合肥大學
HEFEI UNIVERSITY

Programming with Python
47. Zwischenspiel: Debugger

Thomas Weise (汤卫思)
tweise@hfuu.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

Version: 2026-11-09

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-Kode 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

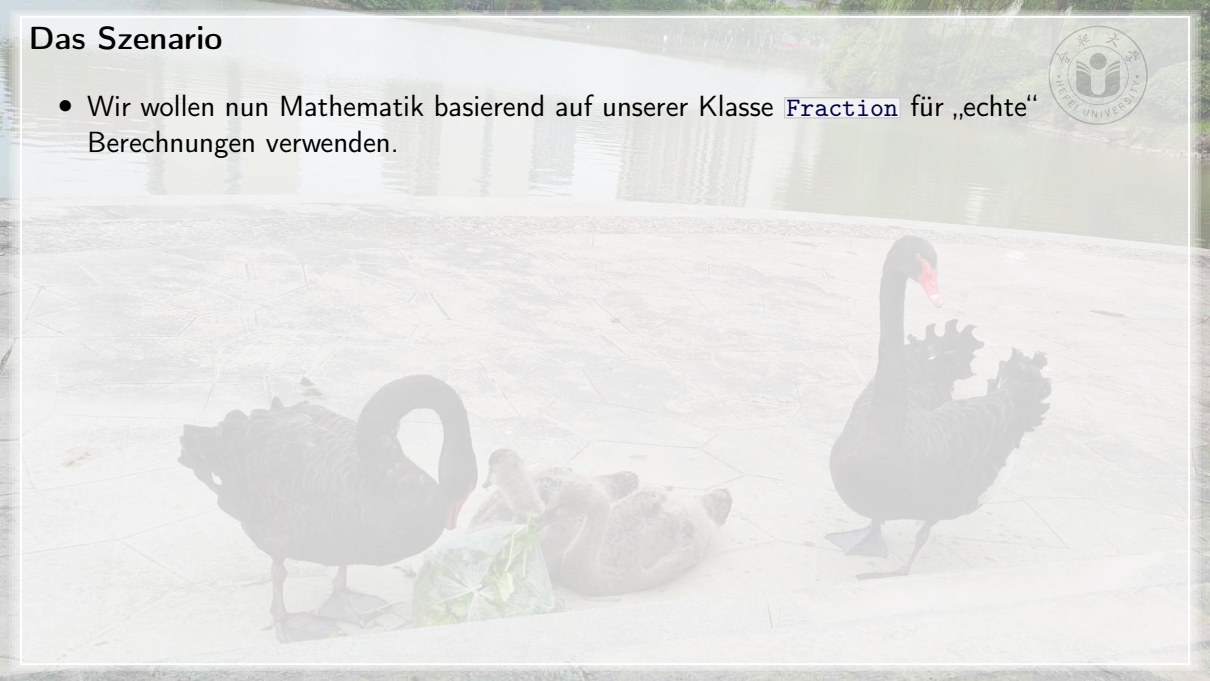


1. Einleitung
2. Die Implementierung
3. Debugging
4. Reparierte Methode
5. Zusammenfassung



Einleitung

Das Szenario



- Wir wollen nun Mathematik basierend auf unserer Klasse `Fraction` für „echte“ Berechnungen verwenden.

Das Szenario

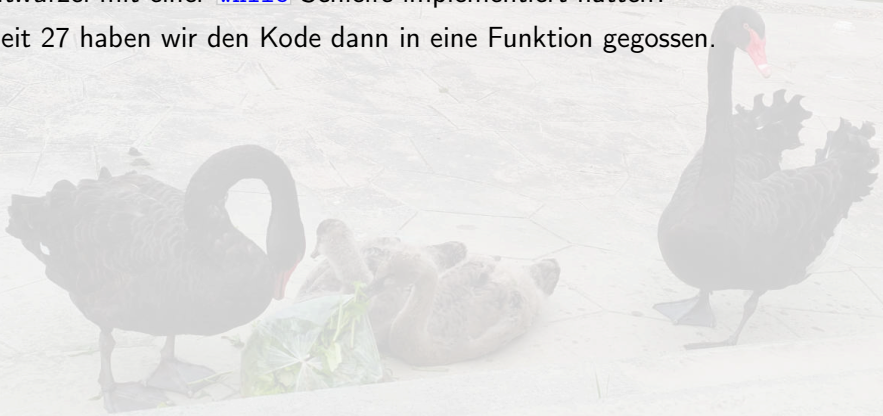


- Wir wollen nun Mathematik basierend auf unserer Klasse `Fraction` für „echte“ Berechnungen verwenden.
- Erinnern Sie sich an Einheit 25, als wie Heron's Methode zum berechnen der Quadratwurzel mit einer `while`-Schleife implementiert hatten?

Das Szenario



- Wir wollen nun Mathematik basierend auf unserer Klasse `Fraction` für „echte“ Berechnungen verwenden.
- Erinnern Sie sich an Einheit 25, als wie Heron's Methode zum berechnen der Quadratwurzel mit einer `while`-Schleife implementiert hatten?
- In Einheit 27 haben wir den Kode dann in eine Funktion gegossen.



Das Szenario



- Wir wollen nun Mathematik basierend auf unserer Klasse `Fraction` für „echte“ Berechnungen verwenden.
- Erinnern Sie sich an Einheit 25, als wie Heron's Methode zum berechnen der Quadratwurzel mit einer `while`-Schleife implementiert hatten?
- In Einheit 27 haben wir den Kode dann in eine Funktion gegossen.
- Wenn wir uns diese Funktion wieder angucken, dann stellen wir fest, dass sie die Quadratwurzel nur mit Vergleichen, Addition, Multiplikation, und Division implementiert.

Das Szenario



- Wir wollen nun Mathematik basierend auf unserer Klasse `Fraction` für „echte“ Berechnungen verwenden.
- Erinnern Sie sich an Einheit 25, als wie Heron's Methode zum berechnen der Quadratwurzel mit einer `while`-Schleife implementiert hatten?
- In Einheit 27 haben wir den Kode dann in eine Funktion gegossen.
- Wenn wir uns diese Funktion wieder angucken, dann stellen wir fest, dass sie die Quadratwurzel nur mit Vergleichen, Addition, Multiplikation, und Division implementiert.
- Diese Operationen gibt es auch für unsere Klasse `Fractions`!

Das Szenario



- Wir wollen nun Mathematik basierend auf unserer Klasse `Fraction` für „echte“ Berechnungen verwenden.
- Erinnern Sie sich an Einheit 25, als wie Heron's Methode zum berechnen der Quadratwurzel mit einer `while`-Schleife implementiert hatten?
- In Einheit 27 haben wir den Kode dann in eine Funktion gegossen.
- Wenn wir uns diese Funktion wieder angucken, dann stellen wir fest, dass sie die Quadratwurzel nur mit Vergleichen, Addition, Multiplikation, und Division implementiert.
- Diese Operationen gibt es auch für unsere Klasse `Fractions`!
- Das heist, wir könnten die Quadratwurzel einer Zahl beliebig genau berechnen!

Das Szenario



- Wir wollen nun Mathematik basierend auf unserer Klasse `Fraction` für „echte“ Berechnungen verwenden.
- Erinnern Sie sich an Einheit 25, als wie Heron's Methode zum berechnen der Quadratwurzel mit einer `while`-Schleife implementiert hatten?
- In Einheit 27 haben wir den Kode dann in eine Funktion gegossen.
- Wenn wir uns diese Funktion wieder angucken, dann stellen wir fest, dass sie die Quadratwurzel nur mit Vergleichen, Addition, Multiplikation, und Division implementiert.
- Diese Operationen gibt es auch für unsere Klasse `Fractions`!
- Das heist, wir könnten die Quadratwurzel einer Zahl beliebig genau berechnen!
- Nun, wir müssten eine Art Abbruchkriterion hinzufügen, aber abgesehen davon...

Das Szenario



- Wir wollen nun Mathematik basierend auf unserer Klasse `Fraction` für „echte“ Berechnungen verwenden.
- Erinnern Sie sich an Einheit 25, als wie Heron's Methode zum berechnen der Quadratwurzel mit einer `while`-Schleife implementiert hatten?
- In Einheit 27 haben wir den Kode dann in eine Funktion gegossen.
- Wenn wir uns diese Funktion wieder angucken, dann stellen wir fest, dass sie die Quadratwurzel nur mit Vergleichen, Addition, Multiplikation, und Division implementiert.
- Diese Operationen gibt es auch für unsere Klasse `Fractions`!
- Das heist, wir könnten die Quadratwurzel einer Zahl beliebig genau berechnen!
- Nun, wir müssten eine Art Abbruchkriterion hinzufügen, aber abgesehen davon...
- ...Wir könnten nun $\sqrt{2}$ auf 700 Ziffern genau berechnen!

Die Anforderung

- Wobei ... bisher haben wir noch keine Möglichkeit, diese Ziffern auch auszugeben.

Die Anforderung



- Wobei ... bisher haben wir noch keine Möglichkeit, diese Ziffern auch auszugeben.
- Unsere `__str__`-Methode liefert Textrepräsentationen von Brüchen in der Form `"a/b"`.

Die Anforderung



- Wobei ... bisher haben wir noch keine Möglichkeit, diese Ziffern auch auszugeben.
- Unsere `__str__`-Methode liefert Textrepräsentationen von Brüchen in der Form `"a/b"`.
- Wenn wir einen Bruch, der $\sqrt{2}$ annähert, berechnet, dann könnte der z. B. als $\frac{6369051672525773}{4503599627370496}$ ausgegeben werden.

Die Anforderung



- Wobei ... bisher haben wir noch keine Möglichkeit, diese Ziffern auch auszugeben.
- Unsere `__str__`-Methode liefert Textrepräsentationen von Brüchen in der Form `"a/b"`.
- Wenn wir einen Bruch, der $\sqrt{2}$ annähert, berechnet, dann könnte der z. B. als $\frac{6369051672525773}{4503599627370496}$ ausgegeben werden.
- Wir hätten aber lieber 1.4142135623730951.

Die Anforderung



- Wobei ... bisher haben wir noch keine Möglichkeit, diese Ziffern auch auszugeben.
- Unsere `__str__`-Methode liefert Textreprästentationen von Brüchen in der Form `"a/b"`.
- Wenn wir einen Bruch, der $\sqrt{2}$ annähert, berechnet, dann könnte der z. B. als $\frac{6369051672525773}{4503599627370496}$ ausgegeben werden.
- Wir hätten aber lieber 1.4142135623730951.
- Also zuerst müssen wir eine Methode `decimal_str` implementieren, die einen Bruch in so einen Ziffern-basierten String umrechnet.

Die Anforderung



- Wobei ... bisher haben wir noch keine Möglichkeit, diese Ziffern auch auszugeben.
- Unsere `__str__`-Methode liefert Textreprästentationen von Brüchen in der Form `"a/b"`.
- Wenn wir einen Bruch, der $\sqrt{2}$ annähert, berechnet, dann könnte der z. B. als $\frac{6369051672525773}{4503599627370496}$ ausgegeben werden.
- Wir hätten aber lieber 1.4142135623730951.
- Also zuerst müssen wir eine Methode `decimal_str` implementieren, die einen Bruch in so einen Ziffern-basierten String umrechnet.
- Da einige Brüche wie $\frac{1}{3}$ und $\frac{1}{7}$ unendlich lange dezimale Repräsentationen haben, braucht unsere Funktion einen Parameter `max_frac`, der die maximale Anzahl der Dezimalstellen angibt.

Die Anforderung



- Wobei ... bisher haben wir noch keine Möglichkeit, diese Ziffern auch auszugeben.
- Unsere `__str__`-Methode liefert Textreprästentationen von Brüchen in der Form `"a/b"`.
- Wenn wir einen Bruch, der $\sqrt{2}$ annähert, berechnet, dann könnte der z. B. als $\frac{6369051672525773}{4503599627370496}$ ausgegeben werden.
- Wir hätten aber lieber 1.4142135623730951.
- Also zuerst müssen wir eine Methode `decimal_str` implementieren, die einen Bruch in so einen Ziffern-basierten String umrechnet.
- Da einige Brüche wie $\frac{1}{3}$ und $\frac{1}{7}$ unendlich lange dezimale Repräsentationen haben, braucht unsere Funktion einen Parameter `max_frac`, der die maximale Anzahl der Dezimalstellen angibt.
- Wir nehmen 100 als Default-Wert.

Die Anforderung



- Wobei ... bisher haben wir noch keine Möglichkeit, diese Ziffern auch auszugeben.
- Unsere `__str__`-Methode liefert Textreprästentationen von Brüchen in der Form `"a/b"`.
- Wenn wir einen Bruch, der $\sqrt{2}$ annähert, berechnet, dann könnte der z. B. als $\frac{6369051672525773}{4503599627370496}$ ausgegeben werden.
- Wir hätten aber lieber 1.4142135623730951.
- Also zuerst müssen wir eine Methode `decimal_str` implementieren, die einen Bruch in so einen Ziffern-basierten String umrechnet.
- Da einige Brüche wie $\frac{1}{3}$ und $\frac{1}{7}$ unendlich lange dezimale Repräsentationen haben, braucht unsere Funktion einen Parameter `max_frac`, der die maximale Anzahl der Dezimalstellen angibt.
- Wir nehmen 100 als Default-Wert.
- OK, implementieren wir das mal.

Die Implementierung



Fraction: decimal_str

- In Datei `fraction_decimal_str_err.py` implementieren wir unsere neue Variante der Klasse `Fraction` mit dieser Funktion.

```python
def decimal_str(self, max_frac: int = 100) -> str:
    """
    Convert the fraction to decimal string.

    :param max_frac: the maximum number of fractional digits
    :return: the string

    >>> Fraction(124, 2).decimal_str()
    '62'
    >>> Fraction(1, 2).decimal_str()
    '0.5'
    >>> Fraction(1, 3).decimal_str(10)
    '0.3333333333'
    >>> Fraction(-101001, 100000000).decimal_str()
    '-0.00101001'
    >>> Fraction(1235, 1000).decimal_str(2)
    '1.24'
    >>> Fraction(99995, 100000).decimal_str(5)
    '0.99995'
    >>> Fraction(91995, 100000).decimal_str(3)
    '0.92'
    >>> Fraction(99995, 100000).decimal_str(4)
    '1'
    """
    a: int = self.a  # Get the numerator.
    if a == 0:  # If the fraction is 0, we return 0.
        return "0"
    negative: Final[bool] = a < 0  # Get the sign of the fraction.
    a = abs(a)  # Make sure that `a` is now positive.
    b: Final[int] = self.b  # Get the denominator.

    digits: Final[list] = []  # A list for collecting digits.
    while (a != 0) and (len(digits) <= max_frac):  # Create digits.
        digits.append(a // b)  # Add the current digit.
        a = 10 * (a % b)  # Ten times the remainder -> next digit.

    if (a // b) >= 5:  # Do we need to round up?
        digits[-1] += 1  # Round up by incrementing last digit.

    if len(digits) <= 1:  # Do we only have an integer part?
        return str((-1 if negative else 1) * digits[0])

    digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
    if negative:  # Do we need to restore the sign?
        digits.insert(0, "-")  # Insert the sign at the beginning.
    return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- In Datei `fraction_decimal_str_err.py` implementieren wir unsere neue Variante der Klasse `Fraction` mit dieser Funktion.
- Hier zeigen wir nur die neue Methode `decimal_str`.

```python
    def decimal_str(self, max_frac: int = 100) -> str:
        """
        Convert the fraction to decimal string.

        :param max_frac: the maximum number of fractional digits
        :return: the string

        >>> Fraction(124, 2).decimal_str()
        '62'
        >>> Fraction(1, 2).decimal_str()
        '0.5'
        >>> Fraction(1, 3).decimal_str(10)
        '0.3333333333'
        >>> Fraction(-101001, 100000000).decimal_str()
        '-0.00101001'
        >>> Fraction(1235, 1000).decimal_str(2)
        '1.24'
        >>> Fraction(99995, 100000).decimal_str(5)
        '0.99995'
        >>> Fraction(91995, 100000).decimal_str(3)
        '0.92'
        >>> Fraction(99995, 100000).decimal_str(4)
        '1'
        """
        a: int = self.a  # Get the numerator.
        if a == 0:  # If the fraction is 0, we return 0.
            return "0"
        negative: Final[bool] = a < 0  # Get the sign of the fraction.
        a = abs(a)  # Make sure that `a` is now positive.
        b: Final[int] = self.b  # Get the denominator.

        digits: Final[list] = []  # A list for collecting digits.
        while (a != 0) and (len(digits) <= max_frac):  # Create digits.
            digits.append(a // b)  # Add the current digit.
            a = 10 * (a % b)  # Ten times the remainder -> next digit.

        if (a // b) >= 5:  # Do we need to round up?
            digits[-1] += 1  # Round up by incrementing last digit.

        if len(digits) <= 1:  # Do we only have an integer part?
            return str((-1 if negative else 1) * digits[0])

        digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
        if negative:  # Do we need to restore the sign?
            digits.insert(0, "-")  # Insert the sign at the beginning.
        return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- In Datei `fraction_decimal_str_err.py` implementieren wir unsere neue Variante der Klasse `Fraction` mit dieser Funktion.
- Hier zeigen wir nur die neue Methode `decimal_str`.
- Wir beginnen unsere Methode `decimal_str`, in dem wir erst den Zähler in eine Variable `a` kopieren.

```
>>> Fraction(124, 2).decimal_str()
'62'
>>> Fraction(1, 2).decimal_str()
'0.5'
>>> Fraction(1, 3).decimal_str(10)
'0.3333333333'
>>> Fraction(-101001, 100000000).decimal_str()
'-0.00101001'
>>> Fraction(1235, 1000).decimal_str(2)
'1.24'
>>> Fraction(99995, 100000).decimal_str(5)
'0.99995'
>>> Fraction(91995, 100000).decimal_str(3)
'0.92'
>>> Fraction(99995, 100000).decimal_str(4)
'1'
"""
a: int = self.a  # Get the numerator.
if a == 0:  # If the fraction is 0, we return 0.
    return "0"
negative: Final[bool] = a < 0  # Get the sign of the fraction.
a = abs(a)  # Make sure that `a` is now positive.
b: Final[int] = self.b  # Get the denominator.

digits: Final[list] = []  # A list for collecting digits.
while (a != 0) and (len(digits) <= max_frac):  # Create digits.
    digits.append(a // b)  # Add the current digit.
    a = 10 * (a % b)  # Ten times the remainder -> next digit.

if (a // b) >= 5:  # Do we need to round up?
    digits[-1] += 1  # Round up by incrementing last digit.

if len(digits) <= 1:  # Do we only have an integer part?
    return str((-1 if negative else 1) * digits[0])

digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
if negative:  # Do we need to restore the sign?
    digits.insert(0, "-")  # Insert the sign at the beginning.
return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- In Datei `fraction_decimal_str_err.py` implementieren wir unsere neue Variante der Klasse `Fraction` mit dieser Funktion.
- Hier zeigen wir nur die neue Methode `decimal_str`.
- Wir beginnen unsere Methode `decimal_str`, in dem wir erst den Zähler in eine Variable `a` kopieren.
- Wenn `a == 0`, dann ist der Bruch 0 und wir liefern direkt `"0"` zurück.

```
        >>> Fraction(124, 2).decimal_str()
        '62'
        >>> Fraction(1, 2).decimal_str()
        '0.5'
        >>> Fraction(1, 3).decimal_str(10)
        '0.3333333333'
        >>> Fraction(-101001, 100000000).decimal_str()
        '-0.00101001'
        >>> Fraction(1235, 1000).decimal_str(2)
        '1.24'
        >>> Fraction(99995, 100000).decimal_str(5)
        '0.99995'
        >>> Fraction(91995, 100000).decimal_str(3)
        '0.92'
        >>> Fraction(99995, 100000).decimal_str(4)
        '1'
        """
        a: int = self.a  # Get the numerator.
        if a == 0:  # If the fraction is 0, we return 0.
            return "0"
        negative: Final[bool] = a < 0  # Get the sign of the fraction.
        a = abs(a)  # Make sure that `a` is now positive.
        b: Final[int] = self.b  # Get the denominator.

        digits: Final[list] = []  # A list for collecting digits.
        while (a != 0) and (len(digits) <= max_frac):  # Create digits.
            digits.append(a // b)  # Add the current digit.
            a = 10 * (a % b)  # Ten times the remainder -> next digit.

        if (a // b) >= 5:  # Do we need to round up?
            digits[-1] += 1  # Round up by incrementing last digit.

        if len(digits) <= 1:  # Do we only have an integer part?
            return str((-1 if negative else 1) * digits[0])

        digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
        if negative:  # Do we need to restore the sign?
            digits.insert(0, "-")  # Insert the sign at the beginning.
        return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- In Datei `fraction_decimal_str_err.py` implementieren wir unsere neue Variante der Klasse `Fraction` mit dieser Funktion.
- Hier zeigen wir nur die neue Methode `decimal_str`.
- Wir beginnen unsere Methode `decimal_str`, in dem wir erst den Zähler in eine Variable `a` kopieren.
- Wenn `a == 0`, dann ist der Bruch 0 und wir liefern direkt `"0"` zurück.
- Sonst prüfen wir, ob der Bruch negativ ist.

```
>>> Fraction(124, 2).decimal_str()
'62'
>>> Fraction(1, 2).decimal_str()
'0.5'
>>> Fraction(1, 3).decimal_str(10)
'0.3333333333'
>>> Fraction(-101001, 100000000).decimal_str()
'-0.00101001'
>>> Fraction(1235, 1000).decimal_str(2)
'1.24'
>>> Fraction(99995, 100000).decimal_str(5)
'0.99995'
>>> Fraction(91995, 100000).decimal_str(3)
'0.92'
>>> Fraction(99995, 100000).decimal_str(4)
'1'
"""
a: int = self.a  # Get the numerator.
if a == 0:  # If the fraction is 0, we return 0.
    return "0"
negative: Final[bool] = a < 0  # Get the sign of the fraction.
a = abs(a)  # Make sure that `a` is now positive.
b: Final[int] = self.b  # Get the denominator.

digits: Final[list] = []  # A list for collecting digits.
while (a != 0) and (len(digits) <= max_frac):  # Create digits.
    digits.append(a // b)  # Add the current digit.
    a = 10 * (a % b)  # Ten times the remainder -> next digit.

if (a // b) >= 5:  # Do we need to round up?
    digits[-1] += 1  # Round up by incrementing last digit.

if len(digits) <= 1:  # Do we only have an integer part?
    return str((-1 if negative else 1) * digits[0])

digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
if negative:  # Do we need to restore the sign?
    digits.insert(0, "-")  # Insert the sign at the beginning.
return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- Hier zeigen wir nur die neue Methode `decimal_str`.
- Wir beginnen unsere Methode `decimal_str`, in dem wir erst den Zähler in eine Variable `a` kopieren.
- Wenn `a == 0`, dann ist der Bruch 0 und wir liefern direkt `"0"` zurück.
- Sonst prüfen wir, ob der Bruch negativ ist.
- Die Boolsche Variable `negative` wird auf `True` gesetzt wenn `a < 0` und sonst auf `False`.

```
>>> Fraction(124, 2).decimal_str()
'62'
>>> Fraction(1, 2).decimal_str()
'0.5'
>>> Fraction(1, 3).decimal_str(10)
'0.3333333333'
>>> Fraction(-101001, 100000000).decimal_str()
'-0.00101001'
>>> Fraction(1235, 1000).decimal_str(2)
'1.24'
>>> Fraction(99995, 100000).decimal_str(5)
'0.99995'
>>> Fraction(91995, 100000).decimal_str(3)
'0.92'
>>> Fraction(99995, 100000).decimal_str(4)
'1'
"""
a: int = self.a  # Get the numerator.
if a == 0:  # If the fraction is 0, we return 0.
    return "0"
negative: Final[bool] = a < 0  # Get the sign of the fraction.
a = abs(a)  # Make sure that `a` is now positive.
b: Final[int] = self.b  # Get the denominator.

digits: Final[list] = []  # A list for collecting digits.
while (a != 0) and (len(digits) <= max_frac):  # Create digits.
    digits.append(a // b)  # Add the current digit.
    a = 10 * (a % b)  # Ten times the remainder -> next digit.

if (a // b) >= 5:  # Do we need to round up?
    digits[-1] += 1  # Round up by incrementing last digit.

if len(digits) <= 1:  # Do we only have an integer part?
    return str((-1 if negative else 1) * digits[0])

digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
if negative:  # Do we need to restore the sign?
    digits.insert(0, "-")  # Insert the sign at the beginning.
return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- Wir beginnen unsere Methode `decimal_str`, in dem wir erst den Zähler in eine Variable `a` kopieren.
- Wenn `a == 0`, dann ist der Bruch 0 und wir liefern direkt `"0"` zurück.
- Sonst prüfen wir, ob der Bruch negativ ist.
- Die Boolsche Variable `negative` wird auf `True` gesetzt wenn `a < 0` und sonst auf `False`.
- Wir stellen dann sicher, dass `a` positiv ist, in dem wir es auf `abs(a)` setzen.

```
>>> Fraction(124, 2).decimal_str()
'62'
>>> Fraction(1, 2).decimal_str()
'0.5'
>>> Fraction(1, 3).decimal_str(10)
'0.3333333333'
>>> Fraction(-101001, 100000000).decimal_str()
'-0.00101001'
>>> Fraction(1235, 1000).decimal_str(2)
'1.24'
>>> Fraction(99995, 100000).decimal_str(5)
'0.99995'
>>> Fraction(91995, 100000).decimal_str(3)
'0.92'
>>> Fraction(99995, 100000).decimal_str(4)
'1'
"""
a: int = self.a  # Get the numerator.
if a == 0:  # If the fraction is 0, we return 0.
    return "0"
negative: Final[bool] = a < 0  # Get the sign of the fraction.
a = abs(a)  # Make sure that `a` is now positive.
b: Final[int] = self.b  # Get the denominator.

digits: Final[list] = []  # A list for collecting digits.
while (a != 0) and (len(digits) <= max_frac):  # Create digits.
    digits.append(a // b)  # Add the current digit.
    a = 10 * (a % b)  # Ten times the remainder -> next digit.

if (a // b) >= 5:  # Do we need to round up?
    digits[-1] += 1  # Round up by incrementing last digit.

if len(digits) <= 1:  # Do we only have an integer part?
    return str((-1 if negative else 1) * digits[0])

digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
if negative:  # Do we need to restore the sign?
    digits.insert(0, "-")  # Insert the sign at the beginning.
return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- Wenn `a == 0`, dann ist der Bruch 0 und wir liefern direkt `"0"` zurück.
- Sonst prüfen wir, ob der Bruch negativ ist.
- Die Boolsche Variable `negative` wird auf `True` gesetzt wenn `a < 0` und sonst auf `False`.
- Wir stellen dann sicher, dass `a` positiv ist, in dem wir es auf `abs(a)` setzen.
- Dann kopieren wir auch den Nenner in eine Variable `b`.

```
>>> Fraction(124, 2).decimal_str()
'62'
>>> Fraction(1, 2).decimal_str()
'0.5'
>>> Fraction(1, 3).decimal_str(10)
'0.3333333333'
>>> Fraction(-101001, 100000000).decimal_str()
'-0.00101001'
>>> Fraction(1235, 1000).decimal_str(2)
'1.24'
>>> Fraction(99995, 100000).decimal_str(5)
'0.99995'
>>> Fraction(91995, 100000).decimal_str(3)
'0.92'
>>> Fraction(99995, 100000).decimal_str(4)
'1'
"""
a: int = self.a  # Get the numerator.
if a == 0:  # If the fraction is 0, we return 0.
    return "0"
negative: Final[bool] = a < 0  # Get the sign of the fraction.
a = abs(a)  # Make sure that `a` is now positive.
b: Final[int] = self.b  # Get the denominator.

digits: Final[list] = []  # A list for collecting digits.
while (a != 0) and (len(digits) <= max_frac):  # Create digits.
    digits.append(a // b)  # Add the current digit.
    a = 10 * (a % b)  # Ten times the remainder -> next digit.

if (a // b) >= 5:  # Do we need to round up?
    digits[-1] += 1  # Round up by incrementing last digit.

if len(digits) <= 1:  # Do we only have an integer part?
    return str((-1 if negative else 1) * digits[0])

digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
if negative:  # Do we need to restore the sign?
    digits.insert(0, "-")  # Insert the sign at the beginning.
return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- Sonst prüfen wir, ob der Bruch negativ ist.
- Die Boolsche Variable `negative` wird auf `True` gesetzt wenn `a < 0` und sonst auf `False`.
- Wir stellen dann sicher, dass `a` positiv ist, in dem wir es auf `abs(a)` setzen.
- Dann kopieren wir auch den Nenner in eine Variable `b`.
- Wir werden die lokalen Variablen `negative` und `b` in unserer Methode nicht verändern, also markieren wir sie beide mit dem Type Hint `Final`.

```
>>> Fraction(124, 2).decimal_str()
'62'
>>> Fraction(1, 2).decimal_str()
'0.5'
>>> Fraction(1, 3).decimal_str(10)
'0.3333333333'
>>> Fraction(-101001, 100000000).decimal_str()
'-0.00101001'
>>> Fraction(1235, 1000).decimal_str(2)
'1.24'
>>> Fraction(99995, 100000).decimal_str(5)
'0.99995'
>>> Fraction(91995, 100000).decimal_str(3)
'0.92'
>>> Fraction(99995, 100000).decimal_str(4)
'1'
"""
a: int = self.a  # Get the numerator.
if a == 0:  # If the fraction is 0, we return 0.
    return "0"
negative: Final[bool] = a < 0  # Get the sign of the fraction.
a = abs(a)  # Make sure that `a` is now positive.
b: Final[int] = self.b  # Get the denominator.

digits: Final[list] = []  # A list for collecting digits.
while (a != 0) and (len(digits) <= max_frac):  # Create digits.
    digits.append(a // b)  # Add the current digit.
    a = 10 * (a % b)  # Ten times the remainder -> next digit.

if (a // b) >= 5:  # Do we need to round up?
    digits[-1] += 1  # Round up by incrementing last digit.

if len(digits) <= 1:  # Do we only have an integer part?
    return str((-1 if negative else 1) * digits[0])

digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
if negative:  # Do we need to restore the sign?
    digits.insert(0, "-")  # Insert the sign at the beginning.
return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- Die Boolsche Variable `negative` wird auf `True` gesetzt wenn `a < 0` und sonst auf `False`.
- Wir stellen dann sicher, dass `a` positiv ist, in dem wir es auf `abs(a)` setzen.
- Dann kopieren wir auch den Nenner in eine Variable `b`.
- Wir werden die lokalen Variablen `negative` und `b` in unserer Methode nicht verändern, also markieren wir sie beide mit dem Type Hint `Final`.

```
>>> Fraction(124, 2).decimal_str()
'62'
>>> Fraction(1, 2).decimal_str()
'0.5'
>>> Fraction(1, 3).decimal_str(10)
```

Gute Praxis

Wann immer Sie eine Variable deklarieren, die Sie nicht vor haben zu ändern, markieren Sie diese mit dem Type Hint `Final`[89].

```
if (a // b) >= 5:  # Do we need to round up?
    digits[-1] += 1  # Round up by incrementing last digit.

if len(digits) <= 1:  # Do we only have an integer part?
    return str((-1 if negative else 1) * digits[0])

digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
if negative:  # Do we need to restore the sign?
    digits.insert(0, "-")  # Insert the sign at the beginning.
return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- Die Boolsche Variable `negative` wird auf `True` gesetzt wenn `a < 0` und sonst auf `False`.
- Wir stellen dann sicher, dass `a` positiv ist, in dem wir es auf `abs(a)` setzen.
- Dann kopieren wir auch den Nenner in eine Variable `b`.
- Wir werden die lokalen Variablen `negative` und `b` in unserer Methode nicht verändern, also markieren wir sie beide mit dem Type Hint `Final`.

```
>>> Fraction(124, 2).decimal_str()
'62'
>>> Fraction(1, 2).decimal_str()
'0.5'
>>> Fraction(1, 3).decimal_str(10)
```

Gute Praxis

Wann immer Sie eine Variable deklarieren, die Sie nicht vor haben zu ändern, markieren Sie diese mit dem Type Hint `Final`[89]. Auf der einen Seite demonstriert das klar die Absicht „das hier ändert sich nicht“ jedem Leser des Kodes.

```
if (a // b) >= 5:  # Do we need to round up?
    digits[-1] += 1  # Round up by incrementing last digit.

if len(digits) <= 1:  # Do we only have an integer part?
    return str((-1 if negative else 1) * digits[0])

digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
if negative:  # Do we need to restore the sign?
    digits.insert(0, "-")  # Insert the sign at the beginning.
return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- Die Boolsche Variable `negative` wird auf `True` gesetzt wenn `a < 0` und sonst auf `False`.
- Wir stellen dann sicher, dass `a` positiv ist, in dem wir es auf `abs(a)` setzen.
- Dann kopieren wir auch den Nenner in eine Variable `b`.
- Wir werden die lokalen Variablen `negative` und `b` in unserer Methode nicht verändern, also markieren wir sie beide mit dem Type Hint `Final`.

```
>>> Fraction(124, 2).decimal_str()
'62'
>>> Fraction(1, 2).decimal_str()
'0.5'
>>> Fraction(1, 3).decimal_str(10)
```

Gute Praxis

Wann immer Sie eine Variable deklarieren, die Sie nicht vor haben zu ändern, markieren Sie diese mit dem Type Hint `Final`[89]. Auf der einen Seite demonstriert das klar die Absicht „das hier ändert sich nicht“ jedem Leser des Kodes. Auf der anderen Seite können Sie dann später mit Werkzeugen wie Mypy feststellen, ob Sie sie nicht doch (aus Versehen?) irgendwo ändern...

```
    if (a // b) >= 5:  # Do we need to round up?
        digits[-1] += 1  # Round up by incrementing last digit.

    if len(digits) <= 1:  # Do we only have an integer part?
        return str((-1 if negative else 1) * digits[0])

    digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
    if negative:  # Do we need to restore the sign?
        digits.insert(0, "-")  # Insert the sign at the beginning.
    return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- Die Boolsche Variable `negative` wird auf `True` gesetzt wenn `a < 0` und sonst auf `False`.
- Wir stellen dann sicher, dass `a` positiv ist, in dem wir es auf `abs(a)` setzen.
- Dann kopieren wir auch den Nenner in eine Variable `b`.
- Wir werden die lokalen Variablen `negative` und `b` in unserer Methode nicht verändern, also markieren wir sie beide mit dem Type Hint `Final`.
- In einer `while`-Schleife füllen wir nun eine Liste `digits` mit den Ziffern die den Bruch repräsentieren.

```
>>> Fraction(124, 2).decimal_str()
'62'
>>> Fraction(1, 2).decimal_str()
'0.5'
>>> Fraction(1, 3).decimal_str(10)
'0.3333333333'
>>> Fraction(-101001, 100000000).decimal_str()
'-0.00101001'
>>> Fraction(1235, 1000).decimal_str(2)
'1.24'
>>> Fraction(99995, 100000).decimal_str(5)
'0.99995'
>>> Fraction(91995, 100000).decimal_str(3)
'0.92'
>>> Fraction(99995, 100000).decimal_str(4)
'1'
"""
a: int = self.a  # Get the numerator.
if a == 0:  # If the fraction is 0, we return 0.
    return "0"
negative: Final[bool] = a < 0  # Get the sign of the fraction.
a = abs(a)  # Make sure that `a` is now positive.
b: Final[int] = self.b  # Get the denominator.

digits: Final[list] = []  # A list for collecting digits.
while (a != 0) and (len(digits) <= max_frac):  # Create digits.
    digits.append(a // b)  # Add the current digit.
    a = 10 * (a % b)  # Ten times the remainder -> next digit.

if (a // b) >= 5:  # Do we need to round up?
    digits[-1] += 1  # Round up by incrementing last digit.

if len(digits) <= 1:  # Do we only have an integer part?
    return str((-1 if negative else 1) * digits[0])

digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
if negative:  # Do we need to restore the sign?
    digits.insert(0, "-")  # Insert the sign at the beginning.
return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- Wir stellen dann sicher, dass `a` positiv ist, in dem wir es auf `abs(a)` setzen.
- Dann kopieren wir auch den Nenner in eine Variable `b`.
- Wir werden die lokalen Variablen `negative` und `b` in unserer Methode nicht verändern, also markieren wir sie beide mit dem Type Hint `Final`.
- In einer `while`-Schleife füllen wir nun eine Liste `digits` mit den Ziffern die den Bruch repräsentieren.
- Die Schleife wird fortgesetzt bis entweder `a == 0` oder bis unsere Liste `digits` `max_frac` Nachkommastellen beinhaltet.

```
>>> Fraction(124, 2).decimal_str()
'62'
>>> Fraction(1, 2).decimal_str()
'0.5'
>>> Fraction(1, 3).decimal_str(10)
'0.3333333333'
>>> Fraction(-101001, 100000000).decimal_str()
'-0.00101001'
>>> Fraction(1235, 1000).decimal_str(2)
'1.24'
>>> Fraction(99995, 100000).decimal_str(5)
'0.99995'
>>> Fraction(91995, 100000).decimal_str(3)
'0.92'
>>> Fraction(99995, 100000).decimal_str(4)
'1'
"""
a: int = self.a  # Get the numerator.
if a == 0:  # If the fraction is 0, we return 0.
    return "0"
negative: Final[bool] = a < 0  # Get the sign of the fraction.
a = abs(a)  # Make sure that `a` is now positive.
b: Final[int] = self.b  # Get the denominator.

digits: Final[list] = []  # A list for collecting digits.
while (a != 0) and (len(digits) <= max_frac):  # Create digits.
    digits.append(a // b)  # Add the current digit.
    a = 10 * (a % b)  # Ten times the remainder -> next digit.

if (a // b) >= 5:  # Do we need to round up?
    digits[-1] += 1  # Round up by incrementing last digit.

if len(digits) <= 1:  # Do we only have an integer part?
    return str((-1 if negative else 1) * digits[0])

digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
if negative:  # Do we need to restore the sign?
    digits.insert(0, "-")  # Insert the sign at the beginning.
return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- Dann kopieren wir auch den Nenner in eine Variable `b`.
- Wir werden die lokalen Variablen `negative` und `b` in unserer Methode nicht verändern, also markieren wir sie beide mit dem Type Hint `Final`.
- In einer `while`-Schleife füllen wir nun eine Liste `digits` mit den Ziffern die den Bruch repräsentieren.
- Die Schleife wird fortgesetzt bis entweder `a == 0` oder bis unsere Liste `digits` `max_frac` Nachkommastellen beinhaltet.
- Nehmen wir mal an, unser Bruch wäre $-\frac{179}{16}$.

```
>>> Fraction(124, 2).decimal_str()
'62'
>>> Fraction(1, 2).decimal_str()
'0.5'
>>> Fraction(1, 3).decimal_str(10)
'0.3333333333'
>>> Fraction(-101001, 100000000).decimal_str()
'-0.00101001'
>>> Fraction(1235, 1000).decimal_str(2)
'1.24'
>>> Fraction(99995, 100000).decimal_str(5)
'0.99995'
>>> Fraction(91995, 100000).decimal_str(3)
'0.92'
>>> Fraction(99995, 100000).decimal_str(4)
'1'
"""
a: int = self.a  # Get the numerator.
if a == 0:  # If the fraction is 0, we return 0.
    return "0"
negative: Final[bool] = a < 0  # Get the sign of the fraction.
a = abs(a)  # Make sure that `a` is now positive.
b: Final[int] = self.b  # Get the denominator.

digits: Final[list] = []  # A list for collecting digits.
while (a != 0) and (len(digits) <= max_frac):  # Create digits.
    digits.append(a // b)  # Add the current digit.
    a = 10 * (a % b)  # Ten times the remainder -> next digit.

if (a // b) >= 5:  # Do we need to round up?
    digits[-1] += 1  # Round up by incrementing last digit.

if len(digits) <= 1:  # Do we only have an integer part?
    return str((-1 if negative else 1) * digits[0])

digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
if negative:  # Do we need to restore the sign?
    digits.insert(0, "-")  # Insert the sign at the beginning.
return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- Wir werden die lokalen Variablen `negative` und `b` in unserer Methode nicht verändern, also markieren wir sie beide mit dem Type Hint `Final`.
- In einer `while`-Schleife füllen wir nun eine Liste `digits` mit den Ziffern die den Bruch repräsentieren.
- Die Schleife wird fortgesetzt bis entweder `a == 0` oder bis unsere Liste `digits` `max_frac` Nachkommastellen beinhaltet.
- Nehmen wir mal an, unser Bruch wäre $-\frac{179}{16}$.
- Dann ist `negative == True`, `a = 179` und `b = 16`.

```
>>> Fraction(124, 2).decimal_str()
'62'
>>> Fraction(1, 2).decimal_str()
'0.5'
>>> Fraction(1, 3).decimal_str(10)
'0.3333333333'
>>> Fraction(-101001, 100000000).decimal_str()
'-0.00101001'
>>> Fraction(1235, 1000).decimal_str(2)
'1.24'
>>> Fraction(99995, 100000).decimal_str(5)
'0.99995'
>>> Fraction(91995, 100000).decimal_str(3)
'0.92'
>>> Fraction(99995, 100000).decimal_str(4)
'1'
"""
a: int = self.a  # Get the numerator.
if a == 0:  # If the fraction is 0, we return 0.
    return "0"
negative: Final[bool] = a < 0  # Get the sign of the fraction.
a = abs(a)  # Make sure that `a` is now positive.
b: Final[int] = self.b  # Get the denominator.

digits: Final[list] = []  # A list for collecting digits.
while (a != 0) and (len(digits) <= max_frac):  # Create digits.
    digits.append(a // b)  # Add the current digit.
    a = 10 * (a % b)  # Ten times the remainder -> next digit.

if (a // b) >= 5:  # Do we need to round up?
    digits[-1] += 1  # Round up by incrementing last digit.

if len(digits) <= 1:  # Do we only have an integer part?
    return str((-1 if negative else 1) * digits[0])

digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
if negative:  # Do we need to restore the sign?
    digits.insert(0, "-")  # Insert the sign at the beginning.
return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- In einer `while`-Schleife füllen wir nun eine Liste `digits` mit den Ziffern die den Bruch repräsentieren.
- Die Schleife wird fortgesetzt bis entweder `a == 0` oder bis unsere Liste `digits` `max_frac` Nachkommastellen beinhaltet.
- Nehmen wir mal an, unser Bruch wäre $-\frac{179}{16}$.
- Dann ist `negative == True`, `a = 179` und `b = 16`.
- Im Schleifenkörper hängen wir das Ergebnis der Ganzzahldivision von `a` durch `b`, also `a // b`, an die Liste `digits` an.

```
>>> Fraction(124, 2).decimal_str()
'62'
>>> Fraction(1, 2).decimal_str()
'0.5'
>>> Fraction(1, 3).decimal_str(10)
'0.3333333333'
>>> Fraction(-101001, 100000000).decimal_str()
'-0.00101001'
>>> Fraction(1235, 1000).decimal_str(2)
'1.24'
>>> Fraction(99995, 100000).decimal_str(5)
'0.99995'
>>> Fraction(91995, 100000).decimal_str(3)
'0.92'
>>> Fraction(99995, 100000).decimal_str(4)
'1'
"""
a: int = self.a  # Get the numerator.
if a == 0:  # If the fraction is 0, we return 0.
    return "0"
negative: Final[bool] = a < 0  # Get the sign of the fraction.
a = abs(a)  # Make sure that `a` is now positive.
b: Final[int] = self.b  # Get the denominator.

digits: Final[list] = []  # A list for collecting digits.
while (a != 0) and (len(digits) <= max_frac):  # Create digits.
    digits.append(a // b)  # Add the current digit.
    a = 10 * (a % b)  # Ten times the remainder -> next digit.

if (a // b) >= 5:  # Do we need to round up?
    digits[-1] += 1  # Round up by incrementing last digit.

if len(digits) <= 1:  # Do we only have an integer part?
    return str((-1 if negative else 1) * digits[0])

digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
if negative:  # Do we need to restore the sign?
    digits.insert(0, "-")  # Insert the sign at the beginning.
return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- Die Schleife wird fortgesetzt bis entweder `a == 0` oder bis unsere Liste `digits` `max_frac` Nachkommastellen beinhaltet.
- Nehmen wir mal an, unser Bruch wäre $-\frac{179}{16}$.
- Dann ist `negative == True`, `a = 179` und `b = 16`.
- Im Schleifenkörper hängen wir das Ergebnis der Ganzzahldivision von `a` durch `b`, also `a // b`, an die Liste `digits` an.
- In der ersten Iteration gibt uns das `179 // 16`.

```
>>> Fraction(124, 2).decimal_str()
'62'
>>> Fraction(1, 2).decimal_str()
'0.5'
>>> Fraction(1, 3).decimal_str(10)
'0.3333333333'
>>> Fraction(-101001, 100000000).decimal_str()
'-0.00101001'
>>> Fraction(1235, 1000).decimal_str(2)
'1.24'
>>> Fraction(99995, 100000).decimal_str(5)
'0.99995'
>>> Fraction(91995, 100000).decimal_str(3)
'0.92'
>>> Fraction(99995, 100000).decimal_str(4)
'1'
"""
a: int = self.a  # Get the numerator.
if a == 0:  # If the fraction is 0, we return 0.
    return "0"
negative: Final[bool] = a < 0  # Get the sign of the fraction.
a = abs(a)  # Make sure that `a` is now positive.
b: Final[int] = self.b  # Get the denominator.

digits: Final[list] = []  # A list for collecting digits.
while (a != 0) and (len(digits) <= max_frac):  # Create digits.
    digits.append(a // b)  # Add the current digit.
    a = 10 * (a % b)  # Ten times the remainder -> next digit.

if (a // b) >= 5:  # Do we need to round up?
    digits[-1] += 1  # Round up by incrementing last digit.

if len(digits) <= 1:  # Do we only have an integer part?
    return str((-1 if negative else 1) * digits[0])

digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
if negative:  # Do we need to restore the sign?
    digits.insert(0, "-")  # Insert the sign at the beginning.
return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- Nehmen wir mal an, unser Bruch wäre $-\frac{179}{16}$.
- Dann ist `negative == True`, `a = 179` und `b = 16`.
- Im Schleifenkörper hängen wir das Ergebnis der Ganzzahldivision von `a` durch `b`, also `a // b`, an die Liste `digits` an.
- In der ersten Iteration gibt uns das `179 // 16`.
- Die erste „Ziffer“, die wir an die Liste `digits` anhängen, ist also `11`.

```
        >>> Fraction(124, 2).decimal_str()
        '62'
        >>> Fraction(1, 2).decimal_str()
        '0.5'
        >>> Fraction(1, 3).decimal_str(10)
        '0.3333333333'
        >>> Fraction(-101001, 100000000).decimal_str()
        '-0.00101001'
        >>> Fraction(1235, 1000).decimal_str(2)
        '1.24'
        >>> Fraction(99995, 100000).decimal_str(5)
        '0.99995'
        >>> Fraction(91995, 100000).decimal_str(3)
        '0.92'
        >>> Fraction(99995, 100000).decimal_str(4)
        '1'
        """
        a: int = self.a  # Get the numerator.
        if a == 0:  # If the fraction is 0, we return 0.
            return "0"
        negative: Final[bool] = a < 0  # Get the sign of the fraction.
        a = abs(a)  # Make sure that `a` is now positive.
        b: Final[int] = self.b  # Get the denominator.

        digits: Final[list] = []  # A list for collecting digits.
        while (a != 0) and (len(digits) <= max_frac):  # Create digits.
            digits.append(a // b)  # Add the current digit.
            a = 10 * (a % b)  # Ten times the remainder -> next digit.

        if (a // b) >= 5:  # Do we need to round up?
            digits[-1] += 1  # Round up by incrementing last digit.

        if len(digits) <= 1:  # Do we only have an integer part?
            return str((-1 if negative else 1) * digits[0])

        digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
        if negative:  # Do we need to restore the sign?
            digits.insert(0, "-")  # Insert the sign at the beginning.
        return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- Dann ist `negative == True`, `a = 179` und `b = 16`.
- Im Schleifenkörper hängen wir das Ergebnis der Ganzzahldivision von `a` durch `b`, also `a // b`, an die Liste `digits` an.
- In der ersten Iteration gibt uns das `179 // 16`.
- Die erste „Ziffer“, die wir an die Liste `digits` anhängen, ist also `11`.
- Das ist der ganzzahlige Teil unseres Bruches und auch die einzige „Ziffer“ größer als 9 die auftauchen kann.

```
>>> Fraction(124, 2).decimal_str()
'62'
>>> Fraction(1, 2).decimal_str()
'0.5'
>>> Fraction(1, 3).decimal_str(10)
'0.3333333333'
>>> Fraction(-101001, 100000000).decimal_str()
'-0.00101001'
>>> Fraction(1235, 1000).decimal_str(2)
'1.24'
>>> Fraction(99995, 100000).decimal_str(5)
'0.99995'
>>> Fraction(91995, 100000).decimal_str(3)
'0.92'
>>> Fraction(99995, 100000).decimal_str(4)
'1'
"""
a: int = self.a  # Get the numerator.
if a == 0:  # If the fraction is 0, we return 0.
    return "0"
negative: Final[bool] = a < 0  # Get the sign of the fraction.
a = abs(a)  # Make sure that `a` is now positive.
b: Final[int] = self.b  # Get the denominator.

digits: Final[list] = []  # A list for collecting digits.
while (a != 0) and (len(digits) <= max_frac):  # Create digits.
    digits.append(a // b)  # Add the current digit.
    a = 10 * (a % b)  # Ten times the remainder -> next digit.

if (a // b) >= 5:  # Do we need to round up?
    digits[-1] += 1  # Round up by incrementing last digit.

if len(digits) <= 1:  # Do we only have an integer part?
    return str((-1 if negative else 1) * digits[0])

digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
if negative:  # Do we need to restore the sign?
    digits.insert(0, "-")  # Insert the sign at the beginning.
return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- Im Schleifenkörper hängen wir das Ergebnis der Ganzzahldivision von `a` durch `b`, also `a // b`, an die Liste `digits` an.
- In der ersten Iteration gibt uns das `179 // 16`.
- Die erste „Ziffer“, die wir an die Liste `digits` anhängen, ist also `11`.
- Das ist der ganzzahlige Teil unseres Bruches und auch die einzige „Ziffer“ größer als 9 die auftauchen kann.
- Jetzt updaten wir `a` zu `10 * (a %b)`.

```
        >>> Fraction(124, 2).decimal_str()
        '62'
        >>> Fraction(1, 2).decimal_str()
        '0.5'
        >>> Fraction(1, 3).decimal_str(10)
        '0.3333333333'
        >>> Fraction(-101001, 100000000).decimal_str()
        '-0.00101001'
        >>> Fraction(1235, 1000).decimal_str(2)
        '1.24'
        >>> Fraction(99995, 100000).decimal_str(5)
        '0.99995'
        >>> Fraction(91995, 100000).decimal_str(3)
        '0.92'
        >>> Fraction(99995, 100000).decimal_str(4)
        '1'
        """
        a: int = self.a  # Get the numerator.
        if a == 0:  # If the fraction is 0, we return 0.
            return "0"
        negative: Final[bool] = a < 0  # Get the sign of the fraction.
        a = abs(a)  # Make sure that `a` is now positive.
        b: Final[int] = self.b  # Get the denominator.

        digits: Final[list] = []  # A list for collecting digits.
        while (a != 0) and (len(digits) <= max_frac):  # Create digits.
            digits.append(a // b)  # Add the current digit.
            a = 10 * (a % b)  # Ten times the remainder -> next digit.

        if (a // b) >= 5:  # Do we need to round up?
            digits[-1] += 1  # Round up by incrementing last digit.

        if len(digits) <= 1:  # Do we only have an integer part?
            return str((-1 if negative else 1) * digits[0])

        digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
        if negative:  # Do we need to restore the sign?
            digits.insert(0, "-")  # Insert the sign at the beginning.
        return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- Im Schleifenkörper hängen wir das Ergebnis der Ganzzahldivision von `a` durch `b`, also `a // b`, an die Liste `digits` an.
- In der ersten Iteration gibt uns das `179 // 16`.
- Die erste „Ziffer“, die wir an die Liste `digits` anhängen, ist also `11`.
- Das ist der ganzzahlige Teil unseres Bruches und auch die einzige „Ziffer“ größer als 9 die auftauchen kann.
- Jetzt updaten wir `a` zu `10 * (a %b)`.
- `%` ist der Rest der Division.

```
        >>> Fraction(124, 2).decimal_str()
        '62'
        >>> Fraction(1, 2).decimal_str()
        '0.5'
        >>> Fraction(1, 3).decimal_str(10)
        '0.3333333333'
        >>> Fraction(-101001, 100000000).decimal_str()
        '-0.00101001'
        >>> Fraction(1235, 1000).decimal_str(2)
        '1.24'
        >>> Fraction(99995, 100000).decimal_str(5)
        '0.99995'
        >>> Fraction(91995, 100000).decimal_str(3)
        '0.92'
        >>> Fraction(99995, 100000).decimal_str(4)
        '1'
        """
        a: int = self.a  # Get the numerator.
        if a == 0:  # If the fraction is 0, we return 0.
            return "0"
        negative: Final[bool] = a < 0  # Get the sign of the fraction.
        a = abs(a)  # Make sure that `a` is now positive.
        b: Final[int] = self.b  # Get the denominator.

        digits: Final[list] = []  # A list for collecting digits.
        while (a != 0) and (len(digits) <= max_frac):  # Create digits.
            digits.append(a // b)  # Add the current digit.
            a = 10 * (a % b)  # Ten times the remainder -> next digit.

        if (a // b) >= 5:  # Do we need to round up?
            digits[-1] += 1  # Round up by incrementing last digit.

        if len(digits) <= 1:  # Do we only have an integer part?
            return str((-1 if negative else 1) * digits[0])

        digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
        if negative:  # Do we need to restore the sign?
            digits.insert(0, "-")  # Insert the sign at the beginning.
        return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- In der ersten Iteration gibt uns das `179 // 16`.
- Die erste „Ziffer“, die wir an die Liste `digits` anhängen, ist also `11`.
- Das ist der ganzzahlige Teil unseres Bruches und auch die einzige „Ziffer“ größer als 9 die auftauchen kann.
- Jetzt updaten wir `a` zu `10 * (a %b)`.
- `%` ist der Rest der Division.
- `a % b` gibt uns den Rest der Division von 179 durch 16, also 3.

```
        >>> Fraction(124, 2).decimal_str()
        '62'
        >>> Fraction(1, 2).decimal_str()
        '0.5'
        >>> Fraction(1, 3).decimal_str(10)
        '0.3333333333'
        >>> Fraction(-101001, 100000000).decimal_str()
        '-0.00101001'
        >>> Fraction(1235, 1000).decimal_str(2)
        '1.24'
        >>> Fraction(99995, 100000).decimal_str(5)
        '0.99995'
        >>> Fraction(91995, 100000).decimal_str(3)
        '0.92'
        >>> Fraction(99995, 100000).decimal_str(4)
        '1'
        """
        a: int = self.a  # Get the numerator.
        if a == 0:  # If the fraction is 0, we return 0.
            return "0"
        negative: Final[bool] = a < 0  # Get the sign of the fraction.
        a = abs(a)  # Make sure that `a` is now positive.
        b: Final[int] = self.b  # Get the denominator.

        digits: Final[list] = []  # A list for collecting digits.
        while (a != 0) and (len(digits) <= max_frac):  # Create digits.
            digits.append(a // b)  # Add the current digit.
            a = 10 * (a % b)  # Ten times the remainder -> next digit.

        if (a // b) >= 5:  # Do we need to round up?
            digits[-1] += 1  # Round up by incrementing last digit.

        if len(digits) <= 1:  # Do we only have an integer part?
            return str((-1 if negative else 1) * digits[0])

        digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
        if negative:  # Do we need to restore the sign?
            digits.insert(0, "-")  # Insert the sign at the beginning.
        return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- Die erste „Ziffer“, die wir an die Liste `digits` anhängen, ist also `11`.
- Das ist der ganzzahlige Teil unseres Bruches und auch die einzige „Ziffer“ größer als 9 die auftauchen kann.
- Jetzt updaten wir `a` zu `10 * (a %b)`.
- `%` ist der Rest der Division.
- `a % b` gibt uns den Rest der Division von 179 durch 16, also 3.
- Wir bekommen also `a = 30`.

```
>>> Fraction(124, 2).decimal_str()
'62'
>>> Fraction(1, 2).decimal_str()
'0.5'
>>> Fraction(1, 3).decimal_str(10)
'0.3333333333'
>>> Fraction(-101001, 100000000).decimal_str()
'-0.00101001'
>>> Fraction(1235, 1000).decimal_str(2)
'1.24'
>>> Fraction(99995, 100000).decimal_str(5)
'0.99995'
>>> Fraction(91995, 100000).decimal_str(3)
'0.92'
>>> Fraction(99995, 100000).decimal_str(4)
'1'
"""
a: int = self.a  # Get the numerator.
if a == 0:  # If the fraction is 0, we return 0.
    return "0"
negative: Final[bool] = a < 0  # Get the sign of the fraction.
a = abs(a)  # Make sure that `a` is now positive.
b: Final[int] = self.b  # Get the denominator.

digits: Final[list] = []  # A list for collecting digits.
while (a != 0) and (len(digits) <= max_frac):  # Create digits.
    digits.append(a // b)  # Add the current digit.
    a = 10 * (a % b)  # Ten times the remainder -> next digit.

if (a // b) >= 5:  # Do we need to round up?
    digits[-1] += 1  # Round up by incrementing last digit.

if len(digits) <= 1:  # Do we only have an integer part?
    return str((-1 if negative else 1) * digits[0])

digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
if negative:  # Do we need to restore the sign?
    digits.insert(0, "-")  # Insert the sign at the beginning.
return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- Das ist der ganzzahlige Teil unseres Bruches und auch die einzige „Ziffer“ größer als 9 die auftauchen kann.
- Jetzt updaten wir `a` zu `10 * (a %b)`.
- `%` ist der Rest der Division.
- `a % b` gibt uns den Rest der Division von 179 durch 16, also 3.
- Wir bekommen also `a = 30`.
- Im zweiten Schleifendurchlauf gibt uns `a // b` also 30 // 16 die Ziffer `1`.

```
>>> Fraction(124, 2).decimal_str()
'62'
>>> Fraction(1, 2).decimal_str()
'0.5'
>>> Fraction(1, 3).decimal_str(10)
'0.3333333333'
>>> Fraction(-101001, 100000000).decimal_str()
'-0.00101001'
>>> Fraction(1235, 1000).decimal_str(2)
'1.24'
>>> Fraction(99995, 100000).decimal_str(5)
'0.99995'
>>> Fraction(91995, 100000).decimal_str(3)
'0.92'
>>> Fraction(99995, 100000).decimal_str(4)
'1'
"""
a: int = self.a  # Get the numerator.
if a == 0:  # If the fraction is 0, we return 0.
    return "0"
negative: Final[bool] = a < 0  # Get the sign of the fraction.
a = abs(a)  # Make sure that `a` is now positive.
b: Final[int] = self.b  # Get the denominator.

digits: Final[list] = []  # A list for collecting digits.
while (a != 0) and (len(digits) <= max_frac):  # Create digits.
    digits.append(a // b)  # Add the current digit.
    a = 10 * (a % b)  # Ten times the remainder -> next digit.

if (a // b) >= 5:  # Do we need to round up?
    digits[-1] += 1  # Round up by incrementing last digit.

if len(digits) <= 1:  # Do we only have an integer part?
    return str((-1 if negative else 1) * digits[0])

digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
if negative:  # Do we need to restore the sign?
    digits.insert(0, "-")  # Insert the sign at the beginning.
return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- Jetzt updaten wir `a` zu `10 * (a %b)`.
- `%` ist der Rest der Division.
- `a % b` gibt uns den Rest der Division von 179 durch 16, also 3.
- Wir bekommen also `a = 30`.
- Im zweiten Schleifendurchlauf gibt uns `a // b` also `30 // 16` die Ziffer `1`.
- Nun wird `10 * (a %b)` 140 als neuer Wert für `a`.

```
>>> Fraction(124, 2).decimal_str()
'62'
>>> Fraction(1, 2).decimal_str()
'0.5'
>>> Fraction(1, 3).decimal_str(10)
'0.3333333333'
>>> Fraction(-101001, 100000000).decimal_str()
'-0.00101001'
>>> Fraction(1235, 1000).decimal_str(2)
'1.24'
>>> Fraction(99995, 100000).decimal_str(5)
'0.99995'
>>> Fraction(91995, 100000).decimal_str(3)
'0.92'
>>> Fraction(99995, 100000).decimal_str(4)
'1'
"""
a: int = self.a  # Get the numerator.
if a == 0:  # If the fraction is 0, we return 0.
    return "0"
negative: Final[bool] = a < 0  # Get the sign of the fraction.
a = abs(a)  # Make sure that `a` is now positive.
b: Final[int] = self.b  # Get the denominator.

digits: Final[list] = []  # A list for collecting digits.
while (a != 0) and (len(digits) <= max_frac):  # Create digits.
    digits.append(a // b)  # Add the current digit.
    a = 10 * (a % b)  # Ten times the remainder -> next digit.

if (a // b) >= 5:  # Do we need to round up?
    digits[-1] += 1  # Round up by incrementing last digit.

if len(digits) <= 1:  # Do we only have an integer part?
    return str((-1 if negative else 1) * digits[0])

digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
if negative:  # Do we need to restore the sign?
    digits.insert(0, "-")  # Insert the sign at the beginning.
return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- `%` ist der Rest der Division.
- `a % b` gibt uns den Rest der Division von 179 durch 16, also 3.
- Wir bekommen also `a = 30`.
- Im zweiten Schleifendurchlauf gibt uns `a // b` also `30 // 16` die Ziffer `1`.
- Nun wird `10 * (a %b)` 140 als neuer Wert für `a`.
- Das führt dann zu `140 // 16`, also 8, als dritte Ziffer und `a` wird zu `10 * (a %b)`, also 120.
- Am Anfang des vierten Schleifendurchlaufs gilt `a = 120`, währen `b = 16` unverändert bleibt.

```
>>> Fraction(124, 2).decimal_str()
'62'
>>> Fraction(1, 2).decimal_str()
'0.5'
>>> Fraction(1, 3).decimal_str(10)
'0.3333333333'
>>> Fraction(-101001, 100000000).decimal_str()
'-0.00101001'
>>> Fraction(1235, 1000).decimal_str(2)
'1.24'
>>> Fraction(99995, 100000).decimal_str(5)
'0.99995'
>>> Fraction(91995, 100000).decimal_str(3)
'0.92'
>>> Fraction(99995, 100000).decimal_str(4)
'1'
"""
a: int = self.a  # Get the numerator.
if a == 0:  # If the fraction is 0, we return 0.
    return "0"
negative: Final[bool] = a < 0  # Get the sign of the fraction.
a = abs(a)  # Make sure that `a` is now positive.
b: Final[int] = self.b  # Get the denominator.

digits: Final[list] = []  # A list for collecting digits.
while (a != 0) and (len(digits) <= max_frac):  # Create digits.
    digits.append(a // b)  # Add the current digit.
    a = 10 * (a % b)  # Ten times the remainder -> next digit.

if (a // b) >= 5:  # Do we need to round up?
    digits[-1] += 1  # Round up by incrementing last digit.

if len(digits) <= 1:  # Do we only have an integer part?
    return str((-1 if negative else 1) * digits[0])

digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
if negative:  # Do we need to restore the sign?
    digits.insert(0, "-")  # Insert the sign at the beginning.
return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- Wir bekommen also `a = 30`.
- Im zweiten Schleifendurchlauf gibt uns `a // b` also `30 // 16` die Ziffer `1`.
- Nun wird `10 * (a %b)` 140 als neuer Wert für `a`.
- Das führt dann zu `140 // 16`, also 8, als dritte Ziffer und `a` wird zu `10 * (a %b)`, also 120.
- Am Anfang des vierten Schleifendurchlaufs gilt `a = 120`, währen `b = 16` unverändert bleibt.
- Der vierte Wert, der an `digits` angehängt wird, ist daher `120 // 16 == 7`.

```
>>> Fraction(124, 2).decimal_str()
'62'
>>> Fraction(1, 2).decimal_str()
'0.5'
>>> Fraction(1, 3).decimal_str(10)
'0.3333333333'
>>> Fraction(-101001, 100000000).decimal_str()
'-0.00101001'
>>> Fraction(1235, 1000).decimal_str(2)
'1.24'
>>> Fraction(99995, 100000).decimal_str(5)
'0.99995'
>>> Fraction(91995, 100000).decimal_str(3)
'0.92'
>>> Fraction(99995, 100000).decimal_str(4)
'1'
"""
a: int = self.a  # Get the numerator.
if a == 0:  # If the fraction is 0, we return 0.
    return "0"
negative: Final[bool] = a < 0  # Get the sign of the fraction.
a = abs(a)  # Make sure that `a` is now positive.
b: Final[int] = self.b  # Get the denominator.

digits: Final[list] = []  # A list for collecting digits.
while (a != 0) and (len(digits) <= max_frac):  # Create digits.
    digits.append(a // b)  # Add the current digit.
    a = 10 * (a % b)  # Ten times the remainder -> next digit.

if (a // b) >= 5:  # Do we need to round up?
    digits[-1] += 1  # Round up by incrementing last digit.

if len(digits) <= 1:  # Do we only have an integer part?
    return str((-1 if negative else 1) * digits[0])

digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
if negative:  # Do we need to restore the sign?
    digits.insert(0, "-")  # Insert the sign at the beginning.
return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- Nun wird `10 * (a %b)` 140 als neuer Wert für `a`.
- Das führt dann zu `140 // 16`, also 8, als dritte Ziffer und `a` wird zu `10 * (a %b)`, also 120.
- Am Anfang des vierten Schleifendurchlaufs gilt `a = 120`, währen `b = 16` unverändert bleibt.
- Der vierte Wert, der an `digits` angehängt wird, ist daher `120 // 16 == 7`.
- Die Variable `a` wird mit dem Ergebnis von `10 * (a %b)` upgedated, was 80 ist.

```
>>> Fraction(124, 2).decimal_str()
'62'
>>> Fraction(1, 2).decimal_str()
'0.5'
>>> Fraction(1, 3).decimal_str(10)
'0.3333333333'
>>> Fraction(-101001, 100000000).decimal_str()
'-0.00101001'
>>> Fraction(1235, 1000).decimal_str(2)
'1.24'
>>> Fraction(99995, 100000).decimal_str(5)
'0.99995'
>>> Fraction(91995, 100000).decimal_str(3)
'0.92'
>>> Fraction(99995, 100000).decimal_str(4)
'1'
"""
a: int = self.a  # Get the numerator.
if a == 0:  # If the fraction is 0, we return 0.
    return "0"
negative: Final[bool] = a < 0  # Get the sign of the fraction.
a = abs(a)  # Make sure that `a` is now positive.
b: Final[int] = self.b  # Get the denominator.

digits: Final[list] = []  # A list for collecting digits.
while (a != 0) and (len(digits) <= max_frac):  # Create digits.
    digits.append(a // b)  # Add the current digit.
    a = 10 * (a % b)  # Ten times the remainder -> next digit.

if (a // b) >= 5:  # Do we need to round up?
    digits[-1] += 1  # Round up by incrementing last digit.

if len(digits) <= 1:  # Do we only have an integer part?
    return str((-1 if negative else 1) * digits[0])

digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
if negative:  # Do we need to restore the sign?
    digits.insert(0, "-")  # Insert the sign at the beginning.
return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- Das führt dann zu `140 // 16`, also 8, als dritte Ziffer und `a` wird zu `10 * (a %b)`, also 120.
- Am Anfang des vierten Schleifendurchlaufs gilt `a = 120`, währen `b = 16` unverändert bleibt.
- Der vierte Wert, der an `digits` angehängt wird, ist daher `120 // 16 == 7`.
- Die Variable `a` wird mit dem Ergebnis von `10 * (a %b)` upgedated, was 80 ist.
- Als letzte Ziffer bekommen wir daher `80 // 16`, nämlich `5`.

```
>>> Fraction(124, 2).decimal_str()
'62'
>>> Fraction(1, 2).decimal_str()
'0.5'
>>> Fraction(1, 3).decimal_str(10)
'0.3333333333'
>>> Fraction(-101001, 100000000).decimal_str()
'-0.00101001'
>>> Fraction(1235, 1000).decimal_str(2)
'1.24'
>>> Fraction(99995, 100000).decimal_str(5)
'0.99995'
>>> Fraction(91995, 100000).decimal_str(3)
'0.92'
>>> Fraction(99995, 100000).decimal_str(4)
'1'
"""
a: int = self.a  # Get the numerator.
if a == 0:  # If the fraction is 0, we return 0.
    return "0"
negative: Final[bool] = a < 0  # Get the sign of the fraction.
a = abs(a)  # Make sure that `a` is now positive.
b: Final[int] = self.b  # Get the denominator.

digits: Final[list] = []  # A list for collecting digits.
while (a != 0) and (len(digits) <= max_frac):  # Create digits.
    digits.append(a // b)  # Add the current digit.
    a = 10 * (a % b)  # Ten times the remainder -> next digit.

if (a // b) >= 5:  # Do we need to round up?
    digits[-1] += 1  # Round up by incrementing last digit.

if len(digits) <= 1:  # Do we only have an integer part?
    return str((-1 if negative else 1) * digits[0])

digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
if negative:  # Do we need to restore the sign?
    digits.insert(0, "-")  # Insert the sign at the beginning.
return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- Am Anfang des vierten Schleifendurchlaufs gilt `a = 120`, währen `b = 16` unverändert bleibt.
- Der vierte Wert, der an `digits` angehängt wird, ist daher `120 // 16 == 7`.
- Die Variable `a` wird mit dem Ergebnis von `10 * (a %b)` upgedated, was 80 ist.
- Als letzte Ziffer bekommen wir daher `80 // 16`, nämlich `5`.
- Das ist die letzte Ziffer, denn `80 % 16` ist 0.

```
        >>> Fraction(124, 2).decimal_str()
        '62'
        >>> Fraction(1, 2).decimal_str()
        '0.5'
        >>> Fraction(1, 3).decimal_str(10)
        '0.3333333333'
        >>> Fraction(-101001, 100000000).decimal_str()
        '-0.00101001'
        >>> Fraction(1235, 1000).decimal_str(2)
        '1.24'
        >>> Fraction(99995, 100000).decimal_str(5)
        '0.99995'
        >>> Fraction(91995, 100000).decimal_str(3)
        '0.92'
        >>> Fraction(99995, 100000).decimal_str(4)
        '1'
        """
        a: int = self.a  # Get the numerator.
        if a == 0:  # If the fraction is 0, we return 0.
            return "0"
        negative: Final[bool] = a < 0  # Get the sign of the fraction.
        a = abs(a)  # Make sure that `a` is now positive.
        b: Final[int] = self.b  # Get the denominator.

        digits: Final[list] = []  # A list for collecting digits.
        while (a != 0) and (len(digits) <= max_frac):  # Create digits.
            digits.append(a // b)  # Add the current digit.
            a = 10 * (a % b)  # Ten times the remainder -> next digit.

        if (a // b) >= 5:  # Do we need to round up?
            digits[-1] += 1  # Round up by incrementing last digit.

        if len(digits) <= 1:  # Do we only have an integer part?
            return str((-1 if negative else 1) * digits[0])

        digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
        if negative:  # Do we need to restore the sign?
            digits.insert(0, "-")  # Insert the sign at the beginning.
        return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- Am Anfang des vierten Schleifendurchlaufs gilt `a = 120`, währen `b = 16` unverändert bleibt.
- Der vierte Wert, der an `digits` angehängt wird, ist daher `120 // 16 == 7`.
- Die Variable `a` wird mit dem Ergebnis von `10 * (a %b)` upgedated, was 80 ist.
- Als letzte Ziffer bekommen wir daher `80 // 16`, nämlich `5`.
- Das ist die letzte Ziffer, denn `80 % 16` ist 0.
- Deshalb trifft `a == 0` nach der fünften Iteration zu.

