

2.3 序列

... details INFO

译者: [Congwen-Wang](#)、[zyl1121](#)

来源: [2.3 Sequences](#)

对应: Lab 04、Disc 04、Lab 05、Disc 05

...

序列 (sequence) 是一组有顺序的值的集合，是计算机科学中的一个强大且基本的抽象概念。序列并不是特定内置类型或抽象数据表示的实例，而是一个包含不同类型数据间共享行为的集合。也就是说，序列有很多种类，但它们都具有共同的行为。特别是：

- **长度 (Length)**：序列的长度是有限的，空序列的长度为 0。
- **元素选择 (Element selection)**：序列中的每个元素都对应一个小于序列长度的非负整数作为其索引，第一个元素的索引从 0 开始。

Python 包含几种内置的序列数据类型，其中最重要的是列表 (list)。

2.3.1 列表

列表 (list) 是一个可以有任意长度的序列。列表有大量的内置行为，以及用于表达这些行为的特定语法。我们已经见过列表字面量 (list literal)，它的计算结果是一个 list 实例，以及一个计算结果为列表中元素值的元素选择表达式。`len` 内置的 `len` 函数返回序列的长度。如下，`digits` 是一个包含四个元素的列表，索引为 3 的元素是 8。

↓
不止 list 中可以调用

```
>>> digits = [1, 8, 2, 8]
>>> len(digits)
4
>>> digits[3]
8
```

此外，多个列表间可以相加，并且列表可以乘以整数。对于序列来说，加法和乘法并不是作用在内部元素上的，而是对序列自身进行组合和复制。也就是说，operator 模块中的 `add` 函数 (和 + 运算符) 会生成一个为传入列表串联的新列表。operator 中的 `mul` 函数 (和 * 运算符) 可接收原列表和整数 `k` 来返回一个内容为原列表内容 `k` 次重复的新列表。

```
>>> [2, 7] + digits * 2
[2, 7, 1, 8, 2, 8, 1, 8, 2, 8]
```

任何值都可以包含在一个列表中，包括另一个列表。在嵌套列表中可以应用多次元素选择，以选择深度嵌套的元素。

```
>>> pairs = [[10, 20], [30, 40]]
>>> pairs[1]
[30, 40]
>>> pairs[1][0]
30
```

列表的浅拷贝

`a = [1, 2]`

`b = a`

将导致 a, b 与同一个对象绑定

执行诸如 `a[0] = 3; b[1] = 4` 之类

的语句都将使列表的元素发生变化

Frames Objects



另一种可能导致此结果的是 `append()` 方法

`s = [1, 2]; t = [3, 4]`

`s.append(t)`

`t[0] = 5`

因此最后 s 为 `[1, 2, [5, 4]]`

Frames Objects



2.3.2 序列遍历

在许多情况下，我们希望依次遍历序列的元素并根据元素值执行一些计算。这种情况十分常见，所以 Python 提供了一个额外的控制语句来处理序列的数据：`for` 循环语句。

考虑统计一个值在序列中出现了多少次的问题。我们可以使用 `while` 循环实现一个函数。

```
>>> def count(s, value):
    """统计在序列 s 中出现了多少次值为 value 的元素"""
    total, index = 0, 0
    while index < len(s):
        if s[index] == value:
            total = total + 1
        index = index + 1
    return total
>>> count(digits, 8)
2
```

Python 的 `for` 循环可以通过直接遍历元素值来简化函数，相比 `while` 循环无需引入变量名 `index`。

```
>>> def count(s, value):
    """统计在序列 s 中出现了多少次值为 value 的元素"""
    total = 0
    for elem in s:
        if elem == value:
            total = total + 1
    return total
>>> count(digits, 8)
2
```

一个 `for` 循环语句由如下格式的单个子句组成：

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

`for` 循环语句按以下过程执行：

1. 执行头部 (header) 中的 `<expression>`，它必须产生一个可迭代 (iterable) 的值 (译者注：可迭代的详细概念可见 4.2 隐式序列) 可见本章笔记 *Iterator and Generator*
2. 对该可迭代值中的每个元素，按顺序：
 1. 将当前帧的 `<name>` 绑定到该元素值
 2. 执行 `<suite>`

此执行过程中使用了可迭代值。列表是序列的一种，而序列是可迭代值，它们中的元素按其顺序进行迭代。Python 还包括其它可迭代类型，但我们现在将重点介绍序列。术语 "iterable" 的一般定义在第 4 章中关于迭代器的部分。

这个计算过程中的一个重要结果是：执行 `for` 语句后，`<name>` 将绑定到序列的最后一个元素。所以 `for` 循环引入了另一种可以通过语句更新环境的方法。

