CSCI 235, Programming Languages, Python, Exercise 3

Deadline: Sunday 03.11.2019, 21.00

Goal of this exercise is that you understand how to build classes in Python. The same principles that apply in C^{++} and Java also apply in Python. A class can have invariants, which must be established by the constructors, and maintained by all methods of the class. Unfortunately, Python does not have private fields, so the user of the class needs some self control.
Download the files **vector.py** and **matrix.py** from Moodle. Your task is to create a class **Rational** in a file **rational.py** that can be used with **Matrix** and **Vector**.

1. First write a function `gcd(n1,n2)` in a new file **rational.py**. This function **must be based** on the Euclidean algorithm.

   During earlier versions of this exercise, a lot of problems were caused by not carefully tested `gcd` functions. Test very carefully. Untested code is unwritten code! The `gcd` function must work with negative arguments, and also when one of the arguments is zero. Test all of this. The `gcd` can be negative when one of the arguments is negative, but your `gcd` function must always compute a correct answer when a `gcd` exists.

   When both arguments are zero, ther is no `gcd`. In that case, your function must:

   ```
   raise ArithmeticError( "gcd(0,0) does not exist" )
   ```

2. After `gcd`, you can add

   ```
   from numbers import *

   class Rational( Number ) :
   # This means that Rational inherits from Number.

   def __init__( self, num, denom = 1 ) :
      self. num = num
      self. denom = denom

      self. normalize( )
   ```

3. In order for this code to work, you first have to write `normalize( self )`. This function must establish the class invariants. These are:

   - `num` and `denom` have no common factors except for 1 and −1.
   - `denom` is not negative.

   Use `//` when dividing out common factors. If you use `/` the result will be `float`.

4. Write `__repr__( self )`. Rationals that are integers (whose `denom` equals 1) should be printed as integers:

   ```
   >>> Rational(1,7)
   1 / 7
   >>> Rational(-1,-7)
   1 / 7
   >>> Rational(-7,-1)
   7
   ```

5. Complete the following methods of `class Rational`:

   ```
   def __neg__( self ) :

   def __add__( self, other ) :
   def __sub__( self, other ) :
   ```

   Both `add` and `sub` must be written in such a way that they work when `other` is not rational. Use `not isinstance( other, Rational )` for testing this.

   ```
   def __radd__( self, other ) :
   def __rsub__( self, other ) :
   ```

   The reverse methods. Again use `not isinstance( other, Rational )`.

6. Also, write the following methods:

   ```
   def __mul__( self, other ) :
   def __truediv__( self, other ) :
   def __rmul__( self, other ) :
   def __rtruediv__( self, other ) :
   ```

   As with the addition operators, these operators must work when `other` is not rational.

7. Implement the equality and inequality operators for natural numbers:

```
def __eq__( self, other ) :
def __ne__( self, other ) :
```

Again, these methods must also work when `other` is not `Rational`. There is no need to write reverse methods, because the interpreter will automatically try `eq` and `ne` both ways.

It is sufficient to compare `num` and `denom`, because rationals are completely normalized. (You implemented this under 3)

8. Implement the remaining comparison operators:

   ```
   def __lt__( self, other ) :
   def __gt__( self, other ) :
   def __le__( self, other ) :
   def __ge__( self, other ) :
   ```

   All four methods must work when `other` is not `Rational`. Again it is sufficient to test `other` for being `Rational`, because the interpreter knows that `lt` is reverse of `gt`, and `le` reverse of `ge`.

9. Now, it will be possible to do some testing with our rational numbers. We can take advantage of the fact that Python uses unbounded precision integers.

   Create a file **mytests.py** starting with

   ```
   from matrix import *
   from vector import *
   from rational import *
   ```

   In this file, define a function `def tests( ):` that does the following:

   - Compute (and print) the product:

     $$\begin{pmatrix} \frac{1}{2} & \frac{1}{3} \\ -\frac{2}{7} & \frac{2}{8} \end{pmatrix} \times \begin{pmatrix} -\frac{1}{3} & \frac{2}{7} \\ \frac{2}{5} & -\frac{1}{7} \end{pmatrix}.$$

     Use `@` for the matrix product.
   - Compute the inverse of

     $$\begin{pmatrix} \frac{1}{2} & \frac{1}{3} \\ -\frac{2}{7} & \frac{2}{8} \end{pmatrix}$$

   - Verify the following properties:
     Matrix multiplication is associative:

     $$(m_1 \times m_2) \times m_3 = m_1 \times (m_2 \times m_3),$$

Matrix multiplication with addition is distributive:

$m_1 \times (m_2 + m_3) = m_1 \times m_2 + m_1 \times m_3$ and $(m_1 + m_2) \times m_3 = m_1 \times m_3 + m_2 \times m_3$,

Matrix multiplication corresponds to composition of application:

$$m_1(m_2(v)) = (m_1 \times m_2)(v),$$

Determinant commutes over multiplication:

$$\det(m_1).\det(m_2) = \det(m_1 \times m_2).$$

Do this by comparing the results using `==`, and raising a `ValueError` if the results are different.

- Finally, verify that the inverse of matrix is indeed its inverse:

$$m \times \text{inv}(m) = I \text{ and } \text{inv}(m) \times m = I.$$

Again, use `==` and raise a `ValueError` if results are different.

Note: Don't mix integers and rationals in a matrix. It causes problems with division. Because Python is dynamically typed, it will not use unbounded precision on the integers and convert them to floats.