CSE 30
Programming Abstractions: Python
Programming Assignment 4

In this project you will create a class called Rational in a file called rational.py, that represents *rational numbers*. You will then import the rational.py module into a program called ContinuedFractions.py, which will evaluate rational numbers associated with (finite) continued fractions.

The Rational Class
As you know, a rational number is simply the ratio of two integers, called *numerator* and *denominator*, respectively, i.e., a fraction. The Rational class will encapsulate these two integers, and define 18 functions described below, implementing rational arithmetic in terms of pure integer arithmetic.

Rational numbers will always be stored as fractions in lowest terms, so the numerator and denominator have no common factors. In other words, the greatest common divisor (GCD) of numerator and denominator will equal 1. You can accomplish this by creating a helper function called $\_\text{gcd}(a, b)$ within the rational.py module, then calling _gcd() from $\_\_\text{init}\_\_(n, d)$, which initializes a new Rational object with numerator n and denominator d. The default value of d will be 1, so that $\text{Rational}(6)$ returns the Rational object representing $6/1$. Also, the denominator will always be stored as a positive integer. Thus, if the user (also called the *client*) of the Rational class does $\text{Rational}(5, -2)$, then the resulting object will have -5 as numerator and 2 as denominator.

It is assumed that everyone knows how to add, subtract, multiply and divide fractions, but just in case, the following formulas are valid.

$$\left(\frac{a}{b}\right) + \left(\frac{c}{d}\right) = \left(\frac{ad + bc}{bd}\right)$$

$$\left(\frac{a}{b}\right) - \left(\frac{c}{d}\right) = \left(\frac{ad - bc}{bd}\right)$$

$$\left(\frac{a}{b}\right) * \left(\frac{c}{d}\right) = \left(\frac{ac}{bd}\right)$$

$$\left(\frac{a}{b}\right) \div \left(\frac{c}{d}\right) = \left(\frac{ad}{bc}\right)$$

In addition to these operators, you will implement the comparison operators: $<, <=, >, >=$ and $!=$. For an example of operator overloading, see the Complex class in

https://classes.soe.ucsc.edu/cse030/Spring21/Examples/Classes/complex.py

which was discussed in lecture. For further information on operator overloading in Python, see the articles

https://www.askpython.com/python/operator-overloading-in-python , and
https://docs.python.org/3/reference/datamodel.html#emulating-numeric-types

You will also define a function called inverse(self), which returns a Rational object representing the multiplicative inverse of $self$, as well as string representations __str__(self) and __repr__(self). The

specifications and doc strings for all of these functions can be seen by examining the client program TestRational.py and its output test-out, which are posted on the webpage at

https://classes.soe.ucsc.edu/cse030/Spring21/Examples/pa4.

Altogether, you will implement the following 17 member functions in class Rational.

```
__init__(self, n, d=1)  # initialize a Rational object
numer(self)             # return the numerator (use the @property decorator)
denom(self)             # return the denominator (use the @property decorator)
__str__(self)           # see the doc string in help() ouput in test-out
__repr__(self)             .
__float__(self)            .
__eq__(self, other)        .
__ne__(self, other)        .
__lt__(self, other)        .
__le__(self, other)        .
__gt__(self, other)        .
__ge__(self, other)        .
__add__(self, other)       .
__sub__(self, other)       .
__mul__(self, other)       .
__truediv__(self, other)   .
inverse(self)              .
```

The last function you'll need to implement is $\_\gcd(a, b)$, described above. See This function should be within the rational.py module, but not inside the Rational class. Its name begins with an underscore "_" which, by convention, indicates to a client that it is not part of the Rational API (Applications Programming Interface). This is just a fancy way of saying that clients are not expected to call this function, and it is only used from within the Rational class. Similarly, the numerator and denominator should be stored in fields whose names have the underscore "_" prefix. The methods numer() and denom() then merely return these fields. These functions should be defined with the @property decorator so that a client can leave off the parentheses () when calling them.

Create the Rational class as described above and test it thoroughly before you move on to the next step in this project. A non-thorough is embodied in the program TestRational.py. Include your Rational.py and TestRational.py and test-out in the same directory, then do

```
$ python3 TestRational.py > my-test-out
$ diff my-test-out test-out
```

If the output of diff gives no output, then the two files test-out and my-test-out match exactly. I would consider this to be a minimal test of correctness for your Rational class.

