



Programming with Python

47. Zwischenspiel: Debugger

Thomas Weise (汤卫思)
tweise@hfuu.edu.cn

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

人工智能与大数据学院
合肥大学
中国安徽省合肥市

Programming with Python



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

Die Webseite mit dem Lehrmaterial dieses Kurses ist <https://thomasweise.github.io/programmingWithPython> (siehe auch den QR-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 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.
- 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 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.
- 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 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.
- 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 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.
- 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.

```
1     def decimal_str(self, max_frac: int = 100) -> str:
2         """
3             Convert the fraction to decimal string.
4
5             :param max_frac: the maximum number of fractional digits
6             :return: the string
7
8             >>> Fraction(124, 2).decimal_str()
9                 '62'
10            >>> Fraction(1, 2).decimal_str()
11                 '0.5'
12            >>> Fraction(1, 3).decimal_str(10)
13                 '0.3333333333'
14            >>> Fraction(-101001, 100000000).decimal_str()
15                 '-0.00101001'
16            >>> Fraction(1235, 1000).decimal_str(2)
17                 '1.24'
18            >>> Fraction(99995, 100000).decimal_str(5)
19                 '0.99995'
20            >>> Fraction(91995, 100000).decimal_str(3)
21                 '0.92'
22            >>> Fraction(99995, 100000).decimal_str(4)
23                 '1'
24             """
25
26             a: int = self.a # Get the numerator.
27             if a == 0: # If the fraction is 0, we return 0.
28                 return "0"
29
30             negative: Final[bool] = a < 0 # Get the sign of the fraction.
31             a = abs(a) # Make sure that `a` is now positive.
32             b: Final[int] = self.b # Get the denominator.
33
34             digits: Final[list] = [] # A list for collecting digits.
35             while (a != 0) and (len(digits) <= max_frac): # Create digits.
36                 digits.append(a // b) # Add the current digit.
37                 a = 10 * (a % b) # Ten times the remainder -> next digit.
38
39             if (a // b) >= 5: # Do we need to round up?
40                 digits[-1] += 1 # Round up by incrementing last digit.
41
42             if len(digits) <= 1: # Do we only have an integer part?
43                 return str((-1 if negative else 1) * digits[0])
44
45             digits.insert(1, ".") # Multiple digits: Insert a decimal dot.
46             if negative: # Do we need to restore the sign?
47                 digits.insert(0, "-") # Insert the sign at the beginning.
48             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`.

```
1     def decimal_str(self, max_frac: int = 100) -> str:
2         """
3             Convert the fraction to decimal string.
4
5             :param max_frac: the maximum number of fractional digits
6             :return: the string
7
8             >>> Fraction(124, 2).decimal_str()
9                 '62'
10            >>> Fraction(1, 2).decimal_str()
11                '0.5'
12            >>> Fraction(1, 3).decimal_str(10)
13                '0.3333333333'
14            >>> Fraction(-101001, 100000000).decimal_str()
15                '-0.00101001'
16            >>> Fraction(1235, 1000).decimal_str(2)
17                '1.24'
18            >>> Fraction(99995, 100000).decimal_str(5)
19                '0.99995'
20            >>> Fraction(91995, 100000).decimal_str(3)
21                '0.92'
22            >>> Fraction(99995, 100000).decimal_str(4)
23                '1'
24
25        """
26
27        a: int = self.a # Get the numerator.
28        if a == 0: # If the fraction is 0, we return 0.
29            return "0"
30
31        negative: Final[bool] = a < 0 # Get the sign of the fraction.
32        a = abs(a) # Make sure that `a` is now positive.
33        b: Final[int] = self.b # Get the denominator.
34
35
36        digits: Final[list] = [] # A list for collecting digits.
37        while (a != 0) and (len(digits) <= max_frac): # Create digits.
38            digits.append(a // b) # Add the current digit.
39            a = 10 * (a % b) # Ten times the remainder -> next digit.
40
41        if (a // b) >= 5: # Do we need to round up?
42            digits[-1] += 1 # Round up by incrementing last digit.
43
44        if len(digits) <= 1: # Do we only have an integer part?
45            return str((-1 if negative else 1) * digits[0])
46
47        digits.insert(1, ".") # Multiple digits: Insert a decimal dot.
48        if negative: # Do we need to restore the sign?
49            digits.insert(0, "-") # Insert the sign at the beginning.
50        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 Boolesche 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 Boolesche 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 Boolese 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 Boolese 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`³³.

```
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`³³. 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`³³. 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 Boolesche 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 updateen 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 updateen 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 $179 \text{ // } 16$.
- 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 updateen 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 updateen 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 update 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, während 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 $a = 120$, während $b = 16$ 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 $a = 120$, während $b = 16$ 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 $a = 120$, während $b = 16$ 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 $80 \text{ // } 16$, 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 stimmt, 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 stimmt, 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 stimmt, 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 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 stimmt, 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 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 stimmt, 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 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 bekommt, 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 bekommt, 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 bekommt, 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.

```
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
>>> 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.

```
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
379
380
381
382
383
384
385
386
387
388
389
389
390
391
392
393
394
395
396
397
398
399
399
400
401
402
403
404
405
406
407
408
409
409
410
411
412
413
414
415
416
417
418
419
419
420
421
422
423
424
425
426
427
428
429
429
430
431
432
433
434
435
436
437
438
439
439
440
441
442
443
444
445
446
447
448
449
449
450
451
452
453
454
455
456
457
458
459
459
460
461
462
463
464
465
466
467
468
469
469
470
471
472
473
474
475
476
477
478
478
479
479
480
481
482
483
484
485
486
487
488
489
489
490
491
492
493
494
495
496
497
498
499
499
500
501
502
503
504
505
506
507
508
509
509
510
511
512
513
514
515
516
517
518
519
519
520
521
522
523
524
525
526
527
528
529
529
530
531
532
533
534
535
536
537
538
539
539
540
541
542
543
544
545
546
547
547
548
548
549
549
550
551
552
553
554
555
556
557
558
559
559
560
561
562
563
564
565
566
567
568
569
569
570
571
572
573
574
575
576
577
578
579
579
580
581
582
583
584
585
586
587
588
589
589
590
591
592
593
594
595
596
597
597
598
598
599
599
600
601
602
603
604
605
606
607
608
609
609
610
611
612
613
614
615
616
617
617
618
618
619
619
620
621
622
623
624
625
626
627
627
628
628
629
629
630
631
632
633
634
635
636
637
637
638
638
639
639
640
641
642
643
644
645
646
646
647
647
648
648
649
649
650
651
652
653
654
655
656
657
657
658
658
659
659
660
661
662
663
664
665
666
667
667
668
668
669
669
670
671
672
673
674
675
675
676
676
677
677
678
678
679
679
680
681
682
683
684
685
686
686
687
687
688
688
689
689
690
691
692
693
694
695
696
696
697
697
698
698
699
699
700
701
702
703
704
705
706
707
707
708
708
709
709
710
711
712
713
714
715
715
716
716
717
717
718
718
719
719
720
721
722
723
724
725
725
726
726
727
727
728
728
729
729
730
731
732
733
734
735
736
736
737
737
738
738
739
739
740
741
742
743
744
745
745
746
746
747
747
748
748
749
749
750
751
752
753
754
755
756
756
757
757
758
758
759
759
760
761
762
763
764
765
766
766
767
767
768
768
769
769
770
771
772
773
774
775
775
776
776
777
777
778
778
779
779
780
781
782
783
784
785
786
786
787
787
788
788
789
789
790
791
792
793
794
795
795
796
796
797
797
798
798
799
799
800
801
802
803
804
805
806
806
807
807
808
808
809
809
810
811
812
813
814
815
815
816
816
817
817
818
818
819
819
820
821
822
823
824
825
825
826
826
827
827
828
828
829
829
830
831
832
833
834
835
835
836
836
837
837
838
838
839
839
840
841
842
843
844
845
845
846
846
847
847
848
848
849
849
850
851
852
853
854
855
855
856
856
857
857
858
858
859
859
860
861
862
863
864
865
865
866
866
867
867
868
868
869
869
870
871
872
873
874
875
875
876
876
877
877
878
878
879
879
880
881
882
883
884
885
885
886
886
887
887
888
888
889
889
890
891
892
893
894
894
895
895
896
896
897
897
898
898
899
899
900
901
902
903
903
904
904
905
905
906
906
907
907
908
908
909
909
910
911
912
913
913
914
914
915
915
916
916
917
917
918
918
919
919
920
921
922
923
924
924
925
925
926
926
927
927
928
928
929
929
930
931
932
933
934
934
935
935
936
936
937
937
938
938
939
939
940
941
942
943
943
944
944
945
945
946
946
947
947
948
948
949
949
950
951
952
953
953
954
954
955
955
956
956
957
957
958
958
959
959
960
961
962
963
963
964
964
965
965
966
966
967
967
968
968
969
969
970
971
972
973
973
974
974
975
975
976
976
977
977
978
978
979
979
980
981
982
983
983
984
984
985
985
986
986
987
987
988
988
989
989
990
991
992
992
993
993
994
994
995
995
996
996
997
997
998
998
999
999
999
999
```

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.

```
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
389
390
391
392
393
394
395
396
397
398
399
399
400
401
402
403
404
405
406
407
408
409
409
410
411
412
413
414
415
416
417
417
418
419
419
420
421
422
423
424
425
426
427
427
428
429
429
430
431
432
433
434
435
436
437
437
438
439
439
440
441
442
443
444
445
445
446
447
448
449
449
450
451
452
453
454
455
455
456
457
458
459
459
460
461
462
463
464
465
465
466
467
468
468
469
469
470
471
472
473
474
475
476
476
477
478
479
479
480
481
482
483
484
485
486
486
487
488
489
489
490
491
492
493
494
495
496
497
498
499
499
500
501
502
503
504
505
506
507
508
509
509
510
511
512
513
514
515
516
517
517
518
519
519
520
521
522
523
524
525
525
526
527
527
528
528
529
529
530
531
532
533
533
534
535
535
536
537
537
538
538
539
539
540
541
541
542
542
543
543
544
544
545
545
546
546
547
547
548
548
549
549
550
551
552
553
554
555
556
557
558
559
559
560
561
562
563
564
565
566
567
568
569
569
570
571
572
573
574
575
576
577
578
579
579
580
581
582
583
584
585
586
587
587
588
589
589
590
591
592
593
594
595
596
597
598
599
599
600
601
602
603
604
605
606
607
608
609
609
610
611
612
613
614
615
616
617
618
618
619
619
620
621
622
623
624
625
625
626
627
627
628
628
629
629
630
631
632
633
633
634
635
635
636
636
637
637
638
638
639
639
640
641
642
643
643
644
645
645
646
646
647
647
648
648
649
649
650
651
652
653
653
654
655
655
656
656
657
657
658
658
659
659
660
661
662
663
663
664
665
665
666
666
667
667
668
668
669
669
670
671
672
673
673
674
675
675
676
676
677
677
678
678
679
679
680
681
682
683
683
684
685
685
686
686
687
687
688
688
689
689
690
691
692
693
693
694
695
695
696
696
697
697
698
698
699
699
700
701
702
703
703
704
705
705
706
706
707
707
708
708
709
709
710
711
712
713
713
714
715
715
716
716
717
717
718
718
719
719
720
721
722
723
723
724
725
725
726
726
727
727
728
728
729
729
730
731
732
733
733
734
735
735
736
736
737
737
738
738
739
739
740
741
742
743
743
744
745
745
746
746
747
747
748
748
749
749
750
751
752
753
753
754
755
755
756
756
757
757
758
758
759
759
760
761
762
763
763
764
765
765
766
766
767
767
768
768
769
769
770
771
772
773
773
774
775
775
776
776
777
777
778
778
779
779
780
781
782
783
783
784
785
785
786
786
787
787
788
788
789
789
790
791
792
793
793
794
795
795
796
796
797
797
798
798
799
799
800
801
802
803
803
804
805
805
806
806
807
807
808
808
809
809
810
811
812
813
813
814
815
815
816
816
817
817
818
818
819
819
820
821
822
823
823
824
825
825
826
826
827
827
828
828
829
829
830
831
832
833
833
834
835
835
836
836
837
837
838
838
839
839
840
841
842
843
843
844
845
845
846
846
847
847
848
848
849
849
850
851
852
853
853
854
855
855
856
856
857
857
858
858
859
859
860
861
862
863
863
864
865
865
866
866
867
867
868
868
869
869
870
871
872
873
873
874
875
875
876
876
877
877
878
878
879
879
880
881
882
883
883
884
885
885
886
886
887
887
888
888
889
889
890
891
892
893
893
894
895
895
896
896
897
897
898
898
899
899
900
901
902
903
903
904
905
905
906
906
907
907
908
908
909
909
910
911
912
913
913
914
915
915
916
916
917
917
918
918
919
919
920
921
922
923
923
924
925
925
926
926
927
927
928
928
929
929
930
931
932
933
933
934
935
935
936
936
937
937
938
938
939
939
940
941
942
943
943
944
945
945
946
946
947
947
948
948
949
949
950
951
952
953
953
954
955
955
956
956
957
957
958
958
959
959
960
961
962
963
963
964
965
965
966
966
967
967
968
968
969
969
970
971
972
973
973
974
975
975
976
976
977
977
978
978
979
979
980
981
982
983
983
984
985
985
986
986
987
987
988
988
989
989
990
991
992
993
993
994
995
995
996
996
997
997
998
998
999
999
999
999
999
999
```

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.

```
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
999
>>> 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.

```
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
>>> 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.

```
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
>>> 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 $\frac{-101001}{1000000000}$.

```
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
>>> 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.

```
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
>>> 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.

```
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
>>> 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.

```
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
```

```
>>> 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.

```
8  
9  
10  
11  
12  
13  
14  
15  
16  
17  
18  
19  
20  
21  
22  
23  
24  
25  
26  
27  
28  
29  
30  
31  
32  
33  
34  
35  
36  
37  
38  
39  
40  
41  
42  
43  
44  
45  
  
    >>> 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: 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.

```
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
409
410
411
412
413
414
415
416
417
418
419
419
420
421
422
423
424
425
426
427
428
429
429
430
431
432
433
434
435
436
437
437
438
439
439
440
441
442
443
443
444
445
446
446
447
448
449
449
450
451
452
453
453
454
455
456
456
457
458
458
459
459
460
461
462
462
463
464
464
465
465
466
467
467
468
468
469
469
470
471
471
472
472
473
473
474
474
475
475
476
476
477
477
478
478
479
479
480
480
481
481
482
482
483
483
484
484
485
485
486
486
487
487
488
488
489
489
490
490
491
491
492
492
493
493
494
494
495
495
496
496
497
497
498
498
499
499
500
500
501
501
502
502
503
503
504
504
505
505
506
506
507
507
508
508
509
509
510
510
511
511
512
512
513
513
514
514
515
515
516
516
517
517
518
518
519
519
520
520
521
521
522
522
523
523
524
524
525
525
526
526
527
527
528
528
529
529
530
530
531
531
532
532
533
533
534
534
535
535
536
536
537
537
538
538
539
539
540
540
541
541
542
542
543
543
544
544
545
545
546
546
547
547
548
548
549
549
550
550
551
551
552
552
553
553
554
554
555
555
556
556
557
557
558
558
559
559
560
560
561
561
562
562
563
563
564
564
565
565
566
566
567
567
568
568
569
569
570
570
571
571
572
572
573
573
574
574
575
575
576
576
577
577
578
578
579
579
580
580
581
581
582
582
583
583
584
584
585
585
586
586
587
587
588
588
589
589
590
590
591
591
592
592
593
593
594
594
595
595
596
596
597
597
598
598
599
599
600
600
601
601
602
602
603
603
604
604
605
605
606
606
607
607
608
608
609
609
610
610
611
611
612
612
613
613
614
614
615
615
616
616
617
617
618
618
619
619
620
620
621
621
622
622
623
623
624
624
625
625
626
626
627
627
628
628
629
629
630
630
631
631
632
632
633
633
634
634
635
635
636
636
637
637
638
638
639
639
640
640
641
641
642
642
643
643
644
644
645
645
646
646
647
647
648
648
649
649
650
650
651
651
652
652
653
653
654
654
655
655
656
656
657
657
658
658
659
659
660
660
661
661
662
662
663
663
664
664
665
665
666
666
667
667
668
668
669
669
670
670
671
671
672
672
673
673
674
674
675
675
676
676
677
677
678
678
679
679
680
680
681
681
682
682
683
683
684
684
685
685
686
686
687
687
688
688
689
689
690
690
691
691
692
692
693
693
694
694
695
695
696
696
697
697
698
698
699
699
700
700
701
701
702
702
703
703
704
704
705
705
706
706
707
707
708
708
709
709
710
710
711
711
712
712
713
713
714
714
715
715
716
716
717
717
718
718
719
719
720
720
721
721
722
722
723
723
724
724
725
725
726
726
727
727
728
728
729
729
730
730
731
731
732
732
733
733
734
734
735
735
736
736
737
737
738
738
739
739
740
740
741
741
742
742
743
743
744
744
745
745
746
746
747
747
748
748
749
749
750
750
751
751
752
752
753
753
754
754
755
755
756
756
757
757
758
758
759
759
760
760
761
761
762
762
763
763
764
764
765
765
766
766
767
767
768
768
769
769
770
770
771
771
772
772
773
773
774
774
775
775
776
776
777
777
778
778
779
779
780
780
781
781
782
782
783
783
784
784
785
785
786
786
787
787
788
788
789
789
790
790
791
791
792
792
793
793
794
794
795
795
796
796
797
797
798
798
799
799
800
800
801
801
802
802
803
803
804
804
805
805
806
806
807
807
808
808
809
809
810
810
811
811
812
812
813
813
814
814
815
815
816
816
817
817
818
818
819
819
820
820
821
821
822
822
823
823
824
824
825
825
826
826
827
827
828
828
829
829
830
830
831
831
832
832
833
833
834
834
835
835
836
836
837
837
838
838
839
839
840
840
841
841
842
842
843
843
844
844
845
845
846
846
847
847
848
848
849
849
850
850
851
851
852
852
853
853
854
854
855
855
856
856
857
857
858
858
859
859
860
860
861
861
862
862
863
863
864
864
865
865
866
866
867
867
868
868
869
869
870
870
871
871
872
872
873
873
874
874
875
875
876
876
877
877
878
878
879
879
880
880
881
881
882
882
883
883
884
884
885
885
886
886
887
887
888
888
889
889
890
890
891
891
892
892
893
893
894
894
895
895
896
896
897
897
898
898
899
899
900
900
901
901
902
902
903
903
904
904
905
905
906
906
907
907
908
908
909
909
910
910
911
911
912
912
913
913
914
914
915
915
916
916
917
917
918
918
919
919
920
920
921
921
922
922
923
923
924
924
925
925
926
926
927
927
928
928
929
929
930
930
931
931
932
932
933
933
934
934
935
935
936
936
937
937
938
938
939
939
940
940
941
941
942
942
943
943
944
944
945
945
946
946
947
947
948
948
949
949
950
950
951
951
952
952
953
953
954
954
955
955
956
956
957
957
958
958
959
959
960
960
961
961
962
962
963
963
964
964
965
965
966
966
967
967
968
968
969
969
970
970
971
971
972
972
973
973
974
974
975
975
976
976
977
977
978
978
979
979
980
980
981
981
982
982
983
983
984
984
985
985
986
986
987
987
988
988
989
989
990
990
991
991
992
992
993
993
994
994
995
995
996
996
997
997
998
998
999
999
999
999
```


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.

```
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
>>> 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.

```
1 $ pytest --timeout=10 --no-header --tb=short --doctest-modules
2     ↪ fraction_decimal_str_err.py
3 ===== test session starts
4     ↪ =====
5 collected 1 item
6
7 fraction_decimal_str_err.py F [100%]
8 ===== FAILURES
9     ↪ =====
10    ----- [doctest] fraction_decimal_str_err.Fraction.decimal_str
11    ↪ -----
12 038      '0.5'
13 039      >>> Fraction(1, 3).decimal_str(10)
14 040      '0.3333333333'
15 041      >>> Fraction(-101001, 100000000).decimal_str()
16 042      '-0.00101001'
17 043      >>> Fraction(1235, 1000).decimal_str(2)
18 044      '1.24'
19 045      >>> Fraction(99995, 100000).decimal_str(5)
20 046      '0.99995'
21 047      >>> Fraction(91995, 100000).decimal_str(3)
22
23 Expected:
24     '0.92'
25 Got:
26     '0.9110'
27
28 /home/runner/work/programmingWithPythonSlidesDE2/
29     ↪ programmingWithPythonSlidesDE2/slides/47_debugger/_git__realm_
30     ↪ git/gh_thomasWeise_programmingWithPythonCode/dunder/
31     ↪ fraction_decimal_str_err.py:47: DocTestFailure
32 ===== short test summary info
33     ↪ =====
34 FAILED fraction_decimal_str_err.py::fraction_decimal_str_err.Fraction.
35     ↪ decimal_str
36 ===== 1 failed in 0.02s
37     ↪ =====
38 # pytest 9.0.2 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 uns, dass `Fraction(91995, 100000).decimal_str(3)` nicht wie erwartet "**0.92**" liefert.

```
1 $ pytest --timeout=10 --no-header --tb=short --doctest-modules
2                                     ↪ fraction_decimal_str_err.py
3 ===== test session starts
4                                     ↪ =====
5 collected 1 item
6
7 fraction_decimal_str_err.py F [100%]
8 ===== FAILURES
9                                     ↪ =====
10 ----- [doctest] fraction_decimal_str_err.Fraction.decimal_str
11                                     ↪ -----
12 038      '0.5'
13 039      >>> Fraction(1, 3).decimal_str(10)
14 040      '0.3333333333'
15 041      >>> Fraction(-101001, 100000000).decimal_str()
16 042      '-0.00101001'
17 043      >>> Fraction(1235, 1000).decimal_str(2)
18 044      '1.24'
19 045      >>> Fraction(99995, 100000).decimal_str(5)
20 046      '0.99995'
21 047      >>> Fraction(91995, 100000).decimal_str(3)
22
23 Expected:
24      '0.92'
25 Got:
26      '0.9110'
27
28 /home/runner/work/programmingWithPythonSlidesDE2/
29                                     ↪ programmingWithPythonSlidesDE2/slides/47_debugger/_git__realm_
30                                     ↪ git/gh_thomasWeise_programmingWithPythonCode/dunder/
31                                     ↪ fraction_decimal_str_err.py:47: DocTestFailure
32 ===== short test summary info
33                                     ↪ =====
34 FAILED fraction_decimal_str_err.py::fraction_decimal_str_err.Fraction.
35                                     ↪ decimal_str
36 ===== 1 failed in 0.02s
37                                     ↪ =====
38 # pytest 9.0.2 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 uns, dass `Fraction(91995, 100000).decimal_str(3)` nicht wie erwartet "0.92" liefert.
- Stattdessen bekommen wir "0.9110".

```
1 $ pytest --timeout=10 --no-header --tb=short --doctest-modules
2     ↪ fraction_decimal_str_err.py
3 ===== test session starts
4     ↪ =====
5 collected 1 item
6
7 fraction_decimal_str_err.py F [100%]
8 ===== FAILURES
9     ↪ =====
10    ----- [doctest] fraction_decimal_str_err.Fraction.decimal_str
11    ↪ -----
12 038      '0.5'
13 039      >>> Fraction(1, 3).decimal_str(10)
14 040      '0.3333333333'
15 041      >>> Fraction(-101001, 100000000).decimal_str()
16 042      '-0.00101001'
17 043      >>> Fraction(1235, 1000).decimal_str(2)
18 044      '1.24'
19 045      >>> Fraction(99995, 100000).decimal_str(5)
20 046      '0.99995'
21 047      >>> Fraction(91995, 100000).decimal_str(3)
22
23 Expected:
24     ↪ '0.92'
25 Got:
26     ↪ '0.9110'
27
28 /home/runner/work/programmingWithPythonSlidesDE2/
29     ↪ programmingWithPythonSlidesDE2/slides/47_debugger/_git__realm_
30     ↪ git/gh_thomasWeise_programmingWithPythonCode/dunder/
31     ↪ fraction_decimal_str_err.py:47: DocTestFailure
32 ===== short test summary info
33     ↪ =====
34 FAILED fraction_decimal_str_err.py::fraction_decimal_str_err.Fraction.
35     ↪ decimal_str
36 ===== 1 failed in 0.02s
37     ↪ =====
38 # pytest 9.0.2 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 uns, 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?

```
1 $ pytest --timeout=10 --no-header --tb=short --doctest-modules
2     ↪ fraction_decimal_str_err.py
3 ===== test session starts
4     ↪ =====
5 collected 1 item
6
7 fraction_decimal_str_err.py F [100%]
8 ===== FAILURES
9     ↪ =====
10 ----- [doctest] fraction_decimal_str_err.Fraction.decimal_str
11     ↪ -----
12 038      '0.5'
13 039      >>> Fraction(1, 3).decimal_str(10)
14 040      '0.3333333333'
15 041      >>> Fraction(-101001, 100000000).decimal_str()
16 042      '-0.00101001'
17 043      >>> Fraction(1235, 1000).decimal_str(2)
18 044      '1.24'
19 045      >>> Fraction(99995, 100000).decimal_str(5)
20 046      '0.99995'
21 047      >>> Fraction(91995, 100000).decimal_str(3)
22 Expected:
23     '0.92'
24 Got:
25     '0.9110'
26
27 /home/runner/work/programmingWithPythonSlidesDE2/
28     ↪ programmingWithPythonSlidesDE2/slides/47_debugger/_git__realm_
29     ↪ git/gh_thomasWeise_programmingWithPythonCode/dunder/
30     ↪ fraction_decimal_str_err.py:47: DocTestFailure
31 ===== short test summary info
32     ↪ =====
33 FAILED fraction_decimal_str_err.py::fraction_decimal_str_err.Fraction.
34     ↪ decimal_str
35 ===== 1 failed in 0.02s
36     ↪ =====
37 # pytest 9.0.2 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 uns, 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?

