

Programming with Python

46. Klassen/Dunder: Arithmetische Operatoren und Vergleiche

Thomas Weise (汤卫思)

tweise@hfu.edu.cn · <https://thomasweise.github.io>

School of Artificial Intelligence and Big Data
Hefei University
Hefei, Anhui, China

人工智能与大数据学院
合肥大学
中国安徽省合肥市

Programming with Python



Dies ist ein Kurs über das Programmieren mit der Programmiersprache Python an der Universität Hefei (合肥大学).

Die Webseite mit dem Lehrmaterial dieses Kurses ist <https://thomasweise.github.io/programmingWithPython> (siehe auch den QR-Code unten rechts). Dort können Sie das Kursbuch (in Englisch) und diese Slides finden. Das Repository mit den Beispielprogrammen in Python finden Sie unter <https://github.com/thomasWeise/programmingWithPythonCode>.



Outline



1. Einleitung
2. Klasse für Brüche
3. Zusammenfassung





Einleitung



Einleitung

- Viel von dem Verhalten der Syntax von Python ist in Dunder-Methoden implementiert.



Einleitung

- Viel von dem Verhalten der Syntax von Python ist in Dunder-Methoden implementiert.
- Selbst arithmetische Operatoren wie `+`, `-`, `*` und `/`!



Einleitung



- Viel von dem Verhalten der Syntax von Python ist in Dunder-Methoden implementiert.
- Selbst arithmetische Operatoren wie `+`, `-`, `*` und `/`!
- Dadurch können wir auch **selber** numerische Typen definieren, wenn wir das wollen.

Einleitung



- Viel von dem Verhalten der Syntax von Python ist in Dunder-Methoden implementiert.
- Selbst arithmetische Operatoren wie `+`, `-`, `*` und `/`!
- Dadurch können wir auch **selber** numerische Typen definieren, wenn wir das wollen.
- Und da wir schon viel Spaß mit Mathematik in diesem Kurs hatten...

Einleitung



- Viel von dem Verhalten der Syntax von Python ist in Dunder-Methoden implementiert.
- Selbst arithmetische Operatoren wie `+`, `-`, `*` und `/`!
- Dadurch können wir auch **selber** numerische Typen definieren, wenn wir das wollen.
- Und da wir schon viel Spaß mit Mathematik in diesem Kurs hatten...
- **Natürlich wollen wir das!**

Einleitung



- Viel von dem Verhalten der Syntax von Python ist in Dunder-Methoden implementiert.
- Selbst arithmetische Operatoren wie `+`, `-`, `*` und `/`!
- Dadurch können wir auch **selber** numerische Typen definieren, wenn wir das wollen.
- Und da wir schon viel Spaß mit Mathematik in diesem Kurs hatten...
- **Natürlich wollen wir das!**
- Wir werden die grundlegenden arithmetischen Operationen für eine Klasse `Fraction` implementieren, die die rationalen Zahlen $q \in \mathbb{Q}$ repräsentiert, also die Zahlen für die gilt $q = \frac{a}{b}$ mit $a, b \in \mathbb{Z}$ and $b \neq 0$.

Einleitung



- Viel von dem Verhalten der Syntax von Python ist in Dunder-Methoden implementiert.
- Selbst arithmetische Operatoren wie `+`, `-`, `*` und `/`!
- Dadurch können wir auch **selber** numerische Typen definieren, wenn wir das wollen.
- Und da wir schon viel Spaß mit Mathematik in diesem Kurs hatten...
- **Natürlich wollen wir das!**
- Wir werden die grundlegenden arithmetischen Operationen für eine Klasse `Fraction` implementieren, die die rationalen Zahlen $q \in \mathbb{Q}$ repräsentiert, also die Zahlen für die gilt $q = \frac{a}{b}$ mit $a, b \in \mathbb{Z}$ and $b \neq 0$.
- Ja, Python hat schon so eine Klasse⁴ aber das ist mir egal.

Einleitung



- Viel von dem Verhalten der Syntax von Python ist in Dunder-Methoden implementiert.
- Selbst arithmetische Operatoren wie `+`, `-`, `*` und `/`!
- Dadurch können wir auch **selber** numerische Typen definieren, wenn wir das wollen.
- Und da wir schon viel Spaß mit Mathematik in diesem Kurs hatten...
- **Natürlich wollen wir das!**
- Wir werden die grundlegenden arithmetischen Operationen für eine Klasse `Fraction` implementieren, die die rationalen Zahlen $q \in \mathbb{Q}$ repräsentiert, also die Zahlen für die gilt $q = \frac{a}{b}$ mit $a, b \in \mathbb{Z}$ and $b \neq 0$.
- Ja, Python hat schon so eine Klasse⁴ aber das ist mir egal.
- Mit anderen Worten: Wir wollen unsere Grundschulmathematik in einen neuen numerischen Typ gießen.

Einleitung



- Viel von dem Verhalten der Syntax von Python ist in Dunder-Methoden implementiert.
- Selbst arithmetische Operatoren wie `+`, `-`, `*` und `/`!
- Dadurch können wir auch **selber** numerische Typen definieren, wenn wir das wollen.
- Und da wir schon viel Spaß mit Mathematik in diesem Kurs hatten...
- **Natürlich wollen wir das!**
- Wir werden die grundlegenden arithmetischen Operationen für eine Klasse `Fraction` implementieren, die die rationalen Zahlen $q \in \mathbb{Q}$ repräsentiert, also die Zahlen für die gilt $q = \frac{a}{b}$ mit $a, b \in \mathbb{Z}$ and $b \neq 0$.
- Ja, Python hat schon so eine Klasse⁴ aber das ist mir egal.
- Mit anderen Worten: Wir wollen unsere Grundschulmathematik in einen neuen numerischen Typ gießen.
- Frischen wir unsere Erinnerung nochmal kurz auf.

Einleitung



- Viel von dem Verhalten der Syntax von Python ist in Dunder-Methoden implementiert.
- Selbst arithmetische Operatoren wie `+`, `-`, `*` und `/`!
- Dadurch können wir auch **selber** numerische Typen definieren, wenn wir das wollen.
- Und da wir schon viel Spaß mit Mathematik in diesem Kurs hatten...
- **Natürlich wollen wir das!**
- Wir werden die grundlegenden arithmetischen Operationen für eine Klasse `Fraction` implementieren, die die rationalen Zahlen $q \in \mathbb{Q}$ repräsentiert, also die Zahlen für die gilt $q = \frac{a}{b}$ mit $a, b \in \mathbb{Z}$ and $b \neq 0$.
- Ja, Python hat schon so eine Klasse⁴ aber das ist mir egal.
- Mit anderen Worten: Wir wollen unsere Grundschulmathematik in einen neuen numerischen Typ gießen.
- Frischen wir unsere Erinnerung nochmal kurz auf: Wenn wir einen Bruch $\frac{a}{b}$ haben, dann wird
 - a als der Numerator oder Nennen bezeichnet.

Einleitung



- Viel von dem Verhalten der Syntax von Python ist in Dunder-Methoden implementiert.
- Selbst arithmetische Operatoren wie `+`, `-`, `*` und `/`!
- Dadurch können wir auch **selber** numerische Typen definieren, wenn wir das wollen.
- Und da wir schon viel Spaß mit Mathematik in diesem Kurs hatten...
- **Natürlich wollen wir das!**
- Wir werden die grundlegenden arithmetischen Operationen für eine Klasse `Fraction` implementieren, die die rationalen Zahlen $q \in \mathbb{Q}$ repräsentiert, also die Zahlen für die gilt $q = \frac{a}{b}$ mit $a, b \in \mathbb{Z}$ and $b \neq 0$.
- Ja, Python hat schon so eine Klasse⁴ aber das ist mir egal.
- Mit anderen Worten: Wir wollen unsere Grundschulmathematik in einen neuen numerischen Typ gießen.
- Frischen wir unsere Erinnerung nochmal kurz auf: Wenn wir einen Bruch $\frac{a}{b}$ haben, dann wird
 - a als der Numerator oder Nennen bezeichnet und
 - b wird Denominator oder Zähler bezeichnet.

Einleitung



- Viel von dem Verhalten der Syntax von Python ist in Dunder-Methoden implementiert.
- Selbst arithmetische Operatoren wie `+`, `-`, `*` und `/`!
- Dadurch können wir auch **selber** numerische Typen definieren, wenn wir das wollen.
- Und da wir schon viel Spaß mit Mathematik in diesem Kurs hatten...
- **Natürlich wollen wir das!**
- Wir werden die grundlegenden arithmetischen Operationen für eine Klasse `Fraction` implementieren, die die rationalen Zahlen $q \in \mathbb{Q}$ repräsentiert, also die Zahlen für die gilt $q = \frac{a}{b}$ mit $a, b \in \mathbb{Z}$ and $b \neq 0$.
- Ja, Python hat schon so eine Klasse⁴ aber das ist mir egal.
- Mit anderen Worten: Wir wollen unsere Grundschulmathematik in einen neuen numerischen Typ gießen.
- Frischen wir unsere Erinnerung nochmal kurz auf: Wenn wir einen Bruch $\frac{a}{b}$ haben, dann wird
 - a als der Numerator oder Nennen bezeichnet und
 - b wird Denominator oder Zähler bezeichnet.
- Los geht's.



Klasse für Brüche



Fraction: Initialisierer und Konstanten

- Fangen wir an, unsere Klasse `Fraction` für Brüche in der Datei zu implementieren.

```
1 """A new numerical type for fractions."""
2
3 from math import gcd
4 from types import NotImplementedType
5 from typing import Final, Union
6
7
8 class Fraction:
9     """The class for fractions, i.e., rational numbers."""
10
11     def __init__(self, a: int, b: int = 1) -> None:
12         """
13         Create a normalized fraction.
14
15         :param a: the numerator
16         :param b: the denominator
17
18         >>> f"{Fraction(12, 1).a}, {Fraction(12, 1).b}"
19         '12, 1'
20         >>> f"{Fraction(12, 2).a}, {Fraction(12, 2).b}"
21         '6, 1'
22         >>> f"{Fraction(2, 12).a}, {Fraction(2, 12).b}"
23         '1, 6'
24         >>> f"{Fraction(2, -12).a}, {Fraction(2, -12).b}"
25         '-1, 6'
26         >>> f"{Fraction(-2, -12).a}, {Fraction(-2, -12).b}"
27         '1, 6'
28         >>> f"{Fraction(-2, 12).a}, {Fraction(-2, 12).b}"
29         '-1, 6'
30         >>> f"{Fraction(0, -9).a}, {Fraction(0, -9).b}"
31         '0, 1'
32         >>> try:
33             ... Fraction(1, 0)
34             ... except ZeroDivisionError as z:
35                 ... print(z)
36             1/0
37             """
38         if b == 0: # A denominator of zero is not permitted.
39             raise ZeroDivisionError(f"{a}/{b}")
40         g: int = gcd(a, b) # We use the GCD to normalize the fraction.
41         sign: int = -1 if ((a < 0) != (b < 0)) else 1 # The sign.
42         #: the numerator of the fraction will also include the sign
43         self.a: Final[int] = sign * abs(a // g)
44         #: the denominator of the fraction will always be positive
45         self.b: Final[int] = abs(b // g)
46
47
48 #: the constant zero
49 ZERO: Final[Fraction] = Fraction(0, 1)
50 #: the constant one
51 ONE: Final[Fraction] = Fraction(1, 1)
52 #: the constant 0.5
53 ONE_HALF: Final[Fraction] = Fraction(1, 2)
```

Fraction: Initialisierer und Konstanten

- Fangen wir an, unsere Klasse `Fraction` für Brüche in der Datei zu implementieren.
- Diesmal arbeiten wir uns Stückchen-weise durch, denn die Klasse ist etwas länger.

```
1 """A new numerical type for fractions."""
2
3 from math import gcd
4 from types import NotImplementedType
5 from typing import Final, Union
6
7
8 class Fraction:
9     """The class for fractions, i.e., rational numbers."""
10
11     def __init__(self, a: int, b: int = 1) -> None:
12         """
13         Create a normalized fraction.
14
15         :param a: the numerator
16         :param b: the denominator
17
18         >>> f"{Fraction(12, 1).a}, {Fraction(12, 1).b}"
19         '12, 1'
20         >>> f"{Fraction(12, 2).a}, {Fraction(12, 2).b}"
21         '6, 1'
22         >>> f"{Fraction(2, 12).a}, {Fraction(2, 12).b}"
23         '1, 6'
24         >>> f"{Fraction(2, -12).a}, {Fraction(2, -12).b}"
25         '-1, 6'
26         >>> f"{Fraction(-2, -12).a}, {Fraction(-2, -12).b}"
27         '1, 6'
28         >>> f"{Fraction(-2, 12).a}, {Fraction(-2, 12).b}"
29         '-1, 6'
30         >>> f"{Fraction(0, -9).a}, {Fraction(0, -9).b}"
31         '0, 1'
32         >>> try:
33             ... Fraction(1, 0)
34             ... except ZeroDivisionError as z:
35                 ... print(z)
36             1/0
37             """
38         if b == 0: # A denominator of zero is not permitted.
39             raise ZeroDivisionError(f"{a}/{b}")
40         g: int = gcd(a, b) # We use the GCD to normalize the fraction.
41         sign: int = -1 if ((a < 0) != (b < 0)) else 1 # The sign.
42         #: the numerator of the fraction will also include the sign
43         self.a: Final[int] = sign * abs(a // g)
44         #: the denominator of the fraction will always be positive
45         self.b: Final[int] = abs(b // g)
46
47
48 #: the constant zero
49 ZERO: Final[Fraction] = Fraction(0, 1)
50 #: the constant one
51 ONE: Final[Fraction] = Fraction(1, 1)
52 #: the constant 0.5
53 ONE_HALF: Final[Fraction] = Fraction(1, 2)
```

Fraction: Initialisierer und Konstanten

- Fangen wir an, unsere Klasse `Fraction` für Brüche in der Datei zu implementieren.
- Diesmal arbeiten wir uns Stückchen-weise durch, denn die Klasse ist etwas länger.
- Zuerst müssen wir uns entscheiden, welche Attribute unsere Klasse braucht.

```
7 class Fraction:
8     """The class for fractions, i.e., rational numbers."""
9
10
11 def __init__(self, a: int, b: int = 1) -> None:
12     """
13     Create a normalized fraction.
14
15     :param a: the numerator
16     :param b: the denominator
17
18     >>> f={Fraction(12, 1).a}, {Fraction(12, 1).b}"
19     '12, 1'
20     >>> f={Fraction(12, 2).a}, {Fraction(12, 2).b}"
21     '6, 1'
22     >>> f={Fraction(2, 12).a}, {Fraction(2, 12).b}"
23     '1, 6'
24     >>> f={Fraction(2, -12).a}, {Fraction(2, -12).b}"
25     '-1, 6'
26     >>> f={Fraction(-2, -12).a}, {Fraction(-2, -12).b}"
27     '1, 6'
28     >>> f={Fraction(-2, 12).a}, {Fraction(-2, 12).b}"
29     '-1, 6'
30     >>> f={Fraction(0, -9).a}, {Fraction(0, -9).b}"
31     '0, 1'
32     >>> try:
33         ... Fraction(1, 0)
34         ... except ZeroDivisionError as z:
35             ... print(z)
36     1/0
37     """
38     if b == 0: # A denominator of zero is not permitted.
39         raise ZeroDivisionError(f"{a}/{b}")
40     g: int = gcd(a, b) # We use the GCD to normalize the fraction
41     sign: int = -1 if ((a < 0) != (b < 0)) else 1 # The sign.
42     #: the numerator of the fraction will also include the sign
43     self.a: Final[int] = sign * abs(a // g)
44     #: the denominator of the fraction will always be positive
45     self.b: Final[int] = abs(b // g)
```

Fraction: Initialisierer und Konstanten

- Fangen wir an, unsere Klasse `Fraction` für Brüche in der Datei zu implementieren.
- Diesmal arbeiten wir uns Stückchen-weise durch, denn die Klasse ist etwas länger.
- Zuerst müssen wir uns entscheiden, welche Attribute unsere Klasse braucht.
- Wir bauen uns also zuerst den Initialisierer `__init__` zusammen.

```
class Fraction:
    """The class for fractions, i.e., rational numbers."""

    def __init__(self, a: int, b: int = 1) -> None:
        """
        Create a normalized fraction.

        :param a: the numerator
        :param b: the denominator

        >>> f={Fraction(12, 1).a}, {Fraction(12, 1).b}
        '12, 1'
        >>> f={Fraction(12, 2).a}, {Fraction(12, 2).b}
        '6, 1'
        >>> f={Fraction(2, 12).a}, {Fraction(2, 12).b}
        '1, 6'
        >>> f={Fraction(2, -12).a}, {Fraction(2, -12).b}
        '-1, 6'
        >>> f={Fraction(-2, -12).a}, {Fraction(-2, -12).b}
        '1, 6'
        >>> f={Fraction(-2, 12).a}, {Fraction(-2, 12).b}
        '-1, 6'
        >>> f={Fraction(0, -9).a}, {Fraction(0, -9).b}
        '0, 1'
        >>> try:
            ...     Fraction(1, 0)
            ... except ZeroDivisionError as z:
            ...     print(z)
        1/0
        """

        if b == 0: # A denominator of zero is not permitted.
            raise ZeroDivisionError(f"{a}/{b}")

        g: int = gcd(a, b) # We use the GCD to normalize the fraction
        sign: int = -1 if ((a < 0) != (b < 0)) else 1 # The sign.
        #: the numerator of the fraction will also include the sign
        self.a: Final[int] = sign * abs(a // g)
        #: the denominator of the fraction will always be positive
        self.b: Final[int] = abs(b // g)
```

Fraction: Initialisierer und Konstanten

- Fangen wir an, unsere Klasse `Fraction` für Brüche in der Datei zu implementieren.
- Diesmal arbeiten wir uns Stückchen-weise durch, denn die Klasse ist etwas länger.
- Zuerst müssen wir uns entscheiden, welche Attribute unsere Klasse braucht.
- Wir bauen uns also zuerst den Initialisierer `__init__` zusammen.
- Weil der Bruch $\frac{a}{b}$ durch zwei Ganzzahlen a und b definiert werden kann, entscheiden wir uns für zwei `int`-Attribute `a` und `b`.

```
class Fraction:
    """The class for fractions, i.e., rational numbers."""

    def __init__(self, a: int, b: int = 1) -> None:
        """
        Create a normalized fraction.

        :param a: the numerator
        :param b: the denominator

        >>> f={Fraction(12, 1).a}, {Fraction(12, 1).b}
        '12, 1'
        >>> f={Fraction(12, 2).a}, {Fraction(12, 2).b}
        '6, 1'
        >>> f={Fraction(2, 12).a}, {Fraction(2, 12).b}
        '1, 6'
        >>> f={Fraction(2, -12).a}, {Fraction(2, -12).b}
        '-1, 6'
        >>> f={Fraction(-2, -12).a}, {Fraction(-2, -12).b}
        '1, 6'
        >>> f={Fraction(-2, 12).a}, {Fraction(-2, 12).b}
        '-1, 6'
        >>> f={Fraction(0, -9).a}, {Fraction(0, -9).b}
        '0, 1'
        >>> try:
            ...     Fraction(1, 0)
            ... except ZeroDivisionError as z:
            ...     print(z)
        1/0
        """
        if b == 0: # A denominator of zero is not permitted.
            raise ZeroDivisionError(f"{a}/{b}")
        g: int = gcd(a, b) # We use the GCD to normalize the fraction
        sign: int = -1 if ((a < 0) != (b < 0)) else 1 # The sign.
        #: the numerator of the fraction will also include the sign
        self.a: Final[int] = sign * abs(a // g)
        #: the denominator of the fraction will always be positive
        self.b: Final[int] = abs(b // g)
```

Fraction: Initialisierer und Konstanten

- Diesmal arbeiten wir uns Stückchen-weise durch, denn die Klasse ist etwas länger.
- Zuerst müssen wir uns entscheiden, welche Attribute unsere Klasse braucht.
- Wir bauen uns also zuerst den Initialisierer `__init__` zusammen.
- Weil der Bruch $\frac{a}{b}$ durch zwei Ganzzahlen a und b definiert werden kann, entscheiden wir uns für zwei `int`-Attribute `a` und `b`.
- Wir wollen, das unserer Brüche unveränderlich.

```
class Fraction:
    """The class for fractions, i.e., rational numbers."""

    def __init__(self, a: int, b: int = 1) -> None:
        """
        Create a normalized fraction.

        :param a: the numerator
        :param b: the denominator

        >>> f={Fraction(12, 1).a}, {Fraction(12, 1).b}
        '12, 1'
        >>> f={Fraction(12, 2).a}, {Fraction(12, 2).b}
        '6, 1'
        >>> f={Fraction(2, 12).a}, {Fraction(2, 12).b}
        '1, 6'
        >>> f={Fraction(2, -12).a}, {Fraction(2, -12).b}
        '-1, 6'
        >>> f={Fraction(-2, -12).a}, {Fraction(-2, -12).b}
        '1, 6'
        >>> f={Fraction(-2, 12).a}, {Fraction(-2, 12).b}
        '-1, 6'
        >>> f={Fraction(0, -9).a}, {Fraction(0, -9).b}
        '0, 1'
        >>> try:
            ...     Fraction(1, 0)
            ... except ZeroDivisionError as z:
            ...     print(z)
        1/0
        """

        if b == 0: # A denominator of zero is not permitted.
            raise ZeroDivisionError(f"{a}/{b}")

        g: int = gcd(a, b) # We use the GCD to normalize the fraction
        sign: int = -1 if ((a < 0) != (b < 0)) else 1 # The sign.
        #: the numerator of the fraction will also include the sign
        self.a: Final[int] = sign * abs(a // g)
        #: the denominator of the fraction will always be positive
        self.b: Final[int] = abs(b // g)
```

Fraction: Initialisierer und Konstanten

- Zuerst müssen wir uns entscheiden, welche Attribute unsere Klasse braucht.
- Wir bauen uns also zuerst den Initialisierer `__init__` zusammen.
- Weil der Bruch $\frac{a}{b}$ durch zwei Ganzzahlen a und b definiert werden kann, entscheiden wir uns für zwei `int`-Attribute `a` und `b`.
- Wir wollen, das unserer Brüche unveränderlich.
- Man kann den Wert von `5` nicht verändern und sollte daher auch den Wert von `1/3` nicht verändern können.

```
class Fraction:
    """The class for fractions, i.e., rational numbers."""

    def __init__(self, a: int, b: int = 1) -> None:
        """
        Create a normalized fraction.

        :param a: the numerator
        :param b: the denominator

        >>> f={Fraction(12, 1).a}, {Fraction(12, 1).b}
        '12, 1'
        >>> f={Fraction(12, 2).a}, {Fraction(12, 2).b}
        '6, 1'
        >>> f={Fraction(2, 12).a}, {Fraction(2, 12).b}
        '1, 6'
        >>> f={Fraction(2, -12).a}, {Fraction(2, -12).b}
        '-1, 6'
        >>> f={Fraction(-2, -12).a}, {Fraction(-2, -12).b}
        '1, 6'
        >>> f={Fraction(-2, 12).a}, {Fraction(-2, 12).b}
        '-1, 6'
        >>> f={Fraction(0, -9).a}, {Fraction(0, -9).b}
        '0, 1'
        >>> try:
            ...     Fraction(1, 0)
            ... except ZeroDivisionError as z:
            ...     print(z)
        1/0
        """
        if b == 0: # A denominator of zero is not permitted.
            raise ZeroDivisionError(f"{a}/{b}")
        g: int = gcd(a, b) # We use the GCD to normalize the fraction
        sign: int = -1 if ((a < 0) != (b < 0)) else 1 # The sign.
        #: the numerator of the fraction will also include the sign
        self.a: Final[int] = sign * abs(a // g)
        #: the denominator of the fraction will always be positive
        self.b: Final[int] = abs(b // g)
```

Fraction: Initialisierer und Konstanten

- Wir bauen uns also zuerst den Initialisierer `__init__` zusammen.
- Weil der Bruch $\frac{a}{b}$ durch zwei Ganzzahlen a und b definiert werden kann, entscheiden wir uns für zwei `int`-Attribute `a` und `b`.
- Wir wollen, dass unserer Brüche unveränderlich.
- Man kann den Wert von `5` nicht verändern und sollte daher auch den Wert von `1/3` nicht verändern können.
- Die Attribute werden daher mit dem Type Hint `Final[int]`²¹ annotiert.

```
class Fraction:
    """The class for fractions, i.e., rational numbers."""

    def __init__(self, a: int, b: int = 1) -> None:
        """
        Create a normalized fraction.

        :param a: the numerator
        :param b: the denominator

        >>> f={Fraction(12, 1).a}, {Fraction(12, 1).b}
        '12, 1'
        >>> f={Fraction(12, 2).a}, {Fraction(12, 2).b}
        '6, 1'
        >>> f={Fraction(2, 12).a}, {Fraction(2, 12).b}
        '1, 6'
        >>> f={Fraction(2, -12).a}, {Fraction(2, -12).b}
        '-1, 6'
        >>> f={Fraction(-2, -12).a}, {Fraction(-2, -12).b}
        '1, 6'
        >>> f={Fraction(-2, 12).a}, {Fraction(-2, 12).b}
        '-1, 6'
        >>> f={Fraction(0, -9).a}, {Fraction(0, -9).b}
        '0, 1'
        >>> try:
            ...     Fraction(1, 0)
            ... except ZeroDivisionError as z:
            ...     print(z)
        1/0
        """

        if b == 0: # A denominator of zero is not permitted.
            raise ZeroDivisionError(f"{a}/{b}")

        g: int = gcd(a, b) # We use the GCD to normalize the fraction
        sign: int = -1 if ((a < 0) != (b < 0)) else 1 # The sign.
        #: the numerator of the fraction will also include the sign
        self.a: Final[int] = sign * abs(a // g)
        #: the denominator of the fraction will always be positive
        self.b: Final[int] = abs(b // g)
```

Fraction: Initialisierer und Konstanten

- Weil der Bruch $\frac{a}{b}$ durch zwei Ganzzahlen a und b definiert werden kann, entscheiden wir uns für zwei `int`-Attribute `a` und `b`.
- Wir wollen, das unserer Brüche unveränderlich.
- Man kann den Wert von `5` nicht verändern und sollte daher auch den Wert von `1/3` nicht verändern können.
- Die Attribute werden daher mit dem Type Hint `Final[int]`²¹ annotiert.
- Unsere Brüche sollten in einer kanonischen Normalform sein.

```
class Fraction:
```

```
    """The class for fractions, i.e., rational numbers."""
```

```
    def __init__(self, a: int, b: int = 1) -> None:
```

```
        """
```

```
        Create a normalized fraction.
```

```
        :param a: the numerator
```

```
        :param b: the denominator
```

```
    >>> f={Fraction(12, 1).a}, {Fraction(12, 1).b}"
```

```
    '12, 1'
```

```
    >>> f={Fraction(12, 2).a}, {Fraction(12, 2).b}"
```

```
    '6, 1'
```

```
    >>> f={Fraction(2, 12).a}, {Fraction(2, 12).b}"
```

```
    '1, 6'
```

```
    >>> f={Fraction(2, -12).a}, {Fraction(2, -12).b}"
```

```
    '-1, 6'
```

```
    >>> f={Fraction(-2, -12).a}, {Fraction(-2, -12).b}"
```

```
    '1, 6'
```

```
    >>> f={Fraction(-2, 12).a}, {Fraction(-2, 12).b}"
```

```
    '-1, 6'
```

```
    >>> f={Fraction(0, -9).a}, {Fraction(0, -9).b}"
```

```
    '0, 1'
```

```
    >>> try:
```

```
        ...     Fraction(1, 0)
```

```
        ... except ZeroDivisionError as z:
```

```
        ...     print(z)
```

```
    1/0
```

```
    """
```

```
    if b == 0: # A denominator of zero is not permitted.
```

```
        raise ZeroDivisionError(f"{a}/{b}")
```

```
    g: int = gcd(a, b) # We use the GCD to normalize the fraction
```

```
    sign: int = -1 if ((a < 0) != (b < 0)) else 1 # The sign.
```

```
    #: the numerator of the fraction will also include the sign
```

```
    self.a: Final[int] = sign * abs(a // g)
```

```
    #: the denominator of the fraction will always be positive
```

```
    self.b: Final[int] = abs(b // g)
```

Fraction: Initialisierer und Konstanten

- Wir wollen, dass unsere Brüche unveränderlich.
- Man kann den Wert von 5 nicht verändern und sollte daher auch den Wert von $1/3$ nicht verändern können.
- Die Attribute werden daher mit dem Type Hint `Final[int]`²¹ annotiert.
- Unsere Brüche sollten in einer kanonischen Normalform sein.
- Es ist möglich, dass zwei Brüche $\frac{a}{b} = \frac{c}{d}$ mit $a \neq c$ und $b \neq d$.

```
class Fraction:
    """The class for fractions, i.e., rational numbers."""

    def __init__(self, a: int, b: int = 1) -> None:
        """
        Create a normalized fraction.

        :param a: the numerator
        :param b: the denominator

        >>> f={Fraction(12, 1).a}, {Fraction(12, 1).b}
        '12, 1'
        >>> f={Fraction(12, 2).a}, {Fraction(12, 2).b}
        '6, 1'
        >>> f={Fraction(2, 12).a}, {Fraction(2, 12).b}
        '1, 6'
        >>> f={Fraction(2, -12).a}, {Fraction(2, -12).b}
        '-1, 6'
        >>> f={Fraction(-2, -12).a}, {Fraction(-2, -12).b}
        '1, 6'
        >>> f={Fraction(-2, 12).a}, {Fraction(-2, 12).b}
        '-1, 6'
        >>> f={Fraction(0, -9).a}, {Fraction(0, -9).b}
        '0, 1'
        >>> try:
            ...     Fraction(1, 0)
            ... except ZeroDivisionError as z:
            ...     print(z)
        1/0
        """
        if b == 0: # A denominator of zero is not permitted.
            raise ZeroDivisionError(f"{a}/{b}")
        g: int = gcd(a, b) # We use the GCD to normalize the fraction
        sign: int = -1 if ((a < 0) != (b < 0)) else 1 # The sign.
        #: the numerator of the fraction will also include the sign
        self.a: Final[int] = sign * abs(a // g)
        #: the denominator of the fraction will always be positive
        self.b: Final[int] = abs(b // g)
```

Fraction: Initialisierer und Konstanten

- Wir wollen, dass unsere Brüche unveränderlich.
- Man kann den Wert von 5 nicht verändern und sollte daher auch den Wert von $\frac{1}{3}$ nicht verändern können.
- Die Attribute werden daher mit dem Type Hint `Final[int]`²¹ annotiert.
- Unsere Brüche sollten in einer kanonischen Normalform sein.
- Es ist möglich, dass zwei Brüche $\frac{a}{b} = \frac{c}{d}$ mit $a \neq c$ und $b \neq d$.
- Das ist der Fall für z. B. $\frac{-9}{3}$ und $\frac{12}{-4}$.

```
class Fraction:
    """The class for fractions, i.e., rational numbers."""

    def __init__(self, a: int, b: int = 1) -> None:
        """
        Create a normalized fraction.

        :param a: the numerator
        :param b: the denominator

        >>> f={Fraction(12, 1).a}, {Fraction(12, 1).b}
        '12, 1'
        >>> f={Fraction(12, 2).a}, {Fraction(12, 2).b}
        '6, 1'
        >>> f={Fraction(2, 12).a}, {Fraction(2, 12).b}
        '1, 6'
        >>> f={Fraction(2, -12).a}, {Fraction(2, -12).b}
        '-1, 6'
        >>> f={Fraction(-2, -12).a}, {Fraction(-2, -12).b}
        '1, 6'
        >>> f={Fraction(-2, 12).a}, {Fraction(-2, 12).b}
        '-1, 6'
        >>> f={Fraction(0, -9).a}, {Fraction(0, -9).b}
        '0, 1'
        >>> try:
            ...     Fraction(1, 0)
            ... except ZeroDivisionError as z:
            ...     print(z)
        1/0
        """
        if b == 0: # A denominator of zero is not permitted.
            raise ZeroDivisionError(f"{a}/{b}")
        g: int = gcd(a, b) # We use the GCD to normalize the fraction
        sign: int = -1 if ((a < 0) != (b < 0)) else 1 # The sign.
        #: the numerator of the fraction will also include the sign
        self.a: Final[int] = sign * abs(a // g)
        #: the denominator of the fraction will always be positive
        self.b: Final[int] = abs(b // g)
```

