`rational.py`: An example Python class

John W. Shipman

2012-01-17 20:56

Abstract

Describes an implementation of a class for representing rational numbers in the Python programming language.

This publication is available in Web form[1] and also as a PDF document[2]. Please forward any comments to **tcc-doc@nmt.edu**.

Table of Contents

1. Introduction

This document describes a Python module for working with rational numbers. It is intended as an example of a Python `class` for students new to object-oriented programming.

This publication assumes that the reader has had a general introduction to the construction of Python classes. In particular, you should know that the name of the class's constructor method is always `__init__`. Method names such as `__init__`, which start and end with two underbar (_) characters are called *special methods*.

[1] http://www.nmt.edu/tcc/help/lang/python/examples/rational/
[2] http://www.nmt.edu/tcc/help/lang/python/examples/rational/rational.pdf

This class makes heavy use of Python special methods to implement the common mathematical operators such as + and -. For example, when you have two instances *x* and *y* of some class, using the - (subtract) operator invokes the special method `__sub__(x, y)`.

Relevant online files:

- Source for the `rational.py` module[3].
- Source for the test driver, `rationaltest`[4].

1.1. Related publications

Here are some links to related Python documentation:

- *Python tutorials*[5]: Basic training for the Python beginner.
- *Python 2.7 Quick Reference*[6].

2. The class interface

An instance of the `Rational` class represents a rational number. Mathematically:

> A *rational number* is the ratio of two integers, the *numerator* and the *denominator*. The denominator cannot be zero.

In grade-school terms, a rational number is a fraction. Examples: 1/2; 113/355; 0/1 (which has the value zero).

In Python, you will need to import the `rational` module. Within this module, there is a *class constructor* that is invoked like this:

```
Rational(n, d)
```

where *n* is the numerator and `d` is the denominator.

This constructor returns an *instance* of the class, that is, an object that represents that specific rational value. Here is a conversational example:

```
>>> from rational import *
>>> half = Rational ( 1, 2 )
>>> print half
1/2
>>> twoSevenths = Rational ( 2, 7 )
>>> print twoSevenths * half
1/7
```

The above example shows that (1/2)*(2/7) equals 1/7.

The constructor will reduce values to their lowest terms:

```
>>> print Rational ( 50, 100 )
1/2
>>> print Rational ( 113, 355 )
```

[3] http://www.nmt.edu/tcc/help/lang/python/examples/rational/rational.py
[4] http://www.nmt.edu/tcc/help/lang/python/examples/rational/rationaltest
[5] http://www.nmt.edu/tcc/help/pubs/lang/pytut/
[6] http://www.nmt.edu/tcc/help/pubs/python27

```
113/355
>>> print Rational ( 2000, 12000 )
1/6
```

Operations allowed on `Rational` instances include:

r* + *s
Sum of two rationals *r* and *s*.

r* - *s
Difference of *r* and *s*.

r* * *s
Product of *r* and *s*.

r* / *s
Quotient of *r* divided by s.

float(*r*)
Returns the `float` value closest to the value of *r*.

str(*r*)
Return a representation of *r* as a string of type `str`.

***R*.mixed()**
For a `Rational` instance *R*, returns the value as a mixed fraction in string form. Examples: `"17"`, `"1/10"`, `"3 and 1/7"`.

More examples:

```
>>> third=Rational(1,3)
>>> fifth=Rational(1,5)
>>> print third * fifth
1/15
>>> print third + fifth
8/15
>>> print third-fifth
2/15
>>> print fifth/third
3/5
>>> print str(fifth)
1/5
>>> print float(fifth)
0.2
>>> print float(third)
0.333333333333
```

The module also provides one ordinary method named `mixed` that returns a string representing an instance as a mixed fraction. Example:

```
>>> badPi = Rational(22,7)
>>> print badPi.mixed()
3 and 1/7
>>> properFraction = Rational(3,5)
>>> print properFraction.mixed()
3/5
>>> wholeNum = Rational(8,2)
```

```
>>> print wholeNum
4/1
>>> print wholeNum.mixed()
4
>>> zero = Rational ( 0, 12345 )
>>> print zero.mixed()
0
```

3. Contents of the `rational.py` module

The actual code of the `rational.py` module is displayed here, with commentary. This document is therefore an example of lightweight literate programming; see the author's *Lightweight literate programming* page[7] for more information about the tools and techniques used in this document.

3.1. Prologue

The `rational.py` file starts with a module documentation string that describes the class interface. This is basically a restatement of the interface as described above, using Cleanroom intended functions to document each attribute and method. For more information on the Cleanroom methodology, see the author's Cleanroom page[8].

`rational.py`

```
'''rational.py:  Module to do rational arithmetic.

  For full documentation, see
http://www.nmt.edu/tcc/help/lang/python/examples/rational/.
```

To simplify fractions (for example, reducing 4/8 to 1/2), we will need a function to find the greatest common divisor of two numbers.

