2.7 对象抽象

::: details INFO
译者：Jesper.Y

来源：2.7 Object Abstraction

对应：Ants、Disc 09
:::

对象系统允许程序员更高效地建立并使用抽象数据描述。其也设计为允许在同一个程序中存在多种抽象数据表现形式。

对象抽象的一个核心概念就是泛型函数，这种函数能够接受多种不同类型的值。我们将思考三种不同的用于实现泛型函数的技术：共享接口，类型派发和类型强制转换。在建立这些概念的过程中，我们也会发现一些 Python 对象系统的特性，这些特性支持泛型函数的创建。

2.7.1 字符串转换

为了高效地展示数据，一个对象值应该像它代表的数据一样进行行为，包括产生一个它自己的字符串表示。在像 Python 这样的交互式语言中，数据值的字符串表示是特别重要的，在交互式会话中，它可以自动地展示表达式的值的字符串形式。

字符串值为人们提供了一种相互传递信息的基本媒介。字符序列可以渲染在屏幕上、打印在纸上、大声阅读出来、转换成盲文或者以莫斯码广播。字符串也是编程的基础，因为他们可以表示 Python 表达式。

Python 规定所有的对象都应该生成两个不同的字符串表示：一种是人类可读的文本，另一种是 Python 可解释的表示式。字符串的构造函数，即 `str`，返回一个人类可读的字符串。如果可能，`repr` 函数返回一个 Python 可解释的表达式，该表达式的求值结果与原对象相同。`repr` 的文档字符串（docstring）解释了这个特性：

> *repr(object) -> string*
>
> *Return the canonical string representation of the object.*
>
> *For most object types, eval(repr(object)) == object*

> 返回对象的标准字符串表示。
>
> 对于大多数对象类型，`eval(repr(object)) == object`。

对于表达式的值调用 `repr` 的结果就是 Python 在交互式会话中所打印的内容。

```
>>> 12e12
120000000000000.0
>>> print(repr(12e12))
120000000000000.0
```

在一些情况下，不存在对原始值的字符串表示时，Python 通常生成一个被尖括号包围的描述。

```
>>> repr(min)
'<built-in function min>'
```

`str` 构造器通常与 `repr` 行为一致，但是在某些情况下它会提供一个更容易解释的文本表示。例如，我们可以看到 `str` 和 `repr` 对于日期对象的不同展示。

```
>>> from datetime import date
>>> tues = date(2011, 9, 12)
>>> repr(tues)
'datetime.date(2011, 9, 12)'
>>> str(tues)
'2011-09-12'
```

定义 `repr` 函数带来了一个新的挑战：我们想要它正确地应用于所有的数据类型，即使是那些实现 `repr` 时还不存在的类型。我们希望它是一个通用的或者多态（polymorphic）的函数，可以被应用于数据的多种（多）不同形式（态）。

在这情况下，对象系统提供了一种优雅的解决方案：`repr` 函数总是在其参数值上调用一个名为 `__repr__` 的方法。

```
>>> tues.__repr__()
'datetime.date(2011, 9, 12)'
```

通过在用户定义类中实现这个相同的方法，我们可以将 `repr` 函数的适用范围扩展到将来我们创建的任何类。这个例子突出了点表达式的另一个优势，那就是它们提供了一种机制，可以把现有的函数的作用域扩展到新的对象类型。

`str` 构造器以类似的方式实现：它在其参数值上调用一个名为 `__str__` 的方法。

```
>>> tues.__str__()
'2011-09-12'
```

这些能够应对多种类型的函数（多态函数）是一个更通用原则的例子：某些函数应该能够适用于多种数据类型。此外，创建这样一个函数的一种方式是使用在每个类中都有不同定义的共享属性名称，这就意味着这些函数在不同的类中会有不同的行为。

2.7.2 专用方法

在 Python 中，某些特殊名称会在特殊情况下被 Python 解释器调用。例如，类的 `__init__` 方法会在对象被创建时自动调用。`__str__` 方法会在打印时自动调用，`__repr__` 方法会在交互式环境显示其值的时候自动调用。

在 Python 中有一些为其他行为而准备的特殊名称。下面介绍其中某些常用的。

真值（True）和假值（False）。我们之前已经看到，数字类型在 Python 中拥有真值：更准确地说，0 是一个假值而其他所有数字都是真值。实际上 Python 中的所有对象都拥有真假值。默认情况下，用户定义类的对象被认为是真值，但是专门的 `__bool__` 方法可以用于覆盖这种行为。如果一个对象定义了 `__bool__` 方法，那么 Python 就会调用这个方法来确定它的真假值。