```
>>> Fraction(124, 2).decimal_str()
'62'
>>> Fraction(1, 2).decimal_str()
'0.5'
>>> Fraction(1, 3).decimal_str(10)
'0.3333333333'
>>> Fraction(-101001, 100000000).decimal_str()
'-0.00101001'
>>> Fraction(1235, 1000).decimal_str(2)
'1.24'
>>> Fraction(99995, 100000).decimal_str(5)
'0.99995'
>>> Fraction(91995, 100000).decimal_str(3)
'0.92'
>>> Fraction(99995, 100000).decimal_str(4)
'1'
"""
a: int = self.a  # Get the numerator.
if a == 0:  # If the fraction is 0, we return 0.
    return "0"
negative: Final[bool] = a < 0  # Get the sign of the fraction.
a = abs(a)  # Make sure that `a` is now positive.
b: Final[int] = self.b  # Get the denominator.

digits: Final[list] = []  # A list for collecting digits.
while (a != 0) and (len(digits) <= max_frac):  # Create digits.
    digits.append(a // b)  # Add the current digit.
    a = 10 * (a % b)  # Ten times the remainder -> next digit.

if (a // b) >= 5:  # Do we need to round up?
    digits[-1] += 1  # Round up by incrementing last digit.

if len(digits) <= 1:  # Do we only have an integer part?
    return str((-1 if negative else 1) * digits[0])

digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
if negative:  # Do we need to restore the sign?
    digits.insert(0, "-")  # Insert the sign at the beginning.
return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- Die Variable `a` wird mit dem Ergebnis von `10 * (a %b)` upgedated, was 80 ist.
- Als letzte Ziffer bekommen wir daher `80 // 16`, nämlich `5`.
- Das ist die letzte Ziffer, denn `80 % 16` ist 0.
- Deshalb trifft `a == 0` nach der fünften Iteration zu.
- Dadurch wird die erste Schleifenbedingung (`a != 0`) `False` und die Schleife bricht ab.

```
>>> Fraction(124, 2).decimal_str()
'62'
>>> Fraction(1, 2).decimal_str()
'0.5'
>>> Fraction(1, 3).decimal_str(10)
'0.3333333333'
>>> Fraction(-101001, 100000000).decimal_str()
'-0.00101001'
>>> Fraction(1235, 1000).decimal_str(2)
'1.24'
>>> Fraction(99995, 100000).decimal_str(5)
'0.99995'
>>> Fraction(91995, 100000).decimal_str(3)
'0.92'
>>> Fraction(99995, 100000).decimal_str(4)
'1'
"""
a: int = self.a  # Get the numerator.
if a == 0:  # If the fraction is 0, we return 0.
    return "0"
negative: Final[bool] = a < 0  # Get the sign of the fraction.
a = abs(a)  # Make sure that `a` is now positive.
b: Final[int] = self.b  # Get the denominator.

digits: Final[list] = []  # A list for collecting digits.
while (a != 0) and (len(digits) <= max_frac):  # Create digits.
    digits.append(a // b)  # Add the current digit.
    a = 10 * (a % b)  # Ten times the remainder -> next digit.

if (a // b) >= 5:  # Do we need to round up?
    digits[-1] += 1  # Round up by incrementing last digit.

if len(digits) <= 1:  # Do we only have an integer part?
    return str((-1 if negative else 1) * digits[0])

digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
if negative:  # Do we need to restore the sign?
    digits.insert(0, "-")  # Insert the sign at the beginning.
return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- Die Variable `a` wird mit dem Ergebnis von `10 * (a %b)` upgedated, was 80 ist.
- Als letzte Ziffer bekommen wir daher `80 // 16`, nämlich `5`.
- Das ist die letzte Ziffer, denn `80 % 16` ist 0.
- Deshalb trifft `a == 0` nach der fünften Iteration zu.
- Dadurch wird die erste Schleifenbedingung (`a != 0`) `False` und die Schleife bricht ab.
- Zu diesem Zeitpunkt ist `digits == [11, 1, 8, 7, 5]`.

```
>>> Fraction(124, 2).decimal_str()
'62'
>>> Fraction(1, 2).decimal_str()
'0.5'
>>> Fraction(1, 3).decimal_str(10)
'0.3333333333'
>>> Fraction(-101001, 100000000).decimal_str()
'-0.00101001'
>>> Fraction(1235, 1000).decimal_str(2)
'1.24'
>>> Fraction(99995, 100000).decimal_str(5)
'0.99995'
>>> Fraction(91995, 100000).decimal_str(3)
'0.92'
>>> Fraction(99995, 100000).decimal_str(4)
'1'
"""
a: int = self.a  # Get the numerator.
if a == 0:  # If the fraction is 0, we return 0.
    return "0"
negative: Final[bool] = a < 0  # Get the sign of the fraction.
a = abs(a)  # Make sure that `a` is now positive.
b: Final[int] = self.b  # Get the denominator.

digits: Final[list] = []  # A list for collecting digits.
while (a != 0) and (len(digits) <= max_frac):  # Create digits.
    digits.append(a // b)  # Add the current digit.
    a = 10 * (a % b)  # Ten times the remainder -> next digit.

if (a // b) >= 5:  # Do we need to round up?
    digits[-1] += 1  # Round up by incrementing last digit.

if len(digits) <= 1:  # Do we only have an integer part?
    return str((-1 if negative else 1) * digits[0])

digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
if negative:  # Do we need to restore the sign?
    digits.insert(0, "-")  # Insert the sign at the beginning.
return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- Als letzte Ziffer bekommen wir daher `80 // 16`, nämlich `5`.
- Das ist die letzte Ziffer, denn `80 % 16` ist 0.
- Deshalb trifft `a == 0` nach der fünften Iteration zu.
- Dadurch wird die erste Schleifenbedingung (`a != 0`) `False` und die Schleife bricht ab.
- Zu diesem Zeitpunkt ist `digits == [11, 1, 8, 7, 5]`.
- Und das stimm, denn $\frac{179}{16} = 11.1875$.

```
>>> Fraction(124, 2).decimal_str()
'62'
>>> Fraction(1, 2).decimal_str()
'0.5'
>>> Fraction(1, 3).decimal_str(10)
'0.3333333333'
>>> Fraction(-101001, 100000000).decimal_str()
'-0.00101001'
>>> Fraction(1235, 1000).decimal_str(2)
'1.24'
>>> Fraction(99995, 100000).decimal_str(5)
'0.99995'
>>> Fraction(91995, 100000).decimal_str(3)
'0.92'
>>> Fraction(99995, 100000).decimal_str(4)
'1'
"""
a: int = self.a  # Get the numerator.
if a == 0:  # If the fraction is 0, we return 0.
    return "0"
negative: Final[bool] = a < 0  # Get the sign of the fraction.
a = abs(a)  # Make sure that `a` is now positive.
b: Final[int] = self.b  # Get the denominator.

digits: Final[list] = []  # A list for collecting digits.
while (a != 0) and (len(digits) <= max_frac):  # Create digits.
    digits.append(a // b)  # Add the current digit.
    a = 10 * (a % b)  # Ten times the remainder -> next digit.

if (a // b) >= 5:  # Do we need to round up?
    digits[-1] += 1  # Round up by incrementing last digit.

if len(digits) <= 1:  # Do we only have an integer part?
    return str((-1 if negative else 1) * digits[0])

digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
if negative:  # Do we need to restore the sign?
    digits.insert(0, "-")  # Insert the sign at the beginning.
return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- Das ist die letzte Ziffer, denn `80 % 16` ist 0.
- Deshalb trifft `a == 0` nach der fünften Iteration zu.
- Dadurch wird die erste Schleifenbedingung (`a != 0`) `False` und die Schleife bricht ab.
- Zu diesem Zeitpunkt ist `digits == [11, 1, 8, 7, 5]`.
- Und das stimm, denn $\frac{179}{16} = 11.1875$.
- Beachten Sie, dass es zwei Bedingungen gibt, bei denen die Schleife abbricht

```
>>> Fraction(124, 2).decimal_str()
'62'
>>> Fraction(1, 2).decimal_str()
'0.5'
>>> Fraction(1, 3).decimal_str(10)
'0.3333333333'
>>> Fraction(-101001, 100000000).decimal_str()
'-0.00101001'
>>> Fraction(1235, 1000).decimal_str(2)
'1.24'
>>> Fraction(99995, 100000).decimal_str(5)
'0.99995'
>>> Fraction(91995, 100000).decimal_str(3)
'0.92'
>>> Fraction(99995, 100000).decimal_str(4)
'1'
"""
a: int = self.a  # Get the numerator.
if a == 0:  # If the fraction is 0, we return 0.
    return "0"
negative: Final[bool] = a < 0  # Get the sign of the fraction.
a = abs(a)  # Make sure that `a` is now positive.
b: Final[int] = self.b  # Get the denominator.

digits: Final[list] = []  # A list for collecting digits.
while (a != 0) and (len(digits) <= max_frac):  # Create digits.
    digits.append(a // b)  # Add the current digit.
    a = 10 * (a % b)  # Ten times the remainder -> next digit.

if (a // b) >= 5:  # Do we need to round up?
    digits[-1] += 1  # Round up by incrementing last digit.

if len(digits) <= 1:  # Do we only have an integer part?
    return str((-1 if negative else 1) * digits[0])

digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
if negative:  # Do we need to restore the sign?
    digits.insert(0, "-")  # Insert the sign at the beginning.
return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- Dadurch wird die erste Schleifenbedingung (`a != 0`) `False` und die Schleife bricht ab.
- Zu diesem Zeitpunkt ist `digits == [11, 1, 8, 7, 5]`.
- Und das stimm, denn $\frac{179}{16} = 11.1875$.
- Beachten Sie, dass es zwei Bedingungen gibt, bei denen die Schleife abbricht
- Sie hört auf, wenn wir den Bruch komplett und vollständig als Textstring mit nicht mehr als als der angegebenen Menge `max_frac` von Ziffern darstellen können.

```
>>> Fraction(124, 2).decimal_str()
'62'
>>> Fraction(1, 2).decimal_str()
'0.5'
>>> Fraction(1, 3).decimal_str(10)
'0.3333333333'
>>> Fraction(-101001, 100000000).decimal_str()
'-0.00101001'
>>> Fraction(1235, 1000).decimal_str(2)
'1.24'
>>> Fraction(99995, 100000).decimal_str(5)
'0.99995'
>>> Fraction(91995, 100000).decimal_str(3)
'0.92'
>>> Fraction(99995, 100000).decimal_str(4)
'1'
"""
a: int = self.a  # Get the numerator.
if a == 0:  # If the fraction is 0, we return 0.
    return "0"
negative: Final[bool] = a < 0  # Get the sign of the fraction.
a = abs(a)  # Make sure that `a` is now positive.
b: Final[int] = self.b  # Get the denominator.

digits: Final[list] = []  # A list for collecting digits.
while (a != 0) and (len(digits) <= max_frac):  # Create digits.
    digits.append(a // b)  # Add the current digit.
    a = 10 * (a % b)  # Ten times the remainder -> next digit.

if (a // b) >= 5:  # Do we need to round up?
    digits[-1] += 1  # Round up by incrementing last digit.

if len(digits) <= 1:  # Do we only have an integer part?
    return str((-1 if negative else 1) * digits[0])

digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
if negative:  # Do we need to restore the sign?
    digits.insert(0, "-")  # Insert the sign at the beginning.
return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- Zu diesem Zeitpunkt ist `digits == [11, 1, 8, 7, 5]`.
- Und das stimm, denn $\frac{179}{16} = 11.1875$.
- Beachten Sie, dass es zwei Bedingungen gibt, bei denen die Schleife abbricht
- Sie hört auf, wenn wir den Bruch komplett und vollständig als Textstring mit nicht mehr als als der angegebenen Menge `max_frac` von Ziffern darstellen können.
- Das war der Fall in unserem Beispiel.

```
>>> Fraction(124, 2).decimal_str()
'62'
>>> Fraction(1, 2).decimal_str()
'0.5'
>>> Fraction(1, 3).decimal_str(10)
'0.3333333333'
>>> Fraction(-101001, 100000000).decimal_str()
'-0.00101001'
>>> Fraction(1235, 1000).decimal_str(2)
'1.24'
>>> Fraction(99995, 100000).decimal_str(5)
'0.99995'
>>> Fraction(91995, 100000).decimal_str(3)
'0.92'
>>> Fraction(99995, 100000).decimal_str(4)
'1'
"""
a: int = self.a  # Get the numerator.
if a == 0:  # If the fraction is 0, we return 0.
    return "0"
negative: Final[bool] = a < 0  # Get the sign of the fraction.
a = abs(a)  # Make sure that `a` is now positive.
b: Final[int] = self.b  # Get the denominator.

digits: Final[list] = []  # A list for collecting digits.
while (a != 0) and (len(digits) <= max_frac):  # Create digits.
    digits.append(a // b)  # Add the current digit.
    a = 10 * (a % b)  # Ten times the remainder -> next digit.

if (a // b) >= 5:  # Do we need to round up?
    digits[-1] += 1  # Round up by incrementing last digit.

if len(digits) <= 1:  # Do we only have an integer part?
    return str((-1 if negative else 1) * digits[0])

digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
if negative:  # Do we need to restore the sign?
    digits.insert(0, "-")  # Insert the sign at the beginning.
return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- Und das stimm, denn $\frac{179}{16} = 11.1875$.
- Beachten Sie, dass es zwei Bedingungen gibt, bei denen die Schleife abbricht
- Sie hört auf, wenn wir den Bruch komplett und vollständig als Textstring mit nicht mehr als als der angegebenen Menge `max_frac` von Ziffern darstellen können.
- Das war der Fall in unserem Beispiel.
- Die Schleife hört auch auf, wenn wir die maximale Anzahl von Ziffern erreichen, also wenn `len(digits)` `max_frac` übertrifft.

```
>>> Fraction(124, 2).decimal_str()
'62'
>>> Fraction(1, 2).decimal_str()
'0.5'
>>> Fraction(1, 3).decimal_str(10)
'0.3333333333'
>>> Fraction(-101001, 100000000).decimal_str()
'-0.00101001'
>>> Fraction(1235, 1000).decimal_str(2)
'1.24'
>>> Fraction(99995, 100000).decimal_str(5)
'0.99995'
>>> Fraction(91995, 100000).decimal_str(3)
'0.92'
>>> Fraction(99995, 100000).decimal_str(4)
'1'
"""
a: int = self.a  # Get the numerator.
if a == 0:  # If the fraction is 0, we return 0.
    return "0"
negative: Final[bool] = a < 0  # Get the sign of the fraction.
a = abs(a)  # Make sure that `a` is now positive.
b: Final[int] = self.b  # Get the denominator.

digits: Final[list] = []  # A list for collecting digits.
while (a != 0) and (len(digits) <= max_frac):  # Create digits.
    digits.append(a // b)  # Add the current digit.
    a = 10 * (a % b)  # Ten times the remainder -> next digit.

if (a // b) >= 5:  # Do we need to round up?
    digits[-1] += 1  # Round up by incrementing last digit.

if len(digits) <= 1:  # Do we only have an integer part?
    return str((-1 if negative else 1) * digits[0])

digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
if negative:  # Do we need to restore the sign?
    digits.insert(0, "-")  # Insert the sign at the beginning.
return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- Beachten Sie, dass es zwei Bedingungen gibt, bei denen die Schleife abbricht
- Sie hört auf, wenn wir den Bruch komplett und vollständig als Textstring mit nicht mehr als als der angegebenen Menge `max_frac` von Ziffern darstellen können.
- Das war der Fall in unserem Beispiel.
- Die Schleife hört auch auf, wenn wir die maximale Anzahl von Ziffern erreichen, also wenn `len(digits)` `max_frac` übertrifft.
- Aber das kann auch zu Problemen führen.

```
        >>> Fraction(124, 2).decimal_str()
        '62'
        >>> Fraction(1, 2).decimal_str()
        '0.5'
        >>> Fraction(1, 3).decimal_str(10)
        '0.3333333333'
        >>> Fraction(-101001, 100000000).decimal_str()
        '-0.00101001'
        >>> Fraction(1235, 1000).decimal_str(2)
        '1.24'
        >>> Fraction(99995, 100000).decimal_str(5)
        '0.99995'
        >>> Fraction(91995, 100000).decimal_str(3)
        '0.92'
        >>> Fraction(99995, 100000).decimal_str(4)
        '1'
        """
        a: int = self.a  # Get the numerator.
        if a == 0:  # If the fraction is 0, we return 0.
            return "0"
        negative: Final[bool] = a < 0  # Get the sign of the fraction.
        a = abs(a)  # Make sure that `a` is now positive.
        b: Final[int] = self.b  # Get the denominator.

        digits: Final[list] = []  # A list for collecting digits.
        while (a != 0) and (len(digits) <= max_frac):  # Create digits.
            digits.append(a // b)  # Add the current digit.
            a = 10 * (a % b)  # Ten times the remainder -> next digit.

        if (a // b) >= 5:  # Do we need to round up?
            digits[-1] += 1  # Round up by incrementing last digit.

        if len(digits) <= 1:  # Do we only have an integer part?
            return str((-1 if negative else 1) * digits[0])

        digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
        if negative:  # Do we need to restore the sign?
            digits.insert(0, "-")  # Insert the sign at the beginning.
        return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- Sie hört auf, wenn wir den Bruch komplett und vollständig als Textstring mit nicht mehr als als der angegebenen Menge `max_frac` von Ziffern darstellen können.
- Das war der Fall in unserem Beispiel.
- Die Schleife hört auch auf, wenn wir die maximale Anzahl von Ziffern erreichen, also wenn `len(digits)` `max_frac` übertrifft.
- Aber das kann auch zu Problemen führen.
- Was passiert z. B., wenn wir $\frac{10006}{10000} = 1.0006$ mit nur *3* Nachkommastellen ausdrücken wollen?

```
        >>> Fraction(124, 2).decimal_str()
        '62'
        >>> Fraction(1, 2).decimal_str()
        '0.5'
        >>> Fraction(1, 3).decimal_str(10)
        '0.3333333333'
        >>> Fraction(-101001, 100000000).decimal_str()
        '-0.00101001'
        >>> Fraction(1235, 1000).decimal_str(2)
        '1.24'
        >>> Fraction(99995, 100000).decimal_str(5)
        '0.99995'
        >>> Fraction(91995, 100000).decimal_str(3)
        '0.92'
        >>> Fraction(99995, 100000).decimal_str(4)
        '1'
        """
        a: int = self.a  # Get the numerator.
        if a == 0:  # If the fraction is 0, we return 0.
            return "0"
        negative: Final[bool] = a < 0  # Get the sign of the fraction.
        a = abs(a)  # Make sure that `a` is now positive.
        b: Final[int] = self.b  # Get the denominator.

        digits: Final[list] = []  # A list for collecting digits.
        while (a != 0) and (len(digits) <= max_frac):  # Create digits.
            digits.append(a // b)  # Add the current digit.
            a = 10 * (a % b)  # Ten times the remainder -> next digit.

        if (a // b) >= 5:  # Do we need to round up?
            digits[-1] += 1  # Round up by incrementing last digit.

        if len(digits) <= 1:  # Do we only have an integer part?
            return str((-1 if negative else 1) * digits[0])

        digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
        if negative:  # Do we need to restore the sign?
            digits.insert(0, "-")  # Insert the sign at the beginning.
        return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- Das war der Fall in unserem Beispiel.
- Die Schleife hört auch auf, wenn wir die maximale Anzahl von Ziffern erreichen, also wenn `len(digits)` `max_frac` übertrifft.
- Aber das kann auch zu Problemen führen.
- Was passiert z. B., wenn wir $\frac{10006}{10000} = 1.0006$ mit nur *3* Nachkommastellen ausdrücken wollen?
- Nach der Schleife steht dann `[1, 0, 0, 0]` in `digits`.

```
    >>> Fraction(124, 2).decimal_str()
    '62'
    >>> Fraction(1, 2).decimal_str()
    '0.5'
    >>> Fraction(1, 3).decimal_str(10)
    '0.3333333333'
    >>> Fraction(-101001, 100000000).decimal_str()
    '-0.00101001'
    >>> Fraction(1235, 1000).decimal_str(2)
    '1.24'
    >>> Fraction(99995, 100000).decimal_str(5)
    '0.99995'
    >>> Fraction(91995, 100000).decimal_str(3)
    '0.92'
    >>> Fraction(99995, 100000).decimal_str(4)
    '1'
    """
    a: int = self.a  # Get the numerator.
    if a == 0:  # If the fraction is 0, we return 0.
        return "0"
    negative: Final[bool] = a < 0  # Get the sign of the fraction.
    a = abs(a)  # Make sure that `a` is now positive.
    b: Final[int] = self.b  # Get the denominator.

    digits: Final[list] = []  # A list for collecting digits.
    while (a != 0) and (len(digits) <= max_frac):  # Create digits.
        digits.append(a // b)  # Add the current digit.
        a = 10 * (a % b)  # Ten times the remainder -> next digit.

    if (a // b) >= 5:  # Do we need to round up?
        digits[-1] += 1  # Round up by incrementing last digit.

    if len(digits) <= 1:  # Do we only have an integer part?
        return str((-1 if negative else 1) * digits[0])

    digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
    if negative:  # Do we need to restore the sign?
        digits.insert(0, "-")  # Insert the sign at the beginning.
    return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- Die Schleife hört auch auf, wenn wir die maximale Anzahl von Ziffern erreichen, also wenn `len(digits)` `max_frac` übertrifft.
- Aber das kann auch zu Problemen führen.
- Was passiert z. B., wenn wir $\frac{10006}{10000} = 1.0006$ mit nur *3* Nachkommastellen ausdrücken wollen?
- Nach der Schleife steht dann `[1, 0, 0, 0]` in `digits`.
- Zu diesem Zeitpunkt hätten wir dann `a = 60000` und `b = 10000`.

```
>>> Fraction(124, 2).decimal_str()
'62'
>>> Fraction(1, 2).decimal_str()
'0.5'
>>> Fraction(1, 3).decimal_str(10)
'0.3333333333'
>>> Fraction(-101001, 100000000).decimal_str()
'-0.00101001'
>>> Fraction(1235, 1000).decimal_str(2)
'1.24'
>>> Fraction(99995, 100000).decimal_str(5)
'0.99995'
>>> Fraction(91995, 100000).decimal_str(3)
'0.92'
>>> Fraction(99995, 100000).decimal_str(4)
'1'
"""
a: int = self.a  # Get the numerator.
if a == 0:  # If the fraction is 0, we return 0.
    return "0"
negative: Final[bool] = a < 0  # Get the sign of the fraction.
a = abs(a)  # Make sure that `a` is now positive.
b: Final[int] = self.b  # Get the denominator.

digits: Final[list] = []  # A list for collecting digits.
while (a != 0) and (len(digits) <= max_frac):  # Create digits.
    digits.append(a // b)  # Add the current digit.
    a = 10 * (a % b)  # Ten times the remainder -> next digit.

if (a // b) >= 5:  # Do we need to round up?
    digits[-1] += 1  # Round up by incrementing last digit.

if len(digits) <= 1:  # Do we only have an integer part?
    return str((-1 if negative else 1) * digits[0])

digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
if negative:  # Do we need to restore the sign?
    digits.insert(0, "-")  # Insert the sign at the beginning.
return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- Aber das kann auch zu Problemen führen.
- Was passiert z. B., wenn wir $\frac{10006}{10000} = 1.0006$ mit nur *3* Nachkommastellen ausdrücken wollen?
- Nach der Schleife steht dann `[1, 0, 0, 0]` in `digits`.
- Zu diesem Zeitpunkt hätten wir dann `a = 60000` und `b = 10000`.
- Die 6 am Ende des Zählers ist ungünstig:

```
>>> Fraction(124, 2).decimal_str()
'62'
>>> Fraction(1, 2).decimal_str()
'0.5'
>>> Fraction(1, 3).decimal_str(10)
'0.3333333333'
>>> Fraction(-101001, 100000000).decimal_str()
'-0.00101001'
>>> Fraction(1235, 1000).decimal_str(2)
'1.24'
>>> Fraction(99995, 100000).decimal_str(5)
'0.99995'
>>> Fraction(91995, 100000).decimal_str(3)
'0.92'
>>> Fraction(99995, 100000).decimal_str(4)
'1'
"""
a: int = self.a  # Get the numerator.
if a == 0:  # If the fraction is 0, we return 0.
    return "0"
negative: Final[bool] = a < 0  # Get the sign of the fraction.
a = abs(a)  # Make sure that `a` is now positive.
b: Final[int] = self.b  # Get the denominator.

digits: Final[list] = []  # A list for collecting digits.
while (a != 0) and (len(digits) <= max_frac):  # Create digits.
    digits.append(a // b)  # Add the current digit.
    a = 10 * (a % b)  # Ten times the remainder -> next digit.

if (a // b) >= 5:  # Do we need to round up?
    digits[-1] += 1  # Round up by incrementing last digit.

if len(digits) <= 1:  # Do we only have an integer part?
    return str((-1 if negative else 1) * digits[0])

digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
if negative:  # Do we need to restore the sign?
    digits.insert(0, "-")  # Insert the sign at the beginning.
return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- Was passiert z. B., wenn wir $\frac{10006}{10000} = 1.0006$ mit nur *3* Nachkommastellen ausdrücken wollen?
- Nach der Schleife steht dann `[1, 0, 0, 0]` in `digits`.
- Zu diesem Zeitpunkt hätten wir dann `a = 60000` und `b = 10000`.
- Die 6 am Ende des Zählers ist ungünstig:
- Wenn wir den Bruch mit drei Nachkommastellen darstellen wollen, dann sollte da 1.001 herauskommen, nicht 1.000, wie in der Liste `digits` steht.

```
>>> Fraction(124, 2).decimal_str()
'62'
>>> Fraction(1, 2).decimal_str()
'0.5'
>>> Fraction(1, 3).decimal_str(10)
'0.3333333333'
>>> Fraction(-101001, 100000000).decimal_str()
'-0.00101001'
>>> Fraction(1235, 1000).decimal_str(2)
'1.24'
>>> Fraction(99995, 100000).decimal_str(5)
'0.99995'
>>> Fraction(91995, 100000).decimal_str(3)
'0.92'
>>> Fraction(99995, 100000).decimal_str(4)
'1'
"""
a: int = self.a  # Get the numerator.
if a == 0:  # If the fraction is 0, we return 0.
    return "0"
negative: Final[bool] = a < 0  # Get the sign of the fraction.
a = abs(a)  # Make sure that `a` is now positive.
b: Final[int] = self.b  # Get the denominator.

digits: Final[list] = []  # A list for collecting digits.
while (a != 0) and (len(digits) <= max_frac):  # Create digits.
    digits.append(a // b)  # Add the current digit.
    a = 10 * (a % b)  # Ten times the remainder -> next digit.

if (a // b) >= 5:  # Do we need to round up?
    digits[-1] += 1  # Round up by incrementing last digit.

if len(digits) <= 1:  # Do we only have an integer part?
    return str((-1 if negative else 1) * digits[0])

digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
if negative:  # Do we need to restore the sign?
    digits.insert(0, "-")  # Insert the sign at the beginning.
return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- Nach der Schleife steht dann `[1, 0, 0, 0]` in `digits`.
- Zu diesem Zeitpunkt hätten wir dann `a = 60000` und `b = 10000`.
- Die 6 am Ende des Zählers ist ungünstig:
- Wenn wir den Bruch mit drei Nachkommastellen darstellen wollen, dann sollte da 1.001 herauskommen, nicht 1.000, wie in der Liste `digits` steht.
- Nun, alles, was wir tun müssen, um das zu reparieren, ist zu prüfen ob die nächste Ziffer, die wir anhängen würden, größer oder gleich 5 wäre.

```
>>> Fraction(124, 2).decimal_str()
'62'
>>> Fraction(1, 2).decimal_str()
'0.5'
>>> Fraction(1, 3).decimal_str(10)
'0.3333333333'
>>> Fraction(-101001, 100000000).decimal_str()
'-0.00101001'
>>> Fraction(1235, 1000).decimal_str(2)
'1.24'
>>> Fraction(99995, 100000).decimal_str(5)
'0.99995'
>>> Fraction(91995, 100000).decimal_str(3)
'0.92'
>>> Fraction(99995, 100000).decimal_str(4)
'1'
"""
a: int = self.a  # Get the numerator.
if a == 0:  # If the fraction is 0, we return 0.
    return "0"
negative: Final[bool] = a < 0  # Get the sign of the fraction.
a = abs(a)  # Make sure that `a` is now positive.
b: Final[int] = self.b  # Get the denominator.

digits: Final[list] = []  # A list for collecting digits.
while (a != 0) and (len(digits) <= max_frac):  # Create digits.
    digits.append(a // b)  # Add the current digit.
    a = 10 * (a % b)  # Ten times the remainder -> next digit.

if (a // b) >= 5:  # Do we need to round up?
    digits[-1] += 1  # Round up by incrementing last digit.

if len(digits) <= 1:  # Do we only have an integer part?
    return str((-1 if negative else 1) * digits[0])

digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
if negative:  # Do we need to restore the sign?
    digits.insert(0, "-")  # Insert the sign at the beginning.
return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- Zu diesem Zeitpunkt hätten wir dann `a = 60000` und `b = 10000`.
- Die 6 am Ende des Zählers ist ungünstig:
- Wenn wir den Bruch mit drei Nachkommastellen darstellen wollen, dann sollte da 1.001 herauskommen, nicht 1.000, wie in der Liste `digits` steht.
- Nun, alles, was wir tun müssen, um das zu reparieren, ist zu prüfen ob die nächste Ziffer, die wir anhängen würden, größer oder gleich 5 wäre.
- Wenn ja, dann erhöhen wir die letzte Ziffer in `digits` um `1`.

```
>>> Fraction(124, 2).decimal_str()
'62'
>>> Fraction(1, 2).decimal_str()
'0.5'
>>> Fraction(1, 3).decimal_str(10)
'0.3333333333'
>>> Fraction(-101001, 100000000).decimal_str()
'-0.00101001'
>>> Fraction(1235, 1000).decimal_str(2)
'1.24'
>>> Fraction(99995, 100000).decimal_str(5)
'0.99995'
>>> Fraction(91995, 100000).decimal_str(3)
'0.92'
>>> Fraction(99995, 100000).decimal_str(4)
'1'
"""
a: int = self.a  # Get the numerator.
if a == 0:  # If the fraction is 0, we return 0.
    return "0"
negative: Final[bool] = a < 0  # Get the sign of the fraction.
a = abs(a)  # Make sure that `a` is now positive.
b: Final[int] = self.b  # Get the denominator.

digits: Final[list] = []  # A list for collecting digits.
while (a != 0) and (len(digits) <= max_frac):  # Create digits.
    digits.append(a // b)  # Add the current digit.
    a = 10 * (a % b)  # Ten times the remainder -> next digit.

if (a // b) >= 5:  # Do we need to round up?
    digits[-1] += 1  # Round up by incrementing last digit.

if len(digits) <= 1:  # Do we only have an integer part?
    return str((-1 if negative else 1) * digits[0])

digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
if negative:  # Do we need to restore the sign?
    digits.insert(0, "-")  # Insert the sign at the beginning.
return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- Die 6 am Ende des Zählers ist ungünstig:
- Wenn wir den Bruch mit drei Nachkommastellen darstellen wollen, dann sollte da 1.001 herauskommen, nicht 1.000, wie in der Liste `digits` steht.
- Nun, alles, was wir tun müssen, um das zu reparieren, ist zu prüfen ob die nächste Ziffer, die wir anhängen würden, größer oder gleich 5 wäre.
- Wenn ja, dann erhöhen wir die letzte Ziffer in `digits` um `1`.
- Das wird sich später als Fehler herausstellen. . .

```
>>> Fraction(124, 2).decimal_str()
'62'
>>> Fraction(1, 2).decimal_str()
'0.5'
>>> Fraction(1, 3).decimal_str(10)
'0.3333333333'
>>> Fraction(-101001, 100000000).decimal_str()
'-0.00101001'
>>> Fraction(1235, 1000).decimal_str(2)
'1.24'
>>> Fraction(99995, 100000).decimal_str(5)
'0.99995'
>>> Fraction(91995, 100000).decimal_str(3)
'0.92'
>>> Fraction(99995, 100000).decimal_str(4)
'1'
"""
a: int = self.a  # Get the numerator.
if a == 0:  # If the fraction is 0, we return 0.
    return "0"
negative: Final[bool] = a < 0  # Get the sign of the fraction.
a = abs(a)  # Make sure that `a` is now positive.
b: Final[int] = self.b  # Get the denominator.

digits: Final[list] = []  # A list for collecting digits.
while (a != 0) and (len(digits) <= max_frac):  # Create digits.
    digits.append(a // b)  # Add the current digit.
    a = 10 * (a % b)  # Ten times the remainder -> next digit.

if (a // b) >= 5:  # Do we need to round up?
    digits[-1] += 1  # Round up by incrementing last digit.

if len(digits) <= 1:  # Do we only have an integer part?
    return str((-1 if negative else 1) * digits[0])

digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
if negative:  # Do we need to restore the sign?
    digits.insert(0, "-")  # Insert the sign at the beginning.
return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- Nun, alles, was wir tun müssen, um das zu reparieren, ist zu prüfen ob die nächste Ziffer, die wir anhängen würden, größer oder gleich 5 wäre.
- Wenn ja, dann erhöhen wir die letzte Ziffer in `digits` um `1`.
- Das wird sich später als Fehler herausstellen...
- Um ein Aufrunden einzuführen, fügen wir ein `if (a // b)>= 5` ein, das dann `digits[-1] += 1` ausführt, wenn seine Kondition zutrifft.

```
>>> Fraction(124, 2).decimal_str()
'62'
>>> Fraction(1, 2).decimal_str()
'0.5'
>>> Fraction(1, 3).decimal_str(10)
'0.3333333333'
>>> Fraction(-101001, 100000000).decimal_str()
'-0.00101001'
>>> Fraction(1235, 1000).decimal_str(2)
'1.24'
>>> Fraction(99995, 100000).decimal_str(5)
'0.99995'
>>> Fraction(91995, 100000).decimal_str(3)
'0.92'
>>> Fraction(99995, 100000).decimal_str(4)
'1'
"""
a: int = self.a  # Get the numerator.
if a == 0:  # If the fraction is 0, we return 0.
    return "0"
negative: Final[bool] = a < 0  # Get the sign of the fraction.
a = abs(a)  # Make sure that `a` is now positive.
b: Final[int] = self.b  # Get the denominator.

digits: Final[list] = []  # A list for collecting digits.
while (a != 0) and (len(digits) <= max_frac):  # Create digits.
    digits.append(a // b)  # Add the current digit.
    a = 10 * (a % b)  # Ten times the remainder -> next digit.

if (a // b) >= 5:  # Do we need to round up?
    digits[-1] += 1  # Round up by incrementing last digit.

if len(digits) <= 1:  # Do we only have an integer part?
    return str((-1 if negative else 1) * digits[0])

digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
if negative:  # Do we need to restore the sign?
    digits.insert(0, "-")  # Insert the sign at the beginning.
return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- Nun, alles, was wir tun müssen, um das zu reparieren, ist zu prüfen ob die nächste Ziffer, die wir anhängen würden, größer oder gleich 5 wäre.
- Wenn ja, dann erhöhen wir die letzte Ziffer in `digits` um `1`.
- Das wird sich später als Fehler herausstellen. . .
- Um ein Aufrunden einzuführen, fügen wir ein `if (a // b)>= 5` ein, das dann `digits[-1] += 1` ausführt, wenn seine Kondition zutrifft.
- An dieser Stelle haben wir jedenfalls eine Repräsentation eines Bruchs als Liste von Ziffern.

```
>>> Fraction(124, 2).decimal_str()
'62'
>>> Fraction(1, 2).decimal_str()
'0.5'
>>> Fraction(1, 3).decimal_str(10)
'0.3333333333'
>>> Fraction(-101001, 100000000).decimal_str()
'-0.00101001'
>>> Fraction(1235, 1000).decimal_str(2)
'1.24'
>>> Fraction(99995, 100000).decimal_str(5)
'0.99995'
>>> Fraction(91995, 100000).decimal_str(3)
'0.92'
>>> Fraction(99995, 100000).decimal_str(4)
'1'
"""
a: int = self.a  # Get the numerator.
if a == 0:  # If the fraction is 0, we return 0.
    return "0"
negative: Final[bool] = a < 0  # Get the sign of the fraction.
a = abs(a)  # Make sure that `a` is now positive.
b: Final[int] = self.b  # Get the denominator.

digits: Final[list] = []  # A list for collecting digits.
while (a != 0) and (len(digits) <= max_frac):  # Create digits.
    digits.append(a // b)  # Add the current digit.
    a = 10 * (a % b)  # Ten times the remainder -> next digit.

if (a // b) >= 5:  # Do we need to round up?
    digits[-1] += 1  # Round up by incrementing last digit.

if len(digits) <= 1:  # Do we only have an integer part?
    return str((-1 if negative else 1) * digits[0])

digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
if negative:  # Do we need to restore the sign?
    digits.insert(0, "-")  # Insert the sign at the beginning.
return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- Wenn ja, dann erhöhen wir die letzte Ziffer in `digits` um `1`.
- Das wird sich später als Fehler herausstellen...
- Um ein Aufrunden einzuführen, fügen wir ein `if (a // b)>= 5` ein, das dann `digits[-1] += 1` ausführt, wenn seine Kondition zutrifft.
- An dieser Stelle haben wir jedenfalls eine Repräsentation eines Bruchs als Liste von Ziffern.
- Jetzt müssen wir diese nur in einen String umrechnen und den dann zurückliefern.

```
>>> Fraction(124, 2).decimal_str()
'62'
>>> Fraction(1, 2).decimal_str()
'0.5'
>>> Fraction(1, 3).decimal_str(10)
'0.3333333333'
>>> Fraction(-101001, 100000000).decimal_str()
'-0.00101001'
>>> Fraction(1235, 1000).decimal_str(2)
'1.24'
>>> Fraction(99995, 100000).decimal_str(5)
'0.99995'
>>> Fraction(91995, 100000).decimal_str(3)
'0.92'
>>> Fraction(99995, 100000).decimal_str(4)
'1'
"""
a: int = self.a  # Get the numerator.
if a == 0:  # If the fraction is 0, we return 0.
    return "0"
negative: Final[bool] = a < 0  # Get the sign of the fraction.
a = abs(a)  # Make sure that `a` is now positive.
b: Final[int] = self.b  # Get the denominator.

digits: Final[list] = []  # A list for collecting digits.
while (a != 0) and (len(digits) <= max_frac):  # Create digits.
    digits.append(a // b)  # Add the current digit.
    a = 10 * (a % b)  # Ten times the remainder -> next digit.

if (a // b) >= 5:  # Do we need to round up?
    digits[-1] += 1  # Round up by incrementing last digit.

if len(digits) <= 1:  # Do we only have an integer part?
    return str((-1 if negative else 1) * digits[0])

digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
if negative:  # Do we need to restore the sign?
    digits.insert(0, "-")  # Insert the sign at the beginning.
return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- Das wird sich später als Fehler herausstellen. . .
- Um ein Aufrunden einzuführen, fügen wir ein `if (a // b)>= 5` ein, das dann `digits[-1] += 1` ausführt, wenn seine Kondition zutrifft.
- An dieser Stelle haben wir jedenfalls eine Repräsentation eines Bruchs als Liste von Ziffern.
- Jetzt müssen wir diese nur in einen String umrechnen und den dann zurückliefern.
- Zuerst schauen wir, ob wir einen Dezimalpunkt („.“) einfügen müssen.

```
>>> Fraction(124, 2).decimal_str()
'62'
>>> Fraction(1, 2).decimal_str()
'0.5'
>>> Fraction(1, 3).decimal_str(10)
'0.3333333333'
>>> Fraction(-101001, 100000000).decimal_str()
'-0.00101001'
>>> Fraction(1235, 1000).decimal_str(2)
'1.24'
>>> Fraction(99995, 100000).decimal_str(5)
'0.99995'
>>> Fraction(91995, 100000).decimal_str(3)
'0.92'
>>> Fraction(99995, 100000).decimal_str(4)
'1'
"""
a: int = self.a  # Get the numerator.
if a == 0:  # If the fraction is 0, we return 0.
    return "0"
negative: Final[bool] = a < 0  # Get the sign of the fraction.
a = abs(a)  # Make sure that `a` is now positive.
b: Final[int] = self.b  # Get the denominator.

digits: Final[list] = []  # A list for collecting digits.
while (a != 0) and (len(digits) <= max_frac):  # Create digits.
    digits.append(a // b)  # Add the current digit.
    a = 10 * (a % b)  # Ten times the remainder -> next digit.

if (a // b) >= 5:  # Do we need to round up?
    digits[-1] += 1  # Round up by incrementing last digit.

if len(digits) <= 1:  # Do we only have an integer part?
    return str((-1 if negative else 1) * digits[0])

digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
if negative:  # Do we need to restore the sign?
    digits.insert(0, "-")  # Insert the sign at the beginning.
return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- Um ein Aufrunden einzuführen, fügen wir ein `if (a // b)>= 5` ein, das dann `digits[-1] += 1` ausführt, wenn seine Kondition zutrifft.
- An dieser Stelle haben wir jedenfalls eine Repräsentation eines Bruchs als Liste von Ziffern.
- Jetzt müssen wir diese nur in einen String umrechnen und den dann zurückliefern.
- Zuerst schauen wir, ob wir einen Dezimalpunkt („.“) einfügen müssen.
- Wenn wir nur eine einzige Ziffer haben, dann ist unser Bruch eine Ganzzahl und wir können ihn genau so zurückliefern.

```
>>> Fraction(124, 2).decimal_str()
'62'
>>> Fraction(1, 2).decimal_str()
'0.5'
>>> Fraction(1, 3).decimal_str(10)
'0.3333333333'
>>> Fraction(-101001, 100000000).decimal_str()
'-0.00101001'
>>> Fraction(1235, 1000).decimal_str(2)
'1.24'
>>> Fraction(99995, 100000).decimal_str(5)
'0.99995'
>>> Fraction(91995, 100000).decimal_str(3)
'0.92'
>>> Fraction(99995, 100000).decimal_str(4)
'1'
"""
a: int = self.a  # Get the numerator.
if a == 0:  # If the fraction is 0, we return 0.
    return "0"
negative: Final[bool] = a < 0  # Get the sign of the fraction.
a = abs(a)  # Make sure that `a` is now positive.
b: Final[int] = self.b  # Get the denominator.

digits: Final[list] = []  # A list for collecting digits.
while (a != 0) and (len(digits) <= max_frac):  # Create digits.
    digits.append(a // b)  # Add the current digit.
    a = 10 * (a % b)  # Ten times the remainder -> next digit.

if (a // b) >= 5:  # Do we need to round up?
    digits[-1] += 1  # Round up by incrementing last digit.

if len(digits) <= 1:  # Do we only have an integer part?
    return str((-1 if negative else 1) * digits[0])

digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
if negative:  # Do we need to restore the sign?
    digits.insert(0, "-")  # Insert the sign at the beginning.
return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- An dieser Stelle haben wir jedenfalls eine Repräsentation eines Bruchs als Liste von Ziffern.
- Jetzt müssen wir diese nur in einen String umrechnen und den dann zurückliefern.
- Zuerst schauen wir, ob wir einen Dezimalpunkt („.“) einfügen müssen.
- Wenn wir nur eine einzige Ziffer haben, dann ist unser Bruch eine Ganzzahl und wir können ihn genau so zurückliefern.
- Wenn `len(digits)<= 1`, dann stellen wir das Vorzeichen wieder her und konvertieren die einzelne zu einem String und liefern diesen zurück.

```
>>> Fraction(124, 2).decimal_str()
'62'
>>> Fraction(1, 2).decimal_str()
'0.5'
>>> Fraction(1, 3).decimal_str(10)
'0.3333333333'
>>> Fraction(-101001, 100000000).decimal_str()
'-0.00101001'
>>> Fraction(1235, 1000).decimal_str(2)
'1.24'
>>> Fraction(99995, 100000).decimal_str(5)
'0.99995'
>>> Fraction(91995, 100000).decimal_str(3)
'0.92'
>>> Fraction(99995, 100000).decimal_str(4)
'1'
"""
a: int = self.a  # Get the numerator.
if a == 0:  # If the fraction is 0, we return 0.
    return "0"
negative: Final[bool] = a < 0  # Get the sign of the fraction.
a = abs(a)  # Make sure that `a` is now positive.
b: Final[int] = self.b  # Get the denominator.

digits: Final[list] = []  # A list for collecting digits.
while (a != 0) and (len(digits) <= max_frac):  # Create digits.
    digits.append(a // b)  # Add the current digit.
    a = 10 * (a % b)  # Ten times the remainder -> next digit.

if (a // b) >= 5:  # Do we need to round up?
    digits[-1] += 1  # Round up by incrementing last digit.

if len(digits) <= 1:  # Do we only have an integer part?
    return str((-1 if negative else 1) * digits[0])

digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
if negative:  # Do we need to restore the sign?
    digits.insert(0, "-")  # Insert the sign at the beginning.
return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- Jetzt müssen wir diese nur in einen String umrechnen und den dann zurückliefern.
- Zuerst schauen wir, ob wir einen Dezimalpunkt („.“) einfügen müssen.
- Wenn wir nur eine einzige Ziffer haben, dann ist unser Bruch eine Ganzzahl und wir können ihn genau so zurückliefern.
- Wenn `len(digits)<= 1`, dann stellen wir das Vorzeichen wieder her und konvertieren die einzelne zu einem String und liefern diesen zurück.
- Sonst müssen wir einen Punkt „.“ nach der ersten Nummer in `digits` einfügen.

```
>>> Fraction(124, 2).decimal_str()
'62'
>>> Fraction(1, 2).decimal_str()
'0.5'
>>> Fraction(1, 3).decimal_str(10)
'0.3333333333'
>>> Fraction(-101001, 100000000).decimal_str()
'-0.00101001'
>>> Fraction(1235, 1000).decimal_str(2)
'1.24'
>>> Fraction(99995, 100000).decimal_str(5)
'0.99995'
>>> Fraction(91995, 100000).decimal_str(3)
'0.92'
>>> Fraction(99995, 100000).decimal_str(4)
'1'
"""
a: int = self.a  # Get the numerator.
if a == 0:  # If the fraction is 0, we return 0.
    return "0"
negative: Final[bool] = a < 0  # Get the sign of the fraction.
a = abs(a)  # Make sure that `a` is now positive.
b: Final[int] = self.b  # Get the denominator.

digits: Final[list] = []  # A list for collecting digits.
while (a != 0) and (len(digits) <= max_frac):  # Create digits.
    digits.append(a // b)  # Add the current digit.
    a = 10 * (a % b)  # Ten times the remainder -> next digit.

if (a // b) >= 5:  # Do we need to round up?
    digits[-1] += 1  # Round up by incrementing last digit.

if len(digits) <= 1:  # Do we only have an integer part?
    return str((-1 if negative else 1) * digits[0])

digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
if negative:  # Do we need to restore the sign?
    digits.insert(0, "-")  # Insert the sign at the beginning.
return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- Zuerst schauen wir, ob wir einen Dezimalpunkt („.“) einfügen müssen.
- Wenn wir nur eine einzige Ziffer haben, dann ist unser Bruch eine Ganzzahl und wir können ihn genau so zurückliefern.
- Wenn `len(digits)<= 1`, dann stellen wir das Vorzeichen wieder her und konvertieren die einzelne zu einem String und liefern diesen zurück.
- Sonst müssen wir einen Punkt „.“ nach der ersten Nummer in `digits` einfügen.
- Das geht via `digits.insert(1, ".")`.

```
>>> Fraction(124, 2).decimal_str()
'62'
>>> Fraction(1, 2).decimal_str()
'0.5'
>>> Fraction(1, 3).decimal_str(10)
'0.3333333333'
>>> Fraction(-101001, 100000000).decimal_str()
'-0.00101001'
>>> Fraction(1235, 1000).decimal_str(2)
'1.24'
>>> Fraction(99995, 100000).decimal_str(5)
'0.99995'
>>> Fraction(91995, 100000).decimal_str(3)
'0.92'
>>> Fraction(99995, 100000).decimal_str(4)
'1'
"""
a: int = self.a  # Get the numerator.
if a == 0:  # If the fraction is 0, we return 0.
    return "0"
negative: Final[bool] = a < 0  # Get the sign of the fraction.
a = abs(a)  # Make sure that `a` is now positive.
b: Final[int] = self.b  # Get the denominator.

digits: Final[list] = []  # A list for collecting digits.
while (a != 0) and (len(digits) <= max_frac):  # Create digits.
    digits.append(a // b)  # Add the current digit.
    a = 10 * (a % b)  # Ten times the remainder -> next digit.

if (a // b) >= 5:  # Do we need to round up?
    digits[-1] += 1  # Round up by incrementing last digit.

if len(digits) <= 1:  # Do we only have an integer part?
    return str((-1 if negative else 1) * digits[0])

digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
if negative:  # Do we need to restore the sign?
    digits.insert(0, "-")  # Insert the sign at the beginning.
return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- Wenn wir nur eine einzige Ziffer haben, dann ist unser Bruch eine Ganzzahl und wir können ihn genau so zurückliefern.
- Wenn `len(digits)<= 1`, dann stellen wir das Vorzeichen wieder her und konvertieren die einzelne zu einem String und liefern diesen zurück.
- Sonst müssen wir einen Punkt „.“ nach der ersten Nummer in `digits` einfügen.
- Das geht via `digits.insert(1, ".")`.
- In unserem Beispiel von $\frac{-179}{16}$ hatten wir erst `digits == [11, 1, 8, 7, 5]`.

```
>>> Fraction(124, 2).decimal_str()
'62'
>>> Fraction(1, 2).decimal_str()
'0.5'
>>> Fraction(1, 3).decimal_str(10)
'0.3333333333'
>>> Fraction(-101001, 100000000).decimal_str()
'-0.00101001'
>>> Fraction(1235, 1000).decimal_str(2)
'1.24'
>>> Fraction(99995, 100000).decimal_str(5)
'0.99995'
>>> Fraction(91995, 100000).decimal_str(3)
'0.92'
>>> Fraction(99995, 100000).decimal_str(4)
'1'
"""
a: int = self.a  # Get the numerator.
if a == 0:  # If the fraction is 0, we return 0.
    return "0"
negative: Final[bool] = a < 0  # Get the sign of the fraction.
a = abs(a)  # Make sure that `a` is now positive.
b: Final[int] = self.b  # Get the denominator.

digits: Final[list] = []  # A list for collecting digits.
while (a != 0) and (len(digits) <= max_frac):  # Create digits.
    digits.append(a // b)  # Add the current digit.
    a = 10 * (a % b)  # Ten times the remainder -> next digit.

if (a // b) >= 5:  # Do we need to round up?
    digits[-1] += 1  # Round up by incrementing last digit.

if len(digits) <= 1:  # Do we only have an integer part?
    return str((-1 if negative else 1) * digits[0])

digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
if negative:  # Do we need to restore the sign?
    digits.insert(0, "-")  # Insert the sign at the beginning.
return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- Wenn `len(digits)<= 1`, dann stellen wir das Vorzeichen wieder her und konvertieren die einzelne zu einem String und liefern diesen zurück.
- Sonst müssen wir einen Punkt „.“ nach der ersten Nummer in `digits` einfügen.
- Das geht via `digits.insert(1, ".")`.
- In unserem Beispiel von $\frac{-179}{16}$ hatten wir erst `digits == [11, 1, 8, 7, 5]`.
- Nach diesem Schritt haben wir `digits == [11, ".", 1, 8, 7, 5]`.

```
    >>> Fraction(124, 2).decimal_str()
    '62'
    >>> Fraction(1, 2).decimal_str()
    '0.5'
    >>> Fraction(1, 3).decimal_str(10)
    '0.3333333333'
    >>> Fraction(-101001, 100000000).decimal_str()
    '-0.00101001'
    >>> Fraction(1235, 1000).decimal_str(2)
    '1.24'
    >>> Fraction(99995, 100000).decimal_str(5)
    '0.99995'
    >>> Fraction(91995, 100000).decimal_str(3)
    '0.92'
    >>> Fraction(99995, 100000).decimal_str(4)
    '1'
    """
    a: int = self.a  # Get the numerator.
    if a == 0:  # If the fraction is 0, we return 0.
        return "0"
    negative: Final[bool] = a < 0  # Get the sign of the fraction.
    a = abs(a)  # Make sure that `a` is now positive.
    b: Final[int] = self.b  # Get the denominator.

    digits: Final[list] = []  # A list for collecting digits.
    while (a != 0) and (len(digits) <= max_frac):  # Create digits.
        digits.append(a // b)  # Add the current digit.
        a = 10 * (a % b)  # Ten times the remainder -> next digit.

    if (a // b) >= 5:  # Do we need to round up?
        digits[-1] += 1  # Round up by incrementing last digit.

    if len(digits) <= 1:  # Do we only have an integer part?
        return str((-1 if negative else 1) * digits[0])

    digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
    if negative:  # Do we need to restore the sign?
        digits.insert(0, "-")  # Insert the sign at the beginning.
    return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- Sonst müssen wir einen Punkt „.“ nach der ersten Nummer in `digits` einfügen.
- Das geht via `digits.insert(1, ".")`.
- In unserem Beispiel von $\frac{-179}{16}$ hatten wir erst `digits == [11, 1, 8, 7, 5]`.
- Nach diesem Schritt haben wir `digits == [11, ".", 1, 8, 7, 5]`.
- Wenn der Bruch negativ war, also wenn `negative` wahr ist, dann fügen wir ein Minus vorne an die Liste an, via `digits.insert(0, "-")`.

```
>>> Fraction(124, 2).decimal_str()
'62'
>>> Fraction(1, 2).decimal_str()
'0.5'
>>> Fraction(1, 3).decimal_str(10)
'0.3333333333'
>>> Fraction(-101001, 100000000).decimal_str()
'-0.00101001'
>>> Fraction(1235, 1000).decimal_str(2)
'1.24'
>>> Fraction(99995, 100000).decimal_str(5)
'0.99995'
>>> Fraction(91995, 100000).decimal_str(3)
'0.92'
>>> Fraction(99995, 100000).decimal_str(4)
'1'
"""
a: int = self.a  # Get the numerator.
if a == 0:  # If the fraction is 0, we return 0.
    return "0"
negative: Final[bool] = a < 0  # Get the sign of the fraction.
a = abs(a)  # Make sure that `a` is now positive.
b: Final[int] = self.b  # Get the denominator.

digits: Final[list] = []  # A list for collecting digits.
while (a != 0) and (len(digits) <= max_frac):  # Create digits.
    digits.append(a // b)  # Add the current digit.
    a = 10 * (a % b)  # Ten times the remainder -> next digit.

if (a // b) >= 5:  # Do we need to round up?
    digits[-1] += 1  # Round up by incrementing last digit.

if len(digits) <= 1:  # Do we only have an integer part?
    return str((-1 if negative else 1) * digits[0])

digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
if negative:  # Do we need to restore the sign?
    digits.insert(0, "-")  # Insert the sign at the beginning.
return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- Das geht via `digits.insert(1, ".")`.
- In unserem Beispiel von $\frac{-179}{16}$ hatten wir erst `digits == [11, 1, 8, 7, 5]`.
- Nach diesem Schritt haben wir `digits == [11, ".", 1, 8, 7, 5]`.
- Wenn der Bruch negativ war, also wenn `negative` wahr ist, dann fügen wir ein Minus vorne an die Liste an, via `digits.insert(0, "-")`.
- In unserem Beispiel bekommen wir also `digits == ["-", 11, ".", 1, 8, 7, 5]`.

```
>>> Fraction(124, 2).decimal_str()
'62'
>>> Fraction(1, 2).decimal_str()
'0.5'
>>> Fraction(1, 3).decimal_str(10)
'0.3333333333'
>>> Fraction(-101001, 100000000).decimal_str()
'-0.00101001'
>>> Fraction(1235, 1000).decimal_str(2)
'1.24'
>>> Fraction(99995, 100000).decimal_str(5)
'0.99995'
>>> Fraction(91995, 100000).decimal_str(3)
'0.92'
>>> Fraction(99995, 100000).decimal_str(4)
'1'
"""
a: int = self.a  # Get the numerator.
if a == 0:  # If the fraction is 0, we return 0.
    return "0"
negative: Final[bool] = a < 0  # Get the sign of the fraction.
a = abs(a)  # Make sure that `a` is now positive.
b: Final[int] = self.b  # Get the denominator.

digits: Final[list] = []  # A list for collecting digits.
while (a != 0) and (len(digits) <= max_frac):  # Create digits.
    digits.append(a // b)  # Add the current digit.
    a = 10 * (a % b)  # Ten times the remainder -> next digit.

if (a // b) >= 5:  # Do we need to round up?
    digits[-1] += 1  # Round up by incrementing last digit.

if len(digits) <= 1:  # Do we only have an integer part?
    return str((-1 if negative else 1) * digits[0])

digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
if negative:  # Do we need to restore the sign?
    digits.insert(0, "-")  # Insert the sign at the beginning.
return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- In unserem Beispiel von $\frac{-179}{16}$ hatten wir erst `digits == [11, 1, 8, 7, 5]`.
- Nach diesem Schritt haben wir `digits == [11, ".", 1, 8, 7, 5]`.
- Wenn der Bruch negativ war, also wenn `negative` wahr ist, dann fügen wir ein Minus vorne an die Liste an, via `digits.insert(0, "-")`.
- In unserem Beispiel bekommen wir also `digits == ["-", 11, ".", 1, 8, 7, 5]`.
- Alles, was wir jetzt noch machen müssen, ist alle Ganzzahlen in Strings zu übersetzen und dann alle Strings aneinander anzuhängen.

```
>>> Fraction(124, 2).decimal_str()
'62'
>>> Fraction(1, 2).decimal_str()
'0.5'
>>> Fraction(1, 3).decimal_str(10)
'0.3333333333'
>>> Fraction(-101001, 100000000).decimal_str()
'-0.00101001'
>>> Fraction(1235, 1000).decimal_str(2)
'1.24'
>>> Fraction(99995, 100000).decimal_str(5)
'0.99995'
>>> Fraction(91995, 100000).decimal_str(3)
'0.92'
>>> Fraction(99995, 100000).decimal_str(4)
'1'
"""
a: int = self.a  # Get the numerator.
if a == 0:  # If the fraction is 0, we return 0.
    return "0"
negative: Final[bool] = a < 0  # Get the sign of the fraction.
a = abs(a)  # Make sure that `a` is now positive.
b: Final[int] = self.b  # Get the denominator.

digits: Final[list] = []  # A list for collecting digits.
while (a != 0) and (len(digits) <= max_frac):  # Create digits.
    digits.append(a // b)  # Add the current digit.
    a = 10 * (a % b)  # Ten times the remainder -> next digit.

if (a // b) >= 5:  # Do we need to round up?
    digits[-1] += 1  # Round up by incrementing last digit.

if len(digits) <= 1:  # Do we only have an integer part?
    return str((-1 if negative else 1) * digits[0])

digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
if negative:  # Do we need to restore the sign?
    digits.insert(0, "-")  # Insert the sign at the beginning.
return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- Nach diesem Schritt haben wir `digits == [11, ".", 1, 8, 7, 5]`.
- Wenn der Bruch negativ war, also wenn `negative` wahr ist, dann fügen wir ein Minus vorne an die Liste an, via `digits.insert(0, "-")`.
- In unserem Beispiel bekommen wir also `digits == ["-", 11, ".", 1, 8, 7, 5]`.
- Alles, was wir jetzt noch machen müssen, ist alle Ganzzahlen in Strings zu übersetzen und dann alle Strings aneinander anzuhängen.
- Das schaffen wir mit einer einzigen Zeile Kode: `"".join(map(str, digits))`.

```
>>> Fraction(124, 2).decimal_str()
'62'
>>> Fraction(1, 2).decimal_str()
'0.5'
>>> Fraction(1, 3).decimal_str(10)
'0.3333333333'
>>> Fraction(-101001, 100000000).decimal_str()
'-0.00101001'
>>> Fraction(1235, 1000).decimal_str(2)
'1.24'
>>> Fraction(99995, 100000).decimal_str(5)
'0.99995'
>>> Fraction(91995, 100000).decimal_str(3)
'0.92'
>>> Fraction(99995, 100000).decimal_str(4)
'1'
"""
a: int = self.a  # Get the numerator.
if a == 0:  # If the fraction is 0, we return 0.
    return "0"
negative: Final[bool] = a < 0  # Get the sign of the fraction.
a = abs(a)  # Make sure that `a` is now positive.
b: Final[int] = self.b  # Get the denominator.

digits: Final[list] = []  # A list for collecting digits.
while (a != 0) and (len(digits) <= max_frac):  # Create digits.
    digits.append(a // b)  # Add the current digit.
    a = 10 * (a % b)  # Ten times the remainder -> next digit.

if (a // b) >= 5:  # Do we need to round up?
    digits[-1] += 1  # Round up by incrementing last digit.

if len(digits) <= 1:  # Do we only have an integer part?
    return str((-1 if negative else 1) * digits[0])

digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
if negative:  # Do we need to restore the sign?
    digits.insert(0, "-")  # Insert the sign at the beginning.
return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- In unserem Beispiel bekommen wir also `digits == ["-", 11, ".", 1, 8, 7, 5]`.
- Alles, was wir jetzt noch machen müssen, ist alle Ganzzahlen in Strings zu übersetzen und dann alle Strings aneinander anzuhängen.
- Das schaffen wir mit einer einzigen Zeile Kode: `"".join(map(str, digits))`.
- `map` liefert uns einen `Iterator`, der die Ergebnisse der Funktion `str` angewandt auf die Elemente von `digits` eins nach dem Anderen zurückliefert.

```
>>> Fraction(124, 2).decimal_str()
'62'
>>> Fraction(1, 2).decimal_str()
'0.5'
>>> Fraction(1, 3).decimal_str(10)
'0.3333333333'
>>> Fraction(-101001, 100000000).decimal_str()
'-0.00101001'
>>> Fraction(1235, 1000).decimal_str(2)
'1.24'
>>> Fraction(99995, 100000).decimal_str(5)
'0.99995'
>>> Fraction(91995, 100000).decimal_str(3)
'0.92'
>>> Fraction(99995, 100000).decimal_str(4)
'1'
"""
a: int = self.a  # Get the numerator.
if a == 0:  # If the fraction is 0, we return 0.
    return "0"
negative: Final[bool] = a < 0  # Get the sign of the fraction.
a = abs(a)  # Make sure that `a` is now positive.
b: Final[int] = self.b  # Get the denominator.

digits: Final[list] = []  # A list for collecting digits.
while (a != 0) and (len(digits) <= max_frac):  # Create digits.
    digits.append(a // b)  # Add the current digit.
    a = 10 * (a % b)  # Ten times the remainder -> next digit.

if (a // b) >= 5:  # Do we need to round up?
    digits[-1] += 1  # Round up by incrementing last digit.

if len(digits) <= 1:  # Do we only have an integer part?
    return str((-1 if negative else 1) * digits[0])

digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
if negative:  # Do we need to restore the sign?
    digits.insert(0, "-")  # Insert the sign at the beginning.
return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- Alles, was wir jetzt noch machen müssen, ist alle Ganzzahlen in Strings zu übersetzen und dann alle Strings aneinander anzuhängen.
- Das schaffen wir mit einer einzigen Zeile Kode: `"".join(map(str, digits))`.
- `map` liefert uns einen `Iterator`, der die Ergebnisse der Funktion `str` angewandt auf die Elemente von `digits` eins nach dem Anderen zurückliefert.
- `str` auf einen String angewandt liefert den String direkt zurück.

```
    >>> Fraction(124, 2).decimal_str()
    '62'
    >>> Fraction(1, 2).decimal_str()
    '0.5'
    >>> Fraction(1, 3).decimal_str(10)
    '0.3333333333'
    >>> Fraction(-101001, 100000000).decimal_str()
    '-0.00101001'
    >>> Fraction(1235, 1000).decimal_str(2)
    '1.24'
    >>> Fraction(99995, 100000).decimal_str(5)
    '0.99995'
    >>> Fraction(91995, 100000).decimal_str(3)
    '0.92'
    >>> Fraction(99995, 100000).decimal_str(4)
    '1'
    """
    a: int = self.a  # Get the numerator.
    if a == 0:  # If the fraction is 0, we return 0.
        return "0"
    negative: Final[bool] = a < 0  # Get the sign of the fraction.
    a = abs(a)  # Make sure that `a` is now positive.
    b: Final[int] = self.b  # Get the denominator.

    digits: Final[list] = []  # A list for collecting digits.
    while (a != 0) and (len(digits) <= max_frac):  # Create digits.
        digits.append(a // b)  # Add the current digit.
        a = 10 * (a % b)  # Ten times the remainder -> next digit.

    if (a // b) >= 5:  # Do we need to round up?
        digits[-1] += 1  # Round up by incrementing last digit.

    if len(digits) <= 1:  # Do we only have an integer part?
        return str((-1 if negative else 1) * digits[0])

    digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
    if negative:  # Do we need to restore the sign?
        digits.insert(0, "-")  # Insert the sign at the beginning.
    return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- Alles, was wir jetzt noch machen müssen, ist alle Ganzzahlen in Strings zu übersetzen und dann alle Strings aneinander anzuhängen.
- Das schaffen wir mit einer einzigen Zeile Kode: `"".join(map(str, digits))`.
- `map` liefert uns einen `Iterator`, der die Ergebnisse der Funktion `str` angewandt auf die Elemente von `digits` eins nach dem Anderen zurückliefert.
- `str` auf einen String angewandt liefert den String direkt zurück.
- Angewandt auf eine Ganzzahl konvertiert es diese zu einem String.

```
    >>> Fraction(124, 2).decimal_str()
    '62'
    >>> Fraction(1, 2).decimal_str()
    '0.5'
    >>> Fraction(1, 3).decimal_str(10)
    '0.3333333333'
    >>> Fraction(-101001, 100000000).decimal_str()
    '-0.00101001'
    >>> Fraction(1235, 1000).decimal_str(2)
    '1.24'
    >>> Fraction(99995, 100000).decimal_str(5)
    '0.99995'
    >>> Fraction(91995, 100000).decimal_str(3)
    '0.92'
    >>> Fraction(99995, 100000).decimal_str(4)
    '1'
    """
    a: int = self.a  # Get the numerator.
    if a == 0:  # If the fraction is 0, we return 0.
        return "0"
    negative: Final[bool] = a < 0  # Get the sign of the fraction.
    a = abs(a)  # Make sure that `a` is now positive.
    b: Final[int] = self.b  # Get the denominator.

    digits: Final[list] = []  # A list for collecting digits.
    while (a != 0) and (len(digits) <= max_frac):  # Create digits.
        digits.append(a // b)  # Add the current digit.
        a = 10 * (a % b)  # Ten times the remainder -> next digit.

    if (a // b) >= 5:  # Do we need to round up?
        digits[-1] += 1  # Round up by incrementing last digit.

    if len(digits) <= 1:  # Do we only have an integer part?
        return str((-1 if negative else 1) * digits[0])

    digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
    if negative:  # Do we need to restore the sign?
        digits.insert(0, "-")  # Insert the sign at the beginning.
    return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- `map` liefert uns einen `Iterator`, der die Ergebnisse der Funktion `str` angewandt auf die Elemente von `digits` eins nach dem Anderen zurückliefert.
- `str` auf einen String angewandt liefert den String direkt zurück.
- Angewandt auf eine Ganzzahl konvertiert es diese zu einem String.
- Die Methode `join` eines Strings hängt alle Element des `Iterables`, das sie als Parameter bekonnt, aneinander und verwendet den String selbst als Separator.

```
    >>> Fraction(124, 2).decimal_str()
    '62'
    >>> Fraction(1, 2).decimal_str()
    '0.5'
    >>> Fraction(1, 3).decimal_str(10)
    '0.3333333333'
    >>> Fraction(-101001, 100000000).decimal_str()
    '-0.00101001'
    >>> Fraction(1235, 1000).decimal_str(2)
    '1.24'
    >>> Fraction(99995, 100000).decimal_str(5)
    '0.99995'
    >>> Fraction(91995, 100000).decimal_str(3)
    '0.92'
    >>> Fraction(99995, 100000).decimal_str(4)
    '1'
    """
    a: int = self.a  # Get the numerator.
    if a == 0:  # If the fraction is 0, we return 0.
        return "0"
    negative: Final[bool] = a < 0  # Get the sign of the fraction.
    a = abs(a)  # Make sure that `a` is now positive.
    b: Final[int] = self.b  # Get the denominator.

    digits: Final[list] = []  # A list for collecting digits.
    while (a != 0) and (len(digits) <= max_frac):  # Create digits.
        digits.append(a // b)  # Add the current digit.
        a = 10 * (a % b)  # Ten times the remainder -> next digit.

    if (a // b) >= 5:  # Do we need to round up?
        digits[-1] += 1  # Round up by incrementing last digit.

    if len(digits) <= 1:  # Do we only have an integer part?
        return str((-1 if negative else 1) * digits[0])

    digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
    if negative:  # Do we need to restore the sign?
        digits.insert(0, "-")  # Insert the sign at the beginning.
    return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- `map` liefert uns einen `Iterator`, der die Ergebnisse der Funktion `str` angewandt auf die Elemente von `digits` eins nach dem Anderen zurückliefert.
- `str` auf einen String angewandt liefert den String direkt zurück.
- Angewandt auf eine Ganzzahl konvertiert es diese zu einem String.
- Die Methode `join` eines Strings hängt alle Element des `Iterables`, das sie als Parameter bekonnt, aneinander und verwendet den String selbst als Separator.
- `"X".join(["a", "b", "c"])` z. B. würde `"aXbXc"` ergeben.

```
    >>> Fraction(124, 2).decimal_str()
    '62'
    >>> Fraction(1, 2).decimal_str()
    '0.5'
    >>> Fraction(1, 3).decimal_str(10)
    '0.3333333333'
    >>> Fraction(-101001, 100000000).decimal_str()
    '-0.00101001'
    >>> Fraction(1235, 1000).decimal_str(2)
    '1.24'
    >>> Fraction(99995, 100000).decimal_str(5)
    '0.99995'
    >>> Fraction(91995, 100000).decimal_str(3)
    '0.92'
    >>> Fraction(99995, 100000).decimal_str(4)
    '1'
    """
    a: int = self.a  # Get the numerator.
    if a == 0:  # If the fraction is 0, we return 0.
        return "0"
    negative: Final[bool] = a < 0  # Get the sign of the fraction.
    a = abs(a)  # Make sure that `a` is now positive.
    b: Final[int] = self.b  # Get the denominator.

    digits: Final[list] = []  # A list for collecting digits.
    while (a != 0) and (len(digits) <= max_frac):  # Create digits.
        digits.append(a // b)  # Add the current digit.
        a = 10 * (a % b)  # Ten times the remainder -> next digit.

    if (a // b) >= 5:  # Do we need to round up?
        digits[-1] += 1  # Round up by incrementing last digit.

    if len(digits) <= 1:  # Do we only have an integer part?
        return str((-1 if negative else 1) * digits[0])

    digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
    if negative:  # Do we need to restore the sign?
        digits.insert(0, "-")  # Insert the sign at the beginning.
    return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- `str` auf einen String angewandt liefert den String direkt zurück.
- Angewandt auf eine Ganzzahl konvertiert es diese zu einem String.
- Die Methode `join` eines Strings hängt alle Element des `Iterables`, das sie als Parameter bekonnt, aneinander und verwendet den String selbst als Separator.
- `"X".join(["a", "b", "c"])` z. B. würde `"aXbXc"` ergeben.
- Wir benutzen den leeren String als Separator, deshalb bekommen wir in unserem Beispiel `"-11.1875"`.

```
>>> Fraction(124, 2).decimal_str()
'62'
>>> Fraction(1, 2).decimal_str()
'0.5'
>>> Fraction(1, 3).decimal_str(10)
'0.3333333333'
>>> Fraction(-101001, 100000000).decimal_str()
'-0.00101001'
>>> Fraction(1235, 1000).decimal_str(2)
'1.24'
>>> Fraction(99995, 100000).decimal_str(5)
'0.99995'
>>> Fraction(91995, 100000).decimal_str(3)
'0.92'
>>> Fraction(99995, 100000).decimal_str(4)
'1'
"""
a: int = self.a  # Get the numerator.
if a == 0:  # If the fraction is 0, we return 0.
    return "0"
negative: Final[bool] = a < 0  # Get the sign of the fraction.
a = abs(a)  # Make sure that `a` is now positive.
b: Final[int] = self.b  # Get the denominator.

digits: Final[list] = []  # A list for collecting digits.
while (a != 0) and (len(digits) <= max_frac):  # Create digits.
    digits.append(a // b)  # Add the current digit.
    a = 10 * (a % b)  # Ten times the remainder -> next digit.

if (a // b) >= 5:  # Do we need to round up?
    digits[-1] += 1  # Round up by incrementing last digit.

if len(digits) <= 1:  # Do we only have an integer part?
    return str((-1 if negative else 1) * digits[0])

digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
if negative:  # Do we need to restore the sign?
    digits.insert(0, "-")  # Insert the sign at the beginning.
return "".join(map(str, digits))  # Join all digits to a string.
```

`Fraction`: Doctests

- Nachdem wir mit der Implementierung von `decimal_str` fertig sind, müssen wir die Methode noch testen.