`rational.py`

```
  Exports:
    gcd ( a, b ):
      [ a and b are integers ->
          return the greatest common divisor of a and b ]
```

Here is the class constructor.

`rational.py`

```
    Rational ( a, b ):
      [ (a is a nonnegative integer) and
        (b is a positive integer) ->
          return a new Rational instance with
          numerator a and denominator b ]
```

We make the numerator and denominator values available outside the class as visible attributes named `n` and `d`, respectively.

`rational.py`

```
    .n:    [ the numerator ]
    .d:    [ the denominator ]
```

[7] http://www.nmt.edu/~shipman/soft/litprog
[8] http://www.nmt.edu/~shipman/soft/clean/

We implement all four of the common mathematical operators: `+`, `-`, `*`, and `/`. These operations are implemented by defining methods that use certain special names, such as `__add__` for addition.

rational.py

```
    .__add__(self, other):
      [ other is a Rational instance ->
          return the sum of self and other as a Rational instance ]
    .__sub__(self, other):
      [ other is a Rational instance ->
          return the difference of self and other as a Rational
          instance ]
    .__mul__(self, other):
      [ other is a Rational instance ->
          return the product of self and other as a Rational
          instance ]
    .__div__(self, other):
      [ other is a Rational instance ->
          return the quotient of self and other as a Rational
          instance ]
```

The built-in Python functions `str()` and `float()` are also implemented using special method names.

rational.py

```
    .__str__(self):
      [ return a string representation of self ]
    .__float__(self):
      [ return a float approximation of self ]
```

Finally, the `.mixed()` method that converts an instance to a string displaying the fraction as a mixed fraction:

rational.py

```
    .mixed(self):
      [ return a string representation of self as a mixed
        fraction ]
'''
```

3.2. The `gcd()` function

This function implements Euclid's algorithm for finding the greatest common divisor of two numbers *a* and *b*.

rational.py

```
def gcd ( a, b ):
    '''Greatest common divisor function; Euclid's algorithm.

      [ a and b are integers ->
          return the greatest common divisor of a and b ]
   '''
```

Euclid's algorithm is easily defined as a recursive function. See *Structure and Interpretation of Computer Programs* by Abelson and Sussman, ISBN 0-262-01153-0, pp. 48-49.

- The GCD of any number *x* and zero is zero.
- The GCD of any two nonzero numbers *a* and *b* is the same as `GCD(`*b*`, `*a*` modulo `*b*`)`.

Defined recursively, this amounts to:

rational.py

```
    if  b == 0:
        return a
    else:
        return gcd(b, a%b)
```

3.3. `class Rational`

Here begins the actual class definition.

rational.py

```

class Rational:
    """An instance represents a rational number.
    """
```

3.4. `Rational.__init__()`: The constructor

The constructor takes two external arguments, the numerator and the denominator. It finds the GCD of those two numbers and divides both of them by that GCD, to reduce the fraction to its lowest terms. It then stores the reduced numerator and denominator in the instance namespace under the attribute names `n` and `d`.

rational.py

```
    def __init__ ( self, a, b ):
        """Constructor for Rational.
        """
        if  b == 0:
            raise ZeroDivisionError, ( "Denominator of a rational "
                "may not be zero." )
        else:
            g  =  gcd ( a, b )
            self.n  =  a / g
            self.d  =  b / g
```

3.5. `Rational.__add__()`: Implement the addition (+) operator

This method will be invoked to perform the "`+`" operator whenever a `Rational` instance appears on the left side of that operator. To simplify life, we assume here that the operand on the right side is also a `Rational` instance.

Basically, what we are doing is adding two fractions. Here is the algebraic rule for adding fractions:

$$\frac{n_1}{d_1} + \frac{n_2}{d_2} = \frac{n_1 \times d_2 + n_2 \times d_1}{d_1 \times d_2} \tag{1}$$

In this method, `self` is the left-hand operand and `other` is the right-hand operand.

rational.py

```
    def __add__ ( self, other ):
        """Add two rational numbers.
        """
```

```
        return Rational ( self.n * other.d + other.n * self.d,
                          self.d * other.d )
```

3.6. `Rational.__sub__()`: Implement subtraction

This method is called when a `Rational` instance appears on the left side of the "`-`" operator. The right-hand operand must be a `Rational` instance as well. See Section 3.5, "`Rational.__add__()`: Implement the addition (+) operator" (p. 6) for the algebra of this operation.

rational.py

```
    def __sub__ ( self, other ):
        """Return self minus other.
        """
        return Rational ( self.n * other.d - other.n * self.d,
                          self.d * other.d )
```