Fraction: Initialisierer und Konstanten

- Man kann den Wert von `5` nicht verändern und sollte daher auch den Wert von `1/3` nicht verändern können.
- Die Attribute werden daher mit dem Type Hint `Final[int]`²¹ annotiert.
- Unsere Brüche sollten in einer kanonischen Normalform sein.
- Es ist möglich, dass zwei Brüche $\frac{a}{b} = \frac{c}{d}$ mit $a \neq c$ und $b \neq d$.
- Das ist der Fall für z. B. $\frac{-9}{3}$ und $\frac{12}{-4}$.
- In solchen Fällen sollte man sie idealerweise in Objekten speichern, die die selben Attributwerte haben.

```
class Fraction:
```

```
    """The class for fractions, i.e., rational numbers."""
```

```
    def __init__(self, a: int, b: int = 1) -> None:
```

```
        """
```

```
        Create a normalized fraction.
```

```
        :param a: the numerator
```

```
        :param b: the denominator
```

```
    >>> f={Fraction(12, 1).a}, {Fraction(12, 1).b}"
```

```
    '12, 1'
```

```
    >>> f={Fraction(12, 2).a}, {Fraction(12, 2).b}"
```

```
    '6, 1'
```

```
    >>> f={Fraction(2, 12).a}, {Fraction(2, 12).b}"
```

```
    '1, 6'
```

```
    >>> f={Fraction(2, -12).a}, {Fraction(2, -12).b}"
```

```
    '-1, 6'
```

```
    >>> f={Fraction(-2, -12).a}, {Fraction(-2, -12).b}"
```

```
    '1, 6'
```

```
    >>> f={Fraction(-2, 12).a}, {Fraction(-2, 12).b}"
```

```
    '-1, 6'
```

```
    >>> f={Fraction(0, -9).a}, {Fraction(0, -9).b}"
```

```
    '0, 1'
```

```
    >>> try:
```

```
        ...     Fraction(1, 0)
```

```
        ... except ZeroDivisionError as z:
```

```
        ...     print(z)
```

```
    1/0
```

```
    """
```

```
    if b == 0: # A denominator of zero is not permitted.
```

```
        raise ZeroDivisionError(f"{a}/{b}")
```

```
    g: int = gcd(a, b) # We use the GCD to normalize the fraction
```

```
    sign: int = -1 if ((a < 0) != (b < 0)) else 1 # The sign.
```

```
    #: the numerator of the fraction will also include the sign
```

```
    self.a: Final[int] = sign * abs(a // g)
```

```
    #: the denominator of the fraction will always be positive
```

```
    self.b: Final[int] = abs(b // g)
```

Fraction: Initialisierer und Konstanten

- Die Attribute werden daher mit dem Type Hint `Final[int]`²¹ annotiert.
- Unsere Brüche sollten in einer kanonischen Normalform sein.
- Es ist möglich, dass zwei Brüche $\frac{a}{b} = \frac{c}{d}$ mit $a \neq c$ und $b \neq d$.
- Das ist der Fall für z. B. $\frac{-9}{3}$ und $\frac{12}{-4}$.
- In solchen Fällen sollte man sie idealerweise in Objekten speichern, die die selben Attributwerte haben.
- Die Brüche $\frac{1}{2}$ und $\frac{2}{4}$ sind gleich.

```
class Fraction:
    """The class for fractions, i.e., rational numbers."""

    def __init__(self, a: int, b: int = 1) -> None:
        """
        Create a normalized fraction.

        :param a: the numerator
        :param b: the denominator

        >>> f={Fraction(12, 1).a}, {Fraction(12, 1).b}
        '12, 1'
        >>> f={Fraction(12, 2).a}, {Fraction(12, 2).b}
        '6, 1'
        >>> f={Fraction(2, 12).a}, {Fraction(2, 12).b}
        '1, 6'
        >>> f={Fraction(2, -12).a}, {Fraction(2, -12).b}
        '-1, 6'
        >>> f={Fraction(-2, -12).a}, {Fraction(-2, -12).b}
        '1, 6'
        >>> f={Fraction(-2, 12).a}, {Fraction(-2, 12).b}
        '-1, 6'
        >>> f={Fraction(0, -9).a}, {Fraction(0, -9).b}
        '0, 1'
        >>> try:
            ...     Fraction(1, 0)
            ... except ZeroDivisionError as z:
            ...     print(z)
        1/0
        """

        if b == 0: # A denominator of zero is not permitted.
            raise ZeroDivisionError(f"{a}/{b}")

        g: int = gcd(a, b) # We use the GCD to normalize the fraction
        sign: int = -1 if ((a < 0) != (b < 0)) else 1 # The sign.
        #: the numerator of the fraction will also include the sign
        self.a: Final[int] = sign * abs(a // g)
        #: the denominator of the fraction will always be positive
        self.b: Final[int] = abs(b // g)
```

Fraction: Initialisierer und Konstanten

- Die Attribute werden daher mit dem Type Hint `Final[int]`²¹ annotiert.
- Unsere Brüche sollten in einer kanonischen Normalform sein.
- Es ist möglich, dass zwei Brüche $\frac{a}{b} = \frac{c}{d}$ mit $a \neq c$ und $b \neq d$.
 - Das ist der Fall für z. B. $\frac{-9}{3}$ und $\frac{12}{-4}$.
- In solchen Fällen sollte man sie idealerweise in Objekten speichern, die die selben Attributwerte haben.
- Die Brüche $\frac{1}{2}$ und $\frac{2}{4}$ sind gleich.
- Sie sollten beide als $\frac{1}{2}$ gespeichert werden.

```
class Fraction:
    """The class for fractions, i.e., rational numbers."""

    def __init__(self, a: int, b: int = 1) -> None:
        """
        Create a normalized fraction.

        :param a: the numerator
        :param b: the denominator

        >>> f={Fraction(12, 1).a}, {Fraction(12, 1).b}
        '12, 1'
        >>> f={Fraction(12, 2).a}, {Fraction(12, 2).b}
        '6, 1'
        >>> f={Fraction(2, 12).a}, {Fraction(2, 12).b}
        '1, 6'
        >>> f={Fraction(2, -12).a}, {Fraction(2, -12).b}
        '-1, 6'
        >>> f={Fraction(-2, -12).a}, {Fraction(-2, -12).b}
        '1, 6'
        >>> f={Fraction(-2, 12).a}, {Fraction(-2, 12).b}
        '-1, 6'
        >>> f={Fraction(0, -9).a}, {Fraction(0, -9).b}
        '0, 1'
        >>> try:
            ...     Fraction(1, 0)
            ... except ZeroDivisionError as z:
            ...     print(z)
        1/0
        """

        if b == 0: # A denominator of zero is not permitted.
            raise ZeroDivisionError(f"{a}/{b}")

        g: int = gcd(a, b) # We use the GCD to normalize the fraction
        sign: int = -1 if ((a < 0) != (b < 0)) else 1 # The sign.
        #: the numerator of the fraction will also include the sign
        self.a: Final[int] = sign * abs(a // g)
        #: the denominator of the fraction will always be positive
        self.b: Final[int] = abs(b // g)
```

Fraction: Initialisierer und Konstanten

- Unsere Brüche sollten in einer kanonischen Normalform sein.
- Es ist möglich, dass zwei Brüche $\frac{a}{b} = \frac{c}{d}$ mit $a \neq c$ und $b \neq d$.
- Das ist der Fall für z. B. $\frac{-9}{3}$ und $\frac{12}{-4}$.
- In solchen Fällen sollte man sie idealerweise in Objekten speichern, die die selben Attributwerte haben.
- Die Brüche $\frac{1}{2}$ und $\frac{2}{4}$ sind gleich.
- Sie sollten beide als $\frac{1}{2}$ gespeichert werden.
- Es ist klar, dass $\frac{a}{b} = \frac{c \cdot a}{c \cdot b}$ gilt für alle Ganzzahlen $a, b, c \in \mathbb{Z}$ und $b, c > 0$.

```
class Fraction:
    """The class for fractions, i.e., rational numbers."""

    def __init__(self, a: int, b: int = 1) -> None:
        """
        Create a normalized fraction.

        :param a: the numerator
        :param b: the denominator

        >>> f={Fraction(12, 1).a}, {Fraction(12, 1).b}
        '12, 1'
        >>> f={Fraction(12, 2).a}, {Fraction(12, 2).b}
        '6, 1'
        >>> f={Fraction(2, 12).a}, {Fraction(2, 12).b}
        '1, 6'
        >>> f={Fraction(2, -12).a}, {Fraction(2, -12).b}
        '-1, 6'
        >>> f={Fraction(-2, -12).a}, {Fraction(-2, -12).b}
        '1, 6'
        >>> f={Fraction(-2, 12).a}, {Fraction(-2, 12).b}
        '-1, 6'
        >>> f={Fraction(0, -9).a}, {Fraction(0, -9).b}
        '0, 1'
        >>> try:
            ...     Fraction(1, 0)
            ... except ZeroDivisionError as z:
            ...     print(z)
        1/0
        """

        if b == 0: # A denominator of zero is not permitted.
            raise ZeroDivisionError(f"{a}/{b}")

        g: int = gcd(a, b) # We use the GCD to normalize the fraction
        sign: int = -1 if ((a < 0) != (b < 0)) else 1 # The sign.
        #: the numerator of the fraction will also include the sign
        self.a: Final[int] = sign * abs(a // g)
        #: the denominator of the fraction will always be positive
        self.b: Final[int] = abs(b // g)
```

Fraction: Initialisierer und Konstanten

- Das ist der Fall für z. B. $\frac{-9}{3}$ und $\frac{12}{-4}$.
- In solchen Fällen sollte man sie idealerweise in Objekten speichern, die die selben Attributwerte haben.
- Die Brüche $\frac{1}{2}$ und $\frac{2}{4}$ sind gleich.
- Sie sollten beide als $\frac{1}{2}$ gespeichert werden.
- Es ist klar, dass $\frac{a}{b} = \frac{c \cdot a}{c \cdot b}$ gilt für alle Ganzzahlen $a, b, c \in \mathbb{Z}$ und $b, c > 0$.
- Bevor wir also a und b speichern, teilen wir beide Zahlen durch ihren *größten gemeinsamen Teiler* (ggT, engl. gcd).

```
class Fraction:
```

```
    """The class for fractions, i.e., rational numbers."""
```

```
    def __init__(self, a: int, b: int = 1) -> None:
```

```
        """
```

```
        Create a normalized fraction.
```

```
        :param a: the numerator
```

```
        :param b: the denominator
```

```
    >>> f={Fraction(12, 1).a}, {Fraction(12, 1).b}"
```

```
    '12, 1'
```

```
    >>> f={Fraction(12, 2).a}, {Fraction(12, 2).b}"
```

```
    '6, 1'
```

```
    >>> f={Fraction(2, 12).a}, {Fraction(2, 12).b}"
```

```
    '1, 6'
```

```
    >>> f={Fraction(2, -12).a}, {Fraction(2, -12).b}"
```

```
    '-1, 6'
```

```
    >>> f={Fraction(-2, -12).a}, {Fraction(-2, -12).b}"
```

```
    '1, 6'
```

```
    >>> f={Fraction(-2, 12).a}, {Fraction(-2, 12).b}"
```

```
    '-1, 6'
```

```
    >>> f={Fraction(0, -9).a}, {Fraction(0, -9).b}"
```

```
    '0, 1'
```

```
    >>> try:
```

```
        ...     Fraction(1, 0)
```

```
        ... except ZeroDivisionError as z:
```

```
        ...     print(z)
```

```
    1/0
```

```
    """
```

```
    if b == 0: # A denominator of zero is not permitted.
```

```
        raise ZeroDivisionError(f"{a}/{b}")
```

```
    g: int = gcd(a, b) # We use the GCD to normalize the fraction
```

```
    sign: int = -1 if ((a < 0) != (b < 0)) else 1 # The sign.
```

```
    #: the numerator of the fraction will also include the sign
```

```
    self.a: Final[int] = sign * abs(a // g)
```

```
    #: the denominator of the fraction will always be positive
```

```
    self.b: Final[int] = abs(b // g)
```

Fraction: Initialisierer und Konstanten

- In solchen Fällen sollte man sie idealerweise in Objekten speichern, die die selben Attributwerde haben.
- Die Brüche $\frac{1}{2}$ und $\frac{2}{4}$ sind gleich.
- Sie sollten beide als $\frac{1}{2}$ gespeichert werden.
- Es ist klar, dass $\frac{a}{b} = \frac{c*a}{c*b}$ gilt für alle Ganzzahlen $a, b, c \in \mathbb{Z}$ und $b, c > 0$.
- Bevor wir also a und b speichern, teilen wir beide Zahlen durch ihren *größten gemeinsamen Teiler* (ggT, engl. gcd).
- In Einheit 26 haben wir den Euklidischen Algorithmus zum Berechnen des ggT implementiert.

```
class Fraction:
```

```
    """The class for fractions, i.e., rational numbers."""
```

```
    def __init__(self, a: int, b: int = 1) -> None:
```

```
        """
```

```
        Create a normalized fraction.
```

```
        :param a: the numerator
```

```
        :param b: the denominator
```

```
    >>> f={Fraction(12, 1).a}, {Fraction(12, 1).b}"
```

```
    '12, 1'
```

```
    >>> f={Fraction(12, 2).a}, {Fraction(12, 2).b}"
```

```
    '6, 1'
```

```
    >>> f={Fraction(2, 12).a}, {Fraction(2, 12).b}"
```

```
    '1, 6'
```

```
    >>> f={Fraction(2, -12).a}, {Fraction(2, -12).b}"
```

```
    '-1, 6'
```

```
    >>> f={Fraction(-2, -12).a}, {Fraction(-2, -12).b}"
```

```
    '1, 6'
```

```
    >>> f={Fraction(-2, 12).a}, {Fraction(-2, 12).b}"
```

```
    '-1, 6'
```

```
    >>> f={Fraction(0, -9).a}, {Fraction(0, -9).b}"
```

```
    '0, 1'
```

```
    >>> try:
```

```
        ...     Fraction(1, 0)
```

```
        ... except ZeroDivisionError as z:
```

```
        ...     print(z)
```

```
    1/0
```

```
    """
```

```
    if b == 0: # A denominator of zero is not permitted.
```

```
        raise ZeroDivisionError(f"{a}/{b}")
```

```
    g: int = gcd(a, b) # We use the GCD to normalize the fraction
```

```
    sign: int = -1 if ((a < 0) != (b < 0)) else 1 # The sign.
```

```
    #: the numerator of the fraction will also include the sign
```

```
    self.a: Final[int] = sign * abs(a // g)
```

```
    #: the denominator of the fraction will always be positive
```

```
    self.b: Final[int] = abs(b // g)
```

Fraction: Initialisierer und Konstanten

- Die Brüche $\frac{1}{2}$ und $\frac{2}{4}$ sind gleich.
- Sie sollten beide als $\frac{1}{2}$ gespeichert werden.
- Es ist klar, dass $\frac{a}{b} = \frac{c \cdot a}{c \cdot b}$ gilt für alle Ganzzahlen $a, b, c \in \mathbb{Z}$ und $b, c > 0$.
- Bevor wir also a und b speichern, teilen wir beide Zahlen durch ihren *größten gemeinsamen Teiler* (ggT, engl. gcd).
- In Einheit 26 haben wir den Euklidischen Algorithmus zum Berechnen des ggT implementiert.
- Diesmal verwenden wir die `gcd`-Funktion aus dem Modul `math` direkt.

```
class Fraction:
```

```
    """The class for fractions, i.e., rational numbers."""
```

```
    def __init__(self, a: int, b: int = 1) -> None:
```

```
        """
```

```
        Create a normalized fraction.
```

```
        :param a: the numerator
```

```
        :param b: the denominator
```

```
    >>> f={Fraction(12, 1).a}, {Fraction(12, 1).b}"
```

```
    '12, 1'
```

```
    >>> f={Fraction(12, 2).a}, {Fraction(12, 2).b}"
```

```
    '6, 1'
```

```
    >>> f={Fraction(2, 12).a}, {Fraction(2, 12).b}"
```

```
    '1, 6'
```

```
    >>> f={Fraction(2, -12).a}, {Fraction(2, -12).b}"
```

```
    '-1, 6'
```

```
    >>> f={Fraction(-2, -12).a}, {Fraction(-2, -12).b}"
```

```
    '1, 6'
```

```
    >>> f={Fraction(-2, 12).a}, {Fraction(-2, 12).b}"
```

```
    '-1, 6'
```

```
    >>> f={Fraction(0, -9).a}, {Fraction(0, -9).b}"
```

```
    '0, 1'
```

```
    >>> try:
```

```
        ...     Fraction(1, 0)
```

```
        ... except ZeroDivisionError as z:
```

```
        ...     print(z)
```

```
    1/0
```

```
    """
```

```
    if b == 0: # A denominator of zero is not permitted.
```

```
        raise ZeroDivisionError(f"{a}/{b}")
```

```
    g: int = gcd(a, b) # We use the GCD to normalize the fraction
```

```
    sign: int = -1 if ((a < 0) != (b < 0)) else 1 # The sign.
```

```
    #: the numerator of the fraction will also include the sign
```

```
    self.a: Final[int] = sign * abs(a // g)
```

```
    #: the denominator of the fraction will always be positive
```

```
    self.b: Final[int] = abs(b // g)
```

Fraction: Initialisierer und Konstanten

- Bevor wir also a und b speichern, teilen wir beide Zahlen durch ihren *größten gemeinsamen Teiler* (ggT, engl. gcd).
- In Einheit 26 haben wir den Euklidischen Algorithmus zum Berechnen des ggT implementiert.
- Diesmal verwenden wir die `gcd`-Funktion aus dem Modul `math` direkt.
- So oder so, in dem wir Zähler a und Nenner b durch ihren ggT teilen, stellen wir sicher, dass die Brüche in der kompaktesten Art dargestellt werden.

```
class Fraction:
    """The class for fractions, i.e., rational numbers."""

    def __init__(self, a: int, b: int = 1) -> None:
        """
        Create a normalized fraction.

        :param a: the numerator
        :param b: the denominator

        >>> f={Fraction(12, 1).a}, {Fraction(12, 1).b}
        '12, 1'
        >>> f={Fraction(12, 2).a}, {Fraction(12, 2).b}
        '6, 1'
        >>> f={Fraction(2, 12).a}, {Fraction(2, 12).b}
        '1, 6'
        >>> f={Fraction(2, -12).a}, {Fraction(2, -12).b}
        '-1, 6'
        >>> f={Fraction(-2, -12).a}, {Fraction(-2, -12).b}
        '1, 6'
        >>> f={Fraction(-2, 12).a}, {Fraction(-2, 12).b}
        '-1, 6'
        >>> f={Fraction(0, -9).a}, {Fraction(0, -9).b}
        '0, 1'
        >>> try:
            ...     Fraction(1, 0)
        ... except ZeroDivisionError as z:
            ...     print(z)
        1/0
        """

        if b == 0: # A denominator of zero is not permitted.
            raise ZeroDivisionError(f"{a}/{b}")

        g: int = gcd(a, b) # We use the GCD to normalize the fraction
        sign: int = -1 if ((a < 0) != (b < 0)) else 1 # The sign.
        #: the numerator of the fraction will also include the sign
        self.a: Final[int] = sign * abs(a // g)
        #: the denominator of the fraction will always be positive
        self.b: Final[int] = abs(b // g)
```

Fraction: Initialisierer und Konstanten

- In Einheit 26 haben wir den Euklidischen Algorithmus zum Berechnen des ggT implementiert.
- Diesmal verwenden wir die `gcd`-Funktion aus dem Modul `math` direkt.
- So oder so, in dem wir Zähler a und Nenner b durch ihren ggT teilen, stellen wir sicher, dass die Brüche in der kompaktesten Art dargestellt werden.
- Das lässt nur noch die Frage offen, wie das Vorzeichen gespeichert werden soll.

```
class Fraction:
    """The class for fractions, i.e., rational numbers."""

    def __init__(self, a: int, b: int = 1) -> None:
        """
        Create a normalized fraction.

        :param a: the numerator
        :param b: the denominator

        >>> f={Fraction(12, 1).a}, {Fraction(12, 1).b}"
        '12, 1'
        >>> f={Fraction(12, 2).a}, {Fraction(12, 2).b}"
        '6, 1'
        >>> f={Fraction(2, 12).a}, {Fraction(2, 12).b}"
        '1, 6'
        >>> f={Fraction(2, -12).a}, {Fraction(2, -12).b}"
        '-1, 6'
        >>> f={Fraction(-2, -12).a}, {Fraction(-2, -12).b}"
        '1, 6'
        >>> f={Fraction(-2, 12).a}, {Fraction(-2, 12).b}"
        '-1, 6'
        >>> f={Fraction(0, -9).a}, {Fraction(0, -9).b}"
        '0, 1'
        >>> try:
            ...     Fraction(1, 0)
            ... except ZeroDivisionError as z:
            ...     print(z)
        1/0
        """

        if b == 0: # A denominator of zero is not permitted.
            raise ZeroDivisionError(f"{a}/{b}")

        g: int = gcd(a, b) # We use the GCD to normalize the fraction
        sign: int = -1 if ((a < 0) != (b < 0)) else 1 # The sign.
        #: the numerator of the fraction will also include the sign
        self.a: Final[int] = sign * abs(a // g)
        #: the denominator of the fraction will always be positive
        self.b: Final[int] = abs(b // g)
```

Fraction: Initialisierer und Konstanten

- In Einheit 26 haben wir den Euklidischen Algorithmus zum Berechnen des ggT implementiert.
- Diesmal verwenden wir die `gcd`-Funktion aus dem Modul `math` direkt.
- So oder so, in dem wir Zähler a und Nenner b durch ihren ggT teilen, stellen wir sicher, dass die Brüche in der kompaktesten Art dargestellt werden.
- Das lässt nur noch die Frage offen, wie das Vorzeichen gespeichert werden soll.
- Natürlich gilt $\frac{-5}{2} = \frac{5}{-2}$ und $\frac{5}{2} = \frac{-5}{-2}$.

```
class Fraction:
    """The class for fractions, i.e., rational numbers."""

    def __init__(self, a: int, b: int = 1) -> None:
        """
        Create a normalized fraction.

        :param a: the numerator
        :param b: the denominator

        >>> f={Fraction(12, 1).a}, {Fraction(12, 1).b}
        '12, 1'
        >>> f={Fraction(12, 2).a}, {Fraction(12, 2).b}
        '6, 1'
        >>> f={Fraction(2, 12).a}, {Fraction(2, 12).b}
        '1, 6'
        >>> f={Fraction(2, -12).a}, {Fraction(2, -12).b}
        '-1, 6'
        >>> f={Fraction(-2, -12).a}, {Fraction(-2, -12).b}
        '1, 6'
        >>> f={Fraction(-2, 12).a}, {Fraction(-2, 12).b}
        '-1, 6'
        >>> f={Fraction(0, -9).a}, {Fraction(0, -9).b}
        '0, 1'
        >>> try:
            ...     Fraction(1, 0)
            ... except ZeroDivisionError as z:
            ...     print(z)
        1/0
        """

        if b == 0: # A denominator of zero is not permitted.
            raise ZeroDivisionError(f"{a}/{b}")

        g: int = gcd(a, b) # We use the GCD to normalize the fraction
        sign: int = -1 if ((a < 0) != (b < 0)) else 1 # The sign.
        #: the numerator of the fraction will also include the sign
        self.a: Final[int] = sign * abs(a // g)
        #: the denominator of the fraction will always be positive
        self.b: Final[int] = abs(b // g)
```

Fraction: Initialisierer und Konstanten

- Diesmal verwenden wir die `gcd`-Funktion aus dem Modul `math` direkt.
- So oder so, in dem wir Zähler a und Nenner b durch ihren ggT teilen, stellen wir sicher, dass die Brüche in der kompaktesten Art dargestellt werden.
- Das lässt nur noch die Frage offen, wie das Vorzeichen gespeichert werden soll.
- Natürlich gilt $\frac{-5}{2} = \frac{5}{-2}$ und $\frac{5}{2} = \frac{-5}{-2}$.
- Wir entscheiden uns dafür, dass das Zeichen des Bruchs immer im Attribut `a` gespeichert werden soll.

```
class Fraction:
```

```
    """The class for fractions, i.e., rational numbers."""
```

```
    def __init__(self, a: int, b: int = 1) -> None:
```

```
        """
```

```
        Create a normalized fraction.
```

```
        :param a: the numerator
```

```
        :param b: the denominator
```

```
    >>> f={Fraction(12, 1).a}, {Fraction(12, 1).b}"
```

```
    '12, 1'
```

```
    >>> f={Fraction(12, 2).a}, {Fraction(12, 2).b}"
```

```
    '6, 1'
```

```
    >>> f={Fraction(2, 12).a}, {Fraction(2, 12).b}"
```

```
    '1, 6'
```

```
    >>> f={Fraction(2, -12).a}, {Fraction(2, -12).b}"
```

```
    '-1, 6'
```

```
    >>> f={Fraction(-2, -12).a}, {Fraction(-2, -12).b}"
```

```
    '1, 6'
```

```
    >>> f={Fraction(-2, 12).a}, {Fraction(-2, 12).b}"
```

```
    '-1, 6'
```

```
    >>> f={Fraction(0, -9).a}, {Fraction(0, -9).b}"
```

```
    '0, 1'
```

```
    >>> try:
```

```
        ...     Fraction(1, 0)
```

```
        ... except ZeroDivisionError as z:
```

```
        ...     print(z)
```

```
    1/0
```

```
    """
```

```
    if b == 0: # A denominator of zero is not permitted.
```

```
        raise ZeroDivisionError(f"{a}/{b}")
```

```
    g: int = gcd(a, b) # We use the GCD to normalize the fraction
```

```
    sign: int = -1 if ((a < 0) != (b < 0)) else 1 # The sign.
```

```
    #: the numerator of the fraction will also include the sign
```

```
    self.a: Final[int] = sign * abs(a // g)
```

```
    #: the denominator of the fraction will always be positive
```

```
    self.b: Final[int] = abs(b // g)
```

Fraction: Initialisierer und Konstanten

- So oder so, in dem wir Zähler a und Nenner b durch ihren ggT teilen, stellen wir sicher, dass die Brüche in der kompaktesten Art dargestellt werden.
- Das lässt nur noch die Frage offen, wie das Vorzeichen gespeichert werden soll.
- Natürlich gilt $\frac{-5}{2} = \frac{5}{-2}$ und $\frac{5}{2} = \frac{-5}{-2}$.
- Wir entscheiden uns dafür, dass das Zeichen des Bruchs immer im Attribut `a` gespeichert werden soll.
- In anderen Worten, wenn $\frac{a}{b} < 0$, dann wird `a` negativ sein, sonst positiv oder 0.

```
class Fraction:
```

```
    """The class for fractions, i.e., rational numbers."""
```

```
    def __init__(self, a: int, b: int = 1) -> None:
```

```
        """
```

```
        Create a normalized fraction.
```

```
        :param a: the numerator
```

```
        :param b: the denominator
```

```
    >>> f={Fraction(12, 1).a}, {Fraction(12, 1).b}"
```

```
    '12, 1'
```

```
    >>> f={Fraction(12, 2).a}, {Fraction(12, 2).b}"
```

```
    '6, 1'
```

```
    >>> f={Fraction(2, 12).a}, {Fraction(2, 12).b}"
```

```
    '1, 6'
```

```
    >>> f={Fraction(2, -12).a}, {Fraction(2, -12).b}"
```

```
    '-1, 6'
```

```
    >>> f={Fraction(-2, -12).a}, {Fraction(-2, -12).b}"
```

```
    '1, 6'
```

```
    >>> f={Fraction(-2, 12).a}, {Fraction(-2, 12).b}"
```

```
    '-1, 6'
```

```
    >>> f={Fraction(0, -9).a}, {Fraction(0, -9).b}"
```

```
    '0, 1'
```

```
    >>> try:
```

```
        ...     Fraction(1, 0)
```

```
        ... except ZeroDivisionError as z:
```

```
        ...     print(z)
```

```
    1/0
```

```
    """
```

```
    if b == 0: # A denominator of zero is not permitted.
```

```
        raise ZeroDivisionError(f"{a}/{b}")
```

```
    g: int = gcd(a, b) # We use the GCD to normalize the fraction
```

```
    sign: int = -1 if ((a < 0) != (b < 0)) else 1 # The sign.
```

```
    #: the numerator of the fraction will also include the sign
```

```
    self.a: Final[int] = sign * abs(a // g)
```

```
    #: the denominator of the fraction will always be positive
```

```
    self.b: Final[int] = abs(b // g)
```

Fraction: Initialisierer und Konstanten

- Das lässt nur noch die Frage offen, wie das Vorzeichen gespeichert werden soll.
- Natürlich gilt $\frac{-5}{2} = \frac{5}{-2}$ und $\frac{5}{2} = \frac{-5}{-2}$.
- Wir entscheiden uns dafür, dass das Zeichen des Bruchs immer im Attribut `a` gespeichert werden soll.
- In anderen Worten, wenn $\frac{a}{b} < 0$, dann wird `a` negativ sein, sonst positiv oder 0.
- Es kann nur sein, dass $\frac{a}{b} < 0$ wenn genau eins von $a < 0$ oder $b < 0$ zutrifft.

```
class Fraction:
    """The class for fractions, i.e., rational numbers."""

    def __init__(self, a: int, b: int = 1) -> None:
        """
        Create a normalized fraction.

        :param a: the numerator
        :param b: the denominator

        >>> f={Fraction(12, 1).a}, {Fraction(12, 1).b}
        '12, 1'
        >>> f={Fraction(12, 2).a}, {Fraction(12, 2).b}
        '6, 1'
        >>> f={Fraction(2, 12).a}, {Fraction(2, 12).b}
        '1, 6'
        >>> f={Fraction(2, -12).a}, {Fraction(2, -12).b}
        '-1, 6'
        >>> f={Fraction(-2, -12).a}, {Fraction(-2, -12).b}
        '1, 6'
        >>> f={Fraction(-2, 12).a}, {Fraction(-2, 12).b}
        '-1, 6'
        >>> f={Fraction(0, -9).a}, {Fraction(0, -9).b}
        '0, 1'
        >>> try:
            ...     Fraction(1, 0)
            ... except ZeroDivisionError as z:
            ...     print(z)
        1/0
        """

        if b == 0: # A denominator of zero is not permitted.
            raise ZeroDivisionError(f"{a}/{b}")

        g: int = gcd(a, b) # We use the GCD to normalize the fraction
        sign: int = -1 if ((a < 0) != (b < 0)) else 1 # The sign.
        #: the numerator of the fraction will also include the sign
        self.a: Final[int] = sign * abs(a // g)
        #: the denominator of the fraction will always be positive
        self.b: Final[int] = abs(b // g)
```