举一个例子，假设我们想让一个只有 0 存款的账号为假值。我们可以为 `Account` 添加一个 `__bool__` 方法来实现这种行为。

```
>>> Account.__bool__ = lambda self: self.balance != 0
```

我们可以调用 `bool` 构造器来看一个对象的真假值，同时我们也可以在布尔上下文中使用任何对象。

```
>>> bool(Account('Jack'))
False
>>> if not Account('Jack'):
        print('Jack has nothing')
Jack has nothing
```

序列操作。我们已经知道使用 `len` 函数可以确定序列的长度。

```
>>> len('Go Bears!')
9
```

`len` 函数调用了它的参数的 `__len__` 方法来确定其长度。所有的内置序列类型都实现了这个方法。

```
>>> 'Go Bears!'.__len__()
9
```

如果序列没有提供 `__bool__` 方法，那么 Python 会使用序列的长度来确定其真假值。空的序列是假值，而非空序列是真值。

```
>>> bool('')
False
>>> bool([])
False
>>> bool('Go Bears!')
True
```

`__getitem__` 方法由元素选择操作符调用，但也可以直接调用它。

```
>>> 'Go Bears!'[3]
'B'
>>> 'Go Bears!'.__getitem__(3)
'B'
```

可调用对象。在 Python 中函数是一等对象，因此它们被作为数据进行传递，并且像其他对象那样拥有属性。Python 还允许我们定义像函数一样可以被“调用的对象”，只要在对象中包含一个 `__call__` 方法。通过这个方法，我们可以定义一个行为像高阶函数的类。

举个例子，思考下面这个高阶函数，它返回一个函数，这个函数将一个常量值加到其参数上。

```
>>> def make_adder(n):
        def adder(k):
            return n + k
        return adder

>>> add_three = make_adder(3)
>>> add_three(4)
7
```

我们可以创建一个 `Adder` 类，定义一个 `__call__` 方法来提供相同的功能。

```
>>> class Adder(object):
        def __init__(self, n):
            self.n = n
        def __call__(self, k):
            return self.n + k
>>> add_three_obj = Adder(3)
>>> add_three_obj(4)
7
```

`Adder` 类表现的就像 `make_adder` 高阶函数，而 `add_three_obj` 表现得像 `add_three`。我们进一步模糊了数据和函数之间的界限。

算术运算。特定的方法也可以定义应用在用户定义的对象上的内置操作符的行为。为了提供这种通用性，Python 遵循特定的协议来应用每个操作符。例如，为了计算包含 `+` 操作符的表达式，Python 会检查操作符左右两侧的运算对象上是否有特定的方法。首先 Python 在左侧运算对象上检查其是否有 `__add__` 方法，然后在右侧运算对象上检查其是否有 `__radd__` 方法。如果找到了其中一个方法，就会将另一个运算对象的值作为它的参数来调用这个方法。下面的章节提供了一些示例。对于对其中进一步的细节感兴趣的读者，Python 文档详尽地描述了 运算符的方法名称。《Dive into Python 3》有一个章节介绍了 特殊方法名称，其中描述了这些特殊方法名称的使用方式。

2.7.3 多重表示

抽象障碍允许我们分离数据的使用和表示。然而在大型程序中，讨论数据类型的“底层表示”可能并不总是有意义。首先，一个数据对象可能由不止一种有用的表示，我们也许会想要设计能够处理多种表示形式的系统。

以一个简单的例子而言，复数可以用两种几乎相同的方式来表示：直角坐标系（实部和虚部）和极坐标系（幅度和角度）。有时直角坐标系更合适而有时极坐标系更合适。事实上，我们可以想象这样一个系统，复数在其中同时以两种形式表示，并且操作复数的函数可以处理任何一种表现形式。我们接下来实现这样是一个系统。需要注意的是，我们正在开发一个系统，这个系统使用通用操作符对复数进行数学运算，以此作为一个简单但不切实的示例程序。复数类型 已经内置在 Python 中，但为了示例我们将实现自己的复数类型。

允许数据多重表示的想法经常出现。大型的软件系统通常是由许多人长时间工作而设计的，同时受随时间变化的需求的影响。在这样的环境下，不可能让所有人都事先对于数据的表示达成一致的选择。除了使用数据抽象屏障将表示与使用隔离外，我们还需要抽象屏障来隔离不同的设计选择并允许不同的选择在同一个程序中共存。

