## 2.2 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. For example, 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 as whole values, then we can shield parts of our program that compute using positions from the details of how those positions are represented. The general technique of isolating the parts of a program that deal with how data are represented from the parts that deal with how 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 isolates how a compound data value 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 as an independent part of the program.

These two parts of a program, the part that operates on abstract data and the part that defines a concrete representation, are connected by a small set of functions that implement 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.

### 2.2.1 Example: Rational Numbers

A rational number is a ratio of integers, and rational numbers constitute an important sub-class of real numbers. A rational number such as `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. Actually dividing integers produces a `float` approximation, losing the exact precision of integers.

```
>>> 1/3
0.3333333333333333
>>> 1/3 == 0.33333333333333300000  # Dividing integers yields an approximation
True
```

However, we can create an exact representation for rational numbers by combining together the numerator and denominator.

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 selecting its numerator and its denominator component. 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 for designing programs: *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 define these three functions, we could then add, multiply, print, and test equality of rational numbers:

```
>>> 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)
```

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 compound value.

### 2.2.2 Pairs

To enable us to implement the concrete level of our data abstraction, Python provides a compound structure called a `list`, which can be constructed by placing expressions within square brackets separated by commas. Such an expression is called a list literal.

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

The elements of a list can be accessed in two ways. The first way is via our familiar method of multiple assignment, which unpacks a list into its elements and binds each element to a different name.

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

A second method for accessing the elements in a list is by the element selection operator, also expressed using square brackets. Unlike a list literal, a square-brackets expression directly following another expression does not evaluate to a `list` value, but instead selects an element from the value of the preceding expression.

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

Lists in Python (and sequences in most other programming languages) are 0-indexed, meaning that the index 0 selects the first element, index 1 selects the second, and so on. One intuition that supports this indexing convention is that the index represents how far an element is offset from the beginning of the list.

The equivalent function for the element selection operator is called `getitem`, and it also uses 0-indexed positions to select elements from a list.

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

Two-element lists are not the only method of representing pairs in Python. Any way of bundling two values together into one can be considered a pair. Lists are a common method to do so. Lists can also contain more than two elements, as we will explore later in the chapter.

**Representing Rational Numbers.** We can now represent a rational number as a pair of two integers: a numerator and a denominator.

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

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

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

Together with the arithmetic operations we defined earlier, we can manipulate rational numbers with the functions we have defined.

```
>>> 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
```

As the example above shows, our rational number implementation does not reduce rational numbers to lowest terms. We can remedy this flaw by changing the implementation of `rational`. If we have a function for computing the greatest common denominator of two integers, we can use it to reduce the numerator and the denominator to lowest terms before constructing the pair. As with many useful tools, such a function already exists in the Python Library.

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

The floor division operator, `//`, expresses integer division, which rounds down the fractional part of the result of division. Since we know that `g` divides both `n` and `d` evenly, integer division is exact in this case. This revised `rational` implementation ensures that rationals are expressed in lowest terms.

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

This improvement was accomplished by changing the constructor without changing any of the functions that implement the actual arithmetic operations.

### 2.2.3 Abstraction Barriers

Before continuing with more examples of compound data and data abstraction, let us consider some of the issues raised by the rational number example. We defined operations in terms of a constructor `rational` and selectors `numer` and `denom`. In general, the underlying idea of data abstraction is to identify a basic set of operations in terms of which all manipulations of values of some kind will be expressed, and then to use only those operations in manipulating the data. By restricting the use of operations in this way, it is much easier to change the representation of abstract data without changing the behavior of a program.

For rational numbers, different parts of the program manipulate rational numbers using different operations, as described in this table.

| Parts of the program that... | Treat rationals as... | Using only... |
|---|---|---|
| Use rational numbers to perform computation | whole data values | `add_rational, mul_rational, rationals_are_equal, print_rational` |
| Create rationals or implement rational operations | numerators and denominators | `rational, numer, denom` |
| Implement selectors and constructor for rationals | two-element lists | list literals and element selection |

In each layer above, the functions in the final column enforce an abstraction barrier. These functions are called by a higher level and implemented using a lower level of abstraction.

An abstraction barrier violation occurs whenever a part of the program that can use a higher level function instead uses a function in a lower level. For example, a function that computes the square of a rational number is best implemented in terms of `mul_rational`, which does not assume anything about the implementation of a rational number.

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

Referring directly to numerators and denominators would violate one abstraction barrier.

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

Assuming that rationals are represented as two-element lists would violate two abstraction barriers.

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

Abstraction barriers make programs easier to maintain and to modify. The fewer functions that depend on a particular representation, the fewer changes are required when one wants to change that representation. All of these implementations of `square_rational` have the correct behavior, but only the first is robust to future changes. The `square_rational` function would not require updating even if we altered the representation of rational numbers. By contrast, `square_rational_violating_once` would need to be changed whenever the selector or constructor signatures changed, and `square_rational_violating_twice` would require updating whenever the implementation of rational numbers changed.

### 2.2.4 The Properties of Data

Abstraction barriers shape the way in which we think about data. A valid representation of a rational number is not restricted to any particular implementation (such as a two-element list); it is a value returned by `rational` that can be passed to `numer`, and `denom`. In addition, the appropriate relationship must hold among the constructor and selectors. That is, if we construct a rational number `x` from integers `n` and `d`, then it should be the case that `numer(x)/denom(x)` is equal to `n/d`.

In general, we can express abstract data using a collection of selectors and constructors, together with some behavior conditions. As long as the behavior conditions are met (such as the division property above), the selectors and constructors constitute a valid representation of a kind of data. The implementation details below an abstraction barrier may change, but if the behavior does not, then the data abstraction remains valid, and any program written using this data abstraction will remain correct.

This point of view can be applied broadly, including to the pair values that we used to implement rational numbers. We never actually said much about what a pair was, only that the language supplied the means to create and manipulate lists with two elements. The behavior we require to implement a pair is that it glues two values together. Stated as a behavior condition,

- If a pair `p` was constructed from values `x` and `y`, then `select(p, 0)` returns `x`, and `select(p, 1)` returns `y`.

We don't actually need the `list` type to create pairs. Instead, we can implement two functions `pair` and `select` that fulfill this description just as well as a two-element list.

```
>>> 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)
```

With this implementation, we can create and manipulate pairs.

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

This use of higher-order functions corresponds to nothing like our intuitive notion of what data should be. Nevertheless, these functions suffice to represent pairs in our programs. Functions are sufficient to represent compound data.

The point of exhibiting the functional representation of a pair is not that Python actually works this way (lists are implemented more directly, for efficiency reasons) but that it could work this way. The functional representation, although obscure, is a perfectly adequate way to represent pairs, since it fulfills the only conditions that pairs need to fulfill. The practice of data abstraction allows us to switch among representations easily.

*Continue*: 2.3 Sequences