Fraction: Initialisierer und Konstanten

- Natürlich gilt $\frac{-5}{2} = \frac{5}{-2}$ und $\frac{5}{2} = \frac{-5}{-2}$.
- Wir entscheiden uns dafür, dass das Zeichen des Bruchs immer im Attribut `a` gespeichert werden soll.
- In anderen Worten, wenn $\frac{a}{b} < 0$, dann wird `a` negativ sein, sonst positiv oder 0.
- Es kann nur sein, dass $\frac{a}{b} < 0$ wenn genau eins von $a < 0$ oder $b < 0$ zutrifft.
- Deshalb können wir das Vorzeichen unseres Bruch als
`-1 if ((a < 0) != (b < 0)) else 1` bestimmen.

```
class Fraction:
```

```
    """The class for fractions, i.e., rational numbers."""
```

```
    def __init__(self, a: int, b: int = 1) -> None:
```

```
        """
```

```
        Create a normalized fraction.
```

```
        :param a: the numerator
```

```
        :param b: the denominator
```

```
    >>> f={Fraction(12, 1).a}, {Fraction(12, 1).b}"
```

```
    '12, 1'
```

```
    >>> f={Fraction(12, 2).a}, {Fraction(12, 2).b}"
```

```
    '6, 1'
```

```
    >>> f={Fraction(2, 12).a}, {Fraction(2, 12).b}"
```

```
    '1, 6'
```

```
    >>> f={Fraction(2, -12).a}, {Fraction(2, -12).b}"
```

```
    '-1, 6'
```

```
    >>> f={Fraction(-2, -12).a}, {Fraction(-2, -12).b}"
```

```
    '1, 6'
```

```
    >>> f={Fraction(-2, 12).a}, {Fraction(-2, 12).b}"
```

```
    '-1, 6'
```

```
    >>> f={Fraction(0, -9).a}, {Fraction(0, -9).b}"
```

```
    '0, 1'
```

```
    >>> try:
```

```
        ...     Fraction(1, 0)
```

```
        ... except ZeroDivisionError as z:
```

```
        ...     print(z)
```

```
    1/0
```

```
    """
```

```
    if b == 0: # A denominator of zero is not permitted.
```

```
        raise ZeroDivisionError(f"{a}/{b}")
```

```
    g: int = gcd(a, b) # We use the GCD to normalize the fraction
```

```
    sign: int = -1 if ((a < 0) != (b < 0)) else 1 # The sign.
```

```
    #: the numerator of the fraction will also include the sign
```

```
    self.a: Final[int] = sign * abs(a // g)
```

```
    #: the denominator of the fraction will always be positive
```

```
    self.b: Final[int] = abs(b // g)
```

Fraction: Initialisierer und Konstanten

- Wir entscheiden uns dafür, dass das Zeichen des Bruchs immer im Attribut `a` gespeichert werden soll.
- In anderen Worten, wenn $\frac{a}{b} < 0$, dann wird `a` negativ sein, sonst positiv oder 0.
- Es kann nur sein, dass $\frac{a}{b} < 0$ wenn genau eins von $a < 0$ oder $b < 0$ zutrifft.
`-1 if ((a < 0) != (b < 0)) else 1`
- Deshalb können wir das Vorzeichen unseres Bruch als bestimmen.
- Im Initialisierer müssen wir auch sicherstellen, dass so etwas wie $\frac{7}{0}$ nicht passiert.

```
class Fraction:
    """The class for fractions, i.e., rational numbers."""

    def __init__(self, a: int, b: int = 1) -> None:
        """
        Create a normalized fraction.

        :param a: the numerator
        :param b: the denominator

        >>> f={Fraction(12, 1).a}, {Fraction(12, 1).b}
        '12, 1'
        >>> f={Fraction(12, 2).a}, {Fraction(12, 2).b}
        '6, 1'
        >>> f={Fraction(2, 12).a}, {Fraction(2, 12).b}
        '1, 6'
        >>> f={Fraction(2, -12).a}, {Fraction(2, -12).b}
        '-1, 6'
        >>> f={Fraction(-2, -12).a}, {Fraction(-2, -12).b}
        '1, 6'
        >>> f={Fraction(-2, 12).a}, {Fraction(-2, 12).b}
        '-1, 6'
        >>> f={Fraction(0, -9).a}, {Fraction(0, -9).b}
        '0, 1'
        >>> try:
            ...     Fraction(1, 0)
            ... except ZeroDivisionError as z:
            ...     print(z)
        1/0
        """

        if b == 0: # A denominator of zero is not permitted.
            raise ZeroDivisionError(f"{a}/{b}")

        g: int = gcd(a, b) # We use the GCD to normalize the fraction
        sign: int = -1 if ((a < 0) != (b < 0)) else 1 # The sign.
        #: the numerator of the fraction will also include the sign
        self.a: Final[int] = sign * abs(a // g)
        #: the denominator of the fraction will always be positive
        self.b: Final[int] = abs(b // g)
```

Fraction: Initialisierer und Konstanten

- In anderen Worten, wenn $\frac{a}{b} < 0$, dann wird `a` negativ sein, sonst positiv oder 0.
- Es kann nur sein, dass $\frac{a}{b} < 0$ wenn genau eins von $a < 0$ oder $b < 0$ zutrifft.
- Deshalb können wir das Vorzeichen unseres Bruch als
`-1 if ((a < 0) != (b < 0)) else 1` bestimmen.
- Im Initialisierer müssen wir auch sicherstellen, dass so etwas wie $\frac{7}{0}$ nicht passiert.
- In diesem Fall lösen wir einen `ZeroDivisionError` aus.

```
class Fraction:
```

```
    """The class for fractions, i.e., rational numbers."""
```

```
    def __init__(self, a: int, b: int = 1) -> None:
```

```
        """
```

```
        Create a normalized fraction.
```

```
        :param a: the numerator
```

```
        :param b: the denominator
```

```
    >>> f={Fraction(12, 1).a}, {Fraction(12, 1).b}"
```

```
    '12, 1'
```

```
    >>> f={Fraction(12, 2).a}, {Fraction(12, 2).b}"
```

```
    '6, 1'
```

```
    >>> f={Fraction(2, 12).a}, {Fraction(2, 12).b}"
```

```
    '1, 6'
```

```
    >>> f={Fraction(2, -12).a}, {Fraction(2, -12).b}"
```

```
    '-1, 6'
```

```
    >>> f={Fraction(-2, -12).a}, {Fraction(-2, -12).b}"
```

```
    '1, 6'
```

```
    >>> f={Fraction(-2, 12).a}, {Fraction(-2, 12).b}"
```

```
    '-1, 6'
```

```
    >>> f={Fraction(0, -9).a}, {Fraction(0, -9).b}"
```

```
    '0, 1'
```

```
    >>> try:
```

```
        ...     Fraction(1, 0)
```

```
        ... except ZeroDivisionError as z:
```

```
        ...     print(z)
```

```
    1/0
```

```
    """
```

```
    if b == 0: # A denominator of zero is not permitted.
```

```
        raise ZeroDivisionError(f"{a}/{b}")
```

```
    g: int = gcd(a, b) # We use the GCD to normalize the fraction
```

```
    sign: int = -1 if ((a < 0) != (b < 0)) else 1 # The sign.
```

```
    #: the numerator of the fraction will also include the sign
```

```
    self.a: Final[int] = sign * abs(a // g)
```

```
    #: the denominator of the fraction will always be positive
```

```
    self.b: Final[int] = abs(b // g)
```

Fraction: Initialisierer und Konstanten

- Es kann nur sein, dass $\frac{a}{b} < 0$ wenn genau eins von $a < 0$ oder $b < 0$ zutrifft.
- Deshalb können wir das Vorzeichen unseres Bruch als
`-1 if ((a < 0) != (b < 0)) else 1` bestimmen.
- Im Initialisierer müssen wir auch sicherstellen, dass so etwas wie $\frac{7}{0}$ nicht passiert.
- In diesem Fall lösen wir einen `ZeroDivisionError` aus.
- Natürlich müssen wir noch richtige Doctests für den Initialisierer bauen.

```
class Fraction:
```

```
    """The class for fractions, i.e., rational numbers."""
```

```
    def __init__(self, a: int, b: int = 1) -> None:
```

```
        """
```

```
        Create a normalized fraction.
```

```
        :param a: the numerator
```

```
        :param b: the denominator
```

```
    >>> f"{Fraction(12, 1).a}, {Fraction(12, 1).b}"
```

```
    '12, 1'
```

```
    >>> f"{Fraction(12, 2).a}, {Fraction(12, 2).b}"
```

```
    '6, 1'
```

```
    >>> f"{Fraction(2, 12).a}, {Fraction(2, 12).b}"
```

```
    '1, 6'
```

```
    >>> f"{Fraction(2, -12).a}, {Fraction(2, -12).b}"
```

```
    '-1, 6'
```

```
    >>> f"{Fraction(-2, -12).a}, {Fraction(-2, -12).b}"
```

```
    '1, 6'
```

```
    >>> f"{Fraction(-2, 12).a}, {Fraction(-2, 12).b}"
```

```
    '-1, 6'
```

```
    >>> f"{Fraction(0, -9).a}, {Fraction(0, -9).b}"
```

```
    '0, 1'
```

```
    >>> try:
```

```
        ...     Fraction(1, 0)
```

```
        ... except ZeroDivisionError as z:
```

```
        ...     print(z)
```

```
    1/0
```

```
    """
```

```
    if b == 0: # A denominator of zero is not permitted.
```

```
        raise ZeroDivisionError(f"{a}/{b}")
```

```
    g: int = gcd(a, b) # We use the GCD to normalize the fraction
```

```
    sign: int = -1 if ((a < 0) != (b < 0)) else 1 # The sign.
```

```
    #: the numerator of the fraction will also include the sign
```

```
    self.a: Final[int] = sign * abs(a // g)
```

```
    #: the denominator of the fraction will always be positive
```

```
    self.b: Final[int] = abs(b // g)
```

Fraction: Initialisierer und Konstanten

- Deshalb können wir das Vorzeichen unseres Bruch als `-1 if ((a < 0) != (b < 0)) else 1` bestimmen.
- Im Initialisierer müssen wir auch sicherstellen, dass so etwas wie $\frac{7}{0}$ nicht passiert.
- In diesem Fall lösen wir einen `ZeroDivisionError` aus.
- Natürlich müssen wir noch richtige Doctests für den Initialisierer bauen.
- Wir prüfen, ob die Werte a und b ordentlich in den Attributen `a` und `b` gespeichert werden.

```
class Fraction:
    """The class for fractions, i.e., rational numbers."""

    def __init__(self, a: int, b: int = 1) -> None:
        """
        Create a normalized fraction.

        :param a: the numerator
        :param b: the denominator

        >>> f={Fraction(12, 1).a}, {Fraction(12, 1).b}
        '12, 1'
        >>> f={Fraction(12, 2).a}, {Fraction(12, 2).b}
        '6, 1'
        >>> f={Fraction(2, 12).a}, {Fraction(2, 12).b}
        '1, 6'
        >>> f={Fraction(2, -12).a}, {Fraction(2, -12).b}
        '-1, 6'
        >>> f={Fraction(-2, -12).a}, {Fraction(-2, -12).b}
        '1, 6'
        >>> f={Fraction(-2, 12).a}, {Fraction(-2, 12).b}
        '-1, 6'
        >>> f={Fraction(0, -9).a}, {Fraction(0, -9).b}
        '0, 1'
        >>> try:
            ...     Fraction(1, 0)
            ... except ZeroDivisionError as z:
            ...     print(z)
        1/0
        """
        if b == 0: # A denominator of zero is not permitted.
            raise ZeroDivisionError(f"{a}/{b}")
        g: int = gcd(a, b) # We use the GCD to normalize the fraction
        sign: int = -1 if ((a < 0) != (b < 0)) else 1 # The sign.
        #: the numerator of the fraction will also include the sign
        self.a: Final[int] = sign * abs(a // g)
        #: the denominator of the fraction will always be positive
        self.b: Final[int] = abs(b // g)
```

Fraction: Initialisierer und Konstanten

- Im Initialisierer müssen wir auch sicherstellen, dass so etwas wie $\frac{7}{0}$ nicht passiert.
- In diesem Fall lösen wir einen `ZeroDivisionError` aus.
- Natürlich müssen wir noch richtige Doctests für den Initialisierer bauen.
- Wir prüfen, ob die Werte a und b ordentlich in den Attributen `a` und `b` gespeichert werden.
- Dann müssen wir prüfen, ob das mit dem Teilen durch den ggT auch funktioniert, also ob der Initialisierer $\frac{12}{2}$ auf $\frac{6}{1}$ umrechnet.

```
class Fraction:
    """The class for fractions, i.e., rational numbers."""

    def __init__(self, a: int, b: int = 1) -> None:
        """
        Create a normalized fraction.

        :param a: the numerator
        :param b: the denominator

        >>> f"{Fraction(12, 1).a}, {Fraction(12, 1).b}"
        '12, 1'
        >>> f"{Fraction(12, 2).a}, {Fraction(12, 2).b}"
        '6, 1'
        >>> f"{Fraction(2, 12).a}, {Fraction(2, 12).b}"
        '1, 6'
        >>> f"{Fraction(2, -12).a}, {Fraction(2, -12).b}"
        '-1, 6'
        >>> f"{Fraction(-2, -12).a}, {Fraction(-2, -12).b}"
        '1, 6'
        >>> f"{Fraction(-2, 12).a}, {Fraction(-2, 12).b}"
        '-1, 6'
        >>> f"{Fraction(0, -9).a}, {Fraction(0, -9).b}"
        '0, 1'
        >>> try:
            ...     Fraction(1, 0)
            ... except ZeroDivisionError as z:
            ...     print(z)
        1/0
        """

        if b == 0: # A denominator of zero is not permitted.
            raise ZeroDivisionError(f"{a}/{b}")

        g: int = gcd(a, b) # We use the GCD to normalize the fraction
        sign: int = -1 if ((a < 0) != (b < 0)) else 1 # The sign.
        #: the numerator of the fraction will also include the sign
        self.a: Final[int] = sign * abs(a // g)
        #: the denominator of the fraction will always be positive
        self.b: Final[int] = abs(b // g)
```

Fraction: Initialisierer und Konstanten

- In diesem Fall lösen wir einen `ZeroDivisionError` aus.
- Natürlich müssen wir noch richtige Doctests für den Initialisierer bauen.
- Wir prüfen, ob die Werte a und b ordentlich in den Attributen `a` und `b` gespeichert werden.
- Dann müssen wir prüfen, ob das mit dem Teilen durch den ggT auch funktioniert, also ob der Initialisierer $\frac{12}{2}$ auf $\frac{6}{1}$ umrechnet.
- Und wir müssen prüfen, dass $\frac{-2}{-12}$ und $\frac{-2}{12}$ korrekt zu $\frac{-1}{6}$ werden, während $\frac{-2}{-12}$ zu $\frac{1}{6}$ wird.

```
class Fraction:
    """The class for fractions, i.e., rational numbers."""

    def __init__(self, a: int, b: int = 1) -> None:
        """
        Create a normalized fraction.

        :param a: the numerator
        :param b: the denominator

        >>> f={Fraction(12, 1).a}, {Fraction(12, 1).b}
        '12, 1'
        >>> f={Fraction(12, 2).a}, {Fraction(12, 2).b}
        '6, 1'
        >>> f={Fraction(2, 12).a}, {Fraction(2, 12).b}
        '1, 6'
        >>> f={Fraction(2, -12).a}, {Fraction(2, -12).b}
        '-1, 6'
        >>> f={Fraction(-2, -12).a}, {Fraction(-2, -12).b}
        '1, 6'
        >>> f={Fraction(-2, 12).a}, {Fraction(-2, 12).b}
        '-1, 6'
        >>> f={Fraction(0, -9).a}, {Fraction(0, -9).b}
        '0, 1'
        >>> try:
            ...     Fraction(1, 0)
            ... except ZeroDivisionError as z:
            ...     print(z)
        1/0
        """
        if b == 0: # A denominator of zero is not permitted.
            raise ZeroDivisionError(f"{a}/{b}")
        g: int = gcd(a, b) # We use the GCD to normalize the fraction
        sign: int = -1 if ((a < 0) != (b < 0)) else 1 # The sign.
        #: the numerator of the fraction will also include the sign
        self.a: Final[int] = sign * abs(a // g)
        #: the denominator of the fraction will always be positive
        self.b: Final[int] = abs(b // g)
```

Fraction: Initialisierer und Konstanten

- Natürlich müssen wir noch richtige Doctests für den Initialisierer bauen.
- Wir prüfen, ob die Werte a und b ordentlich in den Attributen `a` und `b` gespeichert werden.
- Dann müssen wir prüfen, ob das mit dem Teilen durch den ggT auch funktioniert, also ob der Initialisierer $\frac{12}{2}$ auf $\frac{6}{1}$ umrechnet.
- Und wir müssen prüfen, dass $\frac{2}{-12}$ und $\frac{-2}{12}$ korrekt zu $\frac{-1}{6}$ werden, während $\frac{-2}{-12}$ zu $\frac{1}{6}$ wird.
- Der Spezialfall der Zahl 0 muss auch geprüft werden.

```
class Fraction:
    """The class for fractions, i.e., rational numbers."""

    def __init__(self, a: int, b: int = 1) -> None:
        """
        Create a normalized fraction.

        :param a: the numerator
        :param b: the denominator

        >>> f={Fraction(12, 1).a}, {Fraction(12, 1).b}
        '12, 1'
        >>> f={Fraction(12, 2).a}, {Fraction(12, 2).b}
        '6, 1'
        >>> f={Fraction(2, 12).a}, {Fraction(2, 12).b}
        '1, 6'
        >>> f={Fraction(2, -12).a}, {Fraction(2, -12).b}
        '-1, 6'
        >>> f={Fraction(-2, -12).a}, {Fraction(-2, -12).b}
        '1, 6'
        >>> f={Fraction(-2, 12).a}, {Fraction(-2, 12).b}
        '-1, 6'
        >>> f={Fraction(0, -9).a}, {Fraction(0, -9).b}
        '0, 1'
        >>> try:
            ...     Fraction(1, 0)
            ... except ZeroDivisionError as z:
            ...     print(z)
        1/0
        """
        if b == 0: # A denominator of zero is not permitted.
            raise ZeroDivisionError(f"{a}/{b}")
        g: int = gcd(a, b) # We use the GCD to normalize the fraction
        sign: int = -1 if ((a < 0) != (b < 0)) else 1 # The sign.
        #: the numerator of the fraction will also include the sign
        self.a: Final[int] = sign * abs(a // g)
        #: the denominator of the fraction will always be positive
        self.b: Final[int] = abs(b // g)
```

Fraction: Initialisierer und Konstanten

- Wir prüfen, ob die Werte a und b ordentlich in den Attributen `a` und `b` gespeichert werden.
- Dann müssen wir prüfen, ob das mit dem Teilen durch den ggT auch funktioniert, also ob der Initialisierer $\frac{12}{2}$ auf $\frac{6}{1}$ umrechnet.
- Und wir müssen prüfen, dass $\frac{2}{-12}$ und $\frac{-2}{12}$ korrekt zu $\frac{-1}{6}$ werden, während $\frac{-2}{-12}$ zu $\frac{1}{6}$ wird.
- Der Spezialfall der Zahl 0 muss auch geprüft werden.
- Wir wissen, dass $\text{gcd}(0, -9) = -9$, also sollte $\frac{0}{-9}$ zu $\frac{0}{1}$ werden.

```
class Fraction:
```

```
    """The class for fractions, i.e., rational numbers."""
```

```
    def __init__(self, a: int, b: int = 1) -> None:
```

```
        """
```

```
        Create a normalized fraction.
```

```
        :param a: the numerator
```

```
        :param b: the denominator
```

```
    >>> f"{Fraction(12, 1).a}, {Fraction(12, 1).b}"
```

```
    '12, 1'
```

```
    >>> f"{Fraction(12, 2).a}, {Fraction(12, 2).b}"
```

```
    '6, 1'
```

```
    >>> f"{Fraction(2, 12).a}, {Fraction(2, 12).b}"
```

```
    '1, 6'
```

```
    >>> f"{Fraction(2, -12).a}, {Fraction(2, -12).b}"
```

```
    '-1, 6'
```

```
    >>> f"{Fraction(-2, -12).a}, {Fraction(-2, -12).b}"
```

```
    '1, 6'
```

```
    >>> f"{Fraction(-2, 12).a}, {Fraction(-2, 12).b}"
```

```
    '-1, 6'
```

```
    >>> f"{Fraction(0, -9).a}, {Fraction(0, -9).b}"
```

```
    '0, 1'
```

```
    >>> try:
```

```
        ...     Fraction(1, 0)
```

```
        ... except ZeroDivisionError as z:
```

```
        ...     print(z)
```

```
    1/0
```

```
    """
```

```
    if b == 0: # A denominator of zero is not permitted.
```

```
        raise ZeroDivisionError(f"{a}/{b}")
```

```
    g: int = gcd(a, b) # We use the GCD to normalize the fraction
```

```
    sign: int = -1 if ((a < 0) != (b < 0)) else 1 # The sign.
```

```
    #: the numerator of the fraction will also include the sign
```

```
    self.a: Final[int] = sign * abs(a // g)
```

```
    #: the denominator of the fraction will always be positive
```

```
    self.b: Final[int] = abs(b // g)
```

Fraction: Initialisierer und Konstanten

- Dann müssen wir prüfen, ob das mit dem Teilen durch den ggT auch funktioniert, also ob der Initialisierer $\frac{12}{2}$ auf $\frac{6}{1}$ umrechnet.
- Und wir müssen prüfen, dass $\frac{2}{-12}$ und $\frac{-2}{12}$ korrekt zu $\frac{-1}{6}$ werden, während $\frac{-2}{-12}$ zu $\frac{1}{6}$ wird.
- Der Spezialfall der Zahl 0 muss auch geprüft werden.
- Wir wissen, dass `gcd(0, -9) = -9`, also sollte $\frac{0}{-9}$ zu $\frac{0}{1}$ werden.
- Aber es ist besser, das auch nachzuprüfen.

```
class Fraction:
```

```
    """The class for fractions, i.e., rational numbers."""
```

```
    def __init__(self, a: int, b: int = 1) -> None:
```

```
        """
```

```
        Create a normalized fraction.
```

```
        :param a: the numerator
```

```
        :param b: the denominator
```

```
    >>> f={Fraction(12, 1).a}, {Fraction(12, 1).b}"
```

```
    '12, 1'
```

```
    >>> f={Fraction(12, 2).a}, {Fraction(12, 2).b}"
```

```
    '6, 1'
```

```
    >>> f={Fraction(2, 12).a}, {Fraction(2, 12).b}"
```

```
    '1, 6'
```

```
    >>> f={Fraction(2, -12).a}, {Fraction(2, -12).b}"
```

```
    '-1, 6'
```

```
    >>> f={Fraction(-2, -12).a}, {Fraction(-2, -12).b}"
```

```
    '1, 6'
```

```
    >>> f={Fraction(-2, 12).a}, {Fraction(-2, 12).b}"
```

```
    '-1, 6'
```

```
    >>> f={Fraction(0, -9).a}, {Fraction(0, -9).b}"
```

```
    '0, 1'
```

```
    >>> try:
```

```
        ...     Fraction(1, 0)
```

```
        ... except ZeroDivisionError as z:
```

```
        ...     print(z)
```

```
    1/0
```

```
    """
```

```
    if b == 0: # A denominator of zero is not permitted.
```

```
        raise ZeroDivisionError(f"{a}/{b}")
```

```
    g: int = gcd(a, b) # We use the GCD to normalize the fraction
```

```
    sign: int = -1 if ((a < 0) != (b < 0)) else 1 # The sign.
```

```
    #: the numerator of the fraction will also include the sign
```

```
    self.a: Final[int] = sign * abs(a // g)
```

```
    #: the denominator of the fraction will always be positive
```

```
    self.b: Final[int] = abs(b // g)
```

Fraction: Initialisierer und Konstanten

- Und wir müssen prüfen, dass $\frac{-2}{-12}$ und $\frac{-2}{12}$ korrekt zu $\frac{-1}{6}$ werden, während $\frac{-2}{-12}$ zu $\frac{1}{6}$ wird.
- Der Spezialfall der Zahl 0 muss auch geprüft werden.
- Wir wissen, dass `gcd(0, -9) = -9`, also sollte $\frac{0}{-9}$ zu $\frac{0}{1}$ werden.
- Aber es ist besser, das auch nachzuprüfen.
- Genauso müssen wir prüfen, ob auch wirklich ein `ZeroDivisionError` ausgelöst wird, wenn wir versuchen eine Zahl mit einem Nenner von 0 anzulegen.

```
class Fraction:
```

```
    """The class for fractions, i.e., rational numbers."""
```

```
    def __init__(self, a: int, b: int = 1) -> None:
```

```
        """
```

```
        Create a normalized fraction.
```

```
        :param a: the numerator
```

```
        :param b: the denominator
```

```
    >>> f"{Fraction(12, 1).a}, {Fraction(12, 1).b}"
```

```
    '12, 1'
```

```
    >>> f"{Fraction(12, 2).a}, {Fraction(12, 2).b}"
```

```
    '6, 1'
```

```
    >>> f"{Fraction(2, 12).a}, {Fraction(2, 12).b}"
```

```
    '1, 6'
```

```
    >>> f"{Fraction(2, -12).a}, {Fraction(2, -12).b}"
```

```
    '-1, 6'
```

```
    >>> f"{Fraction(-2, -12).a}, {Fraction(-2, -12).b}"
```

```
    '1, 6'
```

```
    >>> f"{Fraction(-2, 12).a}, {Fraction(-2, 12).b}"
```

```
    '-1, 6'
```

```
    >>> f"{Fraction(0, -9).a}, {Fraction(0, -9).b}"
```

```
    '0, 1'
```

```
    >>> try:
```

```
        ...     Fraction(1, 0)
```

```
    ... except ZeroDivisionError as z:
```

```
        ...     print(z)
```

```
    1/0
```

```
    """
```

```
    if b == 0: # A denominator of zero is not permitted.
```

```
        raise ZeroDivisionError(f"{a}/{b}")
```

```
    g: int = gcd(a, b) # We use the GCD to normalize the fraction
```

```
    sign: int = -1 if ((a < 0) != (b < 0)) else 1 # The sign.
```

```
    #: the numerator of the fraction will also include the sign
```

```
    self.a: Final[int] = sign * abs(a // g)
```

```
    #: the denominator of the fraction will always be positive
```

```
    self.b: Final[int] = abs(b // g)
```

Fraction: Initialisierer und Konstanten

- Der Spezialfall der Zahl 0 muss auch geprüft werden.
- Wir wissen, dass `gcd(0, -9) = -9`, also sollte $\frac{0}{-9}$ zu $\frac{0}{1}$ werden.
- Aber es ist besser, das auch nachzuprüfen.
- Genauso müssen wir prüfen, ob auch wirklich ein `ZeroDivisionError` ausgelöst wird, wenn wir versuchen eine Zahl mit einem Nenner von 0 anzulegen.
- Ohne den Code von `__init__` zu lesen, kann ein Benutzer bereits viel über unsere Klasse `Fraction` nur von den Doctests lernen.

```
class Fraction:
    """The class for fractions, i.e., rational numbers."""

    def __init__(self, a: int, b: int = 1) -> None:
        """
        Create a normalized fraction.

        :param a: the numerator
        :param b: the denominator

        >>> f={Fraction(12, 1).a}, {Fraction(12, 1).b}
        '12, 1'
        >>> f={Fraction(12, 2).a}, {Fraction(12, 2).b}
        '6, 1'
        >>> f={Fraction(2, 12).a}, {Fraction(2, 12).b}
        '1, 6'
        >>> f={Fraction(2, -12).a}, {Fraction(2, -12).b}
        '-1, 6'
        >>> f={Fraction(-2, -12).a}, {Fraction(-2, -12).b}
        '1, 6'
        >>> f={Fraction(-2, 12).a}, {Fraction(-2, 12).b}
        '-1, 6'
        >>> f={Fraction(0, -9).a}, {Fraction(0, -9).b}
        '0, 1'
        >>> try:
            ...     Fraction(1, 0)
            ... except ZeroDivisionError as z:
            ...     print(z)
        1/0
        """
        if b == 0: # A denominator of zero is not permitted.
            raise ZeroDivisionError(f"{a}/{b}")
        g: int = gcd(a, b) # We use the GCD to normalize the fraction
        sign: int = -1 if ((a < 0) != (b < 0)) else 1 # The sign.
        #: the numerator of the fraction will also include the sign
        self.a: Final[int] = sign * abs(a // g)
        #: the denominator of the fraction will always be positive
        self.b: Final[int] = abs(b // g)
```

Fraction: Initialisierer und Konstanten

- Wir wissen, dass `gcd(0, -9) = -9`, also sollte $\frac{0}{-9}$ zu $\frac{0}{1}$ werden.
- Aber es ist besser, das auch nachzuprüfen.
- Genauso müssen wir prüfen, ob auch wirklich ein `ZeroDivisionError` ausgelöst wird, wenn wir versuchen eine Zahl mit einem Nenner von 0 anzulegen.
- Ohne den Code von `__init__` zu lesen, kann ein Benutzer bereits viel über unsere Klasse `Fraction` nur von den Doctests lernen.
- Nun ist es so, dass einige spezielle Brüche in vielen Berechnungen vorkommen.

```
class Fraction:
    """The class for fractions, i.e., rational numbers."""

    def __init__(self, a: int, b: int = 1) -> None:
        """
        Create a normalized fraction.

        :param a: the numerator
        :param b: the denominator

        >>> f={Fraction(12, 1).a}, {Fraction(12, 1).b}
        '12, 1'
        >>> f={Fraction(12, 2).a}, {Fraction(12, 2).b}
        '6, 1'
        >>> f={Fraction(2, 12).a}, {Fraction(2, 12).b}
        '1, 6'
        >>> f={Fraction(2, -12).a}, {Fraction(2, -12).b}
        '-1, 6'
        >>> f={Fraction(-2, -12).a}, {Fraction(-2, -12).b}
        '1, 6'
        >>> f={Fraction(-2, 12).a}, {Fraction(-2, 12).b}
        '-1, 6'
        >>> f={Fraction(0, -9).a}, {Fraction(0, -9).b}
        '0, 1'
        >>> try:
            ...     Fraction(1, 0)
            ... except ZeroDivisionError as z:
            ...     print(z)
        1/0
        """

        if b == 0: # A denominator of zero is not permitted.
            raise ZeroDivisionError(f"{a}/{b}")

        g: int = gcd(a, b) # We use the GCD to normalize the fraction
        sign: int = -1 if ((a < 0) != (b < 0)) else 1 # The sign.
        #: the numerator of the fraction will also include the sign
        self.a: Final[int] = sign * abs(a // g)
        #: the denominator of the fraction will always be positive
        self.b: Final[int] = abs(b // g)
```

Fraction: Initialisierer und Konstanten

- Genauso müssen wir prüfen, ob auch wirklich ein `ZeroDivisionError` ausgelöst wird, wenn wir versuchen eine Zahl mit einem Nenner von 0 anzulegen.
- Ohne den Code von `__init__` zu lesen, kann ein Benutzer bereits viel über unsere Klasse `Fraction` nur von den Doctests lernen.
- Nun ist es so, dass einige spezielle Brüche in vielen Berechnungen vorkommen.
- Anstatt sie immer wieder neu anzulegen, könnten wir sie als Konstanten definieren.

```
class Fraction:
    """The class for fractions, i.e., rational numbers."""

    def __init__(self, a: int, b: int = 1) -> None:
        """
        Create a normalized fraction.

        :param a: the numerator
        :param b: the denominator

        >>> f={Fraction(12, 1).a}, {Fraction(12, 1).b}"
        '12, 1'
        >>> f={Fraction(12, 2).a}, {Fraction(12, 2).b}"
        '6, 1'
        >>> f={Fraction(2, 12).a}, {Fraction(2, 12).b}"
        '1, 6'
        >>> f={Fraction(2, -12).a}, {Fraction(2, -12).b}"
        '-1, 6'
        >>> f={Fraction(-2, -12).a}, {Fraction(-2, -12).b}"
        '1, 6'
        >>> f={Fraction(-2, 12).a}, {Fraction(-2, 12).b}"
        '-1, 6'
        >>> f={Fraction(0, -9).a}, {Fraction(0, -9).b}"
        '0, 1'
        >>> try:
            ...     Fraction(1, 0)
            ... except ZeroDivisionError as z:
            ...     print(z)
        1/0
        """

        if b == 0: # A denominator of zero is not permitted.
            raise ZeroDivisionError(f"{a}/{b}")

        g: int = gcd(a, b) # We use the GCD to normalize the fraction
        sign: int = -1 if ((a < 0) != (b < 0)) else 1 # The sign.
        #: the numerator of the fraction will also include the sign
        self.a: Final[int] = sign * abs(a // g)
        #: the denominator of the fraction will always be positive
        self.b: Final[int] = abs(b // g)
```

Fraction: Initialisierer und Konstanten

- Ohne den Code von `__init__` zu lesen, kann ein Benutzer bereits viel über unsere Klasse `Fraction` nur von den Doctests lernen.
- Nun ist es so, dass einige spezielle Brüche in vielen Berechnungen vorkommen.
- Anstatt sie immer wieder neu anzulegen, könnten wir sie als Konstanten definieren.
- Eine Konstante ist eine Variable, die sich niemals verändern kann.

```
class Fraction:
    """The class for fractions, i.e., rational numbers."""

    def __init__(self, a: int, b: int = 1) -> None:
        """
        Create a normalized fraction.

        :param a: the numerator
        :param b: the denominator

        >>> f={Fraction(12, 1).a}, {Fraction(12, 1).b}
        '12, 1'
        >>> f={Fraction(12, 2).a}, {Fraction(12, 2).b}
        '6, 1'
        >>> f={Fraction(2, 12).a}, {Fraction(2, 12).b}
        '1, 6'
        >>> f={Fraction(2, -12).a}, {Fraction(2, -12).b}
        '-1, 6'
        >>> f={Fraction(-2, -12).a}, {Fraction(-2, -12).b}
        '1, 6'
        >>> f={Fraction(-2, 12).a}, {Fraction(-2, 12).b}
        '-1, 6'
        >>> f={Fraction(0, -9).a}, {Fraction(0, -9).b}
        '0, 1'
        >>> try:
            ...     Fraction(1, 0)
            ... except ZeroDivisionError as z:
            ...     print(z)
        1/0
        """

        if b == 0: # A denominator of zero is not permitted.
            raise ZeroDivisionError(f"{a}/{b}")

        g: int = gcd(a, b) # We use the GCD to normalize the fraction
        sign: int = -1 if ((a < 0) != (b < 0)) else 1 # The sign.
        #: the numerator of the fraction will also include the sign
        self.a: Final[int] = sign * abs(a // g)
        #: the denominator of the fraction will always be positive
        self.b: Final[int] = abs(b // g)
```

Fraction: Initialisierer und Konstanten

- Ohne den Code von `__init__` zu lesen, kann ein Benutzer bereits viel über unsere Klasse `Fraction` nur von den Doctests lernen.
- Nun ist es so, dass einige spezielle Brüche in vielen Berechnungen vorkommen.
- Anstatt sie immer wieder neu anzulegen, könnten wir sie als Konstanten definieren.
- Eine Konstante ist eine Variable, die sich niemals verändern kann.

