Data Abstraction and Example: Arithmetic on Rational Numbers in Python

Data Abstraction :

As we consider the wide set of things in the world that we would like to represent in our programs, we find that most of them have compound structure. A date has a year, a month, and a day; a geographic position has latitude and longitude coordinates. To represent positions, we would like our programming language to have the capacity to couple together a latitude and longitude to form a pair, a *compound data* value that our programs can manipulate as a single conceptual unit, but which also has two parts that can be considered individually.

The use of compound data enables us to increase the modularity of our programs. If we can manipulate geographic positions directly as objects in their own right, then we can separate the part of our program that deals with values per se from the details of how those values may be represented. The general technique of isolating the parts of a program that deal with how data are represented from the parts of a program that deal with how those data are manipulated is a powerful design methodology called*data abstraction*. Data abstraction makes programs much easier to design, maintain, and modify.

Data abstraction is similar in character to functional abstraction. When we create a functional abstraction, the details of how a function is implemented can be suppressed, and the particular function itself can be replaced by any other function with the same overall behavior. In other words, we can make an abstraction that separates the way the function is used from the details of how the function is implemented. Analogously, data abstraction is a methodology that enables us to isolate how a compound data object is used from the details of how it is constructed.

The basic idea of data abstraction is to structure programs so that they operate on abstract data. That is, our programs should use data in such a way as to make as few

assumptions about the data as possible. At the same time, a concrete data representation is defined, independently of the programs that use the data. The interface between these two parts of our system will be a set of functions, called selectors and constructors, that implement the abstract data in terms of the concrete representation. To illustrate this technique, we will consider how to design a set of functions for manipulating rational numbers.

As you read the next few sections, keep in mind that most Python code written today uses very high-level abstract data types that are built into the language, like classes, dictionaries, and lists. Since we're building up an understanding of how these abstractions work, we can't use them yet ourselves. As a consequence, we will write some code that isn't typical of the typical way most Python programmers would implement these ideas in the language. What we write is instructive, however, because it demonstrates how these abstractions can be constructed! Remember that computer science isn't just about learning to use programming languages, but also understanding how they work.

Example: Arithmetic on Rational Numbers

Recall that a rational number is a ratio of integers, and rational numbers constitute an important sub-class of real numbers. A rational number like `1/3` or `17/29` is typically written as:

```
<numerator>/<denominator>
```

where both the `<numerator>` and `<denominator>` are placeholders for integer values. Both parts are needed to exactly characterize the value of the rational number.

Rational numbers are important in computer science because they, like integers, can be represented exactly. Irrational numbers (like `pi` or `e` or `sqrt(2)`) are instead approximated using a finite binary expansion. Thus, working with rational numbers should, in principle, allow us to avoid approximation errors in our arithmetic.

However, as soon as we actually divide the numerator by the denominator, we can be left with a truncated decimal approximation (a `float`).

```
>>> 1/3
0.3333333333333333
```

and the problems with this approximation appear when we start to conduct tests:

```
>>> 1/3 == 0.33333333333333300000 # Beware of approximations
True
```

How computers approximate real numbers with finite-length decimal expansions is a topic for another class. The important idea here is that by representing rational numbers as ratios of integers, we avoid the approximation problem entirely. Hence, we would like to keep the numerator and denominator separate for the sake of precision, but treat them as a single unit.

We know from using functional abstractions that we can start programming productively before we have an implementation of some parts of our program. Let us begin by assuming that we already have a way of constructing a rational number from a numerator and a denominator. We also assume that, given a rational number, we have a way of extracting (or selecting) its numerator and its denominator. Let us further assume that the constructor and selectors are available as the following three functions:

- `rational(n, d)` returns the rational number with numerator `n` and denominator `d`.
- `numer(x)` returns the numerator of the rational number `x`.
- `denom(x)` returns the denominator of the rational number `x`.

We are using here a powerful strategy of synthesis: *wishful thinking*. We haven't yet said how a rational number is represented, or how the functions `numer`, `denom`, and `rational` should be implemented. Even so, if we did have these three functions, we could then add, multiply, and test equality of rational numbers by calling them:

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

Now we have the operations on rational numbers defined in terms of the selector functions `numer` and `denom`, and the constructor function `rational`, but we haven't yet defined these functions. What we need is some way to glue together a numerator and a denominator into a unit.

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