序列解包 (Sequence unpacking)：程序中的一个常见情况是序列的元素也是序列，但所有内部序列的长度是固定相同的。`for` 循环可以在头部的 `<name>` 中包含多个名称，来将每个元素序列“解包”到各自的元素中。

例如，我们可能有一个包含以列表为元素的 `pairs`，其中所有内部列表都只包含 2 个元素。

```
>>> pairs = [[1, 2], [2, 2], [2, 3], [4, 4]]
```

此时我们希望找到有多少第一元素和第二元素相同的内部元素对，下面的 `for` 循环在头部中包括两个名称，将 `x` 和 `y` 分别绑定到每对中的第一个元素和第二个元素。

```
>>> same_count = 0
>>> for x, y in pairs:
    if x == y:
        same_count = same_count + 1
>>> same_count
2
```

这种将多个名称绑定到固定长度序列中的多个值的模式称为序列解包（sequence unpacking），这与赋值语句中将多个名称绑定到多个值的模式类似。（译者注：如在 1.2.4 名称与环境中出现过的 `x, y = 3, 4.5`）

范围 (Ranges)：`range` 是 Python 中的另一种内置序列类型，用于表示整数范围。范围是用 `range` 创建的，它有两个整数参数：起始值和结束值加一。（译者注：其实可以有三个参数，第三个参数为步长，感兴趣可以自行搜索）
高中经典了属于是

```
>>> range(1, 10) # 包括 1, 但不包括 10
range(1, 10)
```

将 `range` 的返回结果传入 `list` 构造函数，可以构造出一个包含该 `range` 对象中所有值的列表，从而简单的查看范围内包含的内容。

```
>>> list(range(5, 8))
[5, 6, 7]
```

如果只给出一个参数，参数将作为双参数中的“结束值加一”，获得从 0 到结束值的范围。（译者注：其实单参数就相当于默认了起始值从 0 开始）

```
>>> list(range(4))
[0, 1, 2, 3]
```

范围通常出现在 `for` 循环头部中的表达式，以指定 `<suite>` 应执行的次数。一个惯用的使用方式是：如果 `<name>` 没有在 `<suite>` 中被使用到，则用下划线字符 "_" 作为 `<name>`。

```
>>> for _ in range(3):
    print('Go Bears!')

Go Bears!
Go Bears!
Go Bears!
```

对解释器而言，这个下划线只是环境中的另一个名称，但对程序员具有约定俗成的含义，表示该名称不会出现在任何未来的表达式中。

2.3.3 序列处理

序列是复合数据的一种常见形式，常见到整个程序都可能围绕着这个单一的抽象来组织。具有序列作为输入输出的模块化组件可以混用和匹配以实现数据处理。将序列处理流程中的所有操作链接在一起可以定义复杂组件，其中每个操作都是简单和集中的。

列表推导式 (List Comprehensions)：许多序列操作可以通过对序列中的每个元素使用一个固定表达式进行计算，并将结果值保存在结果序列中。在 Python 中，列表推导式是执行此类计算的表达式。

```
>>> odds = [1, 3, 5, 7, 9]
>>> [x+1 for x in odds]
[2, 4, 6, 8, 10]
```

上面的 `for` 关键字并不是 `for` 循环的一部分，而是列表推导式的一部分，因为它被包含在方括号里。

子表达式 `x+1` 通过绑定到 `odds` 中每个元素的变量 `x` 进行求值，并将结果值收集到列表中。

另一个常见的序列操作是选取原序列中满足某些条件的值。列表推导式也可以表达这种模式，例如选择 `odds` 中所有可以整除 25 的元素。

```
>>> [x for x in odds if 25 % x == 0]
[1, 5]
```

列表推导式的一般形式是：

想想学过的创建二维列表的语句吧

```
[<map expression> for <name> in <sequence expression> if <filter expression>]
```

为了计算列表推导式，Python 首先执行 `<sequence expression>`，它必须返回一个可迭代值。然后将每个元素值按顺序绑定到 `<name>`，再执行 `<filter expression>`，如果结果为真值，则计算 `<map expression>`，`<map expression>` 的结果将被收集到结果列表中。

聚合 (Aggregation)：序列处理中的第三种常见模式是将序列中的所有值聚合为一个值。内置函数 `sum`、`min` 和 `max` 都是聚合函数的示例。

