Rational class example

There are many considerations and options in developing classes. Let us look at the earlier Fraction class. Certainly fractions are thing you are used to working with since grade school.

But thinking deeper, fraction notation is used as a way of expressing an underlying rational number. The Fraction class stored and remembered a particular representation, with a specified numerator and denominator, so 3/6 was not the same as 1/2. Maybe in some places that makes sense, but they are equal mathematically. I presume that as primary student you were told in general to reduce to lowest terms, allowing an easy check of equality.

I propose that a Rational only stores a fraction in lowest terms: it does the simplification immediately inside the constructor. That will make equality testing more sane.

Also there was a messiness in the Fraction class with negative signs as the example was written:

```
f = Fraction(2, -5)
print(f) # prints 2/-5
```

This is certainly not the usual place for a minus sign in a fraction: You would expect -2/5. So besides reducing to lowest terms, our Rational constructor will make sure the stored denominator is positive.

Other numeric types in Python are all immutable. A Fraction was defined to be mutable. I want a Rational to be immutable: have no methods to mutate an existing Rational; use the Python convention for data attributes intended as private, starting with an underscore: `_num` and `_den`. (Again, in Python, this is not a guarantee of privacy, but a clear indication of intention.)

As we discussed earlier, an `int` is mathematically a rational number, and it makes sense to be able to add an `int` and a `Rational`.

If we are extending the use of `+`, note that it makes sense to add a rational number and a real number (or its approximation as a float). Since floats are approximate real numbers it make sense that a sum with a Rational is also a float.

We will also allow all the other basic arithmetic and comparison operations, with the same extensions, as appropriate.

In Python we can provide a string as parameter to a numeric constructor, like float('23.456'). We will allow strings like that for a `Rational` also. A decimal literal is mathematically a rational number, for example 23.456 is another notation for 23456/1000. Another string form with an obvious rational interpretation is something like `'-15/20'`. We will allow string conversions to Rational in forms like `Rational('23.456') and Rational('-15/20')`.

You are always going to want to test your code. *Test-driven design* has you write the tests first, forcing you to think of the practical outcomes you want, before you write the implementation code. This also can be very useful if you are writing for a client. You can say, “Is this exactly how you intended the code to behave?”.

There are many approaches to managing tests. One in Python is to use the module `unittest`. Though we have generally avoided boilerplate in our Python examples, it is premature to talk a lot about the important advanced topic of *inheritance*. Our specialized testing class inherits methods from the class `unittest.TestCase`. You will see how this is expressed in the class heading. The tests asserting various things are well named, so they should make sense, checking on the relationship between two parameters, checking if an expression evaluates to `True` or `False`, or whether code causes a particular kind of error to be raised.