我们将会在最高等级的抽象开始我们的实现，并逐步向具体的表现形式发展。复数是一个数值型（Number），数值可以相加或相乘。数值如何相加和相乘是通过名为 `add` 和 `mul` 的方法抽象出来的。

```
>>> class Number:
        def __add__(self, other):
            return self.add(other)
        def __mul__(self, other):
            return self.mul(other)
```

这个类要求数值型对象拥有 `add` 和 `mul` 方法，但是没有定义他们。此外，它并没有 `__init__` 方法。`Number` 的目的不是直接被初始化，而是作为一个不同特殊数值类的超类（superclass）提供服务。我们的下一个任务就是为复数类型恰当地定义 `add` 和 `mul`。

一个复数可以被认为是一个在二维空间中的点，这个空间有两个相互垂直的轴，即实轴和虚轴。基于这种观点，复数 `c = real + imag * i ( where  i * i = -1 )` 可以被认为在一个平面上的点，它的水平坐标是 `real` 而垂直坐标是 `imag`。复数相加涉及到他们的 `real` 和 `imag` 各自相加。

当复数相乘时，把复数的表现形式认为是极坐标形式会更加自然，即表示为幅度（magnitude）和角度（angle）。两个复数相乘的结果是将一个复数按照另一个复数的长度拉伸，并将其旋转另一个复数的角度而得到的向量。

`Complex` 类继承自 `Number` 类，并且给出了对复数的数学运算。

```
>>> class Complex(Number):
        def add(self, other):
            return ComplexRI(self.real + other.real, self.imag + other.imag)
        def mul(self, other):
            magnitude = self.magnitude * other.magnitude
            return ComplexMA(magnitude, self.angle + other.angle)
```

这个实现假定存在两个表示复数的类，分别对应他们的两种自然表示形式。

- `ComplexRI` 使用实部和虚部构建一个复数。
- `ComplexMA` 使用幅度和角度构建一个复数。

接口。对象属性是一种消息传递的形式，它允许不同的数据类型以不同的方式响应相同的信息。从不同的类中引发类似的行为的一组共享信息是一种强大的抽象方法。接口是一组共享的属性名称，以及对它们的行为的规范。对于复数来说，实现算术运算所需要的接口包括四个属性：`real`、`imam`、`magnitude` 和 `angle`。

为了使复数的算数运算正确，这些属性必须保持一致。也就是说，直角坐标（`real`，`imag`）和极坐标（`magnitude`，`angle`）在复平面上必须表示同一个点。`Complex` 类确定了如何使用这些属性来对复数进行 `add` 和 `mul` 操作，从而以这种方式隐式定义了这样一个接口。

属性。两个或多个属性相互之间保持一个固定关系的要求是一个新的问题。一种解决方案是只使用一种表现方式来存储属性值，并在需要时计算另一种表现方式。

Python 有一种简单的计算属性的特性，可以通过零参数函数实时的计算属性。`@property` 修饰符允许函数在没有调用表达式语法（表达式后跟随圆括号）的情况下被调用。`Complex` 类存储了 `real` 和 `imag` 属性并在需要时计算 `magnitude` 和 `angle` 属性。

```
>>> from math import atan2
>>> class ComplexRI(Complex):
        def __init__(self, real, imag):
            self.real = real
            self.imag = imag
        @property
        def magnitude(self):
            return (self.real ** 2 + self.imag ** 2) ** 0.5
        @property
        def angle(self):
            return atan2(self.imag, self.real)
        def __repr__(self):
            return 'ComplexRI({0:g}, {1:g})'.format(self.real, self.imag)
```

在这个实现下，所有四个属性复数算术运算所需要的属性都可以在不需要任何调用表达式的情况下被访问，并且对于 `real` 和 `imag` 的修改会反映到 `magnitude` 和 `angle` 中。

```
>>> ri = ComplexRI(5, 12)
>>> ri.real
5
>>> ri.magnitude
13.0
>>> ri.real = 9
>>> ri.real
9
>>> ri.magnitude
15.0
```

