Pairs in Python

To enable us to implement the concrete level of our data abstraction, Python provides a compound structure called a `tuple`, which can be constructed by separating values by commas. Although not strictly required, parentheses almost always surround tuples.

```
>>> (1, 2)
(1, 2)
```

The elements of a tuple can be unpacked in two ways. The first way is via our familiar method of multiple assignment.

```
>>> pair = (1, 2)
>>> pair
(1, 2)
>>> x, y = pair
>>> x
1
>>> y
2
```

In fact, multiple assignment has been creating and unpacking tuples all along.

A second method for accessing the elements in a tuple is by the indexing operator, expressed using square brackets.

```
>>> pair[0]
1
>>> pair[1]
2
```

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

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

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

Tuples are native types, which means that there are built-in Python operators to manipulate them. We'll return to the full properties of tuples shortly. At present, we are only interested in how tuples can serve as the glue that implements abstract data types.

Representing Rational Numbers. Tuples offer a natural way to implement rational numbers as a pair of two integers: a numerator and a denominator. We can implement our constructor and selector functions for rational numbers by manipulating 2-element tuples.

```
>>> def rational(n, d):
        return (n, d)
>>> def numer(x):
        return getitem(x, 0)
>>> def denom(x):
        return getitem(x, 1)
```

A function for printing rational numbers completes our implementation of this abstract data type.

```
>>> def rational_to_string(x):
        """Return a string 'n/d' for numerator n and
denominator d."""
        return '{0}/{1}'.format(numer(x), denom(x))
```

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

```
>>> half = rational(1, 2)
>>> rational_to_string(half)
'1/2'
>>> third = rational(1, 3)
>>> rational_to_string(mul_rationals(half, third))
```

```
'1/6'
>>> rational_to_string(add_rationals(third, third))
'6/9'
```

As the final example shows, our rational number implementation does not reduce rational numbers to lowest terms. We can remedy this by changing `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 double slash 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. Now we have

```
>>> rational_to_string(add_rationals(third, third))
'2/3'
```

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

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