```
    >>> Fraction(124, 2).decimal_str()
    '62'
    >>> Fraction(1, 2).decimal_str()
    '0.5'
    >>> Fraction(1, 3).decimal_str(10)
    '0.3333333333'
    >>> Fraction(-101001, 100000000).decimal_str()
    '-0.00101001'
    >>> Fraction(1235, 1000).decimal_str(2)
    '1.24'
    >>> Fraction(99995, 100000).decimal_str(5)
    '0.99995'
    >>> Fraction(91995, 100000).decimal_str(3)
    '0.92'
    >>> Fraction(99995, 100000).decimal_str(4)
    '1'
    """
    a: int = self.a  # Get the numerator.
    if a == 0:  # If the fraction is 0, we return 0.
        return "0"
    negative: Final[bool] = a < 0  # Get the sign of the fraction.
    a = abs(a)  # Make sure that `a` is now positive.
    b: Final[int] = self.b  # Get the denominator.

    digits: Final[list] = []  # A list for collecting digits.
    while (a != 0) and (len(digits) <= max_frac):  # Create digits.
        digits.append(a // b)  # Add the current digit.
        a = 10 * (a % b)  # Ten times the remainder -> next digit.

    if (a // b) >= 5:  # Do we need to round up?
        digits[-1] += 1  # Round up by incrementing last digit.

    if len(digits) <= 1:  # Do we only have an integer part?
        return str((-1 if negative else 1) * digits[0])

    digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
    if negative:  # Do we need to restore the sign?
        digits.insert(0, "-")  # Insert the sign at the beginning.
    return "".join(map(str, digits))  # Join all digits to a string.
```

`Fraction`: Doctests

- Nachdem wir mit der Implementierung von `decimal_str` fertig sind, müssen wir die Methode noch testen.
- Dafür können wir ja Doctests in den Docstring der Methode schreiben.

```
    >>> Fraction(124, 2).decimal_str()
    '62'
    >>> Fraction(1, 2).decimal_str()
    '0.5'
    >>> Fraction(1, 3).decimal_str(10)
    '0.3333333333'
    >>> Fraction(-101001, 100000000).decimal_str()
    '-0.00101001'
    >>> Fraction(1235, 1000).decimal_str(2)
    '1.24'
    >>> Fraction(99995, 100000).decimal_str(5)
    '0.99995'
    >>> Fraction(91995, 100000).decimal_str(3)
    '0.92'
    >>> Fraction(99995, 100000).decimal_str(4)
    '1'
    """
    a: int = self.a  # Get the numerator.
    if a == 0:  # If the fraction is 0, we return 0.
        return "0"
    negative: Final[bool] = a < 0  # Get the sign of the fraction.
    a = abs(a)  # Make sure that `a` is now positive.
    b: Final[int] = self.b  # Get the denominator.

    digits: Final[list] = []  # A list for collecting digits.
    while (a != 0) and (len(digits) <= max_frac):  # Create digits.
        digits.append(a // b)  # Add the current digit.
        a = 10 * (a % b)  # Ten times the remainder -> next digit.

    if (a // b) >= 5:  # Do we need to round up?
        digits[-1] += 1  # Round up by incrementing last digit.

    if len(digits) <= 1:  # Do we only have an integer part?
        return str((-1 if negative else 1) * digits[0])

    digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
    if negative:  # Do we need to restore the sign?
        digits.insert(0, "-")  # Insert the sign at the beginning.
    return "".join(map(str, digits))  # Join all digits to a string.
```

`Fraction`: Doctests

- Nachdem wir mit der Implementierung von `decimal_str` fertig sind, müssen wir die Methode noch testen.
- Dafür können wir ja Doctests in den Docstring der Methode schreiben.
- Wir machen das mit einigen normalen Fällen und ein paar extremen Situationen.

```
    >>> Fraction(124, 2).decimal_str()
    '62'
    >>> Fraction(1, 2).decimal_str()
    '0.5'
    >>> Fraction(1, 3).decimal_str(10)
    '0.3333333333'
    >>> Fraction(-101001, 100000000).decimal_str()
    '-0.00101001'
    >>> Fraction(1235, 1000).decimal_str(2)
    '1.24'
    >>> Fraction(99995, 100000).decimal_str(5)
    '0.99995'
    >>> Fraction(91995, 100000).decimal_str(3)
    '0.92'
    >>> Fraction(99995, 100000).decimal_str(4)
    '1'
    """
    a: int = self.a  # Get the numerator.
    if a == 0:  # If the fraction is 0, we return 0.
        return "0"
    negative: Final[bool] = a < 0  # Get the sign of the fraction.
    a = abs(a)  # Make sure that `a` is now positive.
    b: Final[int] = self.b  # Get the denominator.

    digits: Final[list] = []  # A list for collecting digits.
    while (a != 0) and (len(digits) <= max_frac):  # Create digits.
        digits.append(a // b)  # Add the current digit.
        a = 10 * (a % b)  # Ten times the remainder -> next digit.

    if (a // b) >= 5:  # Do we need to round up?
        digits[-1] += 1  # Round up by incrementing last digit.

    if len(digits) <= 1:  # Do we only have an integer part?
        return str((-1 if negative else 1) * digits[0])

    digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
    if negative:  # Do we need to restore the sign?
        digits.insert(0, "-")  # Insert the sign at the beginning.
    return "".join(map(str, digits))  # Join all digits to a string.
```

`Fraction`: Doctests

- Nachdem wir mit der Implementierung von `decimal_str` fertig sind, müssen wir die Methode noch testen.
- Dafür können wir ja Doctests in den Docstring der Methode schreiben.
- Wir machen das mit einigen normalen Fällen und ein paar extremen Situationen.
- Zuerst prüfen wir, ob Ganzzahlen ordentlich dargestellt werden.

```
    >>> Fraction(124, 2).decimal_str()
    '62'
    >>> Fraction(1, 2).decimal_str()
    '0.5'
    >>> Fraction(1, 3).decimal_str(10)
    '0.3333333333'
    >>> Fraction(-101001, 100000000).decimal_str()
    '-0.00101001'
    >>> Fraction(1235, 1000).decimal_str(2)
    '1.24'
    >>> Fraction(99995, 100000).decimal_str(5)
    '0.99995'
    >>> Fraction(91995, 100000).decimal_str(3)
    '0.92'
    >>> Fraction(99995, 100000).decimal_str(4)
    '1'
    """
    a: int = self.a  # Get the numerator.
    if a == 0:  # If the fraction is 0, we return 0.
        return "0"
    negative: Final[bool] = a < 0  # Get the sign of the fraction.
    a = abs(a)  # Make sure that `a` is now positive.
    b: Final[int] = self.b  # Get the denominator.

    digits: Final[list] = []  # A list for collecting digits.
    while (a != 0) and (len(digits) <= max_frac):  # Create digits.
        digits.append(a // b)  # Add the current digit.
        a = 10 * (a % b)  # Ten times the remainder -> next digit.

    if (a // b) >= 5:  # Do we need to round up?
        digits[-1] += 1  # Round up by incrementing last digit.

    if len(digits) <= 1:  # Do we only have an integer part?
        return str((-1 if negative else 1) * digits[0])

    digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
    if negative:  # Do we need to restore the sign?
        digits.insert(0, "-")  # Insert the sign at the beginning.
    return "".join(map(str, digits))  # Join all digits to a string.
```

`Fraction`: Doctests

- Nachdem wir mit der Implementierung von `decimal_str` fertig sind, müssen wir die Methode noch testen.
- Dafür können wir ja Doctests in den Docstring der Methode schreiben.
- Wir machen das mit einigen normalen Fällen und ein paar extremen Situationen.
- Zuerst prüfen wir, ob Ganzzahlen ordentlich dargestellt werden.
- Es ist klar, dass `Fraction(124, 2).decimal_str()` das Ergebnis `"62"` haben muss.

```
    >>> Fraction(124, 2).decimal_str()
    '62'
    >>> Fraction(1, 2).decimal_str()
    '0.5'
    >>> Fraction(1, 3).decimal_str(10)
    '0.3333333333'
    >>> Fraction(-101001, 100000000).decimal_str()
    '-0.00101001'
    >>> Fraction(1235, 1000).decimal_str(2)
    '1.24'
    >>> Fraction(99995, 100000).decimal_str(5)
    '0.99995'
    >>> Fraction(91995, 100000).decimal_str(3)
    '0.92'
    >>> Fraction(99995, 100000).decimal_str(4)
    '1'
    """
    a: int = self.a  # Get the numerator.
    if a == 0:  # If the fraction is 0, we return 0.
        return "0"
    negative: Final[bool] = a < 0  # Get the sign of the fraction.
    a = abs(a)  # Make sure that `a` is now positive.
    b: Final[int] = self.b  # Get the denominator.

    digits: Final[list] = []  # A list for collecting digits.
    while (a != 0) and (len(digits) <= max_frac):  # Create digits.
        digits.append(a // b)  # Add the current digit.
        a = 10 * (a % b)  # Ten times the remainder -> next digit.

    if (a // b) >= 5:  # Do we need to round up?
        digits[-1] += 1  # Round up by incrementing last digit.

    if len(digits) <= 1:  # Do we only have an integer part?
        return str((-1 if negative else 1) * digits[0])

    digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
    if negative:  # Do we need to restore the sign?
        digits.insert(0, "-")  # Insert the sign at the beginning.
    return "".join(map(str, digits))  # Join all digits to a string.
```

`Fraction`: Doctests

- Dafür können wir ja Doctests in den Docstring der Methode schreiben.
- Wir machen das mit einigen normalen Fällen und ein paar extremen Situationen.
- Zuerst prüfen wir, ob Ganzzahlen ordentlich dargestellt werden.
- Es ist klar, dass `Fraction(124, 2).decimal_str()` das Ergebnis `"62"` haben muss.
- Dann prüfen wir, ob ein normaler Bruch genau übersetzt wird: `Fraction(1, 2).decimal_str()` sollte `"0.5"` ergeben.

```
    >>> Fraction(124, 2).decimal_str()
    '62'
    >>> Fraction(1, 2).decimal_str()
    '0.5'
    >>> Fraction(1, 3).decimal_str(10)
    '0.3333333333'
    >>> Fraction(-101001, 100000000).decimal_str()
    '-0.00101001'
    >>> Fraction(1235, 1000).decimal_str(2)
    '1.24'
    >>> Fraction(99995, 100000).decimal_str(5)
    '0.99995'
    >>> Fraction(91995, 100000).decimal_str(3)
    '0.92'
    >>> Fraction(99995, 100000).decimal_str(4)
    '1'
    """
    a: int = self.a  # Get the numerator.
    if a == 0:  # If the fraction is 0, we return 0.
        return "0"
    negative: Final[bool] = a < 0  # Get the sign of the fraction.
    a = abs(a)  # Make sure that `a` is now positive.
    b: Final[int] = self.b  # Get the denominator.

    digits: Final[list] = []  # A list for collecting digits.
    while (a != 0) and (len(digits) <= max_frac):  # Create digits.
        digits.append(a // b)  # Add the current digit.
        a = 10 * (a % b)  # Ten times the remainder -> next digit.

    if (a // b) >= 5:  # Do we need to round up?
        digits[-1] += 1  # Round up by incrementing last digit.

    if len(digits) <= 1:  # Do we only have an integer part?
        return str((-1 if negative else 1) * digits[0])

    digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
    if negative:  # Do we need to restore the sign?
        digits.insert(0, "-")  # Insert the sign at the beginning.
    return "".join(map(str, digits))  # Join all digits to a string.
```

`Fraction`: Doctests

- Wir machen das mit einigen normalen Fällen und ein paar extremen Situationen.
- Zuerst prüfen wir, ob Ganzzahlen ordentlich dargestellt werden.
- Es ist klar, dass `Fraction(124, 2).decimal_str()` das Ergebnis `"62"` haben muss.
- Dann prüfen wir, ob ein normaler Bruch genau übersetzt wird: `Fraction(1, 2).decimal_str()` sollte `"0.5"` ergeben.
- Als Bruch der nicht genau als Dezimalzahl geschrieben werden kann wählen wir $\frac{1}{3}$ aus.

```
    >>> Fraction(124, 2).decimal_str()
    '62'
    >>> Fraction(1, 2).decimal_str()
    '0.5'
    >>> Fraction(1, 3).decimal_str(10)
    '0.3333333333'
    >>> Fraction(-101001, 100000000).decimal_str()
    '-0.00101001'
    >>> Fraction(1235, 1000).decimal_str(2)
    '1.24'
    >>> Fraction(99995, 100000).decimal_str(5)
    '0.99995'
    >>> Fraction(91995, 100000).decimal_str(3)
    '0.92'
    >>> Fraction(99995, 100000).decimal_str(4)
    '1'
    """
    a: int = self.a  # Get the numerator.
    if a == 0:  # If the fraction is 0, we return 0.
        return "0"
    negative: Final[bool] = a < 0  # Get the sign of the fraction.
    a = abs(a)  # Make sure that `a` is now positive.
    b: Final[int] = self.b  # Get the denominator.

    digits: Final[list] = []  # A list for collecting digits.
    while (a != 0) and (len(digits) <= max_frac):  # Create digits.
        digits.append(a // b)  # Add the current digit.
        a = 10 * (a % b)  # Ten times the remainder -> next digit.

    if (a // b) >= 5:  # Do we need to round up?
        digits[-1] += 1  # Round up by incrementing last digit.

    if len(digits) <= 1:  # Do we only have an integer part?
        return str((-1 if negative else 1) * digits[0])

    digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
    if negative:  # Do we need to restore the sign?
        digits.insert(0, "-")  # Insert the sign at the beginning.
    return "".join(map(str, digits))  # Join all digits to a string.
```

`Fraction`: Doctests

- Zuerst prüfen wir, ob Ganzzahlen ordentlich dargestellt werden.
- Es ist klar, dass `Fraction(124, 2).decimal_str()` das Ergebnis `"62"` haben muss.
- Dann prüfen wir, ob ein normaler Bruch genau übersetzt wird: `Fraction(1, 2).decimal_str()` sollte `"0.5"` ergeben.
- Als Bruch der nicht genau als Dezimalzahl geschrieben werden kann wählen wir $\frac{1}{3}$ aus.
- $\frac{1}{3}$ auf zehn Ziffern nach dem Komma sollte `"0.3333333333"` ergeben.

```
>>> Fraction(124, 2).decimal_str()
'62'
>>> Fraction(1, 2).decimal_str()
'0.5'
>>> Fraction(1, 3).decimal_str(10)
'0.3333333333'
>>> Fraction(-101001, 100000000).decimal_str()
'-0.00101001'
>>> Fraction(1235, 1000).decimal_str(2)
'1.24'
>>> Fraction(99995, 100000).decimal_str(5)
'0.99995'
>>> Fraction(91995, 100000).decimal_str(3)
'0.92'
>>> Fraction(99995, 100000).decimal_str(4)
'1'
"""
a: int = self.a  # Get the numerator.
if a == 0:  # If the fraction is 0, we return 0.
    return "0"
negative: Final[bool] = a < 0  # Get the sign of the fraction.
a = abs(a)  # Make sure that `a` is now positive.
b: Final[int] = self.b  # Get the denominator.

digits: Final[list] = []  # A list for collecting digits.
while (a != 0) and (len(digits) <= max_frac):  # Create digits.
    digits.append(a // b)  # Add the current digit.
    a = 10 * (a % b)  # Ten times the remainder -> next digit.

if (a // b) >= 5:  # Do we need to round up?
    digits[-1] += 1  # Round up by incrementing last digit.

if len(digits) <= 1:  # Do we only have an integer part?
    return str((-1 if negative else 1) * digits[0])

digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
if negative:  # Do we need to restore the sign?
    digits.insert(0, "-")  # Insert the sign at the beginning.
return "".join(map(str, digits))  # Join all digits to a string.
```

`Fraction`: Doctests

- Es ist klar, dass `Fraction(124, 2).decimal_str()` das Ergebnis `"62"` haben muss.
- Dann prüfen wir, ob ein normaler Bruch genau übersetzt wird: `Fraction(1, 2).decimal_str()` sollte `"0.5"` ergeben.
- Als Bruch der nicht genau als Dezimalzahl geschrieben werden kann wählen wir $\frac{1}{3}$ aus.
- $\frac{1}{3}$ auf zehn Ziffern nach dem Komma sollte `"0.3333333333"` ergeben.
- Als Beispiel für einen negativen Bruch und als Beispiel für einen Bruch mit mehreren führenden Nullen wählen wir $\frac{-101001}{100000000}$.

```
    >>> Fraction(124, 2).decimal_str()
    '62'
    >>> Fraction(1, 2).decimal_str()
    '0.5'
    >>> Fraction(1, 3).decimal_str(10)
    '0.3333333333'
    >>> Fraction(-101001, 100000000).decimal_str()
    '-0.00101001'
    >>> Fraction(1235, 1000).decimal_str(2)
    '1.24'
    >>> Fraction(99995, 100000).decimal_str(5)
    '0.99995'
    >>> Fraction(91995, 100000).decimal_str(3)
    '0.92'
    >>> Fraction(99995, 100000).decimal_str(4)
    '1'
    """
    a: int = self.a  # Get the numerator.
    if a == 0:  # If the fraction is 0, we return 0.
        return "0"
    negative: Final[bool] = a < 0  # Get the sign of the fraction.
    a = abs(a)  # Make sure that `a` is now positive.
    b: Final[int] = self.b  # Get the denominator.

    digits: Final[list] = []  # A list for collecting digits.
    while (a != 0) and (len(digits) <= max_frac):  # Create digits.
        digits.append(a // b)  # Add the current digit.
        a = 10 * (a % b)  # Ten times the remainder -> next digit.

    if (a // b) >= 5:  # Do we need to round up?
        digits[-1] += 1  # Round up by incrementing last digit.

    if len(digits) <= 1:  # Do we only have an integer part?
        return str((-1 if negative else 1) * digits[0])

    digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
    if negative:  # Do we need to restore the sign?
        digits.insert(0, "-")  # Insert the sign at the beginning.
    return "".join(map(str, digits))  # Join all digits to a string.
```

`Fraction`: Doctests

- Dann prüfen wir, ob ein normaler Bruch genau übersetzt wird: `Fraction(1, 2).decimal_str()` sollte `"0.5"` ergeben.
- Als Bruch der nicht genau als Dezimalzahl geschrieben werden kann wählen wir $\frac{1}{3}$ aus.
- $\frac{1}{3}$ auf zehn Ziffern nach dem Komma sollte `"0.3333333333"` ergeben.
- Als Beispiel für einen negativen Bruch und als Beispiel für einen Bruch mit mehreren führenden Nullen wählen wir $\frac{-101001}{100000000}$.
- Das sollte `"-0.00101001"` ergeben.

```
    >>> Fraction(124, 2).decimal_str()
    '62'
    >>> Fraction(1, 2).decimal_str()
    '0.5'
    >>> Fraction(1, 3).decimal_str(10)
    '0.3333333333'
    >>> Fraction(-101001, 100000000).decimal_str()
    '-0.00101001'
    >>> Fraction(1235, 1000).decimal_str(2)
    '1.24'
    >>> Fraction(99995, 100000).decimal_str(5)
    '0.99995'
    >>> Fraction(91995, 100000).decimal_str(3)
    '0.92'
    >>> Fraction(99995, 100000).decimal_str(4)
    '1'
    """
    a: int = self.a  # Get the numerator.
    if a == 0:  # If the fraction is 0, we return 0.
        return "0"
    negative: Final[bool] = a < 0  # Get the sign of the fraction.
    a = abs(a)  # Make sure that `a` is now positive.
    b: Final[int] = self.b  # Get the denominator.

    digits: Final[list] = []  # A list for collecting digits.
    while (a != 0) and (len(digits) <= max_frac):  # Create digits.
        digits.append(a // b)  # Add the current digit.
        a = 10 * (a % b)  # Ten times the remainder -> next digit.

    if (a // b) >= 5:  # Do we need to round up?
        digits[-1] += 1  # Round up by incrementing last digit.

    if len(digits) <= 1:  # Do we only have an integer part?
        return str((-1 if negative else 1) * digits[0])

    digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
    if negative:  # Do we need to restore the sign?
        digits.insert(0, "-")  # Insert the sign at the beginning.
    return "".join(map(str, digits))  # Join all digits to a string.
```

`Fraction`: Doctests

- Als Bruch der nicht genau als Dezimalzahl geschrieben werden kann wählen wir $\frac{1}{3}$ aus.
- $\frac{1}{3}$ auf zehn Ziffern nach dem Komma sollte `"0.3333333333"` ergeben.
- Als Beispiel für einen negativen Bruch und als Beispiel für einen Bruch mit mehreren führenden Nullen wählen wir $\frac{-101001}{100000000}$.
- Das sollte `"-0.00101001"` ergeben.
- Um das Runden der letzten Ziffer zu prüfen, erwarten wir das `Fraction(1235, 1000).decimal_str(2)` dann `"1.24"` ergibt.

```
    >>> Fraction(124, 2).decimal_str()
    '62'
    >>> Fraction(1, 2).decimal_str()
    '0.5'
    >>> Fraction(1, 3).decimal_str(10)
    '0.3333333333'
    >>> Fraction(-101001, 100000000).decimal_str()
    '-0.00101001'
    >>> Fraction(1235, 1000).decimal_str(2)
    '1.24'
    >>> Fraction(99995, 100000).decimal_str(5)
    '0.99995'
    >>> Fraction(91995, 100000).decimal_str(3)
    '0.92'
    >>> Fraction(99995, 100000).decimal_str(4)
    '1'
    """
    a: int = self.a  # Get the numerator.
    if a == 0:  # If the fraction is 0, we return 0.
        return "0"
    negative: Final[bool] = a < 0  # Get the sign of the fraction.
    a = abs(a)  # Make sure that `a` is now positive.
    b: Final[int] = self.b  # Get the denominator.

    digits: Final[list] = []  # A list for collecting digits.
    while (a != 0) and (len(digits) <= max_frac):  # Create digits.
        digits.append(a // b)  # Add the current digit.
        a = 10 * (a % b)  # Ten times the remainder -> next digit.

    if (a // b) >= 5:  # Do we need to round up?
        digits[-1] += 1  # Round up by incrementing last digit.

    if len(digits) <= 1:  # Do we only have an integer part?
        return str((-1 if negative else 1) * digits[0])

    digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
    if negative:  # Do we need to restore the sign?
        digits.insert(0, "-")  # Insert the sign at the beginning.
    return "".join(map(str, digits))  # Join all digits to a string.
```

`Fraction`: Doctests

- $\frac{1}{3}$ auf zehn Ziffern nach dem Komma sollte `"0.3333333333"` ergeben.
- Als Beispiel für einen negativen Bruch und als Beispiel für einen Bruch mit mehreren führenden Nullen wählen wir $\frac{-101001}{100000000}$.
- Das sollte `"-0.00101001"` ergeben.
- Um das Runden der letzten Ziffer zu prüfen, erwarten wir das `Fraction(1235, 1000).decimal_str(2)` dann `"1.24"` ergibt.
- Diese Zahl hätte eigentlich drei Nachkommastellen, wir wollen aber nur zwei.

```
>>> Fraction(124, 2).decimal_str()
'62'
>>> Fraction(1, 2).decimal_str()
'0.5'
>>> Fraction(1, 3).decimal_str(10)
'0.3333333333'
>>> Fraction(-101001, 100000000).decimal_str()
'-0.00101001'
>>> Fraction(1235, 1000).decimal_str(2)
'1.24'
>>> Fraction(99995, 100000).decimal_str(5)
'0.99995'
>>> Fraction(91995, 100000).decimal_str(3)
'0.92'
>>> Fraction(99995, 100000).decimal_str(4)
'1'
"""
a: int = self.a  # Get the numerator.
if a == 0:  # If the fraction is 0, we return 0.
    return "0"
negative: Final[bool] = a < 0  # Get the sign of the fraction.
a = abs(a)  # Make sure that `a` is now positive.
b: Final[int] = self.b  # Get the denominator.

digits: Final[list] = []  # A list for collecting digits.
while (a != 0) and (len(digits) <= max_frac):  # Create digits.
    digits.append(a // b)  # Add the current digit.
    a = 10 * (a % b)  # Ten times the remainder -> next digit.

if (a // b) >= 5:  # Do we need to round up?
    digits[-1] += 1  # Round up by incrementing last digit.

if len(digits) <= 1:  # Do we only have an integer part?
    return str((-1 if negative else 1) * digits[0])

digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
if negative:  # Do we need to restore the sign?
    digits.insert(0, "-")  # Insert the sign at the beginning.
return "".join(map(str, digits))  # Join all digits to a string.
```

`Fraction`: Doctests

- Als Beispiel für einen negativen Bruch und als Beispiel für einen Bruch mit mehreren führenden Nullen wählen wir $\frac{-101001}{100000000}$.
- Das sollte `"-0.00101001"` ergeben.
- Um das Runden der letzten Ziffer zu prüfen, erwarten wir das `Fraction(1235, 1000).decimal_str(2)` dann `"1.24"` ergibt.
- Diese Zahl hätte eigentlich drei Nachkommastellen, wir wollen aber nur zwei.
- Da die dritte Ziffer eine 5 wäre, muss also gerundet werden.

```
>>> Fraction(124, 2).decimal_str()
'62'
>>> Fraction(1, 2).decimal_str()
'0.5'
>>> Fraction(1, 3).decimal_str(10)
'0.3333333333'
>>> Fraction(-101001, 100000000).decimal_str()
'-0.00101001'
>>> Fraction(1235, 1000).decimal_str(2)
'1.24'
>>> Fraction(99995, 100000).decimal_str(5)
'0.99995'
>>> Fraction(91995, 100000).decimal_str(3)
'0.92'
>>> Fraction(99995, 100000).decimal_str(4)
'1'
"""
a: int = self.a  # Get the numerator.
if a == 0:  # If the fraction is 0, we return 0.
    return "0"
negative: Final[bool] = a < 0  # Get the sign of the fraction.
a = abs(a)  # Make sure that `a` is now positive.
b: Final[int] = self.b  # Get the denominator.

digits: Final[list] = []  # A list for collecting digits.
while (a != 0) and (len(digits) <= max_frac):  # Create digits.
    digits.append(a // b)  # Add the current digit.
    a = 10 * (a % b)  # Ten times the remainder -> next digit.

if (a // b) >= 5:  # Do we need to round up?
    digits[-1] += 1  # Round up by incrementing last digit.

if len(digits) <= 1:  # Do we only have an integer part?
    return str((-1 if negative else 1) * digits[0])

digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
if negative:  # Do we need to restore the sign?
    digits.insert(0, "-")  # Insert the sign at the beginning.
return "".join(map(str, digits))  # Join all digits to a string.
```

`Fraction`: Doctests

- Das sollte `"-0.00101001"` ergeben.
- Um das Runden der letzten Ziffer zu prüfen, erwarten wir das `Fraction(1235, 1000).decimal_str(2)` dann `"1.24"` ergibt.
- Diese Zahl hätte eigentlich drei Nachkommastellen, wir wollen aber nur zwei.
- Da die dritte Ziffer eine 5 wäre, muss also gerundet werden.
- Anstelle von `"1.235"` oder `"1.23"` müsste korrekterweise `"1.24"` herauskommen.

```
>>> Fraction(124, 2).decimal_str()
'62'
>>> Fraction(1, 2).decimal_str()
'0.5'
>>> Fraction(1, 3).decimal_str(10)
'0.3333333333'
>>> Fraction(-101001, 100000000).decimal_str()
'-0.00101001'
>>> Fraction(1235, 1000).decimal_str(2)
'1.24'
>>> Fraction(99995, 100000).decimal_str(5)
'0.99995'
>>> Fraction(91995, 100000).decimal_str(3)
'0.92'
>>> Fraction(99995, 100000).decimal_str(4)
'1'
"""
a: int = self.a  # Get the numerator.
if a == 0:  # If the fraction is 0, we return 0.
    return "0"
negative: Final[bool] = a < 0  # Get the sign of the fraction.
a = abs(a)  # Make sure that `a` is now positive.
b: Final[int] = self.b  # Get the denominator.

digits: Final[list] = []  # A list for collecting digits.
while (a != 0) and (len(digits) <= max_frac):  # Create digits.
    digits.append(a // b)  # Add the current digit.
    a = 10 * (a % b)  # Ten times the remainder -> next digit.

if (a // b) >= 5:  # Do we need to round up?
    digits[-1] += 1  # Round up by incrementing last digit.

if len(digits) <= 1:  # Do we only have an integer part?
    return str((-1 if negative else 1) * digits[0])

digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
if negative:  # Do we need to restore the sign?
    digits.insert(0, "-")  # Insert the sign at the beginning.
return "".join(map(str, digits))  # Join all digits to a string.
```

`Fraction`: Doctests

- Diese Zahl hätte eigentlich drei Nachkommastellen, wir wollen aber nur zwei.
- Da die dritte Ziffer eine 5 wäre, muss also gerundet werden.
- Anstelle von `"1.235"` oder `"1.23"` müsste korrekterweise `"1.24"` herauskommen.
- `Fraction(99995, 100000).decimal_str(5)`, also 0.99995 auf fünf Nachkommastellen gerundet, sollte `"0.99995"` sein.

```
        >>> Fraction(124, 2).decimal_str()
        '62'
        >>> Fraction(1, 2).decimal_str()
        '0.5'
        >>> Fraction(1, 3).decimal_str(10)
        '0.3333333333'
        >>> Fraction(-101001, 100000000).decimal_str()
        '-0.00101001'
        >>> Fraction(1235, 1000).decimal_str(2)
        '1.24'
        >>> Fraction(99995, 100000).decimal_str(5)
        '0.99995'
        >>> Fraction(91995, 100000).decimal_str(3)
        '0.92'
        >>> Fraction(99995, 100000).decimal_str(4)
        '1'
        """
        a: int = self.a  # Get the numerator.
        if a == 0:  # If the fraction is 0, we return 0.
            return "0"
        negative: Final[bool] = a < 0  # Get the sign of the fraction.
        a = abs(a)  # Make sure that `a` is now positive.
        b: Final[int] = self.b  # Get the denominator.

        digits: Final[list] = []  # A list for collecting digits.
        while (a != 0) and (len(digits) <= max_frac):  # Create digits.
            digits.append(a // b)  # Add the current digit.
            a = 10 * (a % b)  # Ten times the remainder -> next digit.

        if (a // b) >= 5:  # Do we need to round up?
            digits[-1] += 1  # Round up by incrementing last digit.

        if len(digits) <= 1:  # Do we only have an integer part?
            return str((-1 if negative else 1) * digits[0])

        digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
        if negative:  # Do we need to restore the sign?
            digits.insert(0, "-")  # Insert the sign at the beginning.
        return "".join(map(str, digits))  # Join all digits to a string.
```

`Fraction`: Doctests

- Diese Zahl hätte eigentlich drei Nachkommastellen, wir wollen aber nur zwei.
- Da die dritte Ziffer eine 5 wäre, muss also gerundet werden.
- Anstelle von `"1.235"` oder `"1.23"` müsste korrekterweise `"1.24"` herauskommen.
- `Fraction(99995, 100000).decimal_str(5)`, also 0.99995 auf fünf Nachkommastellen gerundet, sollte `"0.99995"` sein.
- `Fraction(91995, 100000).decimal_str(3)` bedeutet 0.91995 auf drei Nachkommastellen zu runden.

```
    >>> Fraction(124, 2).decimal_str()
    '62'
    >>> Fraction(1, 2).decimal_str()
    '0.5'
    >>> Fraction(1, 3).decimal_str(10)
    '0.3333333333'
    >>> Fraction(-101001, 100000000).decimal_str()
    '-0.00101001'
    >>> Fraction(1235, 1000).decimal_str(2)
    '1.24'
    >>> Fraction(99995, 100000).decimal_str(5)
    '0.99995'
    >>> Fraction(91995, 100000).decimal_str(3)
    '0.92'
    >>> Fraction(99995, 100000).decimal_str(4)
    '1'
    """
    a: int = self.a  # Get the numerator.
    if a == 0:  # If the fraction is 0, we return 0.
        return "0"
    negative: Final[bool] = a < 0  # Get the sign of the fraction.
    a = abs(a)  # Make sure that `a` is now positive.
    b: Final[int] = self.b  # Get the denominator.

    digits: Final[list] = []  # A list for collecting digits.
    while (a != 0) and (len(digits) <= max_frac):  # Create digits.
        digits.append(a // b)  # Add the current digit.
        a = 10 * (a % b)  # Ten times the remainder -> next digit.

    if (a // b) >= 5:  # Do we need to round up?
        digits[-1] += 1  # Round up by incrementing last digit.

    if len(digits) <= 1:  # Do we only have an integer part?
        return str((-1 if negative else 1) * digits[0])

    digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
    if negative:  # Do we need to restore the sign?
        digits.insert(0, "-")  # Insert the sign at the beginning.
    return "".join(map(str, digits))  # Join all digits to a string.
```

`Fraction`: Doctests

- Da die dritte Ziffer eine 5 wäre, muss also gerundet werden.
- Anstelle von `"1.235"` oder `"1.23"` müsste korrekterweise `"1.24"` herauskommen.
- `Fraction(99995, 100000).decimal_str(5)`, also 0.99995 auf fünf Nachkommastellen gerundet, sollte `"0.99995"` sein.
- `Fraction(91995, 100000).decimal_str(3)` bedeutet 0.91995 auf drei Nachkommastellen zu runden.
- Die letzte Ziffer, eine 5, wird abgeschnitten.

```
>>> Fraction(124, 2).decimal_str()
'62'
>>> Fraction(1, 2).decimal_str()
'0.5'
>>> Fraction(1, 3).decimal_str(10)
'0.3333333333'
>>> Fraction(-101001, 100000000).decimal_str()
'-0.00101001'
>>> Fraction(1235, 1000).decimal_str(2)
'1.24'
>>> Fraction(99995, 100000).decimal_str(5)
'0.99995'
>>> Fraction(91995, 100000).decimal_str(3)
'0.92'
>>> Fraction(99995, 100000).decimal_str(4)
'1'
"""
a: int = self.a  # Get the numerator.
if a == 0:  # If the fraction is 0, we return 0.
    return "0"
negative: Final[bool] = a < 0  # Get the sign of the fraction.
a = abs(a)  # Make sure that `a` is now positive.
b: Final[int] = self.b  # Get the denominator.

digits: Final[list] = []  # A list for collecting digits.
while (a != 0) and (len(digits) <= max_frac):  # Create digits.
    digits.append(a // b)  # Add the current digit.
    a = 10 * (a % b)  # Ten times the remainder -> next digit.

if (a // b) >= 5:  # Do we need to round up?
    digits[-1] += 1  # Round up by incrementing last digit.

if len(digits) <= 1:  # Do we only have an integer part?
    return str((-1 if negative else 1) * digits[0])

digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
if negative:  # Do we need to restore the sign?
    digits.insert(0, "-")  # Insert the sign at the beginning.
return "".join(map(str, digits))  # Join all digits to a string.
```

`Fraction`: Doctests

- `Fraction(99995, 100000).decimal_str(5)`, also 0.99995 auf fünf Nachkommastellen gerundet, sollte `"0.99995"` sein.
- `Fraction(91995, 100000).decimal_str(3)` bedeutet 0.91995 auf drei Nachkommastellen zu runden.
- Die letzte Ziffer, eine 5, wird abgeschnitten.
- Das bedeutet, das wir aufrunden, was wiederum die vorletzte Ziffer (9) auch aufrunden wird, wodurch die nächste 9 auch aufgerundet wird.

```
>>> Fraction(124, 2).decimal_str()
'62'
>>> Fraction(1, 2).decimal_str()
'0.5'
>>> Fraction(1, 3).decimal_str(10)
'0.3333333333'
>>> Fraction(-101001, 100000000).decimal_str()
'-0.00101001'
>>> Fraction(1235, 1000).decimal_str(2)
'1.24'
>>> Fraction(99995, 100000).decimal_str(5)
'0.99995'
>>> Fraction(91995, 100000).decimal_str(3)
'0.92'
>>> Fraction(99995, 100000).decimal_str(4)
'1'
"""
a: int = self.a  # Get the numerator.
if a == 0:  # If the fraction is 0, we return 0.
    return "0"
negative: Final[bool] = a < 0  # Get the sign of the fraction.
a = abs(a)  # Make sure that `a` is now positive.
b: Final[int] = self.b  # Get the denominator.

digits: Final[list] = []  # A list for collecting digits.
while (a != 0) and (len(digits) <= max_frac):  # Create digits.
    digits.append(a // b)  # Add the current digit.
    a = 10 * (a % b)  # Ten times the remainder -> next digit.

if (a // b) >= 5:  # Do we need to round up?
    digits[-1] += 1  # Round up by incrementing last digit.

if len(digits) <= 1:  # Do we only have an integer part?
    return str((-1 if negative else 1) * digits[0])

digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
if negative:  # Do we need to restore the sign?
    digits.insert(0, "-")  # Insert the sign at the beginning.
return "".join(map(str, digits))  # Join all digits to a string.
```

`Fraction`: Doctests

- `Fraction(99995, 100000).decimal_str(5)`, also 0.99995 auf fünf Nachkommastellen gerundet, sollte `"0.99995"` sein.
- `Fraction(91995, 100000).decimal_str(3)` bedeutet 0.91995 auf drei Nachkommastellen zu runden.
- Die letzte Ziffer, eine 5, wird abgeschnitten.
- Das bedeutet, das wir aufrunden, was wiederum die vorletzte Ziffer (9) auch aufrunden wird, wodurch die nächste 9 auch aufgerundet wird.
- Die 1 wird dann zu einer 2 und wir sollten `"0.92"` bekommen.

```
>>> Fraction(124, 2).decimal_str()
'62'
>>> Fraction(1, 2).decimal_str()
'0.5'
>>> Fraction(1, 3).decimal_str(10)
'0.3333333333'
>>> Fraction(-101001, 100000000).decimal_str()
'-0.00101001'
>>> Fraction(1235, 1000).decimal_str(2)
'1.24'
>>> Fraction(99995, 100000).decimal_str(5)
'0.99995'
>>> Fraction(91995, 100000).decimal_str(3)
'0.92'
>>> Fraction(99995, 100000).decimal_str(4)
'1'
"""
a: int = self.a  # Get the numerator.
if a == 0:  # If the fraction is 0, we return 0.
    return "0"
negative: Final[bool] = a < 0  # Get the sign of the fraction.
a = abs(a)  # Make sure that `a` is now positive.
b: Final[int] = self.b  # Get the denominator.

digits: Final[list] = []  # A list for collecting digits.
while (a != 0) and (len(digits) <= max_frac):  # Create digits.
    digits.append(a // b)  # Add the current digit.
    a = 10 * (a % b)  # Ten times the remainder -> next digit.

if (a // b) >= 5:  # Do we need to round up?
    digits[-1] += 1  # Round up by incrementing last digit.

if len(digits) <= 1:  # Do we only have an integer part?
    return str((-1 if negative else 1) * digits[0])

digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
if negative:  # Do we need to restore the sign?
    digits.insert(0, "-")  # Insert the sign at the beginning.
return "".join(map(str, digits))  # Join all digits to a string.
```

`Fraction`: Doctests

- `Fraction(99995, 100000).decimal_str(5)`, also 0.99995 auf fünf Nachkommastellen gerundet, sollte `"0.99995"` sein.
- `Fraction(91995, 100000).decimal_str(3)` bedeutet 0.91995 auf drei Nachkommastellen zu runden.
- Die letzte Ziffer, eine 5, wird abgeschnitten.
- Das bedeutet, das wir aufrunden, was wiederum die vorletzte Ziffer (9) auch aufrunden wird, wodurch die nächste 9 auch aufgerundet wird.
- Die 1 wird dann zu einer 2 und wir sollten `"0.92"` bekommen.

```
>>> Fraction(124, 2).decimal_str()
'62'
>>> Fraction(1, 2).decimal_str()
'0.5'
>>> Fraction(1, 3).decimal_str(10)
'0.3333333333'
>>> Fraction(-101001, 100000000).decimal_str()
'-0.00101001'
>>> Fraction(1235, 1000).decimal_str(2)
'1.24'
>>> Fraction(99995, 100000).decimal_str(5)
'0.99995'
>>> Fraction(91995, 100000).decimal_str(3)
'0.92'
>>> Fraction(99995, 100000).decimal_str(4)
'1'
"""
a: int = self.a  # Get the numerator.
if a == 0:  # If the fraction is 0, we return 0.
    return "0"
negative: Final[bool] = a < 0  # Get the sign of the fraction.
a = abs(a)  # Make sure that `a` is now positive.
b: Final[int] = self.b  # Get the denominator.

digits: Final[list] = []  # A list for collecting digits.
while (a != 0) and (len(digits) <= max_frac):  # Create digits.
    digits.append(a // b)  # Add the current digit.
    a = 10 * (a % b)  # Ten times the remainder -> next digit.

if (a // b) >= 5:  # Do we need to round up?
    digits[-1] += 1  # Round up by incrementing last digit.

if len(digits) <= 1:  # Do we only have an integer part?
    return str((-1 if negative else 1) * digits[0])

digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
if negative:  # Do we need to restore the sign?
    digits.insert(0, "-")  # Insert the sign at the beginning.
return "".join(map(str, digits))  # Join all digits to a string.
```

`Fraction`: Doctests

- `Fraction(91995, 100000).decimal_str(3)` bedeutet 0.91995 auf drei Nachkommastellen zu runden.
- Die letzte Ziffer, eine 5, wird abgeschnitten.
- Das bedeutet, das wir aufrunden, was wiederum die vorletzte Ziffer (9) auch aufrunden wird, wodurch die nächste 9 auch aufgerundet wird.
- Die 1 wird dann zu einer 2 und wir sollten `"0.92"` bekommen.
- Etwas ähnliches muss passieren, wenn wir `Fraction(99995, 100000).decimal_str(4)` berechnen.

```
>>> Fraction(124, 2).decimal_str()
'62'
>>> Fraction(1, 2).decimal_str()
'0.5'
>>> Fraction(1, 3).decimal_str(10)
'0.3333333333'
>>> Fraction(-101001, 100000000).decimal_str()
'-0.00101001'
>>> Fraction(1235, 1000).decimal_str(2)
'1.24'
>>> Fraction(99995, 100000).decimal_str(5)
'0.99995'
>>> Fraction(91995, 100000).decimal_str(3)
'0.92'
>>> Fraction(99995, 100000).decimal_str(4)
'1'
"""
a: int = self.a  # Get the numerator.
if a == 0:  # If the fraction is 0, we return 0.
    return "0"
negative: Final[bool] = a < 0  # Get the sign of the fraction.
a = abs(a)  # Make sure that `a` is now positive.
b: Final[int] = self.b  # Get the denominator.

digits: Final[list] = []  # A list for collecting digits.
while (a != 0) and (len(digits) <= max_frac):  # Create digits.
    digits.append(a // b)  # Add the current digit.
    a = 10 * (a % b)  # Ten times the remainder -> next digit.

if (a // b) >= 5:  # Do we need to round up?
    digits[-1] += 1  # Round up by incrementing last digit.

if len(digits) <= 1:  # Do we only have an integer part?
    return str((-1 if negative else 1) * digits[0])

digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
if negative:  # Do we need to restore the sign?
    digits.insert(0, "-")  # Insert the sign at the beginning.
return "".join(map(str, digits))  # Join all digits to a string.
```

`Fraction`: Doctests

- Die letzte Ziffer, eine 5, wird abgeschnitten.
- Das bedeutet, das wir aufrunden, was wiederum die vorletzte Ziffer (9) auch aufrunden wird, wodurch die nächste 9 auch aufgerundet wird.
- Die 1 wird dann zu einer 2 und wir sollten `"0.92"` bekommen.
- Etwas ähnliches muss passieren, wenn wir `Fraction(99995, 100000).decimal_str(4)` berechnen.
- 0.9995 auf vier Ziffern gerundet wird die 5 aufrunden, wodurch alle 9en auch aufgerundet werden, so dass wir letztendlich `"1"` bekommen.

```
>>> Fraction(124, 2).decimal_str()
'62'
>>> Fraction(1, 2).decimal_str()
'0.5'
>>> Fraction(1, 3).decimal_str(10)
'0.3333333333'
>>> Fraction(-101001, 100000000).decimal_str()
'-0.00101001'
>>> Fraction(1235, 1000).decimal_str(2)
'1.24'
>>> Fraction(99995, 100000).decimal_str(5)
'0.99995'
>>> Fraction(91995, 100000).decimal_str(3)
'0.92'
>>> Fraction(99995, 100000).decimal_str(4)
'1'
"""
a: int = self.a  # Get the numerator.
if a == 0:  # If the fraction is 0, we return 0.
    return "0"
negative: Final[bool] = a < 0  # Get the sign of the fraction.
a = abs(a)  # Make sure that `a` is now positive.
b: Final[int] = self.b  # Get the denominator.

digits: Final[list] = []  # A list for collecting digits.
while (a != 0) and (len(digits) <= max_frac):  # Create digits.
    digits.append(a // b)  # Add the current digit.
    a = 10 * (a % b)  # Ten times the remainder -> next digit.

if (a // b) >= 5:  # Do we need to round up?
    digits[-1] += 1  # Round up by incrementing last digit.

if len(digits) <= 1:  # Do we only have an integer part?
    return str((-1 if negative else 1) * digits[0])

digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
if negative:  # Do we need to restore the sign?
    digits.insert(0, "-")  # Insert the sign at the beginning.
return "".join(map(str, digits))  # Join all digits to a string.
```

`Fraction`: Doctests

- Das bedeutet, das wir aufrunden, was wiederum die vorletzte Ziffer (9) auch aufrunden wird, wodurch die nächste 9 auch aufgerundet wird.
- Die 1 wird dann zu einer 2 und wir sollten `"0.92"` bekommen.
- Etwas ähnliches muss passieren, wenn wir `Fraction(99995, 100000).decimal_str(4)` berechnen.
- 0.9995 auf vier Ziffern gerundet wird die 5 aufrunden, wodurch alle 9en auch aufgerundet werden, so dass wir letztendlich `"1"` bekommen.
- Führen wir also diese Doctests mit pytest aus.

```
$ pytest --timeout=10 --no-header --tb=short --doctest-modules
    ↪ fraction_decimal_str_err.py
========================= test session starts
    ↪ ==========================
collected 1 item
fraction_decimal_str_err.py F                                          [100%]
================================ FAILURES
    ↪ ================================
-------- [doctest] fraction_decimal_str_err.Fraction.decimal_str
    ↪ --------
037 '0.5'
038 >>> Fraction(1, 3).decimal_str(10)
039 '0.3333333333'
040 >>> Fraction(-101001, 100000000).decimal_str()
041 '-0.00101001'
042 >>> Fraction(1235, 1000).decimal_str(2)
043 '1.24'
044 >>> Fraction(99995, 100000).decimal_str(5)
045 '0.99995'
046 >>> Fraction(91995, 100000).decimal_str(3)
Expected:
    '0.92'
Got:
    '0.9110'
/home/runner/work/programmingWithPythonSlidesDE2/
    ↪ programmingWithPythonSlidesDE2/slides/47_debugger/__git__/realms/
    ↪ git/gh_thomasWeise_programmingWithPythonCode/dunder/
    ↪ fraction_decimal_str_err.py:46: DocTestFailure
======================== short test summary info
    ↪ ========================
FAILED fraction_decimal_str_err.py::fraction_decimal_str_err.Fraction.
    ↪ decimal_str
========================== 1 failed in 0.01s
    ↪ ===========================
# pytest 9.2.0 with pytest-timeout 2.4.0 failed with exit code 1.
```

`Fraction`: Doctests

- Etwas ähnliches muss passieren, wenn wir `Fraction(99995, 100000).decimal_str(4)` berechnen.
- 0.9995 auf vier Ziffern gerundet wird die 5 aufrunden, wodurch alle 9en auch aufgerundet werden, so dass wir letztendlich `"1"` bekommen.
- Führen wir also diese Doctests mit pytest aus.
- Sie schlagen fehl!
- Die Ausgabe zeigt us, dass `Fraction(91995, 100000).decimal_str(3)` nicht wie erwartet `"0.92"` liefert.

```
$ pytest --timeout=10 --no-header --tb=short --doctest-modules
   ↪ fraction_decimal_str_err.py
========================== test session starts
   ↪ ==========================
collected 1 item
fraction_decimal_str_err.py F                                            [100%]
================================== FAILURES
   ↪ ==================================
________ [doctest] fraction_decimal_str_err.Fraction.decimal_str
   ↪ ________
037 '0.5'
038 >>> Fraction(1, 3).decimal_str(10)
039 '0.3333333333'
040 >>> Fraction(-101001, 100000000).decimal_str()
041 '-0.00101001'
042 >>> Fraction(1235, 1000).decimal_str(2)
043 '1.24'
044 >>> Fraction(99995, 100000).decimal_str(5)
045 '0.99995'
046 >>> Fraction(91995, 100000).decimal_str(3)
Expected:
    '0.92'
Got:
    '0.9110'
/home/runner/work/programmingWithPythonSlidesDE2/
   ↪ programmingWithPythonSlidesDE2/slides/47_debugger/__git__/realms/
   ↪ git/gh_thomasWeise_programmingWithPythonCode/dunder/
   ↪ fraction_decimal_str_err.py:46: DocTestFailure
======================== short test summary info
   ↪ ==========================
FAILED fraction_decimal_str_err.py::fraction_decimal_str_err.Fraction.
   ↪ decimal_str
========================== 1 failed in 0.01s
   ↪ ==========================
# pytest 9.2.0 with pytest-timeout 2.4.0 failed with exit code 1.
```

`Fraction`: Doctests

- 0.9995 auf vier Ziffern gerundet wird die 5 aufrunden, wodurch alle 9en auch aufgerundet werden, so dass wir letztendlich `"1"` bekommen.
- Führen wir also diese Doctests mit pytest aus.
- Sie schlagen fehl!
- Die Ausgabe zeigt us, dass `Fraction(91995, 100000).decimal_str(3)` nicht wie erwartet `"0.92"` liefert.
- Stattdessen bekommen wir `"0.9110"`.

```
$ pytest --timeout=10 --no-header --tb=short --doctest-modules
    ↪ fraction_decimal_str_err.py
========================== test session starts
    ↪ ==========================
collected 1 item
fraction_decimal_str_err.py F                                            [100%]
================================ FAILURES
    ↪ ================================
_____ [doctest] fraction_decimal_str_err.Fraction.decimal_str
    ↪ _____
037 '0.5'
038 >>> Fraction(1, 3).decimal_str(10)
039 '0.3333333333'
040 >>> Fraction(-101001, 100000000).decimal_str()
041 '-0.00101001'
042 >>> Fraction(1235, 1000).decimal_str(2)
043 '1.24'
044 >>> Fraction(99995, 100000).decimal_str(5)
045 '0.99995'
046 >>> Fraction(91995, 100000).decimal_str(3)
Expected:
    '0.92'
Got:
    '0.9110'
/home/runner/work/programmingWithPythonSlidesDE2/
    ↪ programmingWithPythonSlidesDE2/slides/47_debugger/__git__/realms/
    ↪ git/gh_thomasWeise_programmingWithPythonCode/dunder/
    ↪ fraction_decimal_str_err.py:46: DocTestFailure
======================== short test summary info
    ↪ ========================
FAILED fraction_decimal_str_err.py::fraction_decimal_str_err.Fraction.
    ↪ decimal_str
========================== 1 failed in 0.01s
    ↪ ===========================
# pytest 9.2.0 with pytest-timeout 2.4.0 failed with exit code 1.
```

`Fraction`: **Doctests**

- Führen wir also diese Doctests mit pytest aus.
- Sie schlagen fehl!
- Die Ausgabe zeigt us, dass `Fraction(91995, 100000).decimal_str(3)` nicht wie erwartet `"0.92"` liefert.
- Stattdessen bekommen wir `"0.9110"`.
- Wo kommt die 0 am Ende her?

```
$ pytest --timeout=10 --no-header --tb=short --doctest-modules
   ↪ fraction_decimal_str_err.py
========================== test session starts
   ↪ ==========================
collected 1 item
fraction_decimal_str_err.py F                                            [100%]
================================ FAILURES
   ↪ ================================
________ [doctest] fraction_decimal_str_err.Fraction.decimal_str
   ↪ ________
037 '0.5'
038 >>> Fraction(1, 3).decimal_str(10)
039 '0.3333333333'
040 >>> Fraction(-101001, 100000000).decimal_str()
041 '-0.00101001'
042 >>> Fraction(1235, 1000).decimal_str(2)
043 '1.24'
044 >>> Fraction(99995, 100000).decimal_str(5)
045 '0.99995'
046 >>> Fraction(91995, 100000).decimal_str(3)
Expected:
    '0.92'
Got:
    '0.9110'
/home/runner/work/programmingWithPythonSlidesDE2/
   ↪ programmingWithPythonSlidesDE2/slides/47_debugger/__git__/realms/
   ↪ git/gh_thomasWeise_programmingWithPythonCode/dunder/
   ↪ fraction_decimal_str_err.py:46: DocTestFailure
======================== short test summary info
   ↪ ========================
FAILED fraction_decimal_str_err.py::fraction_decimal_str_err.Fraction.
   ↪ decimal_str
========================== 1 failed in 0.01s
   ↪ ===========================
# pytest 9.2.0 with pytest-timeout 2.4.0 failed with exit code 1.
```

`Fraction`: **Doctests**

- Führen wir also diese Doctests mit pytest aus.
- Sie schlagen fehl!
- Die Ausgabe zeigt us, dass `Fraction(91995, 100000).decimal_str(3)` nicht wie erwartet `"0.92"` liefert.
- Stattdessen bekommen wir `"0.9110"`.
- Wo kommt die 0 am Ende her?
- Und warum sind da zwei 1en?

```
$ pytest --timeout=10 --no-header --tb=short --doctest-modules
    ↪ fraction_decimal_str_err.py
========================= test session starts
    ↪ =========================
collected 1 item
fraction_decimal_str_err.py F                                            [100%]
================================ FAILURES
    ↪ ================================
________ [doctest] fraction_decimal_str_err.Fraction.decimal_str
    ↪ ________
037 '0.5'
038 >>> Fraction(1, 3).decimal_str(10)
039 '0.3333333333'
040 >>> Fraction(-101001, 100000000).decimal_str()
041 '-0.00101001'
042 >>> Fraction(1235, 1000).decimal_str(2)
043 '1.24'
044 >>> Fraction(99995, 100000).decimal_str(5)
045 '0.99995'
046 >>> Fraction(91995, 100000).decimal_str(3)
Expected:
    '0.92'
Got:
    '0.9110'
/home/runner/work/programmingWithPythonSlidesDE2/
    ↪ programmingWithPythonSlidesDE2/slides/47_debugger/__git__/realms/
    ↪ git/gh_thomasWeise_programmingWithPythonCode/dunder/
    ↪ fraction_decimal_str_err.py:46: DocTestFailure
======================== short test summary info
    ↪ ========================
FAILED fraction_decimal_str_err.py::fraction_decimal_str_err.Fraction.
    ↪ decimal_str
========================== 1 failed in 0.01s
    ↪ ===========================
# pytest 9.2.0 with pytest-timeout 2.4.0 failed with exit code 1.
```

`Fraction`: Doctests

- Führen wir also diese Doctests mit pytest aus.
- Sie schlagen fehl!
- Die Ausgabe zeigt us, dass `Fraction(91995, 100000).decimal_str(3)` nicht wie erwartet `"0.92"` liefert.
- Stattdessen bekommen wir `"0.9110"`.
- Wo kommt die 0 am Ende her?
- Und warum sind da zwei 1en?
- Selbst wenn wir falsch gerundet hätten, dann hätte doch vielleicht 0.919 herauskommen können ... aber doch nicht 0.911??

