Generic Functions in Python

Our implementation of complex numbers has made two data types interchangeable as arguments to the `add_complex` and`mul_complex` functions. Now we will see how to use this same idea not only to define operations that are generic over different representations but also to define operations that are generic over different kinds of arguments that do not share a common interface.

The operations we have defined so far treat the different data types as being completely independent. Thus, there are separate packages for adding, say, two rational numbers, or two complex numbers. What we have not yet considered is the fact that it is meaningful to define operations that cross the type boundaries, such as the addition of a complex number to a rational number. We have gone to great pains to introduce barriers between parts of our programs so that they can be developed and understood separately.

We would like to introduce the cross-type operations in some carefully controlled way, so that we can support them without seriously violating our abstraction boundaries. There is a tension between the outcomes we desire: we would like to be able to add a complex number to a rational number, and we would like to do so using a generic `add` function that does the right thing with all numeric types. At the same time, we would like to separate the concerns of complex numbers and rational numbers whenever possible, in order to maintain a modular program.

Let us revise our implementation of rational numbers to use Python's built-in object system. As before, we will store a rational number as a numerator and denominator in lowest terms.

```
>>> from fractions import gcd
>>> class Rational(object):
        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)
```

Adding and multiplying rational numbers in this new implementation is similar to before.

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

Type dispatching. One way to handle cross-type operations is to design a different function for each possible combination of types for which the operation is valid. For example, we could extend our complex number implementation so that it provides a function for adding complex numbers to rational numbers. We can provide this functionality generically using a technique called*dispatching on type*.

The idea of type dispatching is to write functions that first inspect the type of argument they have received, and then execute code that is appropriate for the type. In Python, the type of an object can be inspected with the built-in `type` function.

```
>>> def iscomplex(z):
        return type(z) in (ComplexRI, ComplexMA)
>>> def isrational(z):
        return type(z) == Rational
```

In this case, we are relying on the fact that each object knows its type, and we can look up that type using the Python `type` function. Even if the `type` function were not available, we could imagine implementing `iscomplex` and `isrational` in terms of a shared class attribute for `Rational`, `ComplexRI`, and `ComplexMA`.

Now consider the following implementation of `add`, which explicitly checks the type of both arguments. We will not use Python's special methods (i.e., `__add__`) in this example.

```
>>> def add_complex_and_rational(z, r):
        return ComplexRI(z.real + r.numer/r.denom,
z.imag)
>>> def add(z1, z2):
    """Add z1 and z2, which may be complex or
rational."""
    if iscomplex(z1) and iscomplex(z2):
        return add_complex(z1, z2)
    elif iscomplex(z1) and isrational(z2):
        return add_complex_and_rational(z1, z2)
    elif isrational(z1) and iscomplex(z2):
        return add_complex_and_rational(z2, z1)
    else:
        return add_rationals(z1, z2)
```

This simplistic approach to type dispatching, which uses a large conditional statement, is not additive. If another numeric type were included in the program, we would have to re-implement `add` with new clauses.

We can create a more flexible implementation of `add` by implementing type dispatch through a dictionary. The first step in extending the flexibility of `add` will be to create a tag set for our classes that abstracts away from the two implementations of complex numbers.

```
>>> def type_tag(x):
    return type_tag.tags[type(x)]
>>> type_tag.tags = {ComplexRI: 'com', ComplexMA: 'com',
Rational: 'rat'}
```

Here, we store the tag set as an attribute of the `type_tag` function to avoid polluting the global namespace. (Recall that functions are objects and therefore may have attributes.)

Next, we use these type tags to index a dictionary that stores the different ways of adding numbers. The keys of the dictionary are tuples of type tags, and the values are type-specific addition functions.

```
>>> def add(z1, z2):
    types = (type_tag(z1), type_tag(z2))
```

```
    return add.implementations[types](z1, z2)
```

This definition of `add` does not have any functionality itself; it relies entirely on a dictionary called `add.implementations` to implement addition. We can populate that dictionary as follows.

```
>>> add.implementations = {}
>>> add.implementations[('com', 'com')] = add_complex
>>> add.implementations[('com', 'rat')] = add_complex_and_rational
>>> add.implementations[('rat', 'com')] = lambda x, y: add_complex_and_rational(y, x)
>>> add.implementations[('rat', 'rat')] = add_rationals
```

This dictionary-based approach to dispatching is additive, because `add.implementations` and `type_tag.tags` can always be extended. Any new numeric type can "install" itself into the existing system by adding new entries to these dictionaries.

While we have introduced some complexity to the system, we now have a generic, extensible `add` function that handles mixed types.

```
>>> add(ComplexRI(1.5, 0), Rational(3, 2))
ComplexRI(3.0, 0)
>>> add(Rational(5, 3), Rational(1, 2))
Rational(13, 6)
```

Data-directed programming. Our dictionary-based implementation of `add` is not addition-specific at all; it does not contain any direct addition logic. It only implements addition because we happen to have populated its `implementations` dictionary with functions that perform addition.

