



HUMBOLDT UNIVERSITY OF BERLIN

EINFÜHRUNG IN DAS WISSENSCHAFTLICHE RECHNEN

# Documentation of Fraction Application Programming Interface and Command Line Interface Calculator

*Christian Parpart & Kei Thoma*

*in Club Mate & Coffee We Trust*

May 17, 2019

## Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Euclidean Algorithm Library</b>	<b>3</b>
2.1	euclidean_algorithm(a, b) . . . . .	3
2.2	least_common_multiple(a, b) . . . . .	4
<b>3</b>	<b>Fraction API</b>	<b>4</b>
3.1	__init__(numerator, denominator) . . . . .	4
3.2	Get Attribute Functions . . . . .	5
3.3	__pos__(self) . . . . .	5
3.4	__neg__(self) . . . . .	5
3.5	__abs__(self) . . . . .	5
3.6	__add__(self, other) . . . . .	6
3.7	__sub__(self, other) . . . . .	6
3.8	__mul__(self, other) . . . . .	7
3.9	__str__(self) . . . . .	7
<b>4</b>	<b>Fraction Calculator CLI</b>	<b>7</b>
<b>5</b>	<b>Future Improvements</b>	<b>7</b>
5.1	Complexity One Horizon . . . . .	7

## 1 Introduction

## 2 Euclidean Algorithm Library

In this module, we first implemented the well known euclidean algorithm which finds the greatest common divisor given two integers. From there, we use the result of the above mentioned algorithm to calculate the least common multiple.

### 2.1 `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

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

or if one prefers to read the statement in code

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

#### Worked Example of the Algorithm

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

Step 0	$a_0 = 195$	$b_0 = 1287$	
Step 1	$a_1 = 1287$	$b_1 = 195 \bmod 1287$	$= 195$
Step 2	$a_2 = 195$	$b_2 = 1287 \bmod 195$	$= 117$
Step 3	$a_3 = 117$	$b_3 = 195 \bmod 117$	$= 78$
Step 4	$a_4 = 78$	$b_4 = 117 \bmod 78$	$= 39$
Step 4	$a_5 = 39$	$b_5 = 78 \bmod 39$	$= 0$

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

## 2.2 `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

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

## 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 `__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 initialization 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**

Fortunately or unfortunately 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 `__pos__`(self)****Returns**

- (self): returns the unchanged self

**Description**

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

**3.4 `__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 unitary 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 `__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 `--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

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

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

**3.7 `--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 conjunction with the unary negative to return the difference of the two given fraction as a new fraction object.

### 3.8 `--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.

### 3.9 `--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. She also overloads the `print()` function.

## 4 Fraction Calculator CLI

## 5 Future Improvements

### 5.1 Complexity One Horizon

### 5.2