通过组合对每个元素进行计算、选择元素子集和聚合元素的模式，我们就可以使用序列处理的方法解决问题。

完美数是等于其约数之和的正整数。`n` 的约数指的是小于 `n` 且可以整除 `n` 的正整数。可以使用列表推导式来列出 `n` 的所有约数。

```
>>> def divisors(n):
    return [1] + [x for x in range(2, n) if n % x == 0]

>>> divisors(4)
[1, 2]
>>> divisors(12)
[1, 2, 3, 4, 6]
```

通过 `divisors`，我们可以使用另一个列表推导式来计算 1 到 1000 的所有完美数。（1 通常也被认为是一个完美数，尽管它不符合我们对约数的定义。）

```
>>> [n for n in range(1, 1000) if sum(divisors(n)) == n]
[1, 6, 28, 496]
```

我们可以重用定义的 `divisors` 来解决另一个问题：在给定面积的情况下计算具有整数边长的矩形的最小周长。矩形的面积等于它的高乘以它的宽，因此给定面积和高度，我们可以计算出宽度。

使用 `assert` 可以规定宽度和高度都能整除面积，以确保边长是整数。

```
>>> def width(area, height):
    assert area % height == 0
    return area // height
```

矩形的周长是其边长之和，由此我们可以定义 `perimeter`。

```
>>> def perimeter(width, height):
    return 2 * width + 2 * height
```

对于边长为整数的矩形来说，高度必是面积的约数，所以我们可以考虑所有可能的高度来计算最小周长。

```
>>> def minimum_perimeter(area):
    heights = divisors(area)
    perimeters = [perimeter(width(area, h), h) for h in heights]
    return min(perimeters)

>>> area = 80
>>> width(area, 5)
16
>>> perimeter(16, 5)
42
>>> perimeter(10, 8)
36
>>> minimum_perimeter(area)
36
>>> [minimum_perimeter(n) for n in range(1, 10)]
[4, 6, 8, 8, 12, 10, 16, 12, 12]
```

高阶函数 (Higher-Order Functions) : 序列处理中常见的模式可以使用高阶函数来表示。首先可以将对序列中每个元素进行表达式求值表示为将某个函数应用于序列中每个元素。

```
>>> def apply_to_all(map_fn, s):
    return [map_fn(x) for x in s]
```

这与 `map()` 类似
详见 [Iterator and Generator](#)

仅选择满足表达式条件的元素也可以通过对每个元素应用函数来表示。

```
>>> def keep_if(filter_fn, s):
    return [x for x in s if filter_fn(x)]
```

这与 `filter()` 类似

最后，许多形式的聚合都可以被表示为：将双参数函数重复应用到 `reduced` 值，并依次对每个元素应用。

```
>>> def reduce(reduce_fn, s, initial):
    reduced = initial
    for x in s:
        reduced = reduce_fn(reduced, x)
    return reduced
```

例如，`reduce` 可用于将序列内的所有元素相乘。使用 `mul` 作为 `reduce_fn`，1 作为初始值，`reduce` 可用于将序列内的数字相乘。

```
>>> reduce(mul, [2, 4, 8], 1)
64
```

同样也可以用这些高阶函数来寻找完美数。

```
>>> def divisors_of(n):
    divides_n = lambda x: n % x == 0
    return [1] + keep_if(divides_n, range(2, n))

>>> divisors_of(12)
[1, 2, 3, 4, 6]
>>> from operator import add
>>> def sum_of_divisors(n):
    return reduce(add, divisors_of(n), 0)

>>> def perfect(n):
    return sum_of_divisors(n) == n

>>> keep_if(perfect, range(1, 1000))
[1, 6, 28, 496]
```

约定俗成的名字 (Conventional Names)：在计算机科学中，`apply_to_all` 更常用的名称是 `map`，而 `keep_if` 更常用的名称是 `filter`。`Python` 中内置的 `map` 和 `filter` 是以上函数的不以列表为返回值的泛化形式，这些函数在第 4 章中介绍。上面的定义等效于将内置 `map` 和 `filter` 函数的结果传入 `List` 构造函数。

```
>>> apply_to_all = lambda map_fn, s: list(map(map_fn, s))
>>> keep_if = lambda filter_fn, s: list(filter(filter_fn, s))
```

`reduce` 函数内置于 `Python` 标准库的 `functools` 模块中。在此版本中，`initial` 参数是可选的。

```
>>> from functools import reduce
>>> from operator import mul
>>> def product(s):
    return reduce(mul, s)

>>> product([1, 2, 3, 4, 5])
120
```

在 `Python` 程序中，更常见的是直接使用列表推导式而不是高阶函数，但这两种序列处理方法都被广泛使用。

2.3.4 序列抽象

我们已经介绍了两种满足序列抽象的内置数据类型：`list` 和 `range`。两者都满足我们在本节开始提到的条件：长度和元素选择。Python 还包含另外两个扩展序列抽象的行为。