```
$ pytest --timeout=10 --no-header --tb=short --doctest-modules
    ↪ fraction_decimal_str_err.py
========================== test session starts
    ↪ ==========================
collected 1 item
fraction_decimal_str_err.py F                                            [100%]
================================ FAILURES
    ↪ ================================
________ [doctest] fraction_decimal_str_err.Fraction.decimal_str
    ↪ ________
037 '0.5'
038 >>> Fraction(1, 3).decimal_str(10)
039 '0.3333333333'
040 >>> Fraction(-101001, 100000000).decimal_str()
041 '-0.00101001'
042 >>> Fraction(1235, 1000).decimal_str(2)
043 '1.24'
044 >>> Fraction(99995, 100000).decimal_str(5)
045 '0.99995'
046 >>> Fraction(91995, 100000).decimal_str(3)
Expected:
    '0.92'
Got:
    '0.9110'
/home/runner/work/programmingWithPythonSlidesDE2/
    ↪ programmingWithPythonSlidesDE2/slides/47_debugger/__git__/realms/
    ↪ git/gh_thomasWeise_programmingWithPythonCode/dunder/
    ↪ fraction_decimal_str_err.py:46: DocTestFailure
======================== short test summary info
    ↪ ========================
FAILED fraction_decimal_str_err.py::fraction_decimal_str_err.Fraction.
    ↪ decimal_str
========================== 1 failed in 0.01s
    ↪ ==========================
# pytest 9.2.0 with pytest-timeout 2.4.0 failed with exit code 1.
```

Debugging

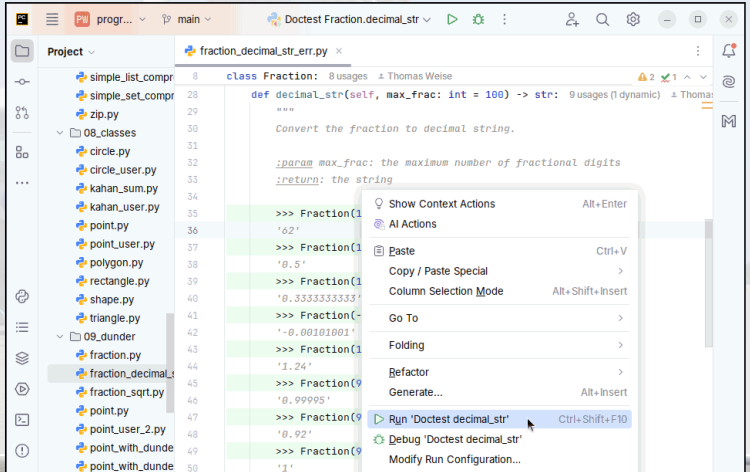


Doctests in PyCharm

- Wir wollen diesen komischen Fehler untersuchen.

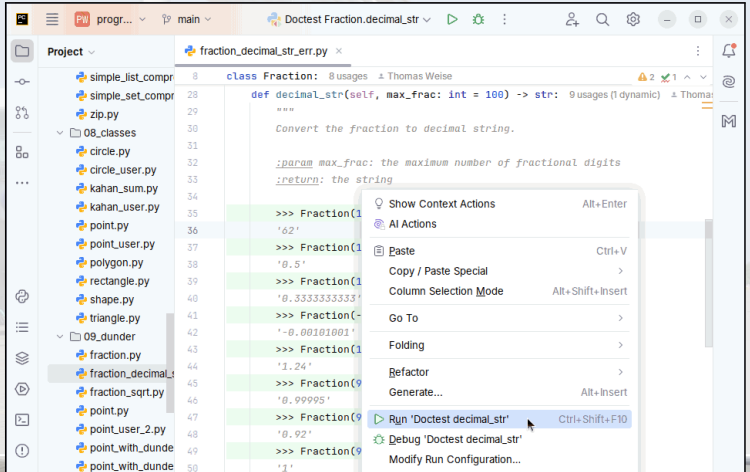
Doctests in PyCharm

- Wir wollen diesen komischen Fehler untersuchen.
- Dafür wollen wir erstmal die Doctests nochmal in PyCharm ausführen.



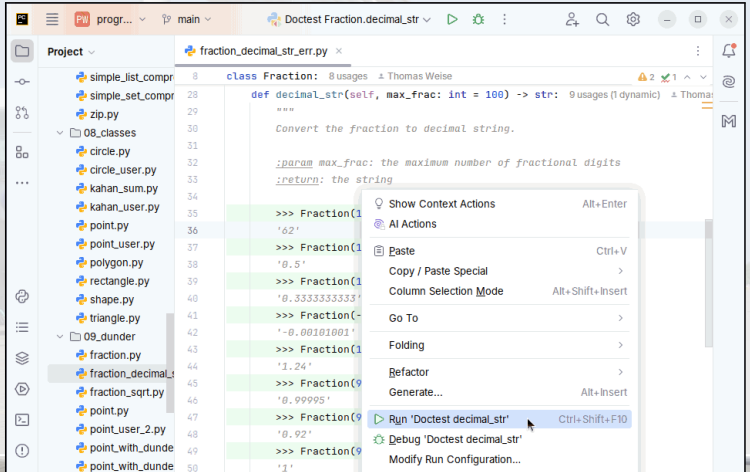
Doctests in PyCharm

- Dafür wollen wir erstmal die Doctests nochmal in PyCharm ausführen.
- Wir öffnen unsere Datei `fraction_decimal_str_err.py` und skrollen zu unserer Methode `decimal_str`.



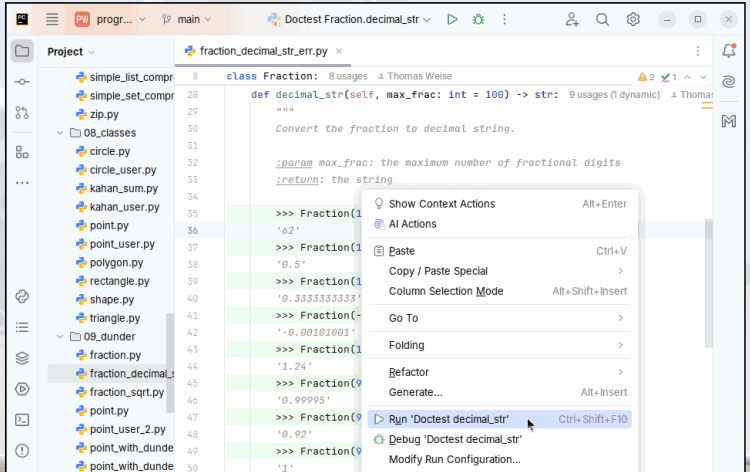
Doctests in PyCharm

- Wir öffnen unsere Datei `fraction_decimal_str_err.py` und skrollen zu unserer Methode `decimal_str`.
- Wir klicken mit der rechten Maustaste und ein Kontextmenü öffnet sich.



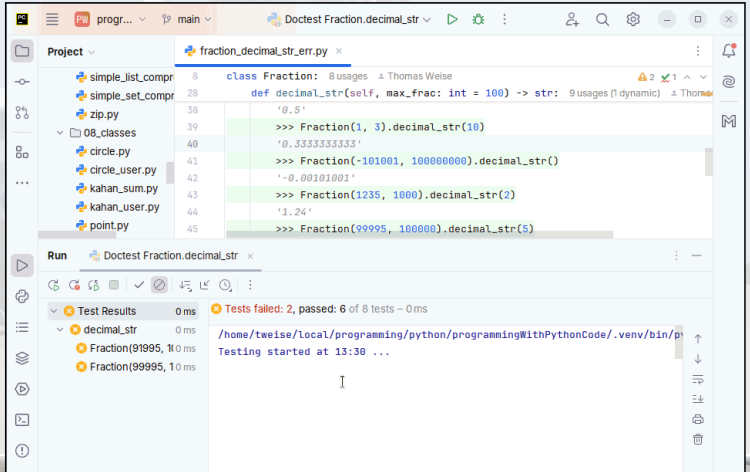
Doctests in PyCharm

- Wir klicken mit der rechten Maustaste und ein Kontextmenü öffnet sich.
- Hier klicken wir mit der linken Maustaste auf `Run 'Doctest decimal_str'`.



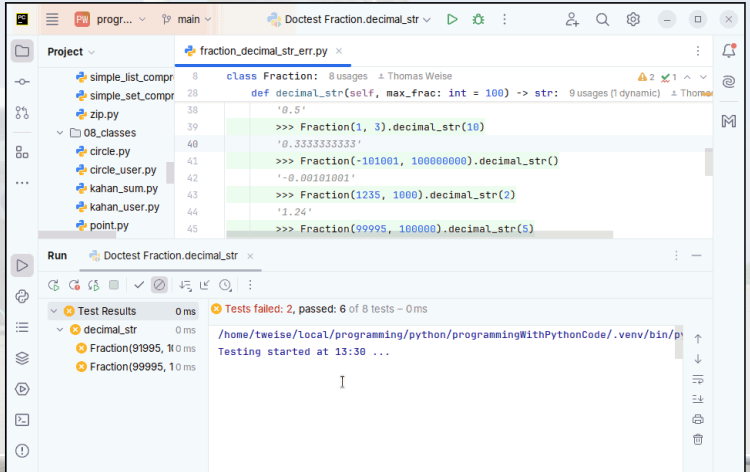
Doctests in PyCharm

- Dadurch werden *alle* Doctests ausgeführt.



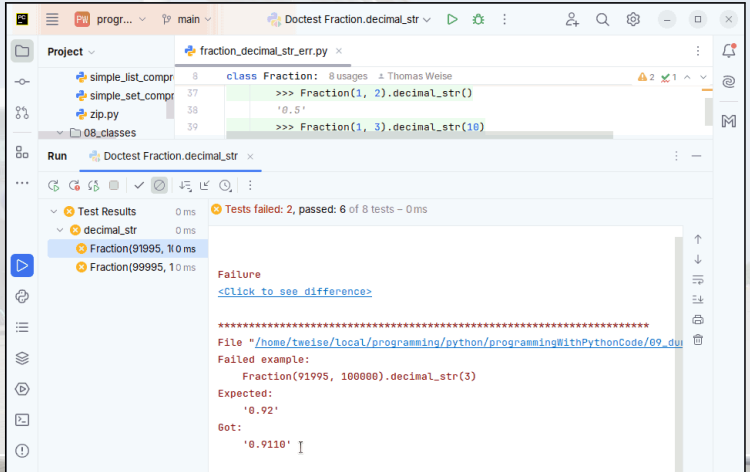
Doctests in PyCharm

- Dadurch werden *alle* Doctests ausgeführt.
- In dem kleinen Fenster unten links können wir die *fehlgeschlagenen* Tests sehen.



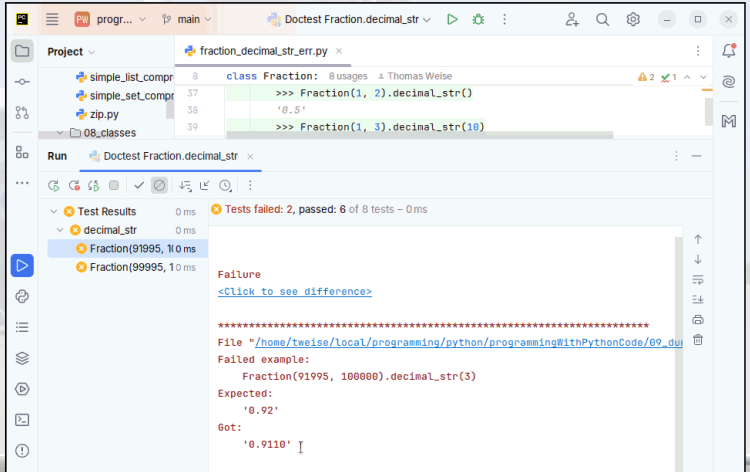
Doctests in PyCharm

- Wir können auf die fehlgeschlagenen Tests klicken, um mehr Informationen zu erhalten.



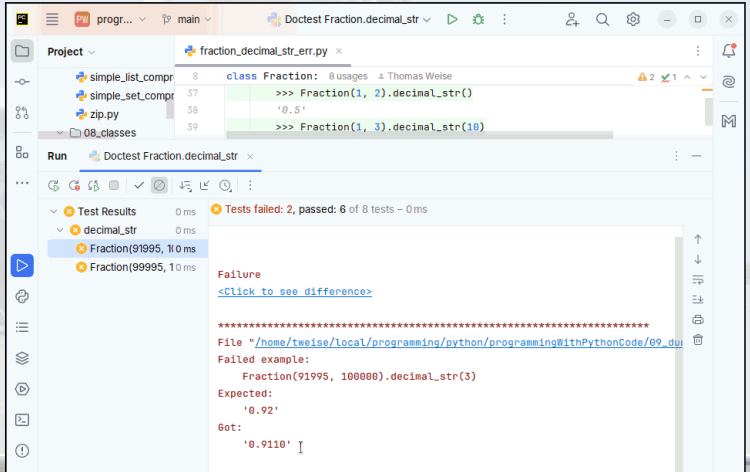
Doctests in PyCharm

- Wir können auf die fehlgeschlagenen Tests klicken, um mehr Informationen zu erhalten.
- Ein links-Klick auf den ersten fehlgeschlagenen Test in diesem Fenster unten links zeigt die Ausgaben dieses Tests im Fenster unten rechts.



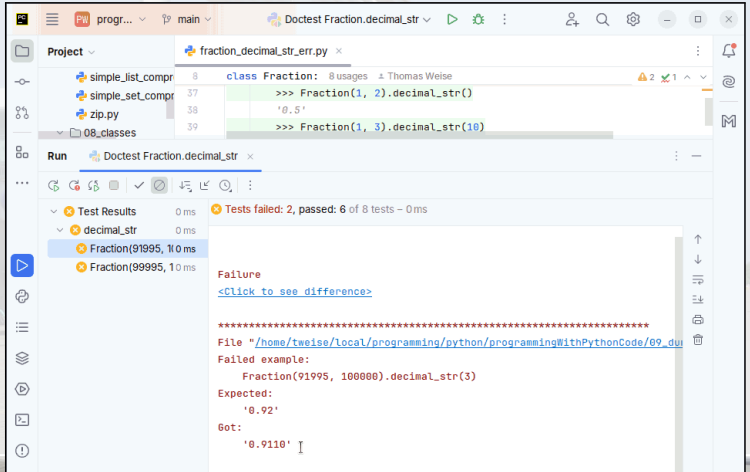
Doctests in PyCharm

- Ein links-Klick auf den ersten fehlgeschlagenen Test in diesem Fenster unten links zeigt die Ausgaben dieses Tests im Fenster unten rechts.
- Das ist die selbe Information, die wir schon gesehen haben.



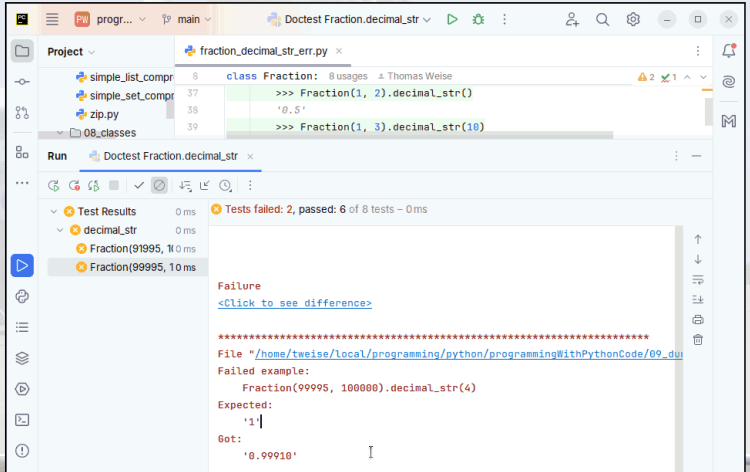
Doctests in PyCharm

- Das ist die selbe Information, die wir schon gesehen haben.
- Was wir noch nicht gesehen hatten, ist das sogar zwei Doctests fehlschlagen.



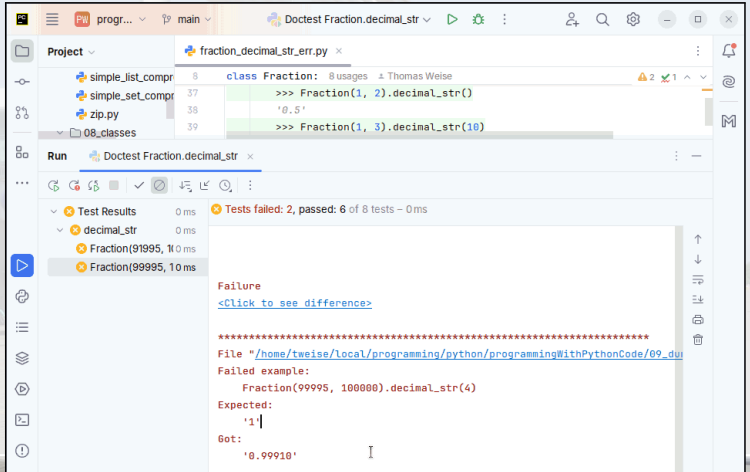
Doctests in PyCharm

- Was wir noch nicht gesehen hatten, ist das sogar zwei Doctests fehlschlagen.
- Ein links-Klick auf den zweiten fehlgeschlagenen Test zeigt uns, dass `Fraction(99995, 100000).decimal_str(4)` nicht wie erwartet `"1"` liefert.



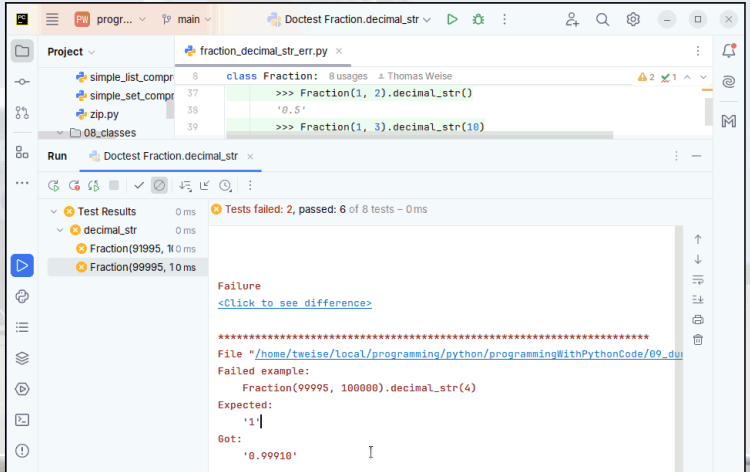
Doctests in PyCharm

- Ein links-Klick auf den zweiten fehlgeschlagenen Test zeigt uns, dass `Fraction(99995, 100000).decimal_str(4)` nicht wie erwartet `"1"` liefert.
- Stattdessen hat es `"0.99910"`. ergeben



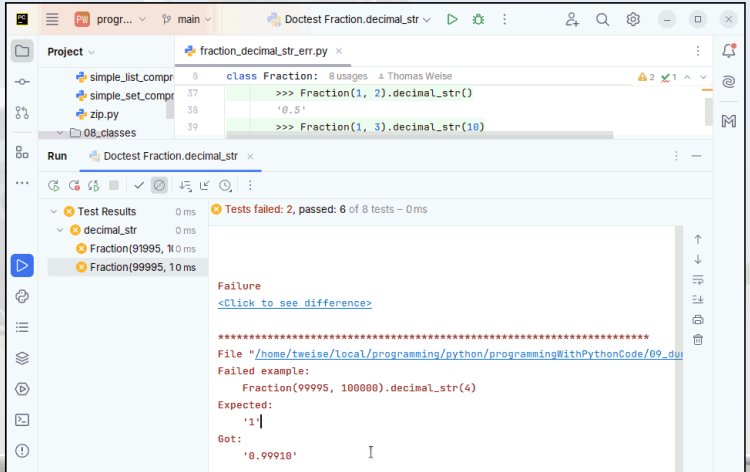
Doctests in PyCharm

- Stattdessen hat es `"0.99910"`. ergeben
- Warum ist da eine `"0"` am Ende unserer Zahl?



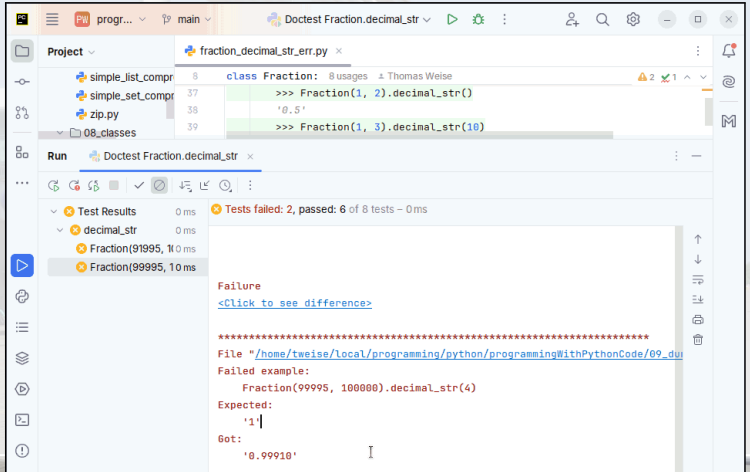
Doctests in PyCharm

- Warum ist da eine `"0"` am Ende unserer Zahl?
- Wo kommt die her?



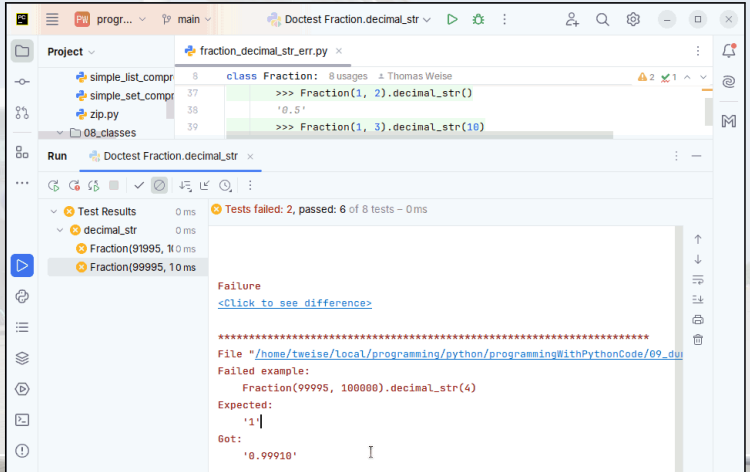
Doctests in PyCharm

- Wo kommt die her?
- Nullen am Ende sollten doch mit unserem Kode gar nicht möglich sein.



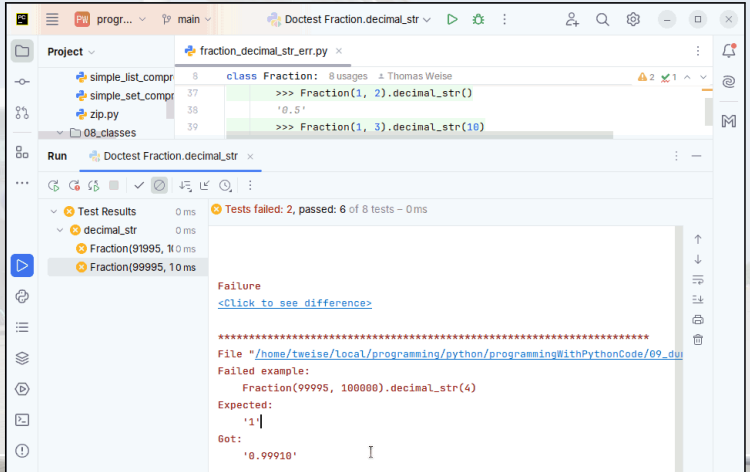
Doctests in PyCharm

- Nullen am Ende sollten doch mit unserem Kode gar nicht möglich sein.
- Außerdem sind da vier Neunen in der Zahl, nicht drei.



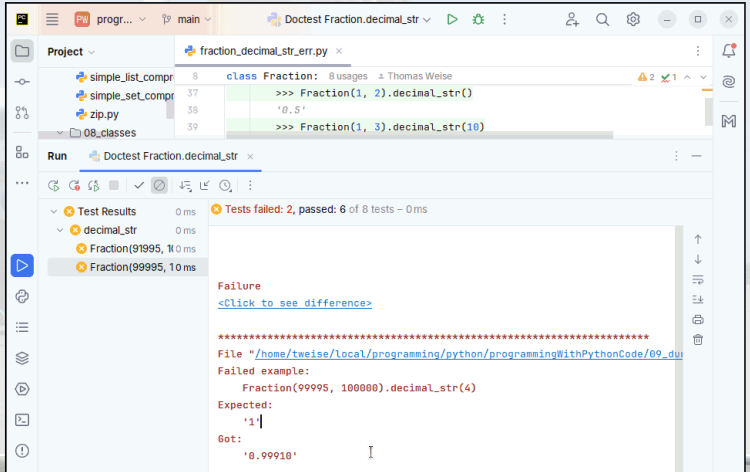
Doctests in PyCharm

- Außerdem sind da vier Neunen in der Zahl, nicht drei.
- Was ist hier schief gegangen?



Doctests in PyCharm

- Was ist hier schief gegangen?
- Wir wissen nicht, warum die Tests fehlschlagen.



Debugger

- Wir fragen uns, was wir jetzt tun können.



Debugger

- Wir fragen uns, was wir jetzt tun können.
- Wenn wir herausfinden wollen, was schief geht, dann wäre es nützlich, wenn wir unser Programm irgendwie Schritt-für-Schritt ausführen könnten.

Debugger

- Wir fragen uns, was wir jetzt tun können.
- Wenn wir herausfinden wollen, was schief geht, dann wäre es nützlich, wenn wir unser Programm irgendwie Schritt-für-Schritt ausführen könnten.
- Als ich erklärt habe, wie `decimal_str` funktioniert, habe ich $\frac{-179}{16}$ als Beispiel für den Ablauf unserer Methode durchexerziert.

Debugger

- Wir fragen uns, was wir jetzt tun können.
- Wenn wir herausfinden wollen, was schief geht, dann wäre es nützlich, wenn wir unser Programm irgendwie Schritt-für-Schritt ausführen könnten.
- Als ich erklärt habe, wie `decimal_str` funktioniert, habe ich $\frac{-179}{16}$ als Beispiel für den Ablauf unserer Methode durchexerziert.
- Wäre es nicht schön, wenn wir unser Programm Schritt-für-Schritt für die Tests durchgehen könnten?

Debugger

- Wir fragen uns, was wir jetzt tun können.
- Wenn wir herausfinden wollen, was schief geht, dann wäre es nützlich, wenn wir unser Programm irgendwie Schritt-für-Schritt ausführen könnten.
- Als ich erklärt habe, wie `decimal_str` funktioniert, habe ich $\frac{-179}{16}$ als Beispiel für den Ablauf unserer Methode durchexerziert.
- Wäre es nicht schön, wenn wir unser Programm Schritt-für-Schritt für die Tests durchgehen könnten?
- Dann könnten wir sehen, was es wirklich macht.

Debugger

- Wir fragen uns, was wir jetzt tun können.
- Wenn wir herausfinden wollen, was schief geht, dann wäre es nützlich, wenn wir unser Programm irgendwie Schritt-für-Schritt ausführen könnten.
- Als ich erklärt habe, wie `decimal_str` funktioniert, habe ich $\frac{-179}{16}$ als Beispiel für den Ablauf unserer Methode durchexerziert.
- Wäre es nicht schön, wenn wir unser Programm Schritt-für-Schritt für die Tests durchgehen könnten?
- Dann könnten wir sehen, was es wirklich macht.
- Nun, das können wir!

Debugger

- Wir fragen uns, was wir jetzt tun können.
- Wenn wir herausfinden wollen, was schief geht, dann wäre es nützlich, wenn wir unser Programm irgendwie Schritt-für-Schritt ausführen könnten.
- Als ich erklärt habe, wie `decimal_str` funktioniert, habe ich $\frac{-179}{16}$ als Beispiel für den Ablauf unserer Methode durchexerziert.
- Wäre es nicht schön, wenn wir unser Programm Schritt-für-Schritt für die Tests durchgehen könnten?
- Dann könnten wir sehen, was es wirklich macht.
- Nun, das können wir!
- Und zwar mit einem Werkzeug genannt Debugger, das mit Python und PyCharm ausgeliefert wird.

Debugger

- Wäre es nicht schön, wenn wir unser Programm Schritt-für-Schritt für die Tests durchgehen könnten?
- Dann könnten wir sehen, was es wirklich macht.
- Nun, das können wir!
- Und zwar mit einem Werkzeug genannt Debugger, das mit Python und PyCharm ausgeliefert wird.

Nützliches Werkzeug

Ein Debugger ist ein Werkzeug, das mit vielen Programmiersprachen und IDEs mit ausgeliefert wird.

Debugger

- Wäre es nicht schön, wenn wir unser Programm Schritt-für-Schritt für die Tests durchgehen könnten?
- Dann könnten wir sehen, was es wirklich macht.
- Nun, das können wir!
- Und zwar mit einem Werkzeug genannt Debugger, das mit Python und PyCharm ausgeliefert wird.

Nützliches Werkzeug

Ein Debugger ist ein Werkzeug, das mit vielen Programmiersprachen und IDEs mit ausgeliefert wird. Es erlaub uns, ein Programm Schritt-für-Schritt auszuführen und dabei die aktuellen Werte von Variablen zu beobachten.

Debugger

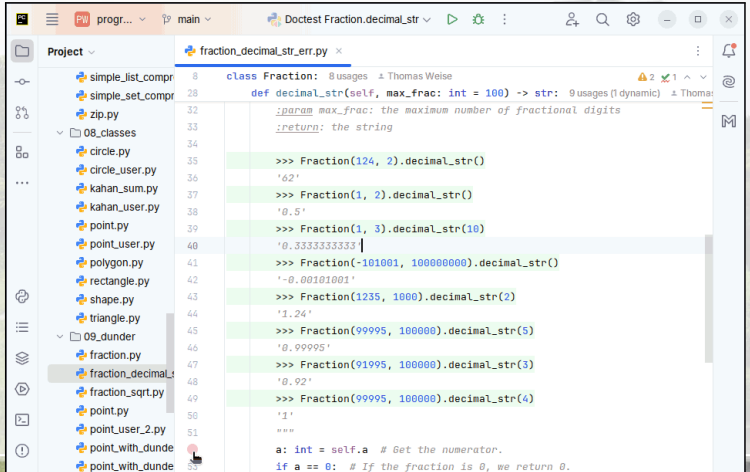
- Wäre es nicht schön, wenn wir unser Programm Schritt-für-Schritt für die Tests durchgehen könnten?
- Dann könnten wir sehen, was es wirklich macht.
- Nun, das können wir!
- Und zwar mit einem Werkzeug genannt Debugger, das mit Python und PyCharm ausgeliefert wird.

Nützliches Werkzeug

Ein Debugger ist ein Werkzeug, das mit vielen Programmiersprachen und IDEs mit ausgeliefert wird. Es erlaub uns, ein Programm Schritt-für-Schritt auszuführen und dabei die aktuellen Werte von Variablen zu beobachten. So können wir Fehler im Kode leichter finden[1,74,107].

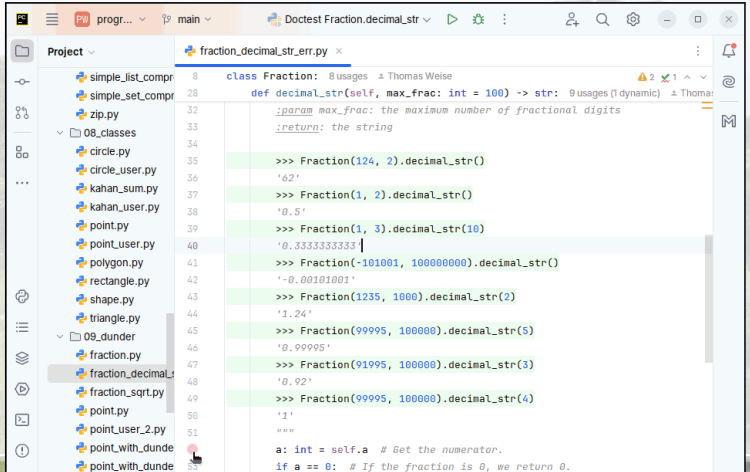
Debuggen in PyCharm

- In PyCharm können wir den Debugger auf ein ganzes Programm anwenden, aber auch auf einen Doctest.



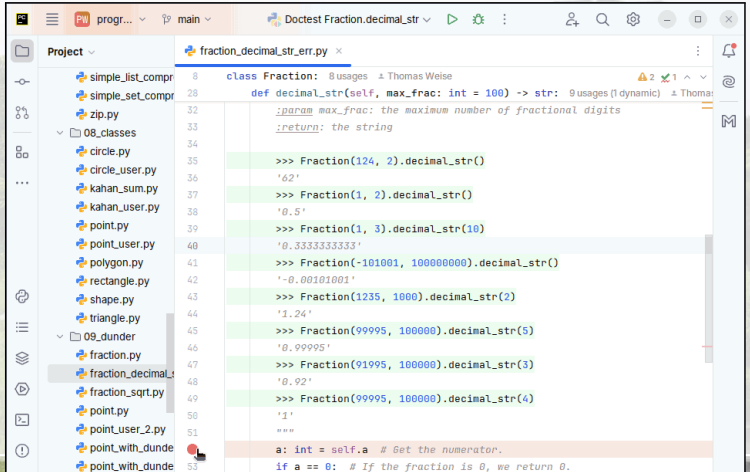
Debuggen in PyCharm

- Dafür müssen wir zuerst unsere Datei `fraction_decimal_str_err.py` öffnen und zu unserer Methode `decimal_str` skrollen.



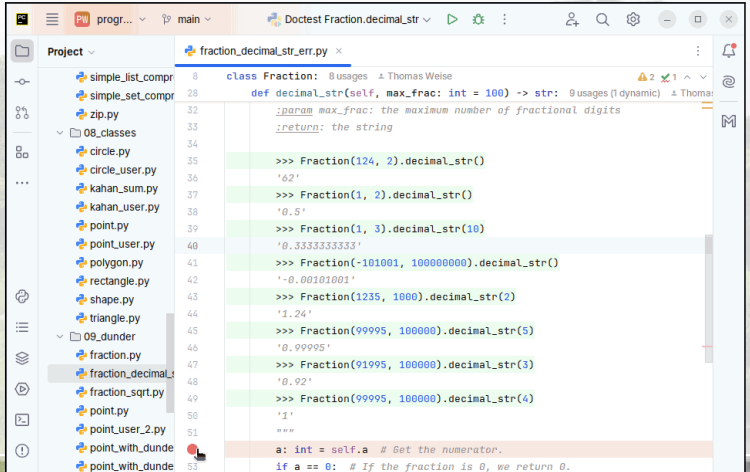
Debuggen in PyCharm

- Auf der linken Seite unseres Kode-Fensters sehen wir eine Spalte mit Zeilennummern.



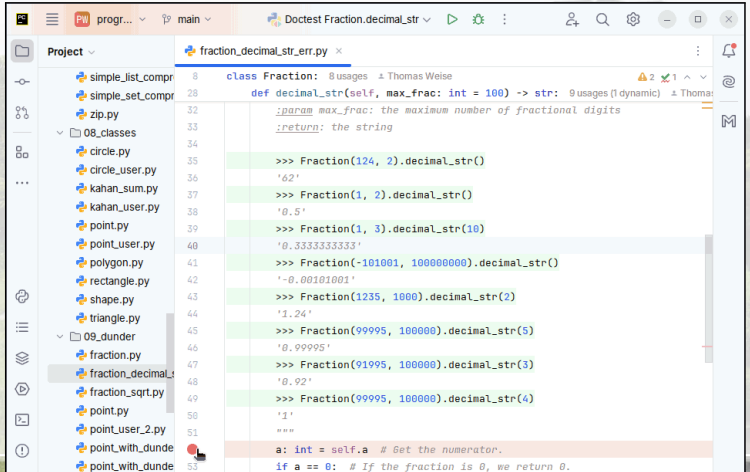
Debuggen in PyCharm

- Auf der linken Seite unseres Kode-Fensters sehen wir eine Spalte mit Zeilennummern.
- Wir können dort links-klicken um einen Breakpoint, also einen Haltepunkt, zu setzen.



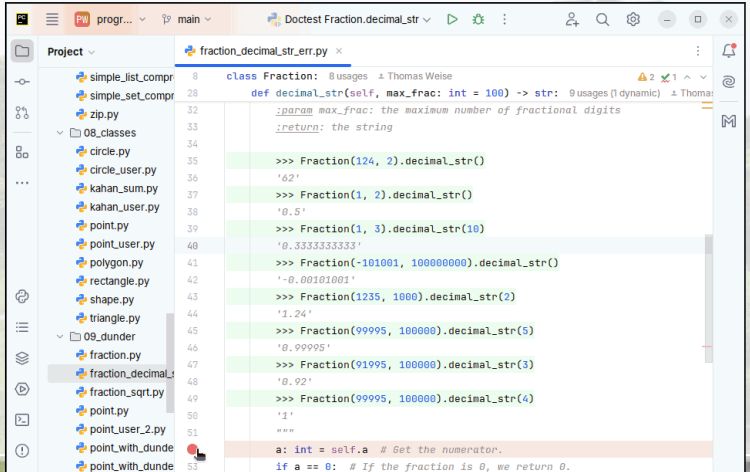
Debuggen in PyCharm

- Wir können dort links-klicken um einen Breakpoint, also einen Haltepunkt, zu setzen.
- Breakpoints sind markierungen in unserer IDE an denen wir später die Ausführung eines Programms pausieren wollen.



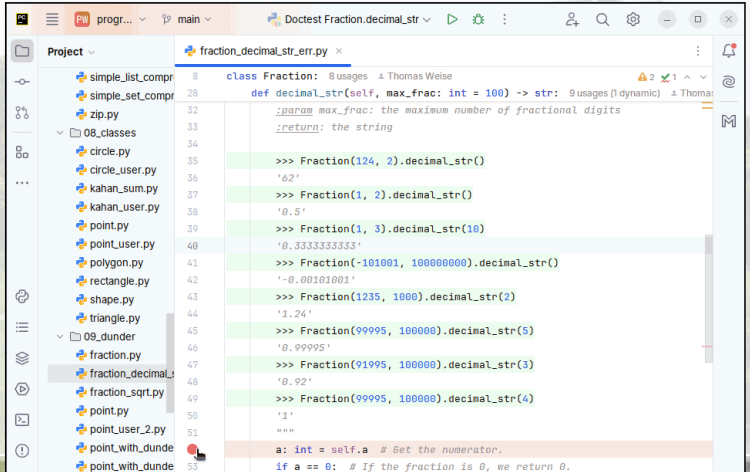
Debuggen in PyCharm

- Breakpoints sind markierungen in unserer IDE an denen wir später die Ausführung eines Programms pausieren wollen.
- Wir wollen genau am Anfang von `decimal_str` pausieren.



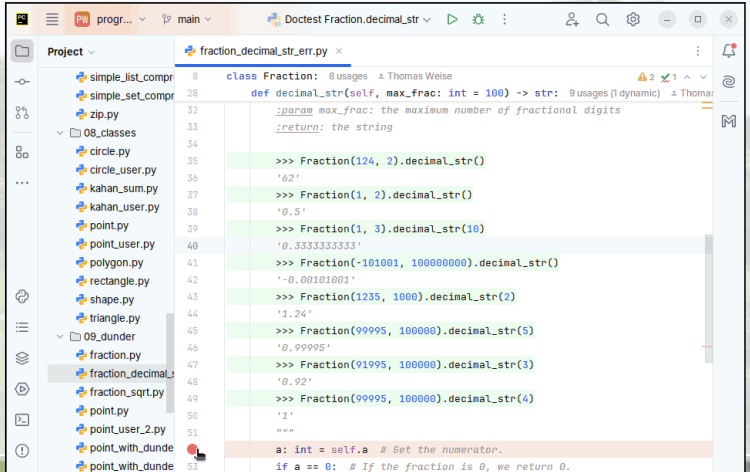
Debuggen in PyCharm

- Wir wollen genau am Anfang von `decimal_str` pausieren.
- Deshalb machen wir genau dort einen Breakpoint hin.



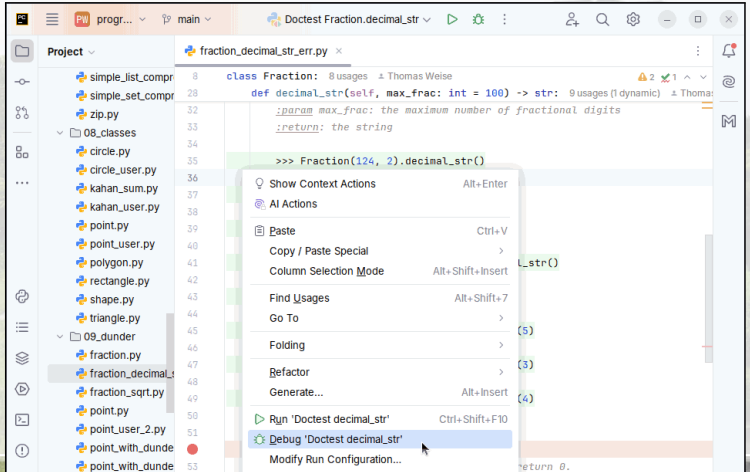
Debuggen in PyCharm

- Deshalb machen wir genau dort einen Breakpoint hin.
- Der Breakpoint wird als roter Ball über der Zeilennummer angezeigt.



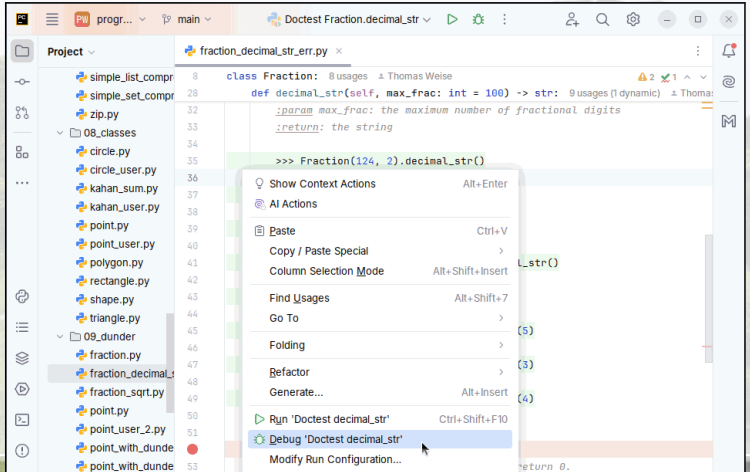
Debuggen in PyCharm

- Um mit dem Debuggen zu beginnen, öffnen wir wieder das Kontextmenü, in dem wir in den Doctest rechts-klicken.



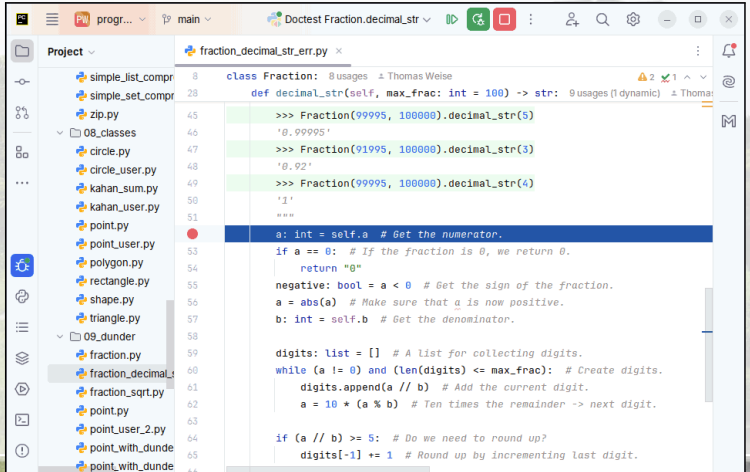
Debuggen in PyCharm

- Um mit dem Debuggen zu beginnen, öffnen wir wieder das Kontextmenü, in dem wir in den Doctest rechts-klicken.
- Dieses mal wählen wir Debug 'Doctest decimal_str' aus.



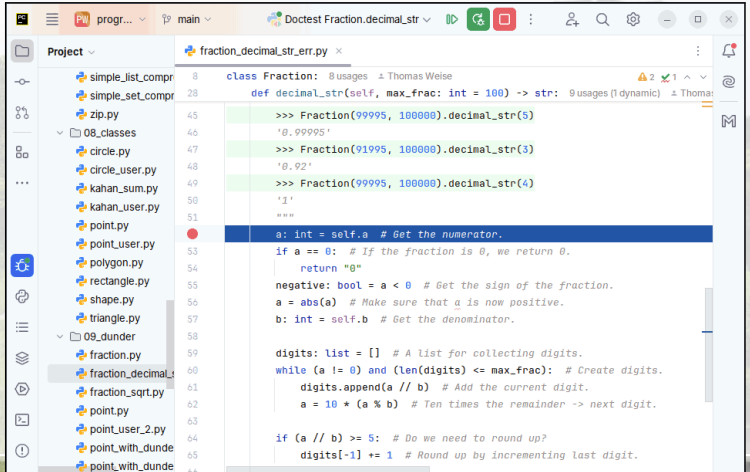
Debuggen in PyCharm

- Die Doctests werden nun ausgeführt.



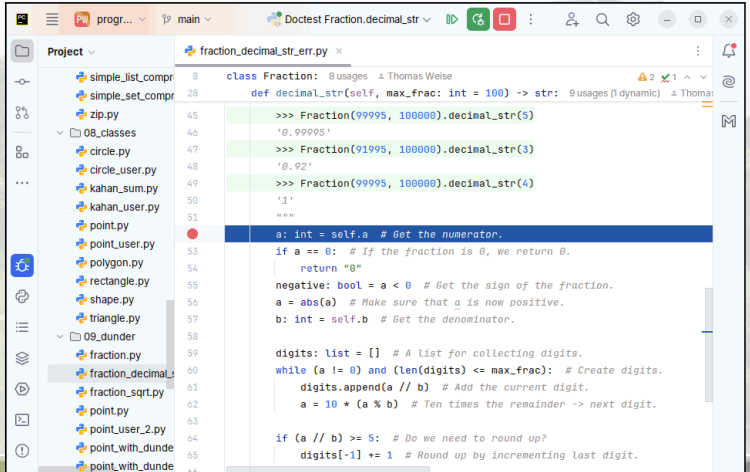
Debuggen in PyCharm

- Die Doctests werden nun ausgeführt.
- Anstatt sie vollständig auszuführen, wird der Debugger aktiv.



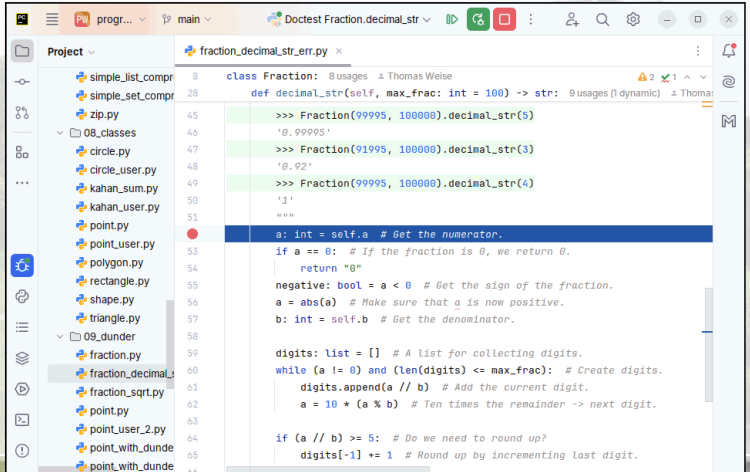
Debuggen in PyCharm

- Anstatt sie vollständig auszuführen, wird der Debugger aktiv.
- Die Ausführung wird genau an unserem Breakpoint pausiert.



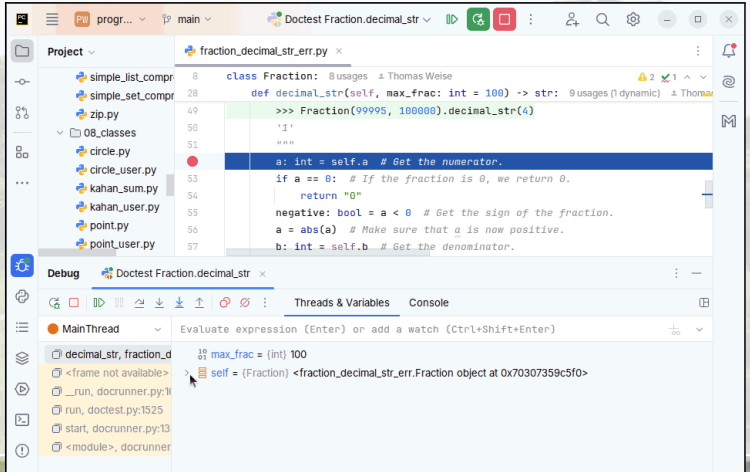
Debuggen in PyCharm

- Die Ausführung wird genau an unserem Breakpoint pausiert.
- Diese Zeile Kode wird noch nicht ausgeführt, abre in blau markiert.



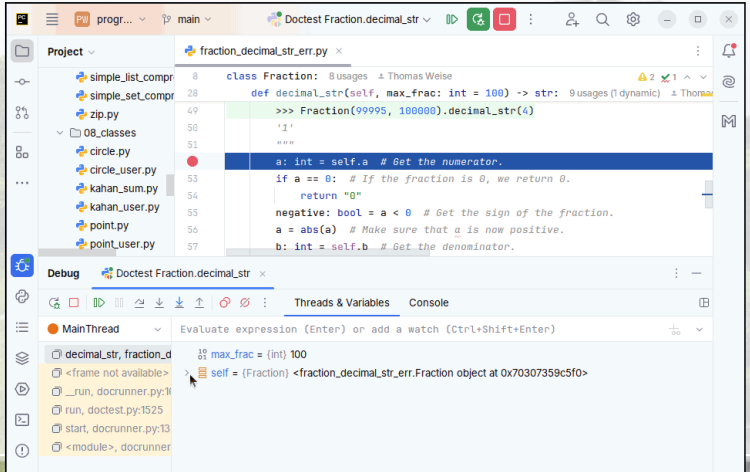
Debuggen in PyCharm

- Bevor wir weitermachen, schauen wir in das untere PyCharm-Fenster.



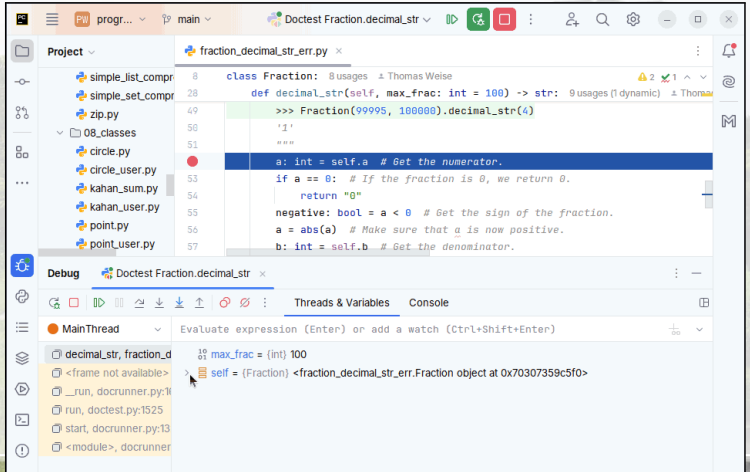
Debuggen in PyCharm

- Bevor wir weitermachen, schauen wir in das untere PyCharm-Fenster.
- Dort gibt es eine `Debug`-Zeile.



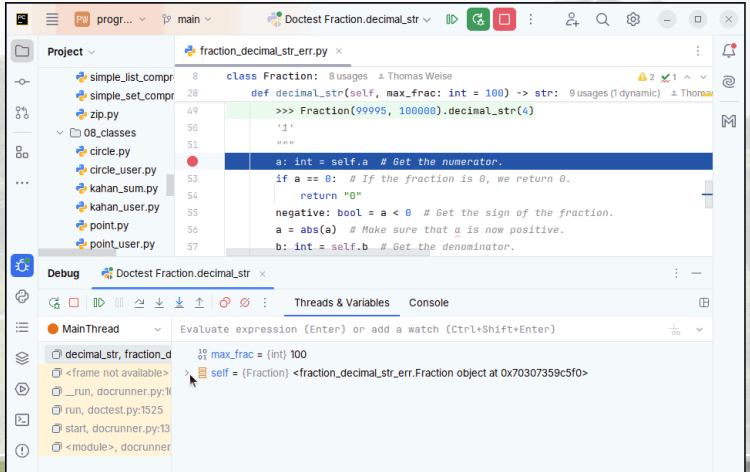
Debuggen in PyCharm

- Dort gibt es eine Debug-Zeile.
- Wir können sie mit der rechten Maustaste aktivieren und hochziehen.



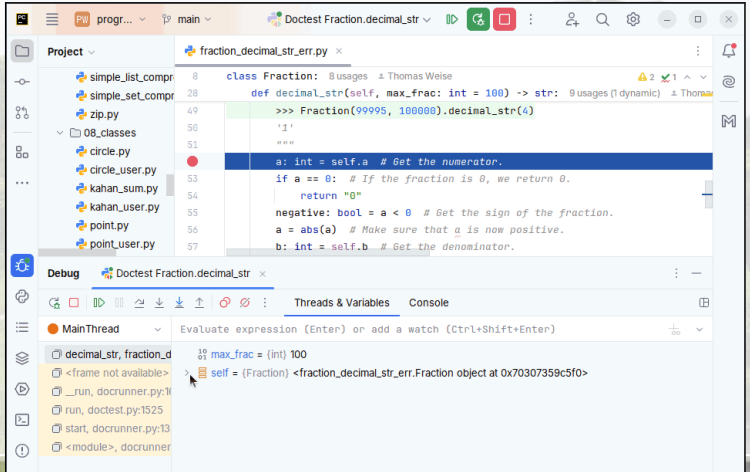
Debuggen in PyCharm

- Wir können sie mit der rechten Maustaste aktivieren und hochziehen.
- Wir sehen nun ein Abteil unserers Fensters das die Debug-Informationen beinhaltet.



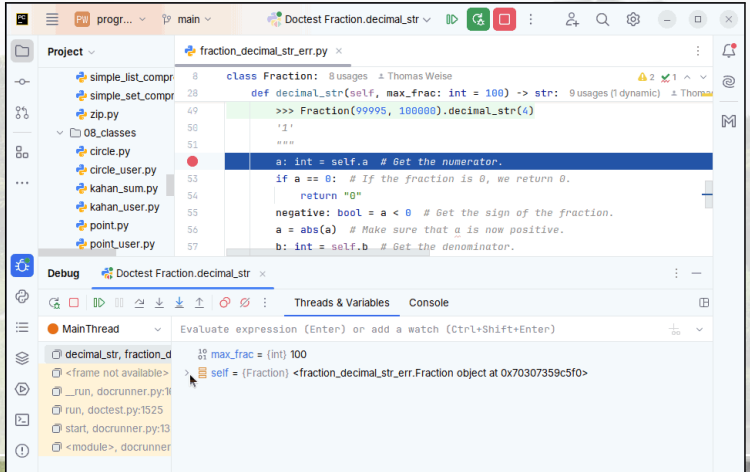
Debuggen in PyCharm

- Wir sehen nun ein Abteil unserers Fensters das die Debug-Informationen beinhaltet.
- Das wichtigste ist der Register Threads & Variables.



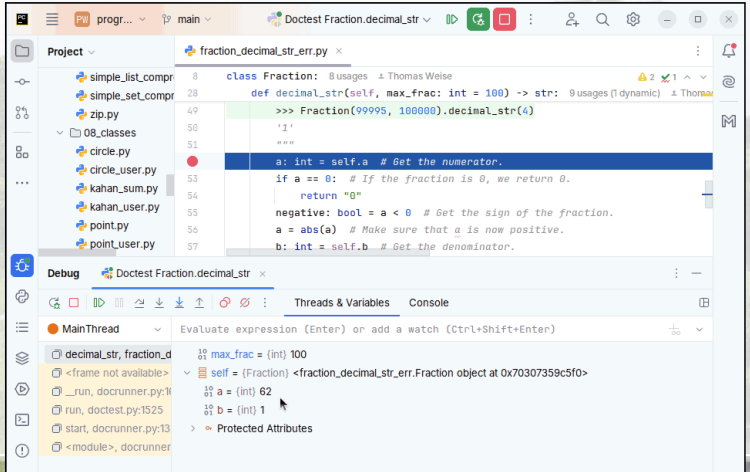
Debuggen in PyCharm

- Das wichtigste ist der Register Threads & Variables.
- Hier können wir die Werte aller lokaler Variablen am aktuellen Ausführungspunkt sehen.



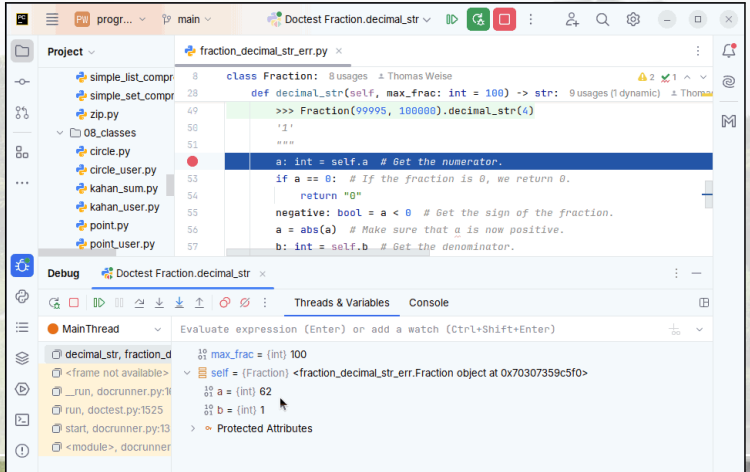
Debuggen in PyCharm

- Wir sehen das `max_frac` den (Default-)Wert `100` hat.



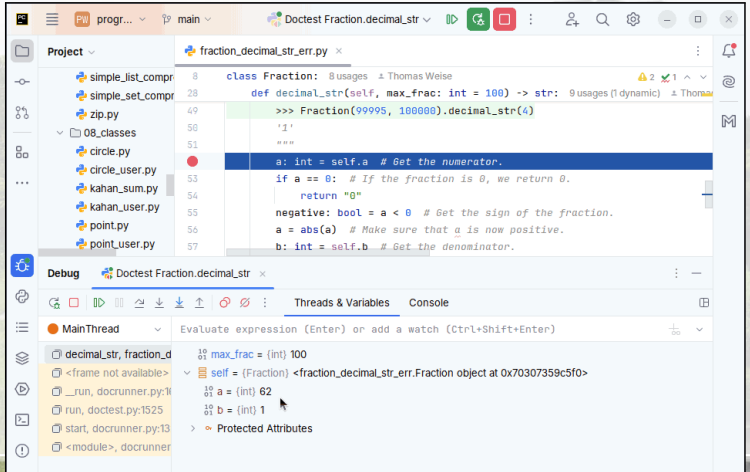
Debuggen in PyCharm

- Wenn wir auf die Variable `self` klicken, sehen wir das der Zähler `a` des aktuellen Bruchs den Wert `62` hat, während der Nenner `b` den Wert `1` hat.



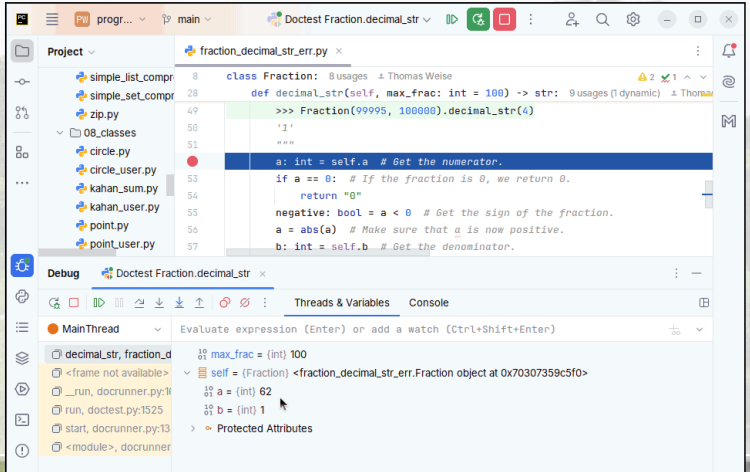
Debuggen in PyCharm

- Das ist genau was wir erwarten.



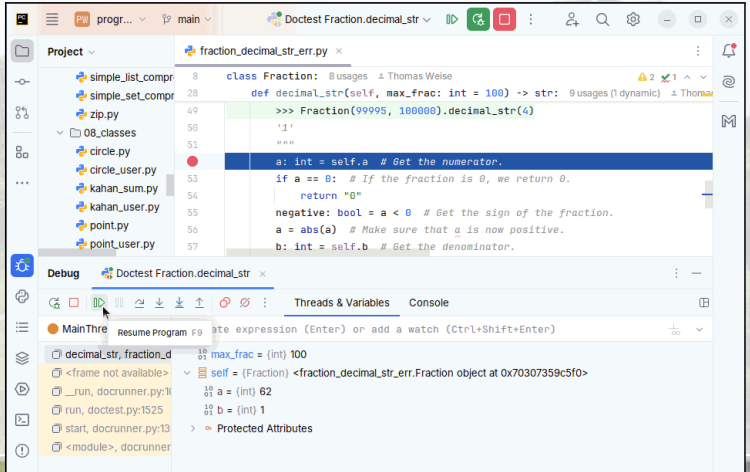
Debuggen in PyCharm

- Das ist genau was wir erwarten.
- Der erste Test Case war ja `Fraction(124, 2).decimal_str()`, also ist der normalisierte Bruch korrekt $\frac{62}{1}$.



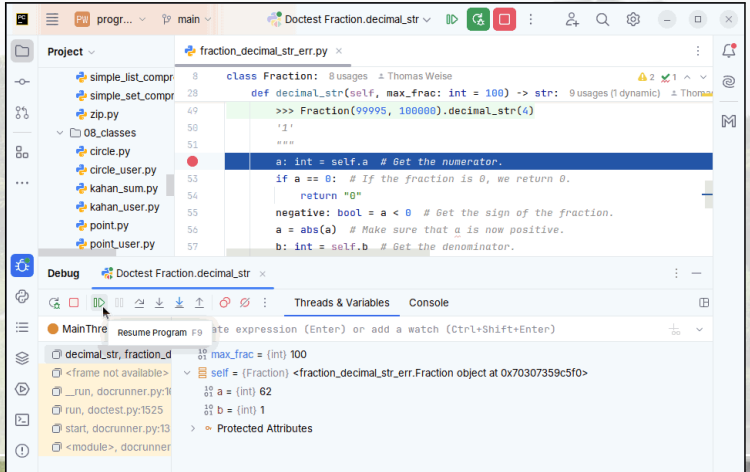
Debuggen in PyCharm

- Wir wissen bereits, dass dieser Test Case erfolgreich durchlaufen werden wird.



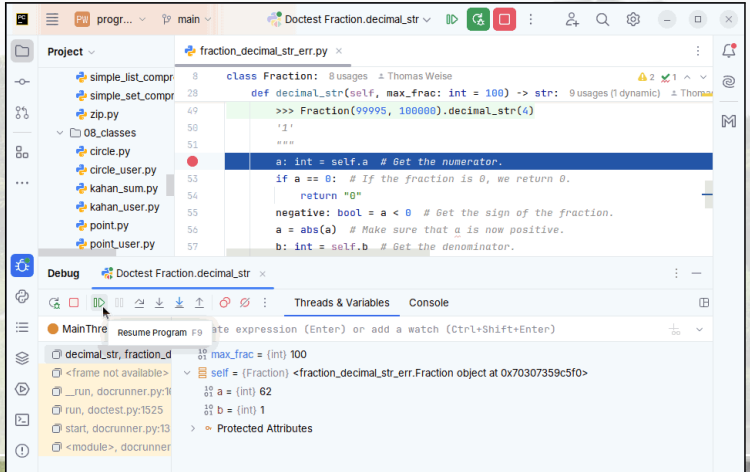
Debuggen in PyCharm

- Wir wissen bereits, dass dieser Test Case erfolgreich durchlaufen werden wird.
- Deshalb interessiert er uns nicht.



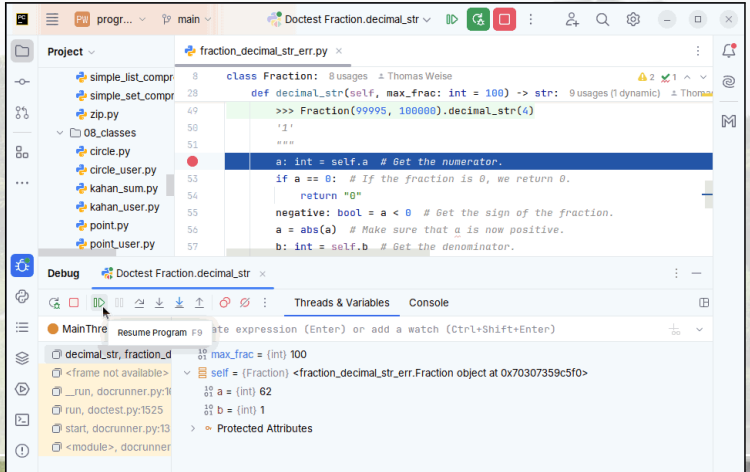
Debuggen in PyCharm

- Deshalb interessiert er uns nicht.
- Wir klicken auf das Symbol ▷ im Debug-Register, wodurch das Program weiter ausgeführt wird.



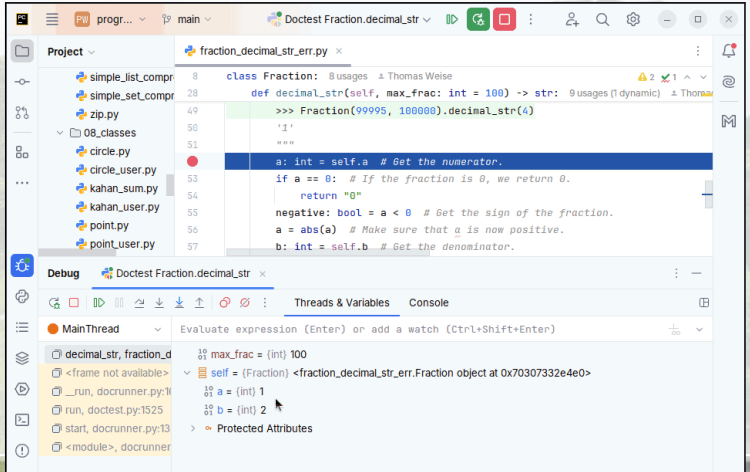
Debuggen in PyCharm

- Wir klicken auf das Symbol im Debug-Register, wodurch das Program weiter ausgeführt wird.
- Alternativ können wir auch einfach F9 drücken.



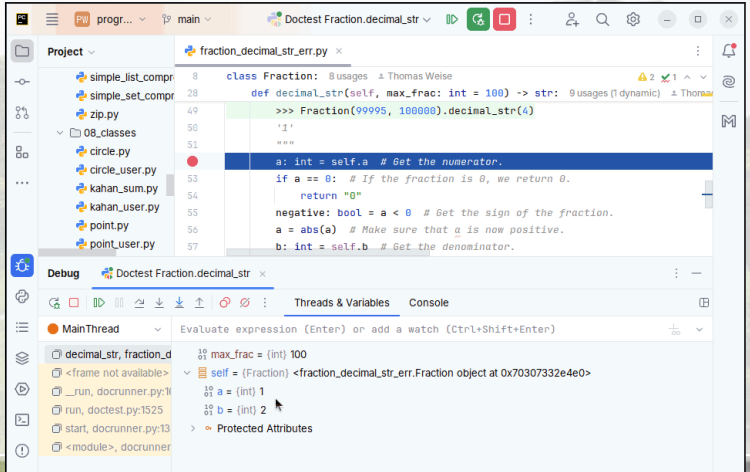
Debuggen in PyCharm

- Die Ausführung des Doctests wird fortgesetzt.



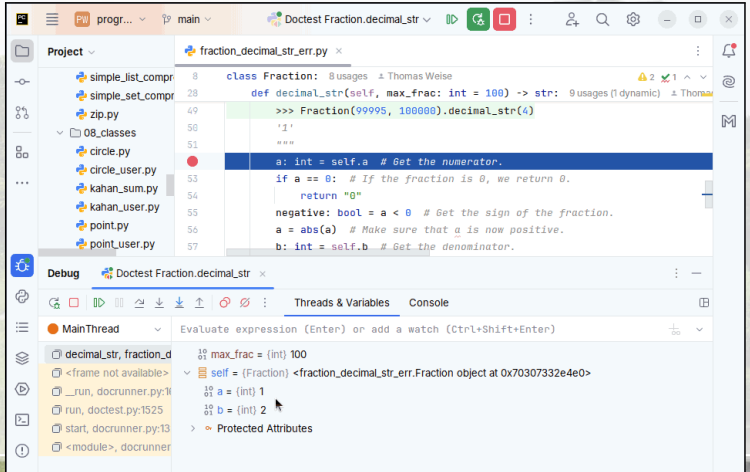
Debuggen in PyCharm

- Die Ausführung des Doctests wird fortgesetzt.
- Sie wird wieder an unserem Breakpoint pausiert.



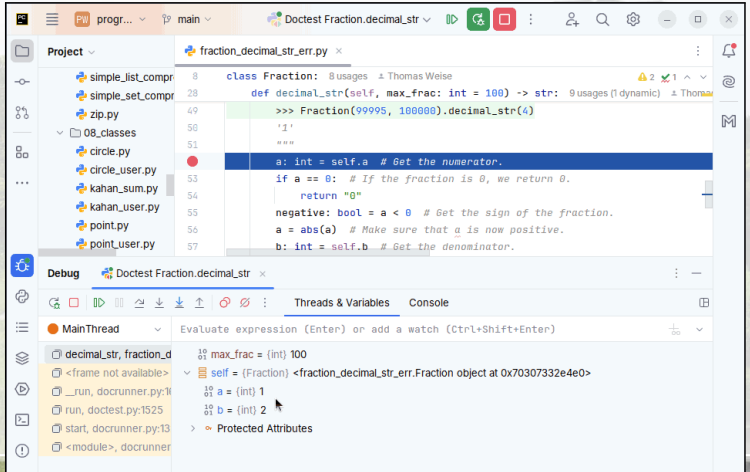
Debuggen in PyCharm

- Sie wird wieder an unserem Breakpoint pausiert.
- Dieses Mal sind wir beim zweiten Doctest angekommen, der `Fraction(1, 2)` als Daten hat.



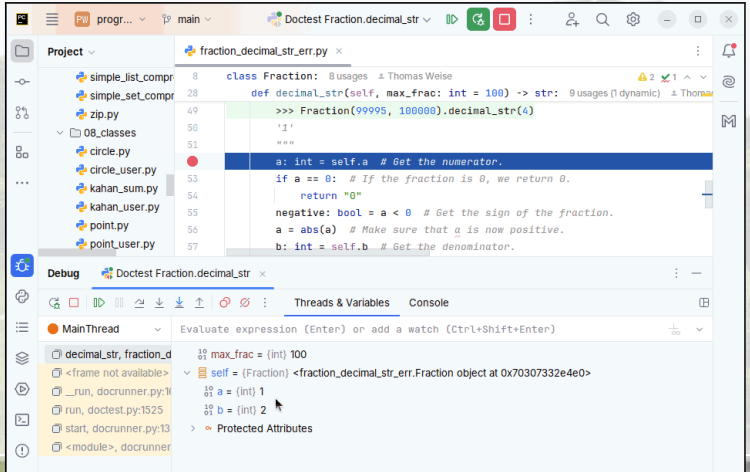
Debuggen in PyCharm

