# 2.4 可变数据

::: details INFO
译者：Joseph_L, Congwen-Wang

来源：2.4 Mutable Data

对应：HW 05、Disc 04、Exam Prep 03
:::

我们已经见识到了抽象在帮助我们应对复杂的大型系统时的重要性，但高效编程还需要一些系统化的原则，从而帮助我们更好地设计整个应用程序。具体来说，我们需要一些策略来模块化一个大型系统。所谓模块化，即将整个系统划分为独立维护开发，但又相互关联的模块。

在创建模块化项目时，一个非常有用的实践是引入可能随时间改变状态的数据。在这种情况下，一个单独的数据对象可以表示一些忽略整个程序其它部分而独立演变的事物。可变对象的行为可能会受到其自身历史状态的影响，就像真实世界中的事物一样。面向对象编程 (object-oriented programming) 的核心就是向数据添加状态。

## 2.4.1 对象隐喻

在本书的开头，我们区分了函数和数据的概念：函数发起某些操作，而数据是被操作的一方。但是当我们将函数引入到数据中时，我们就知道数据本身也可以有行为。函数可以被当作数据进行操作，同时也可以被调用来执行某些行为。

对象 (objects) 将数据的值和行为结合到了一起。对象可以直接表示某些信息，也可以用自身的表现行为来表达想表达的东西。一个对象具体应该怎么和其它对象进行交互，都被封装并绑定到了该对象自身的某些值上。当我们试图打印某个对象时，它自己知道应该如何以文字的形式表示自己。如果一个对象由多个部分组成，它知道应该怎么根据实际情况对外展示那些不同的部分。对象既是数据信息又是操作流程，它把二者结合到一起，从而表达复杂事物的属性、交互和行为。

在 Python 中，对象的行为是通过特定的语法和术语实现的，以日期为例：

```
>>> from datetime import date
```

`date` 这个名称是一个 `class` 类，正如我们所见，一个类代表了一种值。具体的日期被称为这个日期类的实例对象。要想构建一个实例对象，可以用特定的参数去调用该类得到：

```
>>> tues = date(2014, 5, 13)
```

尽管 `tues` 是用基础数字构建出来的，但它具有日期的能力。举个例子，用另一个日期减掉它，我们可以得到一个时间间隔。我们可以打印一下这个间隔：

```
>>> print(date(2014, 5, 19) - tues)
6 days, 0:00:00
```

对象有属性（attributes）的概念，可以理解为该对象中某个值的名字。和其它许多编程语言一样，我们在 Python 中使用点语法来访问一个对象中的某个属性。

```
>>> <expression> . <name>
```

在上面的代码中，`<expression>` 表示一个对象，`<name>` 表示该对象中对某个属性的名称。

与之前介绍的其它变量名称不同，这些属性名称无法在运行环境中直接访问。属性名称是对象实例所独有的，只能通过点语法来访问。

```
>>> tues.year
2014
```

对象还有方法（methods）的概念，其实也是属性，只不过该属性的值是函数。对象知道如何执行这些方法。具体实现起来，方法就是根据其自身的输入参数以及它所在的对象来计算特定结果的函数。举例来说，`strftime`（string format of time）方法接受一个参数，该参数描述了具体的时间展示格式（e.g., %A 表示以完整格式返回星期）。

```
>>> tues.strftime('%A, %B %d')
'Tuesday, May 13'
```

要计算 `strftime` 的返回值需要两个输入：期望展示的时间格式，以及 `tues` 中包含的日期信息。这个方法内部已经有了处理日期相关的逻辑，并且能够返回我们期望的结果。我们从来没有说过 2014 年 5 月 13 日是星期二，但是日期这个类本身就有这种能力，它能够知道一个特定的日期应该是星期几。通过把数据和行为绑定到一起， Python 为我们提供了一个已经完全抽象好的、可靠的 date 对象。

不仅 date 是对象，我们之前提到的数字、字符串、列表、区间等都是对象。它们本身表示数据，同时还拥有它们所代表的数据的行为。它们还有属性和方法。举例来说，字符串有一系列帮助我们处理文本的方法。

```
>>> '1234'.isnumeric()
True
>>> 'rOBERT dE nIRO'.swapcase()
'Robert De Niro'
>>> 'eyes'.upper().endswith('YES')
True
```

实际上，Python 中所有的值都是对象。也就是说，所有的值都有行为和属性，它们拥有它们所代表的数据的行为。

## 2.4.2 序列对象

像数字这样的基本数据类型的实例是不可变（immutable）的。它们所代表的值，在程序运行期间是不可以更改的。 另一方面，列表就是可变的（mutable）。

可变数据用来表示那些会在程序运行期间发生变化的数据。时间每天都在流逝，虽然一个人在一天天地长大、变老，或者有一些其它什么变化，但是这个人还是这个人，这一点是没有发生变化的。类似地，一个对象也可能通过某些操作更改自身的属性。举例来说，一个列表中的数据是可能会发生变化的。大部分变化的发生，都是通过调用列表实例的方法来触发的。

我们可以通过一个简单的扑克牌游戏来介绍一些操作列表的方法。下面代码中的注释解释了每次变更后带来的影响。

大约在公元 9 世纪前后，中国发明了扑克牌。在最早的扑克牌中，只有三种花色，分别代表了当时货币的面额：

```
>>> chinese = ['coin', 'string', 'myriad']  # 一组字符串列表
>>> suits = chinese                         # 为同一个列表指定了两个不同的变量名
```

当扑克牌（可能是经由埃及）传到欧洲后，西班牙的纸牌中只剩下 `coin` 这一种花色：

```
>>> suits.pop()              # 从列表中移除并返回最后一个元素
'myriad'
>>> suits.remove('string')   # 从列表中移除第一个与参数相同的元素
```