Gute Praxis

Konstanten sind Modul-level Variablen denen ein Wert bei ihrer Erstellung zugewiesen wird und die mit dem Type Hint `Final` annotiert werden²¹.

Fraction: Initialisierer und Konstanten

- Ohne den Code von `__init__` zu lesen, kann ein Benutzer bereits viel über unsere Klasse `Fraction` nur von den Doctests lernen.
- Nun ist es so, dass einige spezielle Brüche in vielen Berechnungen vorkommen.
- Anstatt sie immer wieder neu anzulegen, könnten wir sie als Konstanten definieren.
- Eine Konstante ist eine Variable, die sich niemals verändern kann.

Gute Praxis

Konstanten sind Modul-level Variablen denen ein Wert bei ihrer Erstellung zugewiesen wird und die mit dem Type Hint `Final` annotiert werden²¹.

Gute Praxis

Die Namen von Konstanten bestehen nur aus Großbuchstaben, wobei Unterstriche Worte trennen. Beispiele sind `MAX_OVERFLOW` und `TOTAL`²⁸.

Fraction: Initialisierer und Konstanten

- Ohne den Code von `__init__` zu lesen, kann ein Benutzer bereits viel über unsere Klasse `Fraction` nur von den Doctests lernen.
- Nun ist es so, dass einige spezielle Brüche in vielen Berechnungen vorkommen.
- Anstatt sie immer wieder neu anzulegen, könnten wir sie als Konstanten definieren.
- Eine Konstante ist eine Variable, die sich niemals verändern kann.

Gute Praxis

Konstanten sind Modul-level Variablen denen ein Wert bei ihrer Erstellung zugewiesen wird und die mit dem Type Hint `Final` annotiert werden²¹.

Gute Praxis

Die Namen von Konstanten bestehen nur aus Großbuchstaben, wobei Unterstriche Worte trennen. Beispiele sind `MAX_OVERFLOW` und `TOTAL`²⁸.

Gute Praxis

Konstanten werden dokumentiert, in dem ein Kommentar der mit `#:` anfängt direkt über die Konstantendefinition geschrieben wird²⁰.

Fraction: Initialisierer und Konstanten

- Ohne den Code von `__init__` zu lesen, kann ein Benutzer bereits viel über unsere Klasse `Fraction` nur von den Doctests lernen.
- Nun ist es so, dass einige spezielle Brüche in vielen Berechnungen vorkommen.
- Anstatt sie immer wieder neu anzulegen, könnten wir sie als Konstanten definieren.
- Eine Konstante ist eine Variable, die sich niemals verändern kann.
- Wir definieren die drei Konstanten `ZERO`, `ONE` und `ONE_HALF`.

```
8 class Fraction:
9     """The class for fractions, i.e., rational numbers."""
10
11     def __init__(self, a: int, b: int = 1) -> None:
12         """
13         Create a normalized fraction.
14
15         :param a: the numerator
16         :param b: the denominator
17
18         >>> f"{Fraction(12, 1).a}, {Fraction(12, 1).b}"
19         '12, 1'
20         >>> f"{Fraction(12, 2).a}, {Fraction(12, 2).b}"
21         '6, 1'
22         >>> f"{Fraction(2, 12).a}, {Fraction(2, 12).b}"
23         '1, 6'
24         >>> f"{Fraction(2, -12).a}, {Fraction(2, -12).b}"
25         '-1, 6'
26         >>> f"{Fraction(-2, -12).a}, {Fraction(-2, -12).b}"
27         '1, 6'
28         >>> f"{Fraction(-2, 12).a}, {Fraction(-2, 12).b}"
29         '-1, 6'
30         >>> f"{Fraction(0, -9).a}, {Fraction(0, -9).b}"
31         '0, 1'
32         >>> try:
33             ... Fraction(1, 0)
34             ... except ZeroDivisionError as z:
35                 ... print(z)
36         1/0
37         """
38         if b == 0: # A denominator of zero is not permitted.
39             raise ZeroDivisionError(f"{a}/{b}")
40         g: int = gcd(a, b) # We use the GCD to normalize the fraction.
41         sign: int = -1 if ((a < 0) != (b < 0)) else 1 # The sign.
42         #: the numerator of the fraction will also include the sign
43         self.a: Final[int] = sign * abs(a // g)
44         #: the denominator of the fraction will always be positive
45         self.b: Final[int] = abs(b // g)
46
47         #: the constant zero
48         ZERO: Final[Fraction] = Fraction(0, 1)
49         #: the constant one
50         ONE: Final[Fraction] = Fraction(1, 1)
51         #: the constant 0.5
52         ONE_HALF: Final[Fraction] = Fraction(1, 2)
```

Fraction: Initialisierer und Konstanten

- Nun ist es so, dass einige spezielle Brüche in vielen Berechnungen vorkommen.
- Anstatt sie immer wieder neu anzulegen, könnten wir sie als Konstanten definieren.
- Eine Konstante ist eine Variable, die sich niemals verändern kann.
- Wir definieren die drei Konstanten `ZERO`, `ONE` und `ONE_HALF`.
- Diese Brüche werden oft benutzt und sie als Konstanten vorzuhalten spart Laufzeit und Speicher.

```
class Fraction:
    """The class for fractions, i.e., rational numbers."""

    def __init__(self, a: int, b: int = 1) -> None:
        """
        Create a normalized fraction.

        :param a: the numerator
        :param b: the denominator

        >>> f"{Fraction(12, 1).a}, {Fraction(12, 1).b}"
        '12, 1'
        >>> f"{Fraction(12, 2).a}, {Fraction(12, 2).b}"
        '6, 1'
        >>> f"{Fraction(2, 12).a}, {Fraction(2, 12).b}"
        '1, 6'
        >>> f"{Fraction(2, -12).a}, {Fraction(2, -12).b}"
        '-1, 6'
        >>> f"{Fraction(-2, -12).a}, {Fraction(-2, -12).b}"
        '1, 6'
        >>> f"{Fraction(-2, 12).a}, {Fraction(-2, 12).b}"
        '-1, 6'
        >>> f"{Fraction(0, -9).a}, {Fraction(0, -9).b}"
        '0, 1'
        >>> try:
            ... Fraction(1, 0)
            ... except ZeroDivisionError as z:
            ...     print(z)
        1/0
        """
        if b == 0: # A denominator of zero is not permitted.
            raise ZeroDivisionError(f"{a}/{b}")
        g: int = gcd(a, b) # We use the GCD to normalize the fraction.
        sign: int = -1 if ((a < 0) != (b < 0)) else 1 # The sign.
        #: the numerator of the fraction will also include the sign
        self.a: Final[int] = sign * abs(a // g)
        #: the denominator of the fraction will always be positive
        self.b: Final[int] = abs(b // g)

    #: the constant zero
    ZERO: Final[Fraction] = Fraction(0, 1)
    #: the constant one
    ONE: Final[Fraction] = Fraction(1, 1)
    #: the constant 0.5
    ONE_HALF: Final[Fraction] = Fraction(1, 2)
```

Fraction: __str__ und __repr__

- Ist Ihnen aufgefallen, dass wir die Doctests im Docstring unserer Klasse sehr umständlich geschrieben haben?

```
1  def __str__(self) -> str:
2      """
3      Convert this number to a string fractional.
4
5      :return: the string representation
6
7      >>> print(Fraction(-5, 12))
8      -5/12
9      >>> print(Fraction(3, -1))
10     -3
11     >>> print(Fraction(12, 23))
12     12/23
13     """
14     return str(self.a) if self.b == 1 else f"{self.a}/{self.b}"
15
16  def __repr__(self) -> str:
17      """
18      Convert this number to a string.
19
20      :return: the string representation
21
22      >>> Fraction(-5, 12)
23      Fraction(-5, 12)
24      >>> Fraction(3, -1)
25      Fraction(-3, 1)
26      >>> Fraction(12, 23)
27      Fraction(12, 23)
28      """
29     return f"Fraction({self.a}, {self.b})"
```

Fraction: `__str__` und `__repr__`

- Ist Ihnen aufgefallen, dass wir die Doctests im Docstring unserer Klasse sehr umständlich geschrieben haben?
- Das war weil wir `__str__` und `__repr__` noch nicht definiert hatten.

```
1  def __str__(self) -> str:
2      """
3      Convert this number to a string fractional.
4
5      :return: the string representation
6
7      >>> print(Fraction(-5, 12))
8      -5/12
9      >>> print(Fraction(3, -1))
10     -3
11     >>> print(Fraction(12, 23))
12     12/23
13     """
14     return str(self.a) if self.b == 1 else f"{self.a}/{self.b}"
15
16  def __repr__(self) -> str:
17      """
18      Convert this number to a string.
19
20      :return: the string representation
21
22      >>> Fraction(-5, 12)
23      Fraction(-5, 12)
24      >>> Fraction(3, -1)
25      Fraction(-3, 1)
26      >>> Fraction(12, 23)
27      Fraction(12, 23)
28      """
29     return f"Fraction({self.a}, {self.b})"
```

Fraction: `__str__` und `__repr__`

- Ist Ihnen aufgefallen, dass wir die Doctests im Docstring unserer Klasse sehr umständlich geschrieben haben?
- Das war weil wir `__str__` und `__repr__` noch nicht definiert hatten.
- Das machen wir jetzt.

```
1 def __str__(self) -> str:
2     """
3     Convert this number to a string fractional.
4
5     :return: the string representation
6
7     >>> print(Fraction(-5, 12))
8     -5/12
9     >>> print(Fraction(3, -1))
10    -3
11    >>> print(Fraction(12, 23))
12    12/23
13    """
14    return str(self.a) if self.b == 1 else f"{self.a}/{self.b}"
15
16 def __repr__(self) -> str:
17     """
18     Convert this number to a string.
19
20     :return: the string representation
21
22     >>> Fraction(-5, 12)
23     Fraction(-5, 12)
24     >>> Fraction(3, -1)
25     Fraction(-3, 1)
26     >>> Fraction(12, 23)
27     Fraction(12, 23)
28     """
29    return f"Fraction({self.a}, {self.b})"
```

Fraction: `__str__` und `__repr__`

- Ist Ihnen aufgefallen, dass wir die Doctests im Docstring unserer Klasse sehr umständlich geschrieben haben?
- Das war weil wir `__str__` und `__repr__` noch nicht definiert hatten.
- Das machen wir jetzt.
- Die Methode `__str__` soll eine kompakte Repräsentation eines Objekts liefern.

```
1  def __str__(self) -> str:
2      """
3      Convert this number to a string fractional.
4
5      :return: the string representation
6
7      >>> print(Fraction(-5, 12))
8      -5/12
9      >>> print(Fraction(3, -1))
10     -3
11     >>> print(Fraction(12, 23))
12     12/23
13     """
14     return str(self.a) if self.b == 1 else f"{self.a}/{self.b}"
15
16 def __repr__(self) -> str:
17     """
18     Convert this number to a string.
19
20     :return: the string representation
21
22     >>> Fraction(-5, 12)
23     Fraction(-5, 12)
24     >>> Fraction(3, -1)
25     Fraction(-3, 1)
26     >>> Fraction(12, 23)
27     Fraction(12, 23)
28     """
29     return f"Fraction({self.a}, {self.b})"
```

Fraction: `__str__` und `__repr__`

- Ist Ihnen aufgefallen, dass wir die Doctests im Docstring unserer Klasse sehr umständlich geschrieben haben?
- Das war weil wir `__str__` und `__repr__` noch nicht definiert hatten.
- Das machen wir jetzt.
- Die Methode `__str__` soll eine kompakte Repräsentation eines Objekts liefern.
- Wir implementieren sie so, dass sie `self.a` als String zurückliefert, wenn der Nenner 1 ist, also wenn `self.b == 1`.

```
1 def __str__(self) -> str:
2     """
3     Convert this number to a string fractional.
4
5     :return: the string representation
6
7     >>> print(Fraction(-5, 12))
8     -5/12
9     >>> print(Fraction(3, -1))
10    -3
11    >>> print(Fraction(12, 23))
12    12/23
13    """
14    return str(self.a) if self.b == 1 else f"{self.a}/{self.b}"
15
16 def __repr__(self) -> str:
17     """
18     Convert this number to a string.
19
20     :return: the string representation
21
22     >>> Fraction(-5, 12)
23     Fraction(-5, 12)
24     >>> Fraction(3, -1)
25     Fraction(-3, 1)
26     >>> Fraction(12, 23)
27     Fraction(12, 23)
28     """
29    return f"Fraction({self.a}, {self.b})"
```

Fraction: `__str__` und `__repr__`

- Ist Ihnen aufgefallen, dass wir die Doctests im Docstring unserer Klasse sehr umständlich geschrieben haben?
- Das war weil wir `__str__` und `__repr__` noch nicht definiert hatten.
- Das machen wir jetzt.
- Die Methode `__str__` soll eine kompakte Repräsentation eines Objekts liefern.
- Wir implementieren sie so, dass sie `self.a` als String zurückliefert, wenn der Nenner 1 ist, also wenn `self.b == 1`.
- Dann ist der Bruch ja eine Ganzzahl.

```
1 def __str__(self) -> str:
2     """
3     Convert this number to a string fractional.
4
5     :return: the string representation
6
7     >>> print(Fraction(-5, 12))
8     -5/12
9     >>> print(Fraction(3, -1))
10    -3
11    >>> print(Fraction(12, 23))
12    12/23
13    """
14    return str(self.a) if self.b == 1 else f"{self.a}/{self.b}"
15
16 def __repr__(self) -> str:
17     """
18     Convert this number to a string.
19
20     :return: the string representation
21
22     >>> Fraction(-5, 12)
23     Fraction(-5, 12)
24     >>> Fraction(3, -1)
25     Fraction(-3, 1)
26     >>> Fraction(12, 23)
27     Fraction(12, 23)
28     """
29    return f"Fraction({self.a}, {self.b})"
```

Fraction: `__str__` und `__repr__`

- Das war weil wir `__str__` und `__repr__` noch nicht definiert hatten.
- Das machen wir jetzt.
- Die Methode `__str__` soll eine kompakte Repräsentation eines Objekts liefern.
- Wir implementieren sie so, dass sie `self.a` als String zurückliefert, wenn der Nenner 1 ist, also wenn `self.b == 1`.
- Dann ist der Bruch ja eine Ganzzahl.
- Sonst soll sie `f"{self.a}/{self.b}"` liefern.

```
1  def __str__(self) -> str:
2      """
3      Convert this number to a string fractional.
4
5      :return: the string representation
6
7      >>> print(Fraction(-5, 12))
8      -5/12
9      >>> print(Fraction(3, -1))
10     -3
11     >>> print(Fraction(12, 23))
12     12/23
13     """
14     return str(self.a) if self.b == 1 else f"{self.a}/{self.b}"
15
16  def __repr__(self) -> str:
17      """
18      Convert this number to a string.
19
20      :return: the string representation
21
22      >>> Fraction(-5, 12)
23      Fraction(-5, 12)
24      >>> Fraction(3, -1)
25      Fraction(-3, 1)
26      >>> Fraction(12, 23)
27      Fraction(12, 23)
28      """
29     return f"Fraction({self.a}, {self.b})"
```

Fraction: `__str__` und `__repr__`

- Das machen wir jetzt.
- Die Methode `__str__` soll eine kompakte Repräsentation eines Objekts liefern.
- Wir implementieren sie so, dass sie `self.a` als String zurückliefert, wenn der Nenner 1 ist, also wenn `self.b == 1`.
- Dann ist der Bruch ja eine Ganzzahl.
- Sonst soll sie `f"{self.a}/{self.b}"` liefern.
- Das ist klar genug, um den Wert eines Bruches zu erkennen.

```
1 def __str__(self) -> str:
2     """
3     Convert this number to a string fractional.
4
5     :return: the string representation
6
7     >>> print(Fraction(-5, 12))
8     -5/12
9     >>> print(Fraction(3, -1))
10    -3
11    >>> print(Fraction(12, 23))
12    12/23
13    """
14    return str(self.a) if self.b == 1 else f"{self.a}/{self.b}"
15
16 def __repr__(self) -> str:
17     """
18     Convert this number to a string.
19
20     :return: the string representation
21
22     >>> Fraction(-5, 12)
23     Fraction(-5, 12)
24     >>> Fraction(3, -1)
25     Fraction(-3, 1)
26     >>> Fraction(12, 23)
27     Fraction(12, 23)
28     """
29    return f"Fraction({self.a}, {self.b})"
```

Fraction: `__str__` und `__repr__`

- Wir implementieren sie so, dass sie `self.a` als String zurückliefert, wenn der Nenner 1 ist, also wenn `self.b == 1`.
- Dann ist der Bruch ja eine Ganzzahl.
- Sonst soll sie `f"{self.a}/{self.b}"` liefern.
- Das ist klar genug, um den Wert eines Bruches zu erkennen.
- Es ist aber auch uneindeutig, denn wir können `str(Fraction(12, 1))` und `str(12)` nicht voneinander unterscheiden.

```
1 def __str__(self) -> str:
2     """
3     Convert this number to a string fractional.
4
5     :return: the string representation
6
7     >>> print(Fraction(-5, 12))
8     -5/12
9     >>> print(Fraction(3, -1))
10    -3
11    >>> print(Fraction(12, 23))
12    12/23
13    """
14    return str(self.a) if self.b == 1 else f"{self.a}/{self.b}"
15
16 def __repr__(self) -> str:
17     """
18     Convert this number to a string.
19
20     :return: the string representation
21
22     >>> Fraction(-5, 12)
23     Fraction(-5, 12)
24     >>> Fraction(3, -1)
25     Fraction(-3, 1)
26     >>> Fraction(12, 23)
27     Fraction(12, 23)
28     """
29    return f"Fraction({self.a}, {self.b})"
```

Fraction: `__str__` und `__repr__`

- Dann ist der Bruch ja eine Ganzzahl.
- Sonst soll sie `f"{self.a}/{self.b}"` liefern.
- Das ist klar genug, um den Wert eines Bruches zu erkennen.
- Es ist aber auch uneindeutig, denn wir können `str(Fraction(12, 1))` und `str(12)` nicht voneinander unterscheiden.
- Brüche, die Ganzzahlen repräsentieren, haben die gleiche textuelle repräsentation wie diese Ganzzahlen.

```
1 def __str__(self) -> str:
2     """
3     Convert this number to a string fractional.
4
5     :return: the string representation
6
7     >>> print(Fraction(-5, 12))
8     -5/12
9     >>> print(Fraction(3, -1))
10    -3
11    >>> print(Fraction(12, 23))
12    12/23
13    """
14    return str(self.a) if self.b == 1 else f"{self.a}/{self.b}"
15
16 def __repr__(self) -> str:
17     """
18     Convert this number to a string.
19
20     :return: the string representation
21
22     >>> Fraction(-5, 12)
23     Fraction(-5, 12)
24     >>> Fraction(3, -1)
25     Fraction(-3, 1)
26     >>> Fraction(12, 23)
27     Fraction(12, 23)
28     """
29    return f"Fraction({self.a}, {self.b})"
```

Fraction: `__str__` und `__repr__`

- Sonst soll sie `f"{self.a}/{self.b}"` liefern.
- Das ist klar genug, um den Wert eines Bruches zu erkennen.
- Es ist aber auch uneindeutig, denn wir können `str(Fraction(12, 1))` und `str(12)` nicht voneinander unterscheiden.
- Brüche, die Ganzzahlen repräsentieren, haben die gleiche textuelle repräsentation wie diese Ganzzahlen.
- Die `__repr__`-Methode existiert, um eindeutige und klare Ausgaben zu erzeugen.

```
1 def __str__(self) -> str:
2     """
3     Convert this number to a string fractional.
4
5     :return: the string representation
6
7     >>> print(Fraction(-5, 12))
8     -5/12
9     >>> print(Fraction(3, -1))
10    -3
11    >>> print(Fraction(12, 23))
12    12/23
13    """
14    return str(self.a) if self.b == 1 else f"{self.a}/{self.b}"
15
16 def __repr__(self) -> str:
17     """
18     Convert this number to a string.
19
20     :return: the string representation
21
22     >>> Fraction(-5, 12)
23     Fraction(-5, 12)
24     >>> Fraction(3, -1)
25     Fraction(-3, 1)
26     >>> Fraction(12, 23)
27     Fraction(12, 23)
28     """
29    return f"Fraction({self.a}, {self.b})"
```

Fraction: `__str__` und `__repr__`

- Das ist klar genug, um den Wert eines Bruches zu erkennen.
- Es ist aber auch uneindeutig, denn wir können `str(Fraction(12, 1))` und `str(12)` nicht voneinander unterscheiden.
- Brüche, die Ganzzahlen repräsentieren, haben die gleiche textuelle repräsentation wie diese Ganzzahlen.
- Die `__repr__`-Methode existiert, um eindeutige und klare Ausgaben zu erzeugen.
- Wir implementieren sie so, dass sie `f"Fraction({self.a}, {self.b})"` liefert.

```
1 def __str__(self) -> str:
2     """
3     Convert this number to a string fractional.
4
5     :return: the string representation
6
7     >>> print(Fraction(-5, 12))
8     -5/12
9     >>> print(Fraction(3, -1))
10    -3
11    >>> print(Fraction(12, 23))
12    12/23
13    """
14    return str(self.a) if self.b == 1 else f"{self.a}/{self.b}"
15
16 def __repr__(self) -> str:
17     """
18     Convert this number to a string.
19
20     :return: the string representation
21
22     >>> Fraction(-5, 12)
23     Fraction(-5, 12)
24     >>> Fraction(3, -1)
25     Fraction(-3, 1)
26     >>> Fraction(12, 23)
27     Fraction(12, 23)
28     """
29    return f"Fraction({self.a}, {self.b})"
```

Fraction: `__str__` und `__repr__`

- Es ist aber auch uneindeutig, denn wir können `str(Fraction(12, 1))` und `str(12)` nicht voneinander unterscheiden.
- Brüche, die Ganzzahlen repräsentieren, haben die gleiche textuelle repräsentation wie diese Ganzzahlen.
- Die `__repr__`-Methode existiert, um eindeutige und klare Ausgaben zu erzeugen.
- Wir implementieren sie so, dass sie `f"Fraction({self.a}, {self.b})"` liefert.
- In den Docstrings beider Methoden schreiben wir wieder Doctests.

```
1 def __str__(self) -> str:
2     """
3     Convert this number to a string fractional.
4
5     :return: the string representation
6
7     >>> print(Fraction(-5, 12))
8     -5/12
9     >>> print(Fraction(3, -1))
10    -3
11    >>> print(Fraction(12, 23))
12    12/23
13    """
14    return str(self.a) if self.b == 1 else f"{self.a}/{self.b}"
15
16 def __repr__(self) -> str:
17     """
18     Convert this number to a string.
19
20     :return: the string representation
21
22     >>> Fraction(-5, 12)
23     Fraction(-5, 12)
24     >>> Fraction(3, -1)
25     Fraction(-3, 1)
26     >>> Fraction(12, 23)
27     Fraction(12, 23)
28     """
29    return f"Fraction({self.a}, {self.b})"
```

Fraction: `__str__` und `__repr__`

- Brüche, die Ganzzahlen repräsentieren, haben die gleiche textuelle repräsentation wie diese Ganzzahlen.
- Die `__repr__`-Methode existiert, um eindeutige und klare Ausgaben zu erzeugen.
- Wir implementieren sie so, dass sie `f"Fraction({self.a}, {self.b})"` liefert.
- In den Docstrings beider Methoden schreiben wir wieder Doctests.
- Die Dunder-Methode `__str__` wird automatisch verwendet, wenn wir ein Objekt an `print` übergeben.

```
1 def __str__(self) -> str:
2     """
3     Convert this number to a string fractional.
4
5     :return: the string representation
6
7     >>> print(Fraction(-5, 12))
8     -5/12
9     >>> print(Fraction(3, -1))
10    -3
11    >>> print(Fraction(12, 23))
12    12/23
13    """
14    return str(self.a) if self.b == 1 else f"{self.a}/{self.b}"
15
16 def __repr__(self) -> str:
17     """
18     Convert this number to a string.
19
20     :return: the string representation
21
22     >>> Fraction(-5, 12)
23     Fraction(-5, 12)
24     >>> Fraction(3, -1)
25     Fraction(-3, 1)
26     >>> Fraction(12, 23)
27     Fraction(12, 23)
28     """
29    return f"Fraction({self.a}, {self.b})"
```

Fraction: `__str__` und `__repr__`

- Die `__repr__`-Methode existiert, um eindeutige und klare Ausgaben zu erzeugen.
- Wir implementieren sie so, dass sie `f"Fraction({self.a}, {self.b})"` liefert.
- In den Docstrings beider Methoden schreiben wir wieder Doctests.
- Die Dunder-Methode `__str__` wird automatisch verwendet, wenn wir ein Objekt an `print` übergeben.
- Das bedeutet, dass wir den erwarteten Output von `f.__str__()` für einen Bruch `f` mit dem Ergebnis von `print(f)` vergleichen können.

```
1 def __str__(self) -> str:
2     """
3     Convert this number to a string fractional.
4
5     :return: the string representation
6
7     >>> print(Fraction(-5, 12))
8     -5/12
9     >>> print(Fraction(3, -1))
10    -3
11    >>> print(Fraction(12, 23))
12    12/23
13    """
14    return str(self.a) if self.b == 1 else f"{self.a}/{self.b}"
15
16 def __repr__(self) -> str:
17     """
18     Convert this number to a string.
19
20     :return: the string representation
21
22     >>> Fraction(-5, 12)
23     Fraction(-5, 12)
24     >>> Fraction(3, -1)
25     Fraction(-3, 1)
26     >>> Fraction(12, 23)
27     Fraction(12, 23)
28     """
29    return f"Fraction({self.a}, {self.b})"
```

Fraction: `__str__` und `__repr__`

- Wir implementieren sie so, dass sie `f"Fraction({self.a}, {self.b})"` liefert.
- In den Docstrings beider Methoden schreiben wir wieder Doctests.
- Die Dunder-Methode `__str__` wird automatisch verwendet, wenn wir ein Objekt an `print` übergeben.
- Das bedeutet, dass wir den erwarteten Output von `f.__str__()` für einen Bruch `f` mit dem Ergebnis von `print(f)` vergleichen können.
- Andernfalls konvertieren Doctests Objekte immer mit `repr` zu strings.

```
1 def __str__(self) -> str:
2     """
3     Convert this number to a string fractional.
4
5     :return: the string representation
6
7     >>> print(Fraction(-5, 12))
8     -5/12
9     >>> print(Fraction(3, -1))
10    -3
11    >>> print(Fraction(12, 23))
12    12/23
13    """
14    return str(self.a) if self.b == 1 else f"{self.a}/{self.b}"
15
16 def __repr__(self) -> str:
17     """
18     Convert this number to a string.
19
20     :return: the string representation
21
22     >>> Fraction(-5, 12)
23     Fraction(-5, 12)
24     >>> Fraction(3, -1)
25     Fraction(-3, 1)
26     >>> Fraction(12, 23)
27     Fraction(12, 23)
28     """
29    return f"Fraction({self.a}, {self.b})"
```

Fraction: `__str__` und `__repr__`

- In den Docstrings beider Methoden schreiben wir wieder Doctests.
- Die Dunder-Methode `__str__` wird automatisch verwendet, wenn wir ein Objekt an `print` übergeben.
- Das bedeutet, dass wir den erwarteten Output von `f.__str__()` für einen Bruch `f` mit dem Ergebnis von `print(f)` vergleichen können.
- Andernfalls konvertieren Doctests Objekte immer mit `repr` zu strings.
- Das bedeutet, dass die Zeile `Fraction(-5, 12)` in dem Doctest von `__repr__` tatsächlich `repr(Fraction(-5, 12))` aufruft.

```
1 def __str__(self) -> str:
2     """
3     Convert this number to a string fractional.
4
5     :return: the string representation
6
7     >>> print(Fraction(-5, 12))
8     -5/12
9     >>> print(Fraction(3, -1))
10    -3
11    >>> print(Fraction(12, 23))
12    12/23
13    """
14    return str(self.a) if self.b == 1 else f"{self.a}/{self.b}"
15
16 def __repr__(self) -> str:
17     """
18     Convert this number to a string.
19
20     :return: the string representation
21
22     >>> Fraction(-5, 12)
23     Fraction(-5, 12)
24     >>> Fraction(3, -1)
25     Fraction(-3, 1)
26     >>> Fraction(12, 23)
27     Fraction(12, 23)
28     """
29    return f"Fraction({self.a}, {self.b})"
```