- Dieses Mal sind wir beim zweiten Doctest angekommen, der `Fraction(1, 2)` als Daten hat.
- Auch dieser Test Case ist uninteressant.



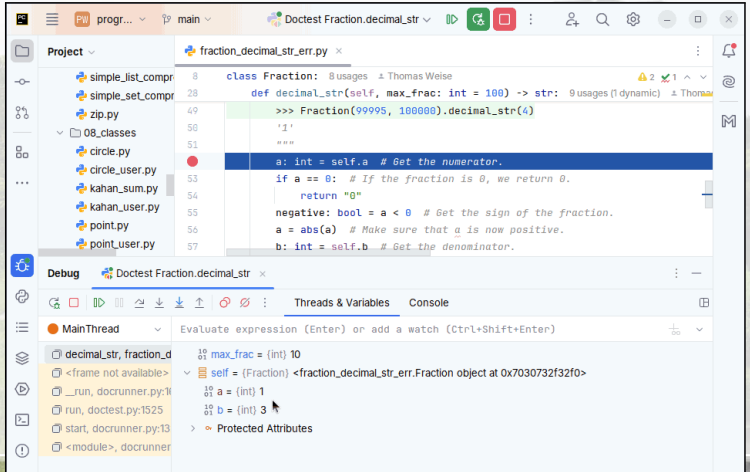
Debuggen in PyCharm

- Auch dieser Test Case ist uninteressant.
- Also klicken wir wieder auf ▷ oder drücken F9 um die Ausführung fortzusetzen.



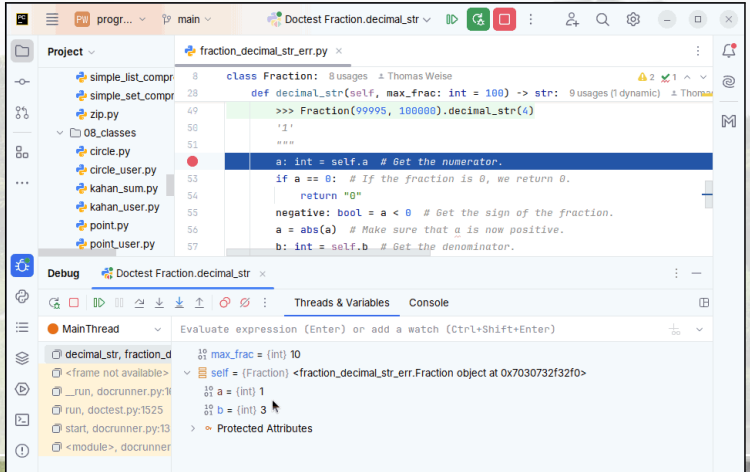
Debuggen in PyCharm

- Das bringt uns an den Anfang des dritten Doctest Case, wo der Bruch $\frac{1}{3}$ mit `max_frac` gleich `10` untersucht wird.



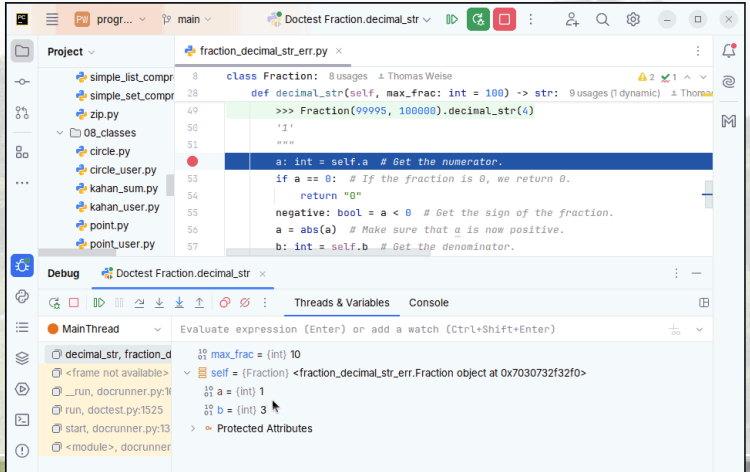
Debuggen in PyCharm

- Das bringt uns an den Anfang des dritten Doctest Case, wo der Bruch $\frac{1}{3}$ mit `max_frac` gleich `10` untersucht wird.
- Auch dieser Test Case wird erfolgreich sein, das wissen wir bereits.



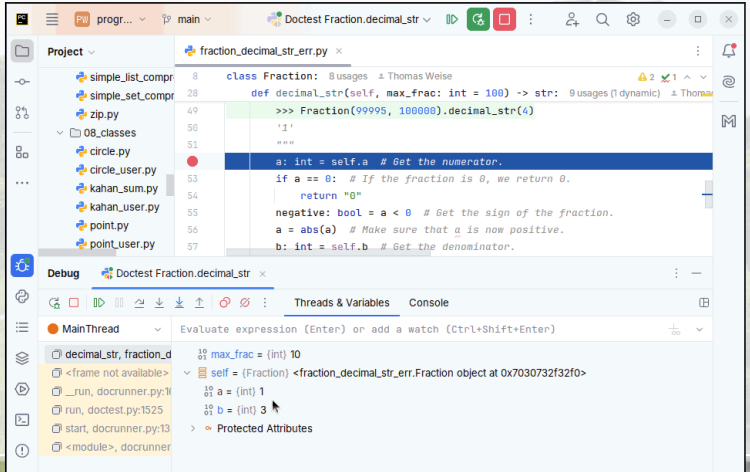
Debuggen in PyCharm

- Auch dieser Test Case wird erfolgreich sein, das wissen wir bereits.
- Er wird uns keine nützlichen Informationen liefern.



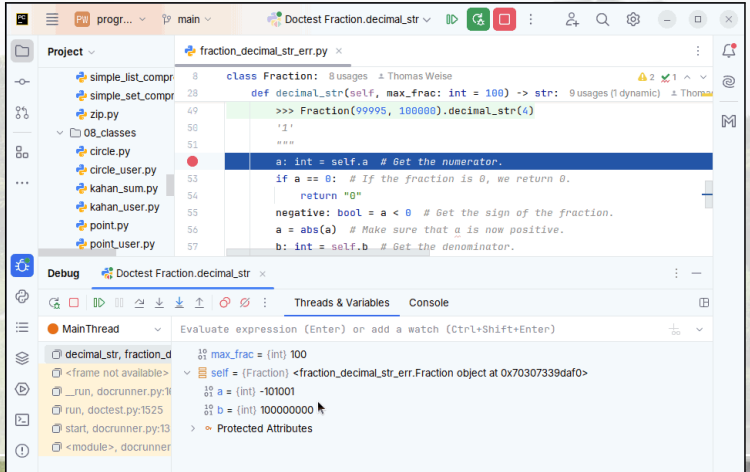
Debuggen in PyCharm

- Er wird uns keine nützlichen Informationen liefern.
- Wir überspringen ihn mit F9.



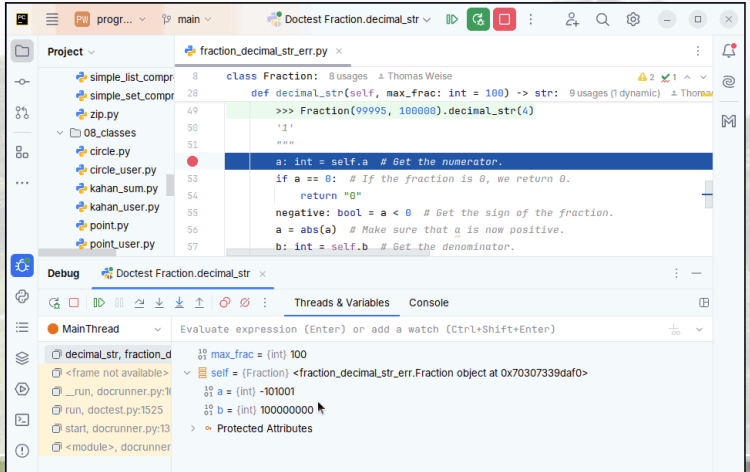
Debuggen in PyCharm

- Als wir den Breakpoint wieder erreichen, sind wir im vierten Doctest Case angekommen: $\frac{-101001}{100000000}$.



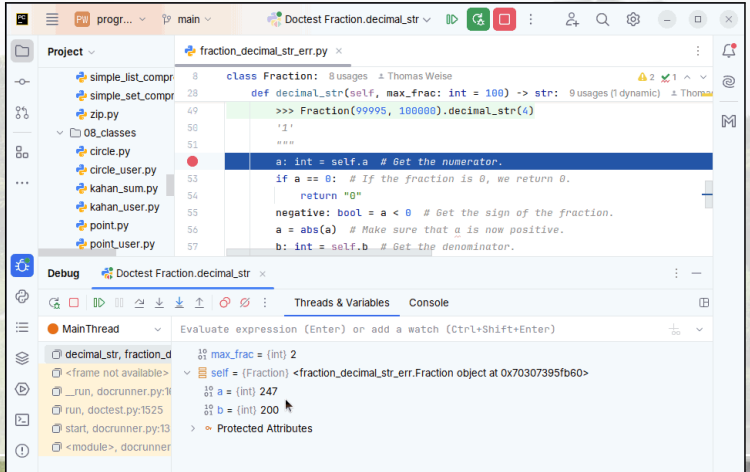
Debuggen in PyCharm

- Als wir den Breakpoint wieder erreichen, sind wir im vierten Doctest Case angekommen: $\frac{-101001}{100000000}$.
- Auch diesen überspringen wir.



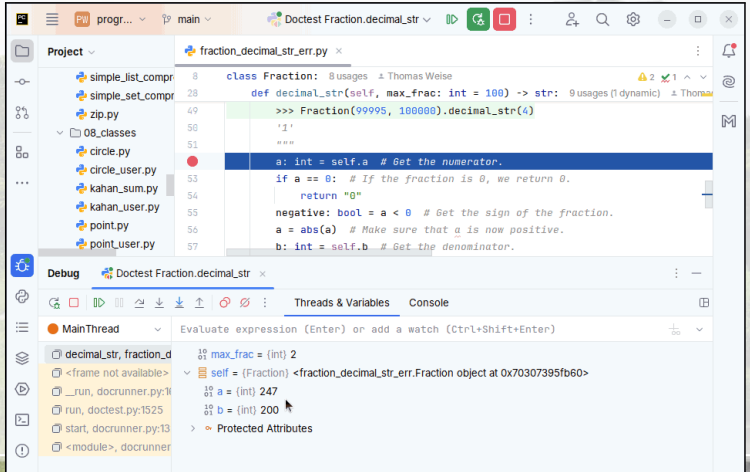
Debuggen in PyCharm

- Wenn der Debugger am fünften Test Case ankommt, sehen wir, dass der Bruch `Fraction(1235, 1000)` korrekt zu $\frac{247}{200}$ normalisiert wurde.



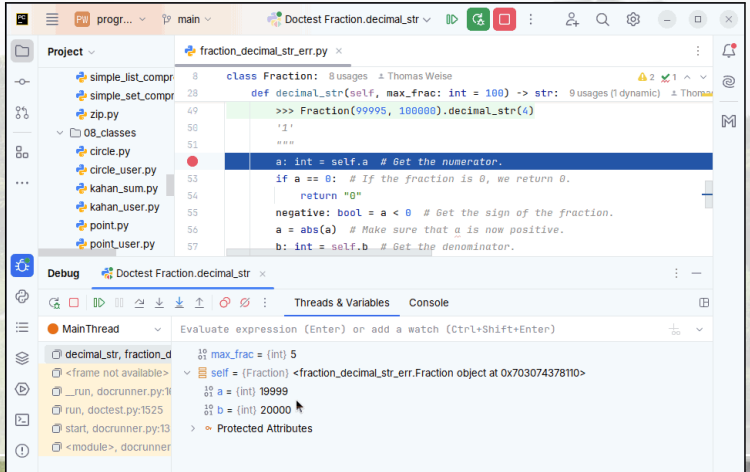
Debuggen in PyCharm

- Wir überspringen den Test Case trotzdem mit F9, denn wir wissen ja, das er erfolgreich seien wird.



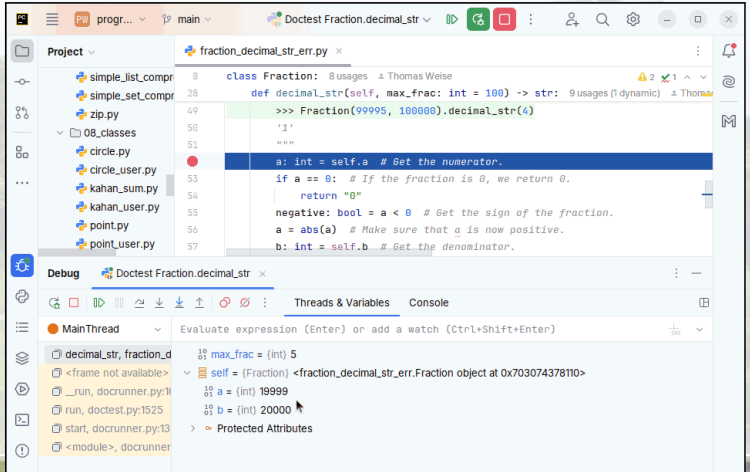
Debuggen in PyCharm

- Das bringt uns zum letzten erfolgreichen Test Case, `Fraction(99995, 100000)`, was dem Bruch $\frac{19999}{20000}$ entspricht.



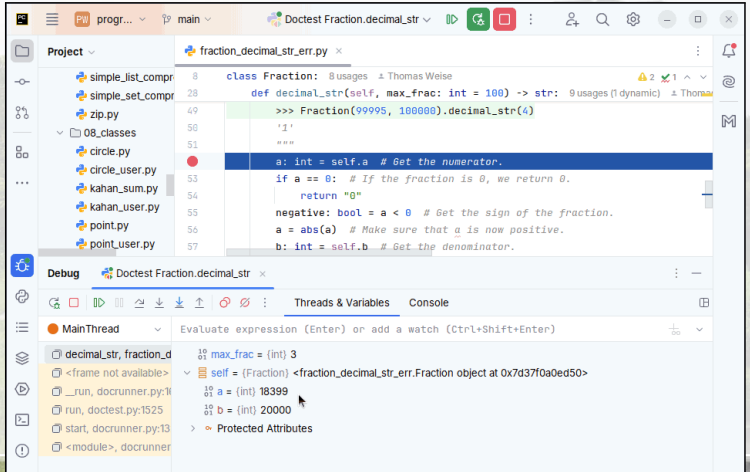
Debuggen in PyCharm

- Nach dem wir diesen Test Case mit ▯▷ übersprungen haben, werden wir endlich in einem Test Case angekommen, der fehlschlagen wird, und den wir deshalb Schritt-für-Schritt durchgehen müssen.



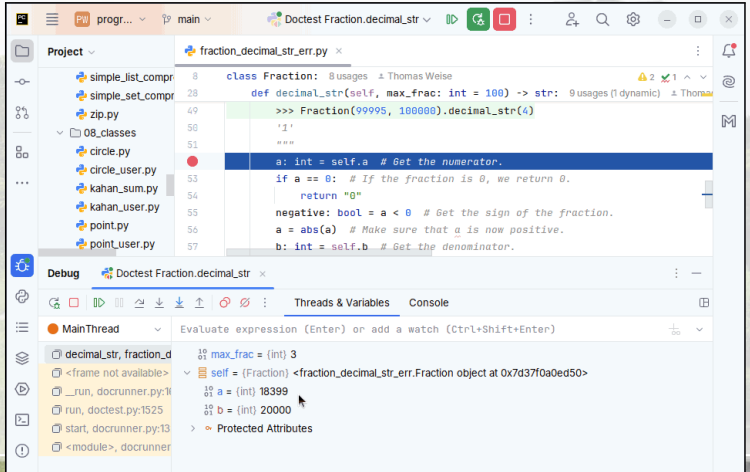
Debuggen in PyCharm

- Wir sind jetzt am Anfang des Doctest Cases `Fraction(91995, 100000).decimal_str(3)` angekommen, der fehlschlagen wird.



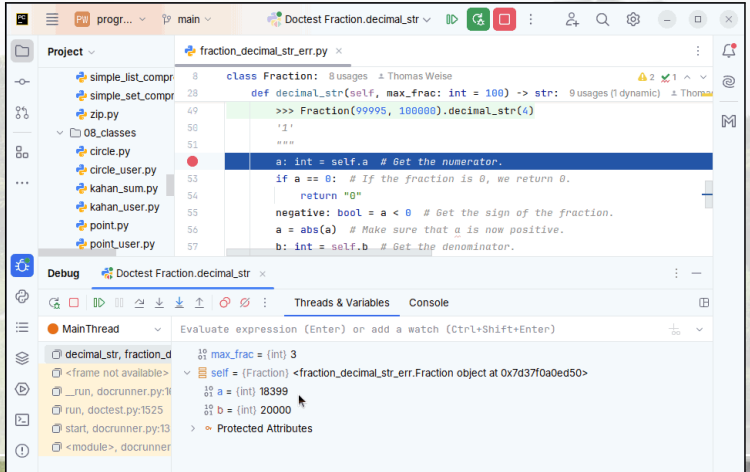
Debuggen in PyCharm

- Der Bruch $\frac{91995}{100000}$ wurde im Initializer `__init__` zu $\frac{18399}{20000}$ normalisiert.



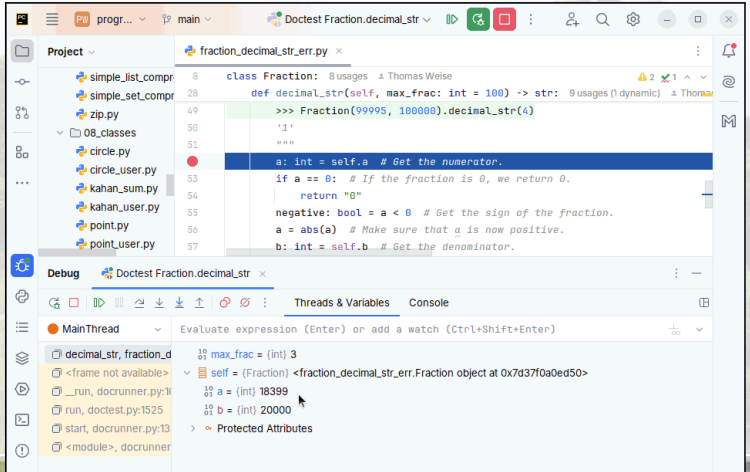
Debuggen in PyCharm

- Der Parameter `max_frac` von `decimal_str` hat den Wert `3`, wir wir im `Threads & Variables`-Fenster sehen.



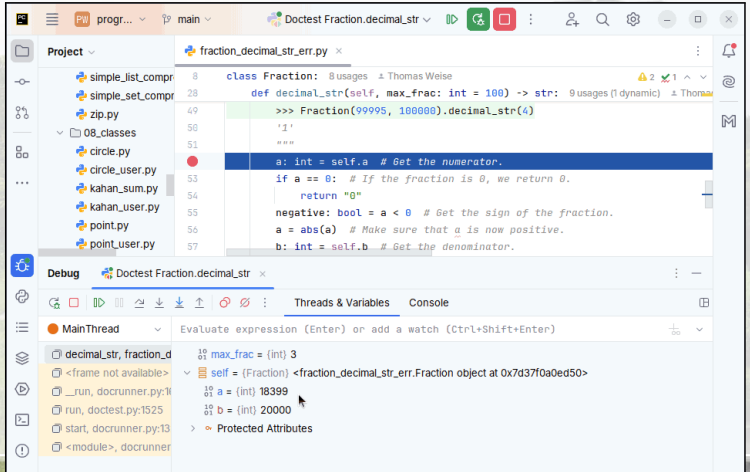
Debuggen in PyCharm

- Der Parameter `max_frac` von `decimal_str` hat den Wert `3`, wir wir im `Threads & Variables`-Fenster sehen.
- Wir wollen nun die Methode `decimal_str` Schritt-für-Schritt ausführen.



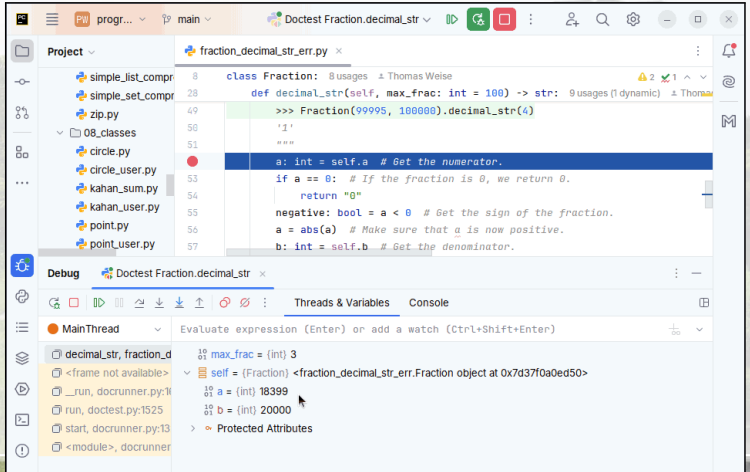
Debuggen in PyCharm

- Wir wollen nun die Methode `decimal_str` Schritt-für-Schritt ausführen.
- Nun hat der Debugger erstmal die Ausführung an der ersten Zeile der Methode pausiert.



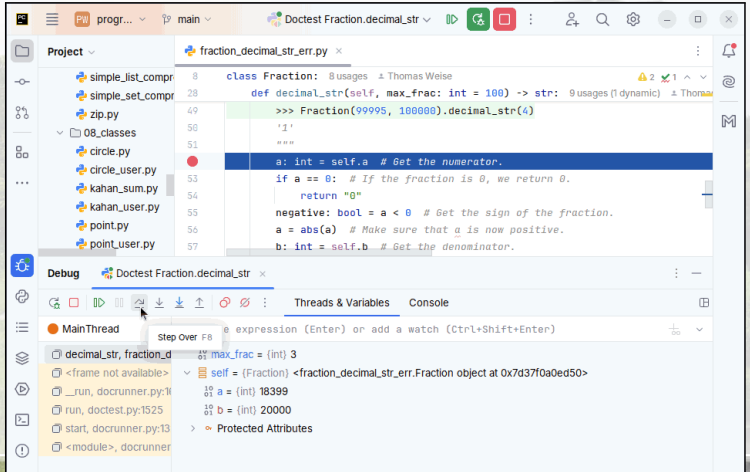
Debuggen in PyCharm

- Nun hat der Debugger erstmal die Ausführung an der ersten Zeile der Methode pausiert.
- Diese Zeile wurde noch nicht ausgeführt.



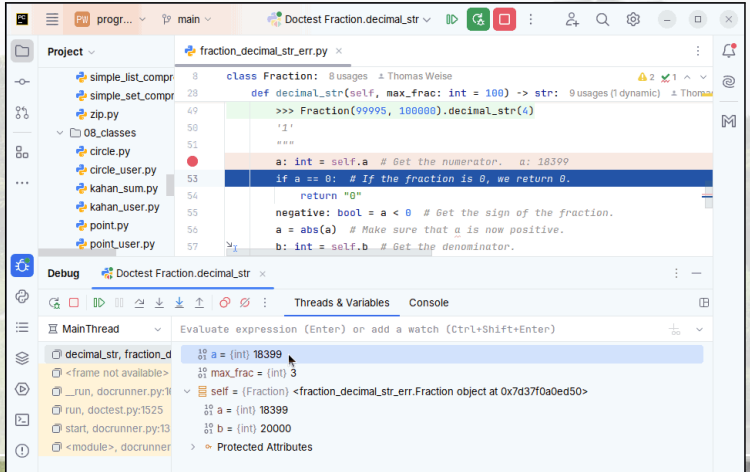
Debuggen in PyCharm

- Wir führen diese Zeile Kode aus, in dem wir entweder auf ⤵ klicken oder F8 drücken.



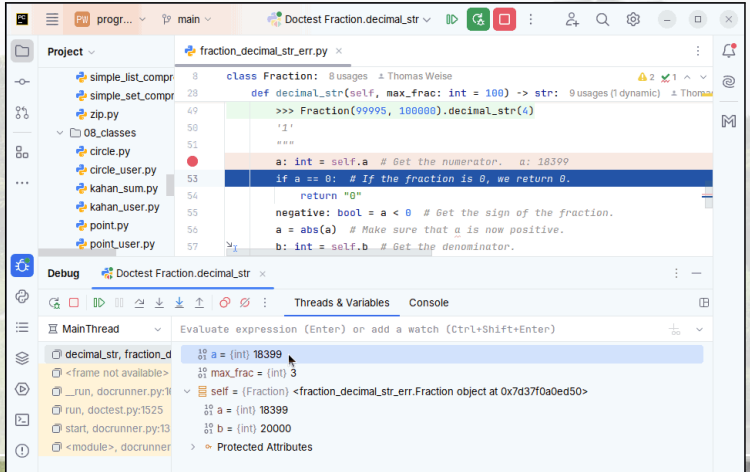
Debuggen in PyCharm

- Wir sehen, dass eine neue Variable im `Threads & Variables`-Fenster auftaucht.



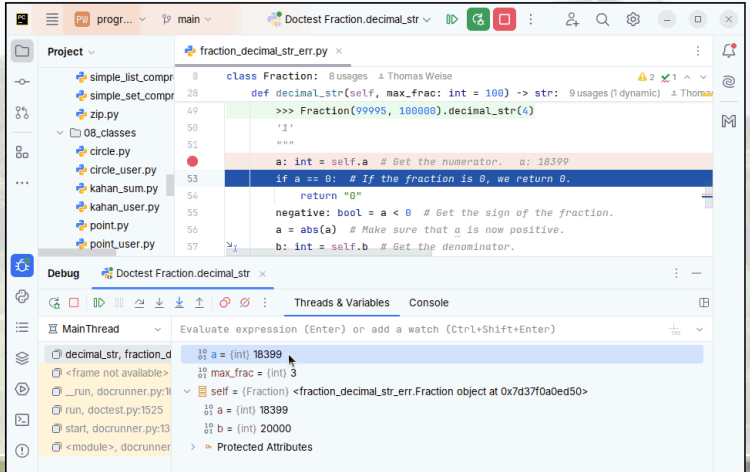
Debuggen in PyCharm

- Wir sehen, dass eine neue Variable im `Threads & Variables`-Fenster auftaucht.
- Da wir `a = self.a` ausgeführt haben, gibt es jetzt die lokale Variable `a` mit dem Wert `18399`.



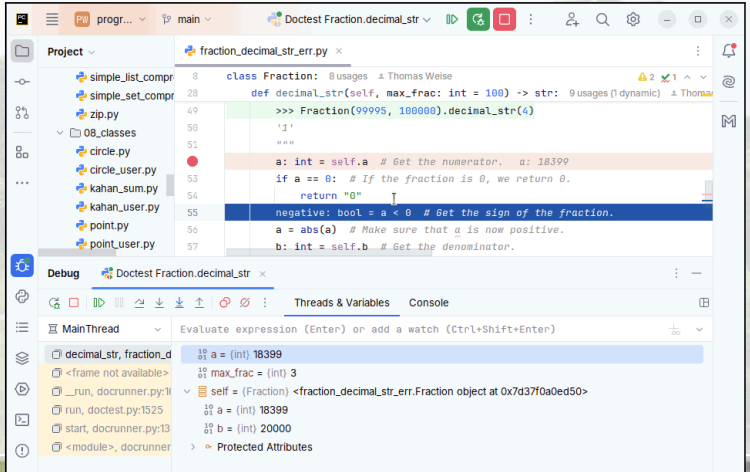
Debuggen in PyCharm

- Da wir `a = self.a` ausgeführt haben, gibt es jetzt die lokale Variable `a` mit dem Wert `18399`.
- Die nächste Kodezeile kann nun ausgeführt werden und ist mit blauer Farbe markiert.



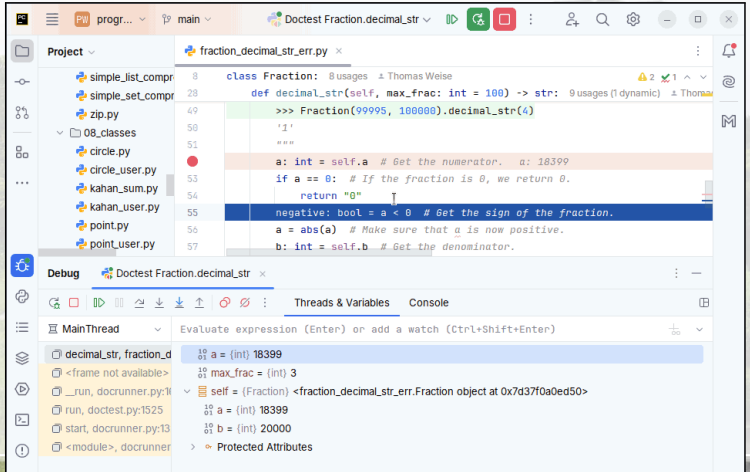
Debuggen in PyCharm

- Wir drücken F8 und führen damit die Zeile `if a == 0:` aus.



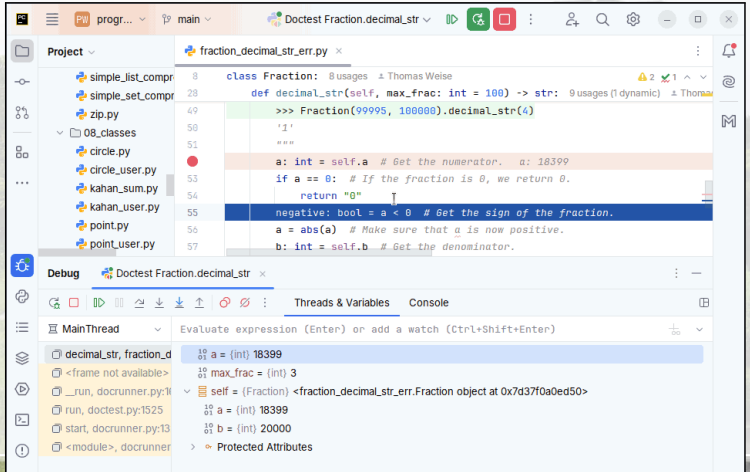
Debuggen in PyCharm

- Wir drücken F8 und führen damit die Zeile `if a == 0:` aus.
- Weil `a == 0` nicht `True` ist, wird der Körper des `if` nicht ausgeführt.



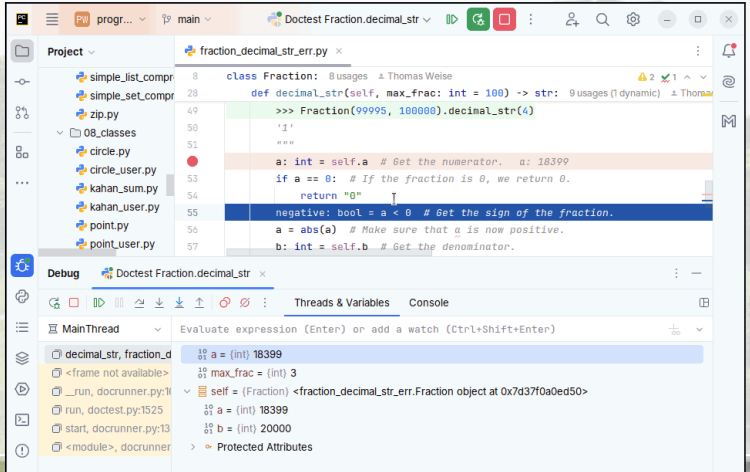
Debuggen in PyCharm

- Weil `a == 0` nicht `True` ist, wird der Körper des `if` nicht ausgeführt.
- Das Programm springt darüber.



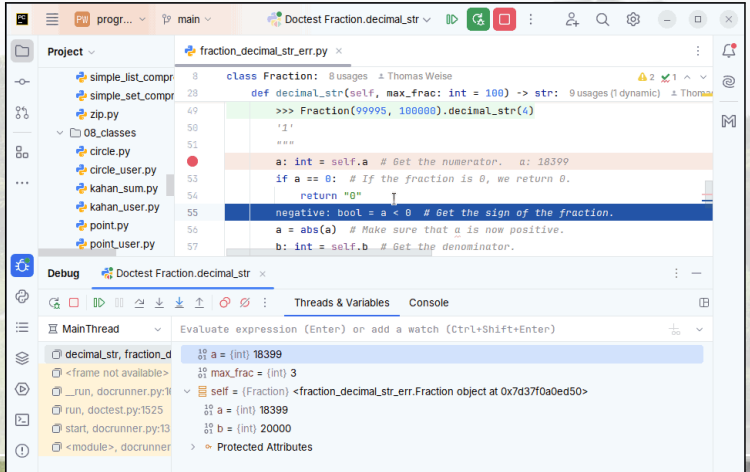
Debuggen in PyCharm

- Das Programm springt darüber.
- Die nächste Kodezeile nach dem `if` ist nun markiert.



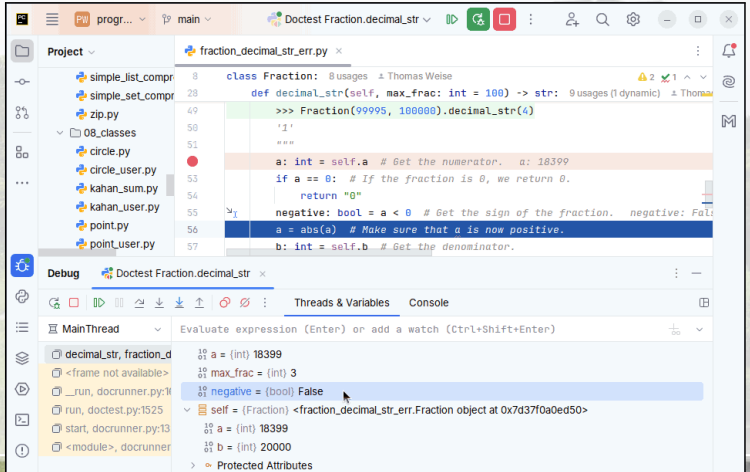
Debuggen in PyCharm

- Die nächste Kodezeile nach dem `if` ist nun markiert.
- Wir führen sie mit F8 aus.



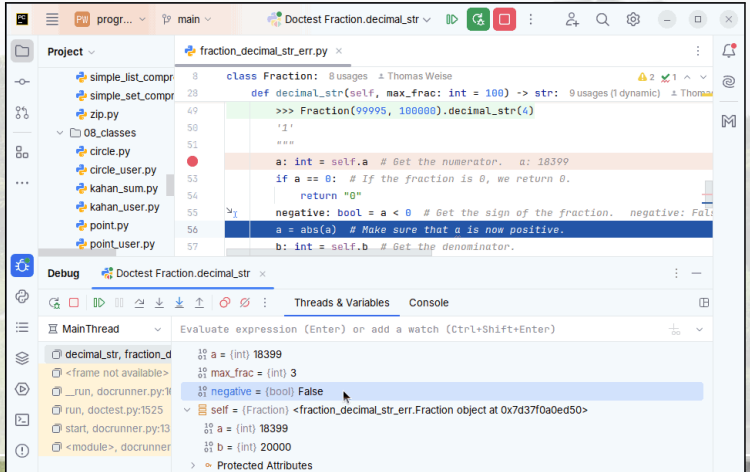
Debuggen in PyCharm

- Die lokale Variable `negative` wird erzeugt.



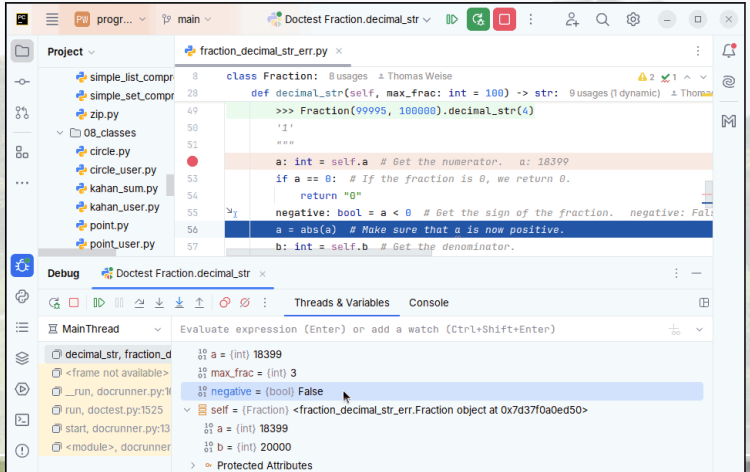
Debuggen in PyCharm

- Die lokale Variable `negative` wird erzeugt.
- Da `a < 0` nämlich `False` ist, ist `negative` ebenfalls `False`.



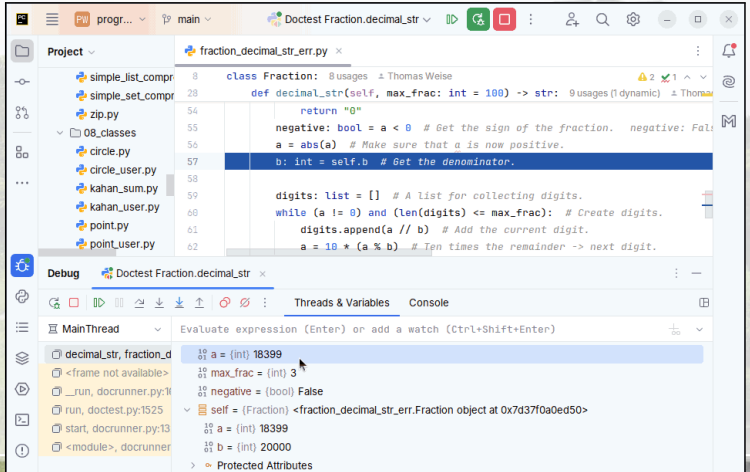
Debuggen in PyCharm

- Da `a < 0` nämlich `False` ist, ist `negative` ebenfalls `False`.
- Die nächste Kodezeile ist markiert und wir führen sie mit F8 aus.



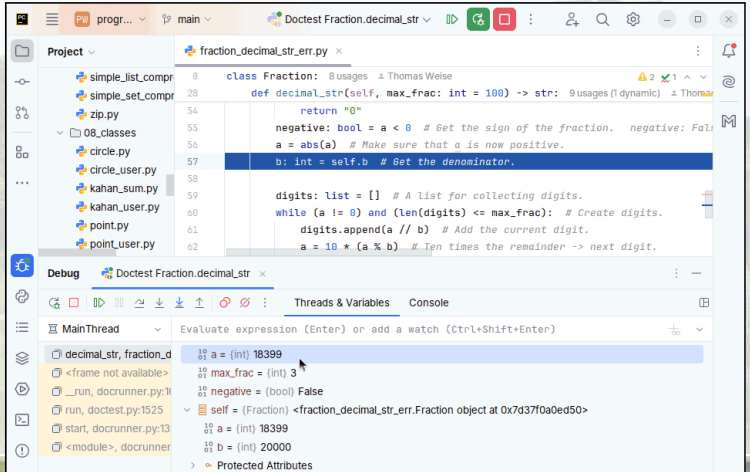
Debuggen in PyCharm

- `a = abs(a)` hat keinen Effekt, denn `a` ist ja schon positiv.



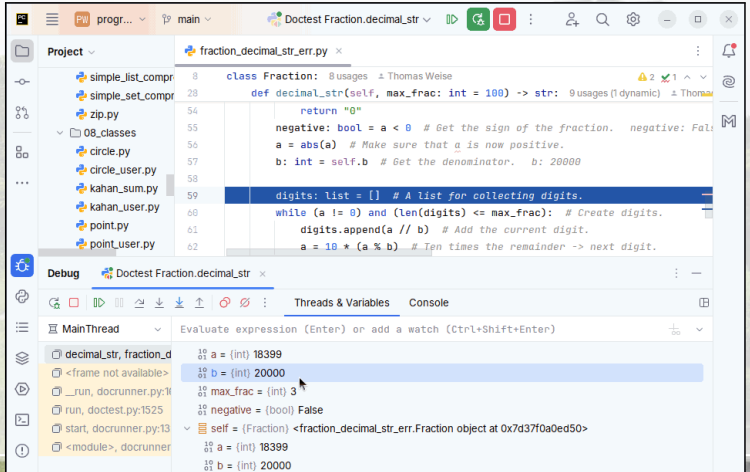
Debuggen in PyCharm

- `a = abs(a)` hat keinen Effekt, denn `a` ist ja schon positiv.
- Wir drücken F8 um weiterzumachen.



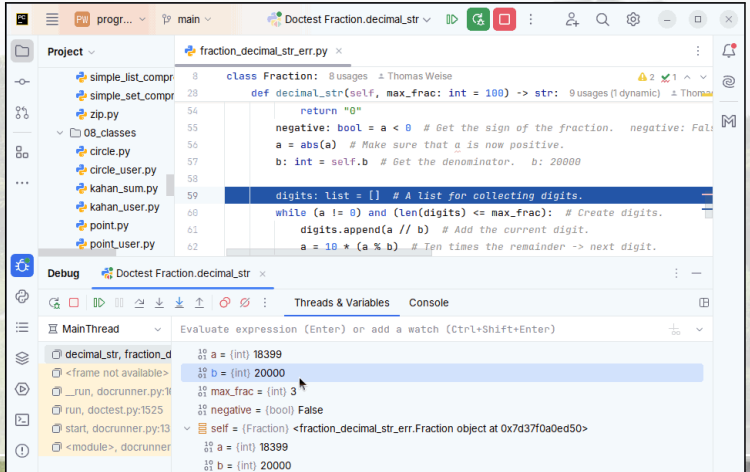
Debuggen in PyCharm

- Das führt `b = self.b` aus.



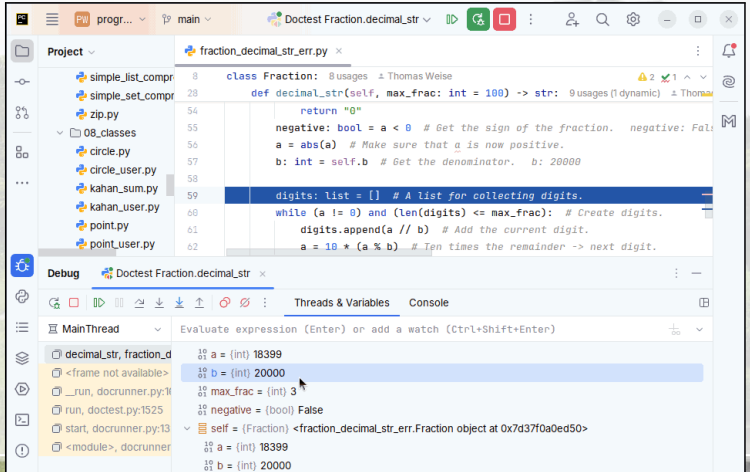
Debuggen in PyCharm

- Das führt `b = self.b` aus.
- Eine neue lokale Variable `b` mit Wert `20000` wird erstellt.



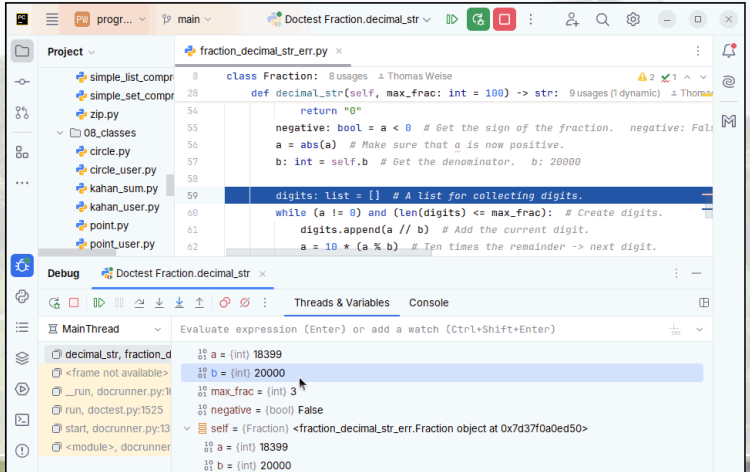
Debuggen in PyCharm

- Eine neue lokale Variable `b` mit Wert `20000` wird erstellt.
- Wir sind jetzt an der letzten Zeile des „trivialen Setups“ unserer Methode `decimal_str`, dem Erstellen der Liste `digits`.



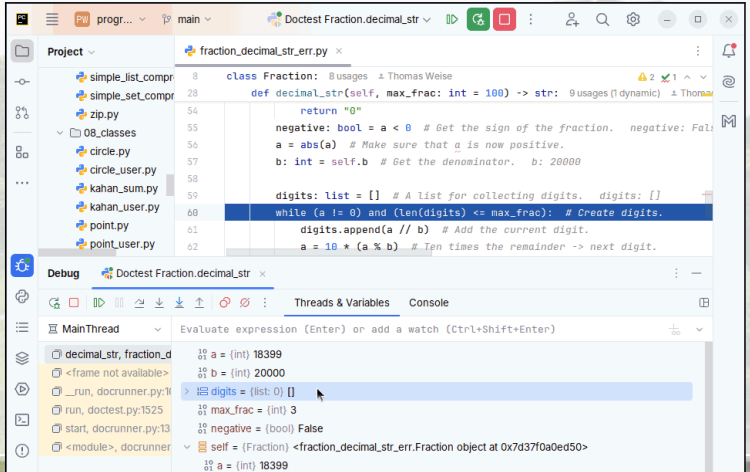
Debuggen in PyCharm

- Wir sind jetzt an der letzten Zeile des „trivialen Setups“ unserer Methode `decimal_str`, dem Erstellen der Liste `digits`.
- Wir drücken F8.



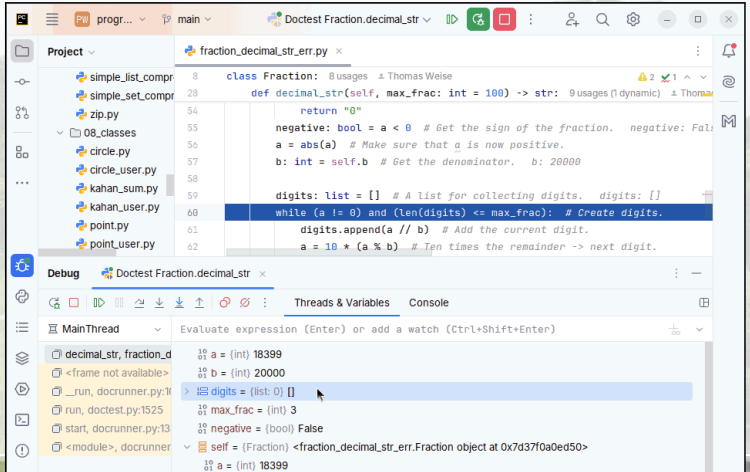
Debuggen in PyCharm

- Die neue Variable `digits` ist wirklich aufgetaucht.



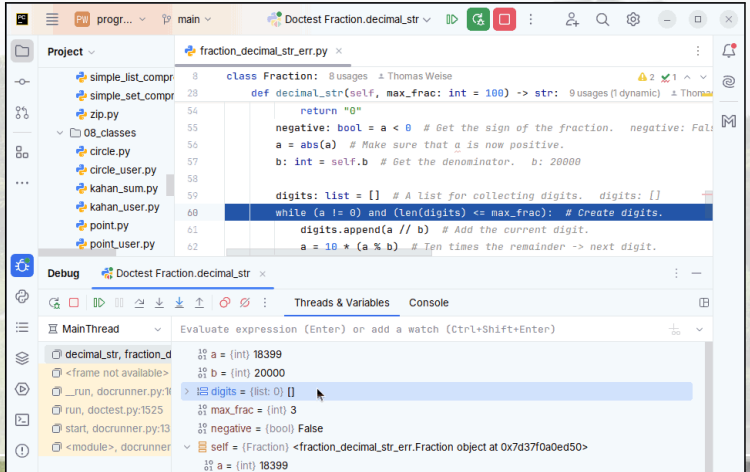
Debuggen in PyCharm

- Die neue Variable `digits` ist wirklich aufgetaucht.
- Sie ist eine leere Liste `[]`.



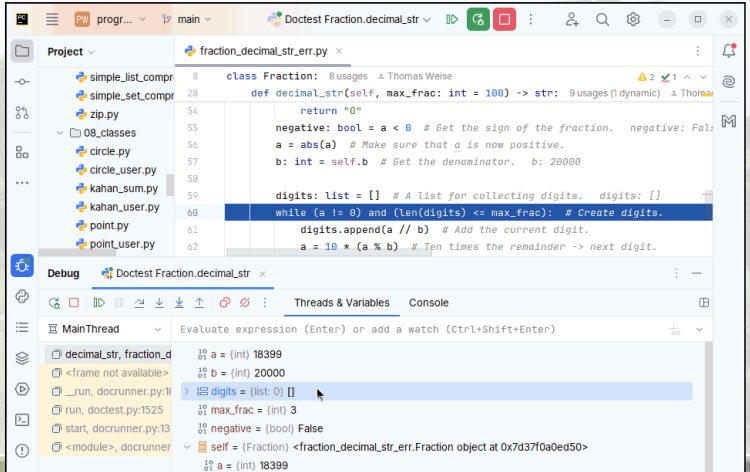
Debuggen in PyCharm

- Sie ist eine leere Liste `[]`.
- Wir sind nun am Anfang der `while`-Schleife.



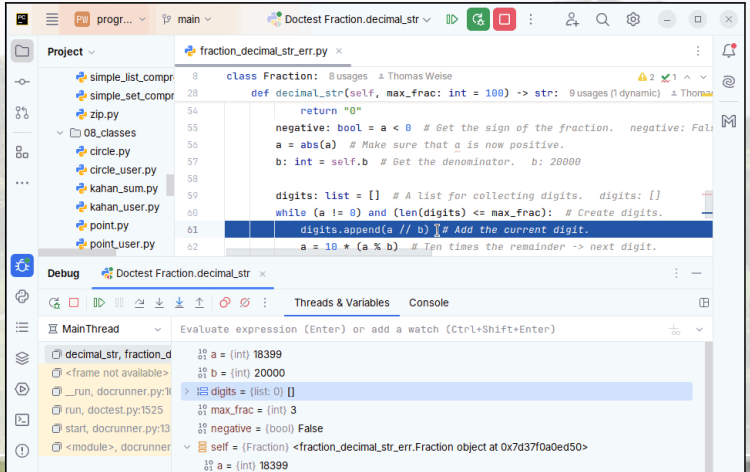
Debuggen in PyCharm

- Wir sind nun am Anfang der `while`-Schleife.
- Wir drücken F8, wodurch die Bedingung am Anfang der Schleife geprüft wird.



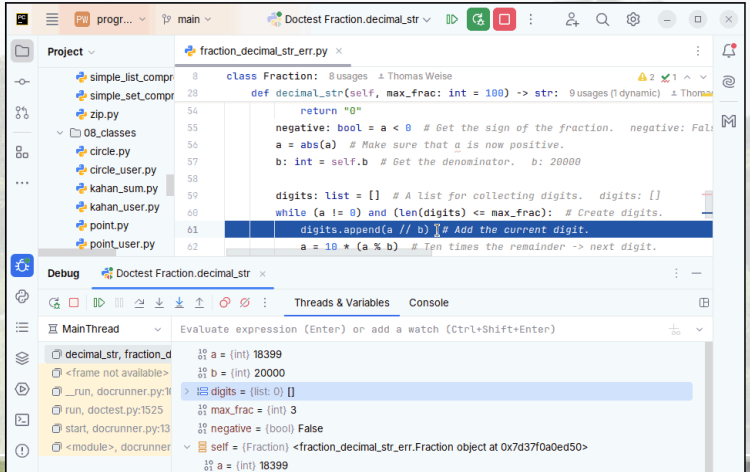
Debuggen in PyCharm

- Wir sehen, dass nun die erste Zeile des Schleifenkörpers markiert ist.



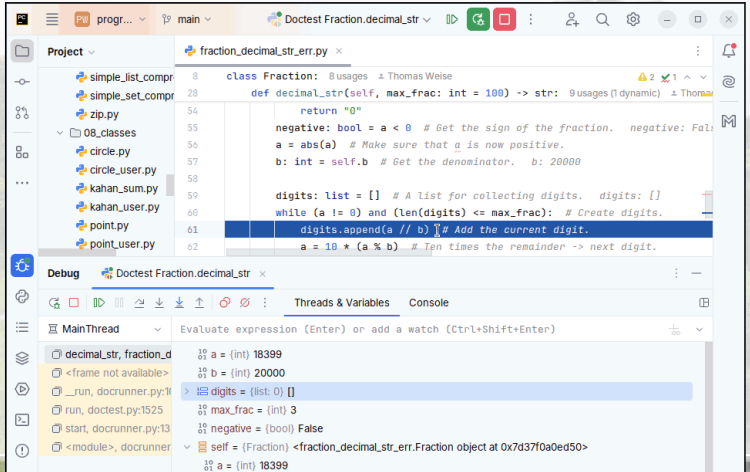
Debuggen in PyCharm

- Wir sehen, dass nun die erste Zeile des Schleifenkörpers markiert ist.
- Das bedeutet, dass `a != 0` und `len(digits)<= max_frac` beide `True` sind.



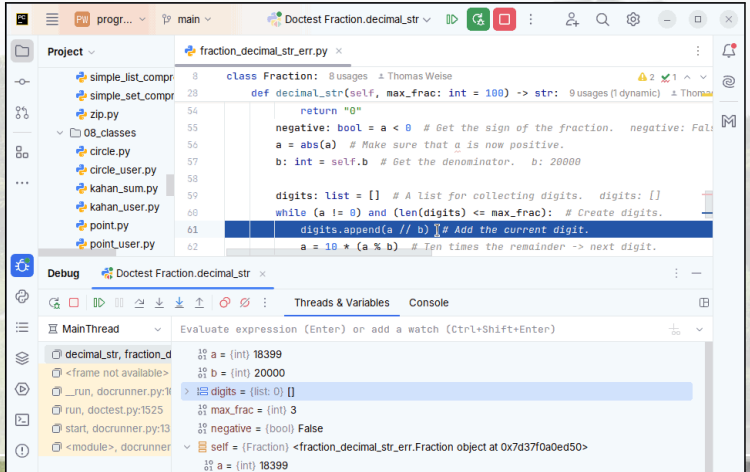
Debuggen in PyCharm

- Und das sollten sie auch sein, denn `a` ist `18399`, `len(digits)` ist 0 und `max_frac` ist 3.



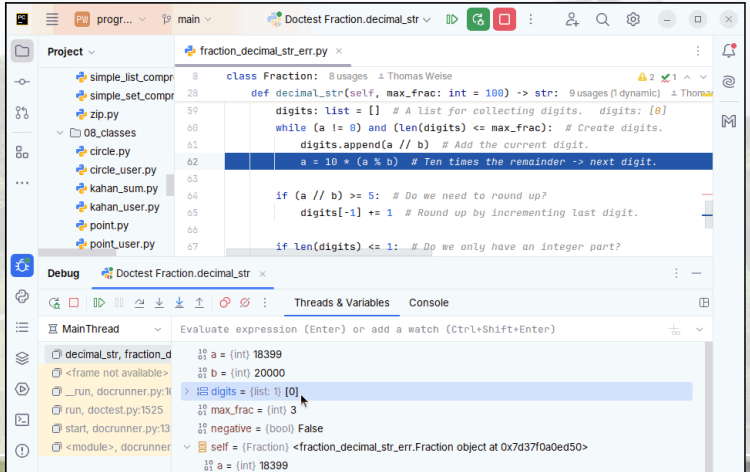
Debuggen in PyCharm

- Wir drücken den ⭳-Knopf, um die erste Zeile des Schleifenkörpers auszuführen.



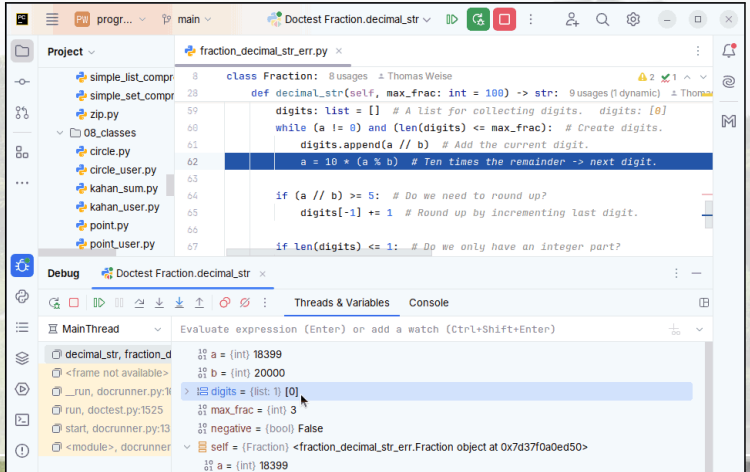
Debuggen in PyCharm

- `digits.append(a // b)` wird nun den Wert `18399 // 20000` an die Liste `digits` anhängen.



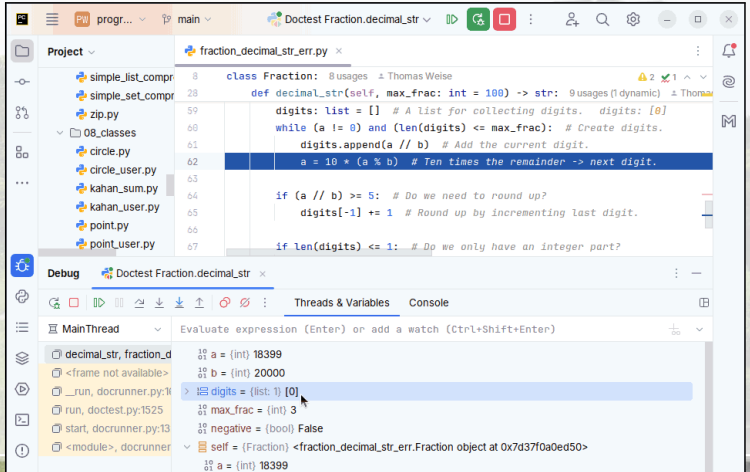
Debuggen in PyCharm

- Weil das das Ergebnis einer Ganzzahldivision ist, bei der der Nenner größer als der Zähler ist, ist `digits` nun `[0]`.



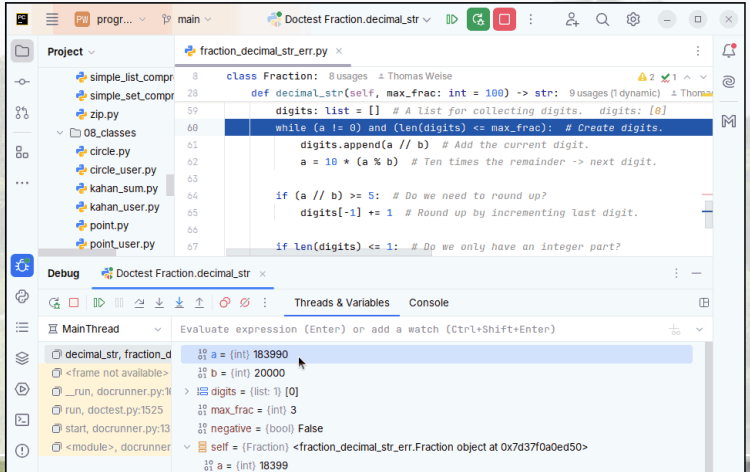
Debuggen in PyCharm

- Weil das das Ergebnis einer Ganzzahldivision ist, bei der der Nenner größer als der Zähler ist, ist `digits` nun `[0]`.
- Wir drücken F8 um weiterzumachen.



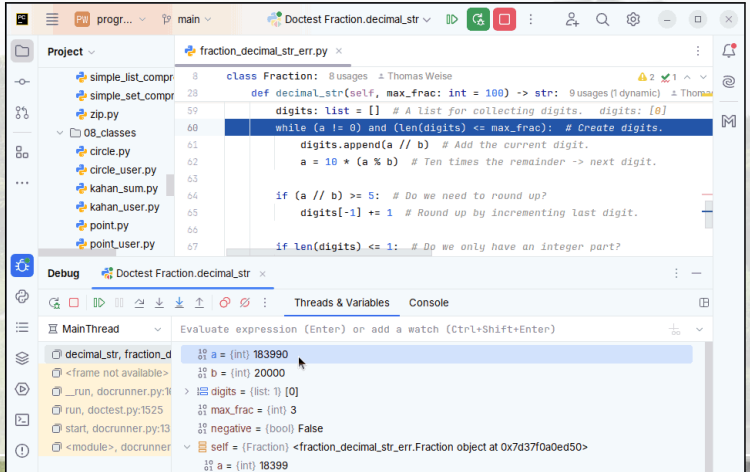
Debuggen in PyCharm

- Nun wird `a = 10 * (a %b)` ausgeführt.



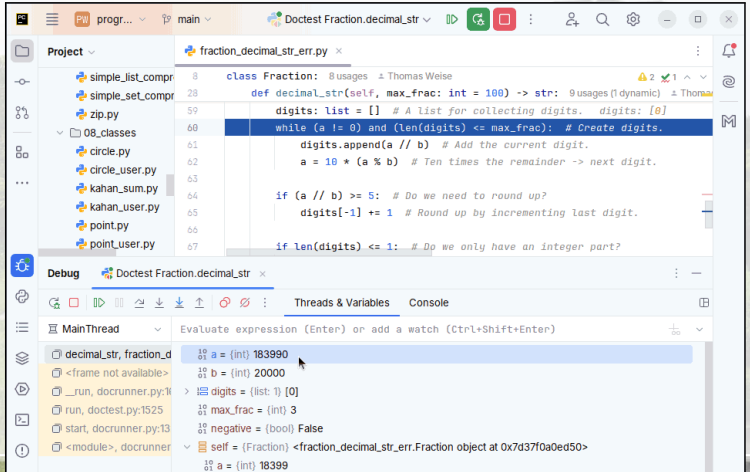
Debuggen in PyCharm

- Nun wird `a = 10 * (a %b)` ausgeführt.
- Weil `18399 %20000` immer noch `18399` ist, wird `a` nun `183990`.



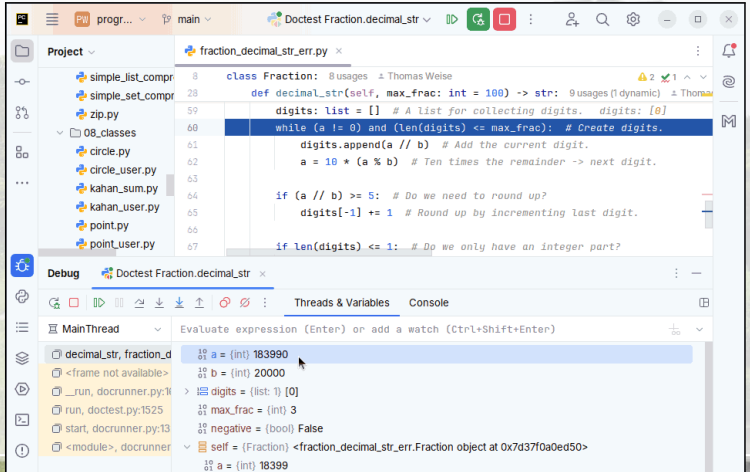
Debuggen in PyCharm

- Weil `18399 %20000` immer noch `18399` ist, wird `a` nun `183990`.
- Nun wird der Kopf der Schleifen mit deren Bedingung wirder markiert.



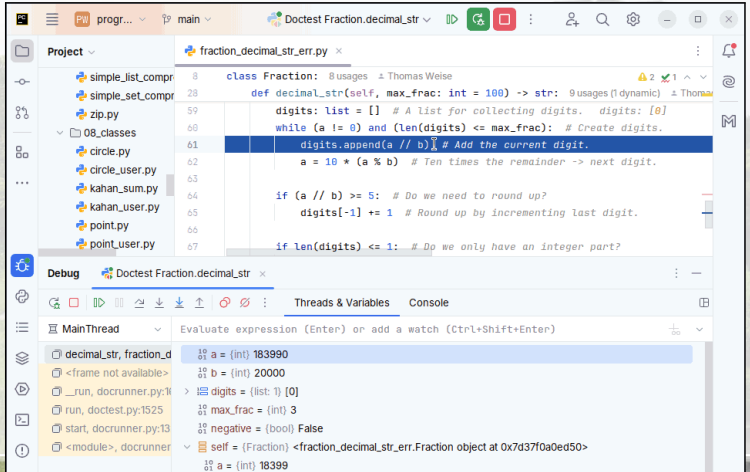
Debuggen in PyCharm

- Nun wird der Kopf der Schleifen mit deren Bedingung wirder markiert.
- Wir drücken ⇲ um ihn auszuführen.



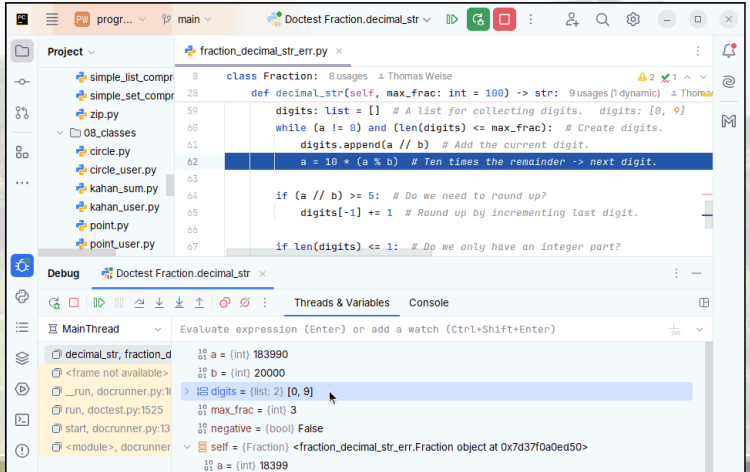
Debuggen in PyCharm

- Die Schleifenbedingung wird immer noch erfüllt, also ist nun wieder die erste Zeile der Schleife markiert.



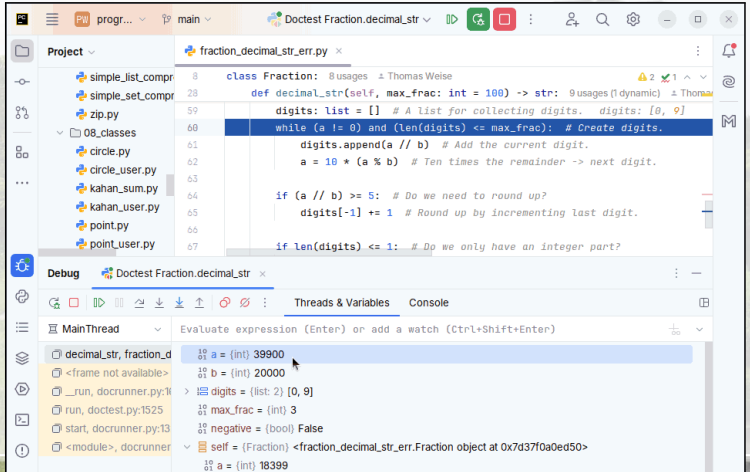
Debuggen in PyCharm

- Nachdem wir F8 drücken, wird `9` an die Liste `digits` angehängt.



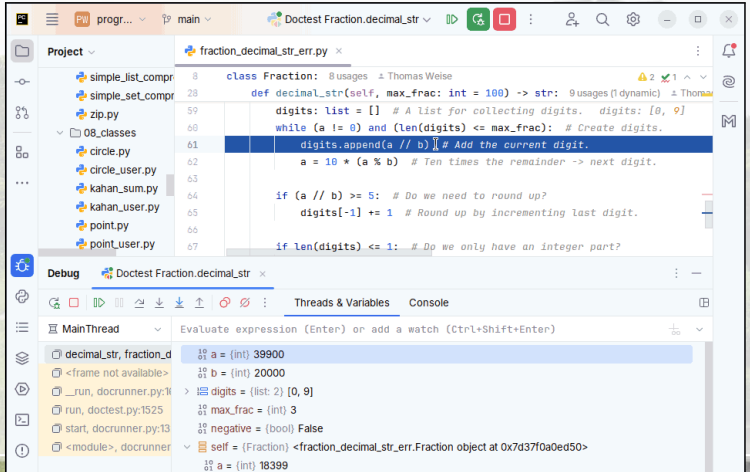
Debuggen in PyCharm

- Nun wird `a` auf `39900` gesetzt.



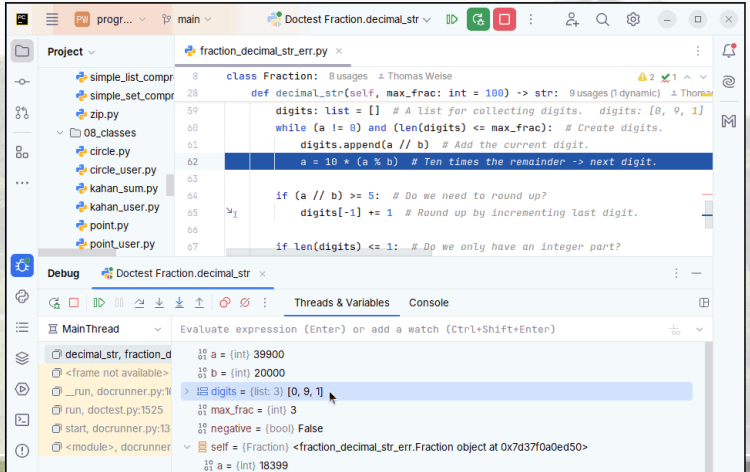
Debuggen in PyCharm

- Wir kommen zur nächsten Iteration der Schleife.



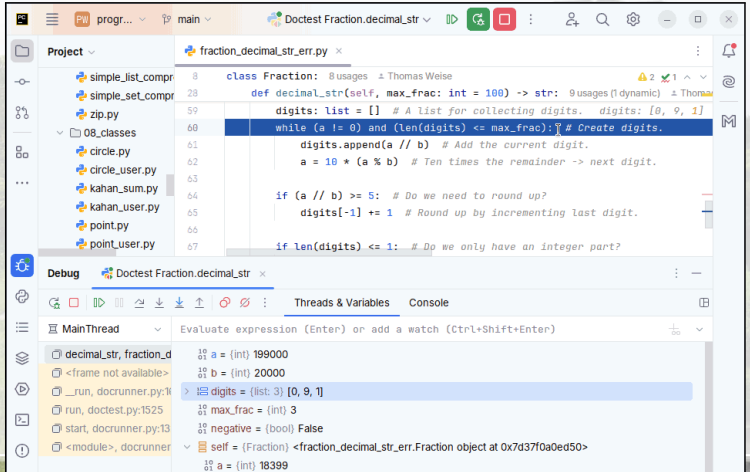
Debuggen in PyCharm

- Nun wird `1` an die Liste `digits` angehängt.



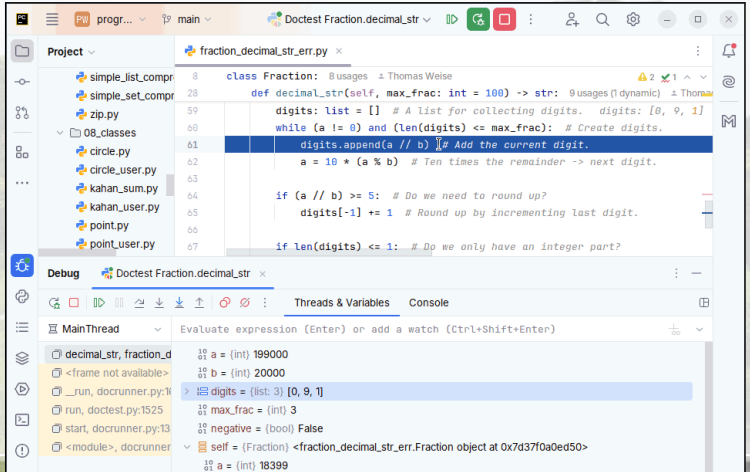
Debuggen in PyCharm

- Dann wird `a` auf `199000` upgdated.



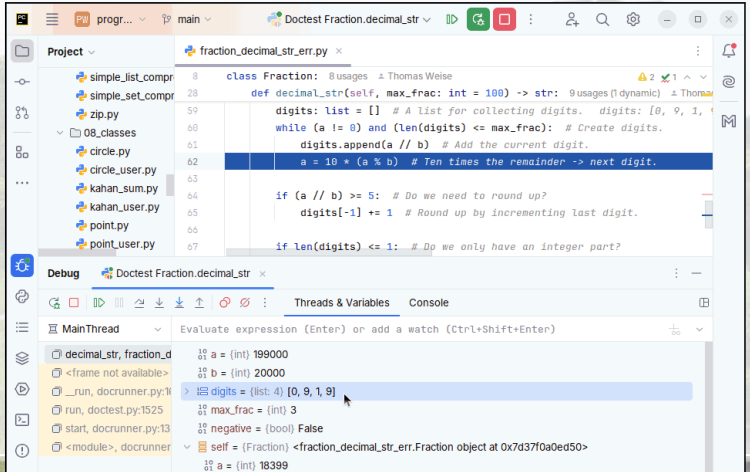
Debuggen in PyCharm

- Ein neuer Schleifendurchlauf begint.



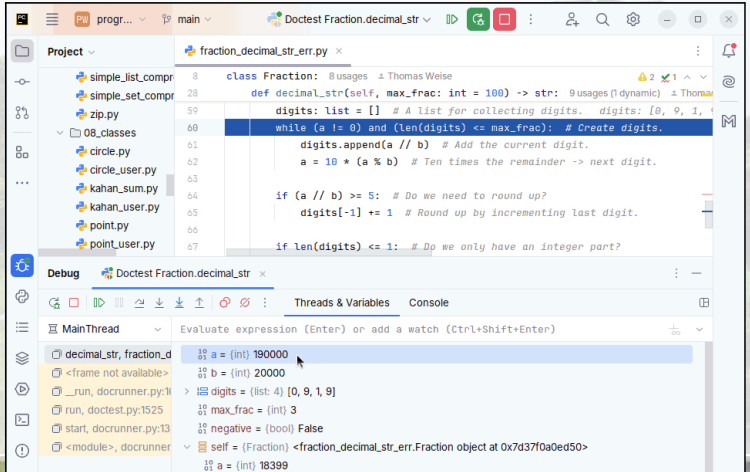
Debuggen in PyCharm

- Nun wird `9` an `digits` angehängt.



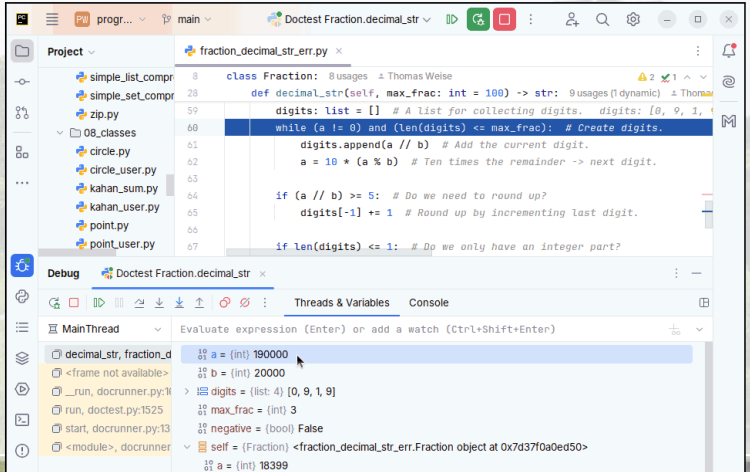
Debuggen in PyCharm

- Danach wird `a` auf `190000` gesetzt.



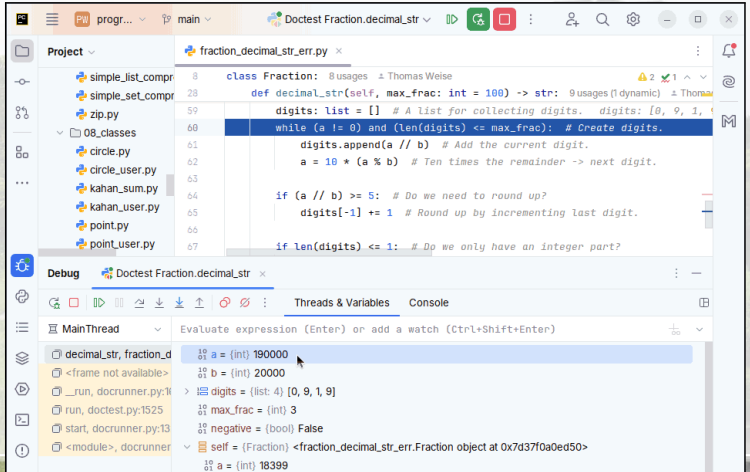
Debuggen in PyCharm

- Danach wird `a` auf `190000` gesetzt.
- Bisher sieht alles gut aus.



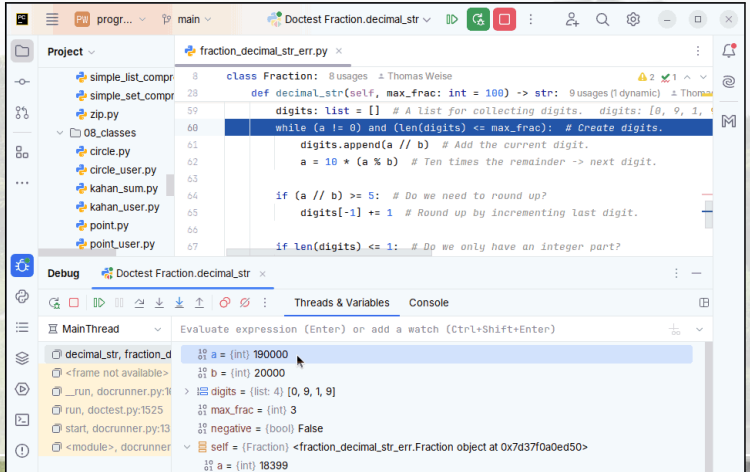
Debuggen in PyCharm

- Bisher sieht alles gut aus.
- Nun ist `digits` zu `[0, 9, 1, 9]` geworden.



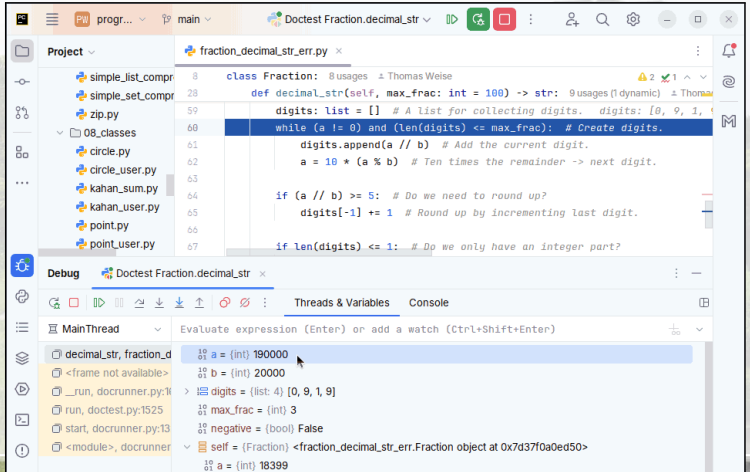
Debuggen in PyCharm

- Nun ist `digits` zu `[0, 9, 1, 9]` geworden.
- Weil `max_frac` gleich `3` ist, trifft `len(digits)<= max_frac` nicht mehr zu.



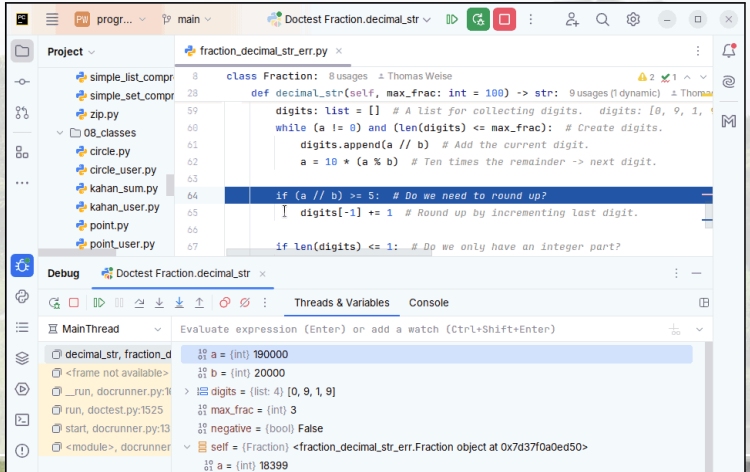
Debuggen in PyCharm

- Weil `max_frac` gleich `3` ist, trifft `len(digits)<= max_frac` nicht mehr zu.
- Wir sind wieder am Kopf der Schleife.



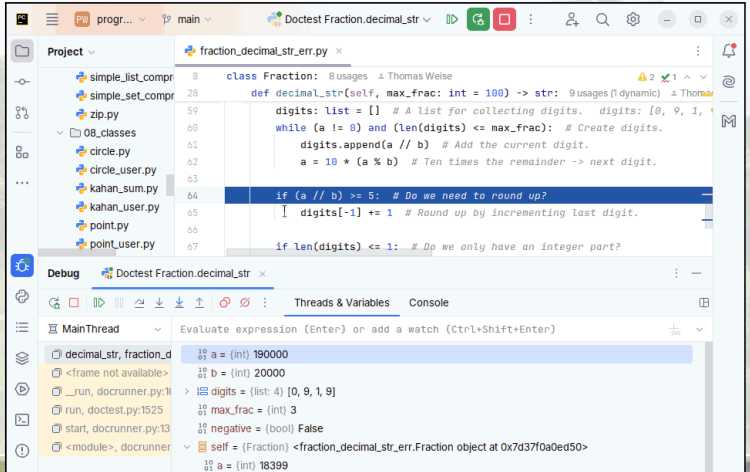
Debuggen in PyCharm

- Wir sind wieder am Kopf der Schleife.
- Wenn wir nun F8 drücken, wird die Schleifenbedingung wieder ausgewertet.



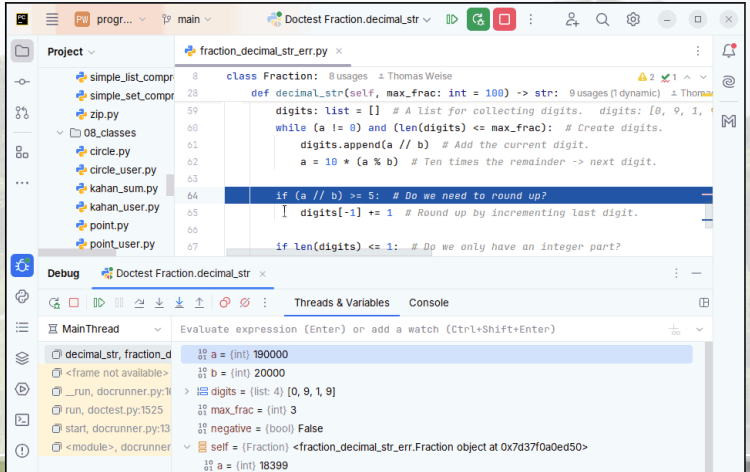
Debuggen in PyCharm

- Wenn wir nun F8 drücken, wird die Schleifenbedingung wieder ausgewertet.
- Dieses Mal ist sie aber `False`.



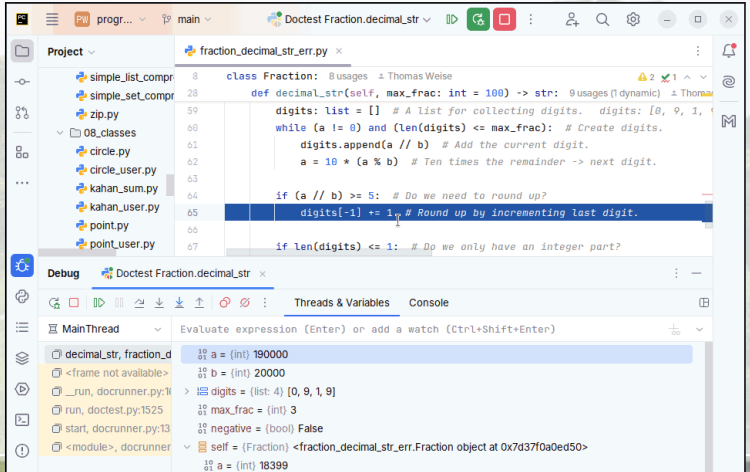
Debuggen in PyCharm

- Dieses Mal ist sie aber `False`.
- Die Schleife terminiert und die nächste Zeile Kode danach wird markiert.



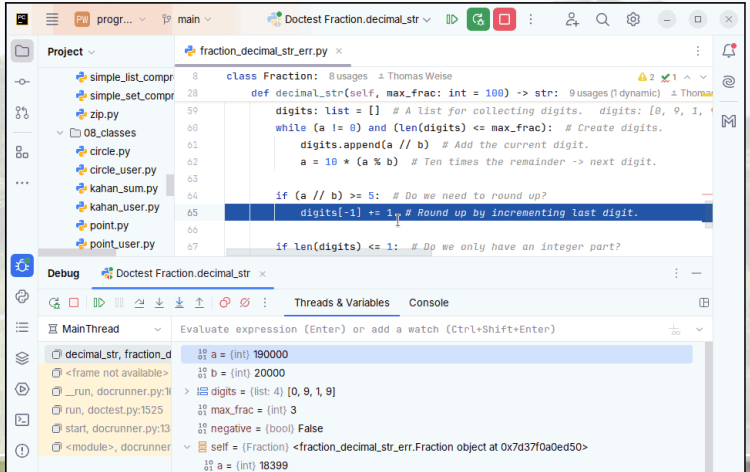
Debuggen in PyCharm

- Wenn wir uns anschauen, was wir bisher berechnet haben, dann stimmt alles.



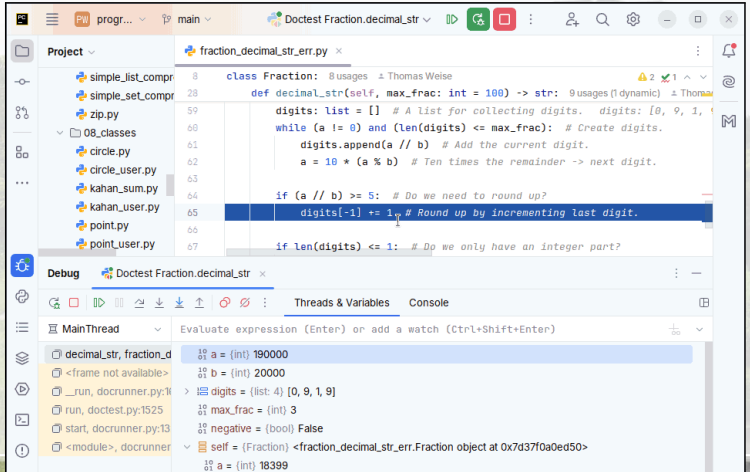
Debuggen in PyCharm

- Wenn wir uns anschauen, was wir bisher berechnet haben, dann stimmt alles.
- Wir wollen den Bruch $\frac{91995}{100000}$ zu einer Dezimal mit drei Nachkommastellen umrechnen.



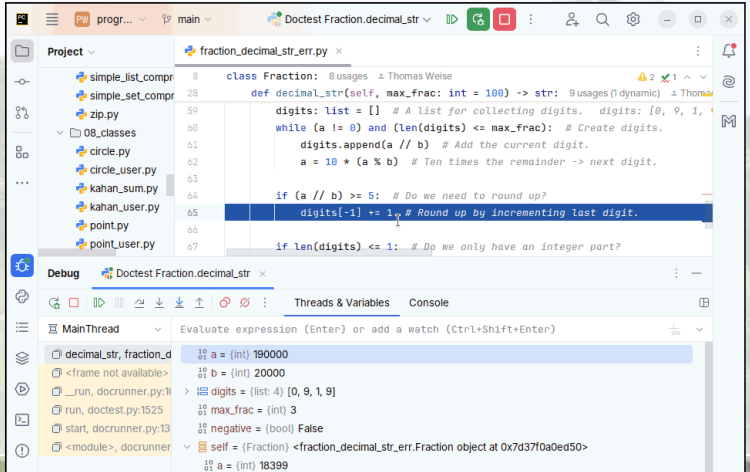
Debuggen in PyCharm

- Wir wollen den Bruch $\frac{91995}{100000}$ zu einer Dezimal mit drei Nachkommastellen umrechnen.
- Bisher haben wir die Ziffern 0, 9, 1, und 9.



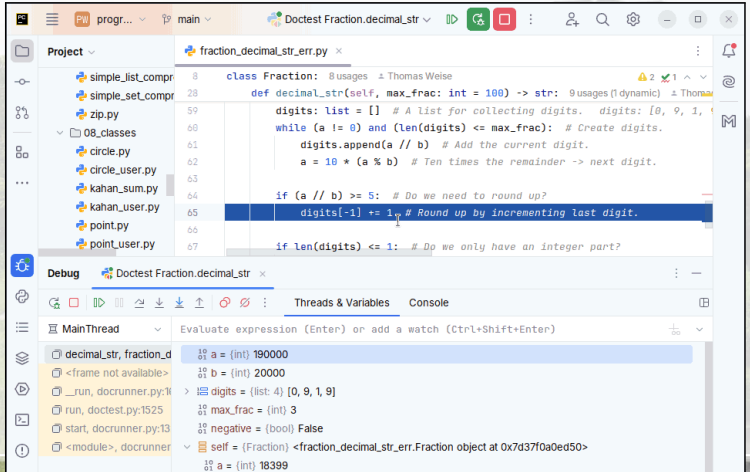
Debuggen in PyCharm

- Bisher haben wir die Ziffern 0, 9, 1, und 9.
- Die nächste Kodezeile, `if (a // b)>= 5`, soll prüfen ob wir die letzte Ziffer aufrunden müssen.



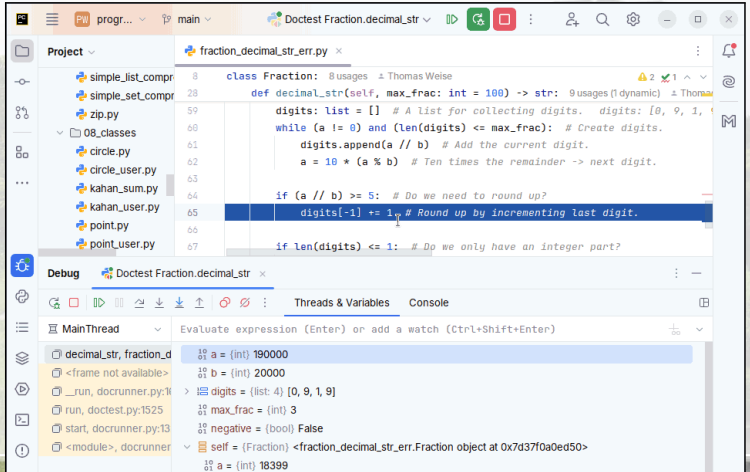
Debuggen in PyCharm

- Die nächste Kodezeile, `if (a // b)>= 5`, soll prüfen ob wir die letzte Ziffer aufrunden müssen.
- Nun, `a` ist `190000` und `b` ist immer noch `20000`, deshalb ist `a // b` gleich `9`.



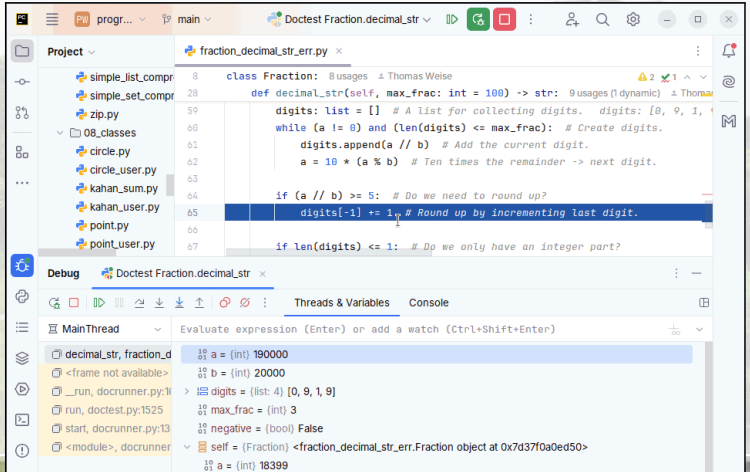
Debuggen in PyCharm

- Nun, `a` ist `190000` und `b` ist immer noch `20000`, deshalb ist `a // b` gleich `9`.
- Die Bedingung sollte also wahr sein.



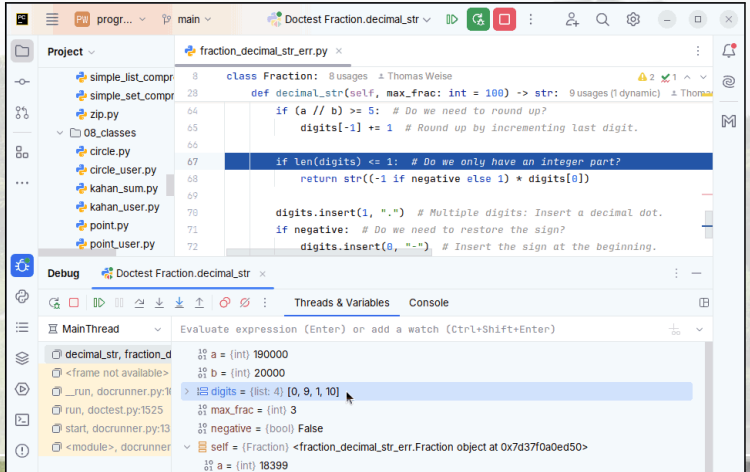
Debuggen in PyCharm

- Die Bedingung sollte also wahr sein.
- Wir drücken ⭲ um das zu prüfen.



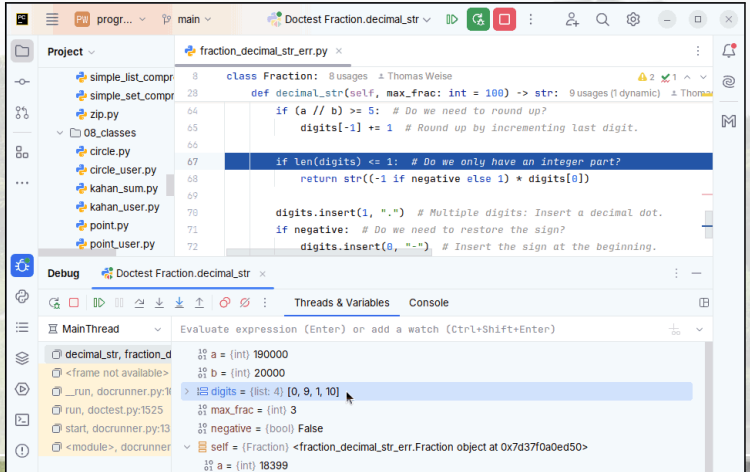
Debuggen in PyCharm

- Jetzt sehen wir den Bug, also den Fehler, in unserem Kode.



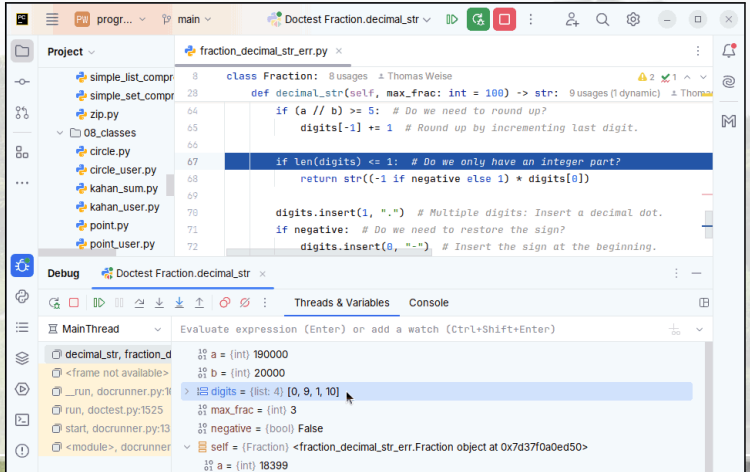
Debuggen in PyCharm

- Jetzt sehen wir den Bug, also den Fehler, in unserem Kode.
- Um aufzurunden, haben wir die letzte Ziffer in unserer Liste `digits` um 1 erhöht.



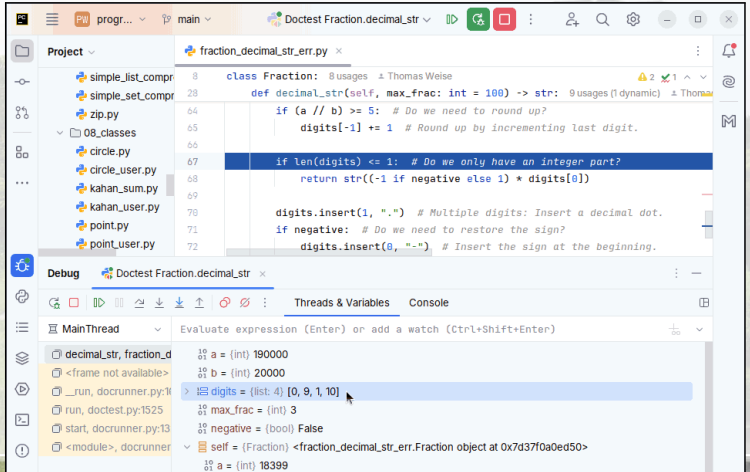
Debuggen in PyCharm

- Um aufzurunden, haben wir die letzte Ziffer in unserer Liste `digits` um 1 erhöht.
- `digits` war `[0, 9, 1, 9]`.



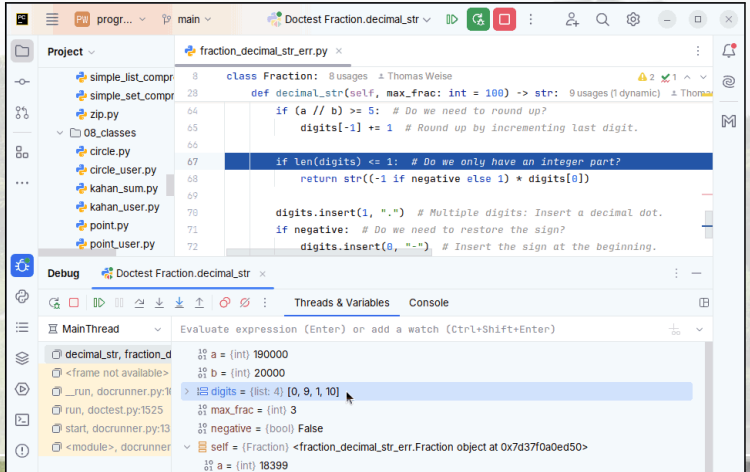
Debuggen in PyCharm

- `digits` war `[0, 9, 1, 9]`.
- Deshalb ist es jetzt `[0, 9, 1, 10]`.



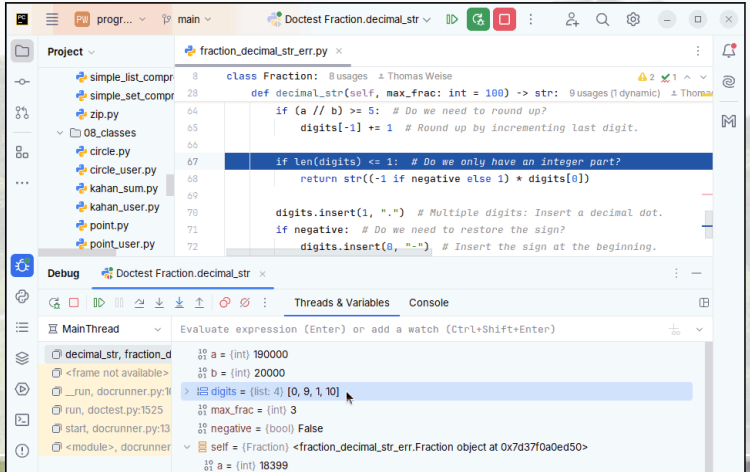
Debuggen in PyCharm

- Deshalb ist es jetzt `[0, 9, 1, 10]`.
- Die komische 0 hinten in der Ausgabe war keine einzelne Ziffer.



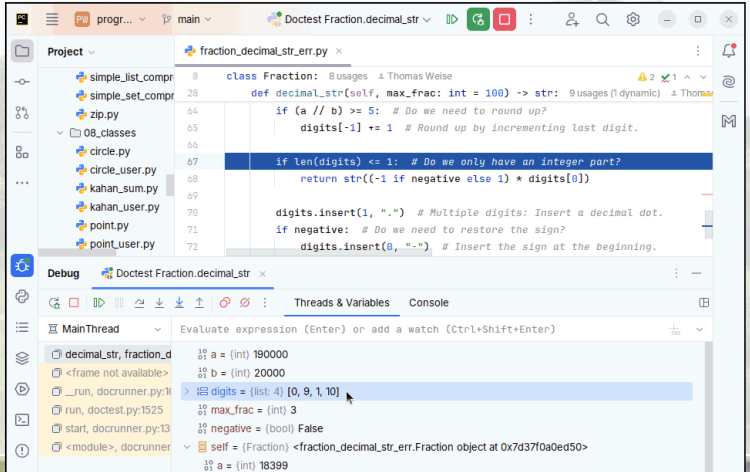
Debuggen in PyCharm

- Die komische 0 hinten in der Ausgabe war keine einzelne Ziffer.
- Es war die hintere 0 einer Zehn.



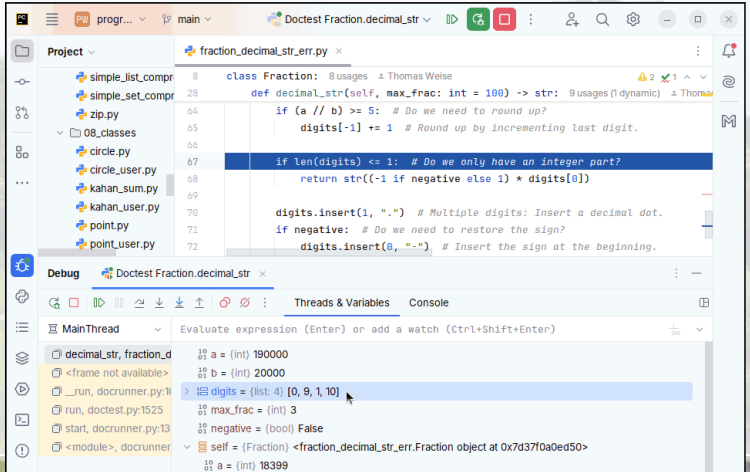
Debuggen in PyCharm

- Es war die hintere 0 einer Zehn.
- Wir haben nicht bedacht, dass wir nicht nur in einfachen Fällen aufrunden.



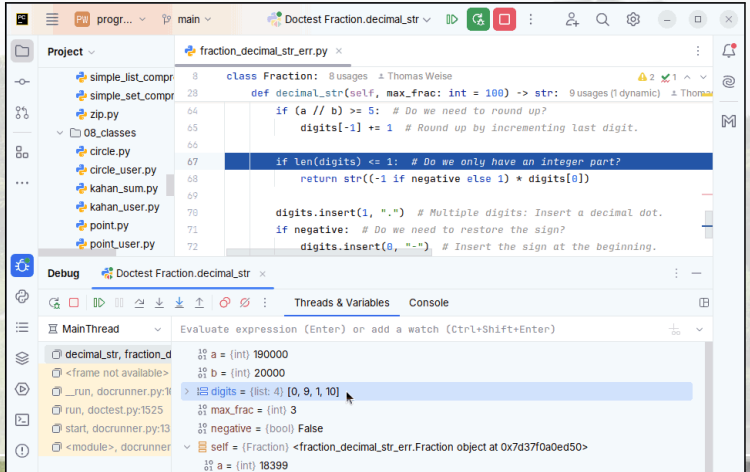
Debuggen in PyCharm

- Wir haben nicht bedacht, dass wir nicht nur in einfachen Fällen aufrunden.
- Klar, das Aufrunden von 1.25 ergibt 1.3 und wir müssen nur eine Ziffer erhöhen.



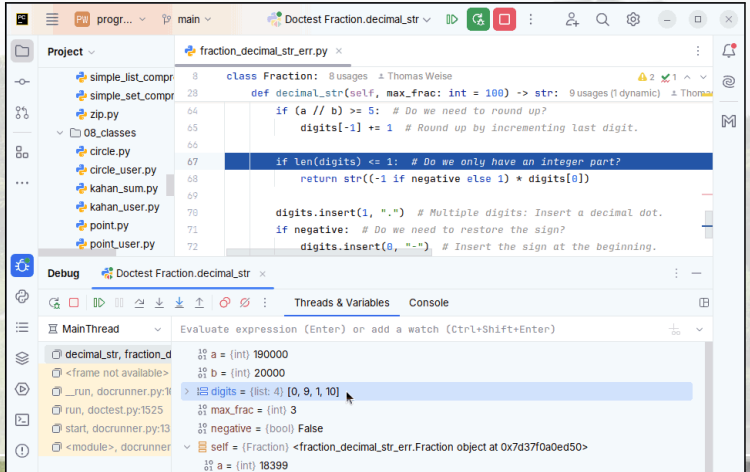
Debuggen in PyCharm

- Klar, das Aufrunden von 1.25 ergibt 1.3 und wir müssen nur eine Ziffer erhöhen.
- Es könnte aber auch Fälle wie 0.9999, geben, wo wir auf 1 runden müssen, selbst wenn wir drei Nachkommastellen Genauigkeit haben wollen.



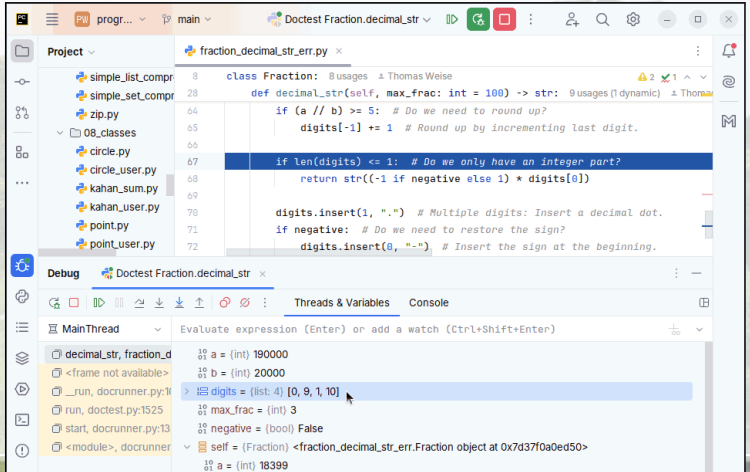
Debuggen in PyCharm

- Es könnte aber auch Fälle wie 0.9999, geben, wo wir auf 1 runden müssen, selbst wenn wir drei Nachkommastellen Genauigkeit haben wollen.
- Unser Kode macht das nicht.



Debuggen in PyCharm

- Unser Kode macht das nicht.
- Wir können hier aufhören, zu debuggen, und zurück zu unserem Kode gehen.



Reparierte Methode



Fraction: decimal_str

- Wir implementieren unsere Methode `decimal_str` jetzt korrekt in .