随着时间推移，又额外演变出了另外三种花色：

```
>>> suits.append('cup')                # 在列表最后插入一个元素
>>> suits.extend(['sword', 'club'])    # 将另外一个列表中的所有元素添加到当前列表最后
```

同时，意大利人给花色 `swords` 叫 `spades`：

```
>>> suits[2] = 'spade'   # 替换某个元素
```

这样我们就得到了一副传统意大利扑克牌的所有花色：

```
>>> suits
['coin', 'cup', 'spade', 'club']
```

现在美国使用的扑克牌实际上是法国的变种，修改了前两种花色：

```
>>> suits[0:2] = ['heart', 'diamond']  # 替换一组数据
>>> suits
['heart', 'diamond', 'spade', 'club']
```

除此之外，还有插入、排序、反转列表的方法。所有这些方法都是直接改变了目标列表的值，而不是创建了一个新的列表对象。

**数据共享和身份（Sharing and Identity）。** 正是由于我们没有在操作数据时创建新的列表，而是直接操作的源数据，这就导致变量 `chinese` 也被改变了，因为它和变量 `suits` 绑定到时同一个列表！

```
>>> chinese  # 这个变量名与 "suits" 指向的是同一个列表对象
['heart', 'diamond', 'spade', 'club']
```

这是我们第一次遇到类似的情况。在之前的代码里，如果某个变量没有出现在被执行的代码中，那该变量的值就不会被这段代码影响。但是对于可变数据来说，对一个变量名执行到方法调用可能会影响到另一个变量名所绑定的数据。

在下面的代码运行示意图中，可以看到虽然我们仅仅针对 `suits` 数据做了一系列操作，但 `chinese` 所绑定到数据还是被改变了。逐行执行下面的代码实例，观察一下整个过程。

我们可以利用列表的构造器函数 `list` 来对一个列表进行复制。复制完成后，两个列表数据的改动不会再影响彼此，除非二者共享了同一份数据。

```
>>> nest = list(suits)  # 复制一个与 suits 相同的列表，并命名为 nest
>>> nest[0] = suits     # 创建一个嵌套列表，列表第一项是另一个列表
```

根据当前的运行环境，改动变量 `suits` 所对应的列表数据会影响到 `nest` 列表的第一个元素，也就是我们上面刚刚创建的嵌套列表，而 `nest` 中的其它元素不受影响：

```
>>> suits.insert(2, 'Joker')  # 在下标为 2 的位置插入一条新元素，其余元素相应向后移动
>>> nest
[['heart', 'diamond', 'Joker', 'spade', 'club'], 'diamond', 'spade', 'club']
```

同样的，对 `nest` 列表的第一个元素撤销上述操作也会影响到 `suits` 列表：

```
>>> nest[0].pop(2)
'Joker'
>>> suits
['heart', 'diamond', 'spade', 'club']
```

逐步执行示例代码，我们可以看到嵌套列表的具体表现。

尽管两个列表的元素值相同，但他们仍然可能是完全不同的两个列表对象，所以我们需要一个机制来验证两个对象是否相同。Python 提供了 `is` 和 `is not` 两种比较操作符来验证两个变量是否指向同一个对象。如果两个对象的值完全相等，则说明它们两个是同一个对象，对其中任意一个对象的改动都将影响到另外一个。身份验证比简单的相等验证更准确。

```
>>> suits is nest[0]
True
>>> suits is ['heart', 'diamond', 'spade', 'club']
False
>>> suits == ['heart', 'diamond', 'spade', 'club']
True
```

最后两个比较说明了 `is` 和 `==` 的区别。前者是检验的是对象的内存地址，而后者只是判断内容是否相同。

**列表推导式。** 列表推导式总会返回一个新列表。举例来说，`unicodedata` 模块记录了 Unicode 字母表中每个字符的官方名称。我们可以通过字符名称找到对应的 unicode 字符，包括卡牌花色。

```
>>> from unicodedata import lookup
>>> [lookup('WHITE ' + s.upper() + ' SUIT') for s in suits]
['♡', '♢', '♤', '♧']
```

返回的这个列表不包含 `suits` 列表中的任何元素，执行列表推导也不会修改 `suits` 列表。

你可以在 Dive into Python3 的 Unicode 章节 阅读更多关于文本表示的 Unicode 标准。

**元组。** 元组是指 Python 内置类型 `tuple` 的实例对象，其是不可变序列。我们可以将不同数据用逗号分隔，用这种字面量的方式即可以创建一个元组。括号并不是必须的，但是一般都会加上。元组中可以放置任意对象。

```
>>> 1, 2 + 3
(1, 5)
>>> ("the", 1, ("and", "only"))
('the', 1, ('and', 'only'))
>>> type( (10, 20) )
<class 'tuple'>
```

空元组或者只有一个元素的元组，有特定的字面量语法：

```
>>> ()    # 0 elements
()
>>> (10,) # 1 element
(10,)
```

和列表相同，元组有确定的长度，并支持元素索引。元组还有一些与列表相同的方法，比如 `count` 和 `index`。

```
>>> code = ("up", "up", "down", "down") + ("left", "right") * 2
>>> len(code)
8
>>> code[3]
'down'
>>> code.count("down")
2
>>> code.index("left")
4
```