```
1 $ pytest --timeout=10 --no-header --tb=short --doctest-modules
2     ↪ fraction_decimal_str_err.py
3 ===== test session starts
4     ↪ =====
5 collected 1 item
6
7 fraction_decimal_str_err.py F [100%]
8 ===== FAILURES
9     ↪ =====
10 ----- [doctest] fraction_decimal_str_err.Fraction.decimal_str
11     ↪ -----
12 038      '0.5'
13 039      >>> Fraction(1, 3).decimal_str(10)
14 040      '0.3333333333'
15 041      >>> Fraction(-101001, 100000000).decimal_str()
16 042      '-0.00101001'
17 043      >>> Fraction(1235, 1000).decimal_str(2)
18 044      '1.24'
19 045      >>> Fraction(99995, 100000).decimal_str(5)
20 046      '0.99995'
21 047      >>> Fraction(91995, 100000).decimal_str(3)
22 Expected:
23     '0.92'
24 Got:
25     '0.9110'
26
27 /home/runner/work/programmingWithPythonSlidesDE2/
28     ↪ programmingWithPythonSlidesDE2/slides/47_debugger/_git__realm_
29     ↪ git/gh_thomasWeise_programmingWithPythonCode/dunder/
30     ↪ fraction_decimal_str_err.py:47: DocTestFailure
31 ===== short test summary info
32     ↪ =====
33 FAILED fraction_decimal_str_err.py::fraction_decimal_str_err.Fraction.
34     ↪ decimal_str
35 ===== 1 failed in 0.02s
36     ↪ =====
37 # pytest 9.0.2 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 uns, 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
  ↪ -----
038     '0.5'
039     >>> Fraction(1, 3).decimal_str(10)
040     '0.3333333333'
041     >>> Fraction(-101001, 100000000).decimal_str()
042     '-0.00101001'
043     >>> Fraction(1235, 1000).decimal_str(2)
044     '1.24'
045     >>> Fraction(99995, 100000).decimal_str(5)
046     '0.99995'
047     >>> Fraction(91995, 100000).decimal_str(3)

Expected:
  '0.92'
Got:
  '0.9110'

/home/runner/work/programmingWithPythonSlidesDE2/
  ↪ programmingWithPythonSlidesDE2/slides/47_debugger/_git__realm/
  ↪ git/gh_thomasWeise_programmingWithPythonCode/dunder/
  ↪ fraction_decimal_str_err.py:47: DocTestFailure
=====
      short test summary info
  ↪ =====
FAILED fraction_decimal_str_err.py::fraction_decimal_str_err.Fraction.
  ↪ decimal_str
=====
      1 failed in 0.02s
  ↪ =====
# pytest 9.0.2 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.

A screenshot of the PyCharm IDE interface. The project navigation bar shows a file named 'fraction_decimal_str_err.py'. The code editor displays a class 'Fraction' with a method 'decimal_str'. A context menu is open over the code, with the 'Run' option highlighted. The menu includes options like 'Show Context Actions', 'AI Actions', 'Paste', 'Copy / Paste Special', 'Column Selection Mode', 'Go To', 'Folding', 'Refactor', 'Generate...', and 'Run 'Doctest decimal_str''. The background shows a blurred view of a university campus.

```
<class 'Fraction'>: 8 usages ± Thomas Weise
  def decimal_str(self, max_frac: int = 100) -> str: 9 usages (1 dynamic) ± Thomas Weise
    """
    Convert the fraction to decimal string.

    :param max_frac: the maximum number of fractional digits
    :return: the string
    >>> Fraction(1
    '62'
    >>> Fraction(1
    '0.5'
    >>> Fraction(1
    '0.3333333333'
    >>> Fraction(-
    '-0.00101001'
    >>> Fraction(1
    '1.24'
    >>> Fraction(9
    '0.99995'
    >>> Fraction(9
    '0.92'
    >>> Fraction(9
    '1'
```

Project tree:

- simple_list_compr
- simple_set_compr
- zip.py
- 08_classes
 - circle.py
 - circle_user.py
 - kahan_sum.py
 - kahan_user.py
 - point.py
 - point_user.py
 - polygon.py
 - rectangle.py
 - shape.py
 - triangle.py
- 09_dunder
 - fraction.py
 - fraction_decimal_str_err.py
 - fraction_sqrt.py
 - point.py
 - point_user_2.py
 - point_with_dunde



Doctests in PyCharm

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

The screenshot shows the PyCharm IDE interface. On the left is the project tree with several files under '08_classes' and '09_dunder'. The main editor window displays the code for the `decimal_str` method:

```
class Fraction: 8 usages ± Thomas Weise
    def decimal_str(self, max_frac: int = 100) -> str: 9 usages (1 dynamic) ± Thomas Weise
        """
        Convert the fraction to decimal string.

        :param max_frac: the maximum number of fractional digits
        :return: the string
    """
    >>> Fraction(1
    '62'
    >>> Fraction(1
    '0.5'
    >>> Fraction(1
    '0.3333333333'
    >>> Fraction(-
    '-0.00101001'
    >>> Fraction(1
    '1.24'
    >>> Fraction(9
    '0.99995'
    >>> Fraction(9
    '0.92'
    >>> Fraction(9
    '1'
```

A context menu is open over the first line of the docstring, showing options like 'Show Context Actions', 'AI Actions', 'Paste', 'Column Selection Mode', 'Go To', 'Folding', 'Refactor', 'Generate...', 'Run 'Doctest decimal_str'', 'Debug 'Doctest decimal_str'', and 'Modify Run Configuration...'. The 'Run 'Doctest decimal_str'' option is highlighted with a blue border.



Doctests in PyCharm

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

The screenshot shows the PyCharm IDE interface. On the left is the Project tool window displaying a file tree with several Python files like `simple_list_compr.py`, `simple_set_compr.py`, `zip.py`, and several files under `08_classes` and `09_dunder` directories. The current file is `fraction_decimal_str_err.py`. In the code editor, there's a class `Fraction` with a method `decimal_str`. A context menu is open over the code, with the option `Run 'Doctest decimal_str'` highlighted at the bottom.

```
<class 'Fraction'>: 8 usages ± Thomas Weise
  def decimal_str(self, max_fractions: int = 100) -> str: 9 usages (1 dynamic) ± Thomas Weise
    """
    Convert the fraction to decimal string.

    :param max_fractions: the maximum number of fractional digits
    :return: the string
    >>> Fraction(1
    '62'
    >>> Fraction(1
    '0.5'
    >>> Fraction(1
    '0.3333333333'
    >>> Fraction(-
    '-0.00101001'
    >>> Fraction(1
    '1.24'
    >>> Fraction(9
    '0.99995'
    >>> Fraction(9
    '0.92'
    >>> Fraction(9
    '1'
```

Context menu options (from top to bottom):

- Show Context Actions
- AI Actions
- Paste
- Copy / Paste Special
- Column Selection Mode
- Go To
- Folding
- Refactor
- Generate...
- Run 'Doctest decimal_str'
- Debug 'Doctest decimal_str'
- Modify Run Configuration...

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'`.

The screenshot shows the PyCharm IDE interface. On the left is the project tree with files like `simple_list_compr.py`, `simple_set_compr.py`, `zip.py`, `08_classes` (containing `circle.py`, `circle_user.py`, `kahan_sum.py`, `kahan_user.py`, `point.py`, `point_user.py`, `polygon.py`, `rectangle.py`, `shape.py`, `triangle.py`), and `09_dunder` (containing `fraction.py`, `fraction_decimal_err.py`, `fraction_sqrt.py`, `point.py`, `point_user_2.py`, `point_with_dunde`, `point_with_dunde`). The main editor window displays the code for `fraction_decimal_err.py`. A context menu is open over the code, with the option `Run 'Doctest decimal_str'` highlighted at the bottom.

```
<code>class Fraction:
    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(1
        '62'
        >>> Fraction(1
        '0.5'
        >>> Fraction(1
        '0.3333333333'
        >>> Fraction(-
        '-0.00101001'
        >>> Fraction(1
        '1.24'
        >>> Fraction(9
        '0.99995'
        >>> Fraction(9
        '0.92'
        >>> Fraction(9
        '1'</code>
```

Doctests in PyCharm

- Dadurch werden *alle* Doctests ausgeführt.

The screenshot shows the PyCharm IDE interface. The top navigation bar includes tabs for 'PC', 'Program', 'main', and 'Doctest Fraction.decimal_str'. The left sidebar shows a project structure with files like simple_list_compr, simple_set_compr, zip.py, and several files under the 08_classes directory (circle.py, circle_user.py, kahan_sum.py, kahan_user.py, point.py). The main editor window displays code for the Fraction class, specifically its decimal_str method. The code includes several doctests:

```
class Fraction:
    def decimal_str(self, max_frac: int = 100) -> str:
        ...
        if self == 0:
            return '0.0'
        sign = '-' if self < 0 else ''
        self = abs(self)
        integer_part = self // 1
        fractional_part = self % 1 * 10**max_frac
        if fractional_part == 0:
            return f'{sign}{integer_part}.0' + '0' * max_frac
        else:
            return f'{sign}{integer_part}.{fractional_part}'
```

The run results window at the bottom shows the output of the doctests:

```
Tests failed: 2, passed: 6 of 8 tests – 0ms
/home/tweisse/local/programming/python/programmingWithPythonCode/.venv/bin/python
Testing started at 13:30 ...
```

Doctests in PyCharm



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

The screenshot shows the PyCharm IDE interface. The top navigation bar includes tabs for 'PC', 'Run', and 'Programs'. The main window displays a project structure with files like simple_list_compr, simple_set_compr, zip.py, and several Fraction-related files. The code editor shows a Fraction class with a decimal_str method and several doctests. The bottom 'Run' tab shows the output of a 'Doctest Fraction.decimal_str' run. It indicates 2 tests failed and 6 passed. The terminal output shows the command run and the start time.

```
class Fraction:
    def decimal_str(self, max_frac: int = 100) -> str:
        ...
        if self == 0:
            return '0.0'
        sign = '-' if self < 0 else ''
        self = abs(self)
        integer_part = self // max_frac
        remainder = self % max_frac
        if remainder == 0:
            return f'{sign}{integer_part}'
        else:
            return f'{sign}{integer_part}.{"0' * (max_frac - 1)}{remainder}'

    >>> 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)
```

Run → Doctest Fraction.decimal_str

Test Results

Tests failed: 2, passed: 6 of 8 tests – 0ms

```
/home/tweise/local/programming/python/programmingWithPythonCode/.venv/bin/python
Testing started at 13:30 ...
```

Doctests in PyCharm



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

The screenshot shows the PyCharm IDE interface. The project navigation bar at the top has tabs for 'Project' (selected), 'fractions', 'main', and 'Doctest Fraction.decimal_str'. The code editor shows a file named 'fraction_decimal_str_err.py' containing Python code for a 'Fraction' class and its 'decimal_str' method. The run tab shows a 'Run' configuration for 'Doctest Fraction.decimal_str'. The 'Run' tool window displays test results: 'Test Results' section shows 'Tests failed: 2, passed: 6 of 8 tests – 0ms'. Under the 'decimal_str' test, there is a 'Failure' message with a link '[Click to see difference](#)'. Below this, the terminal output shows a failed example:

```
*****
File "/home/tweisse/local/programming/python/programmingWithPythonCode/09_du
Failed example:
    Fraction(91995, 100000).decimal_str(3)
Expected:
    '0.92'
Got:
    '0.9110'
```

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.

The screenshot shows the PyCharm IDE interface. The top navigation bar displays 'Doctest Fraction.decimal_str' and the file 'fraction_decimal_str_err.py'. The left sidebar shows a project structure with files like 'simple_list_compr', 'simple_set_compr', 'zip.py', and '08_classes'. The main editor window shows code for a 'Fraction' class and its decimal representation. The 'Run' tab is selected, showing a 'Test Results' section with 'Tests failed: 2, passed: 6 of 8 tests – 0ms'. A specific test failure for 'decimal_str' is highlighted, showing a 'Failure' message: 'Fraction(91995, 100000).decimal_str(3)'. The 'Expected' result is '0.92' and the 'Got' result is '0.9110'. A circular icon with a question mark is visible at the bottom left of the editor area.

```
class Fraction:
    def decimal_str(self):
        return '0.5'
    def decimal_str(self, n):
        return '0.5'

Test Results
Tests failed: 2, passed: 6 of 8 tests – 0ms
decimal_str
Fraction(91995, 100000)
Fraction(99995, 100000)

Failure
<Click to see difference>

*****
File "/home/tweisse/local/programming/python/programmingWithPythonCode/09_du...
Failed example:
    Fraction(91995, 100000).decimal_str(3)
Expected:
    '0.92'
Got:
    '0.9110'
```



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.

The screenshot shows the PyCharm IDE interface. The top navigation bar includes tabs for 'PC', 'Run', and 'Programs'. The 'Run' tab is active, showing a 'Doctest Fraction.decimal_str' run. The 'Project' tool window on the left lists files like 'simple_list_compr.py', 'simple_set_compr.py', 'zip.py', and '08_classes/fraction_decimal_str_err.py'. The code editor on the right displays the following code from 'fraction_decimal_str_err.py':

```
class Fraction:
    def decimal_str():
        return '0.5'
    def decimal_str(10):
        return '0.5'
```

The 'Run' tool window below shows test results:

- Test Results: 0 ms
- decimal_str: 0 ms
- Fraction(91995, 100000): 10 ms
- Fraction(99995, 100000): 10 ms

A red error icon next to the first test indicates failure. The message 'Tests failed: 2, passed: 6 of 8 tests - 0ms' is displayed. A 'Failure' section shows the expected output '0.92' and the actual output '0.9110'.

```
*****
File "/home/tweisse/local/programming/python/programmingWithPythonCode/09_du...
Failed example:
    Fraction(91995, 100000).decimal_str(3)
Expected:
    '0.92'
Got:
    '0.9110' [
```

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.

The screenshot shows the PyCharm IDE interface with the following details:

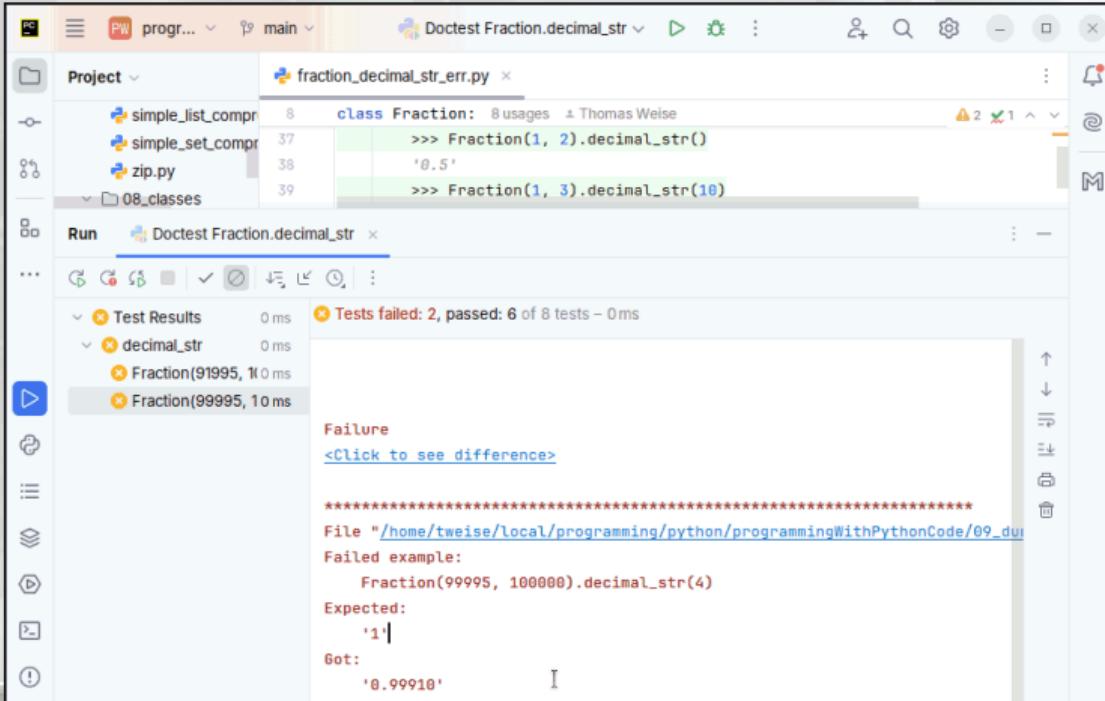
- Project:** fraction_decimal_str_err.py
- Run:** Doctest Fraction.decimal_str
- Test Results:** Tests failed: 2, passed: 6 of 8 tests – 0ms
- Failed Example:** Fraction(91995, 100000).decimal_str(3)
- Expected:** '0.92'
- Got:** '0.9110'

```
*****
File "/home/tweisse/local/programming/python/programmingWithPythonCode/09_du...
Failed example:
    Fraction(91995, 100000).decimal_str(3)
Expected:
    '0.92'
Got:
    '0.9110' [
```



Doctests in PyCharm

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



The screenshot shows the PyCharm IDE interface. The project navigation bar at the top has tabs for "Project", "Run", and "Tools". The "Run" tab is selected, showing a list of tests under "Test Results". One test, "decimal_str", failed, indicated by a red "X". The failure message is displayed below:

```
Failure
<Click to see difference>

*****
File "/home/tweisse/local/programming/python/programmingWithPythonCode/09_du
Failed example:
    Fraction(99995, 100000).decimal_str(4)
Expected:
    '1'
Got:
    '0.99910'
```



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

The screenshot shows the PyCharm IDE interface with the following details:

- Project:** fraction_decimal_str_err.py
- Run:** Doctest Fraction.decimal_str
- Test Results:** Tests failed: 2, passed: 6 of 8 tests – 0ms
- Failed Test:** decimal_str
- Failure Details:**
 - `<Click to see difference>`
 - File: "/home/tweisse/local/programming/python/programmingWithPythonCode/09_du"
 - Failed example:
 `Fraction(99995, 100000).decimal_str(4)`
 - Expected:
 `'1'`
 - Got:
 `'0.99910'`

Doctests in PyCharm



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

The screenshot shows the PyCharm IDE interface. The top navigation bar includes tabs for "PC", "PWI", "program...", "main", and "Doctest Fraction.decimal_str". The left sidebar displays the project structure with files like "simple_list_compr", "simple_set_compr", "zip.py", and "08_classes". The main editor window shows code for a "Fraction" class and its decimal representation. The "Run" tab is selected, showing test results for "decimal_str". The results indicate 2 tests failed and 6 passed. A detailed failure report is shown for the test "Fraction(99995, 100000).decimal_str(4)". The expected output was "1", but the actual output was "0.99910".

```
Doctest Fraction.decimal_str
=====
File "/home/tweisse/local/programming/python/programmingWithPythonCode/09_du
Failed example:
    Fraction(99995, 100000).decimal_str(4)
Expected:
    '1'
Got:
    '0.99910'
```



Doctests in PyCharm

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

The screenshot shows the PyCharm IDE interface. The top navigation bar includes tabs for 'PC', 'File', 'Edit', 'Run', 'View', 'Tools', 'Help', and 'PyCharm'. The 'Run' tab is selected. The main window displays a project structure with files like 'simple_list_compr.py', 'simple_set_compr.py', 'zip.py', and '08_classes/fraction_decimal_str_err.py'. The 'fraction_decimal_str_err.py' file is open, showing code for a 'Fraction' class and its 'decimal_str' method. The 'Run' tool window shows 'Test Results' for 'decimal_str' with 2 failures and 6 passed tests. The failure details show a comparison between expected output '1' and actual output '0.99910'. The bottom status bar indicates the file path is '/home/tweisse/local/programming/python/programmingWithPythonCode/09_du...' and the line number is 'Failed example: 10'.

```
Doctest Fraction.decimal_str
=====
File "/home/tweisse/local/programming/python/programmingWithPythonCode/09_du...
Failed example:
    Fraction(99995, 100000).decimal_str(4)
Expected:
    '1'
Got:
    '0.99910'
```



Doctests in PyCharm

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

The screenshot shows the PyCharm IDE interface with the following details:

- Project:** fraction_decimal_str_err.py
- Run:** Doctest Fraction.decimal_str
- Test Results:** Tests failed: 2, passed: 6 of 8 tests – 0ms
- Failure:** [Click to see difference](#)
- File:** File "/home/tweisse/local/programming/python/programmingWithPythonCode/09_du"
- Failed example:** Fraction(99995, 100000).decimal_str(4)
- Expected:** '1'
- Got:** '0.99910'



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.

The screenshot shows the PyCharm IDE interface. The top navigation bar includes tabs for 'PC', 'PWI', 'program...', 'main...', and 'Doctest Fraction.decimal_str'. The left sidebar displays the project structure with files like 'simple_list_compr', 'simple_set_compr', 'zip.py', and '08_classes'. The main editor window shows code for a 'Fraction' class and its 'decimal_str' method. The 'Run' tab is selected, showing 'Doctest Fraction.decimal_str' with a status of 'Tests failed: 2, passed: 6 of 8 tests – 0ms'. The 'Test Results' section details two failed tests: one for 'decimal_str' and another for 'Fraction(91995, 10)'. The failure message indicates a 'Failure' with a link to 'Click to see difference'. The terminal output at the bottom shows the file path '/home/tweisse/local/programming/python/programmingWithPythonCode/09_du...' and a failed example where 'Fraction(99995, 100000).decimal_str(4)' resulted in '1' instead of '0.99910'.

```
Doctest Fraction.decimal_str
=====
File "/home/tweisse/local/programming/python/programmingWithPythonCode/09_du...
Failed example:
    Fraction(99995, 100000).decimal_str(4)
Expected:
    '1'
Got:
    '0.99910'
```



Doctests in PyCharm

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

The screenshot shows the PyCharm IDE interface. The top bar displays the project name "Doctest Fraction.decimal_str" and the file "fraction_decimal_str_err.py". The left sidebar shows the project structure with files like "simple_list_compr", "simple_set_compr", "zip.py", and "08_classes". The "Run" tab is selected, showing a "Test Results" section with "Tests failed: 2, passed: 6 of 8 tests – 0ms". Below this, under "decimal_str", there are two failed examples:

- Failure
[Click to see difference](#)
- File "[/home/tweisse/local/programming/python/programmingWithPythonCode/09_du](#)
- Failed example:
Fraction(99995, 100000).decimal_str(4)
- Expected:
'1'
- Got:
'0.99910'



Doctests in PyCharm

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

The screenshot shows the PyCharm IDE interface. The top navigation bar includes tabs for 'PC', 'Program', 'main', and 'Doctest Fraction.decimal_str'. The left sidebar displays the project structure with files like 'simple_list_compr', 'simple_set_compr', 'zip.py', and '08_classes'. The main editor window shows code for a 'Fraction' class with two doctests:

```
class Fraction:
    ...
    >>> Fraction(1, 2).decimal_str()
    '0.5'
    >>> Fraction(1, 3).decimal_str(10)
```

The 'Run' tab is selected, showing 'Doctest Fraction.decimal_str' with a status of 'Tests failed: 2, passed: 6 of 8 tests – 0ms'. The 'Test Results' section details the failure:

- Test 'decimal_str': Failed (0 ms)
- Test 'Fraction(91995, 10)': Failed (0 ms)
- Test 'Fraction(99995, 10)': Failed (0 ms)

The 'Failure' section provides the difference between expected and actual results:

```
<Click to see difference>
```

File "/home/tweisse/local/programming/python/programmingWithPythonCode/09_du
Failed example:
 Fraction(99995, 100000).decimal_str(4)
Expected:
 '1'
Got:
 '0.99910'

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.



- 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.



- 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 erlaubt uns, ein Programm Schritt-für-Schritt auszuführen und dabei die aktuellen Werte von Variablen zu beobachten.



- 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.

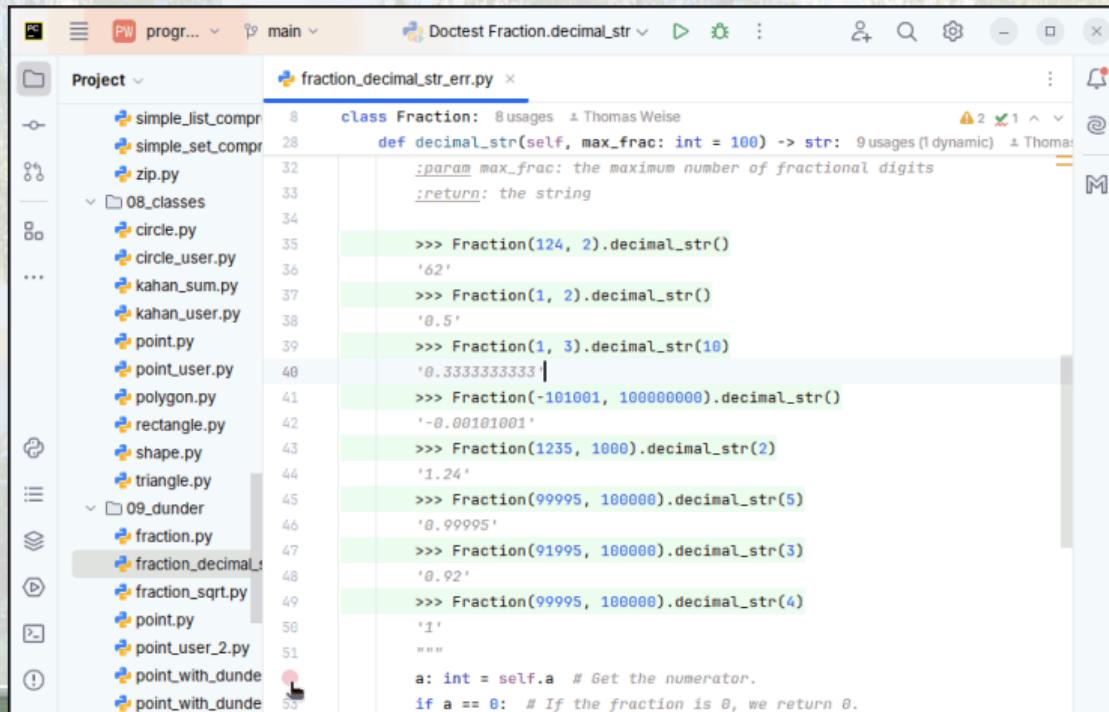
Nützliches Werkzeug

Ein Debugger ist ein Werkzeug, das mit vielen Programmiersprachen und IDEs mit ausgeliefert wird. Es erlaubt uns, ein Programm Schritt-für-Schritt auszuführen und dabei die aktuellen Werte von Variablen zu beobachten. So können wir Fehler im Kode leichter finden^{1,25,40}.