```
>>> Fraction(91995, 100000).decimal_str(3)
'0.92'
>>> Fraction(99995, 100000).decimal_str(4)
'1'
"""
a: int = self.a  # Get the numerator.
if a == 0:  # If the fraction is 0, we return 0.
    return "0"
negative: Final[bool] = a < 0  # Get the sign of the fraction.
a = abs(a)  # Make sure that `a` is now positive.
b: Final[int] = self.b  # Get the denominator.

digits: Final[list] = []  # A list for collecting digits.
while (a != 0) and (len(digits) <= max_frac):  # Create digits.
    digits.append(a // b)  # Add the current digit.
    a = 10 * (a % b)  # Ten times the remainder -> next digit.

if (a // b) >= 5:  # Do we need to round up?
    # This may lead to other digits topple over, e.g., 0.999...
    for i in range(len(digits) - 1, 0, -1):  # except first!
        digits[i] += 1  # Increment the digit at position i.
        if digits[i] != 10:  # Was there no overflow?
            break  # Digits in 1..9, no overflow, we can stop.
        digits[i] = 0  # We got a `10`, so we set it to 0.
    else:  # This is only reached if no `break` was done.
        digits[0] += 1  # Increment the integer part.

while digits[-1] == 0:  # Remove all trailing zeros.
    del digits[-1]  # Delete the trailing zero.

if len(digits) <= 1:  # Do we only have an integer part?
    return str((-1 if negative else 1) * digits[0])

digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
if negative:  # Do we need to restore the sign?
    digits.insert(0, "-")  # Insert the sign at the beginning.
return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- Wir implementieren unsere Methode `decimal_str` jetzt korrekt in .
- Unser Kode zum Aufrunden wird komplexer.

```
>>> Fraction(91995, 100000).decimal_str(3)
'0.92'
>>> Fraction(99995, 100000).decimal_str(4)
'1'
"""
a: int = self.a  # Get the numerator.
if a == 0:  # If the fraction is 0, we return 0.
    return "0"
negative: Final[bool] = a < 0  # Get the sign of the fraction.
a = abs(a)  # Make sure that `a` is now positive.
b: Final[int] = self.b  # Get the denominator.

digits: Final[list] = []  # A list for collecting digits.
while (a != 0) and (len(digits) <= max_frac):  # Create digits.
    digits.append(a // b)  # Add the current digit.
    a = 10 * (a % b)  # Ten times the remainder -> next digit.

if (a // b) >= 5:  # Do we need to round up?
    # This may lead to other digits topple over, e.g., 0.999...
    for i in range(len(digits) - 1, 0, -1):  # except first!
        digits[i] += 1  # Increment the digit at position i.
        if digits[i] != 10:  # Was there no overflow?
            break  # Digits in 1..9, no overflow, we can stop.
        digits[i] = 0  # We got a `10`, so we set it to 0.
    else:  # This is only reached if no `break` was done.
        digits[0] += 1  # Increment the integer part.

while digits[-1] == 0:  # Remove all trailing zeros.
    del digits[-1]  # Delete the trailing zero.

if len(digits) <= 1:  # Do we only have an integer part?
    return str((-1 if negative else 1) * digits[0])

digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
if negative:  # Do we need to restore the sign?
    digits.insert(0, "-")  # Insert the sign at the beginning.
return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- Wir implementieren unsere Methode `decimal_str` jetzt korrekt in .
- Unser Kode zum Aufrunden wird komplexer.
- Zuerst müssen wir über alle Nachkommastellen von hinten nach vorne iterieren.

```
        >>> Fraction(91995, 100000).decimal_str(3)
        '0.92'
        >>> Fraction(99995, 100000).decimal_str(4)
        '1'
        """
        a: int = self.a  # Get the numerator.
        if a == 0:  # If the fraction is 0, we return 0.
            return "0"
        negative: Final[bool] = a < 0  # Get the sign of the fraction.
        a = abs(a)  # Make sure that `a` is now positive.
        b: Final[int] = self.b  # Get the denominator.

        digits: Final[list] = []  # A list for collecting digits.
        while (a != 0) and (len(digits) <= max_frac):  # Create digits.
            digits.append(a // b)  # Add the current digit.
            a = 10 * (a % b)  # Ten times the remainder -> next digit.

        if (a // b) >= 5:  # Do we need to round up?
            # This may lead to other digits topple over, e.g., 0.999...
            for i in range(len(digits) - 1, 0, -1):  # except first!
                digits[i] += 1  # Increment the digit at position i.
                if digits[i] != 10:  # Was there no overflow?
                    break  # Digits in 1..9, no overflow, we can stop.
                digits[i] = 0  # We got a `10`, so we set it to 0.
            else:  # This is only reached if no `break` was done.
                digits[0] += 1  # Increment the integer part.

        while digits[-1] == 0:  # Remove all trailing zeros.
            del digits[-1]  # Delete the trailing zero.

        if len(digits) <= 1:  # Do we only have an integer part?
            return str((-1 if negative else 1) * digits[0])

        digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
        if negative:  # Do we need to restore the sign?
            digits.insert(0, "-")  # Insert the sign at the beginning.
        return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- Wir implementieren unsere Methode `decimal_str` jetzt korrekt in .
- Unser Kode zum Aufrunden wird komplexer.
- Zuerst müssen wir über alle Nachkommastellen von hinten nach vorne iterieren.
- Das geht mit `for i in range(len(digits) - 1, 0, -1)`.

```
        >>> Fraction(91995, 100000).decimal_str(3)
        '0.92'
        >>> Fraction(99995, 100000).decimal_str(4)
        '1'
        """
        a: int = self.a  # Get the numerator.
        if a == 0:  # If the fraction is 0, we return 0.
            return "0"
        negative: Final[bool] = a < 0  # Get the sign of the fraction.
        a = abs(a)  # Make sure that `a` is now positive.
        b: Final[int] = self.b  # Get the denominator.

        digits: Final[list] = []  # A list for collecting digits.
        while (a != 0) and (len(digits) <= max_frac):  # Create digits.
            digits.append(a // b)  # Add the current digit.
            a = 10 * (a % b)  # Ten times the remainder -> next digit.

        if (a // b) >= 5:  # Do we need to round up?
            # This may lead to other digits topple over, e.g., 0.999...
            for i in range(len(digits) - 1, 0, -1):  # except first!
                digits[i] += 1  # Increment the digit at position i.
                if digits[i] != 10:  # Was there no overflow?
                    break  # Digits in 1..9, no overflow, we can stop.
                digits[i] = 0  # We got a `10`, so we set it to 0.
            else:  # This is only reached if no `break` was done.
                digits[0] += 1  # Increment the integer part.

        while digits[-1] == 0:  # Remove all trailing zeros.
            del digits[-1]  # Delete the trailing zero.

        if len(digits) <= 1:  # Do we only have an integer part?
            return str((-1 if negative else 1) * digits[0])

        digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
        if negative:  # Do we need to restore the sign?
            digits.insert(0, "-")  # Insert the sign at the beginning.
        return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- Wir implementieren unsere Methode `decimal_str` jetzt korrekt in .
- Unser Kode zum Aufrunden wird komplexer.
- Zuerst müssen wir über alle Nachkommastellen von hinten nach vorne iterieren.
- Das geht mit `for i in range(len(digits) - 1, 0, -1)`.
- Wenn `len(digits)== 5`, dann iteriert `range(len(digits) - 1, 0, -1)` über die Zahlen 4, 3, 2 und 1.

```
>>> Fraction(91995, 100000).decimal_str(3)
'0.92'
>>> Fraction(99995, 100000).decimal_str(4)
'1'
"""
a: int = self.a  # Get the numerator.
if a == 0:  # If the fraction is 0, we return 0.
    return "0"
negative: Final[bool] = a < 0  # Get the sign of the fraction.
a = abs(a)  # Make sure that `a` is now positive.
b: Final[int] = self.b  # Get the denominator.

digits: Final[list] = []  # A list for collecting digits.
while (a != 0) and (len(digits) <= max_frac):  # Create digits.
    digits.append(a // b)  # Add the current digit.
    a = 10 * (a % b)  # Ten times the remainder -> next digit.

if (a // b) >= 5:  # Do we need to round up?
    # This may lead to other digits topple over, e.g., 0.999...
    for i in range(len(digits) - 1, 0, -1):  # except first!
        digits[i] += 1  # Increment the digit at position i.
        if digits[i] != 10:  # Was there no overflow?
            break  # Digits in 1..9, no overflow, we can stop.
        digits[i] = 0  # We got a `10`, so we set it to 0.
    else:  # This is only reached if no `break` was done.
        digits[0] += 1  # Increment the integer part.

while digits[-1] == 0:  # Remove all trailing zeros.
    del digits[-1]  # Delete the trailing zero.

if len(digits) <= 1:  # Do we only have an integer part?
    return str((-1 if negative else 1) * digits[0])

digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
if negative:  # Do we need to restore the sign?
    digits.insert(0, "-")  # Insert the sign at the beginning.
return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- Unser Kode zum Aufrunden wird komplexer.
- Zuerst müssen wir über alle Nachkommastellen von hinten nach vorne iterieren.
- Das geht mit `for i in range(len(digits) - 1, 0, -1)`.
- Wenn `len(digits)== 5`, dann iteriert `range(len(digits) - 1, 0, -1)` über die Zahlen 4, 3, 2 und 1.
- Wir erhöhen die Ziffer an Index `i` jeweils um 1.

```
        >>> Fraction(91995, 100000).decimal_str(3)
        '0.92'
        >>> Fraction(99995, 100000).decimal_str(4)
        '1'
        """
        a: int = self.a  # Get the numerator.
        if a == 0:  # If the fraction is 0, we return 0.
            return "0"
        negative: Final[bool] = a < 0  # Get the sign of the fraction.
        a = abs(a)  # Make sure that `a` is now positive.
        b: Final[int] = self.b  # Get the denominator.

        digits: Final[list] = []  # A list for collecting digits.
        while (a != 0) and (len(digits) <= max_frac):  # Create digits.
            digits.append(a // b)  # Add the current digit.
            a = 10 * (a % b)  # Ten times the remainder -> next digit.

        if (a // b) >= 5:  # Do we need to round up?
            # This may lead to other digits topple over, e.g., 0.999...
            for i in range(len(digits) - 1, 0, -1):  # except first!
                digits[i] += 1  # Increment the digit at position i.
                if digits[i] != 10:  # Was there no overflow?
                    break  # Digits in 1..9, no overflow, we can stop.
                digits[i] = 0  # We got a `10`, so we set it to 0.
            else:  # This is only reached if no `break` was done.
                digits[0] += 1  # Increment the integer part.

        while digits[-1] == 0:  # Remove all trailing zeros.
            del digits[-1]  # Delete the trailing zero.

        if len(digits) <= 1:  # Do we only have an integer part?
            return str((-1 if negative else 1) * digits[0])

        digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
        if negative:  # Do we need to restore the sign?
            digits.insert(0, "-")  # Insert the sign at the beginning.
        return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- Zuerst müssen wir über alle Nachkommastellen von hinten nach vorne iterieren.
- Das geht mit `for i in range(len(digits) - 1, 0, -1)`.
- Wenn `len(digits)== 5`, dann iteriert `range(len(digits) - 1, 0, -1)` über die Zahlen 4, 3, 2 und 1.
- Wir erhöhen die Ziffer an Index `i` jeweils um 1.
- Wenn das Ergebnis nicht 10 ist, dann können wir die Schleife mit `break` abbrechen.

```
        >>> Fraction(91995, 100000).decimal_str(3)
        '0.92'
        >>> Fraction(99995, 100000).decimal_str(4)
        '1'
        """
        a: int = self.a  # Get the numerator.
        if a == 0:  # If the fraction is 0, we return 0.
            return "0"
        negative: Final[bool] = a < 0  # Get the sign of the fraction.
        a = abs(a)  # Make sure that `a` is now positive.
        b: Final[int] = self.b  # Get the denominator.

        digits: Final[list] = []  # A list for collecting digits.
        while (a != 0) and (len(digits) <= max_frac):  # Create digits.
            digits.append(a // b)  # Add the current digit.
            a = 10 * (a % b)  # Ten times the remainder -> next digit.

        if (a // b) >= 5:  # Do we need to round up?
            # This may lead to other digits topple over, e.g., 0.999...
            for i in range(len(digits) - 1, 0, -1):  # except first!
                digits[i] += 1  # Increment the digit at position i.
                if digits[i] != 10:  # Was there no overflow?
                    break  # Digits in 1..9, no overflow, we can stop.
                digits[i] = 0  # We got a `10`, so we set it to 0.
            else:  # This is only reached if no `break` was done.
                digits[0] += 1  # Increment the integer part.

        while digits[-1] == 0:  # Remove all trailing zeros.
            del digits[-1]  # Delete the trailing zero.

        if len(digits) <= 1:  # Do we only have an integer part?
            return str((-1 if negative else 1) * digits[0])

        digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
        if negative:  # Do we need to restore the sign?
            digits.insert(0, "-")  # Insert the sign at the beginning.
        return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- Das geht mit `for i in range(len(digits) - 1, 0, -1)`.
- Wenn `len(digits)== 5`, dann iteriert `range(len(digits) - 1, 0, -1)` über die Zahlen 4, 3, 2 und 1.
- Wir erhöhen die Ziffer an Index `i` jeweils um 1.
- Wenn das Ergebnis nicht 10 ist, dann können wir die Schleife mit `break` abbrechen.
- Wenn das Ergebnis 10 ist, dann setzen wir es auf 0 und iterieren weiter.

```
>>> Fraction(91995, 100000).decimal_str(3)
'0.92'
>>> Fraction(99995, 100000).decimal_str(4)
'1'
"""
a: int = self.a  # Get the numerator.
if a == 0:  # If the fraction is 0, we return 0.
    return "0"
negative: Final[bool] = a < 0  # Get the sign of the fraction.
a = abs(a)  # Make sure that `a` is now positive.
b: Final[int] = self.b  # Get the denominator.

digits: Final[list] = []  # A list for collecting digits.
while (a != 0) and (len(digits) <= max_frac):  # Create digits.
    digits.append(a // b)  # Add the current digit.
    a = 10 * (a % b)  # Ten times the remainder -> next digit.

if (a // b) >= 5:  # Do we need to round up?
    # This may lead to other digits topple over, e.g., 0.999...
    for i in range(len(digits) - 1, 0, -1):  # except first!
        digits[i] += 1  # Increment the digit at position i.
        if digits[i] != 10:  # Was there no overflow?
            break  # Digits in 1..9, no overflow, we can stop.
        digits[i] = 0  # We got a `10`, so we set it to 0.
    else:  # This is only reached if no `break` was done.
        digits[0] += 1  # Increment the integer part.

while digits[-1] == 0:  # Remove all trailing zeros.
    del digits[-1]  # Delete the trailing zero.

if len(digits) <= 1:  # Do we only have an integer part?
    return str((-1 if negative else 1) * digits[0])

digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
if negative:  # Do we need to restore the sign?
    digits.insert(0, "-")  # Insert the sign at the beginning.
return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- Wenn `len(digits)== 5`, dann iteriert `range(len(digits) - 1, 0, -1)` über die Zahlen 4, 3, 2 und 1.
- Wir erhöhen die Ziffer an Index `i` jeweils um 1.
- Wenn das Ergebnis nicht 10 ist, dann können wir die Schleife mit `break` abbrechen.
- Wenn das Ergebnis 10 ist, dann setzen wir es auf 0 und iterieren weiter.
- Somit wird dann die nächste Nachkommastelle erhöht, usw.

```
>>> Fraction(91995, 100000).decimal_str(3)
'0.92'
>>> Fraction(99995, 100000).decimal_str(4)
'1'
"""
a: int = self.a  # Get the numerator.
if a == 0:  # If the fraction is 0, we return 0.
    return "0"
negative: Final[bool] = a < 0  # Get the sign of the fraction.
a = abs(a)  # Make sure that `a` is now positive.
b: Final[int] = self.b  # Get the denominator.

digits: Final[list] = []  # A list for collecting digits.
while (a != 0) and (len(digits) <= max_frac):  # Create digits.
    digits.append(a // b)  # Add the current digit.
    a = 10 * (a % b)  # Ten times the remainder -> next digit.

if (a // b) >= 5:  # Do we need to round up?
    # This may lead to other digits topple over, e.g., 0.999...
    for i in range(len(digits) - 1, 0, -1):  # except first!
        digits[i] += 1  # Increment the digit at position i.
        if digits[i] != 10:  # Was there no overflow?
            break  # Digits in 1..9, no overflow, we can stop.
        digits[i] = 0  # We got a `10`, so we set it to 0.
    else:  # This is only reached if no `break` was done.
        digits[0] += 1  # Increment the integer part.

while digits[-1] == 0:  # Remove all trailing zeros.
    del digits[-1]  # Delete the trailing zero.

if len(digits) <= 1:  # Do we only have an integer part?
    return str((-1 if negative else 1) * digits[0])

digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
if negative:  # Do we need to restore the sign?
    digits.insert(0, "-")  # Insert the sign at the beginning.
return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- Wir erhöhen die Ziffer an Index `i` jeweils um 1.
- Wenn das Ergebnis nicht 10 ist, dann können wir die Schleife mit `break` abbrechen.
- Wenn das Ergebnis 10 ist, dann setzen wir es auf 0 und iterieren weiter.
- Somit wird dann die nächste Nachkommastelle erhöht, usw.
- Wenn wir bei Index 1 ankommen und trotzdem weiter iterieren müssten, dann ended die Schleife trotzdem.

```
        >>> Fraction(91995, 100000).decimal_str(3)
        '0.92'
        >>> Fraction(99995, 100000).decimal_str(4)
        '1'
        """
        a: int = self.a  # Get the numerator.
        if a == 0:  # If the fraction is 0, we return 0.
            return "0"
        negative: Final[bool] = a < 0  # Get the sign of the fraction.
        a = abs(a)  # Make sure that `a` is now positive.
        b: Final[int] = self.b  # Get the denominator.

        digits: Final[list] = []  # A list for collecting digits.
        while (a != 0) and (len(digits) <= max_frac):  # Create digits.
            digits.append(a // b)  # Add the current digit.
            a = 10 * (a % b)  # Ten times the remainder -> next digit.

        if (a // b) >= 5:  # Do we need to round up?
            # This may lead to other digits topple over, e.g., 0.999...
            for i in range(len(digits) - 1, 0, -1):  # except first!
                digits[i] += 1  # Increment the digit at position i.
                if digits[i] != 10:  # Was there no overflow?
                    break  # Digits in 1..9, no overflow, we can stop.
                digits[i] = 0  # We got a `10`, so we set it to 0.
            else:  # This is only reached if no `break` was done.
                digits[0] += 1  # Increment the integer part.

        while digits[-1] == 0:  # Remove all trailing zeros.
            del digits[-1]  # Delete the trailing zero.

        if len(digits) <= 1:  # Do we only have an integer part?
            return str((-1 if negative else 1) * digits[0])

        digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
        if negative:  # Do we need to restore the sign?
            digits.insert(0, "-")  # Insert the sign at the beginning.
        return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- Wenn das Ergebnis nicht 10 ist, dann können wir die Schleife mit `break` abbrechen.
- Wenn das Ergebnis 10 ist, dann setzen wir es auf 0 und iterieren weiter.
- Somit wird dann die nächste Nachkommastelle erhöht, usw.
- Wenn wir bei Index 1 ankommen und trotzdem weiter iterieren müssten, dann ended die Schleife trotzdem.
- Dann wird aber das `else`-Statement ausgeführt.

```
>>> Fraction(91995, 100000).decimal_str(3)
'0.92'
>>> Fraction(99995, 100000).decimal_str(4)
'1'
"""
a: int = self.a  # Get the numerator.
if a == 0:  # If the fraction is 0, we return 0.
    return "0"
negative: Final[bool] = a < 0  # Get the sign of the fraction.
a = abs(a)  # Make sure that `a` is now positive.
b: Final[int] = self.b  # Get the denominator.

digits: Final[list] = []  # A list for collecting digits.
while (a != 0) and (len(digits) <= max_frac):  # Create digits.
    digits.append(a // b)  # Add the current digit.
    a = 10 * (a % b)  # Ten times the remainder -> next digit.

if (a // b) >= 5:  # Do we need to round up?
    # This may lead to other digits topple over, e.g., 0.999...
    for i in range(len(digits) - 1, 0, -1):  # except first!
        digits[i] += 1  # Increment the digit at position i.
        if digits[i] != 10:  # Was there no overflow?
            break  # Digits in 1..9, no overflow, we can stop.
        digits[i] = 0  # We got a `10`, so we set it to 0.
    else:  # This is only reached if no `break` was done.
        digits[0] += 1  # Increment the integer part.

while digits[-1] == 0:  # Remove all trailing zeros.
    del digits[-1]  # Delete the trailing zero.

if len(digits) <= 1:  # Do we only have an integer part?
    return str((-1 if negative else 1) * digits[0])

digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
if negative:  # Do we need to restore the sign?
    digits.insert(0, "-")  # Insert the sign at the beginning.
return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- Wenn das Ergebnis 10 ist, dann setzen wir es auf 0 und iterieren weiter.
- Somit wird dann die nächste Nachkommastelle erhöht, usw.
- Wenn wir bei Index 1 ankommen und trotzdem weiter iterieren müssten, dann ended die Schleife trotzdem.
- Dann wird aber das `else`-Statement ausgeführt.
- Erinnern wir uns an Einheit 25.

```
        >>> Fraction(91995, 100000).decimal_str(3)
        '0.92'
        >>> Fraction(99995, 100000).decimal_str(4)
        '1'
        """
        a: int = self.a  # Get the numerator.
        if a == 0:  # If the fraction is 0, we return 0.
            return "0"
        negative: Final[bool] = a < 0  # Get the sign of the fraction.
        a = abs(a)  # Make sure that `a` is now positive.
        b: Final[int] = self.b  # Get the denominator.

        digits: Final[list] = []  # A list for collecting digits.
        while (a != 0) and (len(digits) <= max_frac):  # Create digits.
            digits.append(a // b)  # Add the current digit.
            a = 10 * (a % b)  # Ten times the remainder -> next digit.

        if (a // b) >= 5:  # Do we need to round up?
            # This may lead to other digits topple over, e.g., 0.999...
            for i in range(len(digits) - 1, 0, -1):  # except first!
                digits[i] += 1  # Increment the digit at position i.
                if digits[i] != 10:  # Was there no overflow?
                    break  # Digits in 1..9, no overflow, we can stop.
                digits[i] = 0  # We got a `10`, so we set it to 0.
            else:  # This is only reached if no `break` was done.
                digits[0] += 1  # Increment the integer part.

        while digits[-1] == 0:  # Remove all trailing zeros.
            del digits[-1]  # Delete the trailing zero.

        if len(digits) <= 1:  # Do we only have an integer part?
            return str((-1 if negative else 1) * digits[0])

        digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
        if negative:  # Do we need to restore the sign?
            digits.insert(0, "-")  # Insert the sign at the beginning.
        return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- Somit wird dann die nächste Nachkommastelle erhöht, usw.
- Wenn wir bei Index 1 ankommen und trotzdem weiter iterieren müssten, dann ended die Schleife trotzdem.
- Dann wird aber das `else`-Statement ausgeführt.
- Erinnern wir uns an Einheit 25:
- Das `else`-Statement am Ende einer Schleife wird nur dann ausgeführt, wenn die Schleife regulär beendet wurde, also wenn kein `break`-Statement ausgeführt wurde.

```
20      >>> Fraction(91995, 100000).decimal_str(3)
21      '0.92'
22      >>> Fraction(99995, 100000).decimal_str(4)
23      '1'
24      """
25      a: int = self.a  # Get the numerator.
26      if a == 0:  # If the fraction is 0, we return 0.
27          return "0"
28      negative: Final[bool] = a < 0  # Get the sign of the fraction.
29      a = abs(a)  # Make sure that `a` is now positive.
30      b: Final[int] = self.b  # Get the denominator.
31
32      digits: Final[list] = []  # A list for collecting digits.
33      while (a != 0) and (len(digits) <= max_frac):  # Create digits.
34          digits.append(a // b)  # Add the current digit.
35          a = 10 * (a % b)  # Ten times the remainder -> next digit.
36
37      if (a // b) >= 5:  # Do we need to round up?
38          # This may lead to other digits topple over, e.g., 0.999...
39          for i in range(len(digits) - 1, 0, -1):  # except first!
40              digits[i] += 1  # Increment the digit at position i.
41              if digits[i] != 10:  # Was there no overflow?
42                  break  # Digits in 1..9, no overflow, we can stop.
43              digits[i] = 0  # We got a `10`, so we set it to 0.
44          else:  # This is only reached if no `break` was done.
45              digits[0] += 1  # Increment the integer part.
46
47      while digits[-1] == 0:  # Remove all trailing zeros.
48          del digits[-1]  # Delete the trailing zero.
49
50      if len(digits) <= 1:  # Do we only have an integer part?
51          return str((-1 if negative else 1) * digits[0])
52
53      digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
54      if negative:  # Do we need to restore the sign?
55          digits.insert(0, "-")  # Insert the sign at the beginning.
56      return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- Wenn wir bei Index 1 ankommen und trotzdem weiter iterieren müssten, dann ended die Schleife trotzdem.
- Dann wird aber das `else`-Statement ausgeführt.
- Erinnern wir uns an Einheit 25:
- Das `else`-Statement am Ende einer Schleife wird nur dann ausgeführt, wenn die Schleife regulär beendet wurde, also wenn kein `break`-Statement ausgeführt wurde.
- Dann und nur dann wenn die Nachkommastelle and Index 1 auch 10 wurde und deshalb auf 0 gesetzt wurde, dann erhöhen wir die Zahl an Index 0 um 1.

```
        >>> Fraction(91995, 100000).decimal_str(3)
        '0.92'
        >>> Fraction(99995, 100000).decimal_str(4)
        '1'
        """
        a: int = self.a  # Get the numerator.
        if a == 0:  # If the fraction is 0, we return 0.
            return "0"
        negative: Final[bool] = a < 0  # Get the sign of the fraction.
        a = abs(a)  # Make sure that `a` is now positive.
        b: Final[int] = self.b  # Get the denominator.

        digits: Final[list] = []  # A list for collecting digits.
        while (a != 0) and (len(digits) <= max_frac):  # Create digits.
            digits.append(a // b)  # Add the current digit.
            a = 10 * (a % b)  # Ten times the remainder -> next digit.

        if (a // b) >= 5:  # Do we need to round up?
            # This may lead to other digits topple over, e.g., 0.999...
            for i in range(len(digits) - 1, 0, -1):  # except first!
                digits[i] += 1  # Increment the digit at position i.
                if digits[i] != 10:  # Was there no overflow?
                    break  # Digits in 1..9, no overflow, we can stop.
                digits[i] = 0  # We got a `10`, so we set it to 0.
            else:  # This is only reached if no `break` was done.
                digits[0] += 1  # Increment the integer part.

        while digits[-1] == 0:  # Remove all trailing zeros.
            del digits[-1]  # Delete the trailing zero.

        if len(digits) <= 1:  # Do we only have an integer part?
            return str((-1 if negative else 1) * digits[0])

        digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
        if negative:  # Do we need to restore the sign?
            digits.insert(0, "-")  # Insert the sign at the beginning.
        return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- Dann wird aber das `else`-Statement ausgeführt.
- Erinnern wir uns an Einheit 25:
- Das `else`-Statement am Ende einer Schleife wird nur dann ausgeführt, wenn die Schleife regulär beendet wurde, also wenn kein `break`-Statement ausgeführt wurde.
- Dann und nur dann wenn die Nachkommastelle and Index 1 auch 10 wurde und deshalb auf 0 gesetzt wurde, dann erhöhen wir die Zahl an Index 0 um 1.
- Diese Zahl repräsentiert den Ganzzahl-Teil unseres Bruches.

```
        >>> Fraction(91995, 100000).decimal_str(3)
        '0.92'
        >>> Fraction(99995, 100000).decimal_str(4)
        '1'
        """
        a: int = self.a  # Get the numerator.
        if a == 0:  # If the fraction is 0, we return 0.
            return "0"
        negative: Final[bool] = a < 0  # Get the sign of the fraction.
        a = abs(a)  # Make sure that `a` is now positive.
        b: Final[int] = self.b  # Get the denominator.

        digits: Final[list] = []  # A list for collecting digits.
        while (a != 0) and (len(digits) <= max_frac):  # Create digits.
            digits.append(a // b)  # Add the current digit.
            a = 10 * (a % b)  # Ten times the remainder -> next digit.

        if (a // b) >= 5:  # Do we need to round up?
            # This may lead to other digits topple over, e.g., 0.999...
            for i in range(len(digits) - 1, 0, -1):  # except first!
                digits[i] += 1  # Increment the digit at position i.
                if digits[i] != 10:  # Was there no overflow?
                    break  # Digits in 1..9, no overflow, we can stop.
                digits[i] = 0  # We got a `10`, so we set it to 0.
            else:  # This is only reached if no `break` was done.
                digits[0] += 1  # Increment the integer part.

        while digits[-1] == 0:  # Remove all trailing zeros.
            del digits[-1]  # Delete the trailing zero.

        if len(digits) <= 1:  # Do we only have an integer part?
            return str((-1 if negative else 1) * digits[0])

        digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
        if negative:  # Do we need to restore the sign?
            digits.insert(0, "-")  # Insert the sign at the beginning.
        return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- Erinnern wir uns an Einheit 25:
- Das `else`-Statement am Ende einer Schleife wird nur dann ausgeführt, wenn die Schleife regulär beendet wurde, also wenn kein `break`-Statement ausgeführt wurde.
- Dann und nur dann wenn die Nachkommastelle and Index 1 auch 10 wurde und deshalb auf 0 gesetzt wurde, dann erhöhen wir die Zahl an Index 0 um 1.
- Diese Zahl repräsentiert den Ganzzahl-Teil unseres Bruches.
- Hier ist es völlig OK, eine 9 auf 10 aufzurunden.

```
        >>> Fraction(91995, 100000).decimal_str(3)
        '0.92'
        >>> Fraction(99995, 100000).decimal_str(4)
        '1'
        """
        a: int = self.a  # Get the numerator.
        if a == 0:  # If the fraction is 0, we return 0.
            return "0"
        negative: Final[bool] = a < 0  # Get the sign of the fraction.
        a = abs(a)  # Make sure that `a` is now positive.
        b: Final[int] = self.b  # Get the denominator.

        digits: Final[list] = []  # A list for collecting digits.
        while (a != 0) and (len(digits) <= max_frac):  # Create digits.
            digits.append(a // b)  # Add the current digit.
            a = 10 * (a % b)  # Ten times the remainder -> next digit.

        if (a // b) >= 5:  # Do we need to round up?
            # This may lead to other digits topple over, e.g., 0.999...
            for i in range(len(digits) - 1, 0, -1):  # except first!
                digits[i] += 1  # Increment the digit at position i.
                if digits[i] != 10:  # Was there no overflow?
                    break  # Digits in 1..9, no overflow, we can stop.
                digits[i] = 0  # We got a `10`, so we set it to 0.
            else:  # This is only reached if no `break` was done.
                digits[0] += 1  # Increment the integer part.

        while digits[-1] == 0:  # Remove all trailing zeros.
            del digits[-1]  # Delete the trailing zero.

        if len(digits) <= 1:  # Do we only have an integer part?
            return str((-1 if negative else 1) * digits[0])

        digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
        if negative:  # Do we need to restore the sign?
            digits.insert(0, "-")  # Insert the sign at the beginning.
        return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- Das `else`-Statement am Ende einer Schleife wird nur dann ausgeführt, wenn die Schleife regulär beendet wurde, also wenn kein `break`-Statement ausgeführt wurde.
- Dann und nur dann wenn die Nachkommastelle and Index 1 auch 10 wurde und deshalb auf 0 gesetzt wurde, dann erhöhen wir die Zahl an Index 0 um 1.
- Diese Zahl repräsentiert den Ganzzahl-Teil unseres Bruches.
- Hier ist es völlig OK, eine 9 auf 10 aufzurunden.
- Z. B. kann man 9.999 auf 10 runden und 1239.9 auf 240.

```
>>> Fraction(91995, 100000).decimal_str(3)
'0.92'
>>> Fraction(99995, 100000).decimal_str(4)
'1'
"""
a: int = self.a  # Get the numerator.
if a == 0:  # If the fraction is 0, we return 0.
    return "0"
negative: Final[bool] = a < 0  # Get the sign of the fraction.
a = abs(a)  # Make sure that `a` is now positive.
b: Final[int] = self.b  # Get the denominator.

digits: Final[list] = []  # A list for collecting digits.
while (a != 0) and (len(digits) <= max_frac):  # Create digits.
    digits.append(a // b)  # Add the current digit.
    a = 10 * (a % b)  # Ten times the remainder -> next digit.

if (a // b) >= 5:  # Do we need to round up?
    # This may lead to other digits topple over, e.g., 0.999...
    for i in range(len(digits) - 1, 0, -1):  # except first!
        digits[i] += 1  # Increment the digit at position i.
        if digits[i] != 10:  # Was there no overflow?
            break  # Digits in 1..9, no overflow, we can stop.
        digits[i] = 0  # We got a `10`, so we set it to 0.
    else:  # This is only reached if no `break` was done.
        digits[0] += 1  # Increment the integer part.

while digits[-1] == 0:  # Remove all trailing zeros.
    del digits[-1]  # Delete the trailing zero.

if len(digits) <= 1:  # Do we only have an integer part?
    return str((-1 if negative else 1) * digits[0])

digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
if negative:  # Do we need to restore the sign?
    digits.insert(0, "-")  # Insert the sign at the beginning.
return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- Dann und nur dann wenn die Nachkommastelle and Index 1 auch 10 wurde und deshalb auf 0 gesetzt wurde, dann erhöhen wir die Zahl an Index 0 um 1.
- Diese Zahl repräsentiert den Ganzzahl-Teil unseres Bruches.
- Hier ist es völlig OK, eine 9 auf 10 aufzurunden.
- Z. B. kann man 9.999 auf 10 runden und 1239.9 auf 240.
- Dieser neue Kode kann zu Nullen am Ende des Strings führen.

```
        >>> Fraction(91995, 100000).decimal_str(3)
        '0.92'
        >>> Fraction(99995, 100000).decimal_str(4)
        '1'
        """
        a: int = self.a  # Get the numerator.
        if a == 0:  # If the fraction is 0, we return 0.
            return "0"
        negative: Final[bool] = a < 0  # Get the sign of the fraction.
        a = abs(a)  # Make sure that `a` is now positive.
        b: Final[int] = self.b  # Get the denominator.

        digits: Final[list] = []  # A list for collecting digits.
        while (a != 0) and (len(digits) <= max_frac):  # Create digits.
            digits.append(a // b)  # Add the current digit.
            a = 10 * (a % b)  # Ten times the remainder -> next digit.

        if (a // b) >= 5:  # Do we need to round up?
            # This may lead to other digits topple over, e.g., 0.999...
            for i in range(len(digits) - 1, 0, -1):  # except first!
                digits[i] += 1  # Increment the digit at position i.
                if digits[i] != 10:  # Was there no overflow?
                    break  # Digits in 1..9, no overflow, we can stop.
                digits[i] = 0  # We got a `10`, so we set it to 0.
            else:  # This is only reached if no `break` was done.
                digits[0] += 1  # Increment the integer part.

        while digits[-1] == 0:  # Remove all trailing zeros.
            del digits[-1]  # Delete the trailing zero.

        if len(digits) <= 1:  # Do we only have an integer part?
            return str((-1 if negative else 1) * digits[0])

        digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
        if negative:  # Do we need to restore the sign?
            digits.insert(0, "-")  # Insert the sign at the beginning.
        return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- Diese Zahl repräsentiert den Ganzzahl-Teil unseres Bruches.
- Hier ist es völlig OK, eine 9 auf 10 aufzurunden.
- Z. B. kann man 9.999 auf 10 runden und 1239.9 auf 240.
- Dieser neue Kode kann zu Nullen am Ende des Strings führen.
- Wir löschen diese mit einer zusätzlichen `while`-Schleife direkt nach dem Runden.

```
        >>> Fraction(91995, 100000).decimal_str(3)
        '0.92'
        >>> Fraction(99995, 100000).decimal_str(4)
        '1'
        """
        a: int = self.a  # Get the numerator.
        if a == 0:  # If the fraction is 0, we return 0.
            return "0"
        negative: Final[bool] = a < 0  # Get the sign of the fraction.
        a = abs(a)  # Make sure that `a` is now positive.
        b: Final[int] = self.b  # Get the denominator.

        digits: Final[list] = []  # A list for collecting digits.
        while (a != 0) and (len(digits) <= max_frac):  # Create digits.
            digits.append(a // b)  # Add the current digit.
            a = 10 * (a % b)  # Ten times the remainder -> next digit.

        if (a // b) >= 5:  # Do we need to round up?
            # This may lead to other digits topple over, e.g., 0.999...
            for i in range(len(digits) - 1, 0, -1):  # except first!
                digits[i] += 1  # Increment the digit at position i.
                if digits[i] != 10:  # Was there no overflow?
                    break  # Digits in 1..9, no overflow, we can stop.
                digits[i] = 0  # We got a `10`, so we set it to 0.
            else:  # This is only reached if no `break` was done.
                digits[0] += 1  # Increment the integer part.

        while digits[-1] == 0:  # Remove all trailing zeros.
            del digits[-1]  # Delete the trailing zero.

        if len(digits) <= 1:  # Do we only have an integer part?
            return str((-1 if negative else 1) * digits[0])

        digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
        if negative:  # Do we need to restore the sign?
            digits.insert(0, "-")  # Insert the sign at the beginning.
        return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- Diese Zahl repräsentiert den Ganzzahl-Teil unseres Bruches.
- Hier ist es völlig OK, eine 9 auf 10 aufzurunden.
- Z. B. kann man 9.999 auf 10 runden und 1239.9 auf 240.
- Dieser neue Kode kann zu Nullen am Ende des Strings führen.
- Wir löschen diese mit einer zusätzlichen `while`-Schleife direkt nach dem Runden.
- Wir haben nun funktionierenden Kode, der Brüche in Dezimalzahlen umwandelt.

```
        >>> Fraction(91995, 100000).decimal_str(3)
        '0.92'
        >>> Fraction(99995, 100000).decimal_str(4)
        '1'
        """
        a: int = self.a  # Get the numerator.
        if a == 0:  # If the fraction is 0, we return 0.
            return "0"
        negative: Final[bool] = a < 0  # Get the sign of the fraction.
        a = abs(a)  # Make sure that `a` is now positive.
        b: Final[int] = self.b  # Get the denominator.

        digits: Final[list] = []  # A list for collecting digits.
        while (a != 0) and (len(digits) <= max_frac):  # Create digits.
            digits.append(a // b)  # Add the current digit.
            a = 10 * (a % b)  # Ten times the remainder -> next digit.

        if (a // b) >= 5:  # Do we need to round up?
            # This may lead to other digits topple over, e.g., 0.999...
            for i in range(len(digits) - 1, 0, -1):  # except first!
                digits[i] += 1  # Increment the digit at position i.
                if digits[i] != 10:  # Was there no overflow?
                    break  # Digits in 1..9, no overflow, we can stop.
                digits[i] = 0  # We got a `10`, so we set it to 0.
            else:  # This is only reached if no `break` was done.
                digits[0] += 1  # Increment the integer part.

        while digits[-1] == 0:  # Remove all trailing zeros.
            del digits[-1]  # Delete the trailing zero.

        if len(digits) <= 1:  # Do we only have an integer part?
            return str((-1 if negative else 1) * digits[0])

        digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
        if negative:  # Do we need to restore the sign?
            digits.insert(0, "-")  # Insert the sign at the beginning.
        return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: decimal_str

- Hier ist es völlig OK, eine 9 auf 10 aufzurunden.
- Z. B. kann man 9.999 auf 10 runden und 1239.9 auf 240.
- Dieser neue Kode kann zu Nullen am Ende des Strings führen.
- Wir löschen diese mit einer zusätzlichen `while`-Schleife direkt nach dem Runden.
- Wir haben nun funktionierenden Kode, der Brüche in Dezimalzahlen umwandelt.
- All Doctests sind nun erfolgreich.

```
        >>> Fraction(91995, 100000).decimal_str(3)
        '0.92'
        >>> Fraction(99995, 100000).decimal_str(4)
        '1'
        """
        a: int = self.a  # Get the numerator.
        if a == 0:  # If the fraction is 0, we return 0.
            return "0"
        negative: Final[bool] = a < 0  # Get the sign of the fraction.
        a = abs(a)  # Make sure that `a` is now positive.
        b: Final[int] = self.b  # Get the denominator.

        digits: Final[list] = []  # A list for collecting digits.
        while (a != 0) and (len(digits) <= max_frac):  # Create digits.
            digits.append(a // b)  # Add the current digit.
            a = 10 * (a % b)  # Ten times the remainder -> next digit.

        if (a // b) >= 5:  # Do we need to round up?
            # This may lead to other digits topple over, e.g., 0.999...
            for i in range(len(digits) - 1, 0, -1):  # except first!
                digits[i] += 1  # Increment the digit at position i.
                if digits[i] != 10:  # Was there no overflow?
                    break  # Digits in 1..9, no overflow, we can stop.
                digits[i] = 0  # We got a `10`, so we set it to 0.
            else:  # This is only reached if no `break` was done.
                digits[0] += 1  # Increment the integer part.

        while digits[-1] == 0:  # Remove all trailing zeros.
            del digits[-1]  # Delete the trailing zero.

        if len(digits) <= 1:  # Do we only have an integer part?
            return str((-1 if negative else 1) * digits[0])

        digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
        if negative:  # Do we need to restore the sign?
            digits.insert(0, "-")  # Insert the sign at the beginning.
        return "".join(map(str, digits))  # Join all digits to a string.
```

Fraction: sqrt

- So, nun können wir das machen, was wir eigentlich machen wollten.

```python
"""A module with mathematics routines."""

from math import isclose  # Checks if two float numbers are similar.


def factorial(a: int) -> int:  # 1 `int` parameter and `int` result
    """
    Compute the factorial of a positive integer `a`.

    :param a: the number to compute the factorial of
    :return: the factorial of `a`, i.e., `a!`.
    """
    product: int = 1  # Initialize `product` as `1`.
    for i in range(2, a + 1):  # `i` goes from `2` to `a`.
        product *= i  # Multiply `i` to the product.
    return product  # Return the product, which now is the factorial.


def sqrt(number: float) -> float:
    """
    Compute the square root of a given `number`.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v` is approximately `number`.
    """
    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0  # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess   # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess
```

Fraction: sqrt

- So, nun können wir das machen, was wir eigentlich machen wollten:
- Wir wollten Quadratwurzeln mit wahnsinniger Genauigkeit berechnen!

```python
"""A module with mathematics routines."""

from math import isclose  # Checks if two float numbers are similar.


def factorial(a: int) -> int:  # 1 `int` parameter and `int` result
    """
    Compute the factorial of a positive integer `a`.

    :param a: the number to compute the factorial of
    :return: the factorial of `a`, i.e., `a!`.
    """
    product: int = 1  # Initialize `product` as `1`.
    for i in range(2, a + 1):  # `i` goes from `2` to `a`.
        product *= i  # Multiply `i` to the product.
    return product  # Return the product, which now is the factorial.


def sqrt(number: float) -> float:
    """
    Compute the square root of a given `number`.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v` is approximately `number`.
    """
    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0  # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess   # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess
```

Fraction: sqrt

- So, nun können wir das machen, was wir eigentlich machen wollten:
- Wir wollten Quadratwurzeln mit wahnsinniger Genauigkeit berechnen!
- Schauen wir uns dafür unsere Funktion `sqrt` aus `my_math.py` nochmal an.

```python
"""A module with mathematics routines."""

from math import isclose  # Checks if two float numbers are similar.


def factorial(a: int) -> int:  # 1 `int` parameter and `int` result
    """
    Compute the factorial of a positive integer `a`.

    :param a: the number to compute the factorial of
    :return: the factorial of `a`, i.e., `a!`.
    """
    product: int = 1  # Initialize `product` as `1`.
    for i in range(2, a + 1):  # `i` goes from `2` to `a`.
        product *= i  # Multiply `i` to the product.
    return product  # Return the product, which now is the factorial.


def sqrt(number: float) -> float:
    """
    Compute the square root of a given `number`.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v` is approximately `number`.
    """
    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0  # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess   # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess
```

Fraction: sqrt

- So, nun können wir das machen, was wir eigentlich machen wollten:
- Wir wollten Quadratwurzeln mit wahnsinniger Genauigkeit berechnen!
- Schauen wir uns dafür unsere Funktion `sqrt` aus `my_math.py` nochmal an.
- Im Grunde werden wir diese Funktion in unsere neue Datei `fraction_sqrt.py` kopieren und ein paar notwendige Änderungen vornehmen.

```python
"""A square root algorithm based on fractions."""

from fraction import ONE, ONE_HALF, ZERO, Fraction


def sqrt(number: Fraction, max_steps: int = 10) -> Fraction:
    """
    Compute the square root of a given :class:`Fraction`.

    :param number: The rational number to compute the square root of.
    :param max_steps: the maximum number of steps, defaults to `10`
    :return: A value `v` such that `v * v` is approximately `number`.

    >>> sqrt(Fraction(2, 1)).decimal_str(750)
    '1.414213562373095048801688724209698078569671875376948073176679379\
907324784621070388503875343276415727350138462309122970249248360558507372\
21264412149709993583141322266592750559275579995050115278206057147010955\
997160597027453459686201472851741864088919860955232923048430871432145 08\
397626036279952514079896872533965463318088296406206152583523950547457 50\
287759961729835575220337531857011354374603408498847160386899970769900481\
503054402779031645424782306849293691862158057846311159666871301301561 85\
689872372352885092648612494977154218334202485686060146824720771435858 7\
415565706967765372022648544701585880162075847492265722600020855844665214\
583988939443709265918003113882464681570826301005948587040031864803421 94\
897278290641045072636881313739855256117320 4025'
    >>> sqrt(Fraction(4, 1)).decimal_str()
    '2'
    >>> (ONE_HALF * (ONE + sqrt(Fraction(5, 1)))).decimal_str(420)
    '1.618033988749894848204586834365638117720309179805762862135448627\
052604628189024497072072041893911374847540880753868917521266338622235 36\
931793180060766726354433389086595939582905638322661319928290267880 0720\
876689250171169620703222104321626954862629631361443814975870122030 8058\
879544547492461856953648644492410443207713449470495658467885098743 3944 2\
212544877066478091588460749988712400765217057517978834166256249407 5890 7'
    """
    if number < ZERO:  # No negative numbers are permitted.
        raise ArithmeticError(f"Cannot computed sqrt({number}).")
    guess: Fraction = ONE        # This will hold the current guess.
    old_guess: Fraction = ZERO  # 0.0 is just a dummy value != guess.
    while old_guess != guess:  # Repeat until nothing changes anymore.
        old_guess = guess  # The current guess becomes the old guess.
        guess = ONE_HALF * (guess + number / guess)  # The new guess.
        max_steps -= 1       # Reduce the number of remaining steps.
        if max_steps <= 0:  # If we have exhausted the maximum steps...
            break           # ...then we stop (and return the guess).
    return guess  # Return the final guess.
```

Fraction: sqrt

- So, nun können wir das machen, was wir eigentlich machen wollten:
- Wir wollten Quadratwurzeln mit wahnsinniger Genauigkeit berechnen!
- Schauen wir uns dafür unsere Funktion `sqrt` aus `my_math.py` nochmal an.
- Im Grunde werden wir diese Funktion in unsere neue Datei `fraction_sqrt.py` kopieren und ein paar notwendige Änderungen vornehmen.
- Wir müssen alle `floats` in diesem Kode mit `Fractions` austauschen.

```
"""A square root algorithm based on fractions."""

from fraction import ONE, ONE_HALF, ZERO, Fraction


def sqrt(number: Fraction, max_steps: int = 10) -> Fraction:
    """
    Compute the square root of a given :class:`Fraction`.

    :param number: The rational number to compute the square root of.
    :param max_steps: the maximum number of steps, defaults to `10`
    :return: A value `v` such that `v * v` is approximately `number`.

    >>> sqrt(Fraction(2, 1)).decimal_str(750)
    '1.414213562373095048801688724209698078569671875376948073176679379\
907324784621070388503875343276415727350138462309122970249248360558507372\
21264412149709993583141322266592750559275579995050115278206057147010955\
997160597027453459686201472851741864088919860955232923048430871432145 08\
397626036279952514079896872533965463318088296406206152583523950547457 50\
287759961729835575220337531857011354374603408498847160386899970769900481\
503054402779031645424782306849293691862158057846311159666871301301561 85\
689872372352885092648612494977154218334202428568606014682472077143585487\
415565706967765372022648544701585880162075847492265722600208558446665214\
583988939443709265918003113882464681570826301005948587040031864803421 94\
897278290641045072636881313739855256117320 4025'
    >>> sqrt(Fraction(4, 1)).decimal_str()
    '2'
    >>> (ONE_HALF * (ONE + sqrt(Fraction(5, 1)))).decimal_str(420)
    '1.618033988749894848204586834365638117720309179805762862135448627\
052604628189024497072072041893911374847540880753868917521266338622233536\
931793180060766726354433389086595939582905638322661319928290267880675 20\
876689250171696207032221043216269548626296313614438149758701220340805 8\
879544547492461856953648644492410443207713449470495658467885098743394 42\
212544877066478091588460749988712400765217057517978834166256249407589 07'
    """
    if number < ZERO:  # No negative numbers are permitted.
        raise ArithmeticError(f"Cannot computed sqrt({number}).")
    guess: Fraction = ONE        # This will hold the current guess.
    old_guess: Fraction = ZERO  # 0.0 is just a dummy value != guess.
    while old_guess != guess:  # Repeat until nothing changes anymore.
        old_guess = guess  # The current guess becomes the old guess.
        guess = ONE_HALF * (guess + number / guess)  # The new guess.
        max_steps -= 1       # Reduce the number of remaining steps.
        if max_steps <= 0:  # If we have exhausted the maximum steps...
            break           # ...then we stop (and return the guess).
    return guess  # Return the final guess.
```

Fraction: sqrt

- Wir wollten Quadratwurzeln mit wahnsinniger Genauigkeit berechnen!
- Schauen wir uns dafür unsere Funktion `sqrt` aus `my_math.py` nochmal an.
- Im Grunde werden wir diese Funktion in unsere neue Datei `fraction_sqrt.py` kopieren und ein paar notwendige Änderungen vornehmen.
- Wir müssen alle `floats` in diesem Kode mit `Fractions` austauschen.
- Die Signatur der Funktion ändert sich von `def sqrt(number: float) -> float` zu `def sqrt(number: Fraction) -> Fraction`.

```python
"""A square root algorithm based on fractions."""

from fraction import ONE, ONE_HALF, ZERO, Fraction


def sqrt(number: Fraction, max_steps: int = 10) -> Fraction:
    """
    Compute the square root of a given :class:`Fraction`.

    :param number: The rational number to compute the square root of.
    :param max_steps: the maximum number of steps, defaults to `10`
    :return: A value `v` such that `v * v` is approximately `number`.

    >>> sqrt(Fraction(2, 1)).decimal_str(750)
    '1.414213562373095048801688724209698078569671875376948073176679379\
907324784621070388503875343276415727350138462309122970249248360558507372\
21264412149709993583141322266592750559275579995050115278206057147010955\
99716059702745345968620147285174186408891986095523292304843087143214508\
39762603627995251407989687253396546331808829640620615258352395054745750\
28775996172983557522033753185701135437460340849884716038689997069900481\
50305440277903164542478230684929369186215805784631115966687130130156185\
68987237235288509264861249497715421833420428568606014682472077143585487\
41556570696776537202264854470158588016207584749226572260020855844665214\
58398893944370926591800311388246468157082630100594858704003186480342194\
89727829064104507263688131373985525611732204025'
    >>> sqrt(Fraction(4, 1)).decimal_str()
    '2'
    >>> (ONE_HALF * (ONE + sqrt(Fraction(5, 1)))).decimal_str(420)
    '1.618033988749894848204586834365638117720309179805762862135448627\
052604628189024497072072041893911374847540880753868917521266338622233536\
931793180060766726354433389808659593958290563832266131992829026788067520\
876689250171169620703222104321626954862629631361443814975870122034080585\
879544547492461856953648644492410443207713449470495658467885098743394421\
212544877066478091588460749988712400765217057517978834166256249407585907'
    """
    if number < ZERO:  # No negative numbers are permitted.
        raise ArithmeticError(f"Cannot computed sqrt({number}).")
    guess: Fraction = ONE        # This will hold the current guess.
    old_guess: Fraction = ZERO  # 0.0 is just a dummy value != guess.
    while old_guess != guess:  # Repeat until nothing changes anymore.
        old_guess = guess  # The current guess becomes the old guess.
        guess = ONE_HALF * (guess + number / guess)  # The new guess.
        max_steps -= 1      # Reduce the number of remaining steps.
        if max_steps <= 0:  # If we have exhausted the maximum steps...
            break           # ...then we stop (and return the guess).
    return guess  # Return the final guess.
```

Fraction: sqrt

- Im Grunde werden wir diese Funktion in unsere neue Datei `fraction_sqrt.py` kopieren und ein paar notwendige Änderungen vornehmen.
- Wir müssen alle `floats` in diesem Kode mit `Fractions` austauschen.
- Die Signatur der Funktion ändert sich von `def sqrt(number: float) -> float` zu `def sqrt(number: Fraction) -> Fraction`.
- Nun wissen wir ja, dass die Quadratwurzel von Zahlen wie $\sqrt{2}$ irrational sind.

```python
"""A square root algorithm based on fractions."""

from fraction import ONE, ONE_HALF, ZERO, Fraction


def sqrt(number: Fraction, max_steps: int = 10) -> Fraction:
    """
    Compute the square root of a given :class:`Fraction`.

    :param number: The rational number to compute the square root of.
    :param max_steps: the maximum number of steps, defaults to `10`
    :return: A value `v` such that `v * v` is approximately `number`.

