Rational Numbers

This is just something I have been playing with. It's a way of doing rational number arithmetic in Python.

Let's start with the easy bit:

```
# fractions.py
# An implementation of rational number arithmetic

def main():
    print("*** Rational Numbers ***")
    print("Enter 'q' to quit.")
    while 1:
        try:
            temp = input(">> ")
            if temp == 'q':
                return
            print(rat_to_string(evaluate(parse(temp.split()))))
        except ValueError:
            print("Huh?")
```

Here's my main() function. It just waits for the user to enter something. If you enter a 'q', the program will exit. If you type something it doesn't understand it will print: "Huh?". But if you type in some fractions, the string gets chopped up into a list, parsed, evaluated and then turned into another string and printed out. Simples!

Right. So the first thing is that we have a list of expressions to parse. So, for instance, if we typed in:

```
2/3 * 2 - 1/4
```

We would have a list with five elements in it: two fractions, the number '2' and two operators. Here's "parse()":

```
def parse(some_list):
    """Takes a list and returns a list of fractions and operators."""
    ops = ['*', '/', '+', '-']
    new = []
    for item in some_list:
        if item in ops:
            new.append(item)
        else:
            new.append(make_rat(item.split('/')))
    return new

def make_rat(k):
    """Returns a list representing k as a (simplified) fraction."""
    if len(k) == 2:
        a, b = numer(k), denom(k)
        d = gcd(a, b)
        return [a//d, b//d]
    else:
        return [int(k[0]), 1]
```

If parse() finds an operator, it leaves it as it is. Otherwise it tries to make a "rational number" - which us basically just a 2-item list. Whole numbers get turned into themselves over one. Other fractions are checked to see if they are simplified by the gcd() function.

```
def numer(x):
    """Returns the numerator of x."""
    return int(x[0])

def denom(x):
    """Returns the denominator of x"""
    return int(x[1])

# we use gcd to simplify fractions
def gcd(a,b):
    """Takes two integers and returns gcd."""
    if b == 0:
        return a
    else:
        return gcd(b, a % b)
```

I've used numer() and denom() mainly to make the code easier to read compared with using a lot of list indices.

The real work goes on in the evaluate() function:

```
def evaluate(my_list):
    """Takes a list and performs arithmetic until list is length 1."""
    ops = [('*', lambda x,y: times(x,y)),
           ('/', lambda x,y: times(x,reciprocal(y))),
           ('+', lambda x,y: add(x,y)),
           ('-', lambda x,y: add(x,minus(y)))]
    while len(my_list) > 1:
        for key, f in ops:
            if key in my_list:
                i = my_list.index(key)
                my_list[i-1: i+2] = [f(my_list[i - 1], my_list[i +1])]
                break
    return my_list[0]
```

So, first I set up a dictionary with the operators and the functions that I want to be associated with them. Then there's a loop which keeps going until the list just has only one item in it. Each time through, the function tries to replace three list-items of the form "fraction - operator - fraction" with one fraction, based on the arithmetic rules. The "break" makes sure that we only do one sum at a time, which means that the operations are done in the correct order (multiplication, division, addition, subtraction).

```
def add(a, b):
    """Takes 2 fractions and returns the sum as a fraction."""
    return make_rat([numer(a)*denom(b) + numer(b)*denom(a), denom(a)*denom(b)])

def times(a, b):
    """Takes 2 fractions and returns the procuct as a fraction."""
    return make_rat([numer(a) * numer(b) , denom(a) * denom(b)])

def reciprocal(y):
    """Returns reciprocal of y; used in division."""
    return [denom(y), numer(y)]

def minus(y):
    """Returns 0 - y; used in subtraction."""
    return [-numer(y), denom(y)]
```

Rather than write rules for minus and divide, I just make the numerator negative and swap the numerator and denominator, respectively.

The final function just makes the fraction look nicer when we print it out. Whole numbers are printed without the denominator. Top heavy fractions are expressed as mixed numbers.

```
def rat_to_string(rat):
    """Turns a rational number into a string."""
    n, d = numer(rat), denom(rat)
    if d == 1:
        return str(n)
    string = ""
    if n > d:
        w, n = divmod(n, d)
        string = str(w) + " "
    return string + str(n) + "/" + str(d)
```

One thing I want to do with this program is to enable it to cope with brackets. A nice **task** for you might be to see if you can implement this.

Don't forget that the user might type something with more than one set of brackets!

I am planning to use the handy "rindex()" method:

```
>>> a= "2 * (1/2 + (1/3 / 1/2))"
>>> a.index('(')
4
>>> a.index(')')
21
>>> a.rindex(')')
22
```

rindex() tells us the position of the *last* occurrence of a character, which should enable us to find the end of the brackets.

Happy hacking!