类似的，`ComplexMA` 类存储了 `magnitude` 和 `angle` 属性，而在需要的时候计算 `real` 和 `imag`。

```
>>> from math import sin, cos, pi
>>> class ComplexMA(Complex):
        def __init__(self, magnitude, angle):
            self.magnitude = magnitude
            self.angle = angle
            @property
        def real(self):
            return self.magnitude * cos(self.angle)
        @property
        def imag(self):
            return self.magnitude * sin(self.angle)
        def __repr__(self):
            return 'ComplexMA({0:g}, {1:g} * pi)'.format(self.magnitude,
self.angle/pi)
```

对幅度和角度的改变同样也会立即反映到 `real` 和 `imag` 中。

```
>>> ma = ComplexMA(2, pi/2)
>>> ma.imag
2.0
>>> ma.angle = pi
>>> ma.real
-2.0
```

我们的复数实现现在已经完成了。在 `Complex` 类中，每一个实现复数的类都可以被用作任何一个算术运算函数的任何一个参数。

```
>>> from math import pi
>>> ComplexRI(1, 2) + ComplexMA(2, pi/2)
ComplexRI(1, 4)
>>> ComplexRI(0, 1) * ComplexRI(0, 1)
ComplexMA(1, 1 * pi)
```

使用接口来编码多重表示具有十分吸引人的特点。每一种表示形式的类都可以被单独开发；它们只需要就它们共享的属性名称和和对于这些属性的行为条件达成一致。接口还具有可添加性。如果程序员想要添加一个复数的第三方表现形式到同一个程序中，他们只需要创建一个拥有相同属性名称的类即可。

数据的多重表现形式和我们本章开始提到的数据抽象的概念联系密切。使用数据抽象，我们可以改变数据类型的实现而不需要改变程序的含义。通过接口和数据传递，我们可以在同一个程序中拥有多种不同的表现形式。在这两种情况下，一组名称和相应的行为条件定义的抽象使这种灵活性成为了可能。

2.7.4 泛型函数

泛型函数是适用于不同类型的参数的方法或函数。我们已经看过了许多例子。`Complex.add` 方法是泛型的，因为它可以接受 `ComplexRI` 或 `ComplexMA` 作为值。通过确保 `ComplexRI` 和 `ComplexMA` 共用同一个接口，我们已经获得了这种灵活性。使用接口和消息传递只是多种可以被用于实现泛型函数的方法中的一种。在本节我们将会考虑另外两个方法：类型派发和类型强制转换。

假设，除了我们的复数类，我们还实现了一个 `Rational` 类来精确地表示分数。`add` 和 `mul` 方法和之前的章节中出现的 `add_rational` 以及 `mul_rational` 都表示相同的运算。

```
>>> from fractions import gcd
>>> class Rational(Number):
        def __init__(self, numer, denom):
            g = gcd(numer, denom)
            self.numer = numer // g
            self.denom = denom // g
        def __repr__(self):
            return 'Rational({0}, {1})'.format(self.numer, self.denom)
        def add(self, other):
            nx, dx = self.numer, self.denom
            ny, dy = other.numer, other.denom
            return Rational(nx * dy + ny * dx, dx * dy)
        def mul(self, other):
            numer = self.numer * other.numer
            denom = self.denom * other.denom
            return Rational(numer, denom)
```

我们已经通过包含 `add` 和 `mul` 方法实现了 `Number` 超类的接口。因此，我们可以使用熟悉的运算符来对分数进行相加或相乘。

```
>>> Rational(2, 5) + Rational(1, 10)
Rational(1, 2)
>>> Rational(1, 4) * Rational(2, 3)
Rational(1, 6)
```

然而，我们还不能把一个分数加到复数上，即使这样的结合在数学上已经有了准确的定义。我们希望以某种精心受控制的方式引入这种跨类型的操作。以便在不严重违反我们的抽象屏障的情况下支持它。在我们期待的结果中存在着一种矛盾：我们希望能够将一个复数和一个分数相加，并且我们想要使用泛型的 `__add__` 方法来正确的处理所有的数值类型。同时我们希望尽可能地分离复数和分数的关注点，以维护一个模块化的程序。

类型派发。一种实现跨类型操作的方式是选择基于函数或方法的参数类型来选择相应的行为。类型派发的思想是写一个能够检查它所收到的参数的类型的函数，然后根据参数类型执行恰当的代码。

内置的函数 `isinstance` 接受一个对象或一个类。如果对象的类是所给的类或者继承自所给的类，它会返回一个真值。

```
>>> c = ComplexRI(1, 1)
>>> isinstance(c, ComplexRI)
True
>>> isinstance(c, Complex)
True
>>> isinstance(c, ComplexMA)
False
```

