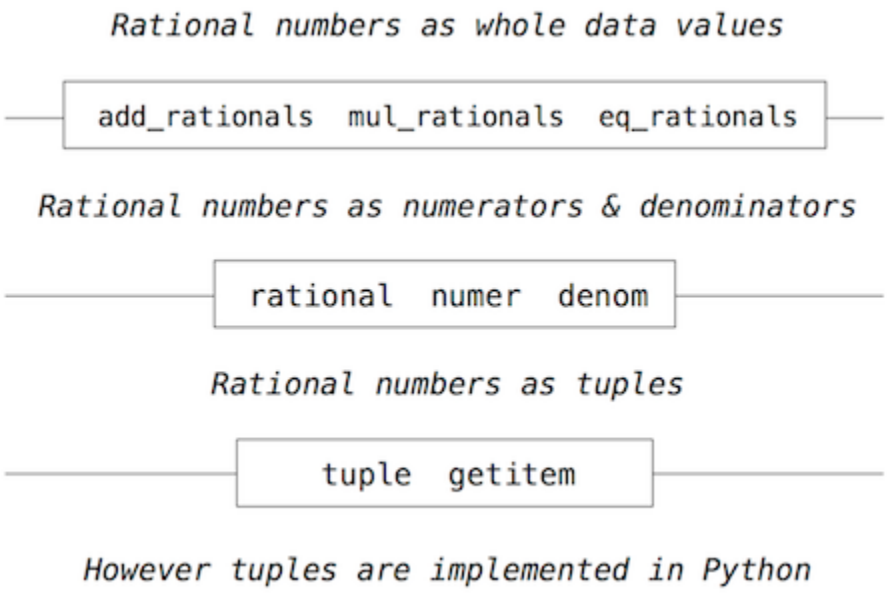
Abstraction Barriers and Properties of Data in Python

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 for each type of value a basic set of operations in terms of which all manipulations of values of that type will be expressed, and then to use only those operations in manipulating the data.

We can envision the structure of the rational number system as a series of layers.



The horizontal lines represent abstraction barriers that isolate different levels of the system. At each level, the barrier separates the functions (above) that use the data abstraction from the functions (below) that implement the data abstraction. Programs that use rational numbers manipulate them solely in terms of the their arithmetic functions: `add_rationals`, `mul_rationals`, and`eq_rationals`. These, in turn, are

implemented solely in terms of the constructor and selectors `rational`, `numer`, and `denom`, which themselves are implemented in terms of tuples. The details of how tuples are implemented are irrelevant to the rest of the layers as long as tuples enable the implementation of the selectors and constructor.

At each layer, the functions within the box enforce the abstraction boundary because they are the only functions that depend upon both the representation above them (by their use) and the implementation below them (by their definitions). In this way, abstraction barriers are expressed as sets of functions.

Abstraction barriers provide many advantages. One advantage is that they makes programs much 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.

The Properties of Data

We began the rational-number implementation by implementing arithmetic operations in terms of three unspecified functions:`rational`, `numer`, and `denom`. At that point, we could think of the operations as being defined in terms of data objects --- numerators, denominators, and rational numbers --- whose behavior was specified by the latter three functions.

But what exactly is meant by data? It is not enough to say "whatever is implemented by the given selectors and constructors." We need to guarantee that these functions together specify the right behavior. 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 think of an abstract data type as defined by some collection of selectors and constructors, together with some behavior conditions. As long as the behavior conditions are met (such as the division property above), these functions constitute a valid representation of the data type.

This point of view can be applied to other data types as well, such as the two-element tuple that we used in order to implement rational numbers. We never actually said much about what a tuple was, only that the language supplied operators to create and manipulate tuples. We can now describe the behavior conditions of two-element tuples, also called pairs, that are relevant to the problem of representing rational numbers.

In order to implement rational numbers, we needed a form of glue for two integers, which had the following behavior:

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

We can implement functions `pair` and `getitem_pair` that fulfill this description just as well as a tuple.

```
>>> def pair(x, y):
        """Return a function that behaves like a two-
element tuple."""
        def dispatch(m):
            if m == 0:
                return x
            elif m == 1:
                return y
        return dispatch
>>> def getitem_pair(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, 12)
>>> getitem_pair(p, 0)
20
>>> getitem_pair(p, 1)
12
```

This use of functions corresponds to nothing like our intuitive notion of what data should be. Nevertheless, these functions suffice to represent compound data in our programs.

The subtle point to notice is that the value returned by `pair` is a function called `dispatch`, which takes an argument `m` and returns either `x` or `y`. Then, `getitem_pair` calls this function to retrieve the appropriate value. We will return to the topic of dispatch functions several times throughout this chapter.

The point of exhibiting the functional representation of a pair is not that Python actually works this way (tuples 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. This example also demonstrates that the ability to manipulate functions as values automatically provides us the ability to represent compound data.

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