Here is my full testing class, in a file testrational.py (http://anh.cs.luc.edu/170/testrational.py), assuming the class `Rational` is in the file rational.py (http://anh.cs.luc.edu/170/rational.py):

```python
''' unit test for Rational '''

import unittest

from rational import Rational

class RationalTestCase(unittest.TestCase):  # must have superclass
    # runs methods with name starting with test_
    def test_constructor(self):
        self.assertEqual(str(Rational(10, -15)), '-2/3')
        self.assertEqual(str(Rational(-10, -15)), '2/3')
        self.assertEqual(str(Rational(-5)), '-5')
        self.assertEqual(str(Rational("-1.98")), '-99/50')
        self.assertEqual(str(Rational(".9")), '9/10')
        self.assertEqual(str(Rational("-125.")), '-125')

    def test_eq(self):
        w = Rational(3)
        x = Rational(6, 8)
        y = Rational("1.2")
        z = Rational(6, 5)
        self.assertEqual(y, z)
        self.assertEqual(y, Rational('6/5'))
        self.assertEqual(Rational(4, 2), 2)
        self.assertEqual(Rational(0), 0)
        self.assertEqual(x, 0.75)
        self.assertEqual(w, 3)

        self.assertNotEqual(x, w)
        self.assertNotEqual(y, 1)
        self.assertNotEqual(Rational(4, 2), 2.1)
        self.assertNotEqual(Rational(0), "garb")

    def test_typeConvert(self):
        w = Rational(3)
        x = Rational(6, 8)
        y = Rational("1.2")
        z = Rational(6, 5)
        self.assertEqual(float(x), 0.75)
        self.assertEqual(int(w), 3)
        self.assertEqual(int(x), 0)
        self.assertEqual(int(y), 1)
        self.assertEqual(int(-y), -1)

        self.assertFalse(bool(Rational(0)))
        self.assertTrue(bool(x))

    def test_arith(self):
        x = Rational(6, 8)
        y = Rational("1.2")
        self.assertEqual(x+y, Rational(39, 20))
        self.assertEqual(x-y, Rational(-9, 20))
        self.assertEqual(x*y, Rational(9, 10))
        self.assertEqual(x/y, Rational(5, 8))
        self.assertEqual(x+2, Rational(11, 4))
        self.assertEqual(x-2, Rational(-5, 4))
        self.assertEqual(x*2, Rational(3, 2))
        self.assertEqual(x/2, Rational(3, 8))
        self.assertEqual(2+x, Rational(11, 4))
        self.assertEqual(2-x, Rational(5, 4))
        self.assertEqual(2*x, Rational(3, 2))
        self.assertEqual(2/x, Rational(8, 3))
        self.assertEqual(x**2, Rational(9, 16))
        self.assertEqual(x**-2, Rational(16, 9))
        self.assertEqual(-x, Rational(-3, 4))

    def test_order(self):
        w = Rational(3)
        x = Rational(6, 8)
        y = Rational("1.2")
        z = Rational(6, 5)
        self.assertLess(x,  y)
        self.assertLess(y,  2)
        self.assertLess(y,  2.1)
        self.assertGreaterEqual(z, y)
        self.assertGreaterEqual(y, x)
        self.assertGreater(2.1, y)
        self.assertLessEqual(z, y)
        self.assertLessEqual(x, y)
        self.assertNotEqual(w, 2)
        self.assertNotEqual(2, w)
        self.assertNotEqual(z, "garb")
        self.assertFalse(x > y)
        self.assertFalse(y <= x)
        self.assertFalse(.5 > y)
        self.assertFalse(1 > y)
        self.assertFalse(y < .5)
        self.assertFalse(y < 1)
        self.assertFalse(x == y)
        self.assertFalse(w == 2)
        self.assertFalse(w != 3)
        self.assertFalse(2 == w)
        self.assertFalse(3 != w)

    def test_errors(self):
        '''part of the code is is intended to expose errors!  We want
        to test those parts, too, to confirm they do cause errors.'''
        x = Rational(6, 8)
        y = Rational("1.2")
        # next line checks if Rational(2, 0) raises an AssertionError
        self.assertRaises(AssertionError, Rational, 2, 0)
        self.assertRaises(AssertionError, Rational, x, y)
        self.assertRaises(AssertionError, Rational, ".2", 2)

        with self.assertRaises(TypeError): # other approach: with statement
             junk = x < "garb"
        with self.assertRaises(TypeError):
             junk = x <= "garb"
        with self.assertRaises(TypeError):
             junk = x >= "garb"
        with self.assertRaises(TypeError):
             junk = "garb" < x

if __name__ == '__main__': # boilerplate to run tests
    unittest.main()
```

When you run this program, it counts the number of testing methods that succeed completely. When a test causes an error, the testing program does not itself fail to run, so it is safe to run against any code. As a result, we can pair the test-driven design with the incremental development we have illustrated several times.

In developing the class `Rational` , you could first just write the code for the the constructor and `__str__` . Those are all that are used by the first test.

Then if you run the testing program, you should see that one test is passed (or if not, you need to do debugging on just a couple of methods).

I really should have broken the testing methods down further. The first test method could be split into at least two, with the new first one only allowing an `__init__` heading like in Fraction, with two int parameters.

You could write that very basic version of the constructor, test it, and then move on to adding the other parameter types and formats allowed, and see if the next test passes....