3.7. `Rational.__mul__()`: Implement multiplication

Here's the formula for multiplying two fractions:

$$\frac{n_1}{d_1} \times \frac{n_2}{d_2} = \frac{n_1 \times n_2}{d_1 \times d_2} \tag{2}$$

rational.py

```
    def __mul__ ( self, other ):
        """Implement multiplication.
        """
        return  Rational ( self.n * other.n, self.d * other.d )
```

3.8. `Rational.__div__()`: Implement division

Here's the formula for dividing one fraction by another:

$$\frac{n_1/d_1}{n_2/d_2} = \frac{n_1 \times d_2}{n_2 \times d_1} \tag{3}$$

rational.py

```
    def __div__ ( self, other ):
        """Implement division.
        """
        return  Rational ( self.n * other.d, self.d * other.n )
```

3.9. `Rational.__str__()`: Convert a rational to a string

The `__str__` method of a class is invoked whenever an instance of that class must be converted to a string. This happens, for instance, when you print an instance with a `print` statement, or when you use the `str()` function on an instance.

rational.py

```
    def __str__ ( self ):
        '''Display self as a string.
        '''
        return "%d/%d" % ( self.n, self.d )
```

3.10. `Rational.__float__()`: Implement the `float()` function

This method is called whenever Python's built-in `float()` function is called to convert an instance of the `Rational` class. To do this, we convert the numerator and the denominator to `float` type and then use a floating division.

rational.py

```
    def __float__ ( self ):
        """Implement the float() conversion function.
        """
        return  float ( self.n ) / float ( self.d )
```

3.11. `Rational.mixed()`: Display as a mixed fraction

This method is used to convert a rational number (which may be an improper fraction) to a "mixed fraction". The general form of a mixed fraction is a phrase of the form "*w* and *n*/*d*" For example, the improper fraction 22/7 is equivalent to the mixed fraction "3 and 1/7".

The result is returned as a string. There are three cases:

- If the denominator is 1, we display just the whole-number part. Example: 17/1 becomes simply "17".
- If the whole-number part is zero but the fractional part is not, we'll display only the fractional part. Example: 5/6 becomes "5/6", not "0 and 5/6".
- In the general case, there is both a whole-number part and a fractional part. Example: 22/7 becomes "3 and 1/7".

First we find the whole-number part and the numerator of the fractional part (the denominator of the fractional part will be the same as the denominator of the original rational). Python conveniently provides the `divmod()` function, which provides both the quotient and the remainder.

rational.py

```
    def mixed ( self ):
        """Render self as a mixed fraction in string form.
        """
        #-- 1 --
        # [ whole  :=  self.n / self.d, truncated
        #   n2  :=  self.n % self.d ]
        whole, n2  =  divmod ( self.n, self.d )
```

Then we separate the three cases.

rational.py

```
        #-- 2 --
        # [ if self.d == 1 ->
        #     return str(self.n)
        #   else if whole == zero ->
        #     return str(n2)+"/"+str(self.d)
        #   else ->
        #     return str(whole)+" and "+str(n2)+"/"+str(self.d) ]
```

```
        if  self.d == 1:
            return str(self.n)
        elif whole == 0:
            return "%s/%s" % (n2, self.d)
        else:
            return "%s and %s/%s" % (whole, n2, self.d)
```

3.12. `rationaltest`: A small test driver

This script exercises the class's functions.

rationaltest

```
#!/usr/bin/env python
#=====================================================================
# rationaltest:  A test driver for the rational.py module.
#---------------------------------------------------------------------

import sys
from rational import *

def main():
    """
    """
    generalTests()
    mixedTests()
    errorTests()

def generalTests():
    """Test basic functionality
    """
    print "  -- Ambition/distraction/uglification/derision"
    third=Rational(1,3)
    print "Should be 1/3:", third
    fifth=Rational(1,5)
    print "Should be 1/5:", fifth
    print "Should be 8/15:", third + fifth
    print "Should be 1/15:", third * fifth
    print "Should be 2/15:", third-fifth
    print "Should be 3/5:", fifth/third

    print "  -- float()"
    print "Should be 0.2:", float(fifth)
    print "Should be 0.3333...:", float(third)

def mixedTests():
    """Test the .mixed() method cases
    """
    print "  -- mixed()"
    badPi = Rational(22,7)
    print "Should be '3 and 1/7':", badPi.mixed()

    properFraction = Rational(3,5)
    print "Should be 3/5:", properFraction.mixed()
```

```
    wholeNum  =  Rational ( 8,2 )
    print "Should be 4:", wholeNum.mixed()

    zero  =  Rational(0,1)
    print "Should be 0:", zero.mixed()

def errorTests():
    """Test error conditions
    """
    try:
        badIdea = Rational ( 5, 0 )
        print "Fail: didn't detect zero denominator."
    except ZeroDivisionError, detail:
        print "Pass: Zero denominator blowed up real good."

#=====================================================================
# Epilogue
#---------------------------------------------------------------------

if __name__ == "__main__":
    main()
```