` ）

```
>>> letter_iter = LetterIter()
>>> letter_iter.__next__()
'a'
>>> letter_iter.__next__()
'b'
>>> next(letter_iter)
'c'
>>> letter_iter.__next__()
'd'
>>> letter_iter.__next__()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 12, in next
StopIteration
```

迭代器是可变异的：它们在前进时纪录序列中的位置。当遍历至末端，这个迭代器就用完了。
例如： `LetterIter` 实例只能迭代一次。直到 `__next__()` 方法触发 `StopIteration` 异常。

通常，迭代器不会被重置；取而代之，会创建一个新实例来开始新的迭代。

迭代器也允许我们通过实现永远不会触发 `StopIteration` 异常的 `__next__` 方法来表示无限级数。
例如，下面的 `Positives` 类迭代无限正整数。

```
>>> class Positives:
```

```
        def __init__(self):
            self.next_positive = 1;
        def __next__(self):
            result = self.next_positive
            self.next_positive += 1
            return result
>>> p = Positives()
>>> next(p)
1
>>> next(p)
2
>>> next(p)
3
```

4.2.9 Streams

TODO

> 译者注：原文未完成

4.2.10 Python 流

流（Streams）提供了另一种隐式表示连续数据的方法。 `Stream` 是一个惰性计算的链表（linked-list）。
类似于第 2 章中的 `Link` 类， `Stream` 实例会对其第一个元素和其余（rest）部分的请求做出响应。就像 Link 一样， `Stream` 的其余部分本身也是一个 `Stream` 。但与 `Link` 不同的是，
`Stream` 的其余部分仅在查找时计算，而不是提前存储。也就是说， `Stream` 的其余部分是惰性计算的。

为了实现这种惰性评估， `Stream` 存储了一个计算其余部分的函数。
每次调用该函数时，其返回值都会被缓存为 `Stream` 的一部分，存储在一个名为 `_rest` 的属性中，该属性以底线命名，表示其不应直接访问。

可访问的 `rest` 属性是一个属性方法，它会返回流的其余部分。通过这种设计， `Stream` 存储了如何计算其余部分的方法，
而不是直接存储实际的数据。

```
>>> class Stream:
        """惰性计算的链表"""
        class empty:
            def __repr__(self):
                return 'Stream.empty'
        empty = empty()
        def __init__(self, first, compute_rest=lambda: empty):
            assert callable(compute_rest), 'compute_rest 必须为可调用'
            self.first = first
            self._compute_rest = compute_rest
        @property
        def rest(self):
            """返回 Stream 的其他部分（缓存部分），如果需要计算，则计算"""
            if self._compute_rest is not None:
                self._rest = self._compute_rest()
                self._compute_rest = None
            return self._rest
```

```
        def __repr__(self):
            return 'Stream({0}, <...>)'.format(repr(self.first))
```

链表是使用嵌套表达式定义的。例如，我们可以创建一个代表元素 1 然后 5 的 `Link`，如下所示：

```
>>> r = Link(1, Link(2 + 3, Link(9)))
```

同样，我们可以创建一个代表同一系列的 `Stream` 。在请求 `Stream` 的其余部分之前， `Stream` 实际上不会计算第二个元素 5。我们通过创建 `lambda` 函数来实现这种效果。

```
s = Stream(1, lambda: Stream(2 + 3, lambda: Stream(9)))
```

这里，1 是 `Stream` 的第一个元素，后面的 `lambda` 表达式返回一个用于计算 `Stream` 的其余部分的函数。

访问链表 r 和 `Stream` s 的元素的过程类似。然而，5 是存储在 r 中，而在 s 中，它是在第一次请求时根据需求通过加法计算的。

```
>>> r.first
1
>>> s.first
1
>>> r.rest.first
5
>>> s.rest.first
5
>>> r.rest
Link(5, Link(9))
>>> s.rest
Stream(5, <...>)
```

r 的剩余部分是一个两元素链表，而 s 的剩余部分包含一个计算剩余部分的函数；它将返回空流（empty stream）的可能性尚未被发现。

当创建 `Stream` 实例时，字段 `self._rest` 为 `None`，表示 `Stream` 的其余部分尚未计算。

当通过点表达式请求 `rest` 属性时，将调用 `rest` 属性方法，该方法会触发 `self._rest = self._compute_rest()` 的计算。
由于 `Stream` 中的缓存机制， `compute_rest` 函数仅被调用一次，然后被丢弃。

