2.7 Object Abstraction

The object system allows programmers to build and use abstract data representations efficiently. It is also designed to allow multiple representations of abstract data to coexist in the same program.

A central concept in object abstraction is a *generic function*, which is a function that can accept values of multiple different types. We will consider three different techniques for implementing generic functions: shared interfaces, type dispatching, and type coercion. In the process of building up these concepts, we will also discover features of the Python object system that support the creation of generic functions.

2.7.1 String Conversion

To represent data effectively, an object value should behave like the kind of data it is meant to represent, including producing a string representation of itself. String representations of data values are especially important in an interactive language such as Python that automatically displays the string representation of the values of expressions in an interactive session.

String values provide a fundamental medium for communicating information among humans. Sequences of characters can be rendered on a screen, printed to paper, read aloud, converted to braille, or broadcast as Morse code. Strings are also fundamental to programming because they can represent Python expressions.

Python stipulates that all objects should produce two different string representations: one that is human-interpretable text and one that is a Python-interpretable expression. The constructor function for strings, `str`, returns a human-readable string. Where possible, the `repr` function returns a Python expression that evaluates to an equal object. The docstring for *repr* explains this property:

```
repr(object) -> string

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

The result of calling `repr` on the value of an expression is what Python prints in an interactive session.

```
>>> 12e12
12000000000000.0
>>> print(repr(12e12))
12000000000000.0
```

In cases where no representation exists that evaluates to the original value, Python typically produces a description surrounded by angled brackets.

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

The `str` constructor often coincides with `repr`, but provides a more interpretable text representation in some cases. For instance, we see a difference between `str` and `repr` with dates.

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

Defining the `repr` function presents a new challenge: we would like it to apply correctly to all data types, even those that did not exist when `repr` was implemented. We would like it to be a generic or *polymorphic function*, one that can be applied to many (*poly*) different forms (*morph*) of data.

The object system provides an elegant solution in this case: the `repr` function always invokes a method called `__repr__` on its argument.

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

By implementing this same method in user-defined classes, we can extend the applicability of `repr` to any class we create in the future. This example highlights another benefit of dot expressions in general, that they provide a mechanism for extending the domain of existing functions to new object types.

The `str` constructor is implemented in a similar manner: it invokes a method called `__str__` on its argument.

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

These polymorphic functions are examples of a more general principle: certain functions should apply to multiple data types. Moreover, one way to create such a function is to use a shared attribute name with a different definition in each class.

2.7.2 Special Methods

In Python, certain *special names* are invoked by the Python interpreter in special circumstances. For instance, the `__init__` method of a class is automatically invoked whenever an object is constructed. The `__str__` method is invoked automatically when printing, and `__repr__` is invoked in an interactive session to display values.

There are special names for many other behaviors in Python. Some of those used most commonly are described below.

True and false values. We saw previously that numbers in Python have a truth value; more specifically, 0 is a false value and all other numbers are true values. In fact, all objects in Python have a truth value. By default, objects of user-defined classes are considered to be true, but the special `__bool__` method can be used to override this behavior. If an object defines the `__bool__` method, then Python calls that method to determine its truth value.

As an example, suppose we want a bank account with 0 balance to be false. We can add a `__bool__` method to the `Account` class to create this behavior.

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

We can call the `bool` constructor to see the truth value of an object, and we can use any object in a boolean context.

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

Sequence operations. We have seen that we can call the `len` function to determine the length of a sequence.

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

The `len` function invokes the `__len__` method of its argument to determine its length. All built-in sequence types implement this method.

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

Python uses a sequence's length to determine its truth value, if it does not provide a `__bool__` method. Empty sequences are false, while non-empty sequences are true.

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

The `__getitem__` method is invoked by the element selection operator, but it can also be invoked directly.

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

Callable objects. In Python, functions are first-class objects, so they can be passed around as data and have attributes like any other object. Python also allows us to define objects that can be "called" like functions by including a `__call__` method. With this method, we can define a class that behaves like a higher-order function.

As an example, consider the following higher-order function, which returns a function that adds a constant value to its argument.

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

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

We can create an `Adder` class that defines a `__call__` method to provide the same functionality.