成员资格 (Membership)：可用于测试某个值在序列中的成员资格。Python 有两个运算符 `in` 和 `not in`，它们的计算结果为 `True` 或 `False`，取决于元素是否出现在序列中。

```
>>> digits
[1, 8, 2, 8]
>>> 2 in digits
True
>>> 1828 not in digits
True
```

切片 (Slicing)：一个切片是原始序列的任意一段连续范围，由一对整数指定。和 `range` 构造函数一样，第一个整数表示起始索引，第二个整数是结束索引加一。（译者注：和前面的 `range` 一样，其实还有第三个参数代表步长，最经典的例子是使用 `s[::-1]` 得到 `s` 的逆序排列，具体可以自行搜索）

在 Python 中，序列切片的表达方式类似于元素选择，都使用方括号，方括号中的冒号用于分隔起始索引和结束索引。

如果起始索引或结束索引被省略则默认为极值：当起始索引被省略，则起始索引为 0；当结束索引被省略，则结束索引为序列长度，即取到序列最后一位。

```
>>> digits[0:2]
[1, 8]
>>> digits[1:]
[8, 2, 8]
```

列举 Python 序列抽象的这些额外行为让我们有机会反思什么构成了有用的数据抽象，毕竟抽象的丰富程度（即它包含多少行为）可能会带来额外负担。额外的行为对使用抽象的用户会有一定帮助。但另一方面，用新数据类型满足一个丰富抽象的要求可能很有挑战性。丰富抽象的另一个负面后果是用户需要更长的时间来学习。

序列具有丰富的抽象性，因为它们在计算中无处不在，所以学习一些复杂的行为是合理的。通常，大多数用户定义的抽象应尽可能简单。

拓展材料：切片表示法还包含各种特殊情况，例如负起始值、结束值和步长。完整的描述详见 *Dive Into Python 3* 中名为 [切片列表](#) 的小节中。在本章中，我们仅使用上述基本功能。

2.3.5 字符串

在计算机科学中，文本值可能比数字更重要。比如 Python 程序是以文本形式编写和存储的。Python 中文本值的内置数据类型称为字符串（string），对应构造函数 `str`。在 Python 中，表示、表达和操作字符串的细节有很多。

字符串是丰富抽象的另一个例子，程序员需要大量的努力才能掌握它。本节简要介绍了字符串的基本行为。

字符串字面量（string literals）可以表示任意文本，使用时将内容用单引号或双引号括起来。

```
>>> 'I am string!'
'I am string!'
>>> "I've got an apostrophe"
"I've got an apostrophe"
>>> '您好'
'您好'
```

单引号与双引号没有什么实质上的区别
除了单引号的表达式中可以出现双引号
反之亦然
例如 'I say, "I love her."
"She says 'she loves me.'"
都是合法的字符串表达形式.

当然字符串中也可以随时使用转义字符，甚至用 Unicode 也行。

我们已经在代码中看到过字符串，比如文档字符串 (docstring)、`print` 的调用中，以及 `assert` 语句中的错误消息。

字符串同样满足我们在本节开头介绍的序列的两个基本条件：它们具有长度且支持元素选择。字符串中的元素是只有一个字符的字符串。字符可以是字母表中的任何单个字母、标点符号或其他符号。

与其他编程语言不同，Python 没有单独的字符类型，任何文本都是字符串。表示单个字符的字符串的长度为 1。

```
>>> city = 'Berkeley'
>>> len(city)
8
>>> city[3]
'k'
```

还有一种较为特殊的字符串 — 原始字符串 (raw strings)
在处理它们时，所有的反斜杠将被保留，不会被解释为转义字符。
例如 `s=r'C:\path\to\folder'` 前面加个 `r` 或 `R` 即可
其中的反斜杠将被保留
这在表示路径，正则表达式时非常有用。

与列表一样，字符串也可以通过加法和乘法进行组合。

```
>>> 'Berkeley' + ', CA'
'Berkeley, CA'
>>> 'Shabu ' * 2
'Shabu Shabu '
```

常用的字符串还有 f 字符串，它提供了一种形成固定格式字符串的方法。

`f"…f expression…" 将自动计算 expression，转换为字符串并拼接至原字符串中。`

成员资格 (Membership)：字符串的行为与 Python 中的其他序列类型有所不同。字符串抽象不符合我们对列表和范围描述的完整序列抽象。具体来说，成员运算符 `in` 应用于字符串时的行为与应用于序列时完全不同，它匹配的是子字符串而不是元素。（译者注：如果字符串的行为和列表的一样，则应该匹配字符串的元素，即单个字符，但实际上匹配的是任意子字符串）

```
>>> 'here' in "Where's Waldo?"
True
```