Fraction: `__str__` und `__repr__`

- Die Dunder-Methode `__str__` wird automatisch verwendet, wenn wir ein Objekt an `print` übergeben.
- Das bedeutet, dass wir den erwarteten Output von `f.__str__()` für einen Bruch `f` mit dem Ergebnis von `print(f)` vergleichen können.
- Andernfalls konvertieren Doctests Objekte immer mit `repr` zu strings.
- Das bedeutet, dass die Zeile `Fraction(-5, 12)` in dem Doctest von `__repr__` tatsächlich `repr(Fraction(-5, 12))` aufruft.
- Mit der String-Konversation aus dem Weg können wir nun mathematische Operatoren implementieren.

```
1 def __str__(self) -> str:
2     """
3     Convert this number to a string fractional.
4
5     :return: the string representation
6
7     >>> print(Fraction(-5, 12))
8     -5/12
9     >>> print(Fraction(3, -1))
10    -3
11    >>> print(Fraction(12, 23))
12    12/23
13    """
14    return str(self.a) if self.b == 1 else f"{self.a}/{self.b}"
15
16 def __repr__(self) -> str:
17     """
18     Convert this number to a string.
19
20     :return: the string representation
21
22     >>> Fraction(-5, 12)
23     Fraction(-5, 12)
24     >>> Fraction(3, -1)
25     Fraction(-3, 1)
26     >>> Fraction(12, 23)
27     Fraction(12, 23)
28     """
29    return f"Fraction({self.a}, {self.b})"
```

Fraction: Addition und Subtraktion

- Wir wollen nun Instanzen unserer Klasse `Fraction` mit den `+` und `-` Operatoren verwendbar machen.

```
def __add__(self, other) -> Union[NotImplementedType, "Fraction"]:  
    """  
    Add this fraction to another fraction.  
  
    :param other: the other number  
    :return: the result of the addition  
  
    >>> print(Fraction(1, 3) + Fraction(1, 2))  
    5/6  
    >>> print(Fraction(1, 2) + Fraction(1, 2))  
    1  
    >>> print(Fraction(21, -12) + Fraction(-33, 42))  
    -71/28  
    """  
    return Fraction((self.a * other.b) + (other.a * self.b),  
                    self.b * other.b) if isinstance(other, Fraction)  
    ↪ \n  
    else NotImplemented  
  
def __sub__(self, other) -> Union[NotImplementedType, "Fraction"]:  
    """  
    Subtract this fraction from another fraction.  
  
    :param other: the other fraction  
    :return: the result of the subtraction  
  
    >>> print(Fraction(1, 3) - Fraction(1, 2))  
    -1/6  
    >>> print(Fraction(1, 2) - Fraction(3, 6))  
    0  
    >>> print(Fraction(21, -12) - Fraction(-33, 42))  
    -27/28  
    """  
    return Fraction(  
        (self.a * other.b) - (other.a * self.b), self.b * other.b) \n  
        if isinstance(other, Fraction) else NotImplemented
```

Fraction: Addition und Subtraktion

- Wir wollen nun Instanzen unserer Klasse `Fraction` mit den `+` und `-` Operatoren verwendbar machen.
- In Python ruft `x + y` nämlich `x.__add__(y)` auf, wenn die Klasse von `x` die Dunder-Methode `__add__` definiert.

```
def __add__(self, other) -> Union[NotImplementedType, "Fraction"]:  
    """  
    Add this fraction to another fraction.  
  
    :param other: the other number  
    :return: the result of the addition  
  
    >>> print(Fraction(1, 3) + Fraction(1, 2))  
    5/6  
    >>> print(Fraction(1, 2) + Fraction(1, 2))  
    1  
    >>> print(Fraction(21, -12) + Fraction(-33, 42))  
    -71/28  
    """  
    return Fraction((self.a * other.b) + (other.a * self.b),  
                    self.b * other.b) if isinstance(other, Fraction)  
    else NotImplemented  
  
def __sub__(self, other) -> Union[NotImplementedType, "Fraction"]:  
    """  
    Subtract this fraction from another fraction.  
  
    :param other: the other fraction  
    :return: the result of the subtraction  
  
    >>> print(Fraction(1, 3) - Fraction(1, 2))  
    -1/6  
    >>> print(Fraction(1, 2) - Fraction(3, 6))  
    0  
    >>> print(Fraction(21, -12) - Fraction(-33, 42))  
    -27/28  
    """  
    return Fraction(  
        (self.a * other.b) - (other.a * self.b), self.b * other.b)  
        if isinstance(other, Fraction) else NotImplemented
```

Fraction: Addition und Subtraktion

- Wir wollen nun Instanzen unserer Klasse `Fraction` mit den `+` und `-` Operatoren verwendbar machen.
- In Python ruft `x + y` nämlich `x.__add__(y)` auf, wenn die Klasse von `x` die Dunder-Methode `__add__` definiert.
- In der Grundschule haben wir gelernt, dass $\frac{a}{b} + \frac{c}{d} = \frac{a*d+c*b}{b*d}$ ist, für $b, d \neq 0$.

```
def __add__(self, other) -> Union[NotImplementedType, "Fraction"]:  
    """  
    Add this fraction to another fraction.  
  
    :param other: the other number  
    :return: the result of the addition  
  
    >>> print(Fraction(1, 3) + Fraction(1, 2))  
    5/6  
    >>> print(Fraction(1, 2) + Fraction(1, 2))  
    1  
    >>> print(Fraction(21, -12) + Fraction(-33, 42))  
    -71/28  
    """  
    return Fraction((self.a * other.b) + (other.a * self.b),  
                    self.b * other.b) if isinstance(other, Fraction)  
    else NotImplemented  
  
def __sub__(self, other) -> Union[NotImplementedType, "Fraction"]:  
    """  
    Subtract this fraction from another fraction.  
  
    :param other: the other fraction  
    :return: the result of the subtraction  
  
    >>> print(Fraction(1, 3) - Fraction(1, 2))  
    -1/6  
    >>> print(Fraction(1, 2) - Fraction(3, 6))  
    0  
    >>> print(Fraction(21, -12) - Fraction(-33, 42))  
    -27/28  
    """  
    return Fraction(  
        (self.a * other.b) - (other.a * self.b), self.b * other.b) \   
        if isinstance(other, Fraction) else NotImplemented
```

Fraction: Addition und Subtraktion

- Wir wollen nun Instanzen unserer Klasse `Fraction` mit den `+` und `-` Operatoren verwendbar machen.
- In Python ruft `x + y` nämlich `x.__add__(y)` auf, wenn die Klasse von `x` die Dunder-Methode `__add__` definiert.
- In der Grundschule haben wir gelernt, dass $\frac{a}{b} + \frac{c}{d} = \frac{a*d+c*b}{b*d}$ ist, für $b, d \neq 0$.
- Wenn `other` auch eine Instanz von `Fraction` ist, dann berechnet `__add__(other)` das Ergebnis genau so und liefert eine neue `Fraction` zurück.

```
def __add__(self, other) -> Union[NotImplementedType, "Fraction"]:  
    """  
    Add this fraction to another fraction.  
  
    :param other: the other number  
    :return: the result of the addition  
  
    >>> print(Fraction(1, 3) + Fraction(1, 2))  
    5/6  
    >>> print(Fraction(1, 2) + Fraction(1, 2))  
    1  
    >>> print(Fraction(21, -12) + Fraction(-33, 42))  
    -71/28  
    """  
    return Fraction((self.a * other.b) + (other.a * self.b),  
                    self.b * other.b) if isinstance(other, Fraction)  
                    else NotImplemented  
  
def __sub__(self, other) -> Union[NotImplementedType, "Fraction"]:  
    """  
    Subtract this fraction from another fraction.  
  
    :param other: the other fraction  
    :return: the result of the subtraction  
  
    >>> print(Fraction(1, 3) - Fraction(1, 2))  
    -1/6  
    >>> print(Fraction(1, 2) - Fraction(3, 6))  
    0  
    >>> print(Fraction(21, -12) - Fraction(-33, 42))  
    -27/28  
    """  
    return Fraction(  
        (self.a * other.b) - (other.a * self.b), self.b * other.b) \  
        if isinstance(other, Fraction) else NotImplemented
```

Fraction: Addition und Subtraktion

- In Python ruft `x + y` nämlich `x.__add__(y)` auf, wenn die Klasse von `x` die Dunder-Methode `__add__` definiert.
- In der Grundschule haben wir gelernt, dass $\frac{a}{b} + \frac{c}{d} = \frac{a*d+c*b}{b*d}$ ist, für $b, d \neq 0$.
- Wenn `other` auch eine Instanz von `Fraction` ist, dann berechnet `__add__(other)` das Ergebnis genau so und liefert eine neue `Fraction` zurück.
- Der Initialisierer des neuen Bruchs wird diesen automatisch unter Benutzung des ggT normalisieren.

```
def __add__(self, other) -> Union[NotImplementedType, "Fraction"]:  
    """  
    Add this fraction to another fraction.  
  
    :param other: the other number  
    :return: the result of the addition  
  
    >>> print(Fraction(1, 3) + Fraction(1, 2))  
    5/6  
    >>> print(Fraction(1, 2) + Fraction(1, 2))  
    1  
    >>> print(Fraction(21, -12) + Fraction(-33, 42))  
    -71/28  
    """  
    return Fraction((self.a * other.b) + (other.a * self.b),  
                    self.b * other.b) if isinstance(other, Fraction)  
        ↪ \  
    else NotImplemented  
  
def __sub__(self, other) -> Union[NotImplementedType, "Fraction"]:  
    """  
    Subtract this fraction from another fraction.  
  
    :param other: the other fraction  
    :return: the result of the subtraction  
  
    >>> print(Fraction(1, 3) - Fraction(1, 2))  
    -1/6  
    >>> print(Fraction(1, 2) - Fraction(3, 6))  
    0  
    >>> print(Fraction(21, -12) - Fraction(-33, 42))  
    -27/28  
    """  
    return Fraction(  
        (self.a * other.b) - (other.a * self.b), self.b * other.b)\  
        if isinstance(other, Fraction) else NotImplemented
```

Fraction: Addition und Subtraktion

- In der Grundschule haben wir gelernt, dass $\frac{a}{b} + \frac{c}{d} = \frac{a*d+c*b}{b*d}$ ist, für $b, d \neq 0$.
- Wenn `other` auch eine Instanz von `Fraction` ist, dann berechnet `__add__(other)` das Ergebnis genau so und liefert eine neue `Fraction` zurück.
- Der Initialisierer des neuen Bruchs wird diesen automatisch unter Benutzung des ggT normalisieren.
- Wenn `other` keine Instanz von `Fraction` ist, dann geben wir einfach `NotImplemented` zurück.

```
def __add__(self, other) -> Union[NotImplementedType, "Fraction"]:  
    """  
    Add this fraction to another fraction.  
  
    :param other: the other number  
    :return: the result of the addition  
  
    >>> print(Fraction(1, 3) + Fraction(1, 2))  
    5/6  
    >>> print(Fraction(1, 2) + Fraction(1, 2))  
    1  
    >>> print(Fraction(21, -12) + Fraction(-33, 42))  
    -71/28  
    """  
    return Fraction((self.a * other.b) + (other.a * self.b),  
                    self.b * other.b) if isinstance(other, Fraction)  
    else NotImplemented  
  
def __sub__(self, other) -> Union[NotImplementedType, "Fraction"]:  
    """  
    Subtract this fraction from another fraction.  
  
    :param other: the other fraction  
    :return: the result of the subtraction  
  
    >>> print(Fraction(1, 3) - Fraction(1, 2))  
    -1/6  
    >>> print(Fraction(1, 2) - Fraction(3, 6))  
    0  
    >>> print(Fraction(21, -12) - Fraction(-33, 42))  
    -27/28  
    """  
    return Fraction(  
        (self.a * other.b) - (other.a * self.b), self.b * other.b)  
        if isinstance(other, Fraction) else NotImplemented
```

Fraction: Addition und Subtraktion

- Wenn `other` auch eine Instanz von `Fraction` ist, dann berechnet `__add__(other)` das Ergebnis genau so und liefert eine neue `Fraction` zurück.
- Der Initialisierer des neuen Bruchs wird diesen automatisch unter Benutzung des ggT normalisieren.
- Wenn `other` keine Instanz von `Fraction` ist, dann geben wir einfach `NotImplemented` zurück.
- Das kennen wir schon von unserer Implementierung von `__eq__` für Punkte.

```
def __add__(self, other) -> Union[NotImplementedType, "Fraction"]:  
    """  
    Add this fraction to another fraction.  
  
    :param other: the other number  
    :return: the result of the addition  
  
    >>> print(Fraction(1, 3) + Fraction(1, 2))  
    5/6  
    >>> print(Fraction(1, 2) + Fraction(1, 2))  
    1  
    >>> print(Fraction(21, -12) + Fraction(-33, 42))  
    -71/28  
    """  
    return Fraction((self.a * other.b) + (other.a * self.b),  
                    self.b * other.b) if isinstance(other, Fraction)  
        ↪ \  
    else NotImplemented  
  
def __sub__(self, other) -> Union[NotImplementedType, "Fraction"]:  
    """  
    Subtract this fraction from another fraction.  
  
    :param other: the other fraction  
    :return: the result of the subtraction  
  
    >>> print(Fraction(1, 3) - Fraction(1, 2))  
    -1/6  
    >>> print(Fraction(1, 2) - Fraction(3, 6))  
    0  
    >>> print(Fraction(21, -12) - Fraction(-33, 42))  
    -27/28  
    """  
    return Fraction(  
        (self.a * other.b) - (other.a * self.b), self.b * other.b) \  
        if isinstance(other, Fraction) else NotImplemented
```

Fraction: Addition und Subtraktion

- Der Initialisierer des neuen Bruchs wird diesen automatisch unter Benutzung des ggT normalisieren.
- Wenn `other` keine Instanz von `Fraction` ist, dann geben wir einfach `NotImplemented` zurück.
- Das kennen wir schon von unserer Implementierung von `__eq__` für Punkte.
- Das ermöglicht Python, nach anderen möglichen Routen zu suchen, eine Addition hinzubekommen.

```
def __add__(self, other) -> Union[NotImplementedType, "Fraction"]:  
    """  
    Add this fraction to another fraction.  
  
    :param other: the other number  
    :return: the result of the addition  
  
    >>> print(Fraction(1, 3) + Fraction(1, 2))  
    5/6  
    >>> print(Fraction(1, 2) + Fraction(1, 2))  
    1  
    >>> print(Fraction(21, -12) + Fraction(-33, 42))  
    -71/28  
    """  
    return Fraction((self.a * other.b) + (other.a * self.b),  
                    self.b * other.b) if isinstance(other, Fraction)  
        ↪ \  
        else NotImplemented  
  
def __sub__(self, other) -> Union[NotImplementedType, "Fraction"]:  
    """  
    Subtract this fraction from another fraction.  
  
    :param other: the other fraction  
    :return: the result of the subtraction  
  
    >>> print(Fraction(1, 3) - Fraction(1, 2))  
    -1/6  
    >>> print(Fraction(1, 2) - Fraction(3, 6))  
    0  
    >>> print(Fraction(21, -12) - Fraction(-33, 42))  
    -27/28  
    """  
    return Fraction(  
        (self.a * other.b) - (other.a * self.b), self.b * other.b) \  
        if isinstance(other, Fraction) else NotImplemented
```

Fraction: Addition und Subtraktion

- Der Initialisierer des neuen Bruchs wird diesen automatisch unter Benutzung des ggT normalisieren.
- Wenn `other` keine Instanz von `Fraction` ist, dann geben wir einfach `NotImplemented` zurück.
- Das kennen wir schon von unserer Implementierung von `__eq__` für Punkte.
- Das ermöglicht Python, nach anderen möglichen Routen zu suchen, eine Addition hinzubekommen.
- Python würde schauen, ob `other` eine `__radd__`-Methode hat, die nicht `NotImplemented` zurückliefert.

```
def __add__(self, other) -> Union[NotImplementedType, "Fraction"]:  
    """  
    Add this fraction to another fraction.  
  
    :param other: the other number  
    :return: the result of the addition  
  
    >>> print(Fraction(1, 3) + Fraction(1, 2))  
    5/6  
    >>> print(Fraction(1, 2) + Fraction(1, 2))  
    1  
    >>> print(Fraction(21, -12) + Fraction(-33, 42))  
    -71/28  
    """  
    return Fraction((self.a * other.b) + (other.a * self.b),  
                    self.b * other.b) if isinstance(other, Fraction)  
        ↪ \  
        else NotImplemented  
  
def __sub__(self, other) -> Union[NotImplementedType, "Fraction"]:  
    """  
    Subtract this fraction from another fraction.  
  
    :param other: the other fraction  
    :return: the result of the subtraction  
  
    >>> print(Fraction(1, 3) - Fraction(1, 2))  
    -1/6  
    >>> print(Fraction(1, 2) - Fraction(3, 6))  
    0  
    >>> print(Fraction(21, -12) - Fraction(-33, 42))  
    -27/28  
    """  
    return Fraction(  
        (self.a * other.b) - (other.a * self.b), self.b * other.b) \  
        if isinstance(other, Fraction) else NotImplemented
```

Fraction: Addition und Subtraktion

- Wenn `other` keine Instanz von `Fraction` ist, dann geben wir einfach `NotImplemented` zurück.
- Das kennen wir schon von unserer Implementierung von `__eq__` für Punkte.
- Das ermöglicht Python, nach anderen möglichen Routen zu suchen, eine Addition hinzubekommen.
- Python würde schauen, ob `other` eine `__radd__`-Methode hat, die nicht `NotImplemented` zurückliefert.
- Wir wollen hier aber nicht alle möglichen arithmetischen Operationen implementieren, also lassen wir `__radd__` mal aus.

```
def __add__(self, other) -> Union[NotImplementedType, "Fraction"]:  
    """  
    Add this fraction to another fraction.  
  
    :param other: the other number  
    :return: the result of the addition  
  
    >>> print(Fraction(1, 3) + Fraction(1, 2))  
    5/6  
    >>> print(Fraction(1, 2) + Fraction(1, 2))  
    1  
    >>> print(Fraction(21, -12) + Fraction(-33, 42))  
    -71/28  
    """  
    return Fraction((self.a * other.b) + (other.a * self.b),  
                    self.b * other.b) if isinstance(other, Fraction)  
        ↪ \  
        else NotImplemented  
  
def __sub__(self, other) -> Union[NotImplementedType, "Fraction"]:  
    """  
    Subtract this fraction from another fraction.  
  
    :param other: the other fraction  
    :return: the result of the subtraction  
  
    >>> print(Fraction(1, 3) - Fraction(1, 2))  
    -1/6  
    >>> print(Fraction(1, 2) - Fraction(3, 6))  
    0  
    >>> print(Fraction(21, -12) - Fraction(-33, 42))  
    -27/28  
    """  
    return Fraction(  
        (self.a * other.b) - (other.a * self.b), self.b * other.b) \  
        if isinstance(other, Fraction) else NotImplemented
```

Fraction: Addition und Subtraktion

- Das kennen wir schon von unserer Implementierung von `__eq__` für Punkte.
- Das ermöglicht Python, nach anderen möglichen Routen zu suchen, eine Addition hinzubekommen.
- Python würde schauen, ob `other` eine `__radd__`-Methode hat, die nicht `NotImplemented` zurückliefert.
- Wir wollen hier aber nicht alle möglichen arithmetischen Operationen implementieren, also lassen wir `__radd__` mal aus.
- So oder so, wir müssen unsere Methode wieder mit Doctests testen.

```
def __add__(self, other) -> Union[NotImplementedType, "Fraction"]:  
    """  
    Add this fraction to another fraction.  
  
    :param other: the other number  
    :return: the result of the addition  
  
    >>> print(Fraction(1, 3) + Fraction(1, 2))  
    5/6  
    >>> print(Fraction(1, 2) + Fraction(1, 2))  
    1  
    >>> print(Fraction(21, -12) + Fraction(-33, 42))  
    -71/28  
    """  
    return Fraction((self.a * other.b) + (other.a * self.b),  
                    self.b * other.b) if isinstance(other, Fraction)  
        ↪ \  
        else NotImplemented  
  
def __sub__(self, other) -> Union[NotImplementedType, "Fraction"]:  
    """  
    Subtract this fraction from another fraction.  
  
    :param other: the other fraction  
    :return: the result of the subtraction  
  
    >>> print(Fraction(1, 3) - Fraction(1, 2))  
    -1/6  
    >>> print(Fraction(1, 2) - Fraction(3, 6))  
    0  
    >>> print(Fraction(21, -12) - Fraction(-33, 42))  
    -27/28  
    """  
    return Fraction(  
        (self.a * other.b) - (other.a * self.b), self.b * other.b) \  
        if isinstance(other, Fraction) else NotImplemented
```

Fraction: Addition und Subtraktion

- Das ermöglicht Python, nach anderen möglichen Routen zu suchen, eine Addition hinzubekommen.
- Python würde schauen, ob `other` eine `__radd__`-Methode hat, die nicht `NotImplemented` zurückliefert.
- Wir wollen hier aber nicht alle möglichen arithmetischen Operationen implementieren, also lassen wir `__radd__` mal aus.
- So oder so, wir müssen unsere Methode wieder mit Doctests testen.
- Wir prüfen ob $\frac{1}{3} + \frac{1}{2}$ wirklich $\frac{5}{6}$ ergibt.

```
def __add__(self, other) -> Union[NotImplementedType, "Fraction"]:  
    """  
    Add this fraction to another fraction.  
  
    :param other: the other number  
    :return: the result of the addition  
  
    >>> print(Fraction(1, 3) + Fraction(1, 2))  
    5/6  
    >>> print(Fraction(1, 2) + Fraction(1, 2))  
    1  
    >>> print(Fraction(21, -12) + Fraction(-33, 42))  
    -71/28  
    """  
    return Fraction((self.a * other.b) + (other.a * self.b),  
                    self.b * other.b) if isinstance(other, Fraction)  
        ↪ \  
        else NotImplemented  
  
def __sub__(self, other) -> Union[NotImplementedType, "Fraction"]:  
    """  
    Subtract this fraction from another fraction.  
  
    :param other: the other fraction  
    :return: the result of the subtraction  
  
    >>> print(Fraction(1, 3) - Fraction(1, 2))  
    -1/6  
    >>> print(Fraction(1, 2) - Fraction(3, 6))  
    0  
    >>> print(Fraction(21, -12) - Fraction(-33, 42))  
    -27/28  
    """  
    return Fraction(  
        (self.a * other.b) - (other.a * self.b), self.b * other.b) \  
        if isinstance(other, Fraction) else NotImplemented
```

Fraction: Addition und Subtraktion

- Python würde schauen, ob `other` eine `__radd__`-Methode hat, die nicht `NotImplemented` zurückliefert.
- Wir wollen hier aber nicht alle möglichen arithmetischen Operationen implementieren, also lassen wir `__radd__` mal aus.
- So oder so, wir müssen unsere Methode wieder mit Doctests testen.
- Wir prüfen ob $\frac{1}{3} + \frac{1}{2}$ wirklich $\frac{5}{6}$ ergibt.
- Wir prüfen ob $\frac{1}{2} + \frac{1}{2}$ wirklich $\frac{1}{1}$ ergibt.

```
def __add__(self, other) -> Union[NotImplementedType, "Fraction"]:  
    """  
    Add this fraction to another fraction.  
  
    :param other: the other number  
    :return: the result of the addition  
  
>>> print(Fraction(1, 3) + Fraction(1, 2))  
5/6  
>>> print(Fraction(1, 2) + Fraction(1, 2))  
1  
>>> print(Fraction(21, -12) + Fraction(-33, 42))  
-71/28  
    """  
    return Fraction((self.a * other.b) + (other.a * self.b),  
                    self.b * other.b) if isinstance(other, Fraction)  
        ↪ \  
    else NotImplemented  
  
def __sub__(self, other) -> Union[NotImplementedType, "Fraction"]:  
    """  
    Subtract this fraction from another fraction.  
  
    :param other: the other fraction  
    :return: the result of the subtraction  
  
>>> print(Fraction(1, 3) - Fraction(1, 2))  
-1/6  
>>> print(Fraction(1, 2) - Fraction(3, 6))  
0  
>>> print(Fraction(21, -12) - Fraction(-33, 42))  
-27/28  
    """  
    return Fraction(  
        (self.a * other.b) - (other.a * self.b), self.b * other.b) \  
        if isinstance(other, Fraction) else NotImplemented
```

Fraction: Addition und Subtraktion

- Wir wollen hier aber nicht alle möglichen arithmetischen Operationen implementieren, also lassen wir `__radd__` mal aus.
- So oder so, wir müssen unsere Methode wieder mit Doctests testen.

- Wir prüfen ob $\frac{1}{3} + \frac{1}{2}$ wirklich $\frac{5}{6}$ ergibt.

- Wir prüfen ob $\frac{1}{2} + \frac{1}{2}$ wirklich $\frac{1}{1}$ ergibt.

- Wir prüfen auch auf korrekte Normalisierung, in dem wir gucken, ob

$$\frac{21}{-12} + \frac{-33}{42} = \frac{882+396}{-504} = \frac{1278}{-504} = \frac{18 \cdot 1278}{18 \cdot -28} = \frac{-71}{28} \text{ stimmt.}$$

```
def __add__(self, other) -> Union[NotImplementedType, "Fraction"]:  
    """  
    Add this fraction to another fraction.  
  
    :param other: the other number  
    :return: the result of the addition  
  
    >>> print(Fraction(1, 3) + Fraction(1, 2))  
    5/6  
    >>> print(Fraction(1, 2) + Fraction(1, 2))  
    1  
    >>> print(Fraction(21, -12) + Fraction(-33, 42))  
    -71/28  
    """  
    return Fraction((self.a * other.b) + (other.a * self.b),  
                    self.b * other.b) if isinstance(other, Fraction)  
                    ↪ \n  
                    else NotImplemented  
  
def __sub__(self, other) -> Union[NotImplementedType, "Fraction"]:  
    """  
    Subtract this fraction from another fraction.  
  
    :param other: the other fraction  
    :return: the result of the subtraction  
  
    >>> print(Fraction(1, 3) - Fraction(1, 2))  
    -1/6  
    >>> print(Fraction(1, 2) - Fraction(3, 6))  
    0  
    >>> print(Fraction(21, -12) - Fraction(-33, 42))  
    -27/28  
    """  
    return Fraction(  
        (self.a * other.b) - (other.a * self.b), self.b * other.b) \n  
        if isinstance(other, Fraction) else NotImplemented
```

Fraction: Addition und Subtraktion

- So oder so, wir müssen unsere Methode wieder mit Doctests testen.
- Wir prüfen ob $\frac{1}{3} + \frac{1}{2}$ wirklich $\frac{5}{6}$ ergibt.
- Wir prüfen ob $\frac{1}{2} + \frac{1}{2}$ wirklich $\frac{1}{1}$ ergibt.
- Wir prüfen auch auf korrekte Normalisierung, in dem wir gucken, ob $\frac{21}{-12} + \frac{-33}{42} = \frac{882+396}{-504} = \frac{1278}{-504} = \frac{18*1278}{18*-504} = \frac{-71}{28}$ stimmt.
- Nachdem wir bestätigt haben, dass diese Tests erfolgreich sind, wollen wir nun die `__sub__`-Methode ganz genauso implementieren.

```
def __add__(self, other) -> Union[NotImplementedType, "Fraction"]:  
    """  
    Add this fraction to another fraction.  
  
    :param other: the other number  
    :return: the result of the addition  
  
    >>> print(Fraction(1, 3) + Fraction(1, 2))  
    5/6  
    >>> print(Fraction(1, 2) + Fraction(1, 2))  
    1  
    >>> print(Fraction(21, -12) + Fraction(-33, 42))  
    -71/28  
    """  
    return Fraction((self.a * other.b) + (other.a * self.b),  
                    self.b * other.b) if isinstance(other, Fraction)  
                    ↪ \n  
                    else NotImplemented  
  
def __sub__(self, other) -> Union[NotImplementedType, "Fraction"]:  
    """  
    Subtract this fraction from another fraction.  
  
    :param other: the other fraction  
    :return: the result of the subtraction  
  
    >>> print(Fraction(1, 3) - Fraction(1, 2))  
    -1/6  
    >>> print(Fraction(1, 2) - Fraction(3, 6))  
    0  
    >>> print(Fraction(21, -12) - Fraction(-33, 42))  
    -27/28  
    """  
    return Fraction(  
        (self.a * other.b) - (other.a * self.b), self.b * other.b)\n        if isinstance(other, Fraction) else NotImplemented
```

Fraction: Addition und Subtraktion

- Wir prüfen ob $\frac{1}{3} + \frac{1}{2}$ wirklich $\frac{5}{6}$ ergibt.
- Wir prüfen ob $\frac{1}{2} + \frac{1}{2}$ wirklich $\frac{1}{1}$ ergibt.
- Wir prüfen auch auf korrekte Normalisierung, in dem wir gucken, ob $\frac{21}{-12} + \frac{-33}{42} = \frac{882+396}{-504} = \frac{1278}{-504} = \frac{18*1278}{18*-504} = \frac{-71}{28}$ stimmt.
- Nachdem wir bestätigt haben, dass diese Tests erfolgreich sind, wollen wir nun die `__sub__`-Methode ganz genauso implementieren.
- Das erlaubt nämlich Subtraktion mit `-`, denn `x - y` ruft `x.__sub__(y)` auf, wenn die Klasse von `x` die `__sub__`-Methode definiert.

```
def __add__(self, other) -> Union[NotImplementedType, "Fraction"]:  
    """  
    Add this fraction to another fraction.  
  
    :param other: the other number  
    :return: the result of the addition  
  
>>> print(Fraction(1, 3) + Fraction(1, 2))  
5/6  
>>> print(Fraction(1, 2) + Fraction(1, 2))  
1  
>>> print(Fraction(21, -12) + Fraction(-33, 42))  
-71/28  
    """  
    return Fraction((self.a * other.b) + (other.a * self.b),  
                    self.b * other.b) if isinstance(other, Fraction)  
                    ↪ \n  
                    else NotImplemented  
  
def __sub__(self, other) -> Union[NotImplementedType, "Fraction"]:  
    """  
    Subtract this fraction from another fraction.  
  
    :param other: the other fraction  
    :return: the result of the subtraction  
  
>>> print(Fraction(1, 3) - Fraction(1, 2))  
-1/6  
>>> print(Fraction(1, 2) - Fraction(3, 6))  
0  
>>> print(Fraction(21, -12) - Fraction(-33, 42))  
-27/28  
    """  
    return Fraction(  
        (self.a * other.b) - (other.a * self.b), self.b * other.b)\n        if isinstance(other, Fraction) else NotImplemented
```