但是，列表中那些用于操作列表元素的方法并不适用于元组，因为元组是不可变的。

尽管无法修改元组的元素，但是如果元组中的元素本身是可变数据，那我们也是可以对该元素进行操作的。

在批量赋值的场景下，元组会被隐式地使用到。将两个值批量赋值给两个变量时，实际上是创建了一个包含两个元素的元组，然后对其进行解构赋值。

## 2.4.3 字典

字典（Dictionary）是 Python 的内置类型，用来存储和操作带有映射关系的数据。一个字典包含一组键值对（key-value pairs），其中键和值都是对象。字典的主要目的是抽象一组基于键值对的数据，在字典中，数据的存取都是基于带有描述性信息的键而不是连续递增的数字。

字典的 key 一般都是字符串（String），因为我们习惯用字符串来表示某个事物的名称。下面这个字典字面量表示了一组罗马数字：

```
>>> numerals = {'I': 1.0, 'V': 5, 'X': 10}
```

在字典元素中查找某个 key 对应的 value，与我们之前在列表中使用的操作符相同：

```
>>> numerals['X']
10
```

在字典中，一个 key 只能对应一个 value。无论是向字典中增加新的键值对，还是修改某个 key 值对应的 value，都可以使用赋值语句实现：

```
>>> numerals['I'] = 1
>>> numerals['L'] = 50
>>> numerals
{'I': 1, 'X': 10, 'L': 50, 'V': 5}
```

注意上面的打印输出，'L' 并没有被插入到字典的末尾。字典是无序的。当我们打印一个字典的时候，键值对会以某种顺序被渲染在页面上，但作为 Python 语言的使用者，我们无法预测这个顺序是什么样的。如果我们多运行几次这个程序，字典输出的顺序可能会有所变化。

> 译者注：Python 3.7 及以上版本的字典顺序会确保为插入顺序，此行为是自 3.6 版开始的 CPython 实现细节，字典会保留插入时的顺序，对键的更新也不会影响顺序，删除后再次添加的键将被插入到末尾

我们也可以在运行环境图中查看字典的结构。

字典类型也提供了一系列遍历字典内容的方法。`keys`、`values` 和 `items` 方法都返回一个可以被遍历的值。

```
>>> sum(numerals.values())
66
```

利用 `dictionary` 构造方法，我们可以将一个由键值对组成的列表转化为一个字典对象。

```
>>> dict([(3, 9), (4, 16), (5, 25)])
{3: 9, 4: 16, 5: 25}
```

但是字典类型也有一些限制：

- 字典的 key 不可以是可变数据，也不能包含可变数据
- 一个 key 只能对应一个 value

第一个限制是由于字典在 Python 内部的实现机制导致的。字典类型具体的实现机制不在这里展开。简单来说，假设是 key 值告诉 Python 应该去内存中的什么位置找对应的键值对，如果 key 值本身发生了变化，那键值对在内存中的位置信息也就丢失了。比如，元组可以被用来做字典的 key 值，但是列表就不可以。

第二个限制是因为字典本身被设计为根据 key 去查找 value，只有 key 和 value 的绑定关系是唯一确定的，我们才能够找到对应的数据。

字典中一个很有用的方法是 `get`，它返回指定 key 在字典中对应的 value；如果该 key 在字典中不存在，则返回默认值。`get` 方法接收两个参数，一个 key，一个默认值。

```
>>> numerals.get('A', 0)
0
>>> numerals.get('V', 0)
5
```

与列表类似，字典也有推导式语法。其中，key 和 value 使用冒号分隔。字典推导式会创建一个新的字典对象。

```
>>> {x: x*x for x in range(3,6)}
{3: 9, 4: 16, 5: 25}
```

## 2.4.4 局部状态

列表和字典拥有局部状态（local state），即它们可以在程序执行过程中的某个时间点修改自身的值。状态（state）就意味着当前的值有可能发生变化。

函数也是有状态的。举例来说，我们可以定义一个函数，来抽象从银行账户中取钱的过程。我们为这个函数命名为 `withdraw`，它接收一个参数，代表取钱的金额。如果账户中有足够的金额，`withdraw` 会返回取完钱以后的余额；否则，`withdraw` 会返回「余额不足」。假设我们账户里有 100 美元，调用 withdraw 应该得到如下结果：

```
>>> withdraw(25)
75
>>> withdraw(25)
50
>>> withdraw(60)
'余额不足'
>>> withdraw(15)
35
```

在上面的代码中，表达式 `withdraw(25)` 被执行了两次，但是返回了不同的结果。因此，我们定义的这个函数不是纯函数（in-pure）。执行这个函数在返回内容的同时，还产生了其它副作用（side effects），导致多次调用同一个函数得到的结果却不相同。这里的副作用之所以会出现，是因为 `withdraw` 函数更改了它所在的栈帧之外的变量。

```
>>> withdraw = make_withdraw(100)
```

`make_withdraw` 的实现需要一种新的声明形式：非局部（nonlocal）声明。当我们调用 `make_withdraw` 的时候，我们将初始余额声明为 `balance` 变量，然后我们再定义并返回一个局部函数 `withdraw`，它会在调用时更新并返回 balance 的值。

```
>>> def make_withdraw(balance):
        """返回一个每次调用都会减少 balance 的 withdraw 函数"""
        def withdraw(amount):
            nonlocal balance                 # 声明 balance 是非局部的
            if amount > balance:
                return '余额不足'
            balance = balance - amount       # 重新绑定
            return balance
        return withdraw
```

当 balance 属性为声明为 nonlocal 后，每当它的值发生更改时，相应的变化都会同步更新到 `balance` 属性第一次被声明的位置。回想一下，在没有 noncal 声明之前，所有对 balance 的重新赋值操作都会在当前环境的第一帧中绑定。非局部语句指示名称不会出现在第一个（局部）帧或最后一个（全局）帧，而是出现在其他地方。

