Humboldt University of Berlin

Einführung in das wissenschaftliche Rechnen

Documentation of Fraction Application Programming Interface and Command Line Interface Calculator

Christian Parpart & Kei Thoma

May 17, 2019

Contents

1 Introduction: Complexity One Horizon

> "Life is really simple, but we insist on making it complicated."
>
> *Confucius*

Our goal was to write a proper fraction calculator. For that, however, we needed to implement the Euclidean algorithm which computes the greatest common divisor. This algorithm is admittingly fast, but certainly not $\mathcal{O}(1)$ fast. So we coded one. Greatest common divisor at complexity one.

This was done by precomputing all greatest common divisor from 1 to n and saving the results to a local file. While the Euclidean algorithm often need multiple iteration to find the solution, looking up a value is always a single step.

2 Euclidean Algorithm Library

The goal of this module is to implement the Euclidean algorithm, which finds the greatest common divisor, with complexity O(1). This is realized by precomputing (iteratively) all greatest common divisor from 1 to n.

We have left another (recursive) version of the algorithm in the module in case of an emergency.

Lastly, this module includes a function to calculate the least common multiple with the help of the aforementioned Euclidean algorithm.

2.1 class GreatestCommonDivisor

This class implements the Euclidean algorithm with ensuring that smaller greatest common divisors have complexity O(1), by precomputing all greatest common divisors between 1 and n (with n being chosen at the constructor). The precomputed greatest common divisors are stored in a local file for faster instantiation later.

2.1.1 def __init__(self, _filename, _dim)

Arguments

1. _filename (string): the file name

2. _dim (int): the largest number to precomputed

Description Constructing the greatest common divisor object by either optionally precomputing all greatest common divisor between 1 and _dim, and writes them to disk (_filename) for future acess.

2.1.2 def compute(_a, _b)

Arguments

1. _a (int): the fist integer
2. _b (int): the second integer

Returns

- (int): the result of the Euclidean algorithm of _a and _b

Description She returns the result of the Euclidean algorithm by actually iteratively computing.

2.1.3 def write_cache(self)

Returns

- None

Description Member method to populate a cache for O(1) access.

2.1.4 def gcd(self, _a, _b)

Arguments

1. _a (int): the fist integer
2. _b (int): the second integer

Returns

- (int): the greatest common divisor of _a and _b

Description Computes GCD of a and b either in O(1) complexity if in range of precomputation, or iteratively.

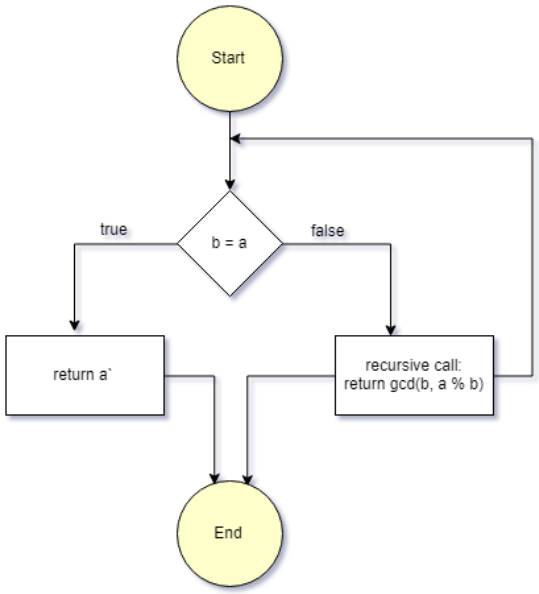

Figure 1: The recursive Euclidean algorithm

2.2 Free Functions

2.2.1 def recursive_euclidean_algorithm(a, b)

Arguments

1. `first_number` (int): the first integer, a; negative values are accepted
2. `second_number` (int): the second integer, b; negative values are accepted

Returns

- (int): the greatest common divisor found via the recursive euclidean algorithm

Description Given two integers, she finds the greatest common divisor via the recursively implemented euclidean algorithm.

The algorithm itself starts with two integers a and b. If $b = 0$ then a is returned and the recursive loop stops. In any other case, this function is called again, but the arguments are modified in the following manner

$$b \mapsto \text{first argument} \qquad a \mod b \mapsto \text{second argument},$$

or if one prefers to read the statement in code (see also figure 1)

```
1 def euclidean_algorithm(a, b):
2 return a if b_5 == 0 else euclidean_algorithm(b, a % b)
```

Worked Example of the Algorithm Let $a = 195$ and $b = 1287$. Following the algorithm above, we have

$$\begin{array}{llll} \text{Step } 0 & a_0 = 195 & b_0 = 1287 & \\ \text{Step } 1 & a_1 = 1287 & b_1 = 195 \mod 1287 & = 195 \\ \text{Step } 2 & a_2 = 195 & b_2 = 1287 \mod 195 & = 117 \\ \text{Step } 3 & a_3 = 117 & b_3 = 195 \mod 117 & = 78 \\ \text{Step } 4 & a_4 = 78 & b_4 = 117 \mod 78 & = 39 \\ \text{Step } 4 & a_5 = 39 & b_5 = 78 \mod 39 & = 0 \end{array}$$

Since $b = 0$, the algorithm is stopped and $a_5 = 39$ is returned.