Debuggen in PyCharm



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



The screenshot shows the PyCharm IDE interface with the following details:

- Project View:** Shows a tree structure of Python files in the project. Files visible include simple_list_compr.py, simple_set_compr.py, zip.py, 08_classes/circle.py, 08_classes/circle_user.py, 08_classes/kahan_sum.py, 08_classes/kahan_user.py, 08_classes/point.py, 08_classes/point_user.py, 08_classes/polygon.py, 08_classes/rectangle.py, 08_classes/shape.py, 08_classes/triangle.py, 09_dunder/fraction.py, 09_dunder/fraction_decimal_str_err.py, 09_dunder/fraction_sqrt.py, 09_dunder/point.py, 09_dunder/point_user_2.py, 09_dunder/point_with_dunder.py, and 09_dunder/point_with_dunde.py.
- Editor View:** The main window displays the code for `fraction_decimal_str_err.py`. The code defines a `Fraction` class and includes several doctests for its `decimal_str` method. The doctests show various outputs for different inputs, such as '62', '0.5', and '0.3333333333'. A cursor is visible at the end of the last doctest line.
- Status Bar:** At the bottom, the status bar shows the path "Doctest Fraction.decimal_str" and the line number "53".

Debuggen in PyCharm



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

```
Project: fraction_decimal_str_err.py
File: fraction_decimal_str_err.py

class Fraction:
    def decimal_str(self, max_frac: int = 100) -> str:
        """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.
```

Debuggen in PyCharm



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

The screenshot shows the PyCharm IDE interface. On the left, the Project tool window displays a file structure with several Python files under '08_classes' and '09_dunder' directories. The main code editor window is open to a file named 'fraction_decimal_str_err.py'. The code defines a class Fraction and its decimal representation. A cursor is visible at the end of the class definition. The code editor has a light blue background with syntax highlighting for different parts of the code. The status bar at the bottom indicates the current file is 'fraction_decimal_str_err.py'.

```
8     class Fraction: 8 usages ± Thomas Weise
28         def decimal_str(self, max_fractions: int = 100) -> str: 9 usages (1 dynamic) ± Thomas Weise
32             :param max_fractions: the maximum number of fractional digits
33             :return: the string
35             >>> Fraction(124, 2).decimal_str()
36             '62'
37             >>> Fraction(1, 2).decimal_str()
38             '0.5'
39             >>> Fraction(1, 3).decimal_str(10)
40             '0.3333333333'
41             >>> Fraction(-101001, 100000000).decimal_str()
42             '-0.00101001'
43             >>> Fraction(1235, 1000).decimal_str(2)
44             '1.24'
45             >>> Fraction(99995, 100000).decimal_str(5)
46             '0.99995'
47             >>> Fraction(91995, 100000).decimal_str(3)
48             '0.92'
49             >>> Fraction(99995, 100000).decimal_str(4)
50             '1'
51             """
52             a: int = self.a # Get the numerator.
53             if a == 0: # If the fraction is 0, we return 0.
```

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.

```
Doctest Fraction.decimal_str ✓
```

```
Project fraction_decimal_str_err.py
```

```
8 class Fraction: 8 usages ± Thomas Weise
28     def decimal_str(self, max_fractions: int = 100) -> str: 9 usages (1 dynamic) ± Thomas Weise
32         :param max_fractions: the maximum number of fractional digits
33         :return: the string
35         >>> Fraction(124, 2).decimal_str()
36         '62'
37         >>> Fraction(1, 2).decimal_str()
38         '0.5'
39         >>> Fraction(1, 3).decimal_str(10)
40         '0.3333333333'
41         >>> Fraction(-101001, 100000000).decimal_str()
42         '-0.00101001'
43         >>> Fraction(1235, 1000).decimal_str(2)
44         '1.24'
45         >>> Fraction(99995, 100000).decimal_str(5)
46         '0.99995'
47         >>> Fraction(91995, 100000).decimal_str(3)
48         '0.92'
49         >>> Fraction(99995, 100000).decimal_str(4)
50         '1'
51         """
53
a: int = self.a # Get the numerator.
if a == 0: # If the fraction is 0, we return 0.
```

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.

The screenshot shows the PyCharm IDE interface. On the left is the Project tool window displaying a file tree with Python files like simple_list_compr.py, simple_set_compr.py, zip.py, and several files under 08_classes and 09_dunder. The main editor window shows a portion of fraction_decimal_str_err.py. A red dot icon, representing a breakpoint, is visible in the gutter of the code editor at line 28. The code itself contains several print statements demonstrating the Fraction class's decimal representation.

```
class Fraction:
    def decimal_str(self, max_frac: int = 100) -> str:
        """param max_frac: the maximum number of fractional digits
        :return: the string
        """
        if self == 0:
            return "0"
        if self < 0:
            sign = "-"
            self *= -1
        else:
            sign = ""
        if max_frac < 0:
            raise ValueError("max_frac must be non-negative")
        if max_frac == 0:
            return sign + str(int(self))
        a, b = self.numerator, self.denominator
        if a % b == 0:
            return sign + str(a // b)
        if max_frac > len(str(b)) - 1:
            max_frac = len(str(b)) - 1
        digits = []
        while a > 0 and len(digits) < max_frac:
            a, r = divmod(a * 10, b)
            digits.append(str(r))
        if a == 0:
            return sign + "." + "".join(digits)
        else:
            return sign + str(a) + "."
    def __str__(self):
        if self == 0:
            return "0"
        if self < 0:
            sign = "-"
            self *= -1
        else:
            sign = ""
        if self == 1:
            return "1"
        if self < 1:
            if self > 0:
                return sign + self.__repr__()
            else:
                return sign + "-" + self.__repr__().lstrip("-")
        else:
            a, b = self.numerator, self.denominator
            if a % b == 0:
                return str(int(a // b))
            if a < 0:
                a = -a
                b = -b
            if b < 0:
                b = -b
            if a < b:
                a, b = b, a
            gcd = a // b
            a //= gcd
            b //= gcd
            if a == 1:
                return str(b)
            if b == 1:
                return str(a)
            return str(a) + "/" + str(b)
```

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.

The screenshot shows the PyCharm IDE interface. The project structure on the left includes files like simple_list_compr, simple_set_compr, zip.py, 08_classes (containing circle.py, circle_user.py, kahan_sum.py, kahan_user.py, point.py, point_user.py, polygon.py, rectangle.py, shape.py, triangle.py), and 09_dunder (containing fraction.py, fraction_decimal_str_err.py). The code editor on the right displays the file fraction_decimal_str_err.py. A red arrow points to the line `a: int = self.a # Get the numerator.`, which is highlighted in orange, indicating it is the current line of execution. The code defines a Fraction class with a decimal_str method that takes a max_fractions parameter and returns a string representation of the fraction.

```
class Fraction:
    def decimal_str(self, max_frac: int = 100) -> str:
        """param max_frac: the maximum number of fractional digits
        :return: the string
        """
        if self == 0:
            return "0"
        if self.denominator == 1:
            return str(self.numerator)
        if max_frac < 0:
            raise ValueError("max_frac must be non-negative")
        if max_frac == 0:
            return f'{self.numerator}/{self.denominator}'
        digits = []
        whole_part = self.numerator // self.denominator
        remainder = self.numerator % self.denominator
        if whole_part != 0:
            digits.append(str(whole_part))
        if remainder == 0:
            return ''.join(digits)
        if max_frac == 1:
            return f'{whole_part}.{"0" * len(digits)}{remainder}'
        digits.append('.')
        for _ in range(max_frac):
            remainder *= 10
            digit = remainder // self.denominator
            digits.append(str(digit))
            remainder %= self.denominator
            if remainder == 0:
                break
        else:
            digits.append("...")
        return ''.join(digits)

    @property
    def __float__(self):
        return self.numerator / self.denominator
```

Debuggen in PyCharm



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

The screenshot shows the PyCharm IDE interface with the following details:

- Title Bar:** Shows "Doctest Fraction.decimal_str" and other standard icons.
- Project View:** On the left, there's a tree view of the project structure. Under "08_classes", files like "circle.py", "kahan_sum.py", and "point.py" are listed. Under "09_dunder", "fraction.py" is listed. The file "fraction_decimal_str_err.py" is currently selected and shown in the main editor area.
- Editor Area:** The code for "fraction_decimal_str_err.py" is displayed:

```
class Fraction:
    def decimal_str(self, max_frac: int = 100) -> str:
        """param max_frac: the maximum number of fractional digits
        :return: the string
        """
        if self == 0:
            return "0"
        sign = "-" if self < 0 else ""
        self = abs(self)
        integer_part, decimal_part = divmod(self, 1)
        if decimal_part == 0:
            return f"{sign}{integer_part}"
        else:
            return f"{sign}{integer_part}.{'0' * max_frac + str(decimal_part)[2:max_frac+2]}"

    def __str__(self):
        return self.decimal_str()
```
- Breakpoint:** A red circular breakpoint icon is visible at the start of the first line of the `decimal_str` method definition (line 28).
- Toolbars and Status Bar:** Standard PyCharm toolbars and status bar are visible at the bottom.

Debuggen in PyCharm



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

The screenshot shows the PyCharm IDE interface. The left sidebar displays a project structure with several Python files under '08_classes' and '09_dunder' directories. The main editor window is titled 'fraction_decimal_str_err.py' and contains the following code:

```
class Fraction: 8 usages ± Thomas Weise
    def decimal_str(self, max_frac: int = 100) -> str: 9 usages (1 dynamic) ± Thomas:
        """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.
```

A red circular icon, representing a breakpoint, is positioned above the line number 53 in the code editor.

Debuggen in PyCharm



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

The screenshot shows the PyCharm IDE interface. On the left is the Project tool window displaying a file structure with several Python files like simple_list_compr.py, simple_set_compr.py, zip.py, etc. In the center, a code editor window is open for the file fraction_decimal_str_err.py. The cursor is positioned over the line of code: `>>> Fraction(124, 2).decimal_str()`. A context menu is open at this position, listing various actions such as Show Context Actions, AI Actions, Paste, Find Usages, Go To, Refactor, Generate..., Run 'Doctest decimal_str', and Debug 'Doctest decimal_str'. The 'Debug' option is highlighted with a blue selection bar. The status bar at the bottom right shows the text 'return 0.'

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.

The screenshot shows the PyCharm IDE interface. In the center, there is a code editor window displaying Python code. A context menu is open over a line of code in the editor. The menu has the following options:

- Show Context Actions
- AI Actions
- Paste
- Copy / Paste Special
- Column Selection Mode
- Find Usages
- Go To
- Folding
- Refactor
- Generate...
- Run 'Doctest decimal_str'
- Debug 'Doctest decimal_str'** (This option is highlighted with a blue selection bar.)
- Modify Run Configuration...

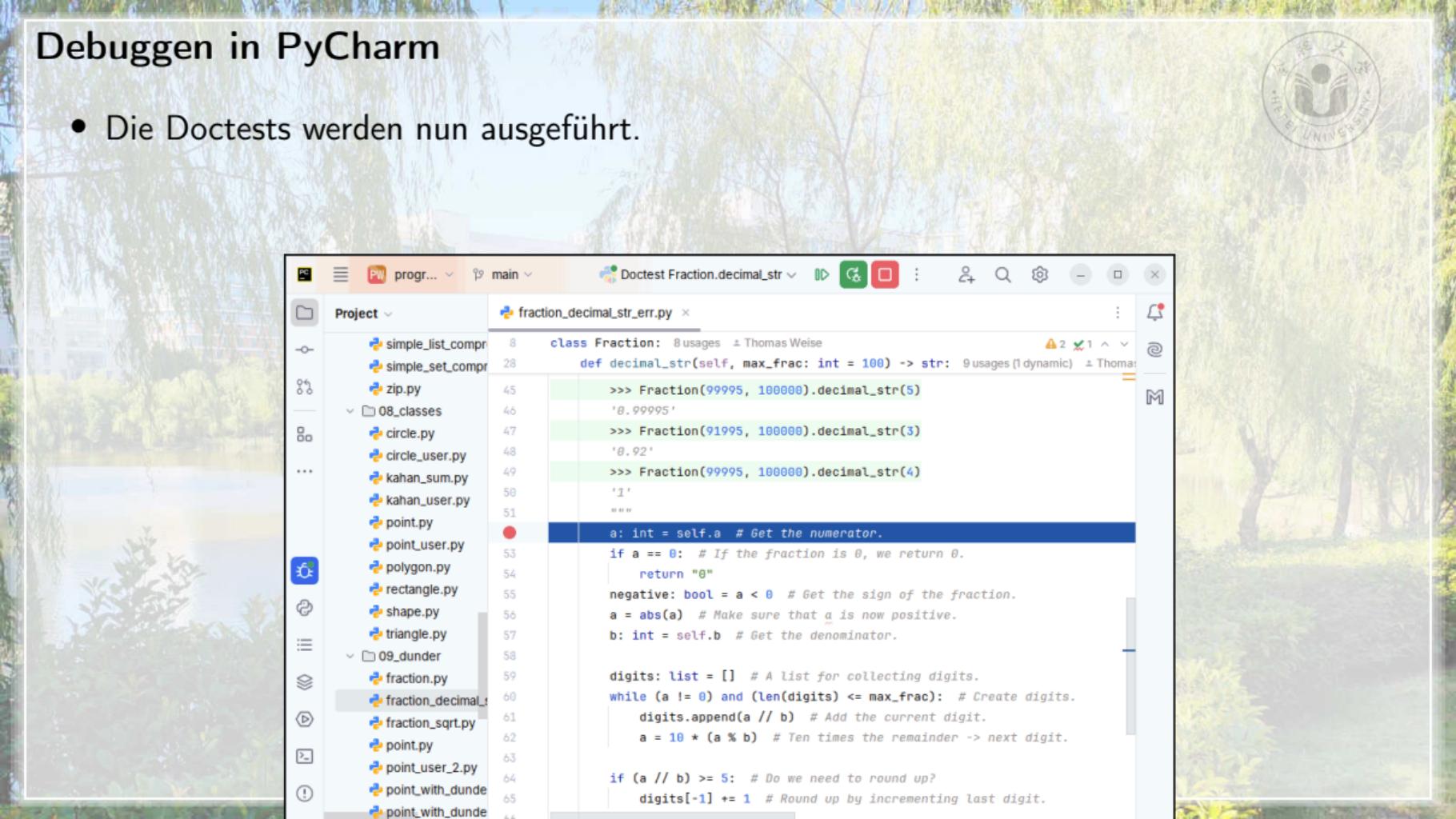
The code in the editor is as follows:

```
class Fraction: 8 usages ± Thomas Weise
    def decimal_str(self, max_frac: int = 100) -> str: 9 usages (1 dynamic) ± Thomas Weise
        :param max_frac: the maximum number of fractional digits
        :return: the string

    >>> Fraction(124, 2).decimal_str()
```

Debuggen in PyCharm

- Die Doctests werden nun ausgeführt.



The screenshot shows the PyCharm IDE interface. The left sidebar displays a project structure with several Python files under '08_classes' and '09_dunder' directories. The main editor window is titled 'Doctest Fraction.decimal_str' and contains the code for the `decimal_str` method of the `Fraction` class. A red dot at the bottom of the code editor indicates a break point. The code uses doctests to demonstrate the conversion of fractions to strings with specified decimal places. The code is as follows:

```
class Fraction:
    def decimal_str(self, max_frac: int = 100) -> str:
        >>> 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: bool = a < 0 # Get the sign of the fraction.
        a = abs(a) # Make sure that a is now positive.
        b: int = self.b # Get the denominator.

        digits: 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.
```

Debuggen in PyCharm



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

A screenshot of the PyCharm IDE interface. The top bar shows the file 'fraction_decimal_str_err.py' is open. The left sidebar shows a project structure with several files under '08_classes' and '09_dunder'. The main code editor window displays Python code for a 'Fraction' class. A red dot at line 52 indicates the current execution point. The code includes doctests and a detailed implementation of the decimal string conversion method.

```
class Fraction:
    def decimal_str(self, max_frac: int = 100) -> str:
        >>> 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: bool = a < 0 # Get the sign of the fraction.
        a = abs(a) # Make sure that a is now positive.
        b: int = self.b # Get the denominator.

        digits: 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.
```

Debuggen in PyCharm



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

A screenshot of the PyCharm IDE interface. The left sidebar shows a project structure with several Python files. The main editor window displays the code for 'fraction_decimal_str_err.py'. A red circle highlights a line of code: 'a: int = self.a # Get the numerator.' This line is underlined with a red dotted line, indicating it is a breakpoint. The code itself is a class Fraction with various methods for decimal conversion. The status bar at the bottom indicates the file is 'Doctest Fraction.decimal_str'.

Debuggen in PyCharm



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

The screenshot shows the PyCharm IDE interface. On the left is the Project tool window displaying a file tree with several Python files. In the center is the Editor tool window showing the code for `fraction_decimal_str_err.py`. A red dot at the bottom of the code editor indicates a breakpoint. The code itself contains several print statements and a complex logic for generating decimal strings from fractions. The line with the breakpoint is highlighted in blue.

```
class Fraction: 8 usages ± Thomas Weise
    def decimal_str(self, max_frac: int = 100) -> str: 9 usages (1 dynamic) ± Thomas Weise
        >>> 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.

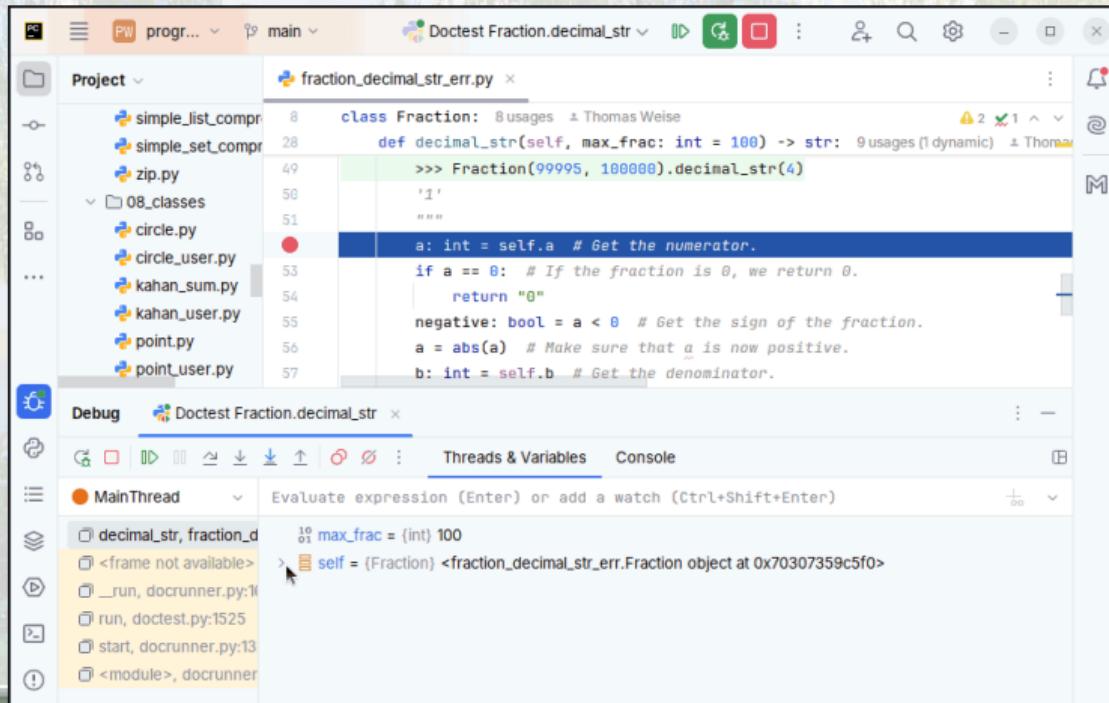
        if a == 0: # If the fraction is 0, we return 0.
            return "0"
        negative: bool = a < 0 # Get the sign of the fraction.
        a = abs(a) # Make sure that a is now positive.
        b: int = self.b # Get the denominator.

        digits: 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.
```

Debuggen in PyCharm

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



The screenshot shows the PyCharm IDE interface. The top part displays the code editor with a Python file named `fraction_decimal_str_err.py`. The code defines a `Fraction` class with a `decimal_str` method. A red dot in the gutter indicates a breakpoint at line 49. The bottom part shows the debugger toolbar with tabs for `Threads & Variables` and `Console`. The `Threads & Variables` tab is active, showing the `MainThread` and a list of frames. The frame for the `decimal_str` method is selected, and its local variables are listed in the sidebar:

- `max_frac = {int} 100`
- `self = {Fraction} <fraction_decimal_str_err.Fraction object at 0x70307359c5f0>`

Debuggen in PyCharm



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

The screenshot shows the PyCharm IDE interface. The top window displays the code for `fraction_decimal_str_err.py`. A red dot at line 57 indicates a breakpoint. The code defines a `Fraction` class with a `decimal_str` method. The bottom window is the **Debug** tool window, which includes tabs for **Threads & Variables** and **Console**. The **Threads & Variables** tab shows the **MainThread** and a list of frames. The **Console** tab shows the expression `self = {Fraction} <fraction_decimal_str_err.Fraction object at 0x70307359c5f0>`.

```
class Fraction:
    def decimal_str(self, max_frac: int = 100) -> str:
        >>> 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: bool = a < 0 # Get the sign of the fraction.
        a = abs(a) # Make sure that a is now positive.
        b: int = self.b # Get the denominator.
```

Threads & Variables Console

MainThread

decimal_str, fraction_d

<frame not available>

self = {Fraction} <fraction_decimal_str_err.Fraction object at 0x70307359c5f0>

Debuggen in PyCharm

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

The screenshot shows the PyCharm IDE interface. The top navigation bar includes tabs for 'PC', 'PWA', 'program...', 'main', and 'Doctest Fraction.decimal_str'. The main window has a 'Project' view on the left showing files like 'simple_list_compr.py', 'simple_set_compr.py', 'zip.py', and several files under '08_classes' such as 'circle.py', 'circle_user.py', 'kahan_sum.py', 'kahan_user.py', 'point.py', and 'point_user.py'. The central code editor displays the 'fraction_decimal_str_err.py' file with the following code:

```
class Fraction:
    def decimal_str(self, max_frac: int = 100) -> str:
        >>> 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: bool = a < 0 # Get the sign of the fraction.
        a = abs(a) # Make sure that a is now positive.
        b: int = self.b # Get the denominator.
```

The line 'a: int = self.a # Get the numerator.' is highlighted in blue and has a red circle at its start, indicating it is a breakpoint. Below the code editor, the 'Debug' tab is selected in the toolbar, and the 'Threads & Variables' tab is active in the bottom panel. The variables pane shows the current state of the MainThread, with the variable 'self' expanded to show its type as <fraction_decimal_str_err.Fraction object at 0x70307359c5f0>. Other visible frames include 'decimal_str, fraction_d' and '<frame not available>'.

Debuggen in PyCharm

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

The screenshot shows the PyCharm IDE interface. The main window displays a Python file named `fraction_decimal_str_err.py`. The code defines a `Fraction` class with a `decimal_str` method. A breakpoint is set at line 49, and the code execution has stopped there. The current line of code is highlighted in green: `>>> Fraction(99995, 100000).decimal_str(4)`. The next line to be executed is highlighted in blue: `a: int = self.a # Get the numerator.`. The status bar at the bottom indicates the current thread is `MainThread`.