以下运行环境图展示了多次调用由 make_withdraw 创建的函数的效果。

第一个 def 声明的表现符合我们的预期：它创建一个新的自定义函数并将该函数以 `make_withdraw` 为名绑定到全局帧中。随后调用 make_withdraw 创建并返回一个局部定义的函数 withdraw。参数 balance 则绑定在该函数的父帧中。最重要的是，在这个示例中，变量名 balance 只有一个绑定关系。

接下来，我们调用 `make_withdraw` 得到函数 `wd`，然后调用 `wd` 方法并入参 5。`withdraw` 函数执行在一个新的环境中，并且该环境的 `parent` 是定义 `withdraw` 函数的环境。跟踪 withdraw 的执行，我们可以发现 Python 中 nonlocal 声明的效果：当前执行帧之外的变量可以通过赋值语句更改。

非局部语句（nonlocal statement）会改变 withdraw 函数定义中剩余的所有赋值语句。在将 balance 声明为 nonlocal 后，任何尝试为 balance 赋值的语句，都不会直接在当前帧中寻找并更改 balance，而是找到定义 balance 变量的帧，并在该帧中更新该变量。如果在声明 nonlocal 之前 balance 还没有赋值，则 nonlocal 声明将会报错。

通过改变 balance 的绑定，我们也改变了 withdraw 函数。下一次调用该函数时，变量 balance 的值将会是 15，而不是 20。因此，当我们第二次调用 withdraw 时，返回值将是 12，而不是 17。第一次调用对 balance 的改变会影响到第二次调用的结果。

第二次调用 withdraw 像往常一样创建了第二个局部帧。并且，这两个 withdraw 帧都具有相同的父级帧。也就是说，它们都集成了 make_withdraw 的运行环境，而变量 balance 就是在该环境中定义和声明的。因此，它们都可以访问到 balance 变量的绑定关系。调用 withdraw 会改变当前运行环境，并且影响到下一次调用 withdraw 的结果。nonlocal 声明语句允许 withdraw 更改 make_withdraw 运行帧中的变量。

自从我们第一次遇到嵌套的 def 语句，我们就发现到嵌套定义的函数可以在访问其作用域之外的变量。访问 nonlocal 声明的变量名称并不需要使用非局部语句。相比之下，只有在非局部语句之后，函数才能更改这些帧中名称的绑定。

通过引入非局部语句，我们为赋值语句创建了双重作用。他们可以更改局部绑定 (local bindings)，也可以更改非局部绑定 (nonlocal bindings)。事实上，赋值语句已经有了很多作用：它们可以创建新的变量，也可以为现有变量重新赋值。赋值也可以改变列表和字典的内容。Python 中赋值语句的多种作用可能会使执行赋值语句时的效果变得不太明显。作为程序员，我们有责任清楚地记录代码，以便其他人可以理解赋值的效果。

**Python 特质 (Python Particulars)**。这种非局部赋值模式是具有高阶函数和词法作用域的编程语言的普遍特征。大多数其他语言根本不需要非局部语句。相反，非局部赋值通常是赋值语句的默认行为。

Python 在变量名称查找方面也有一个不常见的限制：在一个函数体内，多次出现的同一个变量名必须处于同一个运行帧内。因此，Python 无法在非局部帧中查找某个变量名对应的值，然后在局部帧中为同样名称的变量赋值，因为同名变量会在同一函数的两个不同帧中被访问。此限制允许 Python 在执行函数体之前预先计算哪个帧包含哪个名称。当代码违反了这个限制时，程序会产生令人困惑的错误消息。为了演示，请参考下面这个删掉了 nonlocal 语句的 make_withdraw 示例。

出现此 UnboundLocalError 是因为 balance 在第 5 行中被赋值，因此 Python 假定对 balance 的所有引用也必须出现在当前帧中。这个错误发生在第 5 行执行之前，这意味着 Python 在执行第 3 行之前，就以某种方式考虑了第 5 行的代码。等我们研究解释器设计的时候，我们就会看到在执行函数体之前预先计算有关函数体的实际情况是很常见的。此时，Python 的预处理限制了 balance 可能出现的帧，从而导致找不到对应的变量名。添加 nonlocal 声明可以修复这个问题。Python 2 中不存在 nonlocal 声明。

## 2.4.5 非局部 Non-local 赋值的好处

非局部 Non-local 赋值对我们意识到程序是由独立、自治的对象组成的至关重要，这些对象相互交互但又各自维护自己的内部状态。

具体来说，非局部赋值使我们能够维护某个函数的局部状态，但是这些状态又会随着对该函数的连续调用而改变。与某个 withdraw 函数关联的 balance 变量在对该函数的多次调用之间共享。但是，程序的其余部分无法访问与 withdraw 实例关联的 balance 变量。只有 wd 与定义它的 make_withdraw 的帧相关联。如果再次调用 make_withdraw，那么它将创建一个单独的帧，其中包含一个单独的 balance 变量。

我们可以扩展我们的示例来说明这一点。第二次调用 make_withdraw 返回具有不同父级的第二个 withdraw 函数。我们将第二个函数绑定到全局帧中的名称 wd2。

现在，我们看到实际上在两个不同的帧中有两个名称 balance 的绑定，并且每个 withdraw 函数都有不同的父级。名称 wd 绑定到 balance 为 20 的函数，而 wd2 绑定到 balance 为 7 的另一个函数。

调用 wd2 会改变其非局部 balance 名称的绑定，但不会影响绑定到名称为 withdraw 的函数。未来对 wd 的调用不受 wd2 的 balance 变化的影响；wd 的 balance 仍然是 20。