Fraction: Addition und Subtraktion

- Wir prüfen ob $\frac{1}{2} + \frac{1}{2}$ wirklich $\frac{1}{1}$ ergibt.
- Wir prüfen auch auf korrekte Normalisierung, in dem wir gucken, ob $\frac{21}{-12} + \frac{-33}{42} = \frac{882+396}{-504} = \frac{1278}{-504} = \frac{18*1278}{18*-504} = \frac{-71}{28}$ stimmt.
- Nachdem wir bestätigt haben, dass diese Tests erfolgreich sind, wollen wir nun die `__sub__`-Methode ganz genauso implementieren.
- Das erlaubt nämlich Subtraktion mit `-`, denn `x - y` ruft `x.__sub__(y)` auf, wenn die Klasse von `x` die `__sub__`-Methode definiert.
- Es ist klar, dass $\frac{a}{b} - \frac{c}{d} = \frac{a*d-c*b}{b*d}$ für $b, d \neq 0$.

```
def __add__(self, other) -> Union[NotImplementedType, "Fraction"]:  
    """  
    Add this fraction to another fraction.  
  
    :param other: the other number  
    :return: the result of the addition  
  
    >>> print(Fraction(1, 3) + Fraction(1, 2))  
    5/6  
    >>> print(Fraction(1, 2) + Fraction(1, 2))  
    1  
    >>> print(Fraction(21, -12) + Fraction(-33, 42))  
    -71/28  
    """  
    return Fraction((self.a * other.b) + (other.a * self.b),  
                    self.b * other.b) if isinstance(other, Fraction)  
                    else NotImplemented  
  
def __sub__(self, other) -> Union[NotImplementedType, "Fraction"]:  
    """  
    Subtract this fraction from another fraction.  
  
    :param other: the other fraction  
    :return: the result of the subtraction  
  
    >>> print(Fraction(1, 3) - Fraction(1, 2))  
    -1/6  
    >>> print(Fraction(1, 2) - Fraction(3, 6))  
    0  
    >>> print(Fraction(21, -12) - Fraction(-33, 42))  
    -27/28  
    """  
    return Fraction(  
        (self.a * other.b) - (other.a * self.b), self.b * other.b)  
        if isinstance(other, Fraction) else NotImplemented
```

Fraction: Addition und Subtraktion

- Nachdem wir bestätigt haben, dass diese Tests erfolgreich sind, wollen wir nun die `__sub__`-Methode ganz genauso implementieren.
- Das erlaubt nämlich Subtraktion mit `-`, denn `x - y` ruft `x.__sub__(y)` auf, wenn die Klasse von `x` die `__sub__`-Methode definiert.
- Es ist klar, dass $\frac{a}{b} - \frac{c}{d} = \frac{a*d-c*b}{b*d}$ für $b, d \neq 0$.
- Als Doctests nehmen wir dann die selben drei Fälle wie für die `__add__`-Methode.

```
def __add__(self, other) -> Union[NotImplementedType, "Fraction"]:  
    """  
    Add this fraction to another fraction.  
  
    :param other: the other number  
    :return: the result of the addition  
  
    >>> print(Fraction(1, 3) + Fraction(1, 2))  
    5/6  
    >>> print(Fraction(1, 2) + Fraction(1, 2))  
    1  
    >>> print(Fraction(21, -12) + Fraction(-33, 42))  
    -71/28  
    """  
    return Fraction((self.a * other.b) + (other.a * self.b),  
                    self.b * other.b) if isinstance(other, Fraction)  
        ↪ \n  
        else NotImplemented  
  
def __sub__(self, other) -> Union[NotImplementedType, "Fraction"]:  
    """  
    Subtract this fraction from another fraction.  
  
    :param other: the other fraction  
    :return: the result of the subtraction  
  
    >>> print(Fraction(1, 3) - Fraction(1, 2))  
    -1/6  
    >>> print(Fraction(1, 2) - Fraction(3, 6))  
    0  
    >>> print(Fraction(21, -12) - Fraction(-33, 42))  
    -27/28  
    """  
    return Fraction(  
        (self.a * other.b) - (other.a * self.b), self.b * other.b)\n        if isinstance(other, Fraction) else NotImplemented
```

Fraction: Multiplikation, Division,

- Schauen wir uns nun Multiplikation und Division an.

```
1 def __mul__(self, other) -> Union[NotImplementedType, "Fraction"]:  
2     """  
3     Multiply this fraction with another fraction.  
4  
5     :param other: the other fraction  
6     :return: the result of the multiplication  
7  
8     >>> print(Fraction(6, 19) * Fraction(3, -7))  
9     -18/133  
10    """  
11    return Fraction(self.a * other.a, self.b * other.b) \  
12        if isinstance(other, Fraction) else NotImplemented  
13  
14 def __truediv__(self, other) -> Union[NotImplementedType, "Fraction"]:  
15     ↪ """  
16     Divide this fraction by another fraction.  
17  
18     :param other: the other fraction  
19     :return: the result of the division  
20  
21     >>> print(Fraction(6, 19) / Fraction(3, -7))  
22     -14/19  
23     """  
24    return Fraction(self.a * other.b, self.b * other.a) \  
25        if isinstance(other, Fraction) else NotImplemented  
26  
27 def __abs__(self) -> "Fraction":  
28     """  
29     Get the absolute value of this fraction.  
30  
31     :return: the absolute value.  
32  
33     >>> print(abs(Fraction(-1, 2)))  
34     1/2  
35     >>> print(abs(Fraction(3, 5)))  
36     3/5  
37     """  
38    return self if self.a > 0 else Fraction(-self.a, self.b)
```

Fraction: Multiplikation, Division,

- Schauen wir uns nun Multiplikation und Division an.
- Der `*` Operator benutzt die Methode `__mul__`, wenn diese implementiert ist.

```
1 def __mul__(self, other) -> Union[NotImplementedType, "Fraction"]:  
2     """  
3     Multiply this fraction with another fraction.  
4  
5     :param other: the other fraction  
6     :return: the result of the multiplication  
7  
8     >>> print(Fraction(6, 19) * Fraction(3, -7))  
9     -18/133  
10    """  
11    return Fraction(self.a * other.a, self.b * other.b) \  
12           if isinstance(other, Fraction) else NotImplemented  
13  
14 def __truediv__(self, other) -> Union[NotImplementedType, "Fraction"]:  
15     ↪ """  
16     Divide this fraction by another fraction.  
17  
18     :param other: the other fraction  
19     :return: the result of the division  
20  
21     >>> print(Fraction(6, 19) / Fraction(3, -7))  
22     -14/19  
23     """  
24    return Fraction(self.a * other.b, self.b * other.a) \  
25           if isinstance(other, Fraction) else NotImplemented  
26  
27 def __abs__(self) -> "Fraction":  
28     """  
29     Get the absolute value of this fraction.  
30  
31     :return: the absolute value.  
32  
33     >>> print(abs(Fraction(-1, 2)))  
34     1/2  
35     >>> print(abs(Fraction(3, 5)))  
36     3/5  
37     """  
38    return self if self.a > 0 else Fraction(-self.a, self.b)
```

Fraction: Multiplikation, Division,

- Schauen wir uns nun Multiplikation und Division an.
- Der `*` Operator benutzt die Methode `__mul__`, wenn diese implementiert ist.
- Der `/` Operator benutzt die Methode `__truediv__`, wenn diese implementiert ist.

```
def __mul__(self, other) -> Union[NotImplementedType, "Fraction"]:  
    """  
    Multiply this fraction with another fraction.  
  
    :param other: the other fraction  
    :return: the result of the multiplication  
  
    >>> print(Fraction(6, 19) * Fraction(3, -7))  
    -18/133  
    """  
    return Fraction(self.a * other.a, self.b * other.b) \  
        if isinstance(other, Fraction) else NotImplemented  
  
def __truediv__(self, other) -> Union[NotImplementedType, "Fraction"]:  
    """  
    Divide this fraction by another fraction.  
  
    :param other: the other fraction  
    :return: the result of the division  
  
    >>> print(Fraction(6, 19) / Fraction(3, -7))  
    -14/19  
    """  
    return Fraction(self.a * other.b, self.b * other.a) \  
        if isinstance(other, Fraction) else NotImplemented  
  
def __abs__(self) -> "Fraction":  
    """  
    Get the absolute value of this fraction.  
  
    :return: the absolute value.  
  
    >>> print(abs(Fraction(-1, 2)))  
    1/2  
    >>> print(abs(Fraction(3, 5)))  
    3/5  
    """  
    return self if self.a > 0 else Fraction(-self.a, self.b)
```

Fraction: Multiplikation, Division,

- Schauen wir uns nun Multiplikation und Division an.
- Der `*` Operator benutzt die Methode `__mul__`, wenn diese implementiert ist.
- Der `/` Operator benutzt die Methode `__truediv__`, wenn diese implementiert ist.
- Wenn wir die Brüche $\frac{a}{b}$ und $\frac{c}{d}$ multiplizieren, dann bekommen wir $\frac{a*c}{b*d}$ für $b, d \neq 0$.

```
def __mul__(self, other) -> Union[NotImplementedType, "Fraction"]:  
    """  
    Multiply this fraction with another fraction.  
  
    :param other: the other fraction  
    :return: the result of the multiplication  
  
    >>> print(Fraction(6, 19) * Fraction(3, -7))  
    -18/133  
    """  
    return Fraction(self.a * other.a, self.b * other.b) \  
        if isinstance(other, Fraction) else NotImplemented  
  
def __truediv__(self, other) -> Union[NotImplementedType, "Fraction"]:  
    """  
    Divide this fraction by another fraction.  
  
    :param other: the other fraction  
    :return: the result of the division  
  
    >>> print(Fraction(6, 19) / Fraction(3, -7))  
    -14/19  
    """  
    return Fraction(self.a * other.b, self.b * other.a) \  
        if isinstance(other, Fraction) else NotImplemented  
  
def __abs__(self) -> "Fraction":  
    """  
    Get the absolute value of this fraction.  
  
    :return: the absolute value.  
  
    >>> print(abs(Fraction(-1, 2)))  
    1/2  
    >>> print(abs(Fraction(3, 5)))  
    3/5  
    """  
    return self if self.a > 0 else Fraction(-self.a, self.b)
```

Fraction: Multiplikation, Division,

- Schauen wir uns nun Multiplikation und Division an.
- Der `*` Operator benutzt die Methode `__mul__`, wenn diese implementiert ist.
- Der `/` Operator benutzt die Methode `__truediv__`, wenn diese implementiert ist.
- Wenn wir die Brüche $\frac{a}{b}$ und $\frac{c}{d}$ multiplizieren, dann bekommen wir $\frac{a*c}{b*d}$ für $b, d \neq 0$.
- Das Teilen von $\frac{a}{b}$ durch $\frac{c}{d}$ ergibt $\frac{a*d}{b*c}$ für $b, c, d \neq 0$.

```
def __mul__(self, other) -> Union[NotImplementedType, "Fraction"]:  
    """  
    Multiply this fraction with another fraction.  
  
    :param other: the other fraction  
    :return: the result of the multiplication  
  
    >>> print(Fraction(6, 19) * Fraction(3, -7))  
    -18/133  
    """  
    return Fraction(self.a * other.a, self.b * other.b) \  
        if isinstance(other, Fraction) else NotImplemented  
  
def __truediv__(self, other) -> Union[NotImplementedType, "Fraction"]:  
    """  
    Divide this fraction by another fraction.  
  
    :param other: the other fraction  
    :return: the result of the division  
  
    >>> print(Fraction(6, 19) / Fraction(3, -7))  
    -14/19  
    """  
    return Fraction(self.a * other.b, self.b * other.a) \  
        if isinstance(other, Fraction) else NotImplemented  
  
def __abs__(self) -> "Fraction":  
    """  
    Get the absolute value of this fraction.  
  
    :return: the absolute value.  
  
    >>> print(abs(Fraction(-1, 2)))  
    1/2  
    >>> print(abs(Fraction(3, 5)))  
    3/5  
    """  
    return self if self.a > 0 else Fraction(-self.a, self.b)
```

Fraction: Multiplikation, Division,

- Der `*` Operator benutzt die Methode `__mul__`, wenn diese implementiert ist.
- Der `/` Operator benutzt die Methode `__truediv__`, wenn diese implementiert ist.
- Wenn wir die Brüche $\frac{a}{b}$ und $\frac{c}{d}$ multiplizieren, dann bekommen wir $\frac{a*c}{b*d}$ für $b, d \neq 0$.
- Das Teilen von $\frac{a}{b}$ durch $\frac{c}{d}$ ergibt $\frac{a*d}{b*c}$ für $b, c, d \neq 0$.
- Die Dunder-Methoden können nach dem selben Schema wie vorher implementiert werden.

```
def __mul__(self, other) -> Union[NotImplementedType, "Fraction"]:  
    """  
    Multiply this fraction with another fraction.  
  
    :param other: the other fraction  
    :return: the result of the multiplication  
  
    >>> print(Fraction(6, 19) * Fraction(3, -7))  
    -18/133  
    """  
    return Fraction(self.a * other.a, self.b * other.b) \  
        if isinstance(other, Fraction) else NotImplemented  
  
def __truediv__(self, other) -> Union[NotImplementedType, "Fraction"]:  
    """  
    Divide this fraction by another fraction.  
  
    :param other: the other fraction  
    :return: the result of the division  
  
    >>> print(Fraction(6, 19) / Fraction(3, -7))  
    -14/19  
    """  
    return Fraction(self.a * other.b, self.b * other.a) \  
        if isinstance(other, Fraction) else NotImplemented  
  
def __abs__(self) -> "Fraction":  
    """  
    Get the absolute value of this fraction.  
  
    :return: the absolute value.  
  
    >>> print(abs(Fraction(-1, 2)))  
    1/2  
    >>> print(abs(Fraction(3, 5)))  
    3/5  
    """  
    return self if self.a > 0 else Fraction(-self.a, self.b)
```

Fraction: Multiplikation, Division,

- Der `/` Operator benutzt die Methode `__truediv__`, wenn diese implementiert ist.
- Wenn wir die Brüche $\frac{a}{b}$ und $\frac{c}{d}$ multiplizieren, dann bekommen wir $\frac{a*c}{b*d}$ für $b, d \neq 0$.
- Das Teilen von $\frac{a}{b}$ durch $\frac{c}{d}$ ergibt $\frac{a*d}{b*c}$ für $b, c, d \neq 0$.
- Die Dunder-Methoden können nach dem selben Schema wie vorher implementiert werden.
- Wir testen die Multiplikation, in dem wir bestätigen das
$$\frac{6}{19} * \frac{3}{-7} = \frac{6*3}{19*-7} = \frac{18}{-133} = \frac{-18}{133}.$$

```
def __mul__(self, other) -> Union[NotImplementedType, "Fraction"]:  
    """  
    Multiply this fraction with another fraction.  
  
    :param other: the other fraction  
    :return: the result of the multiplication  
  
    >>> print(Fraction(6, 19) * Fraction(3, -7))  
    -18/133  
    """  
    return Fraction(self.a * other.a, self.b * other.b) \  
        if isinstance(other, Fraction) else NotImplemented  
  
def __truediv__(self, other) -> Union[NotImplementedType, "Fraction"]:  
    """  
    Divide this fraction by another fraction.  
  
    :param other: the other fraction  
    :return: the result of the division  
  
    >>> print(Fraction(6, 19) / Fraction(3, -7))  
    -14/19  
    """  
    return Fraction(self.a * other.b, self.b * other.a) \  
        if isinstance(other, Fraction) else NotImplemented  
  
def __abs__(self) -> "Fraction":  
    """  
    Get the absolute value of this fraction.  
  
    :return: the absolute value.  
  
    >>> print(abs(Fraction(-1, 2)))  
    1/2  
    >>> print(abs(Fraction(3, 5)))  
    3/5  
    """  
    return self if self.a > 0 else Fraction(-self.a, self.b)
```

Fraction: Multiplikation, Division,

- Wenn wir die Brüche $\frac{a}{b}$ und $\frac{c}{d}$ multiplizieren, dann bekommen wir $\frac{a*c}{b*d}$ für $b, d \neq 0$.
- Das Teilen von $\frac{a}{b}$ durch $\frac{c}{d}$ ergibt $\frac{a*d}{b*c}$ für $b, c, d \neq 0$.
- Die Dunder-Methoden können nach dem selben Schema wie vorher implementiert werden.
- Wir testen die Multiplikation, in dem wir bestätigen das
$$\frac{6}{19} * \frac{3}{-7} = \frac{6*3}{19*-7} = \frac{18}{-133} = \frac{-18}{133}.$$
- Wir testen die Division, in dem wir bestätigen, dass
$$\frac{6}{19} * \frac{3}{-7} = \frac{6*-7}{19*3} = \frac{-42}{57} = \frac{3*-14}{3*19}$$
wirklich $\frac{-14}{19}$ ergibt.

```
def __mul__(self, other) -> Union[NotImplementedType, "Fraction"]:  
    """  
    Multiply this fraction with another fraction.  
  
    :param other: the other fraction  
    :return: the result of the multiplication  
  
    >>> print(Fraction(6, 19) * Fraction(3, -7))  
    -18/133  
    """  
    return Fraction(self.a * other.a, self.b * other.b) \  
        if isinstance(other, Fraction) else NotImplemented  
  
def __truediv__(self, other) -> Union[NotImplementedType, "Fraction"]:  
    """  
    Divide this fraction by another fraction.  
  
    :param other: the other fraction  
    :return: the result of the division  
  
    >>> print(Fraction(6, 19) / Fraction(3, -7))  
    -14/19  
    """  
    return Fraction(self.a * other.b, self.b * other.a) \  
        if isinstance(other, Fraction) else NotImplemented  
  
def __abs__(self) -> "Fraction":  
    """  
    Get the absolute value of this fraction.  
  
    :return: the absolute value.  
  
    >>> print(abs(Fraction(-1, 2)))  
    1/2  
    >>> print(abs(Fraction(3, 5)))  
    3/5  
    """  
    return self if self.a > 0 else Fraction(-self.a, self.b)
```

Fraction: Multiplikation, Division,

- Das Teilen von $\frac{a}{b}$ durch $\frac{c}{d}$ ergibt $\frac{a*d}{b*c}$ für $b, c, d \neq 0$.
- Die Dunder-Methoden können nach dem selben Schema wie vorher implementiert werden.
- Wir testen die Multiplikation, in dem wir bestätigen das
$$\frac{6}{19} * \frac{3}{-7} = \frac{6*3}{19*-7} = \frac{18}{-133} = \frac{-18}{133}.$$
- Wir testen die Division, in dem wir bestätigen, dass
$$\frac{6}{19} * \frac{3}{-7} = \frac{6*-7}{19*3} = \frac{-42}{57} = \frac{3*-14}{3*19}$$
wirklich $\frac{-14}{19}$ ergibt.
- Wir implementieren auch eine Unterstützung für die `abs`-Function.

```
def __mul__(self, other) -> Union[NotImplementedType, "Fraction"]:  
    """  
    Multiply this fraction with another fraction.  
  
    :param other: the other fraction  
    :return: the result of the multiplication  
  
    >>> print(Fraction(6, 19) * Fraction(3, -7))  
    -18/133  
    """  
    return Fraction(self.a * other.a, self.b * other.b) \  
        if isinstance(other, Fraction) else NotImplemented  
  
def __truediv__(self, other) -> Union[NotImplementedType, "Fraction"]:  
    """  
    Divide this fraction by another fraction.  
  
    :param other: the other fraction  
    :return: the result of the division  
  
    >>> print(Fraction(6, 19) / Fraction(3, -7))  
    -14/19  
    """  
    return Fraction(self.a * other.b, self.b * other.a) \  
        if isinstance(other, Fraction) else NotImplemented  
  
def __abs__(self) -> "Fraction":  
    """  
    Get the absolute value of this fraction.  
  
    :return: the absolute value.  
  
    >>> print(abs(Fraction(-1, 2)))  
    1/2  
    >>> print(abs(Fraction(3, 5)))  
    3/5  
    """  
    return self if self.a > 0 else Fraction(-self.a, self.b)
```

Fraction: Multiplikation, Division,

- Das Teilen von $\frac{a}{b}$ durch $\frac{c}{d}$ ergibt $\frac{a*d}{b*c}$ für $b, c, d \neq 0$.
- Die Dunder-Methoden können nach dem selben Schema wie vorher implementiert werden.
- Wir testen die Multiplikation, in dem wir bestätigen das
$$\frac{6}{19} * \frac{3}{-7} = \frac{6*3}{19*-7} = \frac{18}{-133} = \frac{-18}{133}.$$
- Wir testen die Division, in dem wir bestätigen, dass
$$\frac{6}{19} * \frac{3}{-7} = \frac{6*-7}{19*3} = \frac{-42}{57} = \frac{3*-14}{3*19}$$
wirklich $\frac{-14}{19}$ ergibt.
- Wir implementieren auch eine Unterstützung für die `abs`-Function.
- `abs` liefert den Betrag einer Zahl.

```
def __mul__(self, other) -> Union[NotImplementedType, "Fraction"]:  
    """  
    Multiply this fraction with another fraction.  
  
    :param other: the other fraction  
    :return: the result of the multiplication  
  
    >>> print(Fraction(6, 19) * Fraction(3, -7))  
    -18/133  
    """  
    return Fraction(self.a * other.a, self.b * other.b) \  
        if isinstance(other, Fraction) else NotImplemented  
  
def __truediv__(self, other) -> Union[NotImplementedType, "Fraction"]:  
    """  
    Divide this fraction by another fraction.  
  
    :param other: the other fraction  
    :return: the result of the division  
  
    >>> print(Fraction(6, 19) / Fraction(3, -7))  
    -14/19  
    """  
    return Fraction(self.a * other.b, self.b * other.a) \  
        if isinstance(other, Fraction) else NotImplemented  
  
def __abs__(self) -> "Fraction":  
    """  
    Get the absolute value of this fraction.  
  
    :return: the absolute value.  
  
    >>> print(abs(Fraction(-1, 2)))  
    1/2  
    >>> print(abs(Fraction(3, 5)))  
    3/5  
    """  
    return self if self.a > 0 else Fraction(-self.a, self.b)
```

Fraction: Multiplikation, Division,

- Wir testen die Multiplikation, in dem wir bestätigen das

$$\frac{6}{19} * \frac{3}{-7} = \frac{6*3}{19*-7} = \frac{18}{-133} = \frac{-18}{133}.$$

- Wir testen die Division, in dem wir bestätigen, dass

$$\frac{6}{19} * \frac{3}{-7} = \frac{6*-7}{19*3} = \frac{-42}{57} = \frac{3*-14}{3*19}$$

wirklich $\frac{-14}{19}$ ergibt.

- Wir implementieren auch eine Unterstützung für die `abs`-Function.
- `abs` liefert den Betrag einer Zahl.
- Es gilt das `abs(5) = abs(-5) = 5`.

```
1 def __mul__(self, other) -> Union[NotImplementedType, "Fraction"]:  
2     """  
3     Multiply this fraction with another fraction.  
4  
5     :param other: the other fraction  
6     :return: the result of the multiplication  
7  
8     >>> print(Fraction(6, 19) * Fraction(3, -7))  
9     -18/133  
10    """  
11    return Fraction(self.a * other.a, self.b * other.b) \  
12           if isinstance(other, Fraction) else NotImplemented  
13  
14 def __truediv__(self, other) -> Union[NotImplementedType, "Fraction"]:  
15     ↩ """  
16     Divide this fraction by another fraction.  
17  
18     :param other: the other fraction  
19     :return: the result of the division  
20  
21     >>> print(Fraction(6, 19) / Fraction(3, -7))  
22     -14/19  
23     """  
24    return Fraction(self.a * other.b, self.b * other.a) \  
25           if isinstance(other, Fraction) else NotImplemented  
26  
27 def __abs__(self) -> "Fraction":  
28     """  
29     Get the absolute value of this fraction.  
30  
31     :return: the absolute value.  
32  
33     >>> print(abs(Fraction(-1, 2)))  
34     1/2  
35     >>> print(abs(Fraction(3, 5)))  
36     3/5  
37     """  
38    return self if self.a > 0 else Fraction(-self.a, self.b)
```

Fraction: Multiplikation, Division,

- Wir testen die Division, in dem wir bestätigen, dass
$$\frac{6}{19} * \frac{3}{-7} = \frac{6*-7}{19*3} = \frac{-42}{57} = \frac{3*-14}{3*19}$$
wirklich $\frac{-14}{19}$ ergibt.
- Wir implementieren auch eine Unterstützung für die `abs`-Function.
- `abs` liefert den Betrag einer Zahl.
- Es gilt das `abs(5) = abs(-5) = 5`.
- `abs(x)` ruft `x.__abs__()` auf, wenn diese Methode definiert ist.

```
def __mul__(self, other) -> Union[NotImplementedType, "Fraction"]:  
    """  
    Multiply this fraction with another fraction.  
  
    :param other: the other fraction  
    :return: the result of the multiplication  
  
    >>> print(Fraction(6, 19) * Fraction(3, -7))  
    -18/133  
    """  
    return Fraction(self.a * other.a, self.b * other.b) \  
        if isinstance(other, Fraction) else NotImplemented  
  
def __truediv__(self, other) -> Union[NotImplementedType, "Fraction"]:  
    """  
    Divide this fraction by another fraction.  
  
    :param other: the other fraction  
    :return: the result of the division  
  
    >>> print(Fraction(6, 19) / Fraction(3, -7))  
    -14/19  
    """  
    return Fraction(self.a * other.b, self.b * other.a) \  
        if isinstance(other, Fraction) else NotImplemented  
  
def __abs__(self) -> "Fraction":  
    """  
    Get the absolute value of this fraction.  
  
    :return: the absolute value.  
  
    >>> print(abs(Fraction(-1, 2)))  
    1/2  
    >>> print(abs(Fraction(3, 5)))  
    3/5  
    """  
    return self if self.a > 0 else Fraction(-self.a, self.b)
```

Fraction: Multiplikation, Division,

- Wir implementieren auch eine Unterstützung für die `abs`-Funktion.
- `abs` liefert den Betrag einer Zahl.
- Es gilt das `abs(5) = abs(-5) = 5`.
- `abs(x)` ruft `x.__abs__()` auf, wenn diese Methode definiert ist.
- Wir implementieren diese Methode so:

```
1 def __mul__(self, other) -> Union[NotImplementedType, "Fraction"]:  
2     """  
3     Multiply this fraction with another fraction.  
4  
5     :param other: the other fraction  
6     :return: the result of the multiplication  
7  
8     >>> print(Fraction(6, 19) * Fraction(3, -7))  
9     -18/133  
10    """  
11    return Fraction(self.a * other.a, self.b * other.b) \  
12        if isinstance(other, Fraction) else NotImplemented  
13  
14 def __truediv__(self, other) -> Union[NotImplementedType, "Fraction"]:  
15     ↩ """  
16     Divide this fraction by another fraction.  
17  
18     :param other: the other fraction  
19     :return: the result of the division  
20  
21     >>> print(Fraction(6, 19) / Fraction(3, -7))  
22     -14/19  
23    """  
24    return Fraction(self.a * other.b, self.b * other.a) \  
25        if isinstance(other, Fraction) else NotImplemented  
26  
27 def __abs__(self) -> "Fraction":  
28     """  
29     Get the absolute value of this fraction.  
30  
31     :return: the absolute value.  
32  
33     >>> print(abs(Fraction(-1, 2)))  
34     1/2  
35     >>> print(abs(Fraction(3, 5)))  
36     3/5  
37    """  
38    return self if self.a > 0 else Fraction(-self.a, self.b)
```

Fraction: Multiplikation, Division,

- `abs` liefert den Betrag einer Zahl.
- Es gilt das `abs(5) = abs(-5) = 5`.
- `abs(x)` ruft `x.__abs__()` auf, wenn diese Methode definiert ist.
- Wir implementieren diese Methode so:
- Wenn unser Bruch positiv ist, dann liefern wir ihn direkt zurück.

```
1 def __mul__(self, other) -> Union[NotImplementedType, "Fraction"]:  
2     """  
3     Multiply this fraction with another fraction.  
4  
5     :param other: the other fraction  
6     :return: the result of the multiplication  
7  
8     >>> print(Fraction(6, 19) * Fraction(3, -7))  
9     -18/133  
10    """  
11    return Fraction(self.a * other.a, self.b * other.b) \  
12           if isinstance(other, Fraction) else NotImplemented  
13  
14 def __truediv__(self, other) -> Union[NotImplementedType, "Fraction"]:  
15     ↩ """  
16     Divide this fraction by another fraction.  
17  
18     :param other: the other fraction  
19     :return: the result of the division  
20  
21     >>> print(Fraction(6, 19) / Fraction(3, -7))  
22     -14/19  
23    """  
24    return Fraction(self.a * other.b, self.b * other.a) \  
25           if isinstance(other, Fraction) else NotImplemented  
26  
27 def __abs__(self) -> "Fraction":  
28     """  
29     Get the absolute value of this fraction.  
30  
31     :return: the absolute value.  
32  
33     >>> print(abs(Fraction(-1, 2)))  
34     1/2  
35     >>> print(abs(Fraction(3, 5)))  
36     3/5  
37    """  
38    return self if self.a > 0 else Fraction(-self.a, self.b)
```

Fraction: Multiplikation, Division,

- `abs` liefert den Betrag einer Zahl.
- Es gilt das `abs(5) = abs(-5) = 5`.
- `abs(x)` ruft `x.__abs__()` auf, wenn diese Methode definiert ist.
- Wir implementieren diese Methode so:
- Wenn unser Bruch positiv ist, dann liefern wir ihn direkt zurück.
- Sonst erstellen wir eine neue, positive Variante unseres Bruchs.

```
1 def __mul__(self, other) -> Union[NotImplementedType, "Fraction"]:  
2     """  
3     Multiply this fraction with another fraction.  
4  
5     :param other: the other fraction  
6     :return: the result of the multiplication  
7  
8     >>> print(Fraction(6, 19) * Fraction(3, -7))  
9     -18/133  
10    """  
11    return Fraction(self.a * other.a, self.b * other.b) \  
12           if isinstance(other, Fraction) else NotImplemented  
13  
14 def __truediv__(self, other) -> Union[NotImplementedType, "Fraction"]:  
15     ↪ """  
16     Divide this fraction by another fraction.  
17  
18     :param other: the other fraction  
19     :return: the result of the division  
20  
21     >>> print(Fraction(6, 19) / Fraction(3, -7))  
22     -14/19  
23     """  
24    return Fraction(self.a * other.b, self.b * other.a) \  
25           if isinstance(other, Fraction) else NotImplemented  
26  
27 def __abs__(self) -> "Fraction":  
28     """  
29     Get the absolute value of this fraction.  
30  
31     :return: the absolute value.  
32  
33     >>> print(abs(Fraction(-1, 2)))  
34     1/2  
35     >>> print(abs(Fraction(3, 5)))  
36     3/5  
37     """  
38    return self if self.a > 0 else Fraction(-self.a, self.b)
```

Fraction: Vergleichsoperatoren

- Zuletzt implementieren wir noch die sechs Vergleichsoperatoren als Dunder-Methoden, wie in [26] spezifiziert.

```
def __eq__(self, other) -> bool | NotImplemented:
    """
    Check whether this fraction equals another fraction.

    :param other: the other fraction
    :returns: 'True' if 'self' equals 'other', 'False' otherwise
    """
    return (self.a == other.a) and (self.b == other.b) \
        if isinstance(other, Fraction) else NotImplemented

def __ne__(self, other) -> bool | NotImplemented:
    """
    Check whether this fraction does not equal another fraction.

    :param other: the other fraction
    :returns: 'False' if 'self' equals 'other', 'True' otherwise
    """
    return (self.a != other.a) or (self.b != other.b) \
        if isinstance(other, Fraction) else NotImplemented

def __lt__(self, other) -> bool | NotImplemented:
    """
    Check whether this fraction is less than another fraction.

    :param other: the other fraction
    :returns: 'True' if 'self' less than 'other', 'False' otherwise
    """
    return ((self.a * other.b) < (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented

def __le__(self, other) -> bool | NotImplemented:
    """
    Check whether this fraction is less than or equal to another.

    :param other: the other fraction
    :returns: 'True' if 'self' <= 'other', 'False' otherwise
    """
    return ((self.a * other.b) <= (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented

def __gt__(self, other) -> bool | NotImplemented:
    """
    Check whether this fraction is greater than another fraction.

    :param other: the other fraction
    :returns: 'True' if 'self' > 'other', 'False' otherwise
    """
    return ((self.a * other.b) > (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented

def __ge__(self, other) -> bool | NotImplemented:
    """
    Check whether this fraction is greater than or equal to another.

    :param other: the other fraction
    :returns: 'True' if 'self' >= 'other', 'False' otherwise
    """
    return ((self.a * other.b) >= (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented
```