```
>>> 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
```

Here, the `Adder` class behaves like the `make_adder` higher-order function, and the `add_three_obj` object behaves like the `add_three` function. We have further blurred the line between data and functions.

Arithmetic. Special methods can also define the behavior of built-in operators applied to user-defined objects. In order to provide this generality, Python follows specific protocols to apply each operator. For example, to evaluate expressions that contain the `+` operator, Python checks for special methods on both the left and right operands of the expression. First, Python checks for an `__add__` method on the value of the left operand, then checks for an `__radd__` method on the value of the right operand. If either is found, that method is invoked with the value of the other operand as its argument. Some examples are given in the following sections. For readers interested in further details, the Python documentation describes the exhaustive set of method names for operators. Dive into Python 3 has a chapter on special method names that describes how many of these special method names are used.

2.7.3 Multiple Representations

Abstraction barriers allow us to separate the use and representation of data. However, in large programs, it may not always make sense to speak of "the underlying representation" for a data type in a program. For one thing, there might be more than one useful representation for a data object, and we might like to design systems that can deal with multiple representations.

To take a simple example, complex numbers may be represented in two almost equivalent ways: in rectangular form (real and imaginary parts) and in polar form (magnitude and angle). Sometimes the rectangular form is more appropriate and sometimes the polar form is more appropriate. Indeed, it is perfectly plausible to imagine a system in which complex numbers are represented in both ways, and in which the functions for manipulating complex numbers work with either representation. We implement such a system below. As a side note, we are developing a system that performs arithmetic operations on complex numbers as a simple but unrealistic example of a program that uses generic operations. A complex number type is actually built into Python, but for this example we will implement our own.

The idea of allowing for multiple representations of data arises regularly. Large software systems are often designed by many people working over extended periods of time, subject to requirements that change over time. In such an environment, it is simply not possible for everyone to agree in advance on choices of data representation. In addition to the data-abstraction barriers that isolate representation from use, we need abstraction barriers that isolate different design choices from each other and permit different choices to coexist in a single program.

We will begin our implementation at the highest level of abstraction and work towards concrete representations. A `Complex` number is a `Number`, and numbers can be added or multiplied together. How numbers can be added or multiplied is abstracted by the method names `add` and `mul`.

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

This class requires that Number objects have `add` and `mul` methods, but does not define them. Moreover, it does not have an `__init__` method. The purpose of `Number` is not to be instantiated directly, but instead to serve as a superclass of various specific number classes. Our next task is to define `add` and `mul` appropriately for complex numbers.

A complex number can be thought of as a point in two-dimensional space with two orthogonal axes, the real axis and the imaginary axis. From this perspective, the complex number `c = real + imag * i` (where `i * i = -1`) can be thought of as the point in the plane whose horizontal coordinate is `real` and whose vertical coordinate is `imag`. Adding complex numbers involves adding their respective `real` and `imag` coordinates.

When multiplying complex numbers, it is more natural to think in terms of representing a complex number in polar form, as a `magnitude` and an `angle`. The product of two complex numbers is the vector obtained by stretching one complex number by a factor of the length of the other, and then rotating it

through the angle of the other.

The `Complex` class inherits from `Number` and describes arithmetic for complex numbers.

```
>>> 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)
```

This implementation assumes that two classes exist for complex numbers, corresponding to their two natural representations:

- `ComplexRI` constructs a complex number from real and imaginary parts.
- `ComplexMA` constructs a complex number from a magnitude and angle.

Interfaces. Object attributes, which are a form of message passing, allows different data types to respond to the same message in different ways. A shared set of messages that elicit similar behavior from different classes is a powerful method of abstraction. An *interface* is a set of shared attribute names, along with a specification of their behavior. In the case of complex numbers, the interface needed to implement arithmetic consists of four attributes: `real`, `imag`, `magnitude`, and `angle`.

For complex arithmetic to be correct, these attributes must be consistent. That is, the rectangular coordinates `(real, imag)` and the polar coordinates `(magnitude, angle)` must describe the same point on the complex plane. The `Complex` class implicitly defines this interface by determining how these attributes are used to `add` and `mul` complex numbers.

Properties. The requirement that two or more attribute values maintain a fixed relationship with each other is a new problem. One solution is to store attribute values for only one representation and compute the other representation whenever it is needed.

Python has a simple feature for computing attributes on the fly from zero-argument functions. The `@property` decorator allows functions to be called without call expression syntax (parentheses following an expression). The `ComplexRI` class stores `real` and `imag` attributes and computes `magnitude` and `angle` on demand.

```
>>> 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)
```

As a result of this implementation, all four attributes needed for complex arithmetic can be accessed without any call expressions, and changes to `real` or `imag` are reflected in the `magnitude` and `angle`.

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

Similarly, the `ComplexMA` class stores `magnitude` and `angle`, but computes `real` and `imag` whenever those attributes are looked up.

```
>>> 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)
```

Changes to the magnitude or angle are reflected immediately in the `real` and `imag` attributes.