多行字面量 (Multiline Literals)：字符串可以不限于一行。跨越多行的字符串字面量可以用三重引号括起，我们已经在文档字符串中广泛使用了这种三重引号。

```
>>> """The Zen of Python
claims, Readability counts.
Read more: import this."""
'The Zen of Python\nclaims, "Readability counts."\nRead more: import this.'
```

在上面的打印结果中，`\n`（读作“反斜杠 n”）是一个表示换行的单个元素。尽管它显示为两个字符（反斜杠和 "n"），但为了便于计算长度和元素选择，它被视为单个字符。

字符串强制转换 (String Coercion)：通过以对象值作为参数调用 `str` 的构造函数，可以从 Python 中的任何对象创建字符串。字符串的这一特性在用构造各种类型对象的描述性字符串时非常有用。

```
>>> str(2) + ' is an element of ' + str(digits)
'2 is an element of [1, 8, 2, 8]'
```

拓展材料：在计算机中文本编码是一个复杂的话题。本章中我们将抽象出字符串如何表示的细节。然而对于许多应用程序而言，计算机进行字符串编码的实现细节是必不可少的知识。*Dive Into Python 3* 的[字符串章节](#)提供了字符编码和 Unicode 的描述。

2.3.6 树

使用列表作为其他列表中元素的能力为编程语言提供了一种新的组合方式。这种能力称为数据类型的闭包属性 (closure property)。一般来说，如果某种数据值组合得到的结果也可以用相同的方法进行组合，则该方法具有闭包属性。（译者注：可参考维基百科中对闭包的官方定义：「数学中，若对某个集合的成员进行一种运算，生成的仍然是这个集合的成员，则该集合被称为在这个运算下闭合。」）

闭包是所有组合方式的关键，因为它使我们可以创建层次结构——由“部分”组成的结构，部分其本身也由部分组成。

我们可以使用方框指针表示法 (box-and-pointer notation) 在环境图中可视化列表。列表被描述为包含列表元素的相邻框。数字、字符串、布尔值和 None 等原始值出现在元素框内。复合值如函数值和列表等由箭头指示。

在列表中嵌套列表可能会带来复杂性。树 (tree) 是一种基本的数据抽象，它将层次化的值按照一定的规律进行组织和操作。

一个树有一个根标签 (root label) 和一系列分支 (branch)。树的每个分支都是一棵树，没有分支的树称为叶子 (leaf)。树中包含的任何树都称为该树的子树（例如分支的分支）。树的每个子树的根称为该树中的一个节点 (node)。

树的数据抽象由构造函数 `tree`、选择器 `label` 和 `branches` 组成。我们从简化版本开始讲起。

```
>>> def tree(root_label, branches=[]):
    for branch in branches:
        assert is_tree(branch), '分支必须是树'
    return [root_label] + list(branches)

>>> def label(tree):
    return tree[0]

>>> def branches(tree):
    return tree[1:]
```

另一种可用的表示方法为元组

```
def tree(label, children=()):
    return (label, children)
def label(tree):
    return tree[0]
def children(tree):
    return tree[1]
```

这省去了切割列表获取子树的麻烦。

只有当树有根标签并且所有分支也是树时，树才是结构良好的。在 `tree` 构造函数中使用了 `is_tree` 函数以验证所有分支是否结构良好。

```
>>> def is_tree(tree):
    if type(tree) != list or len(tree) < 1:
        return False
    for branch in branches(tree):
        if not is_tree(branch):
            return False
    return True
```

`is_leaf` 函数检查树是否有分支，若无分支则为叶子节点。

```
>>> def is_leaf(tree):
    return not branches(tree)
```

树可以通过嵌套表达式来构造。以下树 `t` 具有根标签 3 和两个分支。

```
>>> t = tree(3, [tree(1), tree(2, [tree(1), tree(1)])])
>>> t
[3, [1], [2, [1], [1]]]
>>> label(t)
3
>>> branches(t)
[[1], [2, [1], [1]]]
>>> label(branches(t)[1])
2
>>> is_leaf(t)
False
>>> is_leaf(branches(t)[0])
True
```

树本身就是一种递归结构

树递归 (Tree-recursive) 函数可用于构造树。例如，我们定义 The nth Fibonacci tree 是指以第 n 个斐波那契数为根标签的树。那么当 $n > 1$ 时，它的两个分支也是 Fibonacci tree。这可用于说明斐波那契数的树递归计算。

