2.2 数据抽象

::: details INFO
译者：Mancuoj

来源：2.2 Data Abstraction

对应：无
:::

当我们希望在程序中表示世界上广泛的事物时，会发现它们中的大多数都具有复合结构。比如地理位置具有经纬度坐标。为了表示位置，我们希望我们的编程语言能够经度和纬度耦合在一起形成一对复合数据，使它能够作为单个概念单元被程序操作，同时也能作为可以单独考虑的两个部分。

使用复合数据可以使程序更加模块化。如果我们能够将地理位置作为整体值进行操作，那么我们就可以将计算位置的程序部分与位置如何表示的细节隔离开来，这种将“数据表示”与“数据处理”的程序隔离的通用技术是一种强大的设计方法，称为数据抽象。数据抽象会使程序更易于设计、维护和修改。

数据抽象与函数抽象类似。当我们创建一个函数抽象时，函数实现的细节可以被隐藏，而特定的函数本身可以被替换为具有相同整体行为的任何其他函数。换句话说，我们可以创建一个抽象来将函数的使用方式与实现细节分离。类似地，数据抽象可以将复合数据值的使用方式与其构造细节隔离开来。

数据抽象的基本思想是构建程序，以便它们对抽象数据进行操作。也就是说，我们的程序应该以尽可能少的假设来使用数据，同时要将具体的数据表示定义为程序的独立部分。

程序的“操作抽象数据”和“定义具体表示”两个部分，会由一组根据具体表示来实现抽象数据的函数相连。为了说明该技术，我们将思考如何设计一组用于操作有理数的函数。

2.2.1 示例：有理数

有理数是整数的比值，并且有理数是实数的一个重要子类。 `1/3` 或 `17/29` 等有理数通常写为：

```
<分子>/<分母>
```

其中 `<分子>` 和 `<分母>` 都是整数值的占位符，这两个部分能够准确表示有理数的值。实际上的整数除以会产生 `float` 近似值，失去整数的精确精度。

```
>>> 1/3
0.3333333333333333
>>> 1/3 == 0.33333333333333300000  # 整数除法得到近似值
True
```

但是，我们可以通过将分子和分母组合在一起来创建有理数的精确表示。

通过使用函数抽象，我们可以在实现程序的某些部分之前开始高效地编程。我们首先假设已经存在了一个从分子和分母构造有理数的方法，再假设有方法得到一个给定有理数的分子和分母。进一步假设得到以下三个函数：

- `rational(n, d)` 返回分子为 `n`、分母为 `d` 的有理数
- `numer(x)` 返回有理数 `x` 的分子
- `denom(x)` 返回有理数 `x` 的分母

我们在这里使用了一个强大的程序设计策略：一厢情愿（wishful thinking）。即使我们还没有想好有理数是如何表示的，或者函数 `numer`、`denom` 和 `rational` 应该如何实现。但是如果我们确实定义了这三个函数，我们就可以进行加法、乘法、打印和测试有理数是否相等：

```
>>> def add_rationals(x, y):
        nx, dx = numer(x), denom(x)
        ny, dy = numer(y), denom(y)
        return rational(nx * dy + ny * dx, dx * dy)

>>> def mul_rationals(x, y):
        return rational(numer(x) * numer(y), denom(x) * denom(y))

>>> def print_rational(x):
        print(numer(x), '/', denom(x))

>>> def rationals_are_equal(x, y):
        return numer(x) * denom(y) == numer(y) * denom(x)
```

现在我们有了选择器函数 `numer` 和 `denom` 以及构造函数 `rational` 定义的有理数运算，但还没有定义这些函数。我们需要某种方法将分子和分母粘合在一起形成一个复合值。

2.2.2 对

为了使我们能够实现具体的数据抽象，Python 提供了一个名为 `list` 列表的复合结构，可以通过将表达式放在以逗号分隔的方括号内来构造。这样的表达式称为列表字面量。

```
>>> [10, 20]
[10, 20]
```

可以通过两种方式访问 列表元素。第一种方法是通过我们熟悉的多重赋值方法，它将列表解构为单个元素并将每个元素与不同的名称绑定。

```
>>> pair = [10, 20]
>>> pair
[10, 20]
>>> x, y = pair
>>> x
10
>>> y
20
```

访问列表中元素的第二种方法是通过元素选择运算符，也使用方括号表示。与列表字面量不同，直接跟在另一个表达式之后的方括号表达式不会计算为 `list` 值，而是从前面表达式的值中选择一个元素。

```
>>> pair[0]
10
>>> pair[1]
20
```

Python 中的列表（以及大多数其他编程语言中的序列）是从 0 开始索引的，这意味着索引 0 选择第一个元素，索引 1 选择第二个元素，以此类推。对于这种索引约定的一种直觉是，索引表示元素距列表开头的偏移量。

元素选择运算符的等效函数称为 `getitem`，它也使用 0 索引位置从列表中选择元素。

```
>>> from operator import getitem
>>> getitem(pair, 0)
10
>>> getitem(pair, 1)
20
```

双元素列表并不是 Python 中表示对的唯一方法。将两个值捆绑在一起成为一个值的任何方式都可以被认为是一对。列表是一种常用的方法，它也可以包含两个以上的元素，我们将在本章后面进行探讨。

代表有理数：我们现在可以将有理数表示为两个整数的对：一个分子和一个分母。

```
>>> def rational(n, d):
        return [n, d]

>>> def numer(x):
        return x[0]

>>> def denom(x):
        return x[1]
```