A more general version of generic arithmetic would apply arbitrary operators to arbitrary types and use a dictionary to store implementations of various combinations. This fully generic approach to implementing methods is called *data-directed programming*. In our case, we can implement both generic addition and multiplication without redundant logic.

```
>>> def apply(operator_name, x, y):
        tags = (type_tag(x), type_tag(y))
        key = (operator_name, tags)
        return apply.implementations[key](x, y)
```

In this generic `apply` function, a key is constructed from the operator name (e.g., `'add'`) and a tuple of type tags for the arguments. Implementations are also populated using these tags. We enable support for multiplication on complex and rational numbers below.

```
>>> def mul_complex_and_rational(z, r):
        return ComplexMA(z.magnitude * r.numer / r.denom,
z.angle)
>>>    mul_rationals_and_complex    =    lambda    r,    z:
mul_complex_and_rational(z, r)
>>>  apply.implementations  =  {('mul',  ('com',  'com')):
mul_complex,
                              ('mul',    ('com',    'rat')):
mul_complex_and_rational,
                              ('mul',    ('rat',    'com')):
mul_rationals_and_complex,
                              ('mul',    ('rat',    'rat')):
mul_rationals}
```

We can also include the addition implementations from `add` to `apply`, using the dictionary `update` method.

```
>>> adders = add.implementations.items()
>>>  apply.implementations.update({('add',  tags):fn  for
(tags, fn) in adders})
```

Now that apply supports 8 different implementations in a single table, we can use it to manipulate rational and complex numbers quite generically.

```
>>> apply('add', ComplexRI(1.5, 0), Rational(3, 2))
ComplexRI(3.0, 0)
>>> apply('mul', Rational(1, 2), ComplexMA(10, 1))
ComplexMA(5.0, 1)
```

This data-directed approach does manage the complexity of cross-type operators, but it is cumbersome. With such a system, the cost of introducing a new type is not just writing methods for that type, but also the construction and installation of the functions that implement the cross-type operations. This burden can easily require much more code than is needed to define the operations on the type itself.

While the techniques of dispatching on type and data-directed programming do create additive implementations of generic functions, they do not effectively separate implementation concerns; implementors of the individual numeric types need to take account of other types when writing cross-type operations. Combining rational numbers and complex numbers isn't strictly the domain of either type. Formulating coherent policies on the division of responsibility among types can be an overwhelming task in designing systems with many types and cross-type operations.

Coercion. In the general situation of completely unrelated operations acting on completely unrelated types, implementing explicit cross-type operations, cumbersome though it may be, is the best that one can hope for. Fortunately, we can sometimes do better by taking advantage of additional structure that may be latent in our type system. Often the different data types are not completely independent, and there may be ways by which objects of one type may be viewed as being of another type. This process is called*coercion*. For example, if we are asked to arithmetically combine a rational number with a complex number, we can view the rational number as a complex number whose imaginary part is zero. By doing so, we transform the problem to that of combining two complex numbers, which can be handled in the ordinary way by `add_complex` and `mul_complex`.

In general, we can implement this idea by designing coercion functions that transform an object of one type into an equivalent object of another type. Here is a typical coercion function, which transforms a rational number to a complex number with zero imaginary part:

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

Now, we can define a dictionary of coercion functions. This dictionary could be extended as more numeric types are introduced.

```
>>> coercions = {('rat', 'com'): rational_to_complex}
```

It is not generally possible to coerce an arbitrary data object of each type into all other types. For example, there is no way to coerce an arbitrary complex number to a rational number, so there will be no such conversion implementation in the `coercions`dictionary.

Using the `coercions` dictionary, we can write a function called `coerce_apply`, which attempts to coerce arguments into values of the same type, and only then applies an operator. The implementations dictionary of `coerce_apply` does not include any cross-type operator implementations.

```
>>> def coerce_apply(operator_name, x, y):
        tx, ty = type_tag(x), type_tag(y)
        if tx != ty:
            if (tx, ty) in coercions:
                tx, x = ty, coercions[(tx, ty)](x)
            elif (ty, tx) in coercions:
                ty, y = tx, coercions[(ty, tx)](y)
            else:
                return 'No coercion possible.'
        key = (operator_name, tx)
        return coerce_apply.implementations[key](x, y)
```

The `implementations` of `coerce_apply` require only one type tag, because they assume that both values share the same type tag. Hence, we require only four implementations to support generic arithmetic over complex and rational numbers.

```
>>> coerce_apply.implementations = {('mul', 'com'):
mul_complex,
                                    ('mul', 'rat'):
mul_rationals,
                                    ('add', 'com'):
add_complex,
                                    ('add', 'rat'):
add_rationals}
```

With these implementations in place, `coerce_apply` can replace `apply`.

```
>>> coerce_apply('add', ComplexRI(1.5, 0), Rational(3, 2))
ComplexRI(3.0, 0)
>>> coerce_apply('mul', Rational(1, 2), ComplexMA(10, 1))
ComplexMA(5.0, 1.0)
```

This coercion scheme has some advantages over the method of defining explicit cross-type operations. Although we still need to write coercion functions to relate the types, we need to write only one function for each pair of types rather than a different functions for each collection of types and each generic operation. What we are counting on here is the fact that the appropriate transformation between types depends only on the types themselves, not on the particular operation to be applied.

Further advantages come from extending coercion. Some more sophisticated coercion schemes do not just try to coerce one type into another, but instead may try to coerce two different types each into a third common type. Consider a rhombus and a rectangle: neither is a special case of the other, but both can be viewed as quadrilaterals. Another extension to coercion is iterative coercion, in which one data type is coerced into another via intermediate types. Consider that an integer can be converted into a real number by first converting it into a rational number, then converting that rational number into a real number. Chaining coercion in this way can reduce the total number of coercion functions that are required by a program.

Despite its advantages, coercion does have potential drawbacks. For one, coercion functions can lose information when they are applied. In our example, rational numbers are exact representations, but become approximations when they are converted to complex numbers.

Source : http://inst.eecs.berkeley.edu/~cs61A/book/chapters/objects.html#generic-functions