一般地，对于树的递归函数 `function(tree)` 分为以下两个部分：

① 处理节点的标签

② 遍历节点的子树，对每个子树应用 `function`。

相信递归函数的能力，尽管有时候我们不知道递归函数具体的实现方法。

相信“相信”的力量。

搜索实际上就是递归。
处理边界条件，处理递归方法。

```
>>> def fib_tree(n):
    if n == 0 or n == 1:
        return tree(n)
    else:
        left, right = fib_tree(n-2), fib_tree(n-1)
        fib_n = label(left) + label(right)
        return tree(fib_n, [left, right])
>>> fib_tree(5)
[5, [2, [1], [1, [0], [1]]], [3, [1, [0], [1]]], [2, [1], [1, [0], [1]]]]]
```

树递归函数也可用于处理树。例如，`count_leaves` 函数可以计算树的叶子数。

```
>>> def count_leaves(tree):
    if is_leaf(tree):
        return 1
    else:
        branch_counts = [count_leaves(b) for b in branches(tree)]
        return sum(branch_counts)
>>> count_leaves(fib_tree(5))
8
```

分割树 (Partition trees)：树也可以用来表示将一个正整数分割为若干个正整数的过程。比如可通过一个形式为二叉树的分割树来表示将 n 分割为不超过 m 的若干正整数之和的计算过程中所做的选择。在非叶子节点的分割树节点中：

- 根标签是 m 。
- 左侧（索引 0）分支包含划分 n 时至少使用一个 m 的所有方法
- 右侧（索引 1）分支包含划分 n 时使用的正整数不超过 $m - 1$ 的所有方法

分割树叶子节点上的标签表示从树根到叶子的路径是否分割成功。

```
>>> def partition_tree(n, m):
    """返回将 n 分割成不超过 m 的若干正整数之和的分割树"""
    if n == 0:
        return tree(True)
    elif n < 0 or m == 0:
        return tree(False)
    else:
        left = partition_tree(n-m, m)
        right = partition_tree(n, m-1)
        return tree(m, [left, right])

>>> partition_tree(2, 2)
[2, [True], [1, [1, [True], [False]], [False]]]
```

译者注：我们可以以 $n = 6$, $m = 4$ 为例，尝试“将 6 分割为不超过 4 的若干正整数之和”。首先，所有的分割方式可以被分两类：

1. 使用至少一个 4 来分割
2. 使用不超过 3 的若干正整数来分割

让我们再进一步简化：

1. 使用至少一个 4 来分割，即：先将 6 分割出一个 4，再将余下的 $(6 - 4 = 2)$ 分割为不超过 4 的若干整数之和。
2. 使用不超过 3 的若干整数来分割，即：“将 6 分割为不超过 3 的若干整数之和”。

我们发现，它们都可以抽象出同样的形式，只是参数不同。那么就可以用递归的方法来处理，直接将再次调用 `partition_tree` 得到的结果作为自己的左右分支。

除此之外，我们还需明确递归的出口，即什么情况记为分割成功 (`True`)，什么情况记为分割失败 (`False`)，包括：

- 成功分割：一旦分割后的 $n = 0$ ，说明已经完成分割，返回 `True`
- 不成功分割：分割后的 $n < 0$ ，说明正整数之和已经超过最初被分割的 n ，不符合要求； m 递减至 0，不符合需要分割为正整数的要求，这两种情况都应返回 `False`

另一个遍历树的树递归过程是将分割树的所有分割方案打印。每个分区都构造为一个列表，当到达叶子节点且节点标签为 `True` 时就会打印分区。

```
>>> def print_parts(tree, partition=[]):
    if is_leaf(tree):
        if label(tree):
            print(' + '.join(partition))
    else:
        left, right = branches(tree)
        m = str(label(tree))
        print_parts(left, partition + [m])
        print_parts(right, partition)

>>> print_parts(partition_tree(6, 4))
4 + 2
4 + 1 + 1
3 + 3
```

```
3 + 2 + 1
3 + 1 + 1 + 1
2 + 2 + 2
2 + 2 + 1 + 1
2 + 1 + 1 + 1 + 1
1 + 1 + 1 + 1 + 1 + 1
```

切片操作同样适用于树的分支。例如我们可能想限制树的分支数量。二叉树就是这样有分支数量限制的树，二叉树可以是单个叶子节点，也可以是一个最多包含两个二叉树分支的节点。二叉树化（binarization）是一种常见的树转换方法，通过将相邻的分支组合在一起从原始树计算出二叉树。

```
>>> def right_binarize(tree):
    """根据 tree 构造一个右分叉的二叉树"""
    if is_leaf(tree):
        return tree
    if len(tree) > 2:
        tree = [tree[0], tree[1:]]
    return [right_binarize(b) for b in tree]

>>> right_binarize([1, 2, 3, 4, 5, 6, 7])
[1, [2, [3, [4, [5, [6, 7]]]]]]
```