这样，每个 withdraw 实例都保持自己的 balance 状态，但程序中的任何其他函数都无法访问该状态。从更高的层面来看这种情况，我们抽象了一个银行账户，它自己管理自己的状态，其行为方式与世界上所有其它账户一样：随着时间推移，账户的状态会根据账户的取款记录而发生变化。

## 2.4.6 非局部 Non-local 赋值的代价

我们的运行环境计算模型解释了非局部赋值的影响。然而，对于变量名和变量值来说，非局部赋值也引入了一些重要的细微差别。

之前的方式中，我们的值没有改变；只有名字和绑定改变了。当两个名称 a 和 b 都绑定到值 4 时，它们绑定到相同的 4 还是不同的 4 并不重要。当然我们知道，只有一个从未改变过的对象，即 4。

但是，具有状态的函数不会以这种方式运行。当两个名称 wd 和 wd2 都绑定到一个 withdraw 函数时，它们是绑定到同一个函数还是绑定到该函数的不同实例是很重要的。考虑以下示例，它与我们刚刚分析的示例形成对比。在这种情况下，调用 wd2 命名的函数确实改变了 wd 命名的函数的值，因为两个名称都引用同一个函数。

两个名字在世界上共同指代同一个值并不罕见，在我们的程序中也是如此。但是，因为值会随时间变化，我们必须非常小心地理解变化对可能引用这些值的其他名称的影响。

正确理解包含 nonlocal 声明的代码的关键是记住：只有函数调用才能引入新帧。赋值语句只能更改现有帧中的绑定关系。在这种情况下，除非 make_withdraw 被调用两次，否则只能有一个 balance 绑定。

**相同与变化 (Sameness and change)。**这些微妙之处的出现是因为，通过引入改变非局部环境的非纯函数，我们改变了表达式的性质。仅包含纯函数调用的表达式是引用透明 (referentially transparent) 的；即如果在函数中，用一个等于子表达式的值来替换子表达式，它的值不会改变。

重新绑定操作违反了引用透明的条件，因为它们不仅仅是返回一个值；他们还会在执行过程中改变运行环境。当我们引入任意的重新绑定时，我们遇到了一个棘手的认识论问题：两个值相同意味着什么。在我们的计算环境模型中，两个单独定义的函数是不同的，因为对一个函数的更改可能不会反映在另一个函数中。

通常，只要我们从不修改数据对象，我们就可以将数据对象视为其各个部分的总和。例如，有理数是通过给出其分子和分母来确定的。但这种观点在变化的存在下不再有效，其中数据对象和组成它的部分各有各自的“身份”。即使我们通过 withdraw 改变了 balance，银行账户仍然是“同一个”银行账户；相反，我们可以有两个银行账户，它们恰好具有相同的 balance，但是是不同的对象。

尽管它带来了复杂性，但非局部赋值是创建模块化程序的强大工具。程序的不同部分对应于不同的环境帧，可以在整个程序执行过程中单独发展。此外，使用具有局部状态的函数，我们能够实现可变数据类型。事实上，我们可以实现抽象的数据类型，这些数据类型等效于上面介绍的内置列表和 dict 类型。

## 2.4.7 列表和字典实现

Python 语言并不让我们直接访问列表的实现细节，而只提供了语言内置的可以变更数据的方法。为了理解如何使用具有局部状态的函数来表示可变列表，我们现在将开发一个可变链表的实现。

我们将用一个将链表作为其局部状态的函数来表示一个可变链表。像所有可变数据一样，列表需要有一个唯一标识。具体来说，我们不能使用 None 来表示一个空的可变列表，因为两个空列表是不同的（例如，为某一个追加元素，而另一个保持不变），但 None 始终是 None。但是，两个独立的函数则不一样，即使他们的局部状态都是 empty，这两个函数也是不同的两个对象。

如果可变链表是一个函数，它需要什么参数？答案展示了编程中的一般模式：函数是一个 dispatch （调度）函数，其参数首先是一个期望的指令，代表期望这个函数做什么；然后是该方法的需要用到的参数。此指令是一个字符串，用于命名函数应执行的操作。可以将这个 dispatch 函数理解为多个不同函数的抽象：第一个参数确定目标函数的行为，并为该行为入参其他参数。

我们的可变列表将响应五种不同的消息：len、getitem、push_first、pop_first 和 str。前两个实现序列抽象的行为。接下来的两个添加或删除列表的第一个元素。最后一条返回整个链表的字符串表示形式。

```
>>> def mutable_link():
        """返回一个可变链表的函数"""
        contents = empty
        def dispatch(message, value=None):
            nonlocal contents
            if message == 'len':
                return len_link(contents)
            elif message == 'getitem':
                return getitem_link(contents, value)
            elif message == 'push_first':
                contents = link(value, contents)
            elif message == 'pop_first':
                f = first(contents)
                contents = rest(contents)
                return f
            elif message == 'str':
                return join_link(contents, ", ")
        return dispatch
```

我们还可以添加一个辅助函数，用已知的列表来构建一个链表，只需按相反的顺序添加每个元素即可。

```
>>> def to_mutable_link(source):
        """返回一个与原列表相同内容的函数列表"""
        s = mutable_link()
        for element in reversed(source):
            s('push_first', element)
        return s
```

在上面的定义中，函数 reversed 接受并返回一个可遍历的值；这是另一个列表操作函数的示例。

至此，我们就可以构造一个功能实现的可变链表了。请注意，链表本身是一个函数。

```
>>> s = to_mutable_link(suits)
>>> type(s)
<class 'function'>
>>> print(s('str'))
heart, diamond, spade, club
```