```
>>> ma = ComplexMA(2, pi/2)
>>> ma.imag
```

```
2.0
>>> ma.angle = pi
>>> ma.real
-2.0
```

Our implementation of complex numbers is now complete. Either class implementing complex numbers can be used for either argument in either arithmetic function in `Complex`.

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

The interface approach to encoding multiple representations has appealing properties. The class for each representation can be developed separately; they must only agree on the names of the attributes they share, as well as any behavior conditions for those attributes. The interface is also *additive*. If another programmer wanted to add a third representation of complex numbers to the same program, they would only have to create another class with the same attributes.

Multiple representations of data are closely related to the idea of data abstraction with which we began this chapter. Using data abstraction, we were able to change the implementation of a data type without changing the meaning of the program. With interfaces and message passing, we can have multiple different representations within the same program. In both cases, a set of names and corresponding behavior conditions define the abstraction that enables this flexibility.

2.7.4 Generic Functions

Generic functions are methods or functions that apply to arguments of different types. We have seen many examples already. The `Complex.add` method is generic, because it can take either a `ComplexRI` or `ComplexMA` as the value for `other`. This flexibility was gained by ensuring that both `ComplexRI` and `ComplexMA` share an interface. Using interfaces and message passing is only one of several methods used to implement generic functions. We will consider two others in this section: type dispatching and type coercion.

Suppose that, in addition to our complex number classes, we implement a `Rational` class to represent fractions exactly. The `add` and `mul` methods express the same computations as the `add_rational` and `mul_rational` functions from earlier in the chapter.

```
>>> 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)
```

We have implemented the interface of the `Number` superclass by including `add` and `mul` methods. As a result, we can add and multiply rational numbers using familiar operators.

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

However, we cannot yet add a rational number to a complex number, although in mathematics such a combination is well-defined. We would like to introduce this cross-type operation in some carefully controlled way, so that we can support it without seriously violating our abstraction barriers. 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__` method 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.

Type dispatching. One way to implement cross-type operations is to select behavior based on the types of the arguments to a function or method. The idea of type dispatching is to write functions that inspect the type of arguments they receive, then execute code that is appropriate for those types.

The built-in function `isinstance` takes an object and a class. It returns true if the object has a class that either is or inherits from the given class.

```
>>> c = ComplexRI(1, 1)
>>> isinstance(c, ComplexRI)
```

```
True
>>> isinstance(c, Complex)
True
>>> isinstance(c, ComplexMA)
False
```

A simple example of type dispatching is an `is_real` function that uses a different implementation for each type of complex number.

```
>>> 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
```

Type dispatching is not always performed using `isinstance`. For arithmetic, we will give a `type_tag` attribute to `Rational` and `Complex` instances that has a string value. When two values `x` and `y` have the same `type_tag`, then we can combine them directly with `x.add(y)`. If not, we need a cross-type operation.

```
>>> 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
```

To combine complex and rational numbers, we write functions that rely on both of their representations simultaneously. Below, we rely on the fact that a `Rational` can be converted approximately to a `float` value that is a real number. The result can be combined with a complex number.

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

Multiplication involves a similar conversion. In polar form, a real number in the complex plane always has a positive magnitude. The angle 0 indicates a positive number. The angle `pi` indicates a negative number.

```
>>> 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)
```

Both addition and multiplication are commutative, so swapping the argument order can use the same implementations of these cross-type operations.

```
>>> 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)
```

The role of type dispatching is to ensure that these cross-type operations are used at appropriate times. Below, we rewrite the `Number` superclass to use type dispatching for its `__add__` and `__mul__` methods.

We use the `type_tag` attribute to distinguish types of arguments. One could directly use the built-in `isinstance` method as well, but tags simplify the implementation. Using type tags also illustrates that type dispatching is not necessarily linked to the Python object system, but instead a general technique for creating generic functions over heterogeneous domains.

The `__add__` method considers two cases. First, if two arguments have the same type tag, then it assumes that `add` method of the first can take the second as an argument. Otherwise, it checks whether a dictionary of cross-type implementations, called `adders`, contains a function that can add arguments of those type tags. If there is such a function, the `cross_apply` method finds and applies it. The `__mul__` method has a similar structure.

```
>>> 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}
```

In this new definition of the `Number` class, all cross-type implementations are indexed by pairs of type tags in the `adders` and `multipliers` dictionaries.

This dictionary-based approach to type dispatching is extensible. New subclasses of `Number` could install themselves into the system by declaring a type tag and adding cross-type operations to `Number.adders` and `Number.multipliers`. They could also define their own `adders` and `multipliers` in a subclass.

While we have introduced some complexity to the system, we can now mix types in addition and multiplication expressions.

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

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. After doing so, we can use `Complex.add` and `Complex.mul` to combine them.

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

The alternative definition of the `Number` class performs cross-type operations by attempting to coerce both arguments to the same type. The `coercions` dictionary indexes all possible coercions by a pair of type tags, indicating that the corresponding value coerces a value of the first type to a value of the second type.

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.

The `coerce` method returns two values with the same type tag. It inspects the type tags of its arguments, compares them to entries in the `coercions` dictionary, and converts one argument to the type of the other using `coerce_to`. Only one entry in `coercions` is necessary to complete our cross-type arithmetic system, replacing the four cross-type functions in the type-dispatching version of `Number`.

```
>>> 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}
```

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 function for each set 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.

Some programming languages have automatic coercion systems built in. In fact, early versions of Python had a `__coerce__` special method on objects. In the end, the complexity of the built-in coercion system did not justify its use, and so it was removed. Instead, particular operators apply coercion to their arguments as needed.

Continue: 2.8 Efficiency