    >>> sqrt(Fraction(2, 1)).decimal_str(750)
    '1.414213562373095048801688724209698078569671875376948073176679379\
907324784621070388503875343276415727350138462309122970249248360558507372\
21264412149709993583141322266592750559275579995050115278206057147010955\
997160597027453459686201472851741864088919860955232923048430871432145085\
397626036279952514079896872533965463318088296406206152583523950547457505\
287759961729835575220337531857011354374603408498847160386899970699004815\
0530544027790316454247823068492936918621580578463111596668713013015618 5\
689872372352885092648612494977154218334204285686060146824720771435854875\
415565706967765372022648544701585880162075847492265722600208558446652145\
583988939443709265918003113882464681570826301005948587040031864803421945\
897278290641045072636881313739855256117320 4025'
    >>> sqrt(Fraction(4, 1)).decimal_str()
    '2'
    >>> (ONE_HALF * (ONE + sqrt(Fraction(5, 1)))).decimal_str(420)
    '1.618033988749894848204586834365638117720309179805762862135448622 7\
052604628189024497072072041893911374847540880753868917521266338622233536\
931793180060766726354433389086595939582905638322661319928290267880675205\
876689250171169620703222104321626954862629631361443814975870122034080585\
879544547492461856953648644492410443207713449470495658467885098743399442\
212544877066478091588460749988712400765217057517978834166256249407585907'
    """
    if number < ZERO:  # No negative numbers are permitted.
        raise ArithmeticError(f"Cannot computed sqrt({number}).")
    guess: Fraction = ONE        # This will hold the current guess.
    old_guess: Fraction = ZERO  # 0.0 is just a dummy value != guess.
    while old_guess != guess:  # Repeat until nothing changes anymore.
        old_guess = guess  # The current guess becomes the old guess.
        guess = ONE_HALF * (guess + number / guess)  # The new guess.
        max_steps -= 1       # Reduce the number of remaining steps.
        if max_steps <= 0:  # If we have exhausted the maximum steps...
            break           # ...then we stop (and return the guess).
    return guess  # Return the final guess.
```

Fraction: sqrt

- Wir müssen alle `floats` in diesem Kode mit `Fractions` austauschen.
- Die Signatur der Funktion ändert sich von `def sqrt(number: float) -> float` zu `def sqrt(number: Fraction) -> Fraction`.
- Nun wissen wir ja, dass die Quadratwurzel von Zahlen wie $\sqrt{2}$ irrational sind.
- Wir können sie also nicht exakt mit einer Instanz von `Fraction` darstellen.

```
"""A square root algorithm based on fractions."""

from fraction import ONE, ONE_HALF, ZERO, Fraction


def sqrt(number: Fraction, max_steps: int = 10) -> Fraction:
    """
    Compute the square root of a given :class:`Fraction`.

    :param number: The rational number to compute the square root of.
    :param max_steps: the maximum number of steps, defaults to `10`
    :return: A value `v` such that `v * v` is approximately `number`.

    >>> sqrt(Fraction(2, 1)).decimal_str(750)
    '1.414213562373095048801688724209698078569671875376948073176679379\
907324784621070388503875343276415727350138462309122970249248360558507370\
212644121497099935831413222665927505592755799950501152782060571470109955\
997160597027453459686201472851741864088919860955232923048430871432145085\
397626036279952514079896872533965463318088296406206152583523950547457505\
287759961729835575220337531857011354374603408498847160386899970699004815\
503054402779031645424782306849293691862158057846311159666871301301561855\
689872372352885092648612494977154218334202428568606014682472077143585487\
415565706967765372022648544701585880162075847492265722600208558446652145\
583988939443709265918003113882464681570826301005948587040031864803421945\
897278290641045072636881313739855256117320240 25'
    >>> sqrt(Fraction(4, 1)).decimal_str()
    '2'
    >>> (ONE_HALF * (ONE + sqrt(Fraction(5, 1)))).decimal_str(420)
    '1.618033988749894848204586834365638117720309179805762862135448622\
705260462818902449707207204189391137484754088075386891752126633862223536\
931793180060766726354433389086595939582905638322661319928290267880675205\
876689250171169620703222104321626954862629631361443814975870122034080585\
879544547492461856953648644492410443207713449470495658467885098743399442\
212544877066478091588460749988712400765217057517978834166256249407589 07'
    """
    if number < ZERO:  # No negative numbers are permitted.
        raise ArithmeticError(f"Cannot computed sqrt({number}).")
    guess: Fraction = ONE       # This will hold the current guess.
    old_guess: Fraction = ZERO  # 0.0 is just a dummy value != guess.
    while old_guess != guess:  # Repeat until nothing changes anymore.
        old_guess = guess  # The current guess becomes the old guess.
        guess = ONE_HALF * (guess + number / guess)  # The new guess.
        max_steps -= 1      # Reduce the number of remaining steps.
        if max_steps <= 0:  # If we have exhausted the maximum steps...
            break           # ...then we stop (and return the guess).
    return guess  # Return the final guess.
```

Fraction: sqrt

- Wir müssen alle `floats` in diesem Kode mit `Fractions` austauschen.
- Die Signatur der Funktion ändert sich von `def sqrt(number: float) -> float` zu `def sqrt(number: Fraction) -> Fraction`.
- Nun wissen wir ja, dass die Quadratwurzel von Zahlen wie $\sqrt{2}$ irrational sind.
- Wir können sie also nicht exakt mit einer Instanz von `Fraction` darstellen.
- Im Originalkode ist das kein Problem.

```
"""A square root algorithm based on fractions."""

from fraction import ONE, ONE_HALF, ZERO, Fraction


def sqrt(number: Fraction, max_steps: int = 10) -> Fraction:
    """
    Compute the square root of a given :class:`Fraction`.

    :param number: The rational number to compute the square root of.
    :param max_steps: the maximum number of steps, defaults to `10`
    :return: A value `v` such that `v * v` is approximately `number`.

    >>> sqrt(Fraction(2, 1)).decimal_str(750)
    '1.414213562373095048801688724209698078569671875376948073176679737\
9\
907324784621070388503875343276415727350138462309122970249248360558507\
37\
212644121497099935831413222665927505592755799950501152782060571470109\
55\
997160597027453459686201472851741864088919860955232923048430871432145\
08\
397626036279952514079896872533965463318088296406206152583523950547457\
50\
287759961729835575220337531857011354374603408498847160386899970690\
81\
503054402779031645424782306849293691862158057846311159666871301301561\
85\
689872372352885092648612494977154218334202485686060146824720771435854\
87\
415565706967765372022648544701585880162075847492265722600020855844665\
21\
583988939443709265918003113882464681570826301005948587040031864803421\
94\
897278290641045072636881313739855256117320\
4025'
    >>> sqrt(Fraction(4, 1)).decimal_str()
    '2'
    >>> (ONE_HALF * (ONE + sqrt(Fraction(5, 1)))).decimal_str(420)
    '1.6180339887498948482045868343656381177203091798057628621354486227\
\
052604628189024497072072041893911374847540880753868917521266338622233\
536\
931793180060766726354433389808659593958290563832266131992829026788067\
520\
876689250171169620703222104321626954862629631361443814975870122034080\
58\
879544547492461856953648644492410443207713449470495658467885098743394\
42\
212544877066478091588460749988712400765217057517978834166256249407585\
07'
    """
    if number < ZERO:  # No negative numbers are permitted.
        raise ArithmeticError(f"Cannot computed sqrt({number}).")
    guess: Fraction = ONE        # This will hold the current guess.
    old_guess: Fraction = ZERO  # 0.0 is just a dummy value != guess.
    while old_guess != guess:  # Repeat until nothing changes anymore.
        old_guess = guess  # The current guess becomes the old guess.
        guess = ONE_HALF * (guess + number / guess)  # The new guess.
        max_steps -= 1       # Reduce the number of remaining steps.
        if max_steps <= 0:  # If we have exhausted the maximum steps...
            break           # ...then we stop (and return the guess).
    return guess  # Return the final guess.
```

Fraction: sqrt

- Die Signatur der Funktion ändert sich von `def sqrt(number: float) -> float` zu `def sqrt(number: Fraction) -> Fraction`.
- Nun wissen wir ja, dass die Quadratwurzel von Zahlen wie $\sqrt{2}$ irrational sind.
- Wir können sie also nicht exakt mit einer Instanz von `Fraction` darstellen.
- Im Originalkode ist das kein Problem.
- Die Originalfunktion wird wegen der begrenzten Genauigkeit des Datentyps `float` irgendwann aufhören.

```python
"""A square root algorithm based on fractions."""

from fraction import ONE, ONE_HALF, ZERO, Fraction


def sqrt(number: Fraction, max_steps: int = 10) -> Fraction:
    """
    Compute the square root of a given :class:`Fraction`.

    :param number: The rational number to compute the square root of.
    :param max_steps: the maximum number of steps, defaults to `10`
    :return: A value `v` such that `v * v` is approximately `number`.

    >>> sqrt(Fraction(2, 1)).decimal_str(750)
    '1.414213562373095048801688724209698078569671875376948073176679379\
907324784621070388503875343276415727350138462309122970249248360558507372\
21264411219470999358314132226659275055927557999505011527820605714701095\
99716059702745345968620147285174186408891986095523292304843087143214508\
39762603627995251407989687253396546331808829640620615258352395054745750\
28775996172983557522033753185701135437460340849884716038689997069900481\
50305440277903164542478230684929369186215805784631115966687130130156185\
68987237235288509264861249497715421833420428568660601468247207714358548 7\
41556570696776537202264854470158588016207584749226572260020855844665214\
58398899344370926591800311388246468157082630100594858870400318648034219 4\
89727829064104507263688131373985525611732204025'
    >>> sqrt(Fraction(4, 1)).decimal_str()
    '2'
    >>> (ONE_HALF * (ONE + sqrt(Fraction(5, 1)))).decimal_str(420)
    '1.6180339887498948482045868343656381177203091798057628621354486227\
052604628189024497072072041893911374847540880753868917521266338622233536\
93179318006076672635443338908659593958290563832266131992829026788067520\
87668925017116962070322210432162695486262963136144381497587012203408058\
87954454749246185695364864449241044320771344947049565846788509874339442\
21254487706647809158846074998871240076521705751797883416625624940758907'
    """
    if number < ZERO:  # No negative numbers are permitted.
        raise ArithmeticError(f"Cannot computed sqrt({number}).")
    guess: Fraction = ONE        # This will hold the current guess.
    old_guess: Fraction = ZERO  # 0.0 is just a dummy value != guess.
    while old_guess != guess:  # Repeat until nothing changes anymore.
        old_guess = guess  # The current guess becomes the old guess.
        guess = ONE_HALF * (guess + number / guess)  # The new guess.
        max_steps -= 1       # Reduce the number of remaining steps.
        if max_steps <= 0:  # If we have exhausted the maximum steps...
            break           # ...then we stop (and return the guess).
    return guess  # Return the final guess.
```

Fraction: sqrt

- Nun wissen wir ja, dass die Quadratwurzel von Zahlen wie $\sqrt{2}$ irrational sind.
- Wir können sie also nicht exakt mit einer Instanz von `Fraction` darstellen.
- Im Originalkode ist das kein Problem.
- Die Originalfunktion wird wegen der begrenzten Genauigkeit des Datentyps `float` irgendwann aufhören.
- Das passiert, wenn die 15 bis 16 Ziffern Genauigkeit von `float` aufgebraucht sind.

```python
"""A square root algorithm based on fractions."""

from fraction import ONE, ONE_HALF, ZERO, Fraction


def sqrt(number: Fraction, max_steps: int = 10) -> Fraction:
    """
    Compute the square root of a given :class:`Fraction`.

    :param number: The rational number to compute the square root of.
    :param max_steps: the maximum number of steps, defaults to `10`
    :return: A value `v` such that `v * v` is approximately `number`.

    >>> sqrt(Fraction(2, 1)).decimal_str(750)
    '1.414213562373095048801688724209698078569671875376948073176679737\
9073247846210703885038753432764157273501384623091229702492483605585073\
7212644121497099935831413222665927505592755799950501152782060571470109\
9971605970274534596862014728517418640889198609552329230484308714321450\
3976260362799525140798968725339654633180882964062061525835239505474575\
2877599617298355752203375318570113543746034084988471603868999709690048\
1503054402779031645424782306849293691862158057846311159666871301301561\
6898723723528850926486124949771542183342042856860601468247207714358548\
7\
4155657069677653720226485447015858801620758474922657226002085584466521\
4\
5839889394437092659180031138824646815708263010059485870400318648034219\
4\
8972782906410450726368813137398552561173220405'
    >>> sqrt(Fraction(4, 1)).decimal_str()
    '2'
    >>> (ONE_HALF * (ONE + sqrt(Fraction(5, 1)))).decimal_str(420)
    '1.6180339887498948482045868343656381177203091798057628621354486227\
0526046281890244970720720418939111374847540880753868917521266338622233536\
9317931800607667263544333890865959395829056383226613199282902678806752\
0\
8766892501711696207032221043216269548626296313614438149758701220304805\
8\
8795445474924618569536486444924104432077134494704956584678850987433944\
2\
2125448770664780915884607499887124007652170575179788341662562494075890\
7'
    """
    if number < ZERO:  # No negative numbers are permitted.
        raise ArithmeticError(f"Cannot computed sqrt({number}).")
    guess: Fraction = ONE        # This will hold the current guess.
    old_guess: Fraction = ZERO  # 0.0 is just a dummy value != guess.
    while old_guess != guess:  # Repeat until nothing changes anymore.
        old_guess = guess  # The current guess becomes the old guess.
        guess = ONE_HALF * (guess + number / guess)  # The new guess.
        max_steps -= 1       # Reduce the number of remaining steps.
        if max_steps <= 0:  # If we have exhausted the maximum steps...
            break           # ...then we stop (and return the guess).
    return guess  # Return the final guess.
```

Fraction: sqrt

- Wir können sie also nicht exakt mit einer Instanz von `Fraction` darstellen.
- Im Originalkode ist das kein Problem.
- Die Originalfunktion wird wegen der begrenzten Genauigkeit des Datentyps `float` irgendwann aufhören.
- Das passiert, wenn die 15 bis 16 Ziffern Genauigkeit von `float` aufgebraucht sind.
- Mit `Fraction` kann er selbe Kode aber unendlich lange iterieren.

```
"""A square root algorithm based on fractions."""

from fraction import ONE, ONE_HALF, ZERO, Fraction


def sqrt(number: Fraction, max_steps: int = 10) -> Fraction:
    """
    Compute the square root of a given :class:`Fraction`.

    :param number: The rational number to compute the square root of.
    :param max_steps: the maximum number of steps, defaults to `10`
    :return: A value `v` such that `v * v` is approximately `number`.

    >>> sqrt(Fraction(2, 1)).decimal_str(750)
    '1.414213562373095048801688724209698078569671875376948073176679379\
907324784621070388503875343276415727350138462309122970249248360558507372\
21264412149709993583141322266592755055927557999505011527820605714701095\
99716059702745345968620147285174186408891986095523292304843087143214508\
39762603627995251407989687253396546331808829640620615258352395054745750\
28775996172983557522033753185701135437460340849884716038689997069900481\
50305440277903164542478230684929369186215805784631115966687130130156185\
68987237235288509264861249497715421833420428568660601468247207714358548 7\
41556570696776537202264854470158588016207584749226572260020855844665214\
58398893944370926591800311388246468157082630100594858870400318648034219 4\
89727829064104507263688131373985525611732204025'
    >>> sqrt(Fraction(4, 1)).decimal_str()
    '2'
    >>> (ONE_HALF * (ONE + sqrt(Fraction(5, 1)))).decimal_str(420)
    '1.618033988749894848204586834365638117720309179805762862135448627\
052604628189024497072072041893911374847540880753868917521266338622235 36\
931793180060766726354433389086595939582905638322661319928290267880675 20\
876689250171169620703222104321626954862629631361443814975870122034080 58\
879544547492461856953648644492410443207713449470495658467885098743399 42\
212544877066478091588460749988712400765217057517978834166256249407585 07'
    """
    if number < ZERO:  # No negative numbers are permitted.
        raise ArithmeticError(f"Cannot computed sqrt({number}).")
    guess: Fraction = ONE        # This will hold the current guess.
    old_guess: Fraction = ZERO  # 0.0 is just a dummy value != guess.
    while old_guess != guess:  # Repeat until nothing changes anymore.
        old_guess = guess  # The current guess becomes the old guess.
        guess = ONE_HALF * (guess + number / guess)  # The new guess.
        max_steps -= 1       # Reduce the number of remaining steps.
        if max_steps <= 0:  # If we have exhausted the maximum steps...
            break           # ...then we stop (and return the guess).
    return guess  # Return the final guess.
```

Fraction: sqrt

- Im Originalkode ist das kein Problem.
- Die Originalfunktion wird wegen der begrenzten Genauigkeit des Datentyps `float` irgendwann aufhören.
- Das passiert, wenn die 15 bis 16 Ziffern Genauigkeit von `float` aufgebraucht sind.
- Mit `Fraction` kann er selbe Kode aber unendlich lange iterieren.
- Wir haben ja eine Genauigkeit, die nur durch den zur Verfügung stehenden Arbeitsspeicher begrenzt wird.

```
"""A square root algorithm based on fractions."""

from fraction import ONE, ONE_HALF, ZERO, Fraction


def sqrt(number: Fraction, max_steps: int = 10) -> Fraction:
    """
    Compute the square root of a given :class:`Fraction`.

    :param number: The rational number to compute the square root of.
    :param max_steps: the maximum number of steps, defaults to `10`
    :return: A value `v` such that `v * v` is approximately `number`.

    >>> sqrt(Fraction(2, 1)).decimal_str(750)
    '1.414213562373095048801688724209698078569671875376948073176679737\
9\
907324784621070388503875343276415727350138462309122970249248360558507\
37\
212644121497099935831413222665927505592755799950501152782060571470109\
55\
997160597027453459686201472851741864088919860955232923048430871432145\
08\
397626036279952514079896872533965463318088296406206152583523950547457\
50\
287759961729835575220337531857011354374603408498847160386899970699004\
81\
503054402779031645424782306849293691862158057846311159666871301301561\
85\
689872372352885092648612494977154218334204285686060146824720771435854\
87\
415565706967765372022648544701585880162075847492265722600020855844665\
214\
583988939443709265918003113882464681570826301005948587040031864803421\
94\
897278290641045072636881313739855256117320 4025'
    >>> sqrt(Fraction(4, 1)).decimal_str()
    '2'
    >>> (ONE_HALF * (ONE + sqrt(Fraction(5, 1)))).decimal_str(420)
    '1.618033988749894848204586834365638117720309179805762862135448627\
5\
052604628189024497072072041893911374847540880753868917521266338622235\
36\
931793180060766726354433389080659593958290563832266131992829026788067\
520\
876689250171169620703222104321626954862629631361443814975870122034080\
58\
879544547492461856953648644492410443207713449470495658467885098743394\
42\
212544877066478091588460749988712400765217057517978834166256249407589\
07'
    """
    if number < ZERO:  # No negative numbers are permitted.
        raise ArithmeticError(f"Cannot computed sqrt({number}).")
    guess: Fraction = ONE        # This will hold the current guess.
    old_guess: Fraction = ZERO  # 0.0 is just a dummy value != guess.
    while old_guess != guess:  # Repeat until nothing changes anymore.
        old_guess = guess  # The current guess becomes the old guess.
        guess = ONE_HALF * (guess + number / guess)  # The new guess.
        max_steps -= 1       # Reduce the number of remaining steps.
        if max_steps <= 0:  # If we have exhausted the maximum steps...
            break           # ...then we stop (and return the guess).
    return guess  # Return the final guess.
```

Fraction: sqrt

- Die Originalfunktion wird wegen der begrenzten Genauigkeit des Datentyps `float` irgendwann aufhören.
- Das passiert, wenn die 15 bis 16 Ziffern Genauigkeit von `float` aufgebraucht sind.
- Mit `Fraction` kann er selbe Kode aber unendlich lange iterieren.
- Wir haben ja eine Genauigkeit, die nur durch den zur Verfügung stehenden Arbeitsspeicher begrenzt wird.
- Der Kode könnte fast für immer nach besseren Annäherungen suchen.

```
"""A square root algorithm based on fractions."""

from fraction import ONE, ONE_HALF, ZERO, Fraction


def sqrt(number: Fraction, max_steps: int = 10) -> Fraction:
    """
    Compute the square root of a given :class:`Fraction`.

    :param number: The rational number to compute the square root of.
    :param max_steps: the maximum number of steps, defaults to `10`
    :return: A value `v` such that `v * v` is approximately `number`.

    >>> sqrt(Fraction(2, 1)).decimal_str(750)
    '1.414213562373095048801688724209698078569671875376948073176679737\
9\
07324784621070388503875343276415727350138462309122970249248360558507\
37\
21264412149709993583141322266592750559275579995050115278206057147010955\
\
99716059702745345968620147285174186408891986095523292304843087143214508\
\
39762603627995251407989687253396546331808829640620615258352395054745750\
\
28775996172983557522033753185701135437460340849884716038689997069900481\
\
50305440277903164542478230684929369186215805784631115966687130130156185\
\
68987237235288509264861249497715421833420428568660601468247207714358548\
7\
41556570696776537202264854470158588016207584749226572260020855844665214\
\
58398893944370926591800311388246468157082630100594858870400318648034219\
4\
89727829064104507263688131373985525611732204025'
    >>> sqrt(Fraction(4, 1)).decimal_str()
    '2'
    >>> (ONE_HALF * (ONE + sqrt(Fraction(5, 1)))).decimal_str(420)
    '1.618033988749894848204586834365638117720309179805762862135448627\
\
52606426189024497072072041893911374847540880753868917521266338622235\
36\
93179318006076672635443338908659593958290563832266131992829026788067520\
\
87668925017116962070322210432162695486262963136144338149758701220340805\
8\
87954454749246185695364864449241044320771344947049565846788509874339442\
\
21254487706647809158846074998871240076521705751797883416625624940758907'
    """
    if number < ZERO:  # No negative numbers are permitted.
        raise ArithmeticError(f"Cannot computed sqrt({number}).")
    guess: Fraction = ONE        # This will hold the current guess.
    old_guess: Fraction = ZERO  # 0.0 is just a dummy value != guess.
    while old_guess != guess:  # Repeat until nothing changes anymore.
        old_guess = guess  # The current guess becomes the old guess.
        guess = ONE_HALF * (guess + number / guess)  # The new guess.
        max_steps -= 1      # Reduce the number of remaining steps.
        if max_steps <= 0:  # If we have exhausted the maximum steps...
            break           # ...then we stop (and return the guess).
    return guess  # Return the final guess.
```

Fraction: sqrt

- Das passiert, wenn die 15 bis 16 Ziffern Genauigkeit von `float` aufgebraucht sind.
- Mit `Fraction` kann er selbe Kode aber unendlich lange iterieren.
- Wir haben ja eine Genauigkeit, die nur durch den zur Verfügung stehenden Arbeitsspeicher begrenzt wird.
- Der Kode könnte fast für immer nach besseren Annäherungen suchen.
- Wir müssen also die Anzahl der Iterationen mit einem weiteren Parameter begrenzen.

```
"""A square root algorithm based on fractions."""

from fraction import ONE, ONE_HALF, ZERO, Fraction


def sqrt(number: Fraction, max_steps: int = 10) -> Fraction:
    """
    Compute the square root of a given :class:`Fraction`.

    :param number: The rational number to compute the square root of.
    :param max_steps: the maximum number of steps, defaults to `10`
    :return: A value `v` such that `v * v` is approximately `number`.

    >>> sqrt(Fraction(2, 1)).decimal_str(750)
    '1.414213562373095048801688724209698078569671875376948073176679379\
907324784621070388503875343276415727350138462309122970249248360558507372\
21264412149709993583141322266592750559275579995050115278206057147010955\
99716059702745345968620147285174186408891986095523292304843087143214508\
39762603627995251407989687253396546331808829640620615258352395054745750\
28775996172983557522033753185701135437460340849884716038689997069900481\
50305440277903164542478230684929369186215805784631115966687130130156185\
68987237235288509264861249497715421833420428568606014682472077143585487\
41556570696776537202264854470158588016207584749226572260020855844665214\
58398893944370926591800311388246468157082630100594858704003186480342194\
89727829064104507263688131373985525611732204025'
    >>> sqrt(Fraction(4, 1)).decimal_str()
    '2'
    >>> (ONE_HALF * (ONE + sqrt(Fraction(5, 1)))).decimal_str(420)
    '1.618033988749894848204586834365638117720309179805762862135448627\
052604628189024497072072041893911374847540880753868917521266338622235361\
93179318006076672635443338908659593958290563832266131992829026788067520\
87668925017116962070322210432162695486262963136144381497587012203408058\
87954454749246185695364864449241044320771344947049565846788509874339442\
21254487706647809158846074998871240076521705751797883416625624940758907'
    """
    if number < ZERO:  # No negative numbers are permitted.
        raise ArithmeticError(f"Cannot computed sqrt({number}).")
    guess: Fraction = ONE        # This will hold the current guess.
    old_guess: Fraction = ZERO  # 0.0 is just a dummy value != guess.
    while old_guess != guess:  # Repeat until nothing changes anymore.
        old_guess = guess  # The current guess becomes the old guess.
        guess = ONE_HALF * (guess + number / guess)  # The new guess.
        max_steps -= 1       # Reduce the number of remaining steps.
        if max_steps <= 0:  # If we have exhausted the maximum steps...
            break           # ...then we stop (and return the guess).
    return guess  # Return the final guess.
```

Fraction: sqrt

- Mit `Fraction` kann er selbe Kode aber unendlich lange iterieren.
- Wir haben ja eine Genauigkeit, die nur durch den zur Verfügung stehenden Arbeitsspeicher begrenzt wird.
- Der Kode könnte fast für immer nach besseren Annäherungen suchen.
- Wir müssen also die Anzahl der Iterationen mit einem weiteren Parameter begrenzen.
- Wir fügen also den neuen Parameter `max_steps: int = 10` hinzu.

```python
"""A square root algorithm based on fractions."""

from fraction import ONE, ONE_HALF, ZERO, Fraction


def sqrt(number: Fraction, max_steps: int = 10) -> Fraction:
    """
    Compute the square root of a given :class:`Fraction`.

    :param number: The rational number to compute the square root of.
    :param max_steps: the maximum number of steps, defaults to `10`
    :return: A value `v` such that `v * v` is approximately `number`.

    >>> sqrt(Fraction(2, 1)).decimal_str(750)
    '1.414213562373095048801688724209698078569671875376948073176679379\
907324784621070388503875343276415727350138462309122970249248360558507372\
21264412149709993583141322266592750559275579995050115278206057147010955\
99716059702745345968620147285174186408891986095523292304843087143214508\
39762603627995251407989687253396546331808829640620615258352395054745750\
28775996172983557522033753185701135437460340849884716038689997069900481\
50305440277903164542478230684929369186215805784631115966687130130156185\
68987237235288509264861249497715421833420428568606014682472077143585487\
41556570696776537202264854470158588016207584749226572260020855844665214\
58398893944370926591800311388246468157082630100594858870400318648034219\
4\
89727829064104507263688131373985525611732204025'
    >>> sqrt(Fraction(4, 1)).decimal_str()
    '2'
    >>> (ONE_HALF * (ONE + sqrt(Fraction(5, 1)))).decimal_str(420)
    '1.6180339887498948482045868343656381177203091798057628621354486227\
052604628189024497072072041893911374847540880753868917521266338622235367\
931793180060766726354433389808659593958290563832266131992829026788067520\
876689250171169620703222104321626954862629631361443814975870122030408058\
879544547492461856953648644492410443207713449470495658467885098743399442\
21254487706647809158846074998871240076521705751797883416625624940758907'
    """
    if number < ZERO:  # No negative numbers are permitted.
        raise ArithmeticError(f"Cannot computed sqrt({number}).")
    guess: Fraction = ONE        # This will hold the current guess.
    old_guess: Fraction = ZERO  # 0.0 is just a dummy value != guess.
    while old_guess != guess:  # Repeat until nothing changes anymore.
        old_guess = guess  # The current guess becomes the old guess.
        guess = ONE_HALF * (guess + number / guess)  # The new guess.
        max_steps -= 1       # Reduce the number of remaining steps.
        if max_steps <= 0:  # If we have exhausted the maximum steps...
            break           # ...then we stop (and return the guess).
    return guess  # Return the final guess.
```

Fraction: sqrt

- Wir haben ja eine Genauigkeit, die nur durch den zur Verfügung stehenden Arbeitsspeicher begrenzt wird.
- Der Kode könnte fast für immer nach besseren Annäherungen suchen.
- Wir müssen also die Anzahl der Iterationen mit einem weiteren Parameter begrenzen.
- Wir fügen also den neuen Parameter `max_steps: int = 10` hinzu.
- Wir werden später sehen, dass der Default-Wert, der nur 10 Iterationen zulässt, bereits ziemlich gute Annäherungen erlaubt.

```python
"""A square root algorithm based on fractions."""

from fraction import ONE, ONE_HALF, ZERO, Fraction


def sqrt(number: Fraction, max_steps: int = 10) -> Fraction:
    """
    Compute the square root of a given :class:`Fraction`.

    :param number: The rational number to compute the square root of.
    :param max_steps: the maximum number of steps, defaults to `10`
    :return: A value `v` such that `v * v` is approximately `number`.

    >>> sqrt(Fraction(2, 1)).decimal_str(750)
    '1.41421356237309504880168872420969807856967187537694807317667973799\
5073247846210703885038753432764157273501384623091229702492483605585073\
7212644121497099935831413222665927505592755799950501152782060571470109\
9971605970274534596862014728517418640889198609552329230484308714321450\
8397626036279952514079896872533965463318088296406206152583523950547457\
2877599617298355752203375318570113543746034048498847160386899970690040\
5030544027790316454247823068492936918621580578463111596668713013015618\
6898723723528850926486124949771542183342042856860601468247207714358548\
7\
4155657069677653720226485447015858801620758474922657226002085584466521\
4\
5839889394437092659180031138824646815708263010059485870400318648034219\
4\
8972782906410450726368813137398552561173220405\
    '
    >>> sqrt(Fraction(4, 1)).decimal_str()
    '2'
    >>> (ONE_HALF * (ONE + sqrt(Fraction(5, 1)))).decimal_str(420)
    '1.6180339887498948482045868343656381177203091798057628621354486227\
6\
0526046281890244970720720418939111748475408807538689175212663386222353\
6\
9317931800607667263544333890865959395829056383226613199282902678806752\
0\
8766892501711696207032221043216269548626296313614438149758701220340805\
8\
8795445474924618569536486444924104432077134494704956584678850987433944\
2\
2125448770664780915884607499887124007652170575179788341662562494075890\
7'
    """
    if number < ZERO:  # No negative numbers are permitted.
        raise ArithmeticError(f"Cannot computed sqrt({number}).")
    guess: Fraction = ONE        # This will hold the current guess.
    old_guess: Fraction = ZERO  # 0.0 is just a dummy value != guess.
    while old_guess != guess:  # Repeat until nothing changes anymore.
        old_guess = guess  # The current guess becomes the old guess.
        guess = ONE_HALF * (guess + number / guess)  # The new guess.
        max_steps -= 1       # Reduce the number of remaining steps.
        if max_steps <= 0:  # If we have exhausted the maximum steps...
            break           # ...then we stop (and return the guess).
    return guess  # Return the final guess.
```

Fraction: sqrt

- Der Kode könnte fast für immer nach besseren Annäherungen suchen.
- Wir müssen also die Anzahl der Iterationen mit einem weiteren Parameter begrenzen.
- Wir fügen also den neuen Parameter `max_steps: int = 10` hinzu.
- Wir werden später sehen, dass der Default-Wert, der nur 10 Iterationen zulässt, bereits ziemlich gute Annäherungen erlaubt.
- Wir ersetzen auch die Zahlen `0.0`, `0.5` und `1.0` mit unseren Konstanten `ZERO`, `ONE_HALF`, and `ONE`.

```python
"""A square root algorithm based on fractions."""

from fraction import ONE, ONE_HALF, ZERO, Fraction


def sqrt(number: Fraction, max_steps: int = 10) -> Fraction:
    """
    Compute the square root of a given :class:`Fraction`.

    :param number: The rational number to compute the square root of.
    :param max_steps: the maximum number of steps, defaults to `10`
    :return: A value `v` such that `v * v` is approximately `number`.

    >>> sqrt(Fraction(2, 1)).decimal_str(750)
    '1.414213562373095048801688724209698078569671875376948073176679379\
907324784621070388503875343276415727350138462309122970249248360558507372\
21264412149709993583141322266592755055927557999505011527820605714701095\
99716059702745345968620147285174186408891986095523292304843087143214508\
39762603627995251407989687253396546331808829640620615258352395054745750\
28775996172983557522033753185701135437460340849884716038689997069900481\
50305440277903164542478230684929369186215805784631115966687130130156185\
68987237235288509264861249497715421833420428568606014682472077143585487\
41556570696776537202264854470158588016207584749226572260020855844665214\
58398893944370926591800311388246468157082630100594858870400318648034219\
4\
89727829064104507263688131373985525611732204025'
    >>> sqrt(Fraction(4, 1)).decimal_str()
    '2'
    >>> (ONE_HALF * (ONE + sqrt(Fraction(5, 1)))).decimal_str(420)
    '1.618033988749894848204586834365638117720309179805762862135448627\
052604628189024497072072041893911374847540880753868917521266338622235\
931793180060766726354433389086595939582905638322661319928290267880675\
876689250171169620703222104321626954862629631361443814975870122034080\
879544547492461856953648644492410443207713449470495658467885098743394\
21254487706647809158846074998871240076521705751797883416625624940758907'
    """
    if number < ZERO:  # No negative numbers are permitted.
        raise ArithmeticError(f"Cannot computed sqrt({number}).")
    guess: Fraction = ONE       # This will hold the current guess.
    old_guess: Fraction = ZERO  # 0.0 is just a dummy value != guess.
    while old_guess != guess:  # Repeat until nothing changes anymore.
        old_guess = guess  # The current guess becomes the old guess.
        guess = ONE_HALF * (guess + number / guess)  # The new guess.
        max_steps -= 1      # Reduce the number of remaining steps.
        if max_steps <= 0:  # If we have exhausted the maximum steps...
            break           # ...then we stop (and return the guess).
    return guess  # Return the final guess.
```

Fraction: sqrt

- Wir müssen also die Anzahl der Iterationen mit einem weiteren Parameter begrenzen.
- Wir fügen also den neuen Parameter `max_steps: int = 10` hinzu.
- Wir werden später sehen, dass der Default-Wert, der nur 10 Iterationen zulässt, bereits ziemlich gute Annäherungen erlaubt.
- Wir ersetzen auch die Zahlen `0.0`, `0.5` und `1.0` mit unseren Konstanten `ZERO`, `ONE_HALF`, and `ONE`.
- Das müssen wir machen, denn unsere Dunder-Methoden funktionieren nur mit Instanzen von `Fraction`.

```python
"""A square root algorithm based on fractions."""

from fraction import ONE, ONE_HALF, ZERO, Fraction


def sqrt(number: Fraction, max_steps: int = 10) -> Fraction:
    """
    Compute the square root of a given :class:`Fraction`.

    :param number: The rational number to compute the square root of.
    :param max_steps: the maximum number of steps, defaults to `10`
    :return: A value `v` such that `v * v` is approximately `number`.

    >>> sqrt(Fraction(2, 1)).decimal_str(750)
    '1.414213562373095048801688724209698078569671875376948073176679379\
907324784621070388503875343276415727350138462309122970249248360558507372\
21264412149709993583141322266592750559275579995050115278206057147010955\
99716059702745345968620147285174186408891986095523292304843087143214508\
39762603627995251407989687253396546331808829640620615258352395054745750\
28775996172983557522033753185701113543746034084988471603868999706990048 1\
50305440277903164542478230684929369186215805784631115966687130130156185\
68987237235288509264861249497715421833420428568606014682472077143585487\
41556570696776537202264854470158588016207584749226572260020855844665214\
58398893944370926591800311388246468157082630100594858704003186480342194\
89727829064104507263688131373985525611732204025'
    >>> sqrt(Fraction(4, 1)).decimal_str()
    '2'
    >>> (ONE_HALF * (ONE + sqrt(Fraction(5, 1)))).decimal_str(420)
    '1.618033988749894848204586834365638117720309179805762862135448627\
052604628189024497072072041893911374847540880753868917521266338622233536\
931793180060766726354433389086595939582905638322661319928290267880675 20\
876689250171169620703222104321626954862629631361443814975870122034080 58\
879544547492461856953648644492410443207713449470495658467885098743399442\
21254487706647809158846074998871240076521705751797883416625624940758907'
    """
    if number < ZERO:  # No negative numbers are permitted.
        raise ArithmeticError(f"Cannot computed sqrt({number}).")
    guess: Fraction = ONE       # This will hold the current guess.
    old_guess: Fraction = ZERO  # 0.0 is just a dummy value != guess.
    while old_guess != guess:  # Repeat until nothing changes anymore.
        old_guess = guess  # The current guess becomes the old guess.
        guess = ONE_HALF * (guess + number / guess)  # The new guess.
        max_steps -= 1      # Reduce the number of remaining steps.
        if max_steps <= 0:  # If we have exhausted the maximum steps...
            break           # ...then we stop (and return the guess).
    return guess  # Return the final guess.
```

Fraction: sqrt

- Wir fügen also den neuen Parameter `max_steps: int = 10` hinzu.
- Wir werden später sehen, dass der Default-Wert, der nur 10 Iterationen zulässt, bereits ziemlich gute Annäherungen erlaubt.
- Wir ersetzen auch die Zahlen `0.0`, `0.5` und `1.0` mit unseren Konstanten `ZERO`, `ONE_HALF`, and `ONE`.
- Das müssen wir machen, denn unsere Dunder-Methoden funktionieren nur mit Instanzen von `Fraction`.
- Wir hätten sie so implementieren können, dass sie auch mit `int` oder `float` funktionieren...

```python
"""A square root algorithm based on fractions."""

from fraction import ONE, ONE_HALF, ZERO, Fraction


def sqrt(number: Fraction, max_steps: int = 10) -> Fraction:
    """
    Compute the square root of a given :class:`Fraction`.

    :param number: The rational number to compute the square root of.
    :param max_steps: the maximum number of steps, defaults to `10`
    :return: A value `v` such that `v * v` is approximately `number`.

    >>> sqrt(Fraction(2, 1)).decimal_str(750)
    '1.414213562373095048801688724209698078569671875376948073176679379\
907324784621070388503875343276415727350138462309122970249248360558507372\
21264412149709993583141322266592750559275579995050115278206057147010955\
997160597027453459686201472851741864088919860955232923048430871432145 08\
397626036279952514079896872533965463318088296406206152583523950547457 50\
287759961729835575220337531857011354374603404898847160386899970699004 81\
503054402779031645424782306849293691862158057846311159666871301301561 85\
689872372352885092648612494977154218334204285686060146824720771435858 7\
415565706967765372022648544701585880162075847492265722600208558446665 214\
583988939443709265918003113882464681570826301005948587040031864803421 94\
897278290641045072636881313739855256117320 4025'
    >>> sqrt(Fraction(4, 1)).decimal_str()
    '2'
    >>> (ONE_HALF * (ONE + sqrt(Fraction(5, 1)))).decimal_str(420)
    '1.618033988749894848204586834365638117720309179805762862135448622 7\
052604628189024497072072041893911374847540880753868917521266338622235 36\
931793180060766726354433389086595939582905638322661319928290267880 0752 0\
876689250171169620703222104321626954862629631361443814975870122034080 58\
879544547492461856953648644492410443207713449470495658467885098743394 42\
212544877066478091588460749988712400765217057517978834166256249407585 07'
    """
    if number < ZERO:  # No negative numbers are permitted.
        raise ArithmeticError(f"Cannot computed sqrt({number}).")
    guess: Fraction = ONE        # This will hold the current guess.
    old_guess: Fraction = ZERO  # 0.0 is just a dummy value != guess.
    while old_guess != guess:  # Repeat until nothing changes anymore.
        old_guess = guess  # The current guess becomes the old guess.
        guess = ONE_HALF * (guess + number / guess)  # The new guess.
        max_steps -= 1       # Reduce the number of remaining steps.
        if max_steps <= 0:  # If we have exhausted the maximum steps...
            break            # ...then we stop (and return the guess).
    return guess  # Return the final guess.
```

Fraction: sqrt

- Wir werden später sehen, dass der Default-Wert, der nur 10 Iterationen zulässt, bereits ziemlich gute Annäherungen erlaubt.
- Wir ersetzen auch die Zahlen `0.0`, `0.5` und `1.0` mit unseren Konstanten `ZERO`, `ONE_HALF`, and `ONE`.
- Das müssen wir machen, denn unsere Dunder-Methoden funktionieren nur mit Instanzen von `Fraction`.
- Wir hätten sie so implementieren können, dass sie auch mit `int` oder `float` funktionieren...
- Das haben wir nicht gemacht, weil sonst unser Beispielkode aber viel länger geworden wäre.

```
"""A square root algorithm based on fractions."""

from fraction import ONE, ONE_HALF, ZERO, Fraction


def sqrt(number: Fraction, max_steps: int = 10) -> Fraction:
    """
    Compute the square root of a given :class:`Fraction`.

    :param number: The rational number to compute the square root of.
    :param max_steps: the maximum number of steps, defaults to `10`
    :return: A value `v` such that `v * v` is approximately `number`.

    >>> sqrt(Fraction(2, 1)).decimal_str(750)
    '1.414213562373095048801688724209698078569671875376948073176679379\
907324784621070388503875343276415727350138462309122970249248360558507372\
21264412149709993583141322266592750559275579995050115278206057147010955\
997160597027453459686201472851741864088919860955232923048430871432141508\
397626036279952514079896872533965463318088296406206152583523950547457 50\
287759961729835575220337531857011354374603408498847160386899970699004 81\
503054402779031645424782306849293691862158057846311159666871301301561 85\
689872372352885092648612494977154218334202428568606014682472077143585 487\
415565706967765372022648544701585880162075847492265722600208558446665 214\
583988939443709265918003113882464681570826301005948587040031864803421 94\
897278290641045072636881313739855256117320 4025'
    >>> sqrt(Fraction(4, 1)).decimal_str()
    '2'
    >>> (ONE_HALF * (ONE + sqrt(Fraction(5, 1)))).decimal_str(420)
    '1.618033988749894848204586834365638117720309179805762862135448627\
052604628189024497072072041893911374847540880753868917521266336622235 36\
931793180060766726354433389080659593958290563832266131992829026788067520\
876689250171169620703222104321626954862629631361443814975870122034080 58\
879544547492461856953648644492410443207713449470495658467885098743394 42\
212544877066478091588460749988712400765217057517978834166256249407585 07'
    """
    if number < ZERO:  # No negative numbers are permitted.
        raise ArithmeticError(f"Cannot computed sqrt({number}).")
    guess: Fraction = ONE        # This will hold the current guess.
    old_guess: Fraction = ZERO  # 0.0 is just a dummy value != guess.
    while old_guess != guess:  # Repeat until nothing changes anymore.
        old_guess = guess  # The current guess becomes the old guess.
        guess = ONE_HALF * (guess + number / guess)  # The new guess.
        max_steps -= 1       # Reduce the number of remaining steps.
        if max_steps <= 0:  # If we have exhausted the maximum steps...
            break           # ...then we stop (and return the guess).
    return guess  # Return the final guess.
```

Fraction: sqrt

- Wir ersetzen auch die Zahlen `0.0`, `0.5` und `1.0` mit unseren Konstanten `ZERO`, `ONE_HALF`, and `ONE`.
- Das müssen wir machen, denn unsere Dunder-Methoden funktionieren nur mit Instanzen von `Fraction`.
- Wir hätten sie so implementieren können, dass sie auch mit `int` oder `float` funktionieren...
- Das haben wir nicht gemacht, weil sonst unser Beispielkode aber viel länger geworden wäre.
- So oder so, die numerischen Konstanten in unserer Funktion sind nun Instanzen von `Fraction`.

```
"""A square root algorithm based on fractions."""

from fraction import ONE, ONE_HALF, ZERO, Fraction


def sqrt(number: Fraction, max_steps: int = 10) -> Fraction:
    """
    Compute the square root of a given :class:`Fraction`.

    :param number: The rational number to compute the square root of.
    :param max_steps: the maximum number of steps, defaults to `10`
    :return: A value `v` such that `v * v` is approximately `number`.

    >>> sqrt(Fraction(2, 1)).decimal_str(750)
    '1.414213562373095048801688724209698078569671875376948073176679379\
907324784621070388503875343276415727350138462309122970249248360558507372\
21264412149709993583141322266592750559275579995050115278206057147010955\
99716059702745345968620147285174186408891986095523292304843087143214508\
39762603627995251407989687253396546331808829640620615258352395054745750\
28775596172935557522033753185701135437460340848988471603868999706990048 1\
50305440277903164542478230684929369186215805784631115966687130130156185\
68987237235288509264861249497715421833420428568606014682472077143585487\
41556570696776537202264854470158588016207584749226572260020855844665214\
58398893944370926591800311388246468157082630100594858870400318648034219 4\
89727829064104507263688131373985525611732204025'
    >>> sqrt(Fraction(4, 1)).decimal_str()
    '2'
    >>> (ONE_HALF * (ONE + sqrt(Fraction(5, 1)))).decimal_str(420)
    '1.6180339887498948482045868343656381177203091798057628621354486227\
052604628189024497072072041893911374847540880753868917521266338622233536\
931793180060766726354433389808659593958290563832266131992829026788067520\
876689250171169620703222104321626954862629631361443814975870122034808058\
879544547492461856953648644492410443207713449470495658467885098743394 42\
212544877066478091588460749988712400765217057517978834166256249407585907'
    """
    if number < ZERO:  # No negative numbers are permitted.
        raise ArithmeticError(f"Cannot computed sqrt({number}).")
    guess: Fraction = ONE        # This will hold the current guess.
    old_guess: Fraction = ZERO  # 0.0 is just a dummy value != guess.
    while old_guess != guess:  # Repeat until nothing changes anymore.
        old_guess = guess  # The current guess becomes the old guess.
        guess = ONE_HALF * (guess + number / guess)  # The new guess.
        max_steps -= 1       # Reduce the number of remaining steps.
        if max_steps <= 0:  # If we have exhausted the maximum steps...
            break           # ...then we stop (and return the guess).
    return guess  # Return the final guess.
```

Fraction: sqrt

- Das müssen wir machen, denn unsere Dunder-Methoden funktionieren nur mit Instanzen von `Fraction`.
- Wir hätten sie so implementieren können, dass sie auch mit `int` oder `float` funktionieren...
- Das haben wir nicht gemacht, weil sonst unser Beispielkode aber viel länger geworden wäre.
- So oder so, die numerischen Konstanten in unserer Funktion sind nun Instanzen von `Fraction`.
- Unsere neue `sqrt`-Funktion beginnt damit, zu prüfen ob die Eingabezahl negativ ist.

```python
"""A square root algorithm based on fractions."""

from fraction import ONE, ONE_HALF, ZERO, Fraction


def sqrt(number: Fraction, max_steps: int = 10) -> Fraction:
    """
    Compute the square root of a given :class:`Fraction`.

    :param number: The rational number to compute the square root of.
    :param max_steps: the maximum number of steps, defaults to `10`
    :return: A value `v` such that `v * v` is approximately `number`.

    >>> sqrt(Fraction(2, 1)).decimal_str(750)
    '1.414213562373095048801688724209698078569671875376948073176679379\
907324784621070388503875343276415727350138462309122970249248360558507372\
21264412149709993583141322266592750559275579995050115278206057147010955\
997160597027453459686201472851741864088919860955232923048430871432145085\
397626036279952514079896872533965463318088296406206152583523950547457505\
287759961729835575220337531857011354374603408498847160386899970690900481\
503054402779031645424782306849293691862158057846311159666871301301561855\
689872372352885092648612494977154218334202485686060146824720771435854875\
415565706967765372022648544701585880162075847492265722600208558446652145\
583988939443709265918003113882464681570826301005948587040031864803421945\
897278290641045072636881313739855256117320240 25'
    >>> sqrt(Fraction(4, 1)).decimal_str()
    '2'
    >>> (ONE_HALF * (ONE + sqrt(Fraction(5, 1)))).decimal_str(420)
    '1.618033988749894848204586834365638117720309179805762862135448622\
705260462818902449707207204189391137484754088075386891752126633862223536\
931793180060766726354433389808659593958290563832266131992829026788067520\
876689250171696207032221042316269548626296313614438149758701220340805 8\
879544547492461856953648644492410443207713449470495658467885098743399442\
212544877066478091588460749988712400765217057517978834166256249407585907'
    """
    if number < ZERO:  # No negative numbers are permitted.
        raise ArithmeticError(f"Cannot computed sqrt({number}).")
    guess: Fraction = ONE        # This will hold the current guess.
    old_guess: Fraction = ZERO  # 0.0 is just a dummy value != guess.
    while old_guess != guess:  # Repeat until nothing changes anymore.
        old_guess = guess  # The current guess becomes the old guess.
        guess = ONE_HALF * (guess + number / guess)  # The new guess.
        max_steps -= 1       # Reduce the number of remaining steps.
        if max_steps <= 0:  # If we have exhausted the maximum steps...
            break            # ...then we stop (and return the guess).
    return guess  # Return the final guess.
```

Fraction: sqrt

- Das haben wir nicht gemacht, weil sonst unser Beispielkode aber viel länger geworden wäre.
- So oder so, die numerischen Konstanten in unserer Funktion sind nun Instanzen von `Fraction`.
- Unsere neue `sqrt`-Funktion beginnt damit, zu prüfen ob die Eingabezahl negativ ist.
- Wenn ja, dann löst sie einen `ArithmeticError` aus.
- Danach hat sie fast die gleiche Schleife wie unsere originale Funktion.

```python
"""A square root algorithm based on fractions."""

from fraction import ONE, ONE_HALF, ZERO, Fraction


def sqrt(number: Fraction, max_steps: int = 10) -> Fraction:
    """
    Compute the square root of a given :class:`Fraction`.

    :param number: The rational number to compute the square root of.
    :param max_steps: the maximum number of steps, defaults to `10`
    :return: A value `v` such that `v * v` is approximately `number`.

    >>> sqrt(Fraction(2, 1)).decimal_str(750)
    '1.414213562373095048801688724209698078569671875376948073176679379\
907324784621070388503875343276415727350138462309122970249248360558507372\
21264412149709993583141322266592750559275579995050115278206057147010955\
997160597027453459686201472851741864088919860955232923048430871432145 08\
397626036279952514079896872533965463318088296406206152583523950547457 50\
287759961729835575220337531857011354374603408498847160386899970699004 81\
503054402779031645424782306849293691862158057846311159666871301301561 85\
689872372352885092648612494977154218334204285686060146824720771435854 87\
415565706967765372022648544701585880162075847492265722600020855844665214\
583988939443709265918003113882464681570826301005948587040031864803421 94\
897278290641045072636881313739855256117320 4025'
    >>> sqrt(Fraction(4, 1)).decimal_str()
    '2'
    >>> (ONE_HALF * (ONE + sqrt(Fraction(5, 1)))).decimal_str(420)
    '1.618033988749894848204586834365638117720309179805762862135448627\
052604628189024497072072041893911374847540880753868917521266338622235 36\
931793180060766726354433389808659593958290563832266131992829026788067520\
876689250171169620703222104321626954862629631361443814975870122034080 58\
879544547492461856953648644492410443207713449470495658467885098743394 42\
212544877066478091588460749988712400765217057517978834166256249407585 07'
    """
    if number < ZERO:  # No negative numbers are permitted.
        raise ArithmeticError(f"Cannot computed sqrt({number}).")
    guess: Fraction = ONE        # This will hold the current guess.
    old_guess: Fraction = ZERO  # 0.0 is just a dummy value != guess.
    while old_guess != guess:  # Repeat until nothing changes anymore.
        old_guess = guess  # The current guess becomes the old guess.
        guess = ONE_HALF * (guess + number / guess)  # The new guess.
        max_steps -= 1       # Reduce the number of remaining steps.
        if max_steps <= 0:  # If we have exhausted the maximum steps...
            break            # ...then we stop (and return the guess).
    return guess  # Return the final guess.
```

Fraction: sqrt

- So oder so, die numerischen Konstanten in unserer Funktion sind nun Instanzen von `Fraction`.
- Unsere neue `sqrt`-Funktion beginnt damit, zu prüfen ob die Eingabezahl negativ ist.
- Wenn ja, dann löst sie einen `ArithmeticError` aus.
- Danach hat sie fast die gleiche Schleife wie unsere originale Funktion.
- Der Einzige Unterschied ist, dass wir `max_steps` in jedem Schritt verringern.

```python
"""A square root algorithm based on fractions."""

from fraction import ONE, ONE_HALF, ZERO, Fraction


def sqrt(number: Fraction, max_steps: int = 10) -> Fraction:
    """
    Compute the square root of a given :class:`Fraction`.

    :param number: The rational number to compute the square root of.
    :param max_steps: the maximum number of steps, defaults to `10`
    :return: A value `v` such that `v * v` is approximately `number`.

    >>> sqrt(Fraction(2, 1)).decimal_str(750)
    '1.414213562373095048801688724209698078569671875376948073176679379\
907324784621070388503875343276415727350138462309122970249248360558507372\
21264412149709993583141322266592750559275579995050115278206057147010955\
997160597027453459686201472851741864088919860955232923048430871432145 08\
397626036279952514079896872533965463318088296406206152583523950547457 50\
287759961729835575220337531857011354374603408498847160386899970690 0481\
503054402779031645424782306849293691862158057846311159666871301301561 85\
689872372352885092648612494977154218334202485686060146824720771435854 87\
415565706967765372022648544701585880162075847492265722600208558446652 14\
583988939443709265918003113882464681570826301005948587040031864803421 94\
897278290641045072636881313739855256117320 4025'
    >>> sqrt(Fraction(4, 1)).decimal_str()
    '2'
    >>> (ONE_HALF * (ONE + sqrt(Fraction(5, 1)))).decimal_str(420)
    '1.6180339887498948482045868343656381177203091798057628621354486227\
052604628189024497072072041893911374847540880753868917521266338622235 36\
931793180060766726354433389080659593958290563832266131992829026788067520\
876689250171169620703222104321626954862629631361443814975870122030 8058\
879544547492461856953648644492410443207713449470495658467885098743399442\
212544877066478091588460749988712400765217057517978834166256249407585 07'
    """
    if number < ZERO:  # No negative numbers are permitted.
        raise ArithmeticError(f"Cannot computed sqrt({number}).")
    guess: Fraction = ONE        # This will hold the current guess.
    old_guess: Fraction = ZERO  # 0.0 is just a dummy value != guess.
    while old_guess != guess:  # Repeat until nothing changes anymore.
        old_guess = guess  # The current guess becomes the old guess.
        guess = ONE_HALF * (guess + number / guess)  # The new guess.
        max_steps -= 1       # Reduce the number of remaining steps.
        if max_steps <= 0:  # If we have exhausted the maximum steps...
            break            # ...then we stop (and return the guess).
    return guess  # Return the final guess.
```

Fraction: sqrt

- So oder so, die numerischen Konstanten in unserer Funktion sind nun Instanzen von `Fraction`.
- Unsere neue `sqrt`-Funktion beginnt damit, zu prüfen ob die Eingabezahl negativ ist.
- Wenn ja, dann löst sie einen `ArithmeticError` aus.
- Danach hat sie fast die gleiche Schleife wie unsere originale Funktion.
- Der Einzige Unterschied ist, dass wir `max_steps` in jedem Schritt verringern.
- Wir brechen die Schleife mit `break` ab, wenn es `0` wird.

```python
"""A square root algorithm based on fractions."""

from fraction import ONE, ONE_HALF, ZERO, Fraction


def sqrt(number: Fraction, max_steps: int = 10) -> Fraction:
    """
    Compute the square root of a given :class:`Fraction`.

    :param number: The rational number to compute the square root of.
    :param max_steps: the maximum number of steps, defaults to `10`
    :return: A value `v` such that `v * v` is approximately `number`.

    >>> sqrt(Fraction(2, 1)).decimal_str(750)
    '1.414213562373095048801688724209698078569671875376948073176679379\
907324784621070388503875343276415727350138462309122970249248360558507372\
21264412149709993583141322266592750559275579995050115278206057147010955\
997160597027453459686201472851741864088919860955232923048430871432145 08\
397626036279952514079896872533965463318088296406206152583523950547457 50\
287759961729835575220337531857011354374603408498847160386899970699004 81\
503054402779031645424782306849293691862158057846311159666871301301561 85\
689872372352885092648612494977154218334202485686060146824720771435858 7\
415565706967765372022648544701585880162075847492265722600208558446652 14\
583988939443709265918003113882464681570826301005948587040031864803421 94\
897278290641045072636881313739855256117320 4025'
    >>> sqrt(Fraction(4, 1)).decimal_str()
    '2'
    >>> (ONE_HALF * (ONE + sqrt(Fraction(5, 1)))).decimal_str(420)
    '1.6180339887498948482045868343656381177203091798057628621354486227\
052604628189024497072072041893911374847540880753868917521266338622233536\
931793180060766726354433389086595939582905638322661319928290267880 67520\
876689250171169620703222104321626954862629631361443814975870122030 48058\
879544547492461856953648644492410443207713449470495658467885098743394 42\
212544877066478091588460749988712400765217057517978834166256249407585 907'
    """
    if number < ZERO:  # No negative numbers are permitted.
        raise ArithmeticError(f"Cannot computed sqrt({number}).")
    guess: Fraction = ONE       # This will hold the current guess.
    old_guess: Fraction = ZERO  # 0.0 is just a dummy value != guess.
    while old_guess != guess:  # Repeat until nothing changes anymore.
        old_guess = guess  # The current guess becomes the old guess.
        guess = ONE_HALF * (guess + number / guess)  # The new guess.
        max_steps -= 1      # Reduce the number of remaining steps.
        if max_steps <= 0:  # If we have exhausted the maximum steps...
            break           # ...then we stop (and return the guess).
    return guess  # Return the final guess.
```

`sqrt`: Doctests

- Natürlich machen wir wieder Doctests für unsere Funktion.

```
"""A square root algorithm based on fractions."""

from fraction import ONE, ONE_HALF, ZERO, Fraction


def sqrt(number: Fraction, max_steps: int = 10) -> Fraction:
    """
    Compute the square root of a given :class:`Fraction`.

    :param number: The rational number to compute the square root of.
    :param max_steps: the maximum number of steps, defaults to `10`
    :return: A value `v` such that `v * v` is approximately `number`.

    >>> sqrt(Fraction(2, 1)).decimal_str(750)
    '1.414213562373095048801688724209698078569671875376948073176679737\
90732478462107038850387534327641572735013846230912297024924836055850737\
21264412149709993583141322266592755059275579995050115278206057147010955\
99716059702745345968620147285174186408891986095523292304843087143214508\
39762603627995251407989687253396546331808829640626015258352395054745750\
28775996172983557522033753185701135437460340849884716038689997069900481\
50305440277903164542478230684929369186215805784631159666871301301561855\
68987237235288509264861249497715421833420428568606014682472077143585487\
41555670696776537202264854470158588016207584749226572260020855844665214\
58398893944370926591800311388246468157082630100594858704003186480342194\
89727829064104507263688131373985525611732204025'
    >>> sqrt(Fraction(4, 1)).decimal_str()
    '2'
    >>> (ONE_HALF * (ONE + sqrt(Fraction(5, 1)))).decimal_str(420)
    '1.618033988749894848204586834365638117720309179805762862135448622705\
26046281890244970720720418939113748475408807538689175212663386222353694\
93179318006076672635443338908659593958290563832266131992829026788067520\
87668925017116962070322210432162695486262963136144381497587012203408058\
87954454792461856953648644492410443207713449470495658467885098743394442\
21254487706647809158846074998871240076521705751797883416625624940758907'
    """
    if number < ZERO:  # No negative numbers are permitted.
        raise ArithmeticError(f"Cannot computed sqrt({number}).")
    guess: Fraction = ONE        # This will hold the current guess.
    old_guess: Fraction = ZERO  # 0.0 is just a dummy value != guess.
```

`sqrt`: Doctests

- Natürlich machen wir wieder Doctests für unsere Funktion.
- Wir testen zum Beispiel den Ausdruck `sqrt(Fraction(2, 1)).decimal_str(750)`.

```python
"""A square root algorithm based on fractions."""

from fraction import ONE, ONE_HALF, ZERO, Fraction


def sqrt(number: Fraction, max_steps: int = 10) -> Fraction:
    """
    Compute the square root of a given :class:`Fraction`.

    :param number: The rational number to compute the square root of.
    :param max_steps: the maximum number of steps, defaults to `10`
    :return: A value `v` such that `v * v` is approximately `number`.

    >>> sqrt(Fraction(2, 1)).decimal_str(750)
    '1.414213562373095048801688724209698078569671875376948073176679737\
90732478462107038850387534327641572735013846230912297024924836055850737\
21264412149709993583141322266592755059275579995050115278206057147010955\
99716059702745345968620147285174186408891986095523292304843087143214508\
39762603627995251407989687253396546331808829640626015258352395054745750\
28775996172983557522033753185701135437460340849884716038689997069900481\
50305440277903164542478230684929369186215805784631159666871301301561855\
68987237235288509264861249497715421833420428568606014682472077143585487\
41555670969776537202264854470158588016207584749226572260020855844665214\
58398893944370926591800311388246468157082630100594858704003186480342194\
89727829064104507263688131373985525611732204025'
    >>> sqrt(Fraction(4, 1)).decimal_str()
    '2'
    >>> (ONE_HALF * (ONE + sqrt(Fraction(5, 1)))).decimal_str(420)
    '1.61803398874989484820458683436563811772030917980576286213544862270\
52604628189024497070720720418939911374847540880753868917521266338622235536\
93179318006076672635443338908659593958290563832266131992829026788067520\
87668925017116962070322210432162695486262963136144381497587012203408058\
87954454749246185695364864449241044320771344947049565846788509874339442\
21254487706647809158846074998871240076521705751797883416625624940758907'
    """
    if number < ZERO:  # No negative numbers are permitted.
        raise ArithmeticError(f"Cannot computed sqrt({number}).")
    guess: Fraction = ONE        # This will hold the current guess.
    old_guess: Fraction = ZERO  # 0.0 is just a dummy value != guess.
```

`sqrt`: Doctests

- Natürlich machen wir wieder Doctests für unsere Funktion.
- Wir testen zum Beispiel den Ausdruck `sqrt(Fraction(2, 1)).decimal_str(750)`.
- Der Berechnet $\sqrt{\frac{2}{1}} = \sqrt{2}$ mit zehn Schritten von Heron's Algorithmus.

```python
"""A square root algorithm based on fractions."""

from fraction import ONE, ONE_HALF, ZERO, Fraction


def sqrt(number: Fraction, max_steps: int = 10) -> Fraction:
    """
    Compute the square root of a given :class:`Fraction`.

    :param number: The rational number to compute the square root of.
    :param max_steps: the maximum number of steps, defaults to `10`
    :return: A value `v` such that `v * v` is approximately `number`.

    >>> sqrt(Fraction(2, 1)).decimal_str(750)
    '1.414213562373095048801688724209698078569671875376948073176679737\
90732478462107038850387534327641572735013846230912297024924836055850737\
21264412149709993583141322266592750559275579995050115278206057147010955\
99716059702745345968620147285174186408891986095523292304843087143214508\
39762603627995251407989687253396546331808829640626015258352395054745750\
28775996172983557522033753185701135437460340849884716038689997069900481\
50305440277903164542478230684929369186215805784631115966687130130156185\
68987237235288509264861249497715421833420428568606014682472077143585487\
41555670696776537202264854470158588016207584749226572260020855844665214\
58398893944370926591800311388246468157082630100594858704003186480342194\
89727829064104507263688131373985525611732204025'
    >>> sqrt(Fraction(4, 1)).decimal_str()
    '2'
    >>> (ONE_HALF * (ONE + sqrt(Fraction(5, 1)))).decimal_str(420)
    '1.61803398874989484820458683436563811772030917980576286213544862270\
52604628189024497072072041893911374847540880753868917521266338622235360\
93179318006076672635443338908659593958290563832266131992829026788067520\
87668925017116962070322210432162695486262963136144381497587012203408058\
87954454749246185695364864449241044320771344947049565846788509874339442\
21254487706647809158846074998871240076521705751797883416625624940758907'
    """
    if number < ZERO:  # No negative numbers are permitted.
        raise ArithmeticError(f"Cannot computed sqrt({number}).")
    guess: Fraction = ONE        # This will hold the current guess.
    old_guess: Fraction = ZERO  # 0.0 is just a dummy value != guess.
```

`sqrt`: Doctests

- Natürlich machen wir wieder Doctests für unsere Funktion.
- Wir testen zum Beispiel den Ausdruck `sqrt(Fraction(2, 1)).decimal_str(750)`.
- Der Berechnet $\sqrt{\frac{2}{1}} = \sqrt{2}$ mit zehn Schritten von Heron's Algorithmus.
- Danach übersetzt er das Ergebnis zu einer Dezimalzahl mit 700 Nachkommastellen!

```
"""A square root algorithm based on fractions."""

from fraction import ONE, ONE_HALF, ZERO, Fraction


def sqrt(number: Fraction, max_steps: int = 10) -> Fraction:
    """
    Compute the square root of a given :class:`Fraction`.

    :param number: The rational number to compute the square root of.
    :param max_steps: the maximum number of steps, defaults to `10`
    :return: A value `v` such that `v * v` is approximately `number`.

    >>> sqrt(Fraction(2, 1)).decimal_str(750)
    '1.414213562373095048801688724209698078569671875376948073176679737\
90732478462107038850387534327641572735013846230912297024924836055850737\
21264412149709993583141322266592750559275579995050115278206057147010955\
99716059702745345968620147285174186408891986095523292304843087143214508\
39762603627995251407989687253396546331808829640626015258352395054745750\
28775996172983557522033753185701135437460340849884716038689997069900481\
50305440277903164542478230684929369186215805784631115966687130130156185\
68987237235288509264861249497715421833420428568606014682472077143585487\
41556570696776573202264854470158588016207584749226572260020855844665214\
58398893944370926591800311388246468157082630100594858704003186480342194\
89727829064104507263688131373985552611732204025'
    >>> sqrt(Fraction(4, 1)).decimal_str()
    '2'
    >>> (ONE_HALF * (ONE + sqrt(Fraction(5, 1)))).decimal_str(420)
    '1.6180339887498948482045868343656381177203091798057628621354486227\
05260462818902449707207204189391137484754088075386891752126633862223536\
93179318006076672635443338908659593958290563832266131992829026788067520\
87668925017116962070322210432162695486262963136144381497587012203408058\
87954454749246185695364864449241044320771344947049565846788509874339442\
21254487706647809158846074998871240076521705751797883416625624940758907'
    """
    if number < ZERO:  # No negative numbers are permitted.
        raise ArithmeticError(f"Cannot computed sqrt({number}).")
    guess: Fraction = ONE        # This will hold the current guess.
    old_guess: Fraction = ZERO  # 0.0 is just a dummy value != guess.
```

`sqrt`: Doctests

- Natürlich machen wir wieder Doctests für unsere Funktion.
- Wir testen zum Beispiel den Ausdruck `sqrt(Fraction(2, 1)).decimal_str(750)`.
- Der Berechnet $\sqrt{\frac{2}{1}} = \sqrt{2}$ mit zehn Schritten von Heron's Algorithmus.
- Danach übersetzt er das Ergebnis zu einer Dezimalzahl mit 700 Nachkommastellen!
- Wir holen uns den korrekten Wert von [57, 82].

```
"""A square root algorithm based on fractions."""

from fraction import ONE, ONE_HALF, ZERO, Fraction


def sqrt(number: Fraction, max_steps: int = 10) -> Fraction:
    """
    Compute the square root of a given :class:`Fraction`.

    :param number: The rational number to compute the square root of.
    :param max_steps: the maximum number of steps, defaults to `10`
    :return: A value `v` such that `v * v` is approximately `number`.

    >>> sqrt(Fraction(2, 1)).decimal_str(750)
    '1.414213562373095048801688724209698078569671875376948073176679737\
90732478462107038850387534327641572735013846230912297024924836055850737\
21264412149709993583141322266592750559275579995050115278206057147010955\
99716059702745345968620147285174186408891986095523292304843087143214508\
39762603627995251407989687253396546331808829640626015258352395054745750\
28775996172983557522033753185701135437460340849884716038689997069900481\
50305440277903164542478230684929369186215805784631115966687130130156185\
68987237235288509264861249497715421833420428568606014682472077143585487\
41556570696776573202264854470158588016207584749226572260020855844665214\
58398893944370926591800311388246468157082630100594858704003186480342194\
89727829064104507263688131373985552561732204025'
    >>> sqrt(Fraction(4, 1)).decimal_str()
    '2'
    >>> (ONE_HALF * (ONE + sqrt(Fraction(5, 1)))).decimal_str(420)
    '1.61803398874989484820458683436563811772030917980576286213544862270\
52604628189024497072072041893911374847540880753868917521266338622235369\
93179318006076672635443338908659593958290563832266131992829026788067520\
87668925017116962070322210432162695486262963136144381497587012203408058\
87954454749246185695366486444924104432077134494704956568467885098743394462\
21254487706647809158846074998871240076521705751797883416625624940758907'
    """
    if number < ZERO:  # No negative numbers are permitted.
        raise ArithmeticError(f"Cannot computed sqrt({number}).")
    guess: Fraction = ONE        # This will hold the current guess.
    old_guess: Fraction = ZERO  # 0.0 is just a dummy value != guess.
```

`sqrt`: Doctests

- Wir testen zum Beispiel den Ausdruck `sqrt(Fraction(2, 1)).decimal_str(750)`.
- Der Berechnet $\sqrt{\frac{2}{1}} = \sqrt{2}$ mit zehn Schritten von Heron's Algorithmus.
- Danach übersetzt er das Ergebnis zu einer Dezimalzahl mit 700 Nachkommastellen!
- Wir holen uns den korrekten Wert von [57, 82].
- Wenn unsere Funktion diese Zahl nach zehn Schritten nicht auf 700 Nachkommastellen genau liefert, wird dieser Doctest fehlschlagen!

```
"""A square root algorithm based on fractions."""

from fraction import ONE, ONE_HALF, ZERO, Fraction


def sqrt(number: Fraction, max_steps: int = 10) -> Fraction:
    """
    Compute the square root of a given :class:`Fraction`.

    :param number: The rational number to compute the square root of.
    :param max_steps: the maximum number of steps, defaults to `10`
    :return: A value `v` such that `v * v` is approximately `number`.

    >>> sqrt(Fraction(2, 1)).decimal_str(750)
    '1.414213562373095048801688724209698078569671875376948073176679737\
90732478462107038850387534327641572735013846230912297024924836055850737\
21264412149709993583141322266592750559275579995050115278206057147010955\
99716059702745345968620147285174186408891986095523292304843087143214508\
39762603627995251407989687253396546331808829640626015258352395054745750\
28775996172983557522033753185701135437460340849884716038689997069900481\
50305440277903164542478230684929369186215805784631115966687130130156185\
68987237235288509264861249497715421833420428568606014682472077143585487\
41556570696776537202264854470158588016207584749226572260020855844665214\
58398893944370926591800311388246468157082630100594858704003186480342194\
89727829064104507263688131373985525611732204025'
    >>> sqrt(Fraction(4, 1)).decimal_str()
    '2'
    >>> (ONE_HALF * (ONE + sqrt(Fraction(5, 1)))).decimal_str(420)
    '1.6180339887498948482045868343656381177203091798057628621354486227\
05260462818902449707207204189391137484754088075386891752126633862223536\
93179318006076672635443338908659593958290563832266131992829026788067520\
87668925017116962070322210432162695486262963136144381497587012203408058\
87954454749246185695364864449241044320771344947049565846788509874339442\
21254487706647809158846074998871240076521705751797883416625624940758907'
    """
    if number < ZERO:  # No negative numbers are permitted.
        raise ArithmeticError(f"Cannot computed sqrt({number}).")
    guess: Fraction = ONE        # This will hold the current guess.
    old_guess: Fraction = ZERO  # 0.0 is just a dummy value != guess.
```

`sqrt`: Doctests

- Der Berechnet $\sqrt{\frac{2}{1}} = \sqrt{2}$ mit zehn Schritten von Heron's Algorithmus.
- Danach übersetzt er das Ergebnis zu einer Dezimalzahl mit 700 Nachkommastellen!
- Wir holen uns den korrekten Wert von [57, 82].
- Wenn unsere Funktion diese Zahl nach zehn Schritten nicht auf 700 Nachkommastellen genau liefert, wird dieser Doctest fehlschlagen!
- Der zweite Doctest prüft einfach ob $\sqrt{4}$ als nach der `decimal_str`-Übersetzung als 2 ausgegeben wird.

```python
"""A square root algorithm based on fractions."""

from fraction import ONE, ONE_HALF, ZERO, Fraction


def sqrt(number: Fraction, max_steps: int = 10) -> Fraction:
    """
    Compute the square root of a given :class:`Fraction`.

    :param number: The rational number to compute the square root of.
    :param max_steps: the maximum number of steps, defaults to `10`
    :return: A value `v` such that `v * v` is approximately `number`.