此外，我们可以将消息传递给 list 以更改其内容，例如删除第一个元素。

```
>>> s('pop_first')
'heart'
>>> print(s('str'))
diamond, spade, club
```

原则上，操作 push_first 和 pop_first 足以对列表进行任意更改。我们可以完全清空列表，然后用想要的结果替换它的旧内容。

**消息传递 (Message passing)**。我们可以实现 Python 列表的许多有用的数据变更操作，例如 extend（延长链表）和 insert（在特定位置插入）。同时我们有一个选择：我们可以将它们全部实现为函数，利用现有的消息 pop_first 和 push_first 进行实现。或者，我们可以在 dispatch 函数体中添加额外的 elif 子句，每个子句检查一条消息（例如，'extend' ）并直接对内容应用适当的更改。

第二种方法将对数据值的所有操作的逻辑封装在一个响应不同消息的函数中，是一种称为消息传递的能力。使用消息传递的程序定义了调度函数，每个函数都可能具有局部状态，并通过将"消息"作为第一个参数传递给这些函数来组织计算。消息是对应于特定行为的字符串。

**字典实现 (Implementing Dictionaries)**。我们还可以实现一个具有与字典类似行为的值。在这种情况下，我们使用键值对列表来存储字典的内容。每对都是一个双元素列表。

```
>>> def dictionary():
        """返回一个字典的函数实现"""
        records = []
        def getitem(key):
            matches = [r for r in records if r[0] == key]
            if len(matches) == 1:
                key, value = matches[0]
                return value
        def setitem(key, value):
            nonlocal records
            non_matches = [r for r in records if r[0] != key]
            records = non_matches + [[key, value]]
        def dispatch(message, key=None, value=None):
            if message == 'getitem':
                return getitem(key)
            elif message == 'setitem':
                setitem(key, value)
        return dispatch
```

同样，我们使用消息传递方法来组织我们的实现。支持两个消息：getitem 和 setitem。要为键插入值，我们会过滤掉任何具有给定键的现有记录，然后再添加一个。这样就可以确保每个键在记录中只出现一次。为了查找键的值，我们过滤与给定键匹配的记录。我们现在可以使用该实现来存储和检索值。

```
>>> d = dictionary()
>>> d('setitem', 3, 9)
>>> d('setitem', 4, 16)
>>> d('getitem', 3)
9
>>> d('getitem', 4)
16
```

这种字典的实现并未针对快速记录查找进行优化，每次查找都必须遍历所有数据。内置字典类型的效率要高得多。它的实现方式超出了本文的范围。

## 2.4.8 调度字典（Dispatch Dictionaries）

`dispatch` 函数是实现抽象数据消息传递接口的通用方法。为实现消息分发，到目前为止，我们使用条件语句将消息字符串与一组固定的已知消息进行比较。

内置字典数据类型提供了一种查找键值的通用方法。我们可以使用带有字符串键的字典，而不是使用条件来实现调度。

下面的 account 数据是用字典实现的。它有一个构造器 amount 和选择器 check_balance，以及存取资金的功能。此外，帐户的局部状态与实现其行为的函数一起存储在字典中。

在 amount 的方法声明中，使用字典声明了一个 dispatch 变量并将其返回，该字典包含一个帐户可能被操作的各种情况。balance 是一个数字，而消息存款 deposit 和取款 withdraw 则是两个函数。这些函数可以访问 dispatch 字典，因此它们可以读取和更改 balance。通过将 balance 存储在 dispatch 字典中而不是直接存储在帐户帧中，我们避免了在 deposit 和 withdraw 函数中使用 nonlocal 声明。

运算符 += 和 -= 是 Python（和许多其他语言）中先计算，后赋值两个操作的简写。下面的最后两行是等效的。

```
>>> a = 2
>>> a = a + 1
>>> a += 1
```

## 2.4.9 约束传递 (Propagating Constraints)

可变数据让我们能够模拟具有各种变化的系统，也允许我们构建新的抽象类型。在这个扩展示例中，我们结合了非局部赋值、列表和字典来构建一个支持多方向计算的基于约束的系统。将程序表示为约束是一种声明式编程，在这种编程中，程序员声明要解决的问题的框架结构，而不是具体的实现细节。

计算机程序传统上被组织为单向计算，它对预先指定的参数执行操作以产生所需的输出。另一方面，我们通常希望根据数量之间的关系对系统进行建模。例如，我们之前考虑过理想气体状态方程，它通过玻尔兹曼常数 (k) 将理想气体的压力 (p)、体积 (v)、数量 (n) 和温度 (t) 联系起来：

```
p * v = n * k * t
```

这样的方程不是单向的。给定任何四个量，我们可以使用这个方程来计算第五个。然而，将方程式翻译成传统的计算机语言会迫使我们选择一个量来根据其他四个量进行计算。因此，计算压力的函数不能用于计算温度，即使这两个量的计算来自同一个方程。

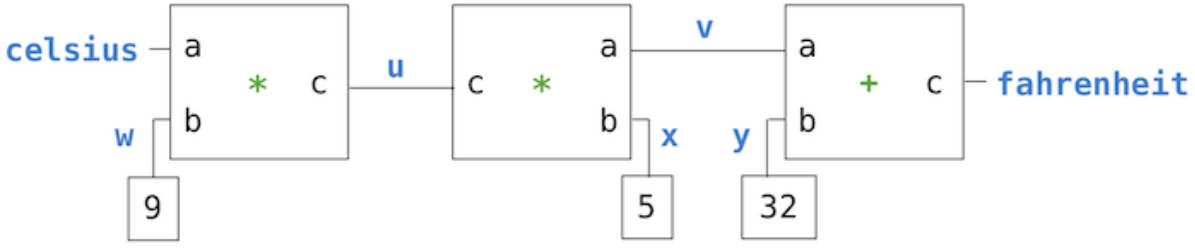
在本节中，我们概述了线性关系的一般模型的设计。我们定义了在数量之间保持的原始约束，例如强制数学关系 a + b = c 的 adder(a, b, c) 约束。