2.2.2 def least_common_multiple(a, b)

Arguments

1. `first_number` (int): the first integer, a; negative values are accepted
2. `second_number` (int): the second integer, b; negative values are accepted

Returns

- (int): the least common multiple calculated with the help of the euclidean algorithm and the formula

Description She calculates the least common multiple using the result of the euclidean algorithm and the following formula

$$\operatorname{lcm}(a, b) = \frac{|a \cdot b|}{\gcd(a, b)}.$$

3 Fraction API

The Fraction class in `fraction.py` implements a fraction, i.e. concepts such as $\frac{1}{2}$ or $-\frac{2}{3}$, mathematically correctly. For this endeavor, Fraction saves three pseudo-private attributes representing the unsigned numerator, the unsigned denominator and finally the sign of the fraction.

After an instance of Fraction is initialized, it is automatically reduced properly to the most minimal form, e.g. $\frac{8}{12}$ naturally becomes $\frac{2}{3}$, with the help of the euclidean algorithm.

Finally, to allow some easy way to handle this class, few build-in operators such as the absolute function and binary addition were overloaded.

3.1 def __init__(numerator, denominator)

Arguments

1. `numerator` (int): the numerator; negative values are allowed, but is then saved as a positive integer at `numerator_`
2. `denominator` (int): the denominator; negative values are allowed, but is then saved as a positive integer at `denominator_`; if no argument is passed, it defaults to 1

Note that even though `numerator_` and `denominator_` are always positive, the sign of the Fraction is determined at the point of initalization and is saved under the boolean attribute `sign_`.

Raises

- `ZeroDivisionError`: if 0 is passed as the parameter for the denominator

Description

The constructor initializes the fraction object with the given numerator and denominator. As mentioned before, the sign of the fraction is saved separately as a boolean (True for negative, False for positive fractions and zero).

3.2 Get Attribute Functions

Fortunataly or unfortunataly depending on one's perspective about dynamic languages, Python does not allow private attributes or methods. However, we don't want that the three attributes, `numerator_`, `denominator_`, and `sign_`, are modifiable from the outside of the Fraction class. Therefore, this class provides three methods, `get_numerator()`, `get_denominator()`, and `get_sign()`, which simply returns the respective attribute.

3.3 def __pos__(self)

Returns

- (self): returns the unchanged self

Description

Overloading the unary plus operator is not very exciting. The fraction object is unchanged and returned immediately.

3.4 def __neg__(self)

Returns

- (self): returns self, but negates the sign, i.e. True becomes False and False becomes True; if the numerator was 0, the sign is unchanged

Description

A little more exciting than the unary plus. This function changes the `sign_` to True if the fraction was positive and to False if the fraction was negative. If the numerator was 0, she returns self without changing the sign.

3.5 def __abs__(self)

Returns

- (self): returns self, but the sign is changed to False

Description

To determine the absolute value of the fraction, she either returns the object unchanged if the fraction was already negative or uses the `__neg__()` to return the positive fraction.

3.6 def __add__(self, other)

Arguments

1. self (Fraction): the fraction on the right side; first summand
2. other (Fraction): the fraction on the left side; second summand

Returns

- (Fraction): the sum of self and other as a new instance of the Fraction class

Description

This magic method overloads the binary infix plus and returns the sum of two fractions as a new instance. The new fraction is calculated in the following manner

$$\frac{a}{b} + \frac{c}{d} = \frac{ad + bc}{bd}. \tag{1}$$

She does not need to find the greatest common denominator here, because the new Fraction will be reduced properly at the constructor.

3.7 def __sub__(self, other)

Arguments

1. self (Fraction): the fraction on the right side; the subtrahend
2. other (Fraction): the fraction on the left side; the minuend

Returns

- (Fraction): the difference of self and other as a new instance of the Fraction class

Description

The antithesis of the binary infix plus operation. As such, she uses the binary plus in conjuction with the unary negative to return the difference of the two given fraction as a new fraction object.

3.8 def __mul__(self, other)

Arguments

1. self (Fraction): the fraction on the right side; the first factor
2. other (Fraction): the fraction on the left side; the second factor

Returns

- (Fraction): the product of self and other as new instance of the Fraction class

Description

She overloads the binary infix multiplication for the Fraction objects. For the calculation, this formula is used

$$\frac{a}{b} \cdot \frac{c}{d} = \frac{ac}{bd}. \tag{2}$$

Again, we do not need to reduce here, because the magic is taken care at initalization.

3.9 def __str__(self)

Returns

- (string): the format of the string is `a/b`

This function returns the attributes of a Fraction instance as a readable string. The `print()` function uses her to print the Fraction object neatly to the console.

4 Fraction Calculator CLI

4.1 def interpret(user_input)

Arguments

1. user_input (string): the string entered to the console by the user

Returns

- (Fraction): the Fraction object constructed by the user's input

Description

This function constructs a fraction object according to the string passed.

4.2 def main()

Returns

- None

Description

The main function of the fraction calculator. When called, the user is asked to input two fraction and their sum is printed on the console. The user can then exit the module by typing any word starting with an n to exit.

5 Future Improvements

One of our immediate goals in the comming future would be to equip the currently limited fraction calculator with more arithmetic options such as subtraction or power. The latter would require an expansion to the fraction class.