`compute_rest` 函数的基本属性是它不带参数，并且返回 `Stream` 或 `Stream.empty`。

惰性评估为我们提供了使用 `Stream` 表示无限连续数据集的能力。例如，我们可以表示从任意起始值开始的递增整数序列。

```
>>> def integer_stream(first):
        def compute_rest():
            return integer_stream(first+1)
        return Stream(first, compute_rest)
>>> positives = integer_stream(1)
>>> positives
Stream(1, <...>)
>>> positives.first
1
```

当第一次调用 `integer_stream` 时，它返回一个 `Stream` ，其第一个是序列中的第一个整数。然而，`integer_stream` 实际上是递归的，
因为该 `Stream` 的 `compute_rest` 再次调用 `integer_stream` ，并带有递增的参数。
我们说 `integer_stream` 是惰性的，因为只有在请求 `integer_stream` 的 `rest` 部分时才会对 `integer_stream` 进行递归调用。

```
>>> positives.first
1
>>> positives.rest.first
2
>>> positives.rest.rest
Stream(3, <...>)
```

同样身为高阶函数并操作序列的函数 `map` 和 `filter` ，也适用于 `Stream` ，尽管它们的实现必须更改以延迟（惰性）应用其参数函数。

`map_stream` 映射一个函数到一个 `Stream` 上，这会产生一个新的 `Stream` ，局部定义的 `compute_rest` 函数确保每当计算其余（rest）部分时，
该函数都会映射到 `Stream` 的其余（rest）部分。

```
>>> def map_stream(fn, s):
        if s is Stream.empty:
            return s
        def compute_rest():
            return map_stream(fn, s.rest)
        return Stream(fn(s,first), compute_rest)
```

可以通过定义一个 `compute_rest` 函数来过滤 `Stream` ，该函数将过滤器函数（filter function）应用于 `Stream` 的其余（rest）部分。
如果过滤器函数拒绝 `Stream` 的第一个元素，则立即计算其余元素。由于 `filter_stream` 是递归的，因此可以多次计算其余部分，直到找到有效的第一个元素。

```
>>> def filter_stream(fn, s);
        if s is Stream.empty:
            return s
        def compute_rest():
            return filter_stream(fn, s.rest)
        if fn(s.first):
            return Stream(s.first, compute_rest)
        else:
            return compute_rest()
```

上述的 `map_stream` 和 `filter_stream` 函数展示了 `Stream` 处理中的常见模式：每当计算其余（rest）部分时，
局部定义的 `compute_rest` 函数都会递归地将处理函数应用于 `Stream` 的其余部分。

要检查 `Stream` 的内容，我们可以将前面 k 个元素转换为 Python 列表。

```
>>> def first_k_as_list(s, k):
        fist_k = []
        while s is not Stream.empty and k > 0:
            first_k.append(s.first)
            s, k = s.rest, k - 1
        return first_k
```

这个简单的函数可以让我们验证我们的 `map_stream` 实现，用一个简单例子：把 3 到 7 取平方

```
>>> s = integer_stream(3)            # 创建整数流
>>> s
Stream(3, <...>)
>>> m = map_stream(lambda x: x*x, s) # 把 fn, s 传入 map_stream 来得到新的流
>>> m
Stream(9, <...>)
>>> first_k_as_list(m, 5)            # 检查前五项
[9, 16, 25, 36, 49]
>>> s
Stream(3, <...>)
>>> m = map_stream(lambda x: x*x, s)
>>> m
Stream(9, <...>)
>>> first_k_as_list(m, 5)
[9, 16, 25, 36, 49]
```

我们可以使用我们的 `filter_stream` 函数，使用 埃拉托斯特尼筛法（sieve of Eratosthenes） 来定义一个质数数列。
这种方法会过滤一个整数流，将所有是其第一个元素的倍数的数字移除。透过连续使用每个质数来过滤，所有的合数将会从流中被移除。

```
>>> def primes(pos_stream):
        def not_divible(x):
            return x % pos_stream.first != 0
        def compute_rest():
            primes(filter_stream(not_divible, pos_stream.rest))
        return Stream(pos_stream.first, compute_rest)
```

通过使用 `first_k_as_list` 来查看 `primes` stream 的前 7 个元素

```
>>> prime_numbers = primes(integer_stream(2))
>>> first_k_as_list(prime_numbers, 7)
[2, 3, 5, 7, 11, 13, 17]
```

流与迭代器形成对比，因为它们可以多次传递给纯函数并每次产生相同的结果。
通过将素（质）数流转换为列表，它不会被「用完」。即流的前缀转换为列表后， `prime_numbers` 的第一个元素仍然是 2。

```
>>> prime_numbers.first
2
```

正如链表为序列抽象提供了简单的实现一样，流提供了一个简单的、功能性的递归数据结构，通过使用高阶函数来实现惰性求值。