我们还定义了一种组合方式，以便可以组合原始约束来表达更复杂的关系。这样，我们的程序就类似于一种编程语言。我们通过构建一个网络来组合约束，在该网络中约束由连接器 (connector) 连接。连接器是一个对象，它“持有”一个值并且可以参与一个或多个约束。

例如，我们知道华氏温度和摄氏温度之间的关系是：

```
9 * c = 5 * (f - 32)
```

该等式是 c 和 f 之间的复杂约束。这样的约束可以被认为是一个由原始加法器 (adder) 、乘法器 (multiplier) 和常量 (constant) 约束组成的网络。



在此图中，我们在左侧看到一个乘法器，其中包含三个端口，标记为 a、b 和 c。这些将乘数连接到网络的其余部分，如下所示：终端连接到连接器 celsius，该连接器将保持摄氏温度。b 端口连接到连接器 w，该连接器链接到常量 9。乘法器约束为 a 和 b 乘积的 c 端链接到另一个乘法器的 c 端，该乘法器的 b 连接到常量 5，它的 a 又连接到加法器的某一项。

这个网络的计算过程如下：当一个连接器被赋予一个值时（由用户或由它链接到的约束框），它会唤醒所有相关的约束（除了刚刚唤醒它的约束）以告诉他们它有值。每个被唤醒的约束框之后轮流询问其连接器，以查看是否有足够的信息来确定连接器的值。如果有，该框设置该连接器，然后唤醒所有关联的约束，依此类推。例如，在摄氏度和华氏度之间的转换中，w、x 和 y 立即被常量框分别设置为 9、5 和 32。连接器唤醒乘法器和加法器，它们确定没有足够的信息继续进行。如果用户（或网络的其他部分）将摄氏连接器设置为一个值（比如 25），最左边的乘法器将被唤醒，它将 u 设置为 25 * 9 = 225。然后你唤醒第二个乘法器，将 v 设置为 45，v 唤醒加法器，将 fahrenheit 连接器设置为 77。

**使用约束系统 (Using the Constraint System)。**要使用约束系统执行上述温度计算，我们首先通过调用连接器 connector 创建两个命名连接器，摄氏度 celsius 和华氏度 fahrenheit。

```
>>> celsius = connector('Celsius')
>>> fahrenheit = connector('Fahrenheit')
```

然后，我们将这些连接器链接到一个反映上图的网络中。函数转换器 (converter) 组装网络中的各种连接器和约束。

```
>>> def converter(c, f):
        """用约束条件连接 c 到 f，将摄氏度转换为华氏度."""
        u, v, w, x, y = [connector() for _ in range(5)]
        multiplier(c, w, u)
        multiplier(v, x, u)
        adder(v, y, f)
        constant(w, 9)
        constant(x, 5)
        constant(y, 32)
```

```
>>> converter(celsius, fahrenheit)
```

我们将使用消息传递系统来协调约束和连接器。约束是不包含局部状态的字典。它们对消息的响应是非纯函数，会更改它们约束的连接器。

连接器是保存当前值并响应操纵该值的消息的字典。约束不会直接更改连接器的值，而是通过发送消息来更改，以便连接器可以通知其他约束以响应更改。这样，一个连接器既代表了一个数字，同时也封装了连接器的行为。

我们可以发送给连接器一条消息来设置它的值。在这里，我们（“user”）将 celsius 的值设置为 25。

```
>>> celsius['set_val']('user', 25)
Celsius = 25
Fahrenheit = 77.0
```

不仅 celsius 的值变为 25，而且它的值通过网络传播，因此 fahrenheit 的值也发生变化。打印这些更改是因为我们在构造它们时命名了这两个连接器。

现在我们可以尝试将 fahrenheit 度设置为一个新值，比如 212。

```
>>> fahrenheit['set_val']('user', 212)
Contradiction detected: 77.0 vs 212
```

连接器报告说它察觉到了一个矛盾：它的值为 77.0，而有人试图将它设置为 212。如果我们真的想用新值应用到网络，我们可以告诉 celsius 忘记它的旧值：

```
>>> celsius['forget']('user')
Celsius is forgotten
Fahrenheit is forgotten
```

连接器 celsius 发现最初设置其值的用户现在收回该值，因此 celsius 同意失去其值，并将这一事实通知网络的其余部分。这个信息最终传播到 fahrenheit，它现在发现它没有理由继续相信它自己的值是 77。因此，它也放弃了它的值。

现在 fahrenheit 没有值，我们可以将其设置为 212：

```
>>> fahrenheit['set_val']('user', 212)
Fahrenheit = 212
Celsius = 100.0
```

这个新值在通过网络传播时会迫使 celsius 的值变为 100。我们使用了完全相同的网络来计算给定 celsius 的 fahrenheit 和给定 fahrenheit 的 celsius。这种计算的非方向性是基于约束的系统的显着特征。

**实现约束系统 (Implementing the Constraint System)。**正如我们所见，连接器是将消息名称映射到函数和数据值的字典。我们将实现响应以下消息的连接器：

```
>>> connector ['set_val'](source, value)  """表示 source 在请求连接器将当前值设为 value"""
>>> connector ['has_val']()  """返回连接器是否已经具有值"""
>>> connector ['val']  """是连接器的当前值"""
>>> connector ['forget'](source)  """告诉连接器 source 请求遗忘它的值"""
>>> connector ['connect'](source)  """告诉连接器参与新的约束，即 source"""
```