The final file rational.py follows. It includes doc strings at the top for users of the class. Just below it is implementor documentation, the Rational instance invariant, so coders looking at the file know what is always true (and must be maintained in elaborations) for instances. Then we have the constructor and methods:

```python
""" Defines class Rational """

import operator

class Rational:
    ''' A class for storing a rational number reduced to lowest terms.
    Allow all the standard operations producing another Rational.
    Rational operands may be mixed with an int to produce another Rational,
    but operations with (potentially inexact) floats produce a float.
    A Rational may be explicitly created from int numerator and denominator
    values or from a fraction string or an int or a Rational.

    >>> Rational(4, -6)
    Rational('-2/3')
    >>> a = Rational('15/20')
    >>> a
    Rational('3/4')
    >>> print a
    3/4
    >>> -4*(2*a + 5)/10
    Rational('-13/5')
    >>> Rational('2.5')
    Rational('5/2')
    >>> a + 2.5
    3.25
    >>> a < 1
    True
    '''

    # Rational instance invariant:
    #   Data attributes _num and _den for numerator and denominator
    #   are in lowest terms with _den always positive
    #   and no Rational method changes these values.
    #   Rationals are intended to be immutable.

    def __init__(self, numerator, denominator=1):
        """Constructs a new Rational.  If denominator is omitted or 1,
        the result can be copied from a numerator Rational or can be
        parsed from a numerator string which may contain either a
        division symbol '/' or a decimal point, but not both.
        Otherwise numerator and denominator must be of type int.
        An error is raised if denominator is 0. """
        if isinstance(numerator, Rational): # check type
            assert denominator == 1 # assert forces error if False
            self._num = numerator._num
            self._den = numerator._den
            return
        if isinstance(numerator, str):
            assert denominator == 1
        if '/' in numerator:
            [numStr, denomStr] = numerator.split('/')
            numerator = int(numStr)
            denominator = int(denomStr)
        else:  # string for int or float
            s = numerator.strip()
            i = s.find('.')
            if i >= 0:  # float literal with decimal
                denominator = 10**(len(s) - i - 1)
                numerator = int(s[:i]+s[i+1:])
            else: # denominator still 1
                numerator = int(numerator)
    assert isinstance(numerator, int)
    assert isinstance(denominator, int)
    assert denominator != 0
    divisor = self.gcd(numerator, denominator) # simplify immediately
    if denominator // divisor < 0: # will make reduced to lowest terms
        divisor = -divisor         #    denominator be > 0
    self._num = numerator // divisor     # lowest terms
    self._den = denominator // divisor   #    and self._den > 0

### static support function

@staticmethod
def gcd(a,b): # note no self parameter for static method
    "Return the greatest common divisor of the pair (a,b)"
    while b != 0:  # Euclidean algorithm
        a, b = b, a%b
    return a

########  Arithmetic Methods  ########
# All binary operators allow the second operand to be an int. If it is
# a float or complex and then convert the Rational to a float first.

def __add__(self, frac2):
    """Returns self + frac2, as new Rational if possible."""
    if isinstance(frac2, (float, complex)): return float(self) + frac2
    if isinstance(frac2, int):
         frac2 = Rational(frac2)
    if not isinstance(frac2, Rational):
        return NotImplemented # need __radd__ for frac2 type
    return Rational(self._num * frac2._den + self._den * frac2._num,
                    self._den * frac2._den)

def __sub__(self, frac2):
    """Returns self - frac2, as new Rational if possible."""
    return self + -frac2

def __mul__(self, frac2):
    """Returns self * frac2, as new Rational if possible."""
```

```
        if isinstance(frac2, (float, complex)): return float(self) * frac2
        if isinstance(frac2, int):
             frac2 = Rational(frac2)
        if not isinstance(frac2, Rational): return NotImplemented
        return Rational(self._num * frac2._num, self._den * frac2._den)

    def __truediv__(self, frac2):
        """Returns self / frac2, as new Rational if possible."""
        if isinstance(frac2, (float, complex)): return float(self) / frac2
        if isinstance(frac2, int):
             frac2 = Rational(frac2)
        if not isinstance(frac2, Rational): return NotImplemented
        return Rational(self._num * frac2._den, self._den * frac2._num)

    def __neg__(self):
        """Returns a new Rational, negating self, -self."""
        return Rational(-self._num, self._den)

    def __abs__(self):
        """Returns the absolute value of the Rational, abs(self).
        """
        if self._num < 0:
            return -self
        return self

    def __pow__(self, n):
        """Returns self ** n, as new Rational if n is int.
        Matches type of n if n is float or complex"""
        if isinstance(n, (float, complex)): # allow tuple of type choices
            return float(self)**n
        if not isinstance(n, int): return NotImplemented
        if n >= 0:
            return Rational(self._num ** n, self._den ** n)
        return Rational(self._den ** -n, self._num ** -n)

    ### operations where the second operand only is a Rational (r prefix) ###
    # For x+y interpreter tries x.__add__(y) first. If that returns
    # NotImplemented, tries y.__radd__(x).  Error if that does not work either.

    def __radd__(self, frac2):
        """Returns frac2 + self, as new Rational if possible."""
        return self + frac2

    def __rsub__(self, frac2):
        """Returns frac2 - self, as new Rational if possible."""
        return -self + frac2

    def __rmul__(self, frac2):
        """Returns frac2 * self, as new Rational if possible."""
        return self * frac2
```

```
def __rtruediv__(self, frac2):
    """Returns frac2/self, as new Rational if possible."""
    return Rational(self._den, self._num) * frac2

########  Comparison Methods  ########
def __eq__(self, frac2):
    """Method translating == and != ; a Rational can equal an int.
    and we choose to allow equality to float or complex and
    to make the comparison legal but False for other types."""
    if isinstance(frac2, (float, complex)): # good idea? - like int
        return float(self) == frac2
    if isinstance(frac2, int):
        frac2 = Rational(frac2)
    return ( isinstance(frac2, Rational) and
            self._num == frac2._num and self._den == frac2._den )

def _ineq(self, frac2, op): # helper function for ordering dunders
    ''' return  self op frac2 for op among < > <= >='''
    if isinstance(frac2, float): # good idea? - like int
        return op(float(self), frac2)
    if isinstance(frac2, int):
        frac2 = Rational(frac2)
    if not isinstance(frac2, Rational): return NotImplemented
    # reduce to integer inequality, using fact: denominators are positive
    return op(self._num * frac2._den, self._den * frac2._num)

def __lt__(self, frac2):
    """Method translating < """
    return self._ineq(frac2, operator.lt)

def __le__(self, frac2):
    """Method translating <= """
    return self._ineq(frac2, operator.le)

def __gt__(self, frac2):
    """Method translating > """
    return self._ineq(frac2, operator.gt)

def __ge__(self, frac2):
    """Method translating >= """
    return self._ineq(frac2, operator.ge)

########  Type Conversion Methods  ########
def __float__(self):
    """Returns a float approximating the current Rational.
    """
    return self._num / self._den

def __int__(self):
        """Returns an integer by truncating the current Rational.
        """
        if self._num >= 0:
            return self._num // self._den
        return -(-self._num // self._den) # force toward 0

    def __bool__(self):
        """Returns True if the current Rational is nonzero.
        (Used for conversion to boolean.)
        """
        return self._num != 0 # standard for numbers

    def __str__(self):
        """Returns a simple string representation of the Rational. """
        if self._den == 1:
            return str(self._num)   # string for an integer
        else: # Uses the fact that a minus sign can only be in the numerator.
            return str(self._num) + '/' + str(self._den)

    def __repr__(self):
        """Returns a string representation that would evaluate to an
        equal Rational if typed into the shell.
        """
        return "Rational('{}')".format(self) # format uses __str__

    def __hash__(self): # so Rationals can be dictionary keys (Comp 271)
        return hash((self._num, self._den))  # tuples have a hash
```