2.3.7 链表

此处链表的实现方法与高中时有明显区别。
这里感觉更像是度为1的树

目前，我们只使用了内置类型来表示序列。但是我们也可以开发未内置于 Python 中的序列表示。链表（linked list）是一种常见的由嵌套对构造的序列表示。下面的环境图阐明了包含 1、2、3 和 4 的四元素序列的链表表示。

链表是一个包括首元素（在本例中为 1）和剩余元素（在本例中为 2、3、4 的复合）的元素对，第二个元素也是一个链表。最内部仅包含 4 的链表的剩余部分为 "empty"，代表空链表。

链表具有递归结构：链表的其余部分也是链表或 "empty"。我们可以定义一个抽象数据表示（abstract data representation）来验证、构建和选择链表的组件。

```
>>> empty = 'empty'
>>> def is_link(s):
    """判断 s 是否为链表"""
    return s == empty or (len(s) == 2 and is_link(s[1]))

>>> def link(first, rest):
    """用 first 和 rest 构建一个链表"""
    assert is_link(rest), "rest 必须是一个链表"
    return [first, rest]

>>> def first(s):
    """返回链表 s 的第一个元素"""
    assert is_link(s), "first 只能用于链表"
    assert s != empty, "空链表没有第一个元素"
    return s[0]

>>> def rest(s):
    """返回 s 的剩余元素"""
    assert is_link(s), "rest 只能用于链表"
    assert s != empty, "空链表没有剩余元素"
```

```
    return s[1]
```

上面，`link` 是一个构造函数，`first` 和 `rest` 是链表抽象数据表示的选择器函数。链表的行为是成对的，构造函数和选择器互为反函数。

- 如果链表 `s` 是由第一个元素 `f` 和剩余元素链表 `r` 构造的，那么 `first(s)` 返回 `f`，`rest(s)` 返回 `r`。

我们可以使用构造函数和选择器来操作链表。

```
>>> four = link(1, link(2, link(3, link(4, empty()))))
>>> first(four)
1
>>> rest(four)
[2, [3, [4, 'empty']]])
```

上述抽象的实现借助于双元素列表来实现“对”。值得注意的是，我们还可以使用函数来实现“对”，并且我们可以使用任意“对”来实现链表，所以我们可以仅使用函数来实现链表。

虽然链表可以按顺序存储一系列值，但我们还没有证明它满足序列抽象的条件。使用定义的抽象数据表示，我们可以实现序列共有的两种行为：长度和元素选择。

```
>>> def len_link(s):
    """返回链表 s 的长度"""
    length = 0
    while s != empty:
        s, length = rest(s), length + 1
    return length

>>> def getitem_link(s, i):
    """返回链表 s 中索引为 i 的元素"""
    while i > 0:
        s, i = rest(s), i - 1
    return first(s)
```

现在，我们可以使用这些函数将链表作为序列来操作。（我们还不能使用内置的 `len` 函数、元素选择语法或 `for` 循环，但很快就可以。）

```
>>> len_link(four)
4
>>> getitem_link(four, 1)
2
```

下面的一系列环境图阐明了 `getitem_link` 在链表中寻找索引为 1 的元素 2 的迭代过程。如下，我们使用 Python 原语定义了链表 `four` 以简化图表。这种实现违反了抽象屏障，但允许我们更简单地检查此示例的计算过程。

首先，函数 `getitem_link` 被调用，创建了一个局部帧（local frame）。

`while` 循环头部中的表达式计算结果为 `True`，导致 `while` 循环中的赋值语句被执行。函数 `rest` 返回以 2 开头的子列表。

接下来，局部帧中的 `s` 将被更新为以原始列表的第二个元素开头的子列表。现在 `while` 循环头部表达式会产生一个 `False` 值，跳出 `while` 循环后 Python 会执行 `getitem_link` 最后一行的 `return` 语句。

这个最终的环境图显示了调用 `first` 的局部帧，其中包含绑定到同一子列表的 `s`。`first` 函数将返回列表 `s` 的首元素 2，这也将作为 `getitem_link` 的返回值。

此示例演示了链表的常用计算模式，迭代中的每个步骤都对初始列表的越来越短的后缀进行操作。这种计算链表长度和查找元素的处理过程需要一些额外时间来计算。而 Python 的内置序列类型以不同的方式实现这些功能，计算序列长度或选择其内部元素并不需要过多的开销。具体的实现方法超出了本文的范围，可以自行了解。