The bottom part of the screen shows the `Threads & Variables` tab of the debugger. It lists the current stack frames:

- `decimal_str, fraction_d`
- `<frame not available>`
- `_run, docrunner.py:10`
- `run, doctest.py:1525`
- `start, docrunner.py:13`
- `<module>, docrunner`

The variable `self` is currently being evaluated, as indicated by the highlighted entry in the list.

```
class Fraction:
    def decimal_str(self, max_frac: int = 100) -> str:
        >>> 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: bool = a < 0 # Get the sign of the fraction.
        a = abs(a) # Make sure that a is now positive.
        b: int = self.b # Get the denominator.
```

Debuggen in PyCharm



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

The screenshot shows the PyCharm IDE interface with a debugger session active. The top navigation bar includes tabs for 'PC', 'Python', 'main', and 'Doctest Fraction.decimal_str'. The main code editor window displays a file named 'fraction_decimal_str_err.py' containing Python code for a 'Fraction' class. A red dot at line 49 indicates a breakpoint. The code defines a method 'decimal_str' that takes a fraction and formats it as a string. The bottom part of the interface shows the 'Threads & Variables' tab selected in the debugger tool window. It lists the 'MainThread' and shows variable information: 'max_frac = {int: 100}' and 'self = {Fraction} <fraction_decimal_str_err.Fraction object at 0x70307359c5f0>'. The 'Evaluate expression (Enter) or add a watch (Ctrl+Shift+Enter)' input field is also visible.

```
class Fraction:
    def decimal_str(self, max_frac: int = 100) -> str:
        >>> 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: bool = a < 0 # Get the sign of the fraction.
        a = abs(a) # Make sure that a is now positive.
        b: int = self.b # Get the denominator.
```

Threads & Variables

MainThread	Evaluate expression (Enter) or add a watch (Ctrl+Shift+Enter)
decimal_str, fraction_d	10 max_frac = {int: 100}
<frame not available>	> self = {Fraction} <fraction_decimal_str_err.Fraction object at 0x70307359c5f0>
_run, docrunner.py:1	
run, doctest.py:1525	
start, docrunner.py:13	
<module>, docrunner	

Debuggen in PyCharm



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

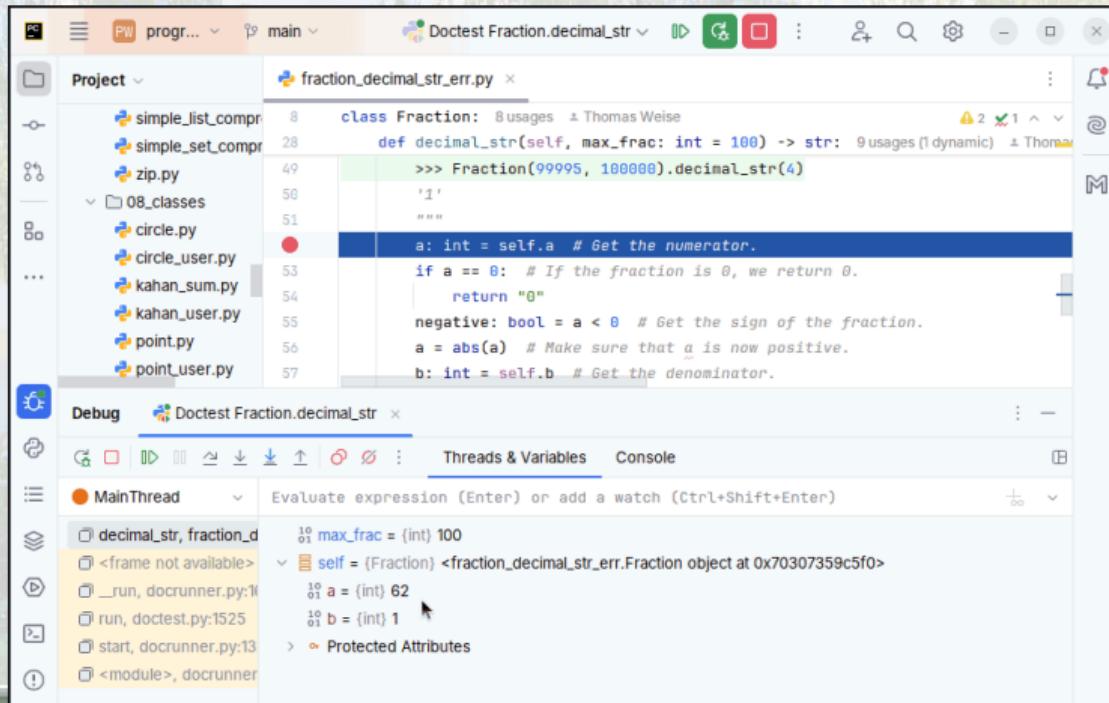
The screenshot shows the PyCharm IDE interface. The top window displays a Python file named `fraction_decimal_str_err.py`. A red dot on the left margin indicates a breakpoint at line 49. The code defines a `Fraction` class with a `decimal_str` method. The method takes a fraction and converts it to a string representation. It handles the case where the fraction is zero and extracts the sign and absolute values of the numerator and denominator. The bottom window is the **Threads & Variables** debugger, which is currently active. It shows the **MainThread** and lists local variables. The variable `self` is highlighted, showing its type as `<Fraction>` and its memory address as `0x70307359c5f0`. Other variables listed include `max_frac` (value 100) and the frame stack.

```
class Fraction: 8 usages ± Thomas Weise
    def decimal_str(self, max_frac: int = 100) -> str: 9 usages (1 dynamic) ± Thomas Weise
        >>> 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: bool = a < 0 # Get the sign of the fraction.
        a = abs(a) # Make sure that a is now positive.
        b: int = self.b # Get the denominator.
```

Threads & Variables
MainThread
Evaluate expression (Enter) or add a watch (Ctrl+Shift+Enter)
decimal_str, fraction_d
<frame not available>
_run, docrunner.py:10
run, doctest.py:1525
start, docrunner.py:13
<module>, docrunner

Debuggen in PyCharm

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



The screenshot shows the PyCharm IDE interface. The top navigation bar includes tabs for 'PC', 'Program', 'main', and 'Doctest Fraction.decimal_str'. The main window displays a code editor for 'fraction_decimal_str_err.py' with the following code:

```
class Fraction:
    def decimal_str(self, max_frac: int = 100) -> str:
        >>> 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: bool = a < 0 # Get the sign of the fraction.
        a = abs(a) # Make sure that a is now positive.
        b: int = self.b # Get the denominator.
```

The code editor has a red dot at line 49, indicating a breakpoint. Below the code editor is a 'Threads & Variables' tab in the 'Debug' tool window. The 'Variables' section shows the following state:

- decimal_str, fraction_d: max_frac = {int} 100
- <frame not available>
- __run, docrunner.py:10
- run, doctest.py:1525
- start, docrunner.py:13
- <module>, docrunner

The variable `a` is highlighted with a yellow background, showing its value as `{int} 62`.

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.

The screenshot shows the PyCharm IDE interface during a debugging session. The top navigation bar includes tabs for 'PC', 'PWI', 'Program', 'main', and 'Doctest Fraction.decimal_str'. The main window displays a Python file named 'fraction_decimal_str_err.py' with code for a 'Fraction' class. A breakpoint is set at line 49, where the code calls `Fraction(99995, 100000).decimal_str(4)`. The current line of execution is highlighted in green. The bottom panel shows the 'Threads & Variables' tab of the debugger, which lists the current thread as 'MainThread'. It shows a variable `self` of type 'Fraction' with attributes `a` (value 62) and `b` (value 1). Other frames listed include '`decimal_str`', '`fraction_decimal_str_err`', and '`__run`'.

```
class Fraction:
    def decimal_str(self, max_frac: int = 100) -> str:
        >>> 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: bool = a < 0 # Get the sign of the fraction.
        a = abs(a) # Make sure that a is now positive.
        b: int = self.b # Get the denominator.
```

Threads & Variables

Frame	Type	Value
<code>decimal_str</code> , <code>fraction_decimal_str_err</code>	Variable	<code>max_frac = 100</code>
<code><frame not available></code>	Variable	<code>self = {Fraction} <fraction_decimal_str_err.Fraction object at 0x70307359c5f0></code>
<code>__run</code> , <code>docrunner.py:10</code>	Variable	<code>a = {int} 62</code>
<code>run</code> , <code>doctest.py:1525</code>	Variable	<code>b = {int} 1</code>
<code>start</code> , <code>docrunner.py:13</code>	Variable	
<code><module></code> , <code>docrunner</code>	Variable	

Debuggen in PyCharm



- Das ist genau was wir erwarten.

The screenshot shows the PyCharm IDE interface with a debugger session active. The top navigation bar includes tabs for 'PC', 'Program', 'main', and 'Doctest Fraction.decimal_str'. The main window displays a Python file named 'fraction_decimal_str_err.py' containing code for a 'Fraction' class. A red dot at line 49 indicates a breakpoint. The code defines a method 'decimal_str' that takes a fraction and formats it as a string. The variable 'a' is highlighted in blue. The bottom panel shows the 'Threads & Variables' tab of the debugger, which lists the current thread 'MainThread' and variables in the stack frames. The variable 'a' is shown with a value of 62 in the current frame.

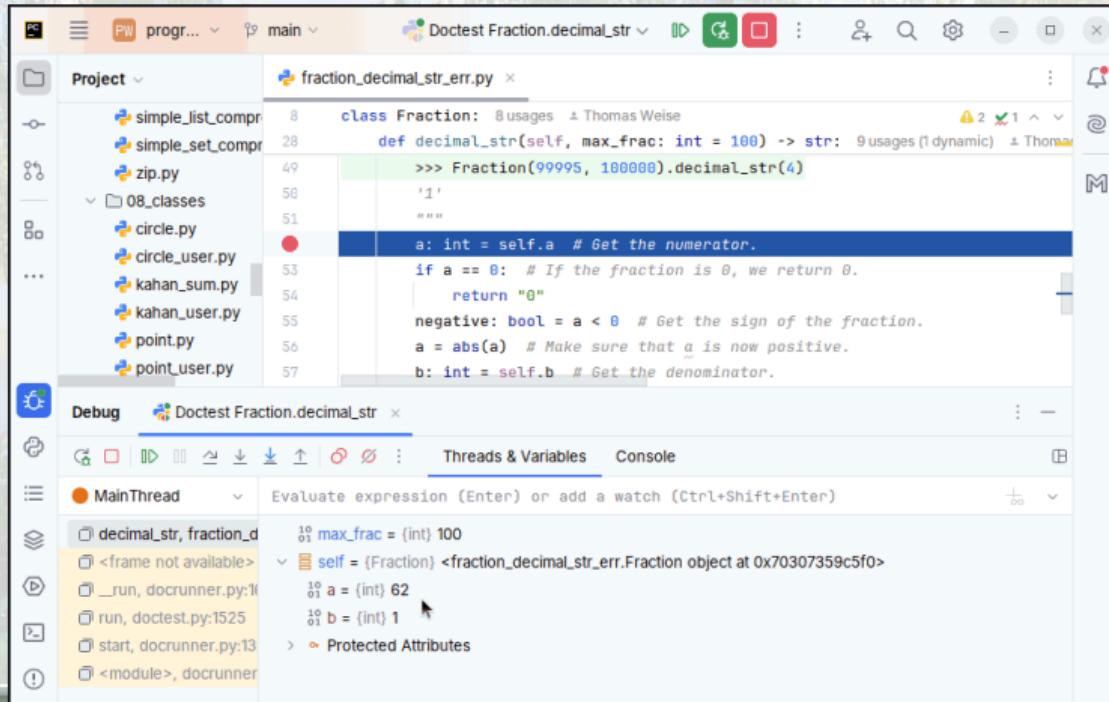
```
class Fraction:
    def decimal_str(self, max_frac: int = 100) -> str:
        >>> Fraction(99995, 100000).decimal_str(4)
        '1'
        """
        a: int = self.a # Get the numerator.
        if a == 0: # If the fraction is 0, we return 0.
            return "0"
        negative: bool = a < 0 # Get the sign of the fraction.
        a = abs(a) # Make sure that a is now positive.
        b: int = self.b # Get the denominator.
```

Threads & Variables

Frame	Variables
decimal_str, fraction_d	max_frac = {int} 100
<frame not available>	self = {Fraction} <fraction_decimal_str_err.Fraction object at 0x70307359c5f0>
_run, docrunner.py:10	a = {int} 62
run, doctest.py:1525	b = {int} 1
start, docrunner.py:13	
<module>, docrunner	

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 $\frac{62}{1}$.



The screenshot shows the PyCharm IDE interface with the following details:

- Project View:** Shows files like simple_list_compr, simple_set_compr, zip.py, and several files under the 08_classes directory (circle.py, circle_user.py, kahan_sum.py, kahan_user.py, point.py, point_user.py).
- Main Editor:** Displays the code for `fraction_decimal_str_err.py`. The cursor is on the line `a: int = self.a # Get the numerator.`. A red dot indicates a breakpoint on the previous line.
- Bottom Panel:** Contains tabs for "Debug" and "Doctest Fraction.decimal_str". The "Threads & Variables" tab is selected.
- Variables View:** Shows the current variable state:
 - decimal_str, fraction_d (checkbox)
 - <frame not available>
 - __run, docrunner.py:1025
 - run, doctest.py:1525
 - start, docrunner.py:13
 - <module>, docrunner

Under "decimal_str":
 - max_frac = {int} 100
 - self = {Fraction} <fraction_decimal_str_err.Fraction object at 0x70307359c5f0>
 - a = {int} 62
 - b = {int} 1

Debuggen in PyCharm

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

The screenshot shows the PyCharm IDE interface with a debugger session active. The top navigation bar includes tabs for 'PC', 'Program', 'main', and 'Doctest Fraction.decimal_str'. The main window displays a project structure with files like simple_list_compr.py, simple_set_compr.py, zip.py, and several files under the 08_classes directory. The code editor shows a class Fraction with a method decimal_str. A red dot at line 49 indicates a breakpoint. The code highlights the line: `a: int = self.a # Get the numerator.`. Below the editor, the 'Debug' tab is selected, and the 'Threads & Variables' tab is active. The variable pane shows a stack trace starting with 'Main Thread' and a local variable 'self' of type 'Fraction'. The bottom status bar shows the command 'Evaluate expression (Enter) or add a watch (Ctrl+Shift+Enter)'.

```
class Fraction:
    def decimal_str(self, max_frac: int = 100) -> str:
        >>> 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: bool = a < 0 # Get the sign of the fraction.
        a = abs(a) # Make sure that a is now positive.
        b: int = self.b # Get the denominator.
```

Threads & Variables Console

Main Thread Resume Program F9 Evaluate expression (Enter) or add a watch (Ctrl+Shift+Enter)

decimal_str, fraction_d

self = (Fraction) <fraction_decimal_str_err.Fraction object at 0x70307359c5f0>

a = (int) 62

b = (int) 1

Protected Attributes

Debuggen in PyCharm



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

The screenshot shows the PyCharm IDE interface. The top navigation bar includes tabs for 'Project', 'fractions_decimal_str.py', and 'Doctest Fraction.decimal_str'. Below the navigation bar is a code editor with a file named 'fraction_decimal_str_err.py'. A red dot at line 49 indicates a breakpoint. The code defines a class 'Fraction' with a method 'decimal_str'. The current line of code is 'a: int = self.a # Get the numerator.' The bottom part of the interface is the debugger tool window, which is currently active. It has tabs for 'Threads & Variables' and 'Console'. The 'Threads & Variables' tab is selected, showing a list of frames. The first frame is 'Main Thread' (highlighted in orange). The variable 'self' is expanded, showing its value as a 'Fraction' object with attributes 'a' (62) and 'b' (1). Other frames listed include '_run_docrunner.py:100', 'run_docrunner.py:1525', 'start_docrunner.py:13', and '<module>_docrunner'.

```
class Fraction:
    def decimal_str(self, max_frac: int = 100) -> str:
        >>> 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: bool = a < 0 # Get the sign of the fraction.
        a = abs(a) # Make sure that a is now positive.
        b: int = self.b # Get the denominator.
```

Threads & Variables Console

Main Thread Resume Program F9 Evaluate expression (Enter) or add a watch (Ctrl+Shift+Enter)

- decimal_str, fraction_d
- <frame not available>
- _run_docrunner.py:100
- run_docrunner.py:1525
- start_docrunner.py:13
- <module>_docrunner

max_frac = int(100)

self = (Fraction) <fraction_decimal_str_err.Fraction object at 0x70307359c5f0>

a = (int) 62

b = (int) 1

Protected Attributes

Debuggen in PyCharm

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

The screenshot shows the PyCharm IDE interface. The main editor window displays the code for the `Fraction` class in `fraction_decimal_str_err.py`. A red dot at line 57 indicates a breakpoint. The code handles the conversion of a fraction to a decimal string. The `Threads & Variables` tab in the Debug tool window is selected, showing the current thread (`Main Thread`) and variable values: `max_frac = 100`, `self = Fraction`, `a = 62`, and `b = 1`. The button in the Debug tool window is highlighted, indicating it can be clicked to resume execution.

```
class Fraction:
    def decimal_str(self, max_frac: int = 100) -> str:
        >>> 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: bool = a < 0 # Get the sign of the fraction.
        a = abs(a) # Make sure that a is now positive.
        b: int = self.b # Get the denominator.
```

Threads & Variables Console

Main Thread Resume Program F9 Evaluate expression (Enter) or add a watch (Ctrl+Shift+Enter)

decimal_str, fraction_d
<frame not available>
self = (Fraction) <fraction_decimal_str_err.Fraction object at 0x70307359c5f0>
a = (int) 62
b = (int) 1
> Protected Attributes

Debuggen in PyCharm



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

The screenshot shows the PyCharm IDE interface. The top navigation bar includes tabs for 'PC', 'File', 'Edit', 'Run', 'View', 'Tools', 'Help', and 'PyCharm'. The title bar shows 'Doctest Fraction.decimal_str' and the file path 'fraction_decimal_str_err.py'. The main area displays the code for the `Fraction` class, specifically the `decimal_str` method. A red dot at line 49 indicates the current execution point. The bottom panel is the 'Debug' tool window, which has tabs for 'Threads & Variables' and 'Console'. It shows a stack trace for the 'Main Thread' with the frame 'decimal_str, fraction_d'. The 'Resume Program' button (F9) is highlighted with a blue rectangle. The status bar at the bottom says 'Evaluate expression (Enter) or add a watch (Ctrl+Shift+Enter)'.

```
class Fraction:
    def decimal_str(self, max_frac: int = 100) -> str:
        >>> 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: bool = a < 0 # Get the sign of the fraction.
        a = abs(a) # Make sure that a is now positive.
        b: int = self.b # Get the denominator.
```

Threads & Variables Console

Main Thread Resume Program F9 Evaluate expression (Enter) or add a watch (Ctrl+Shift+Enter)

decimal_str, fraction_d

<frame not available>

self = (Fraction) <fraction_decimal_str_err.Fraction object at 0x70307359c5f0>

a = (int) 62

b = (int) 1

Protected Attributes

Debuggen in PyCharm

- Die Ausführung des Doctests wird fortgesetzt.



The screenshot shows the PyCharm IDE interface during a debugging session. The top navigation bar includes tabs for 'PC', 'Program', 'main', and 'Doctest Fraction.decimal_str'. The main window displays the code for 'fraction_decimal_str_err.py' with a red dot at line 49, indicating the current execution point. The code defines a class 'Fraction' with a method 'decimal_str' that takes a numerator and denominator and returns a string representation. The debugger toolbar at the bottom has buttons for 'Run', 'Stop', and 'Step Into'. The bottom panel shows the 'Threads & Variables' tab, which lists the current thread ('MainThread') and variables being monitored. The variable 'self' is shown as a 'Fraction' object, and its attributes 'a' and 'b' are both set to 1. A tooltip for 'a' indicates it is of type int.

```
class Fraction:
    def decimal_str(self, max_frac: int = 100) -> str:
        >>> 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: bool = a < 0 # Get the sign of the fraction.
        a = abs(a) # Make sure that a is now positive.
        b: int = self.b # Get the denominator.
```

Threads & Variables

- MainThread

Evaluate expression (Enter) or add a watch (Ctrl+Shift+Enter)

- decimal_str, fraction_d
- <frame not available>
- _run, docrunner.py:10
- run, doctest.py:1525
- start, docrunner.py:13
- <module>, docrunner

self = {Fraction} <fraction_decimal_str_err.Fraction object at 0x70307332e4e0>
a = {int} 1
b = {int} 2

Debuggen in PyCharm



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

The screenshot shows the PyCharm IDE interface. The top navigation bar includes tabs for 'PC', 'Python', 'main', and 'Doctest Fraction.decimal_str'. The main window displays a file named 'fraction_decimal_str_err.py' with the following code:

```
class Fraction:
    def decimal_str(self, max_frac: int = 100) -> str:
        >>> 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: bool = a < 0 # Get the sign of the fraction.
        a = abs(a) # Make sure that a is now positive.
        b: int = self.b # Get the denominator.
```

The line 'a: int = self.a # Get the numerator.' is highlighted in green and has a red circle at its start, indicating it is a breakpoint. The status bar at the bottom shows 'MainThread'.

The bottom panel contains a 'Threads & Variables' tab, which is currently selected. It shows the variable 'self' with the value '`<Fraction> <fraction_decimal_str_err.Fraction object at 0x70307332e4e0>`'. Below it, the variables 'a' and 'b' are listed with values '(int) 1' and '(int) 2' respectively. A tooltip for 'Protected Attributes' is visible.

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.

The screenshot shows the PyCharm IDE interface during a debugging session. The top navigation bar includes tabs for 'PC', 'Program', 'main', and 'Doctest Fraction.decimal_str'. The main window displays a file named 'fraction_decimal_str_err.py' with code for a `Fraction` class. A breakpoint is set on line 49, and the code is paused at that point. The code snippet is as follows:

```
    8     class Fraction: 8 usages ± Thomas Weise
    28         def decimal_str(self, max_frac: int = 100) -> str: 9 usages (1 dynamic) ± Thomas
    49             >>> Fraction(99995, 100000).decimal_str(4)
    50                 '1'
    51                 ...
    52             a: int = self.a # Get the numerator.
    53             if a == 0: # If the fraction is 0, we return 0.
    54                 return "0"
    55             negative: bool = a < 0 # Get the sign of the fraction.
    56             a = abs(a) # Make sure that a is now positive.
    57             b: int = self.b # Get the denominator.
```

The bottom part of the interface shows the 'Threads & Variables' tab of the debugger, which lists the current thread as 'MainThread' and shows variable values for `max_frac`, `self`, `a`, and `b`.

Variable	Type	Value
<code>max_frac</code>	<code>int</code>	100
<code>self</code>	<code>Fraction</code>	<fraction_decimal_str_err.Fraction object at 0x70307332e4e0>
<code>a</code>	<code>int</code>	1
<code>b</code>	<code>int</code>	2

Debuggen in PyCharm



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

The screenshot shows the PyCharm IDE interface. The top navigation bar includes tabs for 'PC', 'File', 'Edit', 'Run', 'View', 'Tools', 'Help', and 'PyCharm'. The title bar displays 'Doctest Fraction.decimal_str' and the file path 'main'. The left sidebar shows the project structure with files like 'simple_list_compr.py', 'simple_set_compr.py', 'zip.py', and a folder '08_classes' containing 'circle.py', 'circle_user.py', 'kahan_sum.py', 'kahan_user.py', 'point.py', and 'point_user.py'. The main editor window shows the code for the `decimal_str` method of the `Fraction` class. A red dot at line 49 indicates the current execution point. The code is as follows:

```
class Fraction:
    def decimal_str(self, max_frac: int = 100) -> str:
        >>> 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: bool = a < 0 # Get the sign of the fraction.
        a = abs(a) # Make sure that a is now positive.
        b: int = self.b # Get the denominator.
```

The bottom panel contains the 'Debug' tool window. It has tabs for 'Threads & Variables' and 'Console'. The 'Threads & Variables' tab is active, showing the 'MainThread' thread. A dropdown menu allows selecting a frame, with 'decimal_str, fraction_d' currently selected. The variable pane lists the local variables: `max_frac` (value 100), `self` (value a `Fraction` object), `a` (value 1), and `b` (value 2). There is also a section for 'Protected Attributes'.

Debuggen in PyCharm

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

The screenshot shows the PyCharm IDE interface. The top navigation bar includes tabs for 'PC', 'PWA', 'Program', 'main', and 'Doctest Fraction.decimal_str'. The main area is a code editor displaying 'fraction_decimal_str_err.py' with the following code:

```
class Fraction:
    def decimal_str(self, max_frac: int = 100) -> str:
        >>> 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: bool = a < 0 # Get the sign of the fraction.
        a = abs(a) # Make sure that a is now positive.
        b: int = self.b # Get the denominator.
```

The code editor has a red dot at line 49, indicating a breakpoint. Below the code editor is a 'Threads & Variables' tab in the debugger tool window. The variable tree shows:

- decimal_str, fraction_d (checkbox)
- <frame not available>
- _run, docrunner.py:102
- run, doctest.py:1525
- start, docrunner.py:13
- <module>, docrunner

The variable 'a' is highlighted with a yellow background, showing its value as 1. The variable 'b' is also highlighted with a yellow background, showing its value as 2.

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.

The screenshot shows the PyCharm IDE interface. The top navigation bar includes tabs for 'Program', 'main', and 'Doctest Fraction.decimal_str'. The main editor window displays the code for the `Fraction` class. A red dot at line 49 indicates the current line of execution. The code is as follows:

```
class Fraction:
    def decimal_str(self, max_frac: int = 100) -> str:
        >>> 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: bool = a < 0 # Get the sign of the fraction.
        a = abs(a) # Make sure that a is now positive.
        b: int = self.b # Get the denominator.
```

The bottom panel shows the 'Threads & Variables' tab of the debugger. It lists the current thread as 'MainThread' and shows variable values:

- `max_frac`: {int} 10
- `self`: {Fraction} <fraction_decimal_str_err.Fraction object at 0x7030732f32f0>
- `a`: {int} 1
- `b`: {int} 3

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.