类型派发的一个简单例子是 `is_real` 函数，它对不同的复数类型使用不同的实现。

```
>>> def is_real(c):
        """Return whether c is a real number with no imaginary part."""
        if isinstance(c, ComplexRI):
            return c.imag == 0
        elif isinstance(c, ComplexMA):
            return c.angle % pi == 0

>>> is_real(ComplexRI(1, 1))
False
>>> is_real(ComplexMA(2, pi))
True
```

类型派发并不总是使用 `isinstance`，对于算术运算，我们会提供一个 `type_tag` 的属性给 `Rational` 和 `Complex` 实例，这个属性拥有一个字符串值。当两个值 `x` 和 `y` 有相同的 `type_tag` 时，我们可以直接使用 `x.add(y)` 来结合它们，否则我们需要跨类型操作。

```
>>> Rational.type_tag = 'rat'
>>> Complex.type_tag = 'com'
>>> Rational(2, 5).type_tag == Rational(1, 2).type_tag
True
>>> ComplexRI(1, 1).type_tag == ComplexMA(2, pi/2).type_tag
True
>>> Rational(2, 5).type_tag == ComplexRI(1, 1).type_tag
False
```

为了结合复数和分数，我们写一个同时依赖于它们两个的表现形式的函数。接下来，我们依靠这样一个事实，即 `Rational` 能够被精确转换为 `float` 值，而这个值是一个实数。转换的结果可以和一个复数值相结合。

```
>>> def add_complex_and_rational(c, r):
        return ComplexRI(c.real + r.numer/r.denom, c.imag)
```

乘法涉及到相似的转换。在极坐标系中，复平面中的实数总是有一个正的幅度值。角度 0 象征一个正数。角度 `pi` 象征一个负数。

```
>>> def mul_complex_and_rational(c, r):
        r_magnitude, r_angle = r.numer/r.denom, 0
        if r_magnitude < 0:
            r_magnitude, r_angle = -r_magnitude, pi
        return ComplexMA(c.magnitude * r_magnitude, c.angle + r_angle)
```

加法和乘法都是可交换的，因此交换参数顺序可以使用相同的跨类型操作实现。

```
>>> def add_rational_and_complex(r, c):
        return add_complex_and_rational(c, r)
>>> def mul_rational_and_complex(r, c):
        return mul_complex_and_rational(c, r)
```

类型派发的作用是保证这些跨类型的操作能在恰当的时候被使用。接下来，我们重写 `Number` 超类来为它的 `__add__` 和 `__mul__` 方法使用类型派发。

我们使用 `type_tag` 属性来区分参数的类型。也可以直接使用内置的 `isinstance` 方法，但是使用标签可以简化实现。使用类型标签也可以说明类型派发并不是必然和 Python 对象属性相联系的。而是一种在异构域上创建泛型函数的通用技术。

`__add__` 方法考虑了两种情况。首先，如果两个参数具有相同的类型标签，那么它会假定第一个参数的 `add` 方法可以接受第二个参数作为其参数。否则，他会检查一个叫做 `adders` 的包含跨类型实现的字典，看其是否包含了可以对这些参数类型进行相加的函数。如果有这样一个函数，`cross_apply` 方法会找到并应用它。`__mul__` 方法有着相似的结构。

```
>>> class Number:
        def __add__(self, other):
            if self.type_tag == other.type_tag:
                return self.add(other)
            elif (self.type_tag, other.type_tag) in self.adders:
                return self.cross_apply(other, self.adders)
        def __mul__(self, other):
            if self.type_tag == other.type_tag:
                return self.mul(other)
            elif (self.type_tag, other.type_tag) in self.multipliers:
                return self.cross_apply(other, self.multipliers)
        def cross_apply(self, other, cross_fns):
            cross_fn = cross_fns[(self.type_tag, other.type_tag)]
            return cross_fn(self, other)
        adders = {("com", "rat"): add_complex_and_rational,
                  ("rat", "com"): add_rational_and_complex}
        multipliers = {("com", "rat"): mul_complex_and_rational,
                       ("rat", "com"): mul_rational_and_complex}
```

在这个 `Number` 类的新定义中，所有的跨类型实现都在 `adders` 和 `multipliers` 字典中使用类型标签对进行索引。

基于字典的类型派发是可扩展的。`Number` 的新子类可以通过声明新的类型标签并添加跨类型操作到 `Number.adders` 和 `Number.multipliers` 中来将自己安装到系统中。它们还可以在子类中定义自己的 `adders` 和 `multipliers`。