连同之前定义的算术运算，我们可以使用我们定义的函数来操作有理数。

```
>>> half = rational(1, 2)
>>> print_rational(half)
1 / 2
>>> third = rational(1, 3)
>>> print_rational(mul_rationals(half, third))
1 / 6
>>> print_rational(add_rationals(third, third))
6 / 9
```

如上面的示例所示，我们的有理数实现不会将有理数简化为最小项。可以通过更改 `rational` 的实现来弥补这个缺陷。如果我们有一个计算两个整数的最大公分母的函数，我们可以用它在构造对之前将分子和分母减少到最低项。与许多有用的工具一样，这样的功能已经存在于 Python 库中。

```
>>> from fractions import gcd
>>> def rational(n, d):
        g = gcd(n, d)
        return (n//g, d//g)
```

`//` 表示整数除法，它会将除法结果的小数部分向下舍入。因为我们知道 `g` 会将 `n` 和 `d` 均分，所以在这种情况下整数除法是精确的。这个修改后的 `rational` 实现会确保有理数以最小项表示。

```
>>> print_rational(add_rationals(third, third))
2 / 3
```

这种改进是通过更改构造函数而不更改任何实现实际算术运算的函数来实现的。

2.2.3 抽象屏障

在继续更多复合数据和数据抽象的示例之前，让我们考虑一下有理数示例引发的一些问题。我们根据构造函数 `rational` 和选择器函数 `numer` 和 `denom` 来定义操作。一般来说，数据抽象的基本思想是确定一组基本操作，根据这些操作可以表达对某种值的所有操作，然后仅使用这些操作来操作数据。通过以这种方式限制操作的使用，在不改变程序行为的情况下改变抽象数据的表示会容易得多。

对于有理数，程序的不同部分使用不同的操作来处理有理数，如此表中所述。

该程序的一部分...	**把有理数当作...**	**仅使用...**
使用有理数进行计算	整个数据值	`add_rational, mul_rational, rationals_are_equal, print_rational`
创建有理数或操作有理数	分子和分母	`rational, numer, denom`
为有理数实现选择器和构造器	二元列表	列表字面量和元素选择

在上面的每一层中，最后一列中的函数会强制实施抽象屏障（abstraction barrier）。这些功能会由更高层次调用，并使用较低层次的抽象实现。

当程序中有一部分本可以使用更高级别函数但却使用了低级函数时，就会违反抽象屏障。例如，计算有理数平方的函数最好用 `mul_rational` 实现，它不对有理数的实现做任何假设。

```
>>> def square_rational(x):
        return mul_rational(x, x)
```

直接引用分子和分母会违反一个抽象屏障。

```
>>> def square_rational_violating_once(x):
        return rational(numer(x) * numer(x), denom(x) * denom(x))
```

假设有理数会表示为双元素列表将违反两个抽象屏障。

```
>>> def square_rational_violating_twice(x):
        return [x[0] * x[0], x[1] * x[1]]
```

抽象屏障使程序更易于维护和修改。依赖于特定表示的函数越少，想要更改该表示时所需的更改就越少。计算有理数平方的所有这些实现都具有正确的行为，但只有第一个函数对未来的更改是健壮的。即使我们修改了有理数的表示，`square_rational` 函数也不需要更新。相比之下，当选择器函数或构造函数签名发生变化后，`square_rational_violating_once` 就需要更改，而只要有理数的实现发生变化，`square_rational_violating_twice` 就需要更新。

2.2.4 数据的属性

抽象屏障塑造了我们思考数据的方式。有理数的表示不限于任何特定的实现（例如二元素列表）；它就是由 `rational` 返回的值，然后可以传递给 `numer` 和 `denom` 。此外，构造器和选择器之间必须保持适当的关系。也就是说，如果我们从整数 `n` 和 `d` 构造一个有理数 `x` ，那么 `numer(x)/denom(x)` 应该等于 `n/d` 。

通常，我们可以使用选择器和构造器的集合以及一些行为条件来表达抽象数据。只要满足行为条件（比如上面的除法属性），选择器和构造器就构成了一种数据的有效表示。抽象屏障下的实现细节可能会改变，但只要行为没有改变，那么数据抽象就仍然有效，并且使用该数据抽象编写的任何程序都将保持正确。

这种观点可以广泛应用，包括我们用来实现有理数的对。我们从来没有真正谈论什么是一对，只是语言提供了创建和操作二元列表的方法。我们需要实现一对的行为是它将两个值粘合在一起。作为一种行为条件，

- 如果一对 `p` 由值 `x` 和 `y` 构成，则 `select(p, 0)` 返回 `x`， `select(p, 1)` 返回 `y`

我们实际上并不一定需要 `list` 类型来创建对，作为替代，我们可以用两个函数 `pair` 和 `select` 来实现这个描述以及一个二元列表。

```
>>> def pair(x, y):
        """Return a function that represents a pair."""
        def get(index):
            if index == 0:
                return x
            elif index == 1:
                return y
        return get

>>> def select(p, i):
        """Return the element at index i of pair p."""
        return p(i)
```

通过这个实现，我们可以创建和操作对。

```
>>> p = pair(20, 14)
>>> select(p, 0)
20
>>> select(p, 1)
14
```

这种高阶函数的使用完全不符合我们对数据应该是什么的直觉概念。但尽管如此，这些函数足以在我们的程序中表示对，也足以表示复合数据。

这种表示对的函数表示的重点并不是 Python 实际上以这种方式工作（出于效率原因，列表更直接地实现），而是它可以以这种方式工作。函数表示虽然晦涩难懂，但却是表示对的一个完全合适的方法，因为它满足了表示对需要满足的唯一条件。数据抽象的实践使我们能够轻松地在表示之间切换。