Continued Fractions

A (finite) *continued fraction* is an algebraic expression of the form

$$x_0 + \cfrac{1}{x_1 + \cfrac{1}{x_2 + \cfrac{1}{\ddots + \cfrac{1}{x_n}}}}$$

where $x_0, x_1, x_2, \ldots, x_n$ are integers. Continued fractions and their properties have been studied by mathematicians for centuries (going back to Euclid in fact). A standard notation for the above expression is $[x_0, x_1, x_2, \ldots, x_n]$, which should remind us of a Python list. This is fortunate, since we will encode continued fractions in exactly this way. See the Wikipedia article

https://en.wikipedia.org/wiki/Continued_fraction

if you want to see some of the many things known about continued fractions. As an exercise, verify that $[1,1,1,1,1] = 8/5$, and that $[1,1,1,1,1,1] = 13/8$. A few more examples should convince you that if

$$x_0 = x_1 = x_2 = \cdots = x_n = 1,$$

then

$$[x_0, x_1, x_2, \ldots, x_n] = \frac{F_{n+2}}{F_{n+1}}$$

where F_n is the n^{th} Fibonacci number.

Observe that each denominator in a continued fraction is itself a continued fraction, giving the following recursive formula.

$$[x_0, x_1, x_2, \ldots, x_n] = x_0 + \frac{1}{\left(x_1 + \dfrac{1}{x_2 + \dfrac{1}{\ddots + \dfrac{1}{x_n}}} \right)} = x_0 + \frac{1}{[x_1, x_2, \ldots, x_n]}$$

Thus we can define the symbol $[x_0, x_1, x_2, \ldots, x_n]$ recursively as

$$[x_0, x_1, x_2, \ldots, x_n] = \begin{cases} x_0 & \text{if } n = 0 \\ x_0 + \dfrac{1}{[x_1, x_2, \ldots, x_n]} & \text{if } n \geq 1 \end{cases}$$

Create a Python program called ContinuedFractions.py which contains a recursive function CF(L) that takes as input a list of ints L, then returns a Rational object representing the rational number corresponding to the continued fraction given by L. The program will of course import the module rational.py to accomplish this. This import statement should be of the type

```
from rational import *
```

so that you don't need to use the prefix "`rational.`" on every Rational operation. Your program will include a function main() that opens two files, one for reading and one for writing. The input file will contain any number of lines, each consisting of a space separated list of integers, and nothing else. Each line represents one continued fraction.

For each such line, separate it into a list of strings (using the string function split()), then convert the list of strings to a list of ints (i.e., the integers from one line in the file). Turn that list into a Rational object using your CF() function, then print out the Rational as a string (using str()). Also print out the decimal

equivalent of the Rational object (using the decimal module, not the `float()` function), correct to (at most) 100 digits of precision. See

https://docs.python.org/3/library/decimal.html

for details on how to use the decimal module. The built in function map(), described here

https://docs.python.org/3/library/functions.html#map

will be useful in turning your initial list of strings into a list of ints. All of this is repeated for each line of the input file. Also, all print statements will be directed to the output file. A matched pair of input-output files called in1 and out1 are included in

https://classes.soe.ucsc.edu/cse030/Spring21/Examples/pa4

Another example called LineReverse.py, and its input-output files (called line-in and line-out, respectively) are also included in the above folder. This program does something else to the lines of a file, which you will see when you study it. You may use any part of it in ContinuedFractions.py. A sample command line session is included below to illustrate proper handling of bad input. Assume here that rational.py, ContinuedFractions.py, in1 and out1 are all contained in the current directory.

```
$ python3 ContinuedFractions.py foo
Usage: $ python3 ContinuedFractions.py <input file> <output file>
$ python3 ContinuedFractions.py foo bar happy
Usage: $ python3 ContinuedFractions.py <input file> <output file>
$ python3 ContinuedFractions.py foo bar
[Errno 2] No such file or directory: 'foo'
Usage: $ python3 ContinuedFractions.py <input file> <output file>
$ python3 ContinuedFractions.py in1 myout1
$ diff out1 myout1
$
```

If the output of diff is nothing, as pictured above, it means that your program output is correct in this case.

Submit the files

rational.py	written by you
ContinuedFractions.py	written by you
TestRational.py	given, do not alter

to pa4 on Gradescope. There are a lot of moving parts in this assignment. No one of them is very difficult, but don't wait until the last minute to ask for help. Start early.