    >>> sqrt(Fraction(2, 1)).decimal_str(750)
    '1.414213562373095048801688724209698078569671875376948073176679737\
90732478462107038850387534327641572735013846230912297024924836055850737\
21264412149709993583141322266592750559275579950501152782060571470109555\
99716059702745345968620147285174186408891986095523292304843087143214508\
39762603627995251407989687253396546331808829640626015258352395054745750\
28775996172983557522033753185701135437460340849884716038689997069900481\
50305440277903164542478230684929369186215805784631115966687130130156185\
68987237235288509264861249497715421833420428568606014682472077143585487\
41556570696776537202264854470158588016207584749226572260020855844665214\
58398893944370926591800311388246468157082630100594858704003186480342194\
89727829064104507263688131373985552561732204025'
    >>> sqrt(Fraction(4, 1)).decimal_str()
    '2'
    >>> (ONE_HALF * (ONE + sqrt(Fraction(5, 1)))).decimal_str(420)
    '1.618033988749894848204586834365638117720309179805762862135448622705\
26046281890244970720720418939111374847540880753868917521266338622235365\
93179318006076672635443338908659593958290563832266131992829026788067520\
87668925017116962070322210432162695486262963136144381497587012203408058\
87954454749246185695368644492410443207713449470495658467885094783394425\
21254487706647809158846074998871240076521705751797883416625624940758907'
    """
    if number < ZERO:  # No negative numbers are permitted.
        raise ArithmeticError(f"Cannot computed sqrt({number}).")
    guess: Fraction = ONE        # This will hold the current guess.
    old_guess: Fraction = ZERO  # 0.0 is just a dummy value != guess.
```

`sqrt`: Doctests

- Danach übersetzt er das Ergebnis zu einer Dezimalzahl mit 700 Nachkommastellen!
- Wir holen uns den korrekten Wert von [57, 82].
- Wenn unsere Funktion diese Zahl nach zehn Schritten nicht auf 700 Nachkommastellen genau liefert, wird dieser Doctest fehlschlagen!
- Der zweite Doctest prüft einfach ob $\sqrt{4}$ als nach der `decimal_str`-Übersetzung als 2 ausgegeben wird.
- Natürlich nähern wir die Quadratwurzel nur durch mehrere Schritte an.

```
"""A square root algorithm based on fractions."""

from fraction import ONE, ONE_HALF, ZERO, Fraction


def sqrt(number: Fraction, max_steps: int = 10) -> Fraction:
    """
    Compute the square root of a given :class:`Fraction`.

    :param number: The rational number to compute the square root of.
    :param max_steps: the maximum number of steps, defaults to `10`
    :return: A value `v` such that `v * v` is approximately `number`.

    >>> sqrt(Fraction(2, 1)).decimal_str(750)
    '1.414213562373095048801688724209698078569671875376948073176679737\
90732478462107038850387534327641572735013846230912297024924836055850737\
21264412149709993583141322266592750559275579995050115278206057147010955\
99716059702745345968620147285174186408891986095523292304843087143214508\
39762603627995251407989687253396546331808829640626015258352395054745750\
28775996172983557522033753185701135437460340849884716038689997069900481\
50305440277903164542478230684929369186215805784631115966687130130156185\
68987237235288509264861249497715421833420428568606014682472077143585487\
41556570696776537202264854470158588016207584749226572260020855844665214\
58398893944370926591800311388246468157082630100594858704003186480342194\
89727829064104507263688131373985525611732204025'
    >>> sqrt(Fraction(4, 1)).decimal_str()
    '2'
    >>> (ONE_HALF * (ONE + sqrt(Fraction(5, 1)))).decimal_str(420)
    '1.6180339887498948482045868343656381177203091798057628621354486227\
05260462818902449707207204189391137484754088075386891752126633862223536\
93179318006076672635443338908659593958290563832266131992829026788067520\
87668925017116962070322210432162695486262963136144381497587012203408058\
87954454749246185695368644492410443207713449470495658467885098743394442\
21254487706647809158846074998871240076521705751797883416625624940758907'
    """
    if number < ZERO:  # No negative numbers are permitted.
        raise ArithmeticError(f"Cannot computed sqrt({number}).")
    guess: Fraction = ONE        # This will hold the current guess.
    old_guess: Fraction = ZERO  # 0.0 is just a dummy value != guess.
```

`sqrt`: Doctests

- Wir holen uns den korrekten Wert von [57, 82].
- Wenn unsere Funktion diese Zahl nach zehn Schritten nicht auf 700 Nachkommastellen genau liefert, wird dieser Doctest fehlschlagen!
- Der zweite Doctest prüft einfach ob $\sqrt{4}$ als nach der `decimal_str`-Übersetzung als 2 ausgegeben wird.
- Natürlich nähern wir die Quadratwurzel nur durch mehrere Schritte an.
- Daher ist der echte Näherungswert vielleicht nicht wirklich 2.

```python
"""A square root algorithm based on fractions."""

from fraction import ONE, ONE_HALF, ZERO, Fraction


def sqrt(number: Fraction, max_steps: int = 10) -> Fraction:
    """
    Compute the square root of a given :class:`Fraction`.

    :param number: The rational number to compute the square root of.
    :param max_steps: the maximum number of steps, defaults to `10`
    :return: A value `v` such that `v * v` is approximately `number`.

    >>> sqrt(Fraction(2, 1)).decimal_str(750)
    '1.414213562373095048801688724209698078569671875376948073176679737\
90732478462107038850387534327641572735013846230912297024924836055850737\
21264412149709993583141322266592755592755799950501152782060571470109555\
99716059702745345968620147285174186408891986095523292304843087143214508\
39762603627995251407989687253396546331808829640626015258352395054745750\
28775996172983557522033753185701135437460340849884716038689997069900481\
50305440277903164542478230684929369186215805784631115966687130130156185\
68987237235288509264861249497715421833420428568606014682472077143585487\
41556570696776537202264854470158588016207584749226572260020855844665214\
58398893944370926591800311388246468157082630100594858704003186480342194\
89727829064104507263688131373985552561732204025'
    >>> sqrt(Fraction(4, 1)).decimal_str()
    '2'
    >>> (ONE_HALF * (ONE + sqrt(Fraction(5, 1)))).decimal_str(420)
    '1.6180339887498948482045868343656381177203091798057628621354486227\
05260462818902449707207204189391137484754088075386891752126633862223536\
93179318006076672635443338908659593958290563832266131992829026788067520\
87668925017116962070322210432162695486262963136144381497587012203408058\
87954454749246185695364864449241044320771344947049565846788509874339442\
21254487706647809158846074998871240076521705751797883416625624940758907'
    """
    if number < ZERO:  # No negative numbers are permitted.
        raise ArithmeticError(f"Cannot computed sqrt({number}).")
    guess: Fraction = ONE        # This will hold the current guess.
    old_guess: Fraction = ZERO  # 0.0 is just a dummy value != guess.
```

`sqrt`: Doctests

- Der zweite Doctest prüft einfach ob $\sqrt{4}$ als nach der `decimal_str`-Übersetzung als 2 ausgegeben wird.
- Natürlich nähern wir die Quadratwurzel nur durch mehrere Schritte an.
- Daher ist der echte Näherungswert vielleicht nicht wirklich 2.
- Wenn wir ihn auf 100 Nachkommastellen runden und ausgeben, dann sollte es uns "2" liefern.

```
"""A square root algorithm based on fractions."""

from fraction import ONE, ONE_HALF, ZERO, Fraction


def sqrt(number: Fraction, max_steps: int = 10) -> Fraction:
    """
    Compute the square root of a given :class:`Fraction`.

    :param number: The rational number to compute the square root of.
    :param max_steps: the maximum number of steps, defaults to `10`
    :return: A value `v` such that `v * v` is approximately `number`.

    >>> sqrt(Fraction(2, 1)).decimal_str(750)
    '1.414213562373095048801688724209698078569671875376948073176679737\
90732478462107038850387534327641572735013846230912297024924836055850737\
21264412149709993583141322266592750559275579950501152782060571470109555\
99716059702745345968620147285174186408891986095523292304843087143214508\
39762603627995251407989687253396546331808829640626015258352395054745750\
28775996172983557522033753185701135437460340849884716038689997069900481\
50305440277903164542478230684929369186215805784631115966687130130156185\
68987237235288509264861249497715421833420428568606014682472077143585487\
41556570696776537202264854470158588016207584749226572260020855844665214\
58398893944370926591800311388246468157082630100594858704003186480342194\
89727829064104507263688131373985552561732204025'
    >>> sqrt(Fraction(4, 1)).decimal_str()
    '2'
    >>> (ONE_HALF * (ONE + sqrt(Fraction(5, 1)))).decimal_str(420)
    '1.618033988749894848204586834365638117720309179805762862135448622705\
26046281890244970720720418939113748475408807538689175212663386222353693\
93179318006076672635443338908659593958290563832266131992829026788067520\
87668925017116962070322210432162695486262963136144381497587012203408058\
87954454749246185695364864449241044320771344947049565846788509874339442\
21254487706647809158846074998871240076521705751797883416625624940758907'
    """
    if number < ZERO:  # No negative numbers are permitted.
        raise ArithmeticError(f"Cannot computed sqrt({number}).")
    guess: Fraction = ONE        # This will hold the current guess.
    old_guess: Fraction = ZERO  # 0.0 is just a dummy value != guess.
```

`sqrt`: Doctests

- Der zweite Doctest prüft einfach ob $\sqrt{4}$ als nach der `decimal_str`-Übersetzung als 2 ausgegeben wird.
- Natürlich nähern wir die Quadratwurzel nur durch mehrere Schritte an.
- Daher ist der echte Näherungswert vielleicht nicht wirklich 2.
- Wenn wir ihn auf 100 Nachkommastellen runden und ausgeben, dann sollte es uns "2" liefern.
- Zu guter Letzt wollen wir den Goldenen Schnitt ϕ[15,30,81] berechnen.

```
"""A square root algorithm based on fractions."""

from fraction import ONE, ONE_HALF, ZERO, Fraction


def sqrt(number: Fraction, max_steps: int = 10) -> Fraction:
    """
    Compute the square root of a given :class:`Fraction`.

    :param number: The rational number to compute the square root of.
    :param max_steps: the maximum number of steps, defaults to `10`
    :return: A value `v` such that `v * v` is approximately `number`.

    >>> sqrt(Fraction(2, 1)).decimal_str(750)
    '1.414213562373095048801688724209698078569671875376948073176679737\
90732478462107038850387534327641572735013846230912297024924836055850737\
21264412149709993583141322266592750559275579950501152782060571470109555\
99716059702745345968620147285174186408891986095523292304843087143214508\
39762603627995251407989687253396546331808829640626015258352395054745750\
28775996172983557522033753185701135437460340849884716038689997069900481\
50305440277903164542478230684929369186215805784631115966687130130156185\
68987237235288509264861249497715421833420428568606014682472077143585487\
41556570696776537202264854470158588016207584749226572260020855844665214\
58398893944370926591800311388246468157082630100594858704003186480342194\
89727829064104507263688131373985552561732204025'
    >>> sqrt(Fraction(4, 1)).decimal_str()
    '2'
    >>> (ONE_HALF * (ONE + sqrt(Fraction(5, 1)))).decimal_str(420)
    '1.61803398874989484820458683436563811772030917980576286213544862270\
52604628189024497070720412893911374847540880753868917521266338622235369\
93179318006076672635443338908659593958290563832266131992829026788067520\
87668925017116962070322210432162695486262963136144381497587012203408058\
87954454749246185695368644492410443207713449470495658467885098743394423\
21254487706647809158846074998871240076521705751797883416625624940758907'
    """
    if number < ZERO:  # No negative numbers are permitted.
        raise ArithmeticError(f"Cannot computed sqrt({number}).")
    guess: Fraction = ONE        # This will hold the current guess.
    old_guess: Fraction = ZERO  # 0.0 is just a dummy value != guess.
```

`sqrt`: Doctests

- Natürlich nähern wir die Quadratwurzel nur durch mehrere Schritte an.
- Daher ist der echte Näherungswert vielleicht nicht wirklich 2.
- Wenn wir ihn auf 100 Nachkommastellen runden und ausgeben, dann sollte es uns "2" liefern.
- Zu guter Letzt wollen wir den Goldenen Schnitt $\phi^{15,30,81}$ berechnen.
- Dieser entspricht $\frac{1+\sqrt{5}}{2}$.

```python
"""A square root algorithm based on fractions."""

from fraction import ONE, ONE_HALF, ZERO, Fraction


def sqrt(number: Fraction, max_steps: int = 10) -> Fraction:
    """
    Compute the square root of a given :class:`Fraction`.

    :param number: The rational number to compute the square root of.
    :param max_steps: the maximum number of steps, defaults to `10`
    :return: A value `v` such that `v * v` is approximately `number`.

    >>> sqrt(Fraction(2, 1)).decimal_str(750)
    '1.414213562373095048801688724209698078569671875376948073176679737\
90732478462107038850387534327641572735013846230912297024924836055850737\
21264412149709993583141322266592750559275579950501152782060571470109555\
99716059702745345968620147285174186408891986095523292304843087143214508\
39762603627995251407989687253396546331808829640626015258352395054745750\
28775996172983557522033753185701135437460340849884716038689997069900481\
50305440277903164542478230684929369186215805784631115966687130130156185\
68987237235288509264861249497715421833420428568606014682472077143585487\
41556570696776537202264854470158588016207584749226572260020855844665214\
58398893944370926591800311388246468157082630100594858704003186480342194\
89727829064104507263688131373985552561732204025'
    >>> sqrt(Fraction(4, 1)).decimal_str()
    '2'
    >>> (ONE_HALF * (ONE + sqrt(Fraction(5, 1)))).decimal_str(420)
    '1.618033988749894848204586834365638117720309179805762862135448622705\
26046281890244970720720418939113748475408807538689175212663386222353616\
93179318006076672635443338908659593958290563832266131992829026788067520\
87668925017116962070322210432162695486262963136144381497587012203408058\
87954454749246185695366486444924104432077134494704956584678859087439344\
21254487706647809158846074998871240076521705751797883416625624940758907'
    """
    if number < ZERO:  # No negative numbers are permitted.
        raise ArithmeticError(f"Cannot computed sqrt({number}).")
    guess: Fraction = ONE        # This will hold the current guess.
    old_guess: Fraction = ZERO  # 0.0 is just a dummy value != guess.
```

`sqrt`: Doctests

- Daher ist der echte Näherungswert vielleicht nicht wirklich 2.
- Wenn wir ihn auf 100 Nachkommastellen runden und ausgeben, dann sollte es uns `"2"` liefern.
- Zu guter Letzt wollen wir den Goldenen Schnitt $\phi^{15,30,81}$ berechnen.
- Dieser entspricht $\frac{1+\sqrt{5}}{2}$.
- Wir können das als `ONE_HALF * (ONE + sqrt(Fraction(5, 1)))` schreiben.

```python
"""A square root algorithm based on fractions."""

from fraction import ONE, ONE_HALF, ZERO, Fraction


def sqrt(number: Fraction, max_steps: int = 10) -> Fraction:
    """
    Compute the square root of a given :class:`Fraction`.

    :param number: The rational number to compute the square root of.
    :param max_steps: the maximum number of steps, defaults to `10`
    :return: A value `v` such that `v * v` is approximately `number`.

    >>> sqrt(Fraction(2, 1)).decimal_str(750)
    '1.414213562373095048801688724209698078569671875376948073176679737\
90732478462107038850387534327641572735013846230912297024924836055850737\
21264412149709993583141322266592755059275579995050115278206057147010955\
99716059702745345968620147285174186408891986095523292304843087143214508\
39762603627995251407989687253396546331808829640626015258352395054745750\
28775996172983557522033753185701135437460340849884716038689997069900481\
50305440277903164542478230684929369186215805784631115966687130130156185\
68987237235288509264861249497715421833420428568606014682472077143585487\
41556570696776537202264854470158588016207584749226572260020855844665214\
58398893944370926591800311388246468157082630100594858704003186480342194\
89727829064104507263688131373985552561173220402 5'
    >>> sqrt(Fraction(4, 1)).decimal_str()
    '2'
    >>> (ONE_HALF * (ONE + sqrt(Fraction(5, 1)))).decimal_str(420)
    '1.618033988749894848204586834365638117720309179805762862135448622705\
26046281890244970720720418939113748475408807538689175212663386222353\
93179318006076672635443338908659593958290563832266131992829026788067520\
87668925017116962070322210432162695486262963136144381497587012203408058\
87954454749246185695364864449241044320771344947049565846788509874339442\
21254487706647809158846074998871240076521705751797883416625624940758907'
    """
    if number < ZERO:  # No negative numbers are permitted.
        raise ArithmeticError(f"Cannot computed sqrt({number}).")
    guess: Fraction = ONE        # This will hold the current guess.
    old_guess: Fraction = ZERO  # 0.0 is just a dummy value != guess.
```

`sqrt`: Doctests

- Wenn wir ihn auf 100 Nachkommastellen runden und ausgeben, dann sollte es uns `"2"` liefern.
- Zu guter Letzt wollen wir den Goldenen Schnitt $\phi^{15,30,81}$ berechnen.
- Dieser entspricht $\frac{1+\sqrt{5}}{2}$.
- Wir können das als `ONE_HALF * (ONE + sqrt(Fraction(5, 1)))` schreiben.
- Im Doctest erwarten wir, dass das Ergebnis auf 420 Nachkommastellen (die wir von [33] nehmen) genau ist.

```
"""A square root algorithm based on fractions."""

from fraction import ONE, ONE_HALF, ZERO, Fraction


def sqrt(number: Fraction, max_steps: int = 10) -> Fraction:
    """
    Compute the square root of a given :class:`Fraction`.

    :param number: The rational number to compute the square root of.
    :param max_steps: the maximum number of steps, defaults to `10`
    :return: A value `v` such that `v * v` is approximately `number`.

    >>> sqrt(Fraction(2, 1)).decimal_str(750)
    '1.414213562373095048801688724209698078569671875376948073176679737\
90732478462107038850387534327641572735013846230912297024924836055850737\
21264412149709993583141322266592750559275579995050115278206057147010955\
99716059702745345968620147285174186408891986095523292304843087143214508\
39762603627995251407989687253396546331808829640626015258352395054745750\
28775996172983557522033753185701135437460340849884716038689997069900481\
50305440277903164542478230684929369186215805784631115966687130130156185\
68987237235288509264861249497715421833420428568606014682472077143585487\
41556570696776537202264854470158588016207584749226572260020855844665214\
58398893944370926591800311388246468157082630100594858704003186480342194\
89727829064104507263688131373985552561732204025'
    >>> sqrt(Fraction(4, 1)).decimal_str()
    '2'
    >>> (ONE_HALF * (ONE + sqrt(Fraction(5, 1)))).decimal_str(420)
    '1.61803398874989484820458683436563811772030917980576286213544862270\
52604628189024497072072041893911374847540880753868917521266338622235366\
93179318006076672635443338908659593958290563832266131992829026788067520\
87668925017116962070322210432162695486262963136144381497587012203408058\
87954454749246185695368464449241044320771344947049565846788509874339442\
21254487706647809158846074998871240076521705751797883416625624940758907'
    """
    if number < ZERO:  # No negative numbers are permitted.
        raise ArithmeticError(f"Cannot computed sqrt({number}).")
    guess: Fraction = ONE        # This will hold the current guess.
    old_guess: Fraction = ZERO  # 0.0 is just a dummy value != guess.
```

`sqrt`: Doctests

- Zu guter Letzt wollen wir den Goldenen Schnitt ϕ[15,30,81] berechnen.
- Dieser entspricht $\frac{1+\sqrt{5}}{2}$.
- Wir können das als `ONE_HALF * (ONE + sqrt(Fraction(5, 1)))` schreiben.
- Im Doctest erwarten wir, dass das Ergebnis auf 420 Nachkommastellen (die wir von [33] nehmen) genau ist.
- Wir führen die Doctests mit pytest aus.

```python
"""A square root algorithm based on fractions."""

from fraction import ONE, ONE_HALF, ZERO, Fraction


def sqrt(number: Fraction, max_steps: int = 10) -> Fraction:
    """
    Compute the square root of a given :class:`Fraction`.

    :param number: The rational number to compute the square root of.
    :param max_steps: the maximum number of steps, defaults to `10`
    :return: A value `v` such that `v * v` is approximately `number`.

    >>> sqrt(Fraction(2, 1)).decimal_str(750)
    '1.414213562373095048801688724209698078569671875376948073176679737\
90732478462107038850387534327641572735013846230912297024924836055850737\
21264412149709993583141322266592750559275579995050115278206057147010955\
99716059702745345968620147285174186408891986095523292304843087143214508\
39762603627995251407989687253396546331808829640626015258352395054745750\
28775996172983557522033753185701135437460340849884716038689997069900481\
50305440277903164542478230684929369186215805784631115966687130130156185\
68987237235288509264861249497715421833420428568606014682472077143585487\
41556570696776537202264854470158588016207584749226572260020855844665214\
58398893944370926591800311388246468157082630100594858704003186480342194\
89727829064104507263688131373985552611732204025'
    >>> sqrt(Fraction(4, 1)).decimal_str()
    '2'
    >>> (ONE_HALF * (ONE + sqrt(Fraction(5, 1)))).decimal_str(420)
    '1.61803398874989484820458683436563811772030917980576286213544862270\
52604628189024497070720420189391137484754088075386891752126633862223536\
93179318006076672635443338908659593958290563832266131992829026788067520\
87668925017116962070322210432162695486262963136144381497587012203408058\
87954454749246185695368646449241044320771344947049556846788509874339442\
21254487706647809158846074998871240076521705751797883416625624940758907'
    """
    if number < ZERO:  # No negative numbers are permitted.
        raise ArithmeticError(f"Cannot computed sqrt({number}).")
    guess: Fraction = ONE       # This will hold the current guess.
    old_guess: Fraction = ZERO  # 0.0 is just a dummy value != guess.
```

`sqrt`: Doctests

- Dieser entspricht $\frac{1+\sqrt{5}}{2}$.
- Wir können das als `ONE_HALF * (ONE + sqrt(Fraction(5, 1)))` schreiben.
- Im Doctest erwarten wir, dass das Ergebnis auf 420 Nachkommastellen (die wir von [33] nehmen) genau ist.
- Wir führen die Doctests mit pytest aus.
- Sie sind alle erfolgreich!

```
1 $ pytest --timeout=10 --no-header --tb=short --doctest-modules
     ↪ fraction_sqrt.py
2 ========================== test session starts
     ↪ ==========================
3 collected 1 item
4 fraction_sqrt.py .                                                    [100%]
5 ========================== 1 passed in 0.01s
     ↪ ==========================
6 # pytest 9.2.0 with pytest-timeout 2.4.0 succeeded with exit code 0.
```

`sqrt`: Doctests

- Wir können das als `ONE_HALF * (ONE + sqrt(Fraction(5, 1)))` schreiben.
- Im Doctest erwarten wir, dass das Ergebnis auf 420 Nachkommastellen (die wir von [33] nehmen) genau ist.
- Wir führen die Doctests mit pytest aus.
- Sie sind alle erfolgreich!
- Wir haben nun Grundschulmathematik in eine Klasse in Python gegossen.

```
$ pytest --timeout=10 --no-header --tb=short --doctest-modules
   ↪ fraction_sqrt.py
========================== test session starts
   ↪ ==========================
collected 1 item
fraction_sqrt.py .                                                       [100%]
========================== 1 passed in 0.01s
   ↪ ==========================
# pytest 9.2.0 with pytest-timeout 2.4.0 succeeded with exit code 0.
```

`sqrt`: Doctests

- Im Doctest erwarten wir, dass das Ergebnis auf 420 Nachkommastellen (die wir von [33] nehmen) genau ist.
- Wir führen die Doctests mit pytest aus.
- Sie sind alle erfolgreich!
- Wir haben nun Grundschulmathematik in eine Klasse in Python gegossen.
- Dann haben wir die in einem 2000 Jahre alten Algorithmus verwendet.

```
$ pytest --timeout=10 --no-header --tb=short --doctest-modules
   ↪ fraction_sqrt.py
========================== test session starts
   ↪ ==========================
collected 1 item
fraction_sqrt.py .                                                       [100%]
========================== 1 passed in 0.01s
   ↪ ===========================
# pytest 9.2.0 with pytest-timeout 2.4.0 succeeded with exit code 0.
```

`sqrt`: Doctests

- Wir führen die Doctests mit pytest aus.
- Sie sind alle erfolgreich!
- Wir haben nun Grundschulmathematik in eine Klasse in Python gegossen.
- Dann haben wir die in einem 2000 Jahre alten Algorithmus verwendet.
- Und mit zehn Schritten des Algorithms konnten wir $\sqrt{2}$ und ϕ auf mehrere hundert Nachkommastellen genau berechnen.

```
$ pytest --timeout=10 --no-header --tb=short --doctest-modules
   ↪ fraction_sqrt.py
========================= test session starts
   ↪ =========================
collected 1 item
fraction_sqrt.py .                                                    [100%]
========================== 1 passed in 0.01s
   ↪ ==========================
# pytest 9.2.0 with pytest-timeout 2.4.0 succeeded with exit code 0.
```

`sqrt`: Doctests

- Sie sind alle erfolgreich!
- Wir haben nun Grundschulmathematik in eine Klasse in Python gegossen.
- Dann haben wir die in einem 2000 Jahre alten Algorithmus verwendet.
- Und mit zehn Schritten des Algorithms konnten wir $\sqrt{2}$ und ϕ auf mehrere hundert Nachkommastellen genau berechnen.
- Ist das nicht irrsinnig cool?

```
$ pytest --timeout=10 --no-header --tb=short --doctest-modules
   ↪ fraction_sqrt.py
========================= test session starts
   ↪ =========================
collected 1 item
fraction_sqrt.py .                                                    [100%]
========================= 1 passed in 0.01s
   ↪ =========================
# pytest 9.2.0 with pytest-timeout 2.4.0 succeeded with exit code 0.
```

Zusammenfassung

Zusammenfassung



- In dieser Einheit haben wir gelernt, wie wir einen Debugger benutzen können.

Zusammenfassung



- In dieser Einheit haben wir gelernt, wie wir einen Debugger benutzen können.
- Der Debugger ist eines der wichtigsten Werkzeuge im Werkzeuggürtel eines Programmierers.

Zusammenfassung



- In dieser Einheit haben wir gelernt, wie wir einen Debugger benutzen können.
- Der Debugger ist eines der wichtigsten Werkzeuge im Werkzeuggürtel eines Programmierers.
- Die Arbeitsweise von Kode wird klar, wenn wir ihn Schritt-für-Schritt ausführen können.

Zusammenfassung



- In dieser Einheit haben wir gelernt, wie wir einen Debugger benutzen können.
- Der Debugger ist eines der wichtigsten Werkzeuge im Werkzeuggürtel eines Programmierers.
- Die Arbeitsweise von Kode wird klar, wenn wir ihn Schritt-für-Schritt ausführen können.
- Wir können Breakpoints setzen und den Prozess laufen lassen, bis er bei ihnen ankommt.

Zusammenfassung



- In dieser Einheit haben wir gelernt, wie wir einen Debugger benutzen können.
- Der Debugger ist eines der wichtigsten Werkzeuge im Werkzeuggürtel eines Programmierers.
- Die Arbeitsweise von Kode wird klar, wenn wir ihn Schritt-für-Schritt ausführen können.
- Wir können Breakpoints setzen und den Prozess laufen lassen, bis er bei ihnen ankommt.
- Dann können wir uns die Werte von Variablen anschauen.

Zusammenfassung



- In dieser Einheit haben wir gelernt, wie wir einen Debugger benutzen können.
- Der Debugger ist eines der wichtigsten Werkzeuge im Werkzeuggürtel eines Programmierers.
- Die Arbeitsweise von Kode wird klar, wenn wir ihn Schritt-für-Schritt ausführen können.
- Wir können Breakpoints setzen und den Prozess laufen lassen, bis er bei ihnen ankommt.
- Dann können wir uns die Werte von Variablen anschauen.
- Dann können wir jede Zeile Kode einzeln ausführen.

Zusammenfassung



- In dieser Einheit haben wir gelernt, wie wir einen Debugger benutzen können.
- Der Debugger ist eines der wichtigsten Werkzeuge im Werkzeuggürtel eines Programmierers.
- Die Arbeitsweise von Kode wird klar, wenn wir ihn Schritt-für-Schritt ausführen können.
- Wir können Breakpoints setzen und den Prozess laufen lassen, bis er bei ihnen ankommt.
- Dann können wir uns die Werte von Variablen anschauen.
- Dann können wir jede Zeile Kode einzeln ausführen.
- Wir können sehen, wie sich jede einzelne Variable ändert.

Zusammenfassung



- In dieser Einheit haben wir gelernt, wie wir einen Debugger benutzen können.
- Der Debugger ist eines der wichtigsten Werkzeuge im Werkzeuggürtel eines Programmierers.
- Die Arbeitsweise von Kode wird klar, wenn wir ihn Schritt-für-Schritt ausführen können.
- Wir können Breakpoints setzen und den Prozess laufen lassen, bis er bei ihnen ankommt.
- Dann können wir uns die Werte von Variablen anschauen.
- Dann können wir jede Zeile Kode einzeln ausführen.
- Wir können sehen, wie sich jede einzelne Variable ändert.
- Wir können verfolgen, wie der Kontrollfluss durch die Zweige von Alternativen fließt und durch Schleifen iteriert.

Zusammenfassung

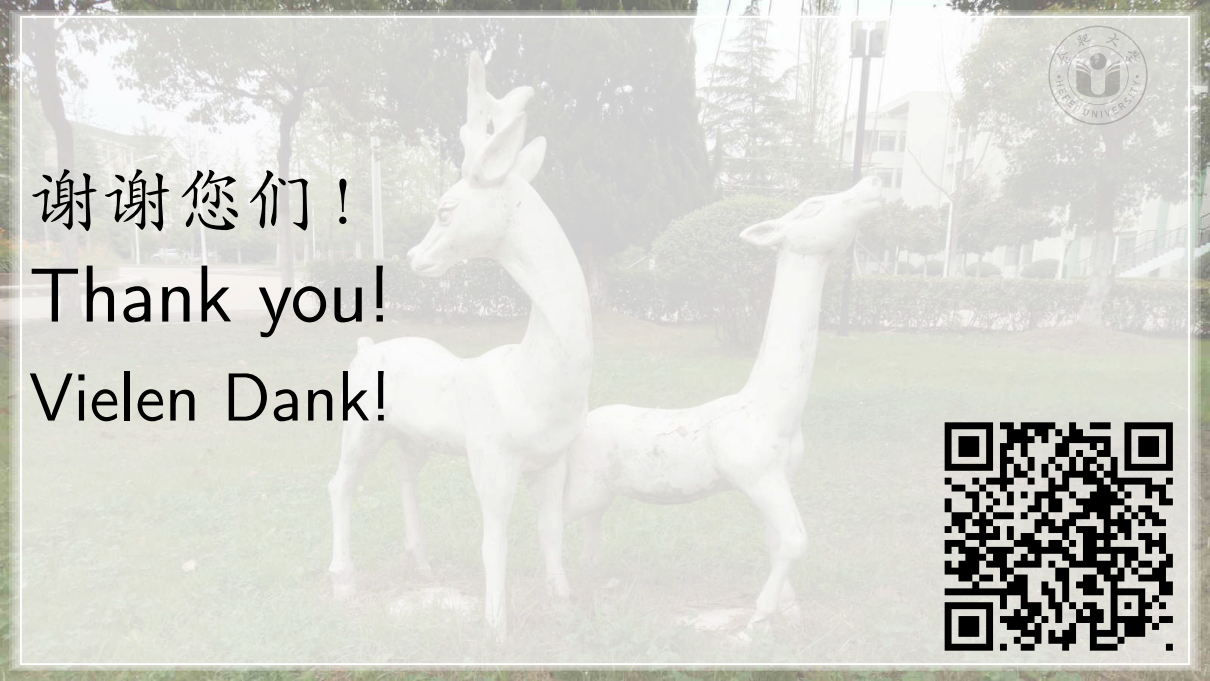


- In dieser Einheit haben wir gelernt, wie wir einen Debugger benutzen können.
- Der Debugger ist eines der wichtigsten Werkzeuge im Werkzeuggürtel eines Programmierers.
- Die Arbeitsweise von Kode wird klar, wenn wir ihn Schritt-für-Schritt ausführen können.
- Wir können Breakpoints setzen und den Prozess laufen lassen, bis er bei ihnen ankommt.
- Dann können wir uns die Werte von Variablen anschauen.
- Dann können wir jede Zeile Kode einzeln ausführen.
- Wir können sehen, wie sich jede einzelne Variable ändert.
- Wir können verfolgen, wie der Kontrollfluss durch die Zweige von Alternativen fließt und durch Schleifen iteriert.
- Wenn unser Kode einen Fehler hat, also einen Bug, dann ist das Benutzen des Debuggers oft der erste Schritt, diesen zu finden.

Zusammenfassung



- In dieser Einheit haben wir gelernt, wie wir einen Debugger benutzen können.
- Der Debugger ist eines der wichtigsten Werkzeuge im Werkzeuggürtel eines Programmierers.
- Die Arbeitsweise von Kode wird klar, wenn wir ihn Schritt-für-Schritt ausführen können.
- Wir können Breakpoints setzen und den Prozess laufen lassen, bis er bei ihnen ankommt.
- Dann können wir uns die Werte von Variablen anschauen.
- Dann können wir jede Zeile Kode einzeln ausführen.
- Wir können sehen, wie sich jede einzelne Variable ändert.
- Wir können verfolgen, wie der Kontrollfluss durch die Zweige von Alternativen fließt und durch Schleifen iteriert.
- Wenn unser Kode einen Fehler hat, also einen Bug, dann ist das Benutzen des Debuggers oft der erste Schritt, diesen zu finden.
- Darum heist er wohl auch Debugger.


谢谢您们！
Thank you!
Vielen Dank!

References I

[1] David J. Agans. *Debugging*. New York, NY, USA: AMACOM, Sep. 2002. ISBN: 978-0-8144-2678-4 (siehe S. 138–147, 356).

[2] Adam Aspin und Karine Aspin. *Query Answers with MariaDB – Volume I: Introduction to SQL Queries*. Tetras Publishing, Okt. 2018. ISBN: 978-1-9996172-4-0. See also[3] (siehe S. 344, 358).

[3] Adam Aspin und Karine Aspin. *Query Answers with MariaDB – Volume II: In-Depth Querying*. Tetras Publishing, Okt. 2018. ISBN: 978-1-9996172-5-7. See also[2] (siehe S. 344, 358).

[4] Daniel J. Barrett. *Efficient Linux at the Command Line*. Sebastopol, CA, USA: O'Reilly Media, Inc., Feb. 2022. ISBN: 978-1-0981-1340-7 (siehe S. 358, 359).

[5] Daniel Bartholomew. *Learning the MariaDB Ecosystem: Enterprise-level Features for Scalability and Availability*. New York, NY, USA: Apress Media, LLC, Okt. 2019. ISBN: 978-1-4842-5514-8 (siehe S. 358).

[6] Kent L. Beck. *JUnit Pocket Guide*. Sebastopol, CA, USA: O'Reilly Media, Inc., Sep. 2004. ISBN: 978-0-596-00743-0 (siehe S. 360).

[7] Richard Beigel. "Irrationality without Number Theory". *The American Mathematical Monthly* 98(4):332–335, Apr. 1991. London, England, UK: Taylor and Francis Ltd. ISSN: 1930-0972. doi:10.1080/00029890.1991.12000762. URL: https://cis.temple.edu/~beigel/papers/irrationality-amm.PS.gz (besucht am 2026-11-04) (siehe S. 357).

[8] Tim Berners-Lee. *Re: Qualifiers on Hypertext links...* Geneva, Switzerland: World Wide Web project, European Organization for Nuclear Research (CERN) und Newsgroups: alt.hypertext, 6. Aug. 1991. URL: https://www.w3.org/People/Berners-Lee/1991/08/art-6484.txt (besucht am 2025-02-05) (siehe S. 360).

[9] Alex Berson. *Client/Server Architecture*. 2. Aufl. Computer Communications Series. New York, NY, USA: McGraw-Hill, 29. März 1996. ISBN: 978-0-07-005664-0 (siehe S. 356).

[10] Silvia Botros und Jeremy Tinley. *High Performance MySQL*. 4. Aufl. Sebastopol, CA, USA: O'Reilly Media, Inc., Nov. 2021. ISBN: 978-1-4920-8051-0 (siehe S. 358).

[11] Ed Bott. *Windows 11 Inside Out*. Hoboken, NJ, USA: Microsoft Press, Pearson Education, Inc., Feb. 2023. ISBN: 978-0-13-769132-6 (siehe S. 358).

References II

[12] Ron Brash und Ganesh Naik. *Bash Cookbook*. Birmingham, England, UK: Packt Publishing Ltd, Juli 2018. ISBN: 978-1-78862-936-2 (siehe S. 356).

[13] Brett Cannon, Jiwon Seo, Yury Selivanov und Larry Hastings. *Function Signature Object*. Python Enhancement Proposal (PEP) 362. Beaverton, OR, USA: Python Software Foundation (PSF), 21. Aug. 2006–4. Juni 2012. URL: https://peps.python.org/pep-0362 (besucht am 2024-12-12) (siehe S. 359).

[14] Jason Cannon. *High Availability for the LAMP Stack*. Shelter Island, NY, USA: Manning Publications, Juni 2022 (siehe S. 357, 359).

[15] Stephan C. Carlson und The Editors of Encyclopaedia Britannica. *Golden Ratio.* Hrsg. von The Editors of Encyclopaedia Britannica. Chicago, IL, USA: Encyclopædia Britannica, Inc., 21. Okt. 2024. URL: https://www.britannica.com/science/golden-ratio (besucht am 2024-12-14) (siehe S. 312–326, 361).

[16] Antonio Cavacini. "Is the CE/BCE notation becoming a standard in scholarly literature?" *Scientometrics* 102(2):1661–1668, Juli 2015. London, England, UK: Springer Nature Limited. ISSN: 0138-9130. doi:10.1007/s11192-014-1352-1 (siehe S. 356).

[17] Josh Centers. *Take Control of iOS 18 and iPadOS 18*. San Diego, CA, USA: Take Control Books, Dez. 2024. ISBN: 978-1-990783-55-5 (siehe S. 357).

[18] Donald D. Chamberlin. "50 Years of Queries". *Communications of the ACM (CACM)* 67(8):110–121, Aug. 2024. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 0001-0782. doi:10.1145/3649887. URL: https://cacm.acm.org/research/50-years-of-queries (besucht am 2025-01-09) (siehe S. 359).

[19] David Clinton und Christopher Negus. *Ubuntu Linux Bible*. 10. Aufl. Bible Series. Chichester, West Sussex, England, UK: John Wiley and Sons Ltd., 10. Nov. 2020. ISBN: 978-1-119-72233-5 (siehe S. 359, 360).

[20] Edgar Frank „Ted" Codd. "A Relational Model of Data for Large Shared Data Banks". *Communications of the ACM (CACM)* 13(6):377–387, Juni 1970. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 0001-0782. doi:10.1145/362384.362685. URL: https://www.seas.upenn.edu/~zives/03f/cis550/codd.pdf (besucht am 2025-01-05) (siehe S. 359).

References III

[21] Christopher Cullen. "Learning from Liu Hui? A Different Way to Do Mathematics". *Notices of the American Mathematical Society* 49(7):783–790, Aug. 2002. Providence, RI, USA: American Mathematical Society (AMS). ISSN: 1088-9477. URL: https://www.ams.org/notices/200207/comm-cullen.pdf (besucht am 2024-08-09) (siehe S. 361).

[22] *Database Language SQL*. Techn. Ber. ANSI X3.135-1986. Washington, D.C., USA: American National Standards Institute (ANSI), 1986 (siehe S. 359).

[23] Joseph W. Dauben. "Archimedes and Liu Hui on Circles and Spheres". *Ontology Studies (Cuadernos de Ontología)* 10:21–38, 2010. Leioa, Bizkaia, Spain: Universidad del País Vasco / Euskal Herriko Unibertsitatea. ISSN: 1576-2270. URL: https://ddd.uab.cat/pub/ontstu/ontstu_a2010n10/ontstu_a2010n10p21.pdf (besucht am 2026-11-03). Also discusses LIU Hui's (刘徽) method of approximating π (siehe S. 361).

[24] Matt David und Blake Barnhill. *How to Teach People SQL*. San Francisco, CA, USA: The Data School, Chart.io, Inc., 10. Dez. 2019–10. Apr. 2023. URL: https://dataschool.com/how-to-teach-people-sql (besucht am 2025-02-27) (siehe S. 359).

[25] *Database Language SQL*. International Standard ISO 9075-1987. Geneva, Switzerland: International Organization for Standardization (ISO), 1987 (siehe S. 359).

[26] Paul Deitel, Harvey Deitel und Abbey Deitel. *Internet & World Wide WebW[: How to Program*. 5. Aufl. Hoboken, NJ, USA: Pearson Education, Inc., Nov. 2011. ISBN: 978-0-13-299045-5 (siehe S. 360).

[27] 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. 358).

[28] "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. 357).

[29] Russell J.T. Dyer. *Learning MySQL and MariaDB*. Sebastopol, CA, USA: O'Reilly Media, Inc., März 2015. ISBN: 978-1-4493-6290-4 (siehe S. 358).

References IV

[30] Euclid of Alexandria (Εὐκλείδης). *Euclid's Elements of Geometry (Στοιχεῖα). The Greek Text of J.L. Heiberg (1883-1885) from Euclidis Elementa, Edidit et Latine Interpretatus est I.L. Heiberg in Aedibus B. G. Teubneri, 1883-1885. Edited, and provided with a modern English translation, by Richard Fitzpatrick*. Hrsg. von Richard Fitzpatrick. Übers. von Johan Ludvig Heiberg. revised and corrected. Austin, TX, USA: The University of Texas at Austin, 2008. ISBN: 978-0-615-17984-1. URL: https://farside.ph.utexas.edu/Books/Euclid/Elements.pdf (besucht am 2024-09-30) (siehe S. 312–326, 361).

[31] Leonhard Euler. "An Essay on Continued Fractions". Übers. von Myra F. Wyman und Bostwick F. Wyman. *Mathematical Systems Theory* 18(1):295–328, Dez. 1985. New York, NY, USA: Springer Science+Business Media, LLC. ISSN: 1432-4350. doi:10.1007/BF01699475. URL: https://www.researchgate.net/publication/301720080 (besucht am 2024-09-24). Translation of [32]. (Siehe S. 347).

[32] Leonhard Euler. "De Fractionibus Continuis Dissertation". *Commentarii Academiae Scientiarum Petropolitanae* 9:98–137, 1737–1744. Petropolis (St. Petersburg), Russia: Typis Academiae. URL: https://scholarlycommons.pacific.edu/cgi/viewcontent.cgi?article=1070 (besucht am 2024-09-24). See [31] for a translation. (Siehe S. 347, 361).

[33] Greg Fee. *The Golden Ratio: (1+sqrt(5))/2 to 20000 Places*. Salt Lake City, UT, USA: Project Gutenberg Literary Archive Foundation, 1. Aug. 1996. URL: https://www.gutenberg.org/ebooks/633 (besucht am 2024-12-14) (siehe S. 312–329).

[34] Luca Ferrari und Enrico Pirozzi. *Learn PostgreSQL*. 2. Aufl. Birmingham, England, UK: Packt Publishing Ltd, Okt. 2023. ISBN: 978-1-83763-564-1 (siehe S. 358).

[35] Michael Filaseta. "The Transcendence of e and π". In: *Math 785: Transcendental Number Theory*. Columbia, SC, USA: University of South Carolina, Frühling 2011. Kap. 6. URL: https://people.math.sc.edu/filaseta/gradcourses/Math785/Math785Notes6.pdf (besucht am 2024-07-05) (siehe S. 361).

[36] 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. 357).

[37] Terry Halpin und Tony Morgan. *Information Modeling and Relational Databases*. 3. Aufl. Burlington, MA, USA/San Mateo, CA, USA: Morgan Kaufmann Publishers, Juli 2024. ISBN: 978-0-443-23791-1 (siehe S. 359).

References V

[38] Jan L. Harrington. *Relational Database Design and Implementation*. 4. Aufl. Burlington, MA, USA/San Mateo, CA, USA: Morgan Kaufmann Publishers, Apr. 2016. ISBN: 978-0-12-849902-3 (siehe S. 359).

[39] Michael Hausenblas. *Learning Modern Linux*. Sebastopol, CA, USA: O'Reilly Media, Inc., Apr. 2022. ISBN: 978-1-0981-0894-6 (siehe S. 358).

[40] Matthew Helmke. *Ubuntu Linux Unleashed 2021 Edition*. 14. Aufl. Reading, MA, USA: Addison-Wesley Professional, Aug. 2020. ISBN: 978-0-13-668539-5 (siehe S. 357, 360).

[41] John Hunt. *A Beginners Guide to Python 3 Programming*. 2. Aufl. Undergraduate Topics in Computer Science (UTICS). Cham, Switzerland: Springer, 2023. ISSN: 1863-7310. ISBN: 978-3-031-35121-1. doi:10.1007/978-3-031-35122-8 (siehe S. 358).

[42] *Information Technology – Database Languages – SQL – Part 1: Framework (SQL/Framework), Part 1*. International Standard ISO/IEC 9075-1:2023(E), Sixth Edition, (ANSI X3.135). Geneva, Switzerland: International Organization for Standardization (ISO) und International Electrotechnical Commission (IEC), Juni 2023. URL: https://standards.iso.org/ittf/PubliclyAvailableStandards/ISO_IEC_9075-1_2023_ed_6_-_id_76583_Publication_PDF_(en).zip (besucht am 2025-01-08). Consists of several parts, see https://modern-sql.com/standard for information where to obtain them. (Siehe S. 359).

[43] Arthur Jones, Kenneth R. Pearson und Sidney A. Morris. "Transcendence of e and π". In: *Abstract Algebra and Famous Impossibilities*. Universitext (UTX). New York, NY, USA: Springer New York, 1991. Kap. 9, S. 115–161. ISSN: 0172-5939. ISBN: 978-1-4419-8552-1. doi:10.1007/978-1-4419-8552-1_8 (siehe S. 361).

[44] Shen Kangshen, John Newsome Crossley und Anthony W.-C. Lun. *The Nine Chapters on the Mathematical Art: Companion and Commentary*. Oxford, Oxfordshire, England, UK: Oxford University Press, 7. Okt. 1999. ISBN: 978-0-19-853936-0. doi:10.1093/oso/9780198539360.001.0001. Translation of the historical Chinese mathematics book *Jiu Zhang Suanshu* (九章算术), including the commentaries of LIU Hui (刘徽). (Siehe S. 361).

[45] 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. 357).

References VI

[46] 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. 358).

[47] Jay LaCroix. *Mastering Ubuntu Server*. 4. Aufl. Birmingham, England, UK: Packt Publishing Ltd, Sep. 2022. ISBN: 978-1-80323-424-3 (siehe S. 359).

[48] 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. 360).

[49] Kent D. Lee und Steve Hubbard. *Data Structures and Algorithms with Python*. Undergraduate Topics in Computer Science (UTICS). Cham, Switzerland: Springer, 2015. ISSN: 1863-7310. ISBN: 978-3-319-13071-2. doi:10.1007/978-3-319-13072-9 (siehe S. 358).

[50] 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. 358).

[51] Gloria Lotha, Aakanksha Gaur, Erik Gregersen, Swati Chopra und William L. Hosch. "Client-Server Architecture". In: *Encyclopaedia Britannica*. Hrsg. von The Editors of Encyclopaedia Britannica. Chicago, IL, USA: Encyclopædia Britannica, Inc., 3. Jan. 2025. URL: https://www.britannica.com/technology/client-server-architecture (besucht am 2025-01-20) (siehe S. 356).

[52] Mark Lutz. *Learning Python*. 6. Aufl. Sebastopol, CA, USA: O'Reilly Media, Inc., März 2025. ISBN: 978-1-0981-7130-8 (siehe S. 358).

[53] *MariaDB Server Documentation*. Milpitas, CA, USA: MariaDB, 2025. URL: https://mariadb.com/kb/en/documentation (besucht am 2025-04-24) (siehe S. 358).

[54] "Mathematical Functions and Operators". In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), 20. Feb. 2025. Kap. 9.3. URL: https://www.postgresql.org/docs/17/functions-math.html (besucht am 2025-02-27) (siehe S. 361).

[55] MDN Contributors. *Signature (Functions)*. San Francisco, CA, USA: Mozilla Corporation, 8. Juni 2023. URL: https://developer.mozilla.org/en-US/docs/Glossary/Signature/Function (besucht am 2024-12-12) (siehe S. 359).

References VII

[56] Jim Melton und Alan R. Simon. *SQL: 1999 – Understanding Relational Language Components*. The Morgan Kaufmann Series in Data Management Systems. Burlington, MA, USA/San Mateo, CA, USA: Morgan Kaufmann Publishers, Juni 2001. ISSN: 1046-1698. ISBN: 978-1-55860-456-8 (siehe S. 359).

[57] Robert Nemiroff und Jerry Bonnell. *The Square Root of Two to 1 Million Digits*. Hanover, MD, USA: Astrophysics Science Division (ASD), National Aeronautics and Space Administration (NASA), 2. Apr. 1997. URL: https://apod.nasa.gov/htmltest/gifcity/sqrt2.1mil (besucht am 2024-12-14) (siehe S. 312–320).

[58] Cameron Newham und Bill Rosenblatt. *Learning the Bash Shell – Unix Shell Programming: Covers Bash 3.0*. 3. Aufl. Sebastopol, CA, USA: O'Reilly Media, Inc., 2005. ISBN: 978-0-596-00965-6 (siehe S. 356).

[59] Ivan Niven. "The Transcendence of π". *The American Mathematical Monthly* 46(8):469–471, Okt. 1939. London, England, UK: Taylor and Francis Ltd. ISSN: 1930-0972. doi:10.2307/2302515 (siehe S. 361).

[60] John J. O'Connor und Edmund F. Robertson. *Liu Hui*. St Andrews, Scotland, UK: University of St Andrews, School of Mathematics and Statistics, Dez. 2003. URL: https://mathshistory.st-andrews.ac.uk/Biographies/Liu_Hui (besucht am 2024-08-10). Discusses LIU Hui's (刘徽) method of approximating π (siehe S. 361).

[61] Regina O. Obe und Leo S. Hsu. *PostgreSQL: Up and Running*. 3. Aufl. Sebastopol, CA, USA: O'Reilly Media, Inc., Okt. 2017. ISBN: 978-1-4919-6336-4 (siehe S. 358).

[62] 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. 360).

[63] 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. 358).

[64] 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. 360).

References VIII

[65] Robert Orfali, Dan Harkey und Jeri Edwards. *Client/Server Survival Guide*. 3. Aufl. Chichester, West Sussex, England, UK: John Wiley and Sons Ltd., 25. Jan. 1999. ISBN: 978-0-471-31615-2 (siehe S. 356).

[66] 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. 358, 360).

[67] Yasset Pérez-Riverol, Laurent Gatto, Rui Wang, Timo Sachsenberg, Julian Uszkoreit, Felipe da Veiga Leprevost, Christian Fufezan, Tobias Ternent, Stephen J. Eglen, 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. 357).

[68] *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), Feb. 2025. URL: https://www.postgresql.org/docs/17/index.html (besucht am 2025-02-25).

[69] *PostgreSQL Essentials: Leveling Up Your Data Work*. Sebastopol, CA, USA: O'Reilly Media, Inc., März 2024 (siehe S. 358).

[70] Abhishek Ratan, Eric Chou, Pradeeban Kathiravelu und Dr. M.O. Faruque Sarker. *Python Network Programming*. Birmingham, England, UK: Packt Publishing Ltd, Jan. 2019. ISBN: 978-1-78883-546-6 (siehe S. 356).

[71] Federico Razzoli. *Mastering MariaDB*. Birmingham, England, UK: Packt Publishing Ltd, Sep. 2014. ISBN: 978-1-78398-154-0 (siehe S. 358).

[72] Mike Reichardt, Michael Gundall und Hans D. Schotten. "Benchmarking the Operation Times of NoSQL and MySQL Databases for Python Clients". In: *47th Annual Conference of the IEEE Industrial Electronics Society (IECON'2021*. 13.–15. Okt. 2021, Toronto, ON, Canada. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers (IEEE), 2021, S. 1–8. ISSN: 2577-1647. ISBN: 978-1-6654-3554-3. doi:10.1109/IECON48115.2021.9589382 (siehe S. 358).

[73] Mark Richards und Neal Ford. *Fundamentals of Software Architecture: An Engineering Approach*. Sebastopol, CA, USA: O'Reilly Media, Inc., Jan. 2020. ISBN: 978-1-4920-4345-4 (siehe S. 356).

[74] Kristian Rother. *Pro Python Best Practices: Debugging, Testing and Maintenance*. New York, NY, USA: Apress Media, LLC, März 2017. ISBN: 978-1-4842-2241-6 (siehe S. 138–147, 356).

References IX

[75] Ernest E. Rothman, Rich Rosen und Brian Jepson. *Mac OS X for Unix Geeks*. 4. Aufl. Sebastopol, CA, USA: O'Reilly Media, Inc., Sep. 2008. ISBN: 978-0-596-52062-5 (siehe S. 358).

[76] 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. 360).

[77] Ahmad Sahar. *iOS 26 Programming for Beginners*. 10. Aufl. Birmingham, England, UK: Packt Publishing Ltd, Nov. 2025. ISBN: 978-1-80602-393-6 (siehe S. 361).

[78] Syamal K. Sen und Ravi P. Agarwal. "Existence of year zero in astronomical counting is advantageous and preserves compatibility with significance of AD, BC, CE, and BCE". In: *Zero – A Landmark Discovery, the Dreadful Void, and the Ultimate Mind*. Amsterdam, The Netherlands: Elsevier B.V., 2016. Kap. 5.5, S. 94–95. ISBN: 978-0-08-100774-7. doi:10.1016/C2015-0-02299-7 (siehe S. 356).

[79] Ellen Siever, Stephen Figgins, Robert Love und Arnold Robbins. *Linux in a Nutshell*. 6. Aufl. Sebastopol, CA, USA: O'Reilly Media, Inc., Sep. 2009. ISBN: 978-0-596-15448-6 (siehe S. 358).

[80] Anna Skoulikari. *Learning Git*. Sebastopol, CA, USA: O'Reilly Media, Inc., Mai 2023. ISBN: 978-1-0981-3391-7 (siehe S. 357).

[81] Neil James Alexander Sloane. *Decimal Expansion of Golden Ratio phi (or tau) = (1 + sqrt(5))/2*. Hrsg. von John Horton Conway. Bd. A001622 der Reihe The On-Line Encyclopedia of Integer Sequences. Highland Park, NJ, USA: The OEIS Foundation Inc., 13. Dez. 2024. URL: https://oeis.org/A001622 (besucht am 2024-12-14) (siehe S. 312–326, 361).

[82] Neil James Alexander Sloane. *Decimal Expansion of Square Root of 2*. Hrsg. von John Horton Conway. Bd. A002193 der Reihe The On-Line Encyclopedia of Integer Sequences. Highland Park, NJ, USA: The OEIS Foundation Inc., 13. Dez. 2024. URL: https://oeis.org/A002193 (besucht am 2024-12-14) (siehe S. 312–320).

[83] Drew Smith. *Modern Apple Platform Administration – macOS and iOS Essentials (2025)*. Birmingham, England, UK: Packt Publishing Ltd, Feb. 2025. ISBN: 978-1-80580-309-6 (siehe S. 357, 358).

[84] John Miles Smith und Philip Yen-Tang Chang. "Optimizing the Performance of a Relational Algebra Database Interface". *Communications of the ACM (CACM)* 18(10):568–579, Okt. 1975. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 0001-0782. doi:10.1145/361020.361025 (siehe S. 359).

References X

[85] "SQL Commands". In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), 20. Feb. 2025. Kap. Part VI. Reference. URL: https://www.postgresql.org/docs/17/sql-commands.html (besucht am 2025-02-25) (siehe S. 359).

[86] Ryan K. Stephens und Ronald R. Plew. *Sams Teach Yourself SQL in 21 Days*. 4. Aufl. Sams Teach Yourself. Indianapolis, IN, USA: SAMS Technical Publishing und Hoboken, NJ, USA: Pearson Education, Inc., Okt. 2002. ISBN: 978-0-672-32451-2 (siehe S. 353, 359).

[87] Ryan K. Stephens, Ronald R. Plew, Bryan Morgan und Jeff Perkins. *SQL in 21 Tagen. Die Datenbank-Abfragesprache SQL vollständig erklärt (in 14/21 Tagen)*. 6. Aufl. Burgthann, Bayern, Germany: Markt+Technik Verlag GmbH, Feb. 1998. ISBN: 978-3-8272-2020-2. Translation of [86] (siehe S. 359).

[88] Philip D. Straffin Jr. "Liu Hui and the First Golden Age of Chinese Mathematics". *Mathematics Magazine* 71(3):163–181, Juni 1998. London, England, UK: Taylor and Francis Ltd. ISSN: 0025-570X. doi:10.2307/2691200. URL: https://www.researchgate.net/publication/237334342 (besucht am 2024-08-10) (siehe S. 361).

[89] Michael J. Sullivan und Ivan Levkivskyi. *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. 31–33).

[90] Allen Taylor. *Introducing SQL and Relational Databases*. New York, NY, USA: Apress Media, LLC, Sep. 2018. ISBN: 978-1-4842-3841-7 (siehe S. 359).

[91] Alkin Tezuysal und Ibrar Ahmed. *Database Design and Modeling with PostgreSQL and MySQL*. Birmingham, England, UK: Packt Publishing Ltd, Juli 2024. ISBN: 978-1-80323-347-5 (siehe S. 358).

[92] The Editors of Encyclopaedia Britannica, Hrsg. *Encyclopaedia Britannica*. Chicago, IL, USA: Encyclopædia Britannica, Inc.

[93] 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. 360).

References XI

[94] Linus Torvalds. "The Linux Edge". *Communications of the ACM (CACM)* 42(4):38–39, Apr. 1999. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 0001-0782. doi:10.1145/299157.299165 (siehe S. 358).

[95] 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. 357, 360).

[96] Bruce M. Van Horn II und Quan Nguyen. *Hands-On Application Development with PyCharm*. 2. Aufl. Birmingham, England, UK: Packt Publishing Ltd, Okt. 2023. ISBN: 978-1-83763-235-0 (siehe S. 358).

[97] 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. 360).

[98] 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. 357).

[99] Sander van Vugt. *Linux Fundamentals*. 2. Aufl. Hoboken, NJ, USA: Pearson IT Certification, Juni 2022. ISBN: 978-0-13-792931-3 (siehe S. 358).

[100] Thomas Weise (汤卫思). *Databases*. 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), 2025. URL: https://thomasweise.github.io/databases (besucht am 2025-01-05) (siehe S. 356, 359).

[101] 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. 356, 358).

[102] Eric Wolfgang Weisstein. "Irrational Number". In: *MathWorld – A Wolfram Web Resource*. Champaign, IL, USA: Wolfram Research, Inc., 22. Aug. 2024. URL: https://mathworld.wolfram.com/IrrationalNumber.html (besucht am 2026-11-04) (siehe S. 357).

[103] Eric Wolfgang Weisstein. *MathWorld – A Wolfram Web Resource*. Champaign, IL, USA: Wolfram Research, Inc., 22. Aug. 2024. URL: https://mathworld.wolfram.com (besucht am 2024-09-24).

References XII

[104] Eric Wolfgang Weisstein. "Transcendental Number". In: *MathWorld – A Wolfram Web Resource*. Champaign, IL, USA: Wolfram Research, Inc., 22. Aug. 2024. URL: https://mathworld.wolfram.com/TranscendentalNumber.html (besucht am 2026-11-04) (siehe S. 359).

[105] *What is a Relational Database?* Armonk, NY, USA: International Business Machines Corporation (IBM), 20. Okt. 2021–12. Dez. 2024. URL: https://www.ibm.com/think/topics/relational-databases (besucht am 2025-01-05) (siehe S. 359).

[106] Ulf Michael „Monty" Widenius, David Axmark und Uppsala, Sweden: MySQL AB. *MySQL Reference Manual – Documentation from the Source*. Sebastopol, CA, USA: O'Reilly Media, Inc., 9. Juli 2002. ISBN: 978-0-596-00265-7 (siehe S. 358).

[107] Kevin Wilson. *Python Made Easy*. Birmingham, England, UK: Packt Publishing Ltd, Aug. 2024. ISBN: 978-1-83664-615-0 (siehe S. 138–147, 356, 358).

[108] Martin Yanev. *PyCharm Productivity and Debugging Techniques*. Birmingham, England, UK: Packt Publishing Ltd, Okt. 2022. ISBN: 978-1-83763-244-2 (siehe S. 358).

[109] Kinza Yasar und Craig S. Mullins. *Definition: Database Management System (DBMS)*. Newton, MA, USA: TechTarget, Inc., Juni 2024. URL: https://www.techtarget.com/searchdatamanagement/definition/database-management-system (besucht am 2025-01-11) (siehe S. 356).

[110] Wenqi Ying (应雯棋), Hrsg. *Commemoration of Ancient Chinese Mathematical Master Liu Hui for his Timeless Influence on Mathematics and Civilizational Exchange*. Bd. 48 (Special Issue) der Reihe CAST Newsletter. China, Beijing (中国北京市): 中国科学技术协会 (China Association for Science and Technology, CAST), Nov. 2024. URL: https://english.cast.org.cn/cms_files/filemanager/1941250207/attach/202412/8f23655a82364d19ad7874eb37b23035.pdf (besucht am 2025-08-24). Proofreader: Yumeng Wei (魏雨萌), Designer: Shan Zhang (张珊) (siehe S. 361).

[111] Giorgio Zarrelli. *Mastering Bash*. Birmingham, England, UK: Packt Publishing Ltd, Juni 2017. ISBN: 978-1-78439-687-9 (siehe S. 356).

[112] Nicola Abdo Ziadeh, Michael B. Rowton, A. Geoffrey Woodhead, Wolfgang Helck, Jean L.A. Filliozat, Hiroyuki Momo, Eric Thompson, E.J. Wiesenberg und Shih-ch'ang Wu. "Chronology – Christian History, Dates, Events". In: *Encyclopaedia Britannica*. Hrsg. von The Editors of Encyclopaedia Britannica. Chicago, IL, USA: Encyclopædia Britannica, Inc., 26. Juli 1999–20. März 2024. URL: https://www.britannica.com/topic/chronology/Christian (besucht am 2025-08-27) (siehe S. 356).

Glossary (in English) I



Bash is a shell used under Ubuntu Linux, i.e., the program that „runs“ in the terminal and interprets your commands, allowing you to start and interact with other programs[12,58,111]. Learn more at https://www.gnu.org/software/bash.

BCE The time notation *before Common Era* is a non-religious but chronological equivalent alternative to the traditional *Before Christ (BC)* notation, which refers to the years *before* the birth of Jesus Christ[16]. The years BCE are counted down, i.e., the larger the year, the farther in the past. The year 1 BCE comes directly before the year 1 Common Era (CE)[78,112].

breakpoint A breakpoint is a mark in a line of code in an Integrated Development Environment (IDE) at which the debugger will pause the execution of a program.

CE The time notation *Common Era* is a non-religious but chronological equivalent alternative to the traditional *Anno Domini (AD)* notation, which refers to the years *after* the birth of Jesus Christ[16]. The years CE are counted upwards, i.e., the smaller they are, the farther they are in the past. The year 1 CE comes directly after the year 1 before Common Era (BCE)[78,112].

client In a client-server architecture, the client is a device or process that requests a service from the server. It initiates the communication with the server, sends a request, and receives the response with the result of the request. Typical examples of clients are web browsers in the internet as well as clients for database management systems (DBMSes), such as psql.

client-server architecture is a system design where a central server receives requests from one or multiple clients[9,51,65,70,73]. These requests and responses are usually sent over network connections. A typical example of such a system is the World Wide Web (WWW), where web servers host websites and make them available to web browsers, the clients. Another typical example is the structure of database (DB) software, where a central server, the DBMS, offers access to the DB to the different clients. Here, the client can be some terminal software shipping with the DBMS, such as psql, or the different applications that access the DBs.

DB A *database* is an organized collection of structured information or data, typically stored electronically in a computer system. Databases are discussed in our book *Databases*[100].

DBMS A *database management system* is the software layer located between the user or application and the DB. The DBMS allows the user/application to create, read, write, update, delete, and otherwise manipulate the data in the DB[109].

debugger A debugger is a tool that lets you execute a program step-by-step while observing the current values of variables. This allows you to find errors in the code more easily[1,74,107]. Learn more about debugging in [101].

Glossary (in English) II



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 functions[36]. They must be delimited by `"""..."""`[36,98].

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`[28] or tools such as pytest[45]. 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 System (VCS) which allows multiple users to work on the same code while preserving the history of the code changes[80,95]. Learn more at https://git-scm.com.

GitHub is a website where software projects can be hosted and managed via the Git VCS[67,95]. Learn more at https://github.com.

IDE An *Integrated Developer Environment* is a program that allows the user do multiple different activities required for software development in one single system. It often offers functionality such as editing source code, debugging, testing, or interaction with a distributed version control system. For Python, we recommend using PyCharm. On Apple systems, Xcode is often used.

iOS is the operating system that powers Apple iPhones[17,83]. Learn more at https://www.apple.com/ios.

iPadOS is the operating system that powers Apple iPads[17]. Learn more at https://www.apple.com/ipados.

irrational An *irrational* number z is a real number that cannot be expressed as fraction $\frac{p}{q}$ of two integers $p \in \mathbb{Z}$ and $q \in \mathbb{N}_1$[7,102]. In other words, it holds that $z \in \mathbb{R}$ and $z \notin \mathbb{Q}$. The set of irrational numbers could be denoted as $\mathbb{R} \setminus \mathbb{Q}$. Irrational numbers have decimal expansions that neither terminate nor become periodic. Examples of irrational numbers are $\sqrt{2}$, ϕ, π, and e.

IT information technology

LAMP Stack A system setup for web applications: Linux, Apache (a web server), MySQL, and the server-side scripting language PHP[14,40].

Glossary (in English) III



Linux is the leading open source operating system, i.e., a free alternative to Microsoft Windows[4,39,79,94,99]. We recommend using it for this course, for software development, and for research. Learn more at https://www.linux.org. Its variant Ubuntu is particularly easy to use and install.

macOS or Mac OS is the operating system that powers Apple Mac(intosh) computers[75,83]. Learn more at https://www.apple.com/macos.

MariaDB An open source relational database management system that has forked off from MySQL[2,3,5,29,53,71]. See https://mariadb.org for more information.

Microsoft Windows is a commercial proprietary operating system[11]. It is widely spread, but we recommend using a Linux variant such as Ubuntu for software development and for our course. Learn more at https://www.microsoft.com/windows.

modulo division is, in Python, done by the operator `%` that computes the remainder of a division. `15 % 6` gives us `3`. The function `divmod(a, b)` yields two results, `a // b` and `a % b`.

Mypy is a static type checking tool for Python[80] that makes use of type hints. Learn more at https://github.com/python/mypy and in [101].

MySQL An open source relational database management system[10,29,72,91,106]. MySQL is famous for its use in the LAMP Stack. See https://www.mysql.com for more information.

numerator The number a of a fraction $\frac{a}{b} \in \mathbb{Q}$ is called the *numerator*.

PostgreSQL An open source object-relational DBMS[34,61,69,91]. See https://postgresql.org for more information.

psql is the client program used to access the PostgreSQL DBMS server.

PyCharm is the convenient Python IDE that we recommend for this course[96,107,108]. It comes in a free edition, so it can be downloaded and used at no cost. Learn more at https://www.jetbrains.com/pycharm.

pytest is a framework for writing and executing unit tests in Python[27,46,63,66,107]. Learn more at https://pytest.org.

Python The Python programming language[41,49,52,101], i.e., what you will learn about in our book[101]. Learn more at https://python.org.

Glossary (in English) IV



relational database A relational DB is a database that organizes data into rows (tuples, records) and columns (attributes), which collectively form tables (relations) where the data points are related to each other[20,37,38,84,90,100,105].

server In a client-server architecture, the server is a process that fulfills the requests of the clients. It usually waits for incoming communication carring the requests from the clients. For each request, it takes the necessary actions, performs the required computations, and then sends a response with the result of the request. Typical examples of servers are web servers[14] in the internet as well as DBMSes. It is also common to refer to the computer running the server processes as server as well, i.e., to call it the „server computer“[47].

signature The signature of a function refers to the parameters and their types, the return type, and the exceptions that the function can raise[55]. In Python, the function `signature` of the module `inspect` provides some information about the signature of a function[13].

SQL The *Structured Query Language* is basically a programming language for querying and manipulating relational databases[18,22,24,25,42,56,85–87,90]. It is understood by many DBMSes. You find the Structured Query Language (SQL) commands supported by PostgreSQL in the reference[85].

terminal A terminal is a text-based window where you can enter commands and execute them[4,19]. Knowing what a terminal is and how to use it is very essential in any programming- or system administration-related task. If you want to open a terminal under Microsoft Windows, you can Druck auf ⊞+R, dann Schreiben von `cmd`, dann Druck auf ↵. Under Ubuntu Linux, Ctrl+Alt+T opens a terminal, which then runs a Bash shell inside.

transcendental A *transcendental* number is a number that is not the root of any integer polynomial, meaning that it is not an algebraic number of any degree[104]. Every real transcendental number must also be irrational, since a rational number is, by definition, an algebraic number of degree one[104]. Examples of transcendental numbers are π and e.

Glossary (in English) V



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[48,97]. 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.

Ubuntu is a variant of the open source operating system Linux[19,40]. We recommend that you use this operating system to follow this class, for software development, and for research. Learn more at https://ubuntu.com. If you are in China, you can download it from https://mirrors.ustc.edu.cn/ubuntu-releases.

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[6,62,64,66,76,93]. 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 code[95]. 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.

WWW World Wide Web[8,26]

Glossary (in English) VI



Xcode is offers the tools for developing, testing, and distributing applications as well as an IDE for Apple platforms such as macOS and iOS[77].

ϕ The golden ratio (or golden section) ϕ is the irrational number $\frac{1+\sqrt{5}}{2}$. It is the ratio of a line segment cut into two pieces of different lengths such that the ratio of the whole segment to that of the longer segment is equal to the ratio of the longer segment to the shorter segment[15,30]. The golden ratio is approximately $\phi \approx 1.618\,033\,988\,749\,894\,848\,204\,586\,834$[81]. Represented as `float` in Python, its value is `1.618033988749895`.

π is the ratio of the circumference U of a circle and its diameter d, i.e., $\pi = U/d$. $\pi \in \mathbb{R}$ is an irrational and transcendental number[35,43,59], which is approximately $\pi \approx 3.141\,592\,653\,589\,793\,238\,462\,643$. An early method to approximate π was developed by the Chinese mathematician LIU Hui (刘徽) based on inscribing regular e-gons into a circle somewhere in the third century CE[60,110] in his commentary to the famous Chinese mathematics book *Jiu Zhang Suanshu* (九章算术)[21,23,44,60,88]. In Python, π is provided by the `math` module as variable `pi` with value `3.141592653589793`. In PostgreSQL, it is provided by the SQL function `pi()` with value `3.141592653589793`[54].

e is Euler's number[32], the base of the natural logarithm. $e \in \mathbb{R}$ is an irrational and transcendental number[35,43], which is approximately $e \approx 2.718\,281\,828\,459\,045\,235\,360$. In Python, it is provided by the `math` module as variable `e` with value `2.718281828459045`. In PostgreSQL, you can obtain it via the SQL function `exp(1)` as value `2.718281828459045`[54].

$\mathbb{N_0}$ the set of the natural numbers *including* 0, i.e., 0, 1, 2, 3, and so on. It holds that $\mathbb{N_0} \subset \mathbb{Z}$ and that $\mathbb{N_0} = \mathbb{N_1} \cup \{0\}$.

$\mathbb{N_1}$ the set of the natural numbers *excluding* 0, i.e., 1, 2, 3, 4, and so on. It holds that $\mathbb{N_1} \subset \mathbb{Z}$ and that $\mathbb{N_1} = \mathbb{N_0} \setminus \{0\}$.

$\mathbb{Q}$ 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 $\mathbb{Q} \subset \mathbb{R}$.

$\mathbb{R}$ the set of the real numbers.

$\mathbb{Z}$ the set of the integers numbers including positive and negative numbers and 0, i.e., . . . , -3, -2, -1, 0, 1, 2, 3, . . . , and so on. It holds that $\mathbb{Z} \subset \mathbb{R}$.