Fraction: Vergleichsoperatoren

- Zuletzt implementieren wir noch die sechs Vergleichsoperatoren als Dunder-Methoden, wie in [26] spezifiziert
 - `__eq__` implementiert die Funktionalität von `==`.

```
def __eq__(self, other) -> bool | NotImplemented:
    """
    Check whether this fraction equals another fraction.

    :param other: the other fraction
    :returns: 'True' if 'self' equals 'other', 'False' otherwise
    """
    return (self.a == other.a) and (self.b == other.b) \
        if isinstance(other, Fraction) else NotImplemented

def __ne__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction does not equal another fraction.

    :param other: the other fraction
    :returns: 'False' if 'self' equals 'other', 'True' otherwise
    """
    return (self.a != other.a) or (self.b != other.b) \
        if isinstance(other, Fraction) else NotImplemented

def __lt__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is less than another fraction.

    :param other: the other fraction
    :returns: 'True' if 'self' less than 'other', 'False' otherwise
    """
    return ((self.a * other.b) < (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented

def __le__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is less than or equal to another.

    :param other: the other fraction
    :returns: 'True' if 'self' <= 'other', 'False' otherwise
    """
    return ((self.a * other.b) <= (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented

def __gt__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is greater than another fraction.

    :param other: the other fraction
    :returns: 'True' if 'self' > 'other', 'False' otherwise
    """
    return ((self.a * other.b) > (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented

def __ge__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is greater than or equal to another.

    :param other: the other fraction
    :returns: 'True' if 'self' >= 'other', 'False' otherwise
    """
    return ((self.a * other.b) >= (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented
```



Fraction: Vergleichsoperatoren

- Zuletzt implementieren wir noch die sechs Vergleichsoperatoren als Dunder-Methoden, wie in [26] spezifiziert
 - `__eq__` implementiert die Funktionalität von `==`.
 - `__ne__` implementiert die Funktionalität von `!=`.

```
def __eq__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction equals another fraction.

    :param other: the other fraction
    :returns: 'True' if 'self' equals 'other', 'False' otherwise
    """
    return (self.a == other.a) and (self.b == other.b) \
        if isinstance(other, Fraction) else NotImplemented

def __ne__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction does not equal another fraction.

    :param other: the other fraction
    :returns: 'False' if 'self' equals 'other', 'True' otherwise
    """
    return (self.a != other.a) or (self.b != other.b) \
        if isinstance(other, Fraction) else NotImplemented

def __lt__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is less than another fraction.

    :param other: the other fraction
    :returns: 'True' if 'self' less than 'other', 'False' otherwise
    """
    return ((self.a * other.b) < (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented

def __le__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is less than or equal to another.

    :param other: the other fraction
    :returns: 'True' if 'self' <= 'other', 'False' otherwise
    """
    return ((self.a * other.b) <= (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented

def __gt__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is greater than another fraction.

    :param other: the other fraction
    :returns: 'True' if 'self' > 'other', 'False' otherwise
    """
    return ((self.a * other.b) > (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented

def __ge__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is greater than or equal to another.

    :param other: the other fraction
    :returns: 'True' if 'self' >= 'other', 'False' otherwise
    """
    return ((self.a * other.b) >= (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented
```



Fraction: Vergleichsoperatoren

- Zuletzt implementieren wir noch die sechs Vergleichsoperatoren als Dunder-Methoden, wie in [26] spezifiziert

- `__eq__` implementiert die Funktionalität von `==`.
- `__ne__` implementiert die Funktionalität von `!=`.
- `__lt__` implementiert die Funktionalität von `<`.

```
def __eq__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction equals another fraction.

    :param other: the other fraction
    :returns: 'True' if 'self' equals 'other', 'False' otherwise
    """
    return (self.a == other.a) and (self.b == other.b) \
        if isinstance(other, Fraction) else NotImplemented

def __ne__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction does not equal another fraction.

    :param other: the other fraction
    :returns: 'False' if 'self' equals 'other', 'True' otherwise
    """
    return (self.a != other.a) or (self.b != other.b) \
        if isinstance(other, Fraction) else NotImplemented

def __lt__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is less than another fraction.

    :param other: the other fraction
    :returns: 'True' if 'self' less than 'other', 'False' otherwise
    """
    return ((self.a * other.b) < (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented

def __le__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is less than or equal to another.

    :param other: the other fraction
    :returns: 'True' if 'self' <= 'other', 'False' otherwise
    """
    return ((self.a * other.b) <= (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented

def __gt__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is greater than another fraction.

    :param other: the other fraction
    :returns: 'True' if 'self' > 'other', 'False' otherwise
    """
    return ((self.a * other.b) > (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented

def __ge__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is greater than or equal to another.

    :param other: the other fraction
    :returns: 'True' if 'self' >= 'other', 'False' otherwise
    """
    return ((self.a * other.b) >= (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented
```



Fraction: Vergleichsoperatoren

- Zuletzt implementieren wir noch die sechs Vergleichsoperatoren als Dunder-Methoden, wie in [26] spezifiziert

- `__eq__` implementiert die Funktionalität von `==`.
- `__ne__` implementiert die Funktionalität von `!=`.
- `__lt__` implementiert die Funktionalität von `<`.
- `__le__` implementiert die Funktionalität von `<=`.

```
def __eq__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction equals another fraction.

    :param other: the other fraction
    :returns: 'True' if 'self' equals 'other', 'False' otherwise
    """
    return (self.a == other.a) and (self.b == other.b) \
        if isinstance(other, Fraction) else NotImplemented

def __ne__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction does not equal another fraction.

    :param other: the other fraction
    :returns: 'False' if 'self' equals 'other', 'True' otherwise
    """
    return (self.a != other.a) or (self.b != other.b) \
        if isinstance(other, Fraction) else NotImplemented

def __lt__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is less than another fraction.

    :param other: the other fraction
    :returns: 'True' if 'self' less than 'other', 'False' otherwise
    """
    return ((self.a * other.b) < (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented

def __le__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is less than or equal to another.

    :param other: the other fraction
    :returns: 'True' if 'self' <= 'other', 'False' otherwise
    """
    return ((self.a * other.b) <= (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented

def __gt__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is greater than another fraction.

    :param other: the other fraction
    :returns: 'True' if 'self' > 'other', 'False' otherwise
    """
    return ((self.a * other.b) > (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented

def __ge__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is greater than or equal to another.

    :param other: the other fraction
    :returns: 'True' if 'self' >= 'other', 'False' otherwise
    """
    return ((self.a * other.b) >= (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented
```



Fraction: Vergleichsoperatoren

- Zuletzt implementieren wir noch die sechs Vergleichsoperatoren als Dunder-Methoden, wie in [26] spezifiziert

- `__eq__` implementiert die Funktionalität von `==`.
- `__ne__` implementiert die Funktionalität von `!=`.
- `__lt__` implementiert die Funktionalität von `<`.
- `__le__` implementiert die Funktionalität von `<=`.
- `__gt__` implementiert die Funktionalität von `>`.

```
def __eq__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction equals another fraction.

    :param other: the other fraction
    :returns: 'True' if 'self' equals 'other', 'False' otherwise
    """
    return (self.a == other.a) and (self.b == other.b) \
        if isinstance(other, Fraction) else NotImplemented

def __ne__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction does not equal another fraction.

    :param other: the other fraction
    :returns: 'False' if 'self' equals 'other', 'True' otherwise
    """
    return (self.a != other.a) or (self.b != other.b) \
        if isinstance(other, Fraction) else NotImplemented

def __lt__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is less than another fraction.

    :param other: the other fraction
    :returns: 'True' if 'self' less than 'other', 'False' otherwise
    """
    return ((self.a * other.b) < (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented

def __le__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is less than or equal to another.

    :param other: the other fraction
    :returns: 'True' if 'self' <= 'other', 'False' otherwise
    """
    return ((self.a * other.b) <= (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented

def __gt__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is greater than another fraction.

    :param other: the other fraction
    :returns: 'True' if 'self' > 'other', 'False' otherwise
    """
    return ((self.a * other.b) > (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented

def __ge__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is greater than or equal to another.

    :param other: the other fraction
    :returns: 'True' if 'self' >= 'other', 'False' otherwise
    """
    return ((self.a * other.b) >= (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented
```



Fraction: Vergleichsoperatoren

- Zuletzt implementieren wir noch die sechs Vergleichsoperatoren als Dunder-Methoden, wie in [26] spezifiziert

- `__ne__` implementiert die Funktionalität von `!=`.
- `__lt__` implementiert die Funktionalität von `<`.
- `__le__` implementiert die Funktionalität von `<=`.
- `__gt__` implementiert die Funktionalität von `>`.
- `__ge__` implementiert die Funktionalität von `>=`.

```
def __eq__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction equals another fraction.

    :param other: the other fraction
    :returns: 'True' if 'self' equals 'other', 'False' otherwise
    """
    return (self.a == other.a) and (self.b == other.b) \
        if isinstance(other, Fraction) else NotImplemented

def __ne__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction does not equal another fraction.

    :param other: the other fraction
    :returns: 'False' if 'self' equals 'other', 'True' otherwise
    """
    return (self.a != other.a) or (self.b != other.b) \
        if isinstance(other, Fraction) else NotImplemented

def __lt__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is less than another fraction.

    :param other: the other fraction
    :returns: 'True' if 'self' less than 'other', 'False' otherwise
    """
    return ((self.a * other.b) < (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented

def __le__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is less than or equal to another.

    :param other: the other fraction
    :returns: 'True' if 'self' <= 'other', 'False' otherwise
    """
    return ((self.a * other.b) <= (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented

def __gt__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is greater than another fraction.

    :param other: the other fraction
    :returns: 'True' if 'self' > 'other', 'False' otherwise
    """
    return ((self.a * other.b) > (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented

def __ge__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is greater than or equal to another.

    :param other: the other fraction
    :returns: 'True' if 'self' >= 'other', 'False' otherwise
    """
    return ((self.a * other.b) >= (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented
```



Fraction: Vergleichsoperatoren

- Zuletzt implementieren wir noch die sechs Vergleichsoperatoren als Dunder-Methoden, wie in [26] spezifiziert
 - `__lt__` implementiert die Funktionalität von `<`.
 - `__le__` implementiert die Funktionalität von `<=`.
 - `__gt__` implementiert die Funktionalität von `>`.
 - `__ge__` implementiert die Funktionalität von `>=`.
- Der Gleichheits- und der Ungleichheitsoperator sind sehr einfach.

```
def __eq__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction equals another fraction.

    :param other: the other fraction
    :returns: 'True' if 'self' equals 'other', 'False' otherwise
    """
    return (self.a == other.a) and (self.b == other.b) \
        if isinstance(other, Fraction) else NotImplemented

def __ne__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction does not equal another fraction.

    :param other: the other fraction
    :returns: 'False' if 'self' equals 'other', 'True' otherwise
    """
    return (self.a != other.a) or (self.b != other.b) \
        if isinstance(other, Fraction) else NotImplemented

def __lt__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is less than another fraction.

    :param other: the other fraction
    :returns: 'True' if 'self' less than 'other', 'False' otherwise
    """
    return ((self.a * other.b) < (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented

def __le__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is less than or equal to another.

    :param other: the other fraction
    :returns: 'True' if 'self' <= 'other', 'False' otherwise
    """
    return ((self.a * other.b) <= (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented

def __gt__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is greater than another fraction.

    :param other: the other fraction
    :returns: 'True' if 'self' > 'other', 'False' otherwise
    """
    return ((self.a * other.b) > (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented

def __ge__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is greater than or equal to another.

    :param other: the other fraction
    :returns: 'True' if 'self' >= 'other', 'False' otherwise
    """
    return ((self.a * other.b) >= (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented
```



Fraction: Vergleichsoperatoren

- Zuletzt implementieren wir noch die sechs Vergleichsoperatoren als Dunder-Methoden, wie in [26] spezifiziert
 - `__le__` implementiert die Funktionalität von `<=`.
 - `__gt__` implementiert die Funktionalität von `>`.
 - `__ge__` implementiert die Funktionalität von `>=`.
- Der Gleichheits- und der Ungleichheitsoperator sind sehr einfach.
- Unsere Brüche sind ja normalisiert.

```
def __eq__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction equals another fraction.

    :param other: the other fraction
    :returns: 'True' if 'self' equals 'other', 'False' otherwise
    """
    return (self.a == other.a) and (self.b == other.b) \
        if isinstance(other, Fraction) else NotImplemented

def __ne__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction does not equal another fraction.

    :param other: the other fraction
    :returns: 'False' if 'self' equals 'other', 'True' otherwise
    """
    return (self.a != other.a) or (self.b != other.b) \
        if isinstance(other, Fraction) else NotImplemented

def __lt__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is less than another fraction.

    :param other: the other fraction
    :returns: 'True' if 'self' less than 'other', 'False' otherwise
    """
    return ((self.a * other.b) < (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented

def __le__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is less than or equal to another.

    :param other: the other fraction
    :returns: 'True' if 'self' <= 'other', 'False' otherwise
    """
    return ((self.a * other.b) <= (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented

def __gt__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is greater than another fraction.

    :param other: the other fraction
    :returns: 'True' if 'self' > 'other', 'False' otherwise
    """
    return ((self.a * other.b) > (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented

def __ge__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is greater than or equal to another.

    :param other: the other fraction
    :returns: 'True' if 'self' >= 'other', 'False' otherwise
    """
    return ((self.a * other.b) >= (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented
```



Fraction: Vergleichsoperatoren

- Zuletzt implementieren wir noch die sechs Vergleichsoperatoren als Dunder-Methoden, wie in [26] spezifiziert
 - `__gt__` implementiert die Funktionalität von `>`.
 - `__ge__` implementiert die Funktionalität von `>=`.
- Der Gleichheits- und der Ungleichheitsoperator sind sehr einfach.
- Unsere Brüche sind ja normalisiert.
- Für zwei Brüche `x` und `y` gilt `x == y` dann und nur dann, wenn `x.a == y.a` und `x.b == y.b`.

```
def __eq__(self, other) -> bool | NotImplemented:
    """
    Check whether this fraction equals another fraction.

    :param other: the other fraction
    :returns: 'True' if 'self' equals 'other', 'False' otherwise
    """
    return (self.a == other.a) and (self.b == other.b) \
        if isinstance(other, Fraction) else NotImplemented

def __ne__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction does not equal another fraction.

    :param other: the other fraction
    :returns: 'False' if 'self' equals 'other', 'True' otherwise
    """
    return (self.a != other.a) or (self.b != other.b) \
        if isinstance(other, Fraction) else NotImplemented

def __lt__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is less than another fraction.

    :param other: the other fraction
    :returns: 'True' if 'self' less than 'other', 'False' otherwise
    """
    return ((self.a * other.b) < (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented

def __le__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is less than or equal to another.

    :param other: the other fraction
    :returns: 'True' if 'self' <= 'other', 'False' otherwise
    """
    return ((self.a * other.b) <= (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented

def __gt__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is greater than another fraction.

    :param other: the other fraction
    :returns: 'True' if 'self' > 'other', 'False' otherwise
    """
    return ((self.a * other.b) > (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented

def __ge__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is greater than or equal to another.

    :param other: the other fraction
    :returns: 'True' if 'self' >= 'other', 'False' otherwise
    """
    return ((self.a * other.b) >= (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented
```



Fraction: Vergleichsoperatoren

- Zuletzt implementieren wir noch die sechs Vergleichsoperatoren als Dunder-Methoden, wie in [26] spezifiziert
 - `__ge__` implementiert die Funktionalität von `>=`.
- Der Gleichheits- und der Ungleichheitsoperator sind sehr einfach.
- Unsere Brüche sind ja normalisiert.
- Für zwei Brüche `x` und `y` gilt `x == y` dann und nur dann, wenn `x.a == y.a` und `x.b == y.b`.
- Ist daher schnell implementiert.

```
def __eq__(self, other) -> bool | NotImplemented:
    """
    Check whether this fraction equals another fraction.

    :param other: the other fraction
    :returns: 'True' if 'self' equals 'other', 'False' otherwise
    """
    return (self.a == other.a) and (self.b == other.b) \
        if isinstance(other, Fraction) else NotImplemented

def __ne__(self, other) -> bool | NotImplemented:
    """
    Check whether this fraction does not equal another fraction.

    :param other: the other fraction
    :returns: 'False' if 'self' equals 'other', 'True' otherwise
    """
    return (self.a != other.a) or (self.b != other.b) \
        if isinstance(other, Fraction) else NotImplemented

def __lt__(self, other) -> bool | NotImplemented:
    """
    Check whether this fraction is less than another fraction.

    :param other: the other fraction
    :returns: 'True' if 'self' less than 'other', 'False' otherwise
    """
    return ((self.a * other.b) < (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented

def __le__(self, other) -> bool | NotImplemented:
    """
    Check whether this fraction is less than or equal to another.

    :param other: the other fraction
    :returns: 'True' if 'self' <= 'other', 'False' otherwise
    """
    return ((self.a * other.b) <= (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented

def __gt__(self, other) -> bool | NotImplemented:
    """
    Check whether this fraction is greater than another fraction.

    :param other: the other fraction
    :returns: 'True' if 'self' > 'other', 'False' otherwise
    """
    return ((self.a * other.b) > (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented

def __ge__(self, other) -> bool | NotImplemented:
    """
    Check whether this fraction is greater than or equal to another.

    :param other: the other fraction
    :returns: 'True' if 'self' >= 'other', 'False' otherwise
    """
    return ((self.a * other.b) >= (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented
```



Fraction: Vergleichsoperatoren

- Der Gleichheits- und der Ungleichheitsoperator sind sehr einfach.
- Unsere Brüche sind ja normalisiert.
- Für zwei Brüche x und y gilt $x == y$ dann und nur dann, wenn $x.a == y.a$ und $x.b == y.b$.
- Ist daher schnell implementiert.
- `__ne__` ist das Komplement davon für den `!=`-Operator.

```
def __eq__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction equals another fraction.

    :param other: the other fraction
    :returns: 'True' if 'self' equals 'other', 'False' otherwise
    """
    return (self.a == other.a) and (self.b == other.b) \
        if isinstance(other, Fraction) else NotImplemented

def __ne__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction does not equal another fraction.

    :param other: the other fraction
    :returns: 'False' if 'self' equals 'other', 'True' otherwise
    """
    return (self.a != other.a) or (self.b != other.b) \
        if isinstance(other, Fraction) else NotImplemented

def __lt__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is less than another fraction.

    :param other: the other fraction
    :returns: 'True' if 'self' less than 'other', 'False' otherwise
    """
    return ((self.a * other.b) < (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented

def __le__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is less than or equal to another.

    :param other: the other fraction
    :returns: 'True' if 'self' <= 'other', 'False' otherwise
    """
    return ((self.a * other.b) <= (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented

def __gt__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is greater than another fraction.

    :param other: the other fraction
    :returns: 'True' if 'self' > 'other', 'False' otherwise
    """
    return ((self.a * other.b) > (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented

def __ge__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is greater than or equal to another.

    :param other: the other fraction
    :returns: 'True' if 'self' >= 'other', 'False' otherwise
    """
    return ((self.a * other.b) >= (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented
```



Fraction: Vergleichsoperatoren

- Unsere Brüche sind ja normalisiert.
- Für zwei Brüche `x` und `y` gilt `x == y` dann und nur dann, wenn `x.a == y.a` und `x.b == y.b`.
- Ist daher schnell implementiert.
- `__ne__` ist das Komplement davon für den `!=`-Operator.
- `x != y` ist `True` wenn `x.a != y.a` oder `x.b != y.b` wahr sind.

```
def __eq__(self, other) -> bool | NotImplemented:
    """
    Check whether this fraction equals another fraction.

    :param other: the other fraction
    :returns: 'True' if 'self' equals 'other', 'False' otherwise
    """
    return (self.a == other.a) and (self.b == other.b) \
        if isinstance(other, Fraction) else NotImplemented

def __ne__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction does not equal another fraction.

    :param other: the other fraction
    :returns: 'False' if 'self' equals 'other', 'True' otherwise
    """
    return (self.a != other.a) or (self.b != other.b) \
        if isinstance(other, Fraction) else NotImplemented

def __lt__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is less than another fraction.

    :param other: the other fraction
    :returns: 'True' if 'self' less than 'other', 'False' otherwise
    """
    return ((self.a * other.b) < (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented

def __le__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is less than or equal to another.

    :param other: the other fraction
    :returns: 'True' if 'self' <= 'other', 'False' otherwise
    """
    return ((self.a * other.b) <= (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented

def __gt__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is greater than another fraction.

    :param other: the other fraction
    :returns: 'True' if 'self' > 'other', 'False' otherwise
    """
    return ((self.a * other.b) > (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented

def __ge__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is greater than or equal to another.

    :param other: the other fraction
    :returns: 'True' if 'self' >= 'other', 'False' otherwise
    """
    return ((self.a * other.b) >= (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented
```



Fraction: Vergleichsoperatoren

- Für zwei Brüche x und y gilt $x == y$ dann und nur dann, wenn $x.a == y.a$ und $x.b == y.b$.
- Ist daher schnell implementiert.
- `__ne__` ist das Komplement davon für den `!=`-Operator.
- $x != y$ ist `True` wenn $x.a != y.a$ oder $x.b != y.b$ wahr sind.
- Die anderen vier Operatoren können implementiert werden, wenn wir uns daran erinnern, wie wir den gemeinsamen Nenner für Addition und Subtraktion verwendet haben.

```
def __eq__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction equals another fraction.

    :param other: the other fraction
    :returns: 'True' if 'self' equals 'other', 'False' otherwise
    """
    return (self.a == other.a) and (self.b == other.b) \
        if isinstance(other, Fraction) else NotImplemented

def __ne__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction does not equal another fraction.

    :param other: the other fraction
    :returns: 'False' if 'self' equals 'other', 'True' otherwise
    """
    return (self.a != other.a) or (self.b != other.b) \
        if isinstance(other, Fraction) else NotImplemented

def __lt__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is less than another fraction.

    :param other: the other fraction
    :returns: 'True' if 'self' less than 'other', 'False' otherwise
    """
    return ((self.a * other.b) < (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented

def __le__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is less than or equal to another.

    :param other: the other fraction
    :returns: 'True' if 'self' <= 'other', 'False' otherwise
    """
    return ((self.a * other.b) <= (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented

def __gt__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is greater than another fraction.

    :param other: the other fraction
    :returns: 'True' if 'self' > 'other', 'False' otherwise
    """
    return ((self.a * other.b) > (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented

def __ge__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is greater than or equal to another.

    :param other: the other fraction
    :returns: 'True' if 'self' >= 'other', 'False' otherwise
    """
    return ((self.a * other.b) >= (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented
```



Fraction: Vergleichsoperatoren

- Ist daher schnell implementiert.
- `__ne__` ist das Komplement davon für den `!=`-Operator.
- `x != y` ist `True` wenn `x.a != y.a` oder `x.b != y.b` wahr sind.
- Die anderen vier Operatoren können implementiert werden, wenn wir uns daran erinnern, wie wir den gemeinsamen Nenner für Addition und Subtraktion verwendet haben.
- Wir haben so
addiert: $\frac{a}{b} + \frac{c}{d} = \frac{a*d}{b*d} + \frac{c*b}{b*d} = \frac{a*d+c*b}{b*d}$.

```
def __eq__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction equals another fraction.

    :param other: the other fraction
    :returns: 'True' if 'self' equals 'other', 'False' otherwise
    """
    return (self.a == other.a) and (self.b == other.b) \
        if isinstance(other, Fraction) else NotImplemented

def __ne__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction does not equal another fraction.

    :param other: the other fraction
    :returns: 'False' if 'self' equals 'other', 'True' otherwise
    """
    return (self.a != other.a) or (self.b != other.b) \
        if isinstance(other, Fraction) else NotImplemented

def __lt__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is less than another fraction.

    :param other: the other fraction
    :returns: 'True' if 'self' less than 'other', 'False' otherwise
    """
    return ((self.a * other.b) < (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented

def __le__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is less than or equal to another.

    :param other: the other fraction
    :returns: 'True' if 'self' <= 'other', 'False' otherwise
    """
    return ((self.a * other.b) <= (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented

def __gt__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is greater than another fraction.

    :param other: the other fraction
    :returns: 'True' if 'self' > 'other', 'False' otherwise
    """
    return ((self.a * other.b) > (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented

def __ge__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is greater than or equal to another.

    :param other: the other fraction
    :returns: 'True' if 'self' >= 'other', 'False' otherwise
    """
    return ((self.a * other.b) >= (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented
```



Fraction: Vergleichsoperatoren

- `__ne__` ist das Komplement davon für den `!=`-Operator.
- `x != y` ist `True` wenn `x.a != y.a` oder `x.b != y.b` wahr sind.
- Die anderen vier Operatoren können implementiert werden, wenn wir uns daran erinnern, wie wir den gemeinsamen Nenner für Addition und Subtraktion verwendet haben.
- Wir haben so addiert: $\frac{a}{b} + \frac{c}{d} = \frac{a*d}{b*d} + \frac{c*b}{b*d} = \frac{a*d+c*b}{b*d}$.
- Wenn wir uns das nochmal angucken, dann stellen wir fest, dass $\frac{a}{b} < \frac{c}{d}$ das Gleiche wie $\frac{a*d}{b*d} < \frac{c*b}{b*d}$ ist.

```
def __eq__(self, other) -> bool | NotImplemented:
    """
    Check whether this fraction equals another fraction.

    :param other: the other fraction
    :returns: 'True' if 'self' equals 'other', 'False' otherwise
    """
    return (self.a == other.a) and (self.b == other.b) \
        if isinstance(other, Fraction) else NotImplemented

def __ne__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction does not equal another fraction.

    :param other: the other fraction
    :returns: 'False' if 'self' equals 'other', 'True' otherwise
    """
    return (self.a != other.a) or (self.b != other.b) \
        if isinstance(other, Fraction) else NotImplemented

def __lt__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is less than another fraction.

    :param other: the other fraction
    :returns: 'True' if 'self' less than 'other', 'False' otherwise
    """
    return ((self.a * other.b) < (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented

def __le__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is less than or equal to another.

    :param other: the other fraction
    :returns: 'True' if 'self' <= 'other', 'False' otherwise
    """
    return ((self.a * other.b) <= (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented

def __gt__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is greater than another fraction.

    :param other: the other fraction
    :returns: 'True' if 'self' > 'other', 'False' otherwise
    """
    return ((self.a * other.b) > (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented

def __ge__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is greater than or equal to another.

    :param other: the other fraction
    :returns: 'True' if 'self' >= 'other', 'False' otherwise
    """
    return ((self.a * other.b) >= (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented
```



Fraction: Vergleichsoperatoren

- $x \neq y$ ist `True` wenn $x.a \neq y.a$ oder $x.b \neq y.b$ wahr sind.
- Die anderen vier Operatoren können implementiert werden, wenn wir uns daran erinnern, wie wir den gemeinsamen Nenner für Addition und Subtraktion verwendet haben.
- Wir haben so addiert: $\frac{a}{b} + \frac{c}{d} = \frac{a*d}{b*d} + \frac{c*b}{b*d} = \frac{a*d+c*b}{b*d}$.
- Wenn wir uns das nochmal angucken, dann stellen wir fest, dass $\frac{a}{b} < \frac{c}{d}$ das Gleiche wie $\frac{a*d}{b*d} < \frac{c*b}{b*d}$ ist.
- Das ist wiederum das Gleiche wie $a * d < c * b$.

```
def __eq__(self, other) -> bool | NotImplemented:
    """
    Check whether this fraction equals another fraction.

    :param other: the other fraction
    :returns: 'True' if 'self' equals 'other', 'False' otherwise
    """
    return (self.a == other.a) and (self.b == other.b) \
        if isinstance(other, Fraction) else NotImplemented

def __ne__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction does not equal another fraction.

    :param other: the other fraction
    :returns: 'False' if 'self' equals 'other', 'True' otherwise
    """
    return (self.a != other.a) or (self.b != other.b) \
        if isinstance(other, Fraction) else NotImplemented

def __lt__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is less than another fraction.

    :param other: the other fraction
    :returns: 'True' if 'self' less than 'other', 'False' otherwise
    """
    return ((self.a * other.b) < (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented

def __le__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is less than or equal to another.

    :param other: the other fraction
    :returns: 'True' if 'self' <= 'other', 'False' otherwise
    """
    return ((self.a * other.b) <= (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented

def __gt__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is greater than another fraction.

    :param other: the other fraction
    :returns: 'True' if 'self' > 'other', 'False' otherwise
    """
    return ((self.a * other.b) > (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented

def __ge__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is greater than or equal to another.

    :param other: the other fraction
    :returns: 'True' if 'self' >= 'other', 'False' otherwise
    """
    return ((self.a * other.b) >= (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented
```



Fraction: Vergleichsoperatoren

- Die anderen vier Operatoren können implementiert werden, wenn wir uns daran erinnern, wie wir den gemeinsamen Nenner für Addition und Subtraktion verwendet haben.
- Wir haben so addiert: $\frac{a}{b} + \frac{c}{d} = \frac{a*d}{b*d} + \frac{c*b}{b*d} = \frac{a*d+c*b}{b*d}$.
- Wenn wir uns das nochmal angucken, dann stellen wir fest, dass $\frac{a}{b} < \frac{c}{d}$ das Gleiche wie $\frac{a*d}{b*d} < \frac{c*b}{b*d}$ ist.
- Das ist wiederum das Gleiche wie $a * d < c * b$.
- Daher ist $\frac{a}{b} \leq \frac{c}{d}$ das Gleiche wie $a * d \leq c * b$.

```
def __eq__(self, other) -> bool | NotImplemented:
    """
    Check whether this fraction equals another fraction.

    :param other: the other fraction
    :returns: 'True' if 'self' equals 'other', 'False' otherwise
    """
    return (self.a == other.a) and (self.b == other.b) \
        if isinstance(other, Fraction) else NotImplemented

def __ne__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction does not equal another fraction.

    :param other: the other fraction
    :returns: 'False' if 'self' equals 'other', 'True' otherwise
    """
    return (self.a != other.a) or (self.b != other.b) \
        if isinstance(other, Fraction) else NotImplemented

def __lt__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is less than another fraction.

    :param other: the other fraction
    :returns: 'True' if 'self' less than 'other', 'False' otherwise
    """
    return ((self.a * other.b) < (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented

def __le__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is less than or equal to another.

    :param other: the other fraction
    :returns: 'True' if 'self' <= 'other', 'False' otherwise
    """
    return ((self.a * other.b) <= (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented

def __gt__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is greater than another fraction.

    :param other: the other fraction
    :returns: 'True' if 'self' > 'other', 'False' otherwise
    """
    return ((self.a * other.b) > (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented

def __ge__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is greater than or equal to another.

    :param other: the other fraction
    :returns: 'True' if 'self' >= 'other', 'False' otherwise
    """
    return ((self.a * other.b) >= (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented
```