The screenshot shows the PyCharm IDE interface. The code editor displays the file `fraction_decimal_str_err.py`. A red dot at line 49 indicates the current line of execution. The code defines a `Fraction` class with a `decimal_str` method. The debugger toolbar at the bottom has tabs for `Debug` and `Doctest Fraction.decimal_str`, with `Debug` selected. The `Threads & Variables` tab is active in the bottom panel, showing the stack trace and variable values for the current thread (MainThread). The variable `max_frac` is set to 10, and the fraction object `self` has numerator 1 and denominator 3.

```
class Fraction:
    def decimal_str(self, max_frac: int = 100) -> str:
        >>> 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: bool = a < 0 # Get the sign of the fraction.
        a = abs(a) # Make sure that a is now positive.
        b: int = self.b # Get the denominator.
```

Threads & Variables Console

MainThread

Evaluate expression (Enter) or add a watch (Ctrl+Shift+Enter)

decimal_str, fraction_d

max_frac = {int} 10

self = {Fraction} <fraction_decimal_str_err.Fraction object at 0x7030732f32f0>

a = {int} 1

b = {int} 3

Protected Attributes

Debuggen in PyCharm



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

The screenshot shows the PyCharm IDE interface. The top navigation bar includes tabs for 'PC', 'PWA', 'Program', 'main', and 'Doctest Fraction.decimal_str'. The main window displays a project structure on the left with files like 'simple_list_compr.py', 'simple_set_compr.py', 'zip.py', and several files under '08_classes' such as 'circle.py', 'circle_user.py', 'kahan_sum.py', 'kahan_user.py', 'point.py', and 'point_user.py'. The central code editor shows a portion of the 'Fraction' class definition:

```
class Fraction:
    def decimal_str(self, max_frac: int = 100) -> str:
        >>> 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: bool = a < 0 # Get the sign of the fraction.
        a = abs(a) # Make sure that a is now positive.
        b: int = self.b # Get the denominator.
```

The line 'a: int = self.a # Get the numerator.' is highlighted in blue and has a red dot at its start, indicating it is the current line of execution. Below the code editor, the 'Threads & Variables' tab is selected in the 'Debug' tool window. The variable tree shows the current state of the 'decimal_str' method:

- max_frac = {int} 10
- self = {Fraction} <fraction_decimal_str_err.Fraction object at 0x7030732f32f0>
- a = {int} 1
- b = {int} 3

The variable 'a' is currently expanded, showing its value as 1. The 'Threads & Variables' tab also contains a text input field: 'Evaluate expression (Enter) or add a watch (Ctrl+Shift+Enter)'.

Debuggen in PyCharm



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

The screenshot shows the PyCharm IDE interface. The top navigation bar includes tabs for 'PC', 'File', 'Edit', 'Run', 'View', 'Tools', 'Main', and 'Doctest Fraction.decimal_str'. The main window has a 'Project' view on the left showing files like 'simple_list_compr.py', 'simple_set_compr.py', 'zip.py', and several files under '08_classes' such as 'circle.py', 'circle_user.py', 'kahan_sum.py', 'kahan_user.py', 'point.py', and 'point_user.py'. The central code editor displays the 'fraction_decimal_str_err.py' file with the following code:

```
class Fraction:
    def decimal_str(self, max_frac: int = 100) -> str:
        >>> 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: bool = a < 0 # Get the sign of the fraction.
        a = abs(a) # Make sure that a is now positive.
        b: int = self.b # Get the denominator.
```

The line 'a: int = self.a # Get the numerator.' is highlighted in blue and has a red dot at its start, indicating it is the current line of execution. Below the code editor is a 'Debug' tool window with tabs for 'Threads & Variables' and 'Console'. The 'Threads & Variables' tab is active, showing the 'MainThread' thread. A list of variables is shown, with 'self' expanded to show its attributes 'a' and 'b':

- decimal_str, fraction_d
- <frame not available>
- _run, docrunner.py:10
- run, doctest.py:1525
- start, docrunner.py:13
- <module>, docrunner

Under 'self = (Fraction) <fraction_decimal_str_err.Fraction object at 0x7030732f32f0>', the attributes 'a' and 'b' are listed with their values: 'a = (int) 1' and 'b = (int) 3'.

Debuggen in PyCharm



- Als wir den Breakpoint wieder erreichen, sind wir im vierten Doctest Case angekommen:
-101001
100000000

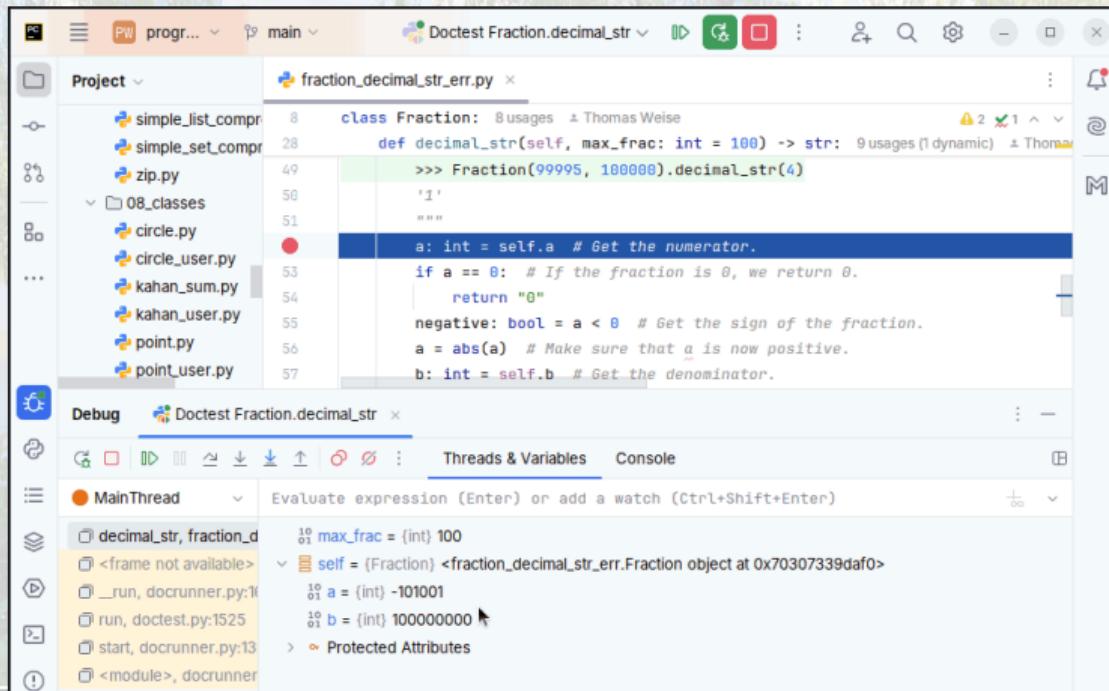
The screenshot shows the PyCharm IDE interface with the following details:

- Project View:** Shows files like simple_list_compr, simple_set_compr, zip.py, and several files under the 08_classes directory (circle.py, circle_user.py, kahan_sum.py, kahan_user.py, point.py, point_user.py). The file fraction_decimal_str_err.py is open.
- Code Editor:** Displays the Fraction class definition. A red dot at line 49 indicates a breakpoint. The current line of code is highlighted: `a: int = self.a # Get the numerator.`
- Toolbars:** Standard PyCharm toolbars for file operations, search, and navigation.
- Bottom Panel:** Shows the "Threads & Variables" tab selected in the "Debug" tool window. It lists the MainThread and shows variable values: `max_frac = {int} 100`, `self = {Fraction} <fraction_decimal_str_err.Fraction object at 0x70307339daf0>`, `a = {int} -101001`, and `b = {int} 100000000`.

Debuggen in PyCharm



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



The screenshot shows the PyCharm IDE interface during a debugging session. The top navigation bar includes tabs for 'PC', 'Run', 'Doctests', and 'Fraction.decimal_str'. The main window displays the file 'fraction_decimal_str_err.py' with code for a 'Fraction' class. A breakpoint is set at line 57, which is highlighted in red. The code snippet is as follows:

```
    8     class Fraction: 8 usages ± Thomas Weise
    28         def decimal_str(self, max_frac: int = 100) -> str: 9 usages (1 dynamic) ± Thomas Weise
    49             >>> Fraction(99995, 100000).decimal_str(4)
    50                 '-1'
    51                 """
    52                 a: int = self.a # Get the numerator.
    53                 if a == 0: # If the fraction is 0, we return 0.
    54                     return "0"
    55                 negative: bool = a < 0 # Get the sign of the fraction.
    56                 a = abs(a) # Make sure that a is now positive.
    57                 b: int = self.b # Get the denominator.
```

The bottom panel shows the 'Threads & Variables' tab of the debugger, with the 'MainThread' selected. It lists variables and their values:

- max_frac = {int} 100
- self = {Fraction} <fraction_decimal_str_err.Fraction object at 0x70307339daf0>
- a = {int} -101001
- b = {int} 100000000

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.

The screenshot shows the PyCharm IDE interface during a debugging session. The top window displays the code for `fraction_decimal_str_err.py`, specifically the `decimal_str` method of the `Fraction` class. A red dot at line 49 indicates the current execution point. The bottom window shows the debugger tool window with the "Threads & Variables" tab selected. It lists the current thread as "MainThread". In the variable list, the variable `self` is expanded, showing its attributes `max_frac`, `a`, and `b`. The value of `a` is shown as `(int) 247` and the value of `b` as `(int) 200`.

```
class Fraction:
    def decimal_str(self, max_frac: int = 100) -> str:
        >>> 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: bool = a < 0 # Get the sign of the fraction.
        a = abs(a) # Make sure that a is now positive.
        b: int = self.b # Get the denominator.
```

MainThread

Evaluate expression (Enter) or add a watch (Ctrl+Shift+Enter)

- decimal_str, fraction_d
- <frame not available>
- __run, docrunner.py:10
- run, doctest.py:1525
- start, docrunner.py:13
- <module>, docrunner

max_frac = {int} 2
self = {Fraction} <fraction_decimal_str_err.Fraction object at 0x70307395fb60>
a = {int} 247
b = {int} 200

Debuggen in PyCharm



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

The screenshot shows the PyCharm IDE interface. The top navigation bar includes tabs for 'PC', 'Program', 'main', and 'Doctest Fraction.decimal_str'. The main area displays the code for the `fraction_decimal_str_err.py` file, specifically the `Fraction` class. A red dot at line 49 indicates a breakpoint. The code snippet is as follows:

```
    8     class Fraction: 8 usages ± Thomas Weise
    28         def decimal_str(self, max_frac: int = 100) -> str: 9 usages (1 dynamic) ± Thomas Weise
    49             >>> Fraction(99995, 100000).decimal_str(4)
    50                 '1'
    51                 """
    52             a: int = self.a # Get the numerator.
    53             if a == 0: # If the fraction is 0, we return 0.
    54                 return "0"
    55             negative: bool = a < 0 # Get the sign of the fraction.
    56             a = abs(a) # Make sure that a is now positive.
    57             b: int = self.b # Get the denominator.
```

The bottom part of the interface shows the 'Threads & Variables' tab of the debugger. It lists the current thread as 'MainThread' and shows variable values for `max_frac` (2), `self` (a `Fraction` object), `a` (247), and `b` (200).

Debuggen in PyCharm



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

The screenshot shows the PyCharm IDE interface. The top navigation bar includes tabs for 'PC', 'PWI', 'program...', 'main', and 'Doctest Fraction.decimal_str'. The main area displays the code for the `fraction_decimal_str_err.py` file. A red dot at line 49 indicates the current line of execution. The code defines a `Fraction` class with a `decimal_str` method. The method takes a numerator and denominator, handles zero cases, and returns a string representation of the fraction. The bottom part of the screenshot shows the 'Threads & Variables' tab of the debugger, which lists the current thread (MainThread) and variables in the stack frames. It shows the variable `self` with its value as a `Fraction` object, and the variables `a` and `b` both set to `19999`.

```
class Fraction:
    def decimal_str(self, max_frac: int = 100) -> str:
        >>> 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: bool = a < 0 # Get the sign of the fraction.
        a = abs(a) # Make sure that a is now positive.
        b: int = self.b # Get the denominator.
```

Threads & Variables

Frame	Value
decimal_str, fraction_decimal_str_err.py	max_frac = 5
<frame not available>	self = Fraction object at 0x703074378110
_run, docrunner.py:10	a = 19999
run, doctest.py:1525	b = 20000
start, docrunner.py:13	
<module>, docrunner	

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.

A screenshot of the PyCharm IDE interface. The top navigation bar shows 'Doctest Fraction.decimal_str' and various tool icons. The main area is divided into two panes: the left pane shows a project structure with files like simple_list_compr, simple_set_compr, zip.py, and several .py files under '08_classes'. The right pane displays the code for 'fraction_decimal_str_err.py'. A red dot at line 57 indicates a breakpoint. The code defines a 'Fraction' class with a method 'decimal_str'. The current line of code is 'a: int = self.a # Get the numerator.' The bottom part of the interface shows the 'Threads & Variables' tab of the debugger, which lists the 'MainThread' and shows variable values: 'max_frac' (10), 'self' (a Fraction object), 'a' (19999), and 'b' (20000).

```
class Fraction:
    def decimal_str(self, max_frac: int = 100) -> str:
        >>> 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: bool = a < 0 # Get the sign of the fraction.
        a = abs(a) # Make sure that a is now positive.
        b: int = self.b # Get the denominator.
```

Threads & Variables Console

MainThread

Evaluate expression (Enter) or add a watch (Ctrl+Shift+Enter)

decimal_str, fraction_d

<frame not available>

_run, docrunner.py:10

run, doctest.py:1525

start, docrunner.py:13

<module>, docrunner

max_frac = 10

self = Fraction object at 0x703074378110

a = 19999

b = 20000

Protected Attributes

Debuggen in PyCharm



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

The screenshot shows the PyCharm IDE interface with the following details:

- Project View:** Shows files like `simple_list_compr.py`, `simple_set_compr.py`, `zip.py`, and a `08_classes` directory containing `circle.py`, `circle_user.py`, `kahan_sum.py`, `kahan_user.py`, `point.py`, and `point_user.py`.
- Editor:** Displays the code for `fraction_decimal_str_err.py`. A red dot at line 49 indicates the current line of execution. The code defines a `Fraction` class with a `decimal_str` method that handles the conversion of fractions to strings.
- Toolbars:** Standard PyCharm toolbars for file operations, search, and navigation.
- Bottom Panel:** Shows the `Threads & Variables` tab of the debugger, which is currently active. It lists the `MainThread` and shows variable values:
 - `max_frac = {int} 3`
 - `self = {Fraction} <fraction_decimal_str_err.Fraction object at 0x7d37f0a0ed50>`
 - `a = {int} 18399`
 - `b = {int} 20000`

Debuggen in PyCharm

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



The screenshot shows the PyCharm IDE interface during a debugging session. The top window displays the code for `fraction_decimal_str_err.py`. A red dot at line 49 indicates the current execution point. The code defines a `Fraction` class with an `__init__` method that takes a numerator and denominator, and a `decimal_str` method that returns a string representation of the fraction. The bottom window shows the debugger tool window with the "Threads & Variables" tab selected. It lists the current thread as "MainThread" and shows variable values: `max_frac` is set to 3, and `self` is a `Fraction` object with attributes `a` (18399) and `b` (20000). Other frames listed include `__run__`, `doctrunner.py`, and `start`.

```
class Fraction:
    def __init__(self, a, b):
        self.a = a
        self.b = b
    def decimal_str(self, max_frac: int = 100) -> str:
        if self.a == 0:
            return "0"
        negative: bool = self.a < 0
        self.a = abs(self.a)
        a: int = self.a # Get the numerator.
        if a == 0:
            return "0"
        negative: bool = a < 0 # Get the sign of the fraction.
        a = abs(a) # Make sure that a is now positive.
        b: int = self.b # Get the denominator.
        # Implementation of decimal conversion follows...
```

Threads & Variables

Variable	Type	Value
max_frac	int	3
self	<code>Fraction</code>	<code><fraction_decimal_str_err.Fraction object at 0x7d37f0a0ed50></code>
a	int	18399
b	int	20000

Debuggen in PyCharm



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

The screenshot shows the PyCharm IDE interface during a debugging session. The top window displays the code for `fraction_decimal_str_err.py`. A red dot at line 49 indicates the current execution point. The code defines a `Fraction` class with a `decimal_str` method. The method takes a fraction and a `max_frac` parameter. It handles the case where the fraction is zero and extracts the numerator and denominator. The bottom window is the `Threads & Variables` tool window, which shows the current thread is `MainThread`. In the variables pane, the variable `max_frac` is shown with a value of 3. Other variables like `a` (18399) and `b` (20000) are also listed.

```
class Fraction:
    def decimal_str(self, max_frac: int = 100) -> str:
        >>> 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: bool = a < 0 # Get the sign of the fraction.
        a = abs(a) # Make sure that a is now positive.
        b: int = self.b # Get the denominator.
```

Threads & Variables

- MainThread

Evaluate expression (Enter) or add a watch (Ctrl+Shift+Enter)

- max_frac = {int} 3
- self = {Fraction} <fraction_decimal_str_err.Fraction object at 0x7d37f0a0ed50>
- a = {int} 18399
- b = {int} 20000

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.

The screenshot shows the PyCharm IDE interface. The top window displays the code for `fraction_decimal_str_err.py`. A red dot at line 57 indicates the current execution point. The code defines a `Fraction` class with a `decimal_str` method. The method takes a fraction and a `max_frac` parameter. It handles the case where the fraction is zero and extracts the numerator and denominator. The bottom window is the `Threads & Variables` tool window, which is currently active. It shows the `MainThread` and lists variables from the current frame. The variable `a` is highlighted with a yellow background, showing its value as `18399`.

```
class Fraction:
    def decimal_str(self, max_frac: int = 100) -> str:
        >>> 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: bool = a < 0 # Get the sign of the fraction.
        a = abs(a) # Make sure that a is now positive.
        b: int = self.b # Get the denominator.

        Evaluate expression (Enter) or add a watch (Ctrl+Shift+Enter)
        decimal_str, fraction_d
        <frame not available>
        __run, docrunner.py:1025
        run, doctest.py:1525
        start, docrunner.py:13
        <module>, docrunner
        max_frac = {int} 3
        self = {Fraction} <fraction_decimal_str_err.Fraction object at 0x7d37f0a0ed50>
        a = {int} 18399
        b = {int} 20000
        Protected Attributes
```

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.

The screenshot shows the PyCharm IDE interface. The top navigation bar displays "Doctest Fraction.decimal_str" and the file "fraction_decimal_str_err.py". The left sidebar shows a project structure with files like simple_list_compr, simple_set_compr, zip.py, and several files under the 08_classes directory (circle.py, circle_user.py, kahan_sum.py, kahan_user.py, point.py, point_user.py). The main code editor window shows the following Python code:

```
class Fraction:
    def decimal_str(self, max_frac: int = 100) -> str:
        >>> 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: bool = a < 0 # Get the sign of the fraction.
        a = abs(a) # Make sure that a is now positive.
        b: int = self.b # Get the denominator.
```

The line `a: int = self.a # Get the numerator.` is highlighted in green and has a red circular breakpoint icon to its left. Below the code editor, the "Threads & Variables" tab is selected in the "Debug" tool window. The variable pane shows the current state of the MainThread:

variable	value
max_frac	{int} 3
self	{Fraction} <fraction_decimal_str_err.Fraction object at 0x7d37f0a0ed50>
a	{int} 18399
b	{int} 20000

Debuggen in PyCharm



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

The screenshot shows the PyCharm IDE interface with the following details:

- Project View:** Shows files like simple_list_compr, simple_set_compr, zip.py, and several files under the 08_classes directory (circle.py, circle_user.py, kahan_sum.py, kahan_user.py, point.py, point_user.py).
- Main Editor:** Displays the code for Fraction.decimal_str. The current line, "a: int = self.a # Get the numerator.", is highlighted in blue and has a red circular breakpoint icon to its left. The line above it, "def decimal_str(self, max_frac: int = 100) -> str:", is also highlighted.
- Bottom Bar:** Shows tabs for "Debug" and "Doctest Fraction.decimal_str".
- Toolbars:** Includes standard PyCharm icons for file operations, search, and settings.
- Bottom Panel:** Contains tabs for "Threads & Variables" and "Console".
- Variables View:** Shows the current state of variables:
 - decimal_str, fraction_d (checkboxed)
 - <frame not available>
 - self = {Fraction} <fraction_decimal_str_err.Fraction object at 0x7d37f0a0ed50>
 - a = {int} 18399
 - b = {int} 20000

Debuggen in PyCharm



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

The screenshot shows the PyCharm IDE interface with the following details:

- Project View:** Shows the project structure with files like simple_list_compr.py, simple_set_compr.py, zip.py, and several files under the 08_classes directory (circle.py, circle_user.py, kahan_sum.py, kahan_user.py, point.py, point_user.py).
- Code Editor:** Displays the code for Fraction.decimal_str(). A red dot at line 49 indicates the current execution point. The code is as follows:

```
    class Fraction:
        def decimal_str(self, max_frac: int = 100) -> str:
            >>> 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: bool = a < 0 # Get the sign of the fraction.
            a = abs(a) # Make sure that a is now positive.
            b: int = self.b # Get the denominator.
```

- Toolbars:** Includes standard PyCharm toolbars for file operations, search, and navigation.
- Bottom Panel:** Contains tabs for "Threads & Variables" and "Console".
- Breakpoint Bar:** Shows a red dot at line 49, indicating the current breakpoint.
- Call Stack:** Shows the call stack with the current frame being "decimal_str, fraction_d".
- Registers:** Shows registers with values: max_frac = 3, self = <Fraction>, a = 18399, b = 20000.

Debuggen in PyCharm

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



The screenshot shows the PyCharm IDE interface during a debugging session. The top window displays the code for `fraction_decimal_str_err.py`. A red dot at line 53 indicates the current execution point. The bottom window, titled "Threads & Variables", shows the variable `a` being evaluated. The expression `decimal_str(fraction.DecimalStr)` is entered in the "Evaluate expression" field, resulting in the value `18399`.

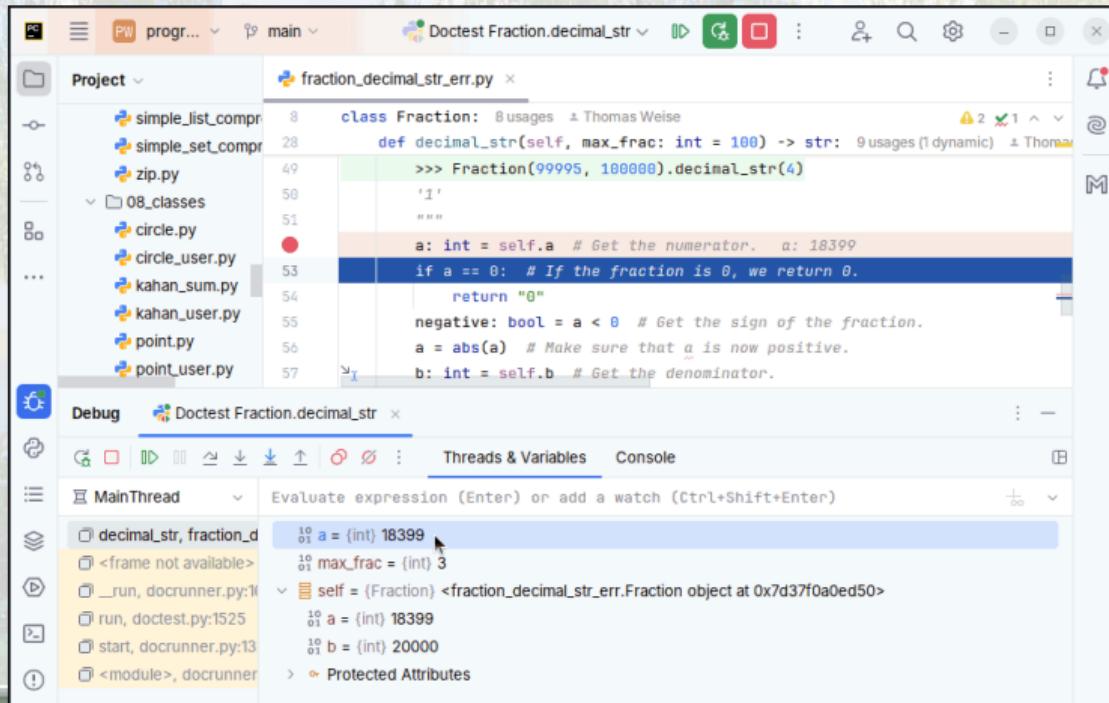
```
class Fraction:
    def decimal_str(self, max_frac: int = 100) -> str:
        >>> Fraction(99995, 100000).decimal_str(4)
        '1'
        ...
        a: int = self.a # Get the numerator. a: 18399
        if a == 0: # If the fraction is 0, we return 0.
            return "0"
        negative: bool = a < 0 # Get the sign of the fraction.
        a = abs(a) # Make sure that a is now positive.
        b: int = self.b # Get the denominator.
```

Threads & Variables

Frame	Value
decimal_str, fraction.DecimalStr	18399
<frame not available>	3
_run, docrunner.py:1	
run, docrunner.py:1525	
start, docrunner.py:13	
<module>, docrunner	

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.



The screenshot shows the PyCharm IDE interface. The top part displays the code for `fraction_decimal_str_err.py`. A red dot marks the current line of execution, which contains the assignment `a: int = self.a # Get the numerator. a: 18399`. Below the code editor, the toolbar includes icons for Run, Stop, Step Over, Step Into, Step Out, and Break. The bottom part shows the **Threads & Variables** tab of the debugger. It lists the current thread as `MainThread` and provides an input field to evaluate expressions. The variable `a` is listed with the value `18399`. Other variables shown include `max_frac` (value 3), `self` (a `Fraction` object), and `b` (value 20000). The **Protected Attributes** section is also visible.

```
class Fraction: 8 usages ± Thomas Weise
    def decimal_str(self, max_frac: int = 100) -> str: 9 usages (1 dynamic) ± Thomas Weise
        >>> Fraction(99995, 100000).decimal_str(4)
        '1'
        ...
        a: int = self.a # Get the numerator. a: 18399
        if a == 0: # If the fraction is 0, we return 0.
            return "0"
        negative: bool = a < 0 # Get the sign of the fraction.
        a = abs(a) # Make sure that a is now positive.
        b: int = self.b # Get the denominator.
```

Threads & Variables

Frame	Value
decimal_str, fraction_d	18399
<frame not available>	
_run, docrunner.py:1	
run, doctest.py:1525	
start, docrunner.py:13	
<module>, docrunner	

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.