约束也是字典，它通过两条消息从连接器接收信息：

```
>>> constraint['new_val']()  """表示与约束相连的某个连接器具有新的值。"""
>>> constraint['forget']()  """表示与约束相连的某个连接器遗忘了值。"""
```

当约束收到这些消息时，它们会将消息传播到其他连接器。

adder 函数在三个连接器上构造一个加法器约束，其中前两个必须与第三个相加：a + b = c。为了支持多向约束传播，加法器还必须指定它从 c 中减去 a 得到 b，同样地从 c 中减去 b 得到 a。

```
>>> from operator import add, sub
>>> def adder(a, b, c):
        """约束 a+b=c"""
        return make_ternary_constraint(a, b, c, add, sub, sub)
```

我们想实现一个通用的三元（三向）约束，它使用来自 adder 的三个连接器和三个函数来创建一个接受 new_val 和 forget 消息的约束。对消息的响应是局部函数，它们被放置在名为 constraint 的字典中。

```
>>> def make_ternary_constraint(a, b, c, ab, ca, cb):
        """约束 ab(a,b)=c，ca(c,a)=b，cb(c,b)=a"""
        def new_value():
            av, bv, cv = [connector['has_val']() for connector in (a, b, c)]
            if av and bv:
                c['set_val'](constraint, ab(a['val'], b['val']))
            elif av and cv:
                b['set_val'](constraint, ca(c['val'], a['val']))
            elif bv and cv:
                a['set_val'](constraint, cb(c['val'], b['val']))
        def forget_value():
            for connector in (a, b, c):
                connector['forget'](constraint)
        constraint = {'new_val': new_value, 'forget': forget_value}
        for connector in (a, b, c):
            connector['connect'](constraint)
        return constraint
```

字典 constraint 是一个调度字典，也是约束对象本身。它本身可以响应约束接收到的两条消息，但也作为 source 参数传给连接器。

每当约束被告知其连接器之一具有值时，就会调用约束的局部函数 new_value。该函数首先检查 a 和 b 是否都有值。如果是，它告诉 c 将其值设置为函数 ab 的返回值，在加法器的情况下为 add。约束将自身（constraint）作为连接器的 source 参数传递，该连接器是加法器对象。如果 a 和 b 不同时都有值，则约束检查 a 和 c，依此类推。

如果约束被告知它的一个连接器遗忘了它的值，它会请求它的所有连接器遗忘它们的值。（实际上只有那些由此约束设置的值会丢失。）

乘法器与加法器非常相似。

```
>>> from operator import mul, truediv
>>> def multiplier(a, b, c):
        """约束 a*b=c"""
        return make_ternary_constraint(a, b, c, mul, truediv, truediv)
```

常量也是一种约束，但它永远不会发送任何消息，因为它只涉及它在构造时设置的单个连接器。

```
>>> def constant(connector, value):
        """常量赋值"""
        constraint = {}
        connector['set_val'](constraint, value)
        return constraint
```

这三个约束足以实现我们的温度转换网络。

**连接器表示 (Representing connectors)。**连接器也是字典，其中包含一组值，也包括有局部状态的响应函数。连接器必须跟踪 `informant` 变量，它提供了当前的值，以及它参与的 constraints 列表。

函数 connector 具有用于设置和遗忘值的局部函数，这些值是对来自约束的消息的响应。

```
>>> def connector(name=None):
        """限制条件之间的连接器"""
        informant = None
        constraints = []
```

```
        def set_value(source, value):
            nonlocal informant
            val = connector['val']
            if val is None:
                informant, connector['val'] = source, value
                if name is not None:
                    print(name, '=', value)
                inform_all_except(source, 'new_val', constraints)
            else:
                if val != value:
                    print('Contradiction detected:', val, 'vs', value)
        def forget_value(source):
            nonlocal informant
            if informant == source:
                informant, connector['val'] = None, None
                if name is not None:
                    print(name, 'is forgotten')
                inform_all_except(source, 'forget', constraints)
        connector = {'val': None,
                     'set_val': set_value,
                     'forget': forget_value,
                     'has_val': lambda: connector['val'] is not None,
                     'connect': lambda source: constraints.append(source)}
        return connector
```

连接器也是约束用于与连接器通信的五个消息的调度字典。四个响应是函数，最后的响应是值本身。

当有设置连接器值的请求时调用局部函数 set_value。如果连接器当前没有值，它将设置它的值并记住请求设置值的源约束作为 informant。然后连接器将通知除了请求设置值约束以外的所有参与的约束。这是使用以下迭代函数完成的。

```
>>> def inform_all_except(source, message, constraints):
        """告知信息除了 source 外的所有约束条件"""
        for c in constraints:
            if c != source:
                c[message]()
```

如果要求连接器遗忘其值，它会调用局部函数 forget-value，该函数首先检查以确保请求来自与最初设置值相同的约束。如果是这样，连接器会通知其关联的约束该值的丢失情况。

对消息 `has_val` 的响应表明当前连接器是否有值。对消息 `connect` 的响应会将源约束添加到约束列表中。

我们设计的约束程序引入了许多将在面向对象编程中再次出现的思想。约束和连接器都是通过消息操作的抽象。当连接器的值发生变化时，它会通过一条消息进行更改，该消息不仅会更改值，还会验证它（检查源）并传播其效果（通知其他约束）。事实上，我们将在本章后面使用具有字符串值键和函数值的字典的类似架构来实现面向对象的系统。