Further comments:

The class doc string follows a common convention. It acts as if the class were being used in a Python shell to illustrate behavior.

The optional denominator parameter in `__init__` allows all the versions with a single actual parameter. For those cases we assert that the second parameter has not been changed from the default, 1.

You may want to play computer on an example to see that the behavior is correct if the constructor gets a string parameter like `'23.456'` .

You can see there are a lot more dunder methods than we discussed in More Advanced Topics (ootopics.html#more-advanced-topics)! I include all the cases where the second operand of a binary operation could be a Rational, while the first operand is some other type (`__rmul__` , ...).

This version shows some refactoring. I first wrote all the operation methods for ordering (< <= > >=) with a lot of duplicate code. Then I notice that the only difference was one integer comparison in the last line. Hence the introduction of the helping method `_ineq` , which also allows me to show off functions as parameters!

The comparison operators do not need a special __r...__ version. For instance the interpreter assumes `f < n` is equivalent to `n > f` , and `n == f` is equivalent to `f == n` .

For the last method, `__hash__` , wait for a data structures class for an explanation of how it makes dictionary entry access allowed in Python and fast.

Since we do have the ordering operator methods, we can use built-in methods that expect an ordering.

The file sortrationals.py, shows how we can simply sort a list of Rationals. It also shows how `__str__` and `__repr__` are both used:

```
'''Illustrating that giving ordering operators meaning
allows sort to work.'''

from rational import Rational

line = input('Enter rational numbers on one line:\n'
             ' like:  42 7.13 -10/12\n')

rList = [Rational(v) for v in line.split()]
rList.sort()

print('Now sorted list, as printed directly by Python:')
print(rList)

print('Now as a line of common math expressions:')
print(' '.join([str(v) for v in rList]))
```

This is a major enough example that we can show lots of comparisons when we get to Java!