The screenshot shows the PyCharm IDE interface. The top window displays the code for `fraction_decimal_str_err.py`. A red dot at line 53 indicates the current execution point. The code defines a `Fraction` class with a `decimal_str` method. The method takes a numerator and denominator, handles zero cases, and returns a string representation of the fraction. The bottom window, titled "Debug", shows the "Threads & Variables" tab. It lists variables and their values: `a` is 18399, `max_frac` is 3, and `self` is a `Fraction` object. The variable `a` is highlighted in blue, indicating it is being evaluated or has been modified during the debug session.

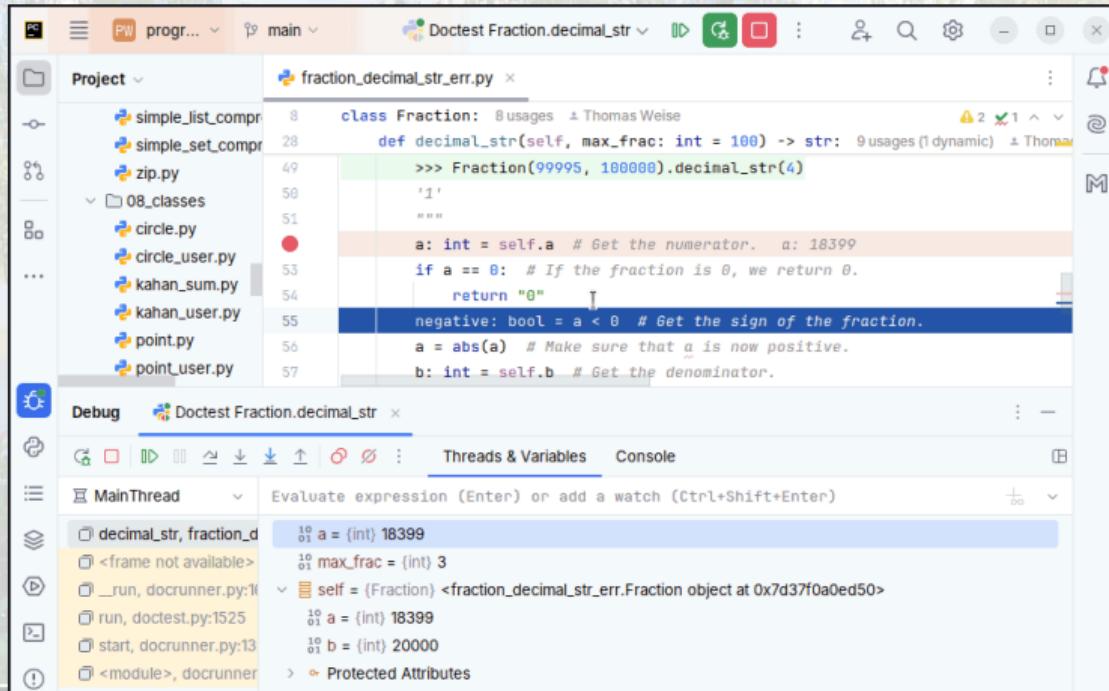
```
class Fraction:
    def decimal_str(self, max_frac: int = 100) -> str:
        >>> Fraction(99995, 100000).decimal_str(4)
        '1'
        ...
        a: int = self.a # Get the numerator. a: 18399
        if a == 0: # If the fraction is 0, we return 0.
            return "0"
        negative: bool = a < 0 # Get the sign of the fraction.
        a = abs(a) # Make sure that a is now positive.
        b: int = self.b # Get the denominator.
```

Threads & Variables

Variable	Type	Value
<code>a</code>	int	18399
<code>max_frac</code>	int	3
<code>self</code>	<code>Fraction</code>	<fraction_decimal_str_err.Fraction object at 0x7d37f0a0ed50>
<code>b</code>	int	20000

Debuggen in PyCharm

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



The screenshot shows the PyCharm IDE interface with the following details:

- Project View:** Shows files like `simple_list_compr.py`, `simple_set_compr.py`, `zip.py`, and several files under the `08_classes` directory including `circle.py`, `circle_user.py`, `kahan_sum.py`, `kahan_user.py`, `point.py`, and `point_user.py`.
- Code Editor:** Displays the `fraction_decimal_str_err.py` file. The cursor is at line 55, where the condition `a == 0` is highlighted in red, indicating it is being executed.
- Debug Tab:** Shows the current state of variables:
 - `a = 18399` (int)
 - `max_frac = 3` (int)
 - `self = Fraction` (Fraction object)
 - `b = 20000` (int)
- Threads & Variables:** Shows the current thread is `MainThread`.
- Console:** Placeholder for evaluating expressions or adding watches.

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.

The screenshot shows the PyCharm IDE interface. The top navigation bar includes tabs for 'PC', 'File', 'Edit', 'Run', 'View', 'Tools', 'Help', and 'PyCharm'. The title bar displays 'Doctest Fraction.decimal_str' and the file path 'main'. The main area is divided into two panes: 'Project' on the left and 'Code Editor' on the right. The 'Project' pane shows a file structure with files like 'simple_list_compr.py', 'simple_set_compr.py', 'zip.py', and several files under the '08_classes' directory. The 'Code Editor' pane shows the code for the 'Fraction' class, specifically the implementation of the `decimal_str` method. A red dot at line 51 indicates the current execution point. The code highlights the condition `a == 0` in red, indicating it is being evaluated. The bottom part of the interface features a 'Debug' tool window with tabs for 'Threads & Variables' and 'Console'. The 'Threads & Variables' tab is active, showing a list of variables and their values. The variable `a` is listed with the value `18399`. The 'Console' tab is also visible. The status bar at the bottom shows the message 'Evaluate expression (Enter) or add a watch (Ctrl+Shift+Enter)'.

```
class Fraction:
    def decimal_str(self, max_frac: int = 100) -> str:
        >>> Fraction(99995, 100000).decimal_str(4)
        '1'
        ...
        a: int = self.a # Get the numerator. a: 18399
        if a == 0: # If the fraction is 0, we return 0.
            return "0"
        negative: bool = a < 0 # Get the sign of the fraction.
        a = abs(a) # Make sure that a is now positive.
        b: int = self.b # Get the denominator.
```

Threads & Variables

Variable	Type	Value
a	int	18399
max_frac	int	3
self	Fraction	<fraction_decimal_str_err.Fraction object at 0x7d37f0a0ed50>
a	int	18399
b	int	20000

Debuggen in PyCharm



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

The screenshot shows the PyCharm IDE interface. The top window displays the code for `fraction_decimal_str_err.py`. A red dot at line 51 indicates the current execution point. The code defines a `Fraction` class with a `decimal_str` method. The method handles the case where the numerator is zero by returning "0". It also handles negative fractions by setting `negative` to `bool(a < 0)` and making `a` positive by calling `abs(a)`. The denominator is obtained from `self.b`.

The bottom window shows the debugger's state. The stack trace lists frames: `decimal_str`, `<frame not available>`, `_run`, `docrunner.py`, `run`, `doctest.py`, `start`, `docrunner.py`, and `<module>`. The variable `a` is currently set to `18399`. The variable `max_frac` is set to `3`. The variable `self` is a `Fraction` object with `a = 18399` and `b = 20000`.

```
class Fraction:
    def decimal_str(self, max_frac: int = 100) -> str:
        >>> Fraction(99995, 100000).decimal_str(4)
        '1'
        ...
        a: int = self.a # Get the numerator. a: 18399
        if a == 0: # If the fraction is 0, we return 0.
            return "0"
        negative: bool = a < 0 # Get the sign of the fraction.
        a = abs(a) # Make sure that a is now positive.
        b: int = self.b # Get the denominator.
```

```
18 a = {int} 18399
18 max_frac = {int} 3
self = {Fraction} <fraction_decimal_str_err.Fraction object at 0x7d37f0a0ed50>
19 a = {int} 18399
19 b = {int} 20000
> Protected Attributes
```

Debuggen in PyCharm



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

The screenshot shows the PyCharm IDE interface during a debugging session. The top navigation bar includes tabs for 'PC', 'Python', 'Program', 'main', and 'Doctest Fraction.decimal_str'. The main window displays a project structure with files like simple_list_compr, simple_set_compr, zip.py, and several files under the 08_classes directory (circle.py, circle_user.py, kahan_sum.py, kahan_user.py, point.py, point_user.py). The code editor shows a portion of the `decimal_str` method from the `Fraction` class:

```
    class Fraction:
        def decimal_str(self, max_frac: int = 100) -> str:
            >>> Fraction(99995, 100000).decimal_str(4)
            '1'
            ...
            a: int = self.a # Get the numerator. a: 18399
            if a == 0: # If the fraction is 0, we return 0.
                return "0"
            negative: bool = a < 0 # Get the sign of the fraction.
            a = abs(a) # Make sure that a is now positive.
            b: int = self.b # Get the denominator.
```

The line `a: int = self.a # Get the numerator. a: 18399` is highlighted in green, indicating it is the next line to be executed. The line `negative: bool = a < 0 # Get the sign of the fraction.` is highlighted in blue, indicating it is the current line being debugged. The bottom part of the interface shows the 'Threads & Variables' tab of the debugger, with a list of variables and their values:

Variable	Value
<code>a</code>	{int} 18399
<code>max_frac</code>	{int} 3
<code>self</code>	{Fraction} <fraction_decimal_str_err.Fraction object at 0x7d37f0a0ed50>
<code>a</code>	{int} 18399
<code>b</code>	{int} 20000

Debuggen in PyCharm



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

The screenshot shows the PyCharm IDE interface. The top navigation bar includes tabs for 'PC', 'Python', 'File', 'Edit', 'Run', 'View', 'Tools', 'Help', and 'PyCharm'. The title bar displays 'Doctest Fraction.decimal_str' and the file path 'main'. The left sidebar shows a project structure with files like 'simple_list_compr.py', 'simple_set_compr.py', 'zip.py', and several files under '08_classes' such as 'circle.py', 'circle_user.py', 'kahan_sum.py', 'kahan_user.py', 'point.py', and 'point_user.py'. The main code editor window shows a portion of the 'fraction_decimal_str_err.py' file:

```
class Fraction: 8 usages ± Thomas Weise
    def decimal_str(self, max_frac: int = 100) -> str: 9 usages (1 dynamic) ± Thomas Weise
        >>> Fraction(99995, 100000).decimal_str(4)
        '1'
        ...
        a: int = self.a # Get the numerator. a: 18399
        if a == 0: # If the fraction is 0, we return 0.
            ...
            return "0"
        negative: bool = a < 0 # Get the sign of the fraction.
        a = abs(a) # Make sure that a is now positive.
        b: int = self.b # Get the denominator.
```

The line 'negative: bool = a < 0' is highlighted in blue, indicating it is the next line to be executed. The bottom part of the interface shows the 'Debug' tool window with tabs for 'Threads & Variables' and 'Console'. The 'Threads & Variables' tab is active, displaying the current stack trace and variable values:

- MainThread
- Evaluate expression (Enter) or add a watch (Ctrl+Shift+Enter)
- decimal_str, fraction_d
- <frame not available>
- _run, docrunner.py:1
- run, doctest.py:1525
- start, docrunner.py:13
- <module>, docrunner

Under 'Variables', the value of 'a' is shown as `18399`. The 'Console' tab is also visible at the bottom.

Debuggen in PyCharm

- Die lokale Variable `negative` wird erzeugt.



The screenshot shows the PyCharm IDE interface. The top window displays the code for `fraction_decimal_str_err.py`. A red dot at line 55 indicates a breakpoint. The code defines a `Fraction` class with a `decimal_str` method. The variable `negative` is highlighted in blue. The bottom window is the debugger, titled "Debug Doctest Fraction.decimal_str". It shows the stack trace and the current frame's variables. The variable `negative` is listed as `negative = {bool} False`.

```
class Fraction:
    def decimal_str(self, max_frac: int = 100) -> str:
        >>> Fraction(99995, 100000).decimal_str(4)
        '1'
        ...
        a: int = self.a # Get the numerator. a: 18399
        if a == 0: # If the fraction is 0, we return 0.
            return "0"
        negative: bool = a < 0 # Get the sign of the fraction. negative: False
        a = abs(a) # Make sure that a is now positive.
        b: int = self.b # Get the denominator.
```

Threads & Variables Console

MainThread Evaluate expression (Enter) or add a watch (Ctrl+Shift+Enter)

- decimal_str, fraction_d
- <frame not available>
- __run, docrunner.py:10
- run, doctest.py:1525
- start, docrunner.py:13
- <module>, docrunner

negative = {bool} False

self = {Fraction} <fraction_decimal_str_err.Fraction object at 0x7d37f0a0ed50>

a = {int} 18399

b = {int} 20000

Protected Attributes

Debuggen in PyCharm



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

A screenshot of the PyCharm IDE interface. The top navigation bar shows 'PC' and 'program...' with a dropdown arrow, followed by 'main'. The title bar says 'Doctest Fraction.decimal_str'. The main area shows a code editor with Python code for a 'Fraction' class. A red dot at line 51 indicates a breakpoint. The code highlights the line 'negative: bool = a < 0'. The bottom part of the interface shows the 'Threads & Variables' tab of the debugger, which lists variables and their values. The variable 'negative' is highlighted in blue and has a tooltip 'negative: bool = a < 0'. Other variables listed include 'a', 'max_frac', and 'self'.

```
class Fraction:
    def decimal_str(self, max_frac: int = 100) -> str:
        >>> Fraction(99995, 100000).decimal_str(4)
        '1'
        """
        a: int = self.a # Get the numerator. a: 18399
        if a == 0: # If the fraction is 0, we return 0.
            return "0"
        negative: bool = a < 0 # Get the sign of the fraction. negative: False
        a = abs(a) # Make sure that a is now positive.
        b: int = self.b # Get the denominator.
```

Threads & Variables

Variable	Type	Value
a	int	18399
max_frac	int	3
negative	bool	False
self	Fraction	<fraction_decimal_str_err.Fraction object at 0x7d37f0a0ed50>
a	int	18399
b	int	20000

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.

The screenshot shows the PyCharm IDE interface. The top navigation bar includes tabs for 'PC', 'PWA', 'Program', 'main', and 'Doctest Fraction.decimal_str'. The main window displays a Python file named 'fraction_decimal_str_err.py' with the following code:

```
class Fraction:
    def decimal_str(self, max_frac: int = 100) -> str:
        >>> Fraction(99995, 100000).decimal_str(4)
        '1'
        """
        a: int = self.a # Get the numerator. a: 18399
        if a == 0: # If the fraction is 0, we return 0.
            return "0"
        negative: bool = a < 0 # Get the sign of the fraction. negative: False
        a = abs(a) # Make sure that a is now positive.
        b: int = self.b # Get the denominator.
        """

        # The current line being executed is highlighted in green: a = abs(a)
```

The bottom part of the interface shows the 'Threads & Variables' tool window. It lists variables and their values:

- `a`: `{int} 18399`
- `max_frac`: `{int} 3`
- `negative`: `{bool} False` (highlighted in blue)
- `self`: `{Fraction} <fraction_decimal_str_err.Fraction object at 0x7d37f0a0ed50>`
- `a`: `{int} 18399`
- `b`: `{int} 20000`

Debuggen in PyCharm

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

The screenshot shows the PyCharm IDE interface. The top part is the code editor with the file `fraction_decimal_str_err.py` open. The code defines a `Fraction` class with a `decimal_str` method. A breakpoint is set at the line `a = abs(a)`. The bottom part is the debugger tool window, titled "Debug Doctest Fraction.decimal_str". It shows the stack trace and the current frame's variables:

```
MainThread
Evaluate expression (Enter) or add a watch (Ctrl+Shift+Enter)
decimal_str, fraction_d
<frame not available>
__run, docrunner.py:1025
run, doctest.py:1525
start, docrunner.py:13
<module>, docrunner

10 a = (int) 18399
10 max_frac = (int) 3
10 negative = (bool) False
self = {Fraction} <fraction_decimal_str_err.Fraction object at 0x7d37f0a0ed50>
10 a = (int) 18399
10 b = (int) 20000
Protected Attributes
```

Debuggen in PyCharm



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

The screenshot shows the PyCharm IDE interface. The top navigation bar includes tabs for 'PC', 'PWA', 'program...', 'main', and 'Doctest Fraction.decimal_str'. The main window displays a project structure with files like 'simple_list_compr.py', 'simple_set_compr.py', 'zip.py', and several files under '08_classes' such as 'circle.py', 'circle_user.py', 'kahan_sum.py', 'kahan_user.py', 'point.py', and 'point_user.py'. The current file is 'fraction_decimal_str_err.py', which contains the following code:

```
class Fraction:
    def decimal_str(self, max_frac: int = 100) -> str:
        negative: bool = a < 0 # Get the sign of the fraction. negative: False
        a = abs(a) # Make sure that a is now positive.
        b: int = self.b # Get the denominator.

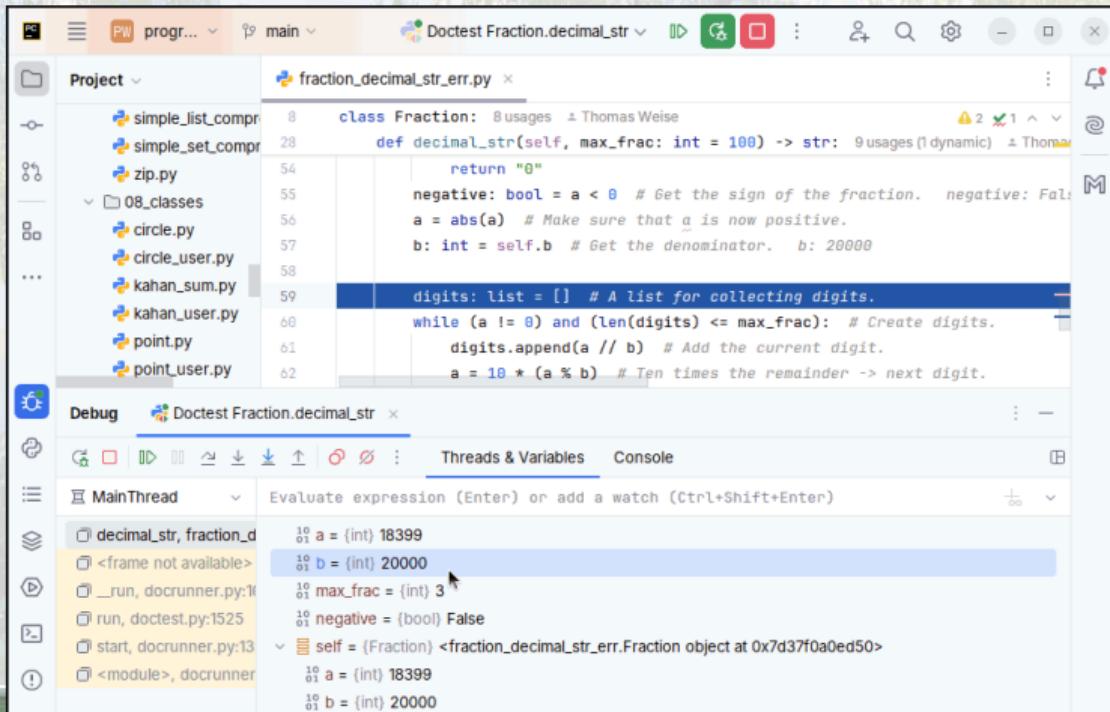
        digits: list = [] # A list for collecting digits.
        while (a != 0) and (len(digits) <= max_frac): # Create digits.
            digits.append(a // b) # Add the current digit.
            a = 10 * (a % b) # Ten times the remainder -> next digit.
```

The bottom part of the interface shows the 'Threads & Variables' tab of the debugger. It lists variables and their values:

- `a` = `{int} 18399`
- `max_frac` = `{int} 3`
- `negative` = `{bool} False`
- `self` = `{Fraction} <fraction_decimal_str_err.Fraction object at 0x7d37f0a0ed50>`
- `a` = `{int} 18399`
- `b` = `{int} 20000`

Debuggen in PyCharm

- Das führt `b = self.b` aus.



The screenshot shows the PyCharm IDE interface. The top bar displays the project name "Doctest Fraction.decimal_str" and various icons. The left sidebar shows the project structure with files like simple_list_compr, simple_set_compr, zip.py, and several files under the 08_classes directory. The main code editor window shows a Python class Fraction with a method decimal_str. A specific line of code is highlighted: "b: int = self.b # Get the denominator. b: 20000". Below the code editor is a "Threads & Variables" tab in the debugger tool window. The variable pane shows the current values of variables: a = 18399, b = 20000, max_frac = 3, negative = False, and self = Fraction object at 0x7d37f0a0ed50. The variable b is selected, indicated by a blue selection bar and a cursor hovering over it.

```
class Fraction:
    def decimal_str(self, max_frac: int = 100) -> str:
        return "0"
        negative: bool = a < 0 # Get the sign of the fraction. negative: False
        a = abs(a) # Make sure that a is now positive.
        b: int = self.b # Get the denominator. b: 20000
        digits: 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.
```

Threads & Variables

```
a = {int} 18399
b = {int} 20000
max_frac = {int} 3
negative = {bool} False
self = {Fraction} <fraction_decimal_str_err.Fraction object at 0x7d37f0a0ed50>
```

Debuggen in PyCharm



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

The screenshot shows the PyCharm IDE interface. The top part is the code editor with the file `fraction_decimal_str_err.py` open. The code defines a `Fraction` class with a `decimal_str` method. The method calculates the decimal representation of a fraction by repeatedly dividing the numerator by the denominator and collecting digits. A bug is highlighted at line 59 where the code `a = 10 * (a % b)` is shown, indicating a multiplication by 10 instead of 100. The bottom part of the interface is the debugger tool window, titled "Debug Doctest Fraction.decimal_str". It shows the current stack trace with the frame `decimal_str, fraction_d` selected. In the "Threads & Variables" tab, a variable `b` is listed with the value `20000`, which corresponds to the bug in the code.

```
class Fraction:
    def decimal_str(self, max_frac: int = 100) -> str:
        negative: bool = a < 0 # Get the sign of the fraction. negative: False
        a = abs(a) # Make sure that a is now positive.
        b: int = self.b # Get the denominator. b: 20000
        digits: 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.
```

Threads & Variables

```
a = {int} 18399
b = {int} 20000
max_frac = {int} 3
negative = {bool} False
self = {Fraction} <fraction_decimal_str_err.Fraction object at 0x7d37f0a0ed50>
a = {int} 18399
b = {int} 20000
```

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`.

The screenshot shows the PyCharm IDE interface. The top navigation bar includes tabs for 'PC', 'File', 'Edit', 'Run', 'View', 'Tools', 'Help', and 'PyCharm'. The title bar displays 'Doctest Fraction.decimal_str' and the file path 'main'. The main area is divided into two panes: 'Project' on the left and 'Editor' on the right. The 'Project' pane shows a file structure with files like 'simple_list_compr.py', 'simple_set_compr.py', 'zip.py', and several files under '08_classes' such as 'circle.py', 'circle_user.py', 'kahan_sum.py', 'kahan_user.py', 'point.py', and 'point_user.py'. The 'Editor' pane shows the code for the 'Fraction' class, specifically the 'decimal_str' method. The cursor is positioned on the line 'digits: list = [] # A list for collecting digits.'. Below the editor, the 'Threads & Variables' tab is selected in the 'Debug' tool window. The variable 'b' is listed with its value as '20000'. Other variables shown include 'a' (18399), 'max_frac' (3), 'negative' (False), and 'self' (a Fraction object). The bottom status bar indicates the current thread is 'MainThread'.

```
class Fraction:
    def decimal_str(self, max_frac: int = 100) -> str:
        negative: bool = a < 0 # Get the sign of the fraction. negative: False
        a = abs(a) # Make sure that a is now positive.
        b: int = self.b # Get the denominator. b: 20000
        digits: 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.
```

Threads & Variables

Variable	Type	Value
a	int	18399
b	int	20000
max_frac	int	3
negative	bool	False
self	Fraction	<fraction_decimal_str_err.Fraction object at 0x7d37f0a0ed50>

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**.

The screenshot shows the PyCharm IDE interface. The code editor displays a Python file named `fraction_decimal_str_err.py`. The cursor is positioned on the line `digits: list = [] # A list for collecting digits.`. The code editor toolbar includes icons for file operations, search, and settings. Below the code editor is the **Threads & Variables** tab of the debugger, which lists variables and their values. The variable `a` is shown as `{int} 18399` and `b` as `{int} 20000`. The background of the slide features a photograph of a lake and trees.