递归操作 (Recursive manipulation)：`len_link` 和 `getitem_link` 都是以迭代形式实现的。它们逐渐剥离嵌套对的每一层，直到到达列表的末尾（在 `len_link` 中）或找到所需的元素（在 `getitem_link` 中）。我们还可以通过递归的方式实现长度计算和元素选择。

```
>>> def len_link_recursive(s):
    """返回链表 s 的长度"""
    if s == empty:
        return 0
    return 1 + len_link_recursive(rest(s))

>>> def getitem_link_recursive(s, i):
    """返回链表 s 中索引为 i 的元素"""
    if i == 0:
        return first(s)
    return getitem_link_recursive(rest(s), i - 1)

>>> len_link_recursive(four)
4
>>> getitem_link_recursive(four, 1)
2
```

这些递归实现沿着元素对形成的链，直到到达列表末尾（在 `len_link_recursive` 中）或找到所需元素（在 `getitem_link_recursive` 中）。

递归对于转换和组合链表也很有用。

```
>>> def extend_link(s, t):
    """返回一个在 s 链表的末尾连接 t 链表后的延长链表"""
    assert is_link(s) and is_link(t)
    if s == empty:
        return t
    else:
        return link(first(s), extend_link(rest(s), t))

>>> extend_link(four, four)
[1, [2, [3, [4, [1, [2, [3, [4, 'empty']]]]]]]]

>>> def apply_to_all_link(f, s):
    """应用 f 到 s 中的每个元素"""
    assert is_link(s)
    if s == empty:
        return s
    else:
```

```

        return link(f(first(s)), apply_to_all_link(f, rest(s)))

>>> apply_to_all_link(lambda x: x*x, four)
[1, [4, [9, [16, 'empty']]))

>>> def keep_if_link(f, s):
    """返回 s 中 f(e) 为 True 的元素"""
    assert is_link(s)
    if s == empty:
        return s
    else:
        kept = keep_if_link(f, rest(s))
        if f(first(s)):
            return link(first(s), kept)
        else:
            return kept

>>> keep_if_link(lambda x: x%2 == 0, four)
[2, [4, 'empty']]

>>> def join_link(s, separator):
    """返回由 separator 分隔的 s 中的所有元素组成的字符串"""
    if s == empty:
        return ""
    elif rest(s) == empty:
        return str(first(s))
    else:
        return str(first(s)) + separator + join_link(rest(s), separator)
>>> join_link(four, ", ")
'1, 2, 3, 4'

```

递归构造 (Recursive Construction) : 链表在递增地构造序列时特别有用，这种情况在递归计算中经常出现。

第一章中的 `count_partitions` 函数通过树递归过程计算了将 n 分割为不超过 m 的若干正整数之和的所有方法数。我们可以使用序列显式列举具体的分割方案。

我们使用与 `count_partitions` 相同的递归分析，将 n 分割为不超过 m 的若干正整数之和，包括：

1. 将 $n - m$ 分割为不超过 m 的若干正整数之和
2. 将 n 分割为不超过 $m - 1$ 的若干正整数之和

对于递归终止条件，我们发现 0 只有一个空分割方案，而分割负整数和使用小于 1 的整数是不可能的。

```

>>> def partitions(n, m):
    """返回一个包含 n 的分割方案的链表，其中每个正整数不超过 m"""
    if n == 0:
        return link(empty, empty) # 包含空分割的链表
    elif n < 0 or m == 0:
        return empty
    else:
        using_m = partitions(n-m, m)
        with_m = apply_to_all_link(lambda s: link(m, s), using_m)
        without_m = partitions(n, m-1)
        return extend_link(with_m, without_m)

```

在递归的情况下，我们构造了两个分割子列表。第一种情况使用 `m`，因此我们将 `m` 添加到 `using_m` 结果的每一项以形成 `with_m`。

`partitions` 的结果是高度嵌套的：链表中包含链表，每个链表都为以 `list` 为值的嵌套对。使用 `join_link` 给结果链表加上分隔符，我们可以以人类可读的方式显示分割方案。

```
>>> def print_partitions(n, m):
    lists = partitions(n, m)
    strings = apply_to_all_link(lambda s: join_link(s, " + "), lists)
    print(join_link(strings, "\n"))

>>> print_partitions(6, 4)
4 + 2
4 + 1 + 1
3 + 3
3 + 2 + 1
3 + 1 + 1 + 1
2 + 2 + 2
2 + 2 + 1 + 1
2 + 1 + 1 + 1 + 1
1 + 1 + 1 + 1 + 1 + 1
```