尽管我们已经引入了一些复杂的东西到系统中，但现在我们可以在加法和乘法表达式中混合不同类型。

```
>>> ComplexRI(1.5, 0) + Rational(3, 2)
ComplexRI(3, 0)
>>> Rational(-1, 2) * ComplexMA(4, pi/2)
ComplexMA(2, 1.5 * pi)
```

强制转换。在完全不相关的操作和完全不相关的类型的一般情况下，实现显式的跨类型操作即是会有些繁琐，但仍然是我们所能期待的最好的结果。幸运的是，通过利用类型系统中可能潜在额外的结构，有时我们可以做得更好。通常不同的数据类型并不是完全不相关的，可能存在一些方法将一种类型视为另一种类型。这个过程被称为强制转换。例如，如果我们被要求使用算术方法结合一个分数和一个复数，则可以把分数看作一个虚部为零的复数。然后我们就可以使用 `Complex.add` 和 `Complex.mul` 来结合它们。

通常来说，我们可以通过设计一个强制转换函数实现这个想法，这个函数可以把一种类型的对象转换为另一种类型的等价对象。这里有一个典型的强制转换函数，它可以把一个分数转换为一个虚部为零的复数。

```
>>> def rational_to_complex(r):
        return ComplexRI(r.numer/r.denom, 0)
```

`Number` 类的另一种定义通过尝试强制两种参数转换为相同类型来执行跨类型操作。`coercions` 字典通过类型标签对索引了所有可能的强制转换，指示相关值将第一个类型的值转换为第二个类型。

通常情况下不可能随意地将任何一种类型的数据对象转换为其他所有类型。例如，不可能将任意的复数类型转换为分数，因此在 `coercions` 字典中不会有这样的转换实现。

`coerce` 方法返回两个具有相同类型标签的值。它会检查它的参数类型标签，将其和 `coercions` 字典中的条目进行比较，然后使用 `coerce_to` 将一个参数转换为另一个参数的类型。在 `coercions` 字典中只需要有一个条目就可以完成跨类型算术系统，从而取代了类型派发版本的 `Number` 中的四个跨类型函数。

```
>>> class Number:
        def __add__(self, other):
            x, y = self.coerce(other)
            return x.add(y)
        def __mul__(self, other):
            x, y = self.coerce(other)
            return x.mul(y)
        def coerce(self, other):
            if self.type_tag == other.type_tag:
                return self, other
            elif (self.type_tag, other.type_tag) in self.coercions:
                return (self.coerce_to(other.type_tag), other)
            elif (other.type_tag, self.type_tag) in self.coercions:
                return (self, other.coerce_to(self.type_tag))
        def coerce_to(self, other_tag):
            coercion_fn = self.coercions[(self.type_tag, other_tag)]
            return coercion_fn(self)
        coercions = {('rat', 'com'): rational_to_complex}
```

这个强制转换方案相比定义确定的跨类型操作具有一些优势。即使我们仍然需要写强制转换函数将各种类型联系起来，但对于每个类型对只需要一个函数而不是对每组类型和每个通用操作都编写不同函数。我们在这里依赖一个事实，即类型之间恰当地转换仅仅取决于类型自身，而与被应用到的特定操作无关。

扩展强制转换可以带来更多优势。一些更加精于设计的强制转换方案不仅仅只是尝试将一种类型转换到另一种，而是会尝试将两种不同的类型都转换为第三种通用类型。例如一个菱形和一个矩形：它们都不是对方的特殊情况，但是它们都可以被看作四边形。另一种强制转换的扩展是迭代强制转换，将一种数据类型通过中间类型转换到另一种类型。例如整数可以通过第一次转换成有理数，然后从有理数转换为

实数。链式强制转换可以减少程序中所要求的强制转换的函数总量。

即使有很多优势，但是强制转换也有潜在的缺点。例如，强制转换函数可能在应用时丢失信息。在我们的例子中，分数是准确的表示，但是当它被转换成复数时则变成了近似值。

有些编程语言自带了自动类型强制转换的功能。事实上，Python 在早期版本中，对象上就有一个叫 `__coerce__` 的特殊方法用于实现这一功能。但是，随着时间的推移，人们发现这种内置的强制转换系统的实用价值并不能抵消它的复杂性，因此这个特性最终被移除了。取而代之的是，特定的运算符会根据需要对其参数进行强制转换。