Fraction: Vergleichsoperatoren

- Wir haben so
addiert: $\frac{a}{b} + \frac{c}{d} = \frac{a*d}{b*d} + \frac{c*b}{b*d} = \frac{a*d+c*b}{b*d}$.
- Wenn wir uns das nochmal angucken,
dann stellen wir fest, das $\frac{a}{b} < \frac{c}{d}$ das
Gleiche wir $\frac{a*d}{b*d} < \frac{c*b}{b*d}$ ist.
- Das ist wiederum das Gleiche
wir $a * d < c * b$.
- Daher ist $\frac{a}{b} \leq \frac{c}{d}$ das Gleiche wie
 $a * d \leq c * b$.
- Der „größer-“ und der
„größer-gleich-Operator“ können
genauso nur andersherum
implementiert werden.

```
def __eq__(self, other) -> bool | NotImplemented:
    """
    Check whether this fraction equals another fraction.

    :param other: the other fraction
    :returns: 'True' if 'self' equals 'other', 'False' otherwise
    """
    return (self.a == other.a) and (self.b == other.b) \
        if isinstance(other, Fraction) else NotImplemented

def __ne__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction does not equal another fraction.

    :param other: the other fraction
    :returns: 'False' if 'self' equals 'other', 'True' otherwise
    """
    return (self.a != other.a) or (self.b != other.b) \
        if isinstance(other, Fraction) else NotImplemented

def __lt__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is less than another fraction.

    :param other: the other fraction
    :returns: 'True' if 'self' less than 'other', 'False' otherwise
    """
    return ((self.a * other.b) < (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented

def __le__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is less than or equal to another.

    :param other: the other fraction
    :returns: 'True' if 'self' <= 'other', 'False' otherwise
    """
    return ((self.a * other.b) <= (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented

def __gt__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is greater than another fraction.

    :param other: the other fraction
    :returns: 'True' if 'self' > 'other', 'False' otherwise
    """
    return ((self.a * other.b) > (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented

def __ge__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is greater than or equal to another.

    :param other: the other fraction
    :returns: 'True' if 'self' >= 'other', 'False' otherwise
    """
    return ((self.a * other.b) >= (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented
```



Fraction: Vergleichsoperatoren

- Wenn wir uns das nochmal angucken, dann stellen wir fest, das $\frac{a}{b} < \frac{c}{d}$ das Gleiche wir $\frac{a*d}{b*d} < \frac{c*b}{b*d}$ ist.
- Das ist wiederum das Gleiche wir $a * d < c * b$.
- Daher ist $\frac{a}{b} \leq \frac{c}{d}$ das Gleiche wie $a * d \leq c * b$.
- Der „größer-“ und der „größer-gleich-Operator“ können genauso nur andersherum implementiert werden.
- Diesmal lasse ich die Doctests aus Platzgründen weg.

```
def __eq__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction equals another fraction.

    :param other: the other fraction
    :returns: 'True' if 'self' equals 'other', 'False' otherwise
    """
    return (self.a == other.a) and (self.b == other.b) \
        if isinstance(other, Fraction) else NotImplemented

def __ne__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction does not equal another fraction.

    :param other: the other fraction
    :returns: 'False' if 'self' equals 'other', 'True' otherwise
    """
    return (self.a != other.a) or (self.b != other.b) \
        if isinstance(other, Fraction) else NotImplemented

def __lt__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is less than another fraction.

    :param other: the other fraction
    :returns: 'True' if 'self' less than 'other', 'False' otherwise
    """
    return ((self.a * other.b) < (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented

def __le__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is less than or equal to another.

    :param other: the other fraction
    :returns: 'True' if 'self' <= 'other', 'False' otherwise
    """
    return ((self.a * other.b) <= (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented

def __gt__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is greater than another fraction.

    :param other: the other fraction
    :returns: 'True' if 'self' > 'other', 'False' otherwise
    """
    return ((self.a * other.b) > (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented

def __ge__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is greater than or equal to another.

    :param other: the other fraction
    :returns: 'True' if 'self' >= 'other', 'False' otherwise
    """
    return ((self.a * other.b) >= (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented
```



Fraction: Vergleichsoperatoren

- Das ist wiederum das Gleiche wie $a * d < c * b$.
- Daher ist $\frac{a}{b} \leq \frac{c}{d}$ das Gleiche wie $a * d \leq c * b$.
- Der „größer-“ und der „größer-gleich-Operator“ können genauso nur andersherum implementiert werden.
- Diesmal lasse ich die Doctests aus Platzgründen weg.
- Genau genommen habe ich manches hier im Text etwas verkürzt.

```
def __eq__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction equals another fraction.

    :param other: the other fraction
    :returns: 'True' if 'self' equals 'other', 'False' otherwise
    """
    return (self.a == other.a) and (self.b == other.b) \
        if isinstance(other, Fraction) else NotImplemented

def __ne__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction does not equal another fraction.

    :param other: the other fraction
    :returns: 'False' if 'self' equals 'other', 'True' otherwise
    """
    return (self.a != other.a) or (self.b != other.b) \
        if isinstance(other, Fraction) else NotImplemented

def __lt__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is less than another fraction.

    :param other: the other fraction
    :returns: 'True' if 'self' less than 'other', 'False' otherwise
    """
    return ((self.a * other.b) < (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented

def __le__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is less than or equal to another.

    :param other: the other fraction
    :returns: 'True' if 'self' <= 'other', 'False' otherwise
    """
    return ((self.a * other.b) <= (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented

def __gt__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is greater than another fraction.

    :param other: the other fraction
    :returns: 'True' if 'self' > 'other', 'False' otherwise
    """
    return ((self.a * other.b) > (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented

def __ge__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is greater than or equal to another.

    :param other: the other fraction
    :returns: 'True' if 'self' >= 'other', 'False' otherwise
    """
    return ((self.a * other.b) >= (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented
```



Fraction: Vergleichsoperatoren

- Das ist wiederum das Gleiche wie $a * d < c * b$.
- Daher ist $\frac{a}{b} \leq \frac{c}{d}$ das Gleiche wie $a * d \leq c * b$.
- Der „größer-“ und der „größer-gleich-Operator“ können genauso nur andersherum implementiert werden.
- Diesmal lasse ich die Doctests aus Platzgründen weg.
- Genau genommen habe ich manches hier im Text etwas verkürzt.
- Der Initialisierer `__init__` prüft z. B. nicht, ob seine Parameter wirklich `int` sind...

```
1 def __eq__(self, other) -> bool | NotImplementedType:
2     """
3     Check whether this fraction equals another fraction.
4
5     :param other: the other fraction
6     :returns: 'True' if 'self' equals 'other', 'False' otherwise
7     """
8     return (self.a == other.a) and (self.b == other.b) \
9         if isinstance(other, Fraction) else NotImplemented
10
11 def __ne__(self, other) -> bool | NotImplementedType:
12     """
13     Check whether this fraction does not equal another fraction.
14
15     :param other: the other fraction
16     :returns: 'False' if 'self' equals 'other', 'True' otherwise
17     """
18     return (self.a != other.a) or (self.b != other.b) \
19         if isinstance(other, Fraction) else NotImplemented
20
21 def __lt__(self, other) -> bool | NotImplementedType:
22     """
23     Check whether this fraction is less than another fraction.
24
25     :param other: the other fraction
26     :returns: 'True' if 'self' less than 'other', 'False' otherwise
27     """
28     return ((self.a * other.b) < (other.a * self.b)) \
29         if isinstance(other, Fraction) else NotImplemented
30
31 def __le__(self, other) -> bool | NotImplementedType:
32     """
33     Check whether this fraction is less than or equal to another.
34
35     :param other: the other fraction
36     :returns: 'True' if 'self' <= 'other', 'False' otherwise
37     """
38     return ((self.a * other.b) <= (other.a * self.b)) \
39         if isinstance(other, Fraction) else NotImplemented
40
41 def __gt__(self, other) -> bool | NotImplementedType:
42     """
43     Check whether this fraction is greater than another fraction.
44
45     :param other: the other fraction
46     :returns: 'True' if 'self' > 'other', 'False' otherwise
47     """
48     return ((self.a * other.b) > (other.a * self.b)) \
49         if isinstance(other, Fraction) else NotImplemented
50
51 def __ge__(self, other) -> bool | NotImplementedType:
52     """
53     Check whether this fraction is greater than or equal to another.
54
55     :param other: the other fraction
56     :returns: 'True' if 'self' >= 'other', 'False' otherwise
57     """
58     return ((self.a * other.b) >= (other.a * self.b)) \
59         if isinstance(other, Fraction) else NotImplemented
```



Fraction: Vergleichsoperatoren

- Daher ist $\frac{a}{b} \leq \frac{c}{d}$ das Gleiche wie $a * d \leq c * b$.
- Der „größer-“ und der „größer-gleich-Operator“ können genauso nur andersherum implementiert werden.
- Diesmal lasse ich die Doctests aus Platzgründen weg.
- Genau genommen habe ich manches hier im Text etwas verkürzt.
- Der Initialisierer `__init__` prüft z. B. nicht, ob seine Parameter wirklich `int` sind...
- Das sollten wir in zusätzlichen Unit Tests prüfen.

```
def __eq__(self, other) -> bool | NotImplemented:
    """
    Check whether this fraction equals another fraction.

    :param other: the other fraction
    :returns: 'True' if 'self' equals 'other', 'False' otherwise
    """
    return (self.a == other.a) and (self.b == other.b) \
        if isinstance(other, Fraction) else NotImplemented

def __ne__(self, other) -> bool | NotImplemented:
    """
    Check whether this fraction does not equal another fraction.

    :param other: the other fraction
    :returns: 'False' if 'self' equals 'other', 'True' otherwise
    """
    return (self.a != other.a) or (self.b != other.b) \
        if isinstance(other, Fraction) else NotImplemented

def __lt__(self, other) -> bool | NotImplemented:
    """
    Check whether this fraction is less than another fraction.

    :param other: the other fraction
    :returns: 'True' if 'self' less than 'other', 'False' otherwise
    """
    return ((self.a * other.b) < (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented

def __le__(self, other) -> bool | NotImplemented:
    """
    Check whether this fraction is less than or equal to another.

    :param other: the other fraction
    :returns: 'True' if 'self' <= 'other', 'False' otherwise
    """
    return ((self.a * other.b) <= (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented

def __gt__(self, other) -> bool | NotImplemented:
    """
    Check whether this fraction is greater than another fraction.

    :param other: the other fraction
    :returns: 'True' if 'self' > 'other', 'False' otherwise
    """
    return ((self.a * other.b) > (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented

def __ge__(self, other) -> bool | NotImplemented:
    """
    Check whether this fraction is greater than or equal to another.

    :param other: the other fraction
    :returns: 'True' if 'self' >= 'other', 'False' otherwise
    """
    return ((self.a * other.b) >= (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented
```



Fraction: Vergleichsoperatoren

- Diesmal lasse ich die Doctests aus Platzgründen weg.
- Genau genommen habe ich manches hier im Text etwas verkürzt.
- Der Initialisierer `__init__` prüft z. B. nicht, ob seine Parameter wirklich `int` sind...
- Das sollten wir in zusätzlichen Unit Tests prüfen.
- Sie sollten aber niemals solche Tests weglassen.

```
def __eq__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction equals another fraction.

    :param other: the other fraction
    :returns: 'True' if 'self' equals 'other', 'False' otherwise
    """
    return (self.a == other.a) and (self.b == other.b) \
        if isinstance(other, Fraction) else NotImplemented

def __ne__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction does not equal another fraction.

    :param other: the other fraction
    :returns: 'False' if 'self' equals 'other', 'True' otherwise
    """
    return (self.a != other.a) or (self.b != other.b) \
        if isinstance(other, Fraction) else NotImplemented

def __lt__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is less than another fraction.

    :param other: the other fraction
    :returns: 'True' if 'self' less than 'other', 'False' otherwise
    """
    return ((self.a * other.b) < (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented

def __le__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is less than or equal to another.

    :param other: the other fraction
    :returns: 'True' if 'self' <= 'other', 'False' otherwise
    """
    return ((self.a * other.b) <= (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented

def __gt__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is greater than another fraction.

    :param other: the other fraction
    :returns: 'True' if 'self' > 'other', 'False' otherwise
    """
    return ((self.a * other.b) > (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented

def __ge__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is greater than or equal to another.

    :param other: the other fraction
    :returns: 'True' if 'self' >= 'other', 'False' otherwise
    """
    return ((self.a * other.b) >= (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented
```



Fraction: Vergleichsoperatoren

- Diesmal lasse ich die Doctests aus Platzgründen weg.
- Genau genommen habe ich manches hier im Text etwas verkürzt.
- Der Initialisierer `__init__` prüft z. B. nicht, ob seine Parameter wirklich `int` sind...
- Das sollten wir in zusätzlichen Unit Tests prüfen.
- Sie sollten aber niemals solche Tests weglassen.
- In Ihrem Programmcode haben Sie Platz.

```
def __eq__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction equals another fraction.

    :param other: the other fraction
    :returns: 'True' if 'self' equals 'other', 'False' otherwise
    """
    return (self.a == other.a) and (self.b == other.b) \
        if isinstance(other, Fraction) else NotImplemented

def __ne__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction does not equal another fraction.

    :param other: the other fraction
    :returns: 'False' if 'self' equals 'other', 'True' otherwise
    """
    return (self.a != other.a) or (self.b != other.b) \
        if isinstance(other, Fraction) else NotImplemented

def __lt__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is less than another fraction.

    :param other: the other fraction
    :returns: 'True' if 'self' less than 'other', 'False' otherwise
    """
    return ((self.a * other.b) < (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented

def __le__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is less than or equal to another.

    :param other: the other fraction
    :returns: 'True' if 'self' <= 'other', 'False' otherwise
    """
    return ((self.a * other.b) <= (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented

def __gt__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is greater than another fraction.

    :param other: the other fraction
    :returns: 'True' if 'self' > 'other', 'False' otherwise
    """
    return ((self.a * other.b) > (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented

def __ge__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is greater than or equal to another.

    :param other: the other fraction
    :returns: 'True' if 'self' >= 'other', 'False' otherwise
    """
    return ((self.a * other.b) >= (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented
```



Fraction: Vergleichsoperatoren

- Genau genommen habe ich manches hier im Text etwas verkürzt.
- Der Initialisierer `__init__` prüft z. B. nicht, ob seine Parameter wirklich `int` sind...
- Das sollten wir in zusätzlichen Unit Tests prüfen.
- Sie sollten aber niemals solche Tests weglassen.
- In Ihrem Programmcode haben Sie Platz.
- Der muss ja nicht auf eine Slide passen...

```
def __eq__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction equals another fraction.

    :param other: the other fraction
    :returns: 'True' if 'self' equals 'other', 'False' otherwise
    """
    return (self.a == other.a) and (self.b == other.b) \
        if isinstance(other, Fraction) else NotImplemented

def __ne__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction does not equal another fraction.

    :param other: the other fraction
    :returns: 'False' if 'self' equals 'other', 'True' otherwise
    """
    return (self.a != other.a) or (self.b != other.b) \
        if isinstance(other, Fraction) else NotImplemented

def __lt__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is less than another fraction.

    :param other: the other fraction
    :returns: 'True' if 'self' less than 'other', 'False' otherwise
    """
    return ((self.a * other.b) < (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented

def __le__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is less than or equal to another.

    :param other: the other fraction
    :returns: 'True' if 'self' <= 'other', 'False' otherwise
    """
    return ((self.a * other.b) <= (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented

def __gt__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is greater than another fraction.

    :param other: the other fraction
    :returns: 'True' if 'self' > 'other', 'False' otherwise
    """
    return ((self.a * other.b) > (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented

def __ge__(self, other) -> bool | NotImplementedType:
    """
    Check whether this fraction is greater than or equal to another.

    :param other: the other fraction
    :returns: 'True' if 'self' >= 'other', 'False' otherwise
    """
    return ((self.a * other.b) >= (other.a * self.b)) \
        if isinstance(other, Fraction) else NotImplemented
```



Ergebnis der Doctests

- Und was kommt bei den Doctests raus?



Ergebnis der Doctests



- Und was kommt bei den Doctests raus?

```
$ pytest --timeout=10 --no-header --tb=short --doctest-modules fraction.  
  ↳ py  
===== test session starts  
  ↳ =====  
collected 9 items  
  
fraction.py ..... [100%]  
  
===== 9 passed in 0.03s  
  ↳ =====  
# pytest 9.1.1 with pytest-timeout 2.4.0 succeeded with exit code 0.
```

Ergebnis der Doctests



- Und was kommt bei den Doctests raus?
- Dann passt's ja.

```
$ pytest --timeout=10 --no-header --tb=short --doctest-modules fraction.  
  ↳ py  
===== test session starts  
  ↳ =====  
collected 9 items  
  
fraction.py ..... [100%]  
  
===== 9 passed in 0.03s  
  ↳ =====  
# pytest 9.1.1 with pytest-timeout 2.4.0 succeeded with exit code 0.
```



Zusammenfassung



Zusammenfassung

- Es gibt noch ein paar mehr Dunder-Methoden, um arithmetische Operationen zu implementieren.



Zusammenfassung

- Es gibt noch ein paar mehr Dunder-Methoden, um arithmetische Operationen zu implementieren.
- Und das macht wirklich Spaß.



Zusammenfassung



- Es gibt noch ein paar mehr Dunder-Methoden, um arithmetische Operationen zu implementieren.
- Und das macht wirklich Spaß.
- Wir können unsere eigenen Klassen für Brüche und komplexe Zahlen haben!

Zusammenfassung



- Es gibt noch ein paar mehr Dunder-Methoden, um arithmetische Operationen zu implementieren.
- Und das macht wirklich Spaß.
- Wir können unsere eigenen Klassen für Brüche und komplexe Zahlen haben!
- Na gut, Python hat die schon. . .

Zusammenfassung



- Es gibt noch ein paar mehr Dunder-Methoden, um arithmetische Operationen zu implementieren.
- Und das macht wirklich Spaß.
- Wir können unsere eigenen Klassen für Brüche und komplexe Zahlen haben!
- Na gut, Python hat die schon. . .
- Aber wir könnten komplexe Zahlen mit Brüchen implementieren. . .

Zusammenfassung



- Es gibt noch ein paar mehr Dunder-Methoden, um arithmetische Operationen zu implementieren.
- Und das macht wirklich Spaß.
- Wir können unsere eigenen Klassen für Brüche und komplexe Zahlen haben!
- Na gut, Python hat die schon. . .
- Aber wir könnten komplexe Zahlen mit Brüchen implementieren. . .
- The sky is the limit!



谢谢你们！
Thank you!
Vielen Dank!



References I



- [1] Kent L. Beck. *JUnit Pocket Guide*. Sebastopol, CA, USA: O'Reilly Media, Inc., Sep. 2004. ISBN: **978-0-596-00743-0** (siehe S. **154**).
- [2] Alfredo Deza und Noah Gift. *Testing In Python*. San Francisco, CA, USA: Pragmatic AI Labs, Feb. 2020. ISBN: **979-8-6169-6064-1** (siehe S. **153**).
- [3] "Doctest – Test Interactive Python Examples". In: *Python 3 Documentation. The Python Standard Library*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: <https://docs.python.org/3/library/doctest.html> (besucht am 2024-11-07) (siehe S. **153**).
- [4] "[fractions](#) – Rational Numbers". In: *Python 3 Documentation. The Python Standard Library*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: <https://docs.python.org/3/library/fractions.html> (besucht am 2024-12-12) (siehe S. **5–16**).
- [5] David Goodger und Guido van Rossum. *Docstring Conventions*. Python Enhancement Proposal (PEP) 257. Beaverton, OR, USA: Python Software Foundation (PSF), 29. Mai–13. Juni 2001. URL: <https://peps.python.org/pep-0257> (besucht am 2024-07-27) (siehe S. **153**).
- [6] John Hunt. *A Beginners Guide to Python 3 Programming*. 2. Aufl. Undergraduate Topics in Computer Science (UTICS). Cham, Switzerland: Springer, 2023. ISBN: **978-3-031-35121-1**. doi:10.1007/978-3-031-35122-8 (siehe S. **153**).
- [7] Holger Krekel und pytest-Dev Team. "How to Run Doctests". In: *pytest Documentation*. Release 8.4. Freiburg, Baden-Württemberg, Germany: merlinux GmbH. Kap. 2.8, S. 65–69. URL: <https://docs.pytest.org/en/stable/how-to/doctest.html> (besucht am 2024-11-07) (siehe S. **153**).
- [8] Holger Krekel und pytest-Dev Team. *pytest Documentation*. Release 8.4. Freiburg, Baden-Württemberg, Germany: merlinux GmbH. URL: <https://readthedocs.org/projects/pytest/downloads/pdf/latest> (besucht am 2024-11-07) (siehe S. **153**).
- [9] Łukasz Langa. *Literature Overview for Type Hints*. Python Enhancement Proposal (PEP) 482. Beaverton, OR, USA: Python Software Foundation (PSF), 8. Jan. 2015. URL: <https://peps.python.org/pep-0482> (besucht am 2024-10-09) (siehe S. **154**).
- [10] Kent D. Lee und Steve Hubbard. *Data Structures and Algorithms with Python*. Undergraduate Topics in Computer Science (UTICS). Cham, Switzerland: Springer, 2015. ISBN: **978-3-319-13071-2**. doi:10.1007/978-3-319-13072-9 (siehe S. **153**).
- [11] Jukka Lehtosalo, Ivan Levkivskyi, Jared Hance, Ethan Smith, Guido van Rossum, Jelle „JelleZijlstra“ Zijlstra, Michael J. Sullivan, Shantanu Jain, Xuanda Yang, Jingchen Ye, Nikita Sobolev und Mypy Contributors. *Mypy – Static Typing for Python*. San Francisco, CA, USA: GitHub Inc, 2024. URL: <https://github.com/python/mypy> (besucht am 2024-08-17) (siehe S. **153**).

References II



- [12] Mark Lutz. *Learning Python*. 6. Aufl. Sebastopol, CA, USA: O'Reilly Media, Inc., März 2025. ISBN: **978-1-0981-7130-8** (siehe S. **153**).
- [13] A. Jefferson Offutt. "Unit Testing Versus Integration Testing". In: *Test: Faster, Better, Sooner – IEEE International Test Conference (ITC'1991)*. 26.–30. Okt. 1991, Nashville, TN, USA. Los Alamitos, CA, USA: IEEE Computer Society, 1991. Kap. Paper P2.3, S. 1108–1109. ISSN: **1089-3539**. ISBN: **978-0-8186-9156-0**. doi:**10.1109/TEST.1991.519784** (siehe S. **154**).
- [14] Brian Okken. *Python Testing with pytest*. Flower Mound, TX, USA: Pragmatic Bookshelf by The Pragmatic Programmers, L.L.C., Feb. 2022. ISBN: **978-1-68050-860-4** (siehe S. **153**).
- [15] Michael Olan. "Unit Testing: Test Early, Test Often". *Journal of Computing Sciences in Colleges (JCSC)* 19(2):319–328, Dez. 2003. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: **1937-4771**. doi:**10.5555/948785.948830**. URL: <https://www.researchgate.net/publication/255673967> (besucht am 2025-09-05) (siehe S. **154**).
- [16] Ashwin Pajankar. *Python Unit Test Automation: Automate, Organize, and Execute Unit Tests in Python*. New York, NY, USA: Apress Media, LLC, Dez. 2021. ISBN: **978-1-4842-7854-3** (siehe S. **153**, **154**).
- [17] Yasset Pérez-Riverol, Laurent Gatto, Rui Wang, Timo Sachsenberg, Julian Uszkoreit, Felipe da Veiga Leprevost, Christian Fufezan, Tobias Ternent, Stephen J. Eglén, Daniel S. Katz, Tom J. Pollard, Alexander Konovalov, Robert M. Flight, Kai Blin und Juan Antonio Vizcaíno. "Ten Simple Rules for Taking Advantage of Git and GitHub". *PLOS Computational Biology* 12(7), 14. Juli 2016. San Francisco, CA, USA: Public Library of Science (PLOS). ISSN: **1553-7358**. doi:**10.1371/JOURNAL.PCBI.1004947** (siehe S. **153**).
- [18] Per Runeson. "A Survey of Unit Testing Practices". *IEEE Software* 23(4):22–29, Juli–Aug. 2006. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers (IEEE). ISSN: **0740-7459**. doi:**10.1109/MS.2006.91** (siehe S. **154**).
- [19] Anna Skoulikari. *Learning Git*. Sebastopol, CA, USA: O'Reilly Media, Inc., Mai 2023. ISBN: **978-1-0981-3391-7** (siehe S. **153**).
- [20] Sphinx Developers. "Doc Comments and Docstrings". In: `sphinx.ext.autodoc` – *Include Documentation from Docstrings*. 13. Okt. 2024. URL: <https://www.sphinx-doc.org/en/master/usage/extensions/autodoc.html#doc-comments-and-docstrings> (besucht am 2024-12-12) (siehe S. **57–59**).

References III



- [21] Michael J. Sullivan und Ivan Levkivskiy. *Adding a `Final` Qualifier to `typing`*. Python Enhancement Proposal (PEP) 591. Beaverton, OR, USA: Python Software Foundation (PSF), 15. März 2019. URL: <https://peps.python.org/pep-0591> (besucht am 2024-11-19) (siehe S. 18–31, 57–59).
- [22] *Python 3 Documentation. The Python Standard Library*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: <https://docs.python.org/3/library> (besucht am 2025-04-27).
- [23] George K. Thiruvathukal, Konstantin Läufer und Benjamin Gonzalez. "Unit Testing Considered Useful". *Computing in Science & Engineering* 8(6):76–87, Nov.–Dez. 2006. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers (IEEE). ISSN: 1521-9615. doi:10.1109/MCSE.2006.124. URL: <https://www.researchgate.net/publication/220094077> (besucht am 2024-10-01) (siehe S. 154).
- [24] Mariot Tsitoara. *Beginning Git and GitHub: Version Control, Project Management and Teamwork for the New Developer*. New York, NY, USA: Apress Media, LLC, März 2024. ISBN: 979-8-8688-0215-7 (siehe S. 153, 154).
- [25] Adam Turner, Bénédict Tran, Chris Sewell, François Freitag, Jakob Lykke Andersen, Jean-François B., Stephen Finucane, Takayuki Shimizukawa, Takeshi Komiya und Sphinx Developers. *Sphinx – Create Intelligent and Beautiful Documentation with Ease*. 13. Okt. 2024. URL: <https://www.sphinx-doc.org> (besucht am 2024-12-12) (siehe S. 153).
- [26] Guido van Rossum und David Ascher. *Rich Comparisons*. Python Enhancement Proposal (PEP) 207. Beaverton, OR, USA: Python Software Foundation (PSF), 25. Juli 2000. URL: <https://peps.python.org/pep-0207> (besucht am 2024-12-08) (siehe S. 113–123).
- [27] Guido van Rossum und Łukasz Langa. *Type Hints*. Python Enhancement Proposal (PEP) 484. Beaverton, OR, USA: Python Software Foundation (PSF), 29. Sep. 2014. URL: <https://peps.python.org/pep-0484> (besucht am 2024-08-22) (siehe S. 154).
- [28] Guido van Rossum, Barry Warsaw und Alyssa Coghlan. *Style Guide for Python Code*. Python Enhancement Proposal (PEP) 8. Beaverton, OR, USA: Python Software Foundation (PSF), 5. Juli 2001. URL: <https://peps.python.org/pep-0008> (besucht am 2024-07-27) (siehe S. 57–59, 153).
- [29] Thomas Weise (汤卫思). *Programming with Python*. Hefei, Anhui, China (中国安徽省合肥市): Hefei University (合肥大学), School of Artificial Intelligence and Big Data (人工智能与大数据学院), 2024–2026. URL: <https://thomasweise.github.io/programmingWithPython> (besucht am 2025-01-05) (siehe S. 153).
- [30] Kevin Wilson. *Python Made Easy*. Birmingham, England, UK: Packt Publishing Ltd, Aug. 2024. ISBN: 978-1-83664-615-0 (siehe S. 153).

Glossary (in English) I



denominator The number b of a fraction $\frac{a}{b} \in \mathbb{Q}$ is called the *denominator*.

docstring Docstrings are special string constants in Python that contain documentation for modules or functions⁵. They must be delimited by `"""..."""`^{5,28}.

doctest *doctests* are unit tests in the form of as small pieces of code in the docstrings that look like interactive Python sessions. The first line of a statement in such a Python snippet is indented with Python»> and the following lines by `...`. These snippets can be executed by modules like `doctest`³ or tools such as `pytest`⁷. Their output is the compared to the text following the snippet in the docstring. If the output matches this text, the test succeeds. Otherwise it fails.

Git is a distributed Version Control Systems (VCS) which allows multiple users to work on the same code while preserving the history of the code changes^{19,24}. Learn more at <https://git-scm.com>.

GitHub is a website where software projects can be hosted and managed via the Git VCS^{17,24}. Learn more at <https://github.com>.

Mypy is a static type checking tool for Python¹¹ that makes use of type hints. Learn more at <https://github.com/python/mypy> and in²⁹.

numerator The number a of a fraction $\frac{a}{b} \in \mathbb{Q}$ is called the *numerator*.

pytest is a framework for writing and executing unit tests in Python^{2,8,14,16,30}. Learn more at <https://pytest.org>.

Python The Python programming language^{6,10,12,29}, i.e., what you will learn about in our book²⁹. Learn more at <https://python.org>.

Sphinx Sphinx is a tool for generating software documentation²⁵. It supports Python can use both docstrings and type hints to generate beautiful documents. Learn more at <https://www.sphinx-doc.org>.

Glossary (in English) II



type hint are annotations that help programmers and static code analysis tools such as Mypy to better understand what **type** a variable or function parameter is supposed to be^{9,27}. Python is a dynamically typed programming language where you do not need to specify the type of, e.g., a variable. This creates problems for code analysis, both automated as well as manual: For example, it may not always be clear whether a variable or function parameter should be an integer or floating point number. The annotations allow us to explicitly state which type is expected. They are *ignored* during the program execution. They are a basically a piece of documentation.

unit test Software development is centered around creating the program code of an application, library, or otherwise useful system. A *unit test* is an *additional* code fragment that is not part of that productive code. It exists to execute (a part of) the productive code in a certain scenario (e.g., with specific parameters), to observe the behavior of that code, and to compare whether this behavior meets the specification^{1,13,15,16,18,23}. If not, the unit test fails. The use of unit tests is at least threefold: First, they help us to detect errors in the code. Second, program code is usually not developed only once and, from then on, used without change indefinitely. Instead, programs are often updated, improved, extended, and maintained over a long time. Unit tests can help us to detect whether such changes in the program code, maybe after years, violate the specification or, maybe, cause another, depending, module of the program to violate its specification. Third, they are part of the documentation or even specification of a program.

VCS A *Version Control System* is a software which allows you to manage and preserve the historical development of your program code²⁴. A distributed VCS allows multiple users to work on the same code and upload their changes to the server, which then preserves the change history. The most popular distributed VCS is Git.

Q the set of the rational numbers, i.e., the set of all numbers that can be the result of $\frac{a}{b}$ with $a, b \in \mathbb{Z}$ and $b \neq 0$. a is called the numerator and b is called the denominator. It holds that $\mathbb{Z} \subset \mathbb{Q}$ and $\mathbb{Q} \subset \mathbb{R}$.

R the set of the real numbers.

Z the set of the integers numbers including positive and negative numbers and 0, i.e., $\dots, -3, -2, -1, 0, 1, 2, 3, \dots$, and so on. It holds that $\mathbb{Z} \subset \mathbb{R}$.