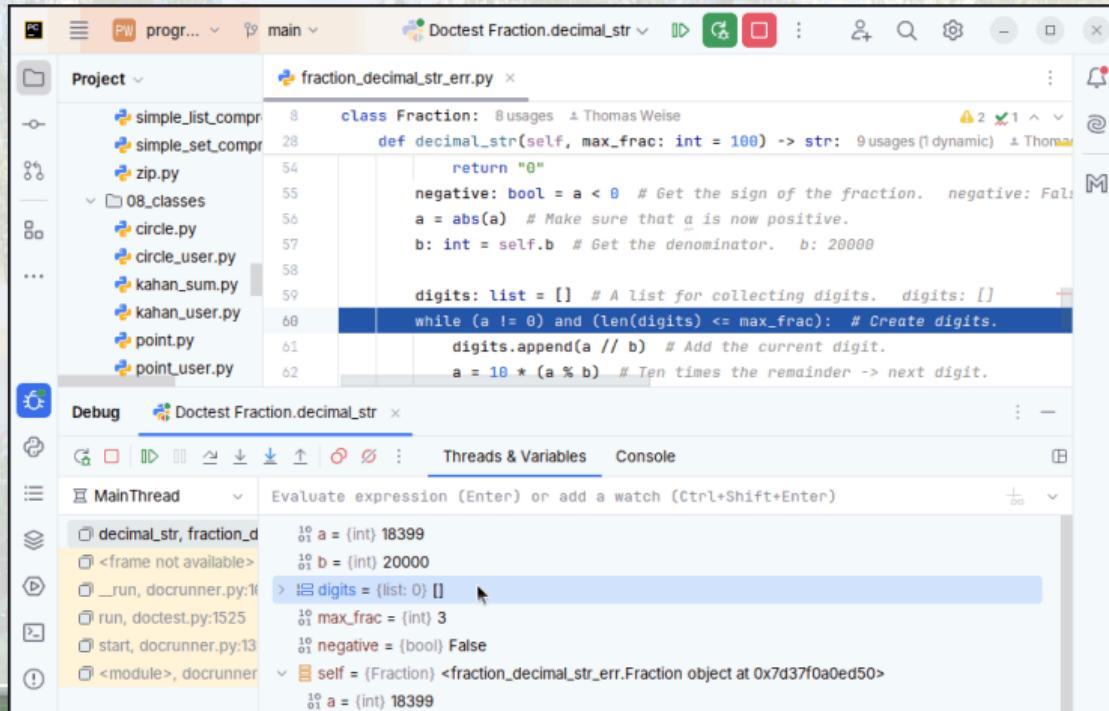
```
class Fraction:
    def decimal_str(self, max_frac: int = 100) -> str:
        negative: bool = a < 0 # Get the sign of the fraction. negative: False
        a = abs(a) # Make sure that a is now positive.
        b: int = self.b # Get the denominator. b: 20000
        digits: list = [] # A list for 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.
```

Threads & Variables

Variable	Type	Value
<code>a</code>	<code>int</code>	<code>18399</code>
<code>b</code>	<code>int</code>	<code>20000</code>
<code>max_frac</code>	<code>int</code>	<code>3</code>
<code>negative</code>	<code>bool</code>	<code>False</code>
<code>self</code>	<code>Fraction</code>	<code><fraction_decimal_str_err.Fraction object at 0x7d37f0a0ed50></code>
<code>a</code>	<code>int</code>	<code>18399</code>
<code>b</code>	<code>int</code>	<code>20000</code>

Debuggen in PyCharm

- Die neue Variable `digits` ist wirklich aufgetaucht.



The screenshot shows the PyCharm IDE interface. The top window is the code editor displaying `fraction_decimal_str_err.py`. The code defines a `Fraction` class with a `decimal_str` method. A breakpoint is set at line 60, where the variable `digits` is being assigned. The bottom window is the debugger, showing the `MainThread` and the expression `digits = [list: 0] []` in the Evaluate expression field. The background of the slide features a circular logo of the University of Regensburg.

```
class Fraction:
    def decimal_str(self, max_frac: int = 100) -> str:
        negative: bool = a < 0 # Get the sign of the fraction. negative: False
        a = abs(a) # Make sure that a is now positive.
        b: int = self.b # Get the denominator. b: 20000
        digits: list = [] # A list for collecting digits. 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.
```

Threads & Variables Console

```
a = {int} 18399
b = {int} 20000
digits = [list: 0] []
max_frac = {int} 3
negative = {bool} False
self = {Fraction} <fraction_decimal_str_err.Fraction object at 0x7d37f0a0ed50>
a = {int} 18399
```

Debuggen in PyCharm



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

The screenshot shows the PyCharm IDE interface. The top window displays the code for `fraction_decimal_str_err.py`. A specific line of code is highlighted:

```
    digits: list = [] # A list for collecting digits. 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.
```

The bottom window is the "Threads & Variables" tab of the debugger. It shows the variable `digits` with its value set to `[list: 0] []`. Other variables shown include `a`, `b`, `max_frac`, `negative`, and `self`.

Debuggen in PyCharm



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

The screenshot shows the PyCharm IDE interface. The top navigation bar includes tabs for 'PC', 'PW', 'program...', 'main', and 'Doctest Fraction.decimal_str'. The main window displays a Python file named 'fraction_decimal_str_err.py' with code for a 'Fraction' class. A specific line of code is highlighted in blue: `while (a != 0) and (len(digits) <= max_frac): # Create digits.`. Below the code editor, the 'Threads & Variables' tab is selected in the 'Debug' tool window. The variable pane shows the current state of variables:

- `a = {int} 18399`
- `b = {int} 20000`
- `digits = {list: 0} []` (highlighted in blue)
- `max_frac = {int} 3`
- `negative = {bool} False`
- `self = {Fraction} <fraction_decimal_str_err.Fraction object at 0x7d37f0a0ed50>`
- `a = {int} 18399`

Debuggen in PyCharm



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

The screenshot shows the PyCharm IDE interface. The top navigation bar includes tabs for 'PC', 'PW', 'program...', 'main', and 'Doctest Fraction.decimal_str'. The main window displays a Python file named 'fraction_decimal_str_err.py' with code for a 'Fraction' class. The cursor is positioned on the line: `while (a != 0) and (len(digits) <= max_frac): # Create digits.`. Below the code editor, the 'Threads & Variables' tab is selected in the 'Debug' tool window. The variable pane shows the current state of variables: `a = 18399`, `b = 20000`, and `digits = [list: 0] []`. The bottom status bar shows the line number 10.

```
class Fraction:
    def decimal_str(self, max_frac: int = 100) -> str:
        negative: bool = a < 0 # Get the sign of the fraction. negative: False
        a = abs(a) # Make sure that a is now positive.
        b: int = self.b # Get the denominator. b: 20000
        digits: list = [] # A list for collecting digits. 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.
```

Threads & Variables

MainThread

Evaluate expression (Enter) or add a watch (Ctrl+Shift+Enter)

decimal_str, fraction_d
<frame not available>
__run, docrunner.py:1025
run, doctest.py:1525
start, docrunner.py:13
<module>, docrunner

10 a = {int} 18399
10 b = {int} 20000
> 10 digits = [list: 0] []
10 max_frac = {int} 3
10 negative = {bool} False
self = {Fraction} <fraction_decimal_str_err.Fraction object at 0x7d37f0a0ed50>

10 a = {int} 18399

Debuggen in PyCharm



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

The screenshot shows the PyCharm IDE interface. The top navigation bar includes tabs for 'PC', 'Program', 'File', and 'main'. The title bar says 'Doctest Fraction.decimal_str'.

The left sidebar shows a project structure with files like 'simple_list_compr.py', 'simple_set_compr.py', 'zip.py', and several files under '08_classes' such as 'circle.py', 'circle_user.py', 'kahan_sum.py', 'kahan_user.py', 'point.py', and 'point_user.py'. The file 'Fraction_decimal_str_err.py' is open in the main editor area.

The code in 'Fraction_decimal_str_err.py' is as follows:

```
class Fraction:
    def decimal_str(self, max_frac: int = 100) -> str:
        if a < 0:
            negative: bool = a < 0
            a = abs(a)
        b: int = self.b
        digits: list = []
        while (a != 0) and (len(digits) <= max_frac):
            digits.append(a // b)
            a = 10 * (a % b)
```

The line 'digits.append(a // b)' is highlighted with a blue selection bar, indicating it is the current line being debugged.

The bottom panel contains a 'Threads & Variables' tab and a 'Console' tab. The 'Threads & Variables' tab is active, showing the MainThread. A watch expression 'digits = [list: 0]' is entered in the input field. Other variables listed include 'a', 'b', 'max_frac', 'negative', and 'self'. The variable 'a' has a value of 18399.

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.

A screenshot of the PyCharm IDE interface. The top navigation bar shows 'PC' and 'program...' with a dropdown for 'main'. The title bar says 'Doctest Fraction.decimal_str'. The main area is a code editor with the file 'fraction_decimal_str_err.py' open. The code defines a class 'Fraction' with a method 'decimal_str'. A line of code is highlighted: 'while (a != 0) and (len(digits) <= max_frac):'. Below the code editor is a toolbar with icons for file operations and a 'Debug' button. The bottom part of the interface is the debugger panel, titled 'Debug Doctest Fraction.decimal_str'. It shows the 'Threads & Variables' tab is selected. In the variables list, there are entries for 'a' (value 18399), 'b' (value 20000), and 'digits' (value [list: 0]). The variable 'digits' is currently selected. Other entries include 'max_frac' (value 3), 'negative' (value False), and 'self' (value Fraction object).

```
class Fraction:
    def decimal_str(self, max_frac: int = 100) -> str:
        return "0"
        negative: bool = a < 0 # Get the sign of the fraction. negative: False
        a = abs(a) # Make sure that a is now positive.
        b: int = self.b # Get the denominator. b: 20000
        digits: list = [] # A list for collecting digits. 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.
```

Threads & Variables Console

MainThread

Evaluate expression (Enter) or add a watch (Ctrl+Shift+Enter)

- decimal_str, fraction_d
- <frame not available>
- _run, docrunner.py:1
- run, doctest.py:1525
- start, docrunner.py:13
- <module>, docrunner

10 a = {int} 18399
10 b = {int} 20000
> 10 digits = {list: 0} []
10 max_frac = {int} 3
10 negative = {bool} False
10 self = {Fraction} <fraction_decimal_str_err.Fraction object at 0x7d37f0a0ed50>
10 a = {int} 18399

Debuggen in PyCharm



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

The screenshot shows the PyCharm IDE interface with the following details:

- Project View:** Shows the project structure with files like `simple_list_compr.py`, `simple_set_compr.py`, `zip.py`, and several files under the `08_classes` directory including `circle.py`, `circle_user.py`, `kahan_sum.py`, `kahan_user.py`, `point.py`, and `point_user.py`.
- Main Editor:** Displays the code for `fraction_decimal_str_err.py`. The current line of code is highlighted: `digits.append(a // b)`.
- Toolbars:** Standard PyCharm toolbars for file operations, search, and navigation.
- Bottom Panel:** Shows the `Threads & Variables` tab selected, displaying the current state of variables:
 - `a = {int} 18399`
 - `b = {int} 20000`
 - `digits = {list: 0} []`
 - `max_frac = {int} 3`
 - `negative = {bool} False`
 - `self = {Fraction} <fraction_decimal_str_err.Fraction object at 0x7d37f0a0ed50>`
 - `a = {int} 18399`

Debuggen in PyCharm



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

The screenshot shows the PyCharm IDE interface. The top navigation bar includes tabs for 'PC', 'PWA', 'Program', 'main', and 'Doctest Fraction.decimal_str'. The main window displays a Python file named 'fraction_decimal_str_err.py' with code for a 'Fraction' class. A specific line of code is highlighted: '10 digits.append(a // b)' with a comment '# Add the current digit.' Below the code editor, the 'Threads & Variables' tab is selected in the 'Debug' tool window. The variable tree shows the current state of variables: 'a = {int} 18399' and 'b = {int} 20000'. A tooltip for 'digits' is visible, showing its definition as '10 digits = [list: 0] []'. Other variables listed include 'max_frac', 'negative', 'self', and 'start'. The bottom status bar shows the memory address of the 'self' variable: '0x7d37f0a0ed50'.

```
class Fraction:
    def decimal_str(self, max_frac: int = 100) -> str:
        return "0"
        negative: bool = a < 0 # Get the sign of the fraction. negative: False
        a = abs(a) # Make sure that a is now positive.
        b: int = self.b # Get the denominator. b: 20000
        digits: list = [] # A list for collecting digits. 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.
```

Threads & Variables

```
a = {int} 18399
b = {int} 20000
digits = [list: 0] []
max_frac = {int} 3
negative = {bool} False
self = {Fraction} <fraction_decimal_str_err.Fraction object at 0x7d37f0a0ed50>
```

Debuggen in PyCharm



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

The screenshot shows the PyCharm IDE interface. The top navigation bar includes tabs for 'PC', 'Program', 'main', and 'Doctest Fraction.decimal_str'. The main window displays a Python file named 'fraction_decimal_str_err.py' with code for a 'Fraction' class. A specific line of code is highlighted: `digits.append(a // b)`. The code block continues with comments about rounding logic and handling integer parts. Below the code editor, the 'Threads & Variables' tab of the 'Debug' tool window is selected. It shows a list of variables and their values, with the variable `digits` currently selected. The value of `digits` is shown as `[list: 1] [0]`, which is highlighted with a blue selection bar.

```
class Fraction:
    def decimal_str(self, max_frac: int = 100) -> str:
        digits: list = [] # A list for collecting digits. digits: [0]
        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?
```

Threads & Variables

MainThread	Evaluate expression (Enter) or add a watch (Ctrl+Shift+Enter)
decimal_str, fraction_d	18 a = {int} 18399
<frame not available>	18 b = {int} 20000
_run, docrunner.py:1	
run, docrunner.py:1525	
start, docrunner.py:13	
<module>, docrunner	

igits = [list: 1] [0]

```
max_frac = {int} 3
negative = {bool} False
self = {Fraction} <fraction_decimal_str_err.Fraction object at 0x7d37f0a0ed50>
a = {int} 18399
```

Debuggen in PyCharm



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

The screenshot shows the PyCharm IDE interface. The top navigation bar includes tabs for 'Project', 'File', 'Edit', 'Run', 'View', 'Tools', 'Help', and icons for 'Run', 'Stop', 'Break', and 'Run Configuration'. The title bar displays 'Doctest Fraction.decimal_str' and the file path 'main'. The main area shows a code editor with Python code for a 'Fraction' class. A specific line of code is highlighted: 'a = 10 * (a % b) # Ten times the remainder -> next digit.' Below the code editor is a 'Threads & Variables' tab in the bottom navigation bar. The bottom panel shows the 'Threads & Variables' tool window with a list of frames and their variable values. The variable 'digits' is shown as '[list: 1] [0]'.

```
class Fraction:
    def decimal_str(self, max_frac: int = 100) -> str:
        digits: list = [] # A list for collecting digits.
        while (a != 0) and (len(digits) <= max_frac):
            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?
```

Threads & Variables

Frame	Value
decimal_str, fraction_d	a = {int} 18399 b = {int} 20000
<frame not available>	
__run, docrunner.py:1	
run, docrunner.py:1525	
start, docrunner.py:13	
<module>, docrunner	

Evaluate expression (Enter) or add a watch (Ctrl+Shift+Enter)

! digits = {list: 1} [0]

max_frac = {int} 3
negative = {bool} False

self = {Fraction} <fraction_decimal_str_err.Fraction object at 0x7d37f0a0ed50>

a = {int} 18399

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.

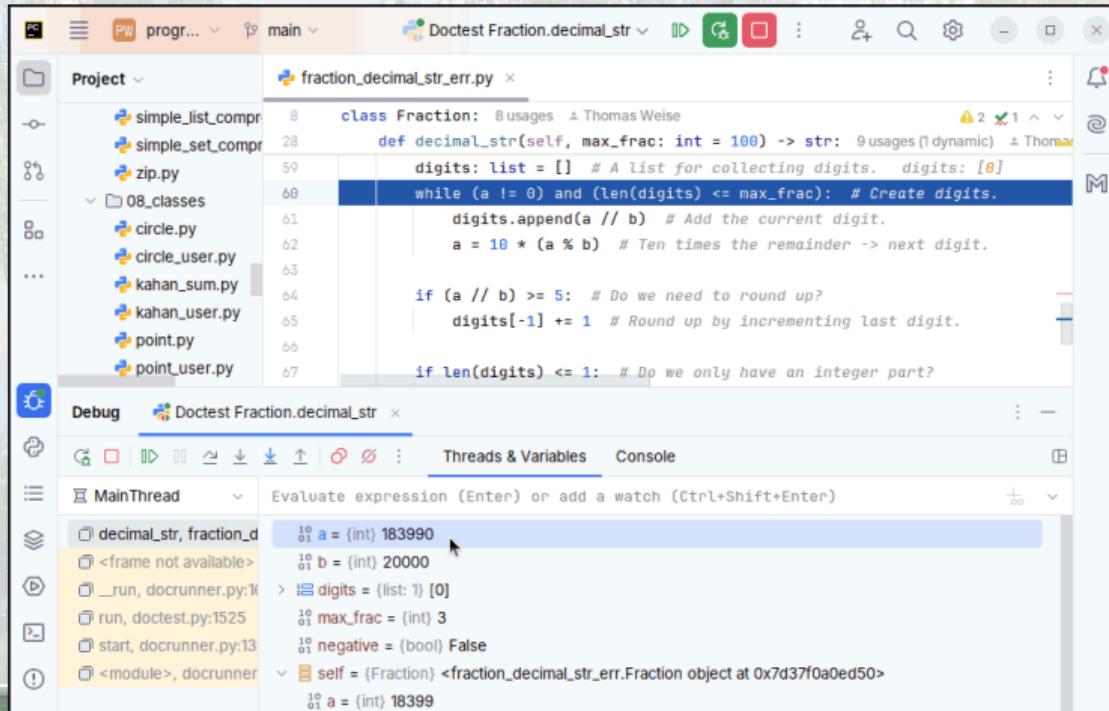
A screenshot of the PyCharm IDE interface. The top window shows a Python file named 'fraction_decimal_str_err.py' with code for a Fraction class. A breakpoint is set at line 62, where the variable 'digits' is shown to have the value [0]. The bottom window is the 'Threads & Variables' tab of the debugger, showing the current thread 'MainThread'. It lists several variables with their values:

- a = {int} 18399
- b = {int} 20000
- ! digits = {list: 1} [0]
- max_frac = {int} 3
- negative = {bool} False
- self = {Fraction} <fraction_decimal_str_err.Fraction object at 0x7d37f0a0ed50>
- a = {int} 18399

The variable 'digits' is highlighted with a blue selection bar.

Debuggen in PyCharm

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



The screenshot shows the PyCharm IDE interface. The top navigation bar includes tabs for 'PC', 'PW', 'program...', 'main', and 'Doctest Fraction.decimal_str'. The main window has a 'Project' view on the left showing files like 'simple_list_compr.py', 'simple_set_compr.py', 'zip.py', and several files under '08_classes' such as 'circle.py', 'circle_user.py', 'kahan_sum.py', 'kahan_user.py', 'point.py', and 'point_user.py'. The central code editor displays the 'fraction_decimal_str_err.py' file with the following code:

```
class Fraction:
    def decimal_str(self, max_frac: int = 100) -> str:
        digits: 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?
```

The line `a = 10 * (a % b)` is highlighted in blue, indicating it is the current line of execution. Below the code editor, the 'Debug' tool window is open, showing the 'Threads & Variables' tab. It lists variables and their values:

- `a`: `{int} 183990`
- `b`: `{int} 20000`
- `digits`: `{list: 1} [0]`
- `max_frac`: `{int} 3`
- `negative`: `{bool} False`
- `self`: `{Fraction} <fraction_decimal_str_err.Fraction object at 0x7d37f0a0ed50>`

The variable `a` is currently selected in the list.

Debuggen in PyCharm



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

The screenshot shows the PyCharm IDE interface during a debugging session. The top navigation bar includes tabs for 'PC', 'PWA', 'Program', 'main', and 'Doctest Fraction.decimal_str'. The main window displays a project structure with files like 'simple_list_compr.py', 'simple_set_compr.py', 'zip.py', and several files under '08_classes' such as 'circle.py', 'circle_user.py', 'kahan_sum.py', 'kahan_user.py', 'point.py', and 'point_user.py'. The code editor shows a portion of the 'Fraction' class with a specific line highlighted: `a = 10 * (a % b)`. Below the code editor, the 'Threads & Variables' tab is selected in the debugger tool window. The variable pane shows the current values of variables: `a = {int} 183990`, `b = {int} 20000`, `digits = {list: 1} [0]`, `max_frac = {int} 3`, `negative = {bool} False`, and `self = {Fraction} <fraction_decimal_str_err.Fraction object at 0x7d37f0a0ed50>`. The bottom of the debugger window also shows the value `a = {int} 18399`.

Debuggen in PyCharm



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

The screenshot shows the PyCharm IDE interface during a debugging session. The top navigation bar includes tabs for 'PC', 'PWA', 'Program', 'main', and 'Doctest Fraction.decimal_str'. The main window displays a project structure with files like `simple_list_compr.py`, `simple_set_compr.py`, `zip.py`, and several files under the `08_classes` directory. The code editor shows a portion of `fraction_decimal_str_err.py`. A line of code is highlighted: `while (a != 0) and (len(digits) <= max_frac): # Create digits.`. The bottom part of the interface shows the 'Threads & Variables' tab of the debugger, which lists the current thread as 'MainThread'. It also shows a watch list with the expression `a = {int} 183990` and other local variables: `b = {int} 20000`, `digits = {list: 1} [0]`, `max_frac = {int} 3`, `negative = {bool} False`, and `self = {Fraction} <fraction_decimal_str_err.Fraction object at 0x7d37f0a0ed50>`. The variable `a` is currently selected in the watch list.

Debuggen in PyCharm



- Nun wird der Kopf der Schleifen mit deren Bedingung wieder markiert.
- Wir drücken Shift+F10 um ihn auszuführen.

The screenshot shows the PyCharm IDE interface. The top navigation bar includes tabs for 'PC', 'PWA', 'Program', 'main', and 'Doctest Fraction.decimal_str'. The main area displays a Python file named 'fraction_decimal_str_err.py' with the following code:

```
class Fraction:
    def decimal_str(self, max_frac: int = 100) -> str:
        digits: 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?
```

The code editor has several annotations: a yellow warning icon at line 28, a green checkmark at line 59, and a red error icon at line 60. The line 60 annotation points to the condition in the while loop. The bottom part of the interface shows the 'Threads & Variables' tab of the debugger, which lists variables and their values:

Variable	Value
a	{int} 183990
b	{int} 20000
digits	{list: 1} [0]
max_frac	{int} 3
negative	{bool} False
self	{Fraction} <fraction_decimal_str_err.Fraction object at 0x7d37f0a0ed50>
a	{int} 18399

Debuggen in PyCharm

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

The screenshot shows the PyCharm IDE interface during a debugging session. The top navigation bar displays the project name "Doctest Fraction.decimal_str" and the file "fraction_decimal_str_err.py". The code editor shows a portion of the `Fraction` class definition, specifically the `decimal_str` method. A line of code is highlighted: `while (a != 0) and (len(digits) <= max_frac):`. The line number 61 is visible above the code. The bottom part of the interface is the debugger tool window, which has tabs for "Threads & Variables" and "Console". The "Variables" tab is active, showing a list of variables with their current values. The variable `a` is highlighted with a blue selection bar, showing its value as `183990`. Other variables listed include `b`, `max_frac`, `negative`, and `self`.

```
class Fraction:
    def decimal_str(self, max_frac: int = 100) -> str:
        digits: 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?
```

Threads & Variables

Variable	Type	Value
a	int	183990
b	int	20000
max_frac	int	3
negative	bool	False
self	Fraction	<fraction_decimal_str_err.Fraction object at 0x7d37f0a0ed50>

Debuggen in PyCharm



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

The screenshot shows the PyCharm IDE interface. The top window displays the code for `fraction_decimal_str_err.py`. A breakpoint is set at line 62, where the variable `digits` is being modified. The bottom window shows the debugger's Threads & Variables tab, where the variable `digits` is highlighted with a value of `[0, 9]`.

```
class Fraction: 8 usages ± Thomas Weise
    def decimal_str(self, max_frac: int = 100) -> str: 9 usages (!dynamic) ± Thomas Weise
        digits: list = [] # A list for collecting digits. digits: [0, 9]
        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?
```

Threads & Variables

Frame	Value
decimal_str, fraction_d	a = {int} 183990 b = {int} 20000
<frame not available>	
__run, docrunner.py:1	
run, docrunner.py:1525	
start, docrunner.py:13	
<module>, docrunner	

Debuggen in PyCharm

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



The screenshot shows the PyCharm IDE interface. The top navigation bar includes tabs for 'PC', 'Program', 'File', 'Edit', 'Run', 'View', 'Tools', 'Help', and 'File'. The title bar displays 'Doctest Fraction.decimal_str' and the file path 'main'. The left sidebar shows the project structure with files like 'simple_list_compr.py', 'simple_set_compr.py', 'zip.py', and several files under '08_classes' such as 'circle.py', 'circle_user.py', 'kahan_sum.py', 'kahan_user.py', 'point.py', and 'point_user.py'. The main editor window shows the code for the 'Fraction' class, specifically the implementation of the 'decimal_str' method. A line of code at line 60 is highlighted: 'while (a != 0) and (len(digits) <= max_frac): # Create digits.' The bottom part of the interface is the debugger, with tabs for 'Threads & Variables' and 'Console'. The 'Threads & Variables' tab is active, showing the current thread 'MainThread'. In the variable list, the variable 'a' is highlighted and has a tooltip showing its value as '`39900`'. Other variables listed include 'b', 'digits', 'max_frac', 'negative', and 'self'. The 'Evaluate expression (Enter) or add a watch (Ctrl+Shift+Enter)' input field also contains the value '`39900`'.

```
class Fraction:
    def decimal_str(self, max_frac: int = 100) -> str:
        digits: 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?
```

Threads & Variables

MainThread

Evaluate expression (Enter) or add a watch (Ctrl+Shift+Enter)

- decimal_str, fraction_d
- <frame not available>
- _run, docrunner.py:1
- run, doctest.py:1525
- start, docrunner.py:13
- <module>, docrunner

10 a = {int} 39900

10 b = {int} 20000

> 10 digits = {list: 2} [0, 9]

10 max_frac = {int} 3

10 negative = {bool} False

10 self = {Fraction} <fraction_decimal_str_err.Fraction object at 0x7d37f0a0ed50>

10 a = {int} 18399

Debuggen in PyCharm

- Wir kommen zur nächsten Iteration der Schleife.



The screenshot shows the PyCharm IDE interface during a debugging session. The top navigation bar includes tabs for 'PC', 'PWA', 'Program', 'main', and 'Doctest Fraction.decimal_str'. The main window displays a project structure with files like 'simple_list_compr.py', 'simple_set_compr.py', 'zip.py', and several files under '08_classes' such as 'circle.py', 'circle_user.py', 'kahan_sum.py', 'kahan_user.py', 'point.py', and 'point_user.py'. The code editor shows a portion of 'fraction_decimal_str_err.py' with the following code:

```
class Fraction:
    def decimal_str(self, max_frac: int = 100) -> str:
        digits: list = [] # A list for collecting digits. digits: [0, 9]
        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?
```

The bottom panel shows the 'Threads & Variables' tab of the debugger, which lists variables and their values:

Variable	Value
a	{int} 39900
b	{int} 20000
digits	{list: 2} [0, 9]
max_frac	{int} 3
negative	{bool} False
self	{Fraction} <fraction_decimal_str_err.Fraction object at 0x7d37f0a0ed50>

Debuggen in PyCharm

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



The screenshot shows the PyCharm IDE interface. The top navigation bar includes tabs for 'PC', 'Program', 'main', and 'Doctest Fraction.decimal_str'. The main window displays a project structure with files like 'simple_list_compr.py', 'simple_set_compr.py', 'zip.py', and several files under '08_classes' such as 'circle.py', 'circle_user.py', 'kahan_sum.py', 'kahan_user.py', 'point.py', and 'point_user.py'. The current file open is 'fraction_decimal_str_err.py'. The code in this file defines a 'Fraction' class with a method 'decimal_str' that calculates digits of a division. A breakpoint is set at line 62, where the code appends a digit to a list named 'digits'. The code is annotated with comments explaining its logic. The bottom part of the interface shows the 'Threads & Variables' tab of the debugger, which lists variables and their values. The variable 'digits' is highlighted, showing its value as '[0, 9, 1]'. Other visible variables include 'a' (39900), 'b' (20000), 'max_frac' (100), 'negative' (False), and 'self' (a Fraction object).

```
class Fraction:
    def decimal_str(self, max_frac: int = 100) -> str:
        digits: list = [] # A list for collecting digits. digits: [0, 9, 1]
        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?
```

MainThread

Evaluate expression (Enter) or add a watch (Ctrl+Shift+Enter)

- decimal_str, fraction_d
- <frame not available>
- _run, docrunner.py:1
- run, doctest.py:1525
- start, docrunner.py:13
- <module>, docrunner

10 a = {int} 39900
10 b = {int} 20000
> 10 digits = {list: 3} [0, 9, 1]
10 max_frac = {int} 3
10 negative = {bool} False
10 self = {Fraction} <fraction_decimal_str_err.Fraction object at 0x7d37f0a0ed50>
10 a = {int} 18399

Debuggen in PyCharm

- Dann wird `a` auf 199000 upgdated.

The screenshot shows the PyCharm IDE interface. The top bar displays the file path `Programmierung/main` and the current file `Fraction.decimal_str`. The main area is a code editor with the following code:

```
class Fraction:
    def decimal_str(self, max_frac: int = 100) -> str:
        digits: 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.

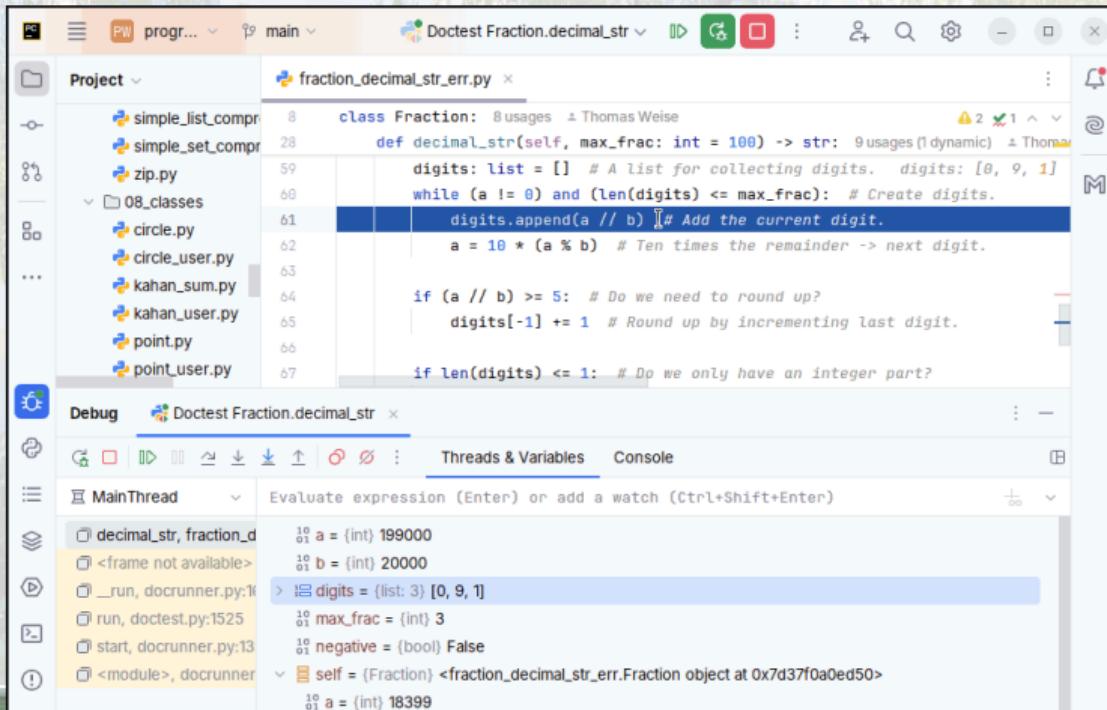
        if len(digits) <= 1: # Do we only have an integer part?
```

The line `while (a != 0) and (len(digits) <= max_frac):` is highlighted with a blue selection bar. Below the code editor, the `Threads & Variables` tab of the `Debug` tool window is active. The variable `digits` is listed with the value `[0, 9, 1]`.

```
decimal_str, fraction_d
<frame not available>
__run, docrunner.py:1525
run, doctest.py:1525
start, docrunner.py:13
<module>, docrunner
a = {int} 199000
b = {int} 20000
digits = {list: 3} [0, 9, 1]
max_frac = {int} 3
negative = {bool} False
self = {Fraction} <fraction_decimal_str_err.Fraction object at 0x7d37f0a0ed50>
a = {int} 18399
```

Debuggen in PyCharm

- Ein neuer Schleifendurchlauf beginnt.



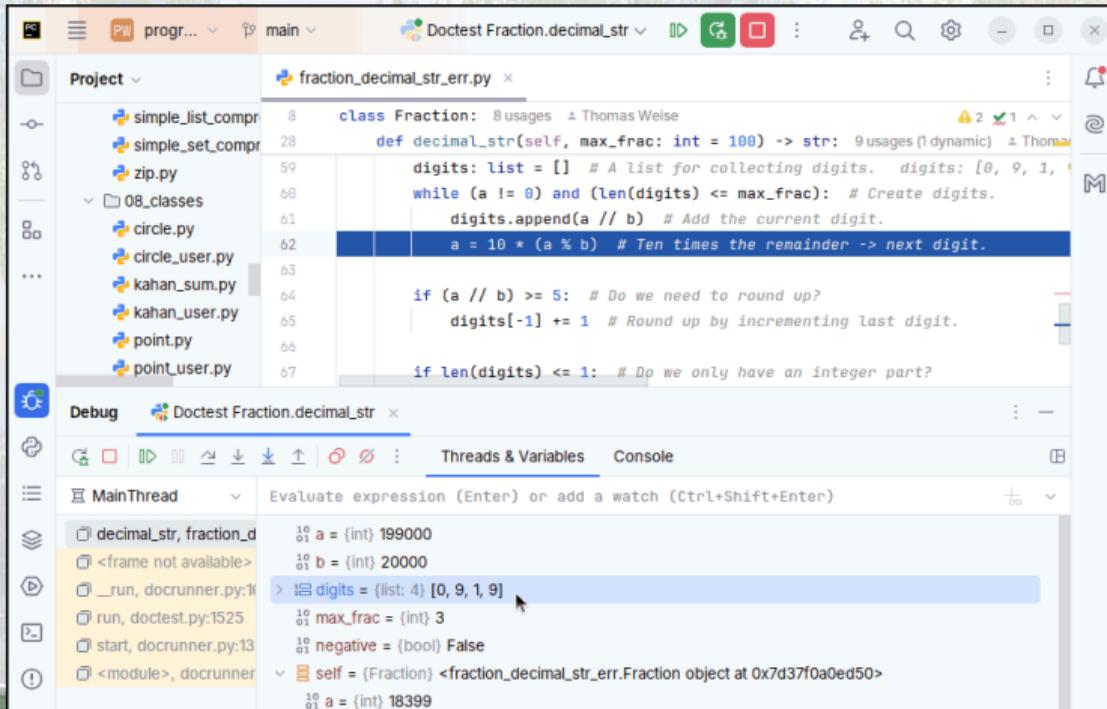
The screenshot shows the PyCharm IDE interface. The top part is the code editor with the file `fraction_decimal_str_err.py` open. The code defines a `Fraction` class with a `decimal_str` method. A breakpoint is set at line 61, which contains the line `digits.append(a // b)`. The bottom part is the debugger tool window titled "Debug". It shows the "Threads & Variables" tab selected. The variable `digits` is expanded, showing its value as `[0, 9, 1]`. Other variables shown include `a`, `b`, `max_frac`, `negative`, and `self`. The background of the IDE shows a blurred view of a lake and trees.

```
class Fraction:
    def decimal_str(self, max_frac: int = 100) -> str:
        digits: list = []
        while (a != 0) and (len(digits) <= max_frac):
            digits.append(a // b)
            a = 10 * (a % b)
        if (a // b) >= 5:
            digits[-1] += 1
        if len(digits) <= 1:
            return str(a)
        else:
            return ''.join(str(digit) for digit in digits)

a = 199000
b = 20000
max_frac = 3
negative = False
self = Fraction object at 0x7d37f0a0ed50
a = 18399
```

Debuggen in PyCharm

- Nun wird 9 an `digits` angehängt.



The screenshot shows the PyCharm IDE interface. The top part is the code editor with the file `fraction_decimal_str_err.py` open. The code defines a `Fraction` class with a `decimal_str` method. A breakpoint is set at line 62, where the line `a = 10 * (a % b)` is highlighted. The bottom part is the debugger tool window titled "Debug". It shows the "Threads & Variables" tab selected. In the variables pane, there is a list of local variables:

- `a = {int} 199000`
- `b = {int} 20000`
- `digits = {list: 4} [0, 9, 1, 9]` (This line is highlighted in blue, indicating it is the current expression being evaluated.)
- `max_frac = {int} 3`
- `negative = {bool} False`
- `self = {Fraction} <fraction_decimal_str_err.Fraction object at 0x7d37f0a0ed50>`
- `a = {int} 18399`

Debuggen in PyCharm

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



The screenshot shows the PyCharm IDE interface during a debugging session. The top navigation bar includes tabs for 'PC', 'Program', 'main', and 'Doctest Fraction.decimal_str'. The main window displays a project structure with files like 'simple_list_compr.py', 'simple_set_compr.py', 'zip.py', and several files under '08_classes' such as 'circle.py', 'circle_user.py', 'kahan_sum.py', 'kahan_user.py', 'point.py', and 'point_user.py'. The code editor shows a portion of 'fraction_decimal_str_err.py' with a breakpoint at line 60. The code defines a `Fraction` class with a `decimal_str` method. The variable `a` is highlighted in blue. The bottom panel shows the 'Threads & Variables' tab of the debugger, which lists the current thread 'MainThread' and variables. The variable `a` is shown with its value as `{int} 190000`. Other listed variables include `b`, `digits`, `max_frac`, `negative`, and `self`.

```
class Fraction:
    def decimal_str(self, max_frac: int = 100) -> str:
        digits: 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?
```

Threads & Variables

Variable	Type	Value
a	{int}	190000
b	{int}	20000
digits	{list: 4}	[0, 9, 1, 9]
max_frac	{int}	3
negative	{bool}	False
self	{Fraction}	<fraction_decimal_str_err.Fraction object at 0x7d37f0a0ed50>

Debuggen in PyCharm



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

The screenshot shows the PyCharm IDE interface. The top part is the code editor with the file `fraction_decimal_str_err.py` open. The code defines a `Fraction` class with a `decimal_str` method. A breakpoint is set at line 60, where the variable `a` is being modified. The bottom part is the debugger tool window, titled "Debug Doctest Fraction.decimal_str". It shows the current thread is "MainThread" and displays the expression `a = {int} 190000` in the "Evaluate expression" field. The stack trace shows the call path from `__run` in `docrunner.py` to `run` in `doctest.py`.

```
class Fraction:
    def decimal_str(self, max_frac: int = 100) -> str:
        digits: list = []
        while (a != 0) and (len(digits) <= max_frac):
            digits.append(a // b)
            a = 10 * (a % b)
        if (a // b) >= 5:
            digits[-1] += 1
        if len(digits) <= 1:
            return str(a)
        else:
            return digits[-1] + '.' + digits[:-1]

    def __str__(self) -> str:
        return self.decimal_str()
```

Threads & Variables

```
MainThread
decimal_str, fraction_d
<frame not available>
__run, docrunner.py:1525
run, doctest.py:1525
start, docrunner.py:13
<module>, docrunner
```

Evaluate expression (Enter) or add a watch (Ctrl+Shift+Enter)

```
19 a = {int} 190000
19 b = {int} 20000
19 digits = {list: 4} [0, 9, 1, 9]
19 max_frac = {int} 3
19 negative = {bool} False
19 self = {Fraction} <fraction_decimal_str_err.Fraction object at 0x7d37f0a0ed50>
19 a = {int} 18399
```

Debuggen in PyCharm



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

The screenshot shows the PyCharm IDE interface. The top navigation bar includes tabs for 'PC', 'PWA', 'Program', 'main', and 'Doctest Fraction.decimal_str'. The main area is a code editor with the file 'fraction_decimal_str_err.py' open. The code defines a class `Fraction` with a method `decimal_str`. A breakpoint is set at line 60, where the variable `digits` is assigned the value `[0, 9, 1, 9]`. Below the code editor, the 'Debug' tool window is active, showing the 'Threads & Variables' tab. It lists the current thread 'MainThread' and provides an 'Evaluate expression' input field. The expression `a = {int} 190000` is entered, and its result `b = {int} 20000` is shown. Other variables listed include `digits`, `max_frac`, `negative`, and `self`.

```
class Fraction:
    def decimal_str(self, max_frac: int = 100) -> str:
        digits: 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?
```

Threads & Variables

Variable	Type	Value
a	{int}	190000
b	{int}	20000
digits	{list: 4}	[0, 9, 1, 9]
max_frac	{int}	3
negative	{bool}	False
self	{Fraction}	<fraction_decimal_str_err.Fraction object at 0x7d37f0a0ed50>

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.

A screenshot of the PyCharm IDE interface. The top window shows a Python file named 'fraction_decimal_str_err.py' with code for a Fraction class. A specific line of code is highlighted: 'while (a != 0) and (len(digits) <= max_frac):'. Below this, the code continues with digit appending logic and rounding logic. The bottom window is a debugger tool window titled 'Debug' with tabs for 'Threads & Variables' and 'Console'. It shows a list of frames, with the first frame expanded to show local variables: 'a = {int} 190000', 'b = {int} 20000', 'digits = {list: 4} [0, 9, 1, 9]', 'max_frac = {int} 3', and 'negative = {bool} False'. The variable 'a' is currently selected. The background of the entire interface is a blurred image of a lake and trees.

Debuggen in PyCharm



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

A screenshot of the PyCharm IDE interface. The top navigation bar shows 'PC' and 'program...' with a dropdown arrow, followed by 'main'. The title bar says 'Doctest Fraction.decimal_str'. The main area is a code editor for 'fraction_decimal_str_err.py' with the following code:

```
class Fraction: 8 usages ± Thomas Weise
    def decimal_str(self, max_frac: int = 100) -> str: 9 usages (!dynamic) ± Thomas Weise
        digits: list = [] # A list for collecting digits. digits: [0, 9, 1, 9]
        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?
```

The status bar at the bottom shows 'MainThread'. The bottom panel contains a 'Threads & Variables' tab and a 'Console' tab. The 'Variables' tab is active, showing the current state of variables:

Variable	Type	Value
a	{int}	190000
b	{int}	20000
digits	{list: 4}	[0, 9, 1, 9]
max_frac	{int}	3
negative	{bool}	False
self	{Fraction}	<fraction_decimal_str_err.Fraction object at 0x7d37f0a0ed50>

Debuggen in PyCharm



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

The screenshot shows the PyCharm IDE interface. The top navigation bar includes tabs for 'PC', 'PWA', 'Program', 'main', and 'Doctest Fraction.decimal_str'. The main area displays a Python file named 'fraction_decimal_str_err.py' with code related to a 'Fraction' class. A specific line of code is highlighted in blue: 'if (a // b) >= 5: # Do we need to round up?'. Below the code editor, the 'Debug' tab is selected in the toolbar. The bottom panel shows the 'Threads & Variables' tab of the debugger, with a list of variables and their values. The variable 'a' is set to 190000 and 'b' to 20000. Other variables listed include 'digits', 'max_frac', 'negative', and 'self'. The background of the slide features a photograph of a willow tree with drooping branches over water.

```
class Fraction:
    def decimal_str(self, max_frac: int = 100) -> str:
        digits: 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?
```

Threads & Variables

Variable	Type	Value
a	{int}	190000
b	{int}	20000
digits	{list: 4}	[0, 9, 1, 9]
max_frac	{int}	3
negative	{bool}	False
self	{Fraction}	<fraction_decimal_str_err.Fraction object at 0x7d37f0a0ed50>

Debuggen in PyCharm



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

The screenshot shows the PyCharm IDE interface. The top window displays the code for `fraction_decimal_str_err.py`. A specific line of code is highlighted:

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

The bottom window is the debugger, titled "Debug Doctest Fraction.decimal_str". It shows the current stack trace and the values of variables:

MainThread

Evaluate expression (Enter) or add a watch (Ctrl+Shift+Enter)

```
a = {int} 190000  
b = {int} 20000  
digits = {list: 4} [0, 9, 1, 9]  
max_frac = {int} 3  
negative = {bool} False  
self = {Fraction} <fraction_decimal_str_err.Fraction object at 0x7d37f0a0ed50>  
a = {int} 18399
```

Debuggen in PyCharm



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

The screenshot shows the PyCharm IDE interface during a debugging session. The top navigation bar includes tabs for 'PC', 'PWA', 'Program', 'main', and 'Doctest Fraction.decimal_str'. The main window displays a project structure with files like 'simple_list_compr.py', 'simple_set_compr.py', 'zip.py', and several files under '08_classes' such as 'circle.py', 'circle_user.py', 'kahan_sum.py', 'kahan_user.py', 'point.py', and 'point_user.py'. The code editor shows a portion of 'fraction_decimal_str_err.py' with the following code:

```
class Fraction:
    def decimal_str(self, max_frac: int = 100) -> str:
        digits: 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?
```

The line `if (a // b) >= 5:` is highlighted in blue, indicating it is the current line being executed or has just been executed. The bottom panel features a debugger tool window with tabs for 'Threads & Variables' and 'Console'. The 'Threads & Variables' tab is active, showing the current thread 'MainThread' and a list of variables:

- `a = {int} 190000`
- `b = {int} 20000`
- `digits = {list: 4} [0, 9, 1, 9]`
- `max_frac = {int} 3`
- `negative = {bool} False`
- `self = {Fraction} <fraction_decimal_str_err.Fraction object at 0x7d37f0a0ed50>`

The variable `self` is currently selected in the list.

Debuggen in PyCharm



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

The screenshot shows the PyCharm IDE interface. The top navigation bar includes tabs for 'PC', 'Program', 'main', and 'Doctest Fraction.decimal_str'. The main area displays a Python file named 'fraction_decimal_str_err.py' with the following code:

```
class Fraction:
    def decimal_str(self, max_frac: int = 100) -> str:
        digits: list = [] # A list for collecting digits.
        while (a != 0) and (len(digits) <= max_frac):
            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?
```

The line of code being debugged is highlighted in blue: `if (a // b) >= 5: # Do we need to round up?`. The bottom panel shows the 'Threads & Variables' tab of the debugger, with the 'MainThread' selected. It lists local variables and their values:

- a = {int} 190000
- b = {int} 20000
- digits = {list: 4} [0, 9, 1, 9]
- max_frac = {int} 3
- negative = {bool} False
- self = {Fraction} <fraction_decimal_str_err.Fraction object at 0x7d37f0a0ed50>

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.

The screenshot shows the PyCharm IDE interface. The top navigation bar includes tabs for 'Project', 'File', 'Edit', 'Run', 'View', 'Tools', 'Help', and 'PyCharm'. The title bar displays 'Doctest Fraction.decimal_str' and the file path 'main'. The left sidebar shows a project structure with files like 'simple_list_compr.py', 'simple_set_compr.py', 'zip.py', and several files under '08_classes' such as 'circle.py', 'circle_user.py', 'kahan_sum.py', 'kahan_user.py', 'point.py', and 'point_user.py'. The main code editor window contains Python code for a 'Fraction' class. A specific line of code is highlighted in blue: 'if digits[-1] += 1: # Round up by incrementing last digit.' The bottom part of the interface is the debugger tool window, titled 'Debug Doctest Fraction.decimal_str'. It shows a stack trace with frames like 'decimal_str', 'fraction_d', and 'run'. The 'Threads & Variables' tab is selected, showing local variable values: 'a = {int} 190000', 'b = {int} 20000', 'max_frac = {int} 3', and 'negative = {bool} False'. The 'Evaluate expression (Enter) or add a watch (Ctrl+Shift+Enter)' input field also displays 'a = {int} 190000'.

```
class Fraction:
    def decimal_str(self, max_frac: int = 100) -> str:
        digits: list = [] # A list for collecting digits.
        while (a != 0) and (len(digits) <= max_frac):
            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?
```

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.

The screenshot shows the PyCharm IDE interface. The top bar displays the project name "Doctest Fraction.decimal_str" and the file "fraction_decimal_str_err.py". The code editor shows a class `Fraction` with a method `decimal_str`. A breakpoint is set at line 65, where the code increments the last digit if the current division result is 5 or greater. The bottom part of the interface is the debugger toolbar, which includes buttons for running, stopping, and stepping through code. The "Threads & Variables" tab is selected, showing a list of variables and their values. The variable `a` is set to 190000 and `b` to 20000. The variable `digits` is shown as a list [0, 9, 1, 9]. The variable `self` is a `Fraction` object.

```
class Fraction:
    def decimal_str(self, max_frac: int = 100) -> str:
        digits: list = [] # A list for collecting digits.
        while (a != 0) and (len(digits) <= max_frac):
            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?
```

Evaluate expression (Enter) or add a watch (Ctrl+Shift+Enter)

- decimal_str, fraction_d
- <frame not available>
- _run, docrunner.py:1
- run, doctest.py:1525
- start, docrunner.py:13
- <module>, docrunner

```
19 a = {int} 190000
19 b = {int} 20000
19 digits = {list: 4} [0, 9, 1, 9]
19 max_frac = {int} 3
19 negative = {bool} False
19 self = {Fraction} <fraction_decimal_str_err.Fraction object at 0x7d37f0a0ed50>
19 a = {int} 18399
```

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.

The screenshot shows the PyCharm IDE interface. The top window is the code editor displaying `fraction_decimal_str_err.py`. The code defines a `Fraction` class with a `decimal_str` method. A conditional statement `if (a // b) >= 5:` is highlighted in blue. Below the code editor is the debugger toolbar, which includes buttons for running, stopping, and stepping through code. The bottom window is the debugger's "Threads & Variables" tab, showing the current stack frames and variable values. The variable `a` is set to 190000 and `b` to 20000, with the expression `190000 // 20000` evaluated to 9.

```
class Fraction: 8 usages ± Thomas Weise
    def decimal_str(self, max_frac: int = 100) -> str: 9 usages (!dynamic) ± Thomas Weise
        digits: list = [] # A list for collecting digits. digits: [0, 9, 1, 9]
        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?
```

```
decimal_str, fraction_decimal_str_err.py:10 a = {int} 190000
decimal_str, fraction_decimal_str_err.py:10 b = {int} 20000
run, docrunner.py:1525 digits = {list: 4} [0, 9, 1, 9]
run, docrunner.py:1525 max_frac = {int} 3
run, docrunner.py:1525 negative = {bool} False
run, docrunner.py:1525 self = {Fraction} <fraction_decimal_str_err.Fraction object at 0x7d37f0a0ed50>
run, docrunner.py:1525 a = {int} 18399
```

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`.

The screenshot shows the PyCharm IDE interface with the following details:

- Project View:** Shows the project structure with files like `simple_list_compr.py`, `simple_set_compr.py`, `zip.py`, and several files under the `08_classes` directory including `circle.py`, `circle_user.py`, `kahan_sum.py`, `kahan_user.py`, `point.py`, and `point_user.py`.
- Main Editor:** Displays the code for `fraction_decimal_str_err.py`. The line `if (a // b) >= 5:` is highlighted in blue, indicating it is currently being debugged. The line `digits[-1] += 1` is also highlighted, showing the step-by-step execution flow.
- Debug Tool Window:** Shows the current thread as `MainThread` and provides options to evaluate expressions or add watches. The expression `a = {int} 190000` is currently selected.
- Call Stack:** Lists the call stack frames, starting with `decimal_str, fraction_d` at the top, followed by `<frame not available>`, `__run, docrunner.py:1`, `run, doctest.py:1525`, `start, docrunner.py:13`, and `<module>, docrunner` at the bottom.

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.

The screenshot shows the PyCharm IDE interface. The top window displays the code for `fraction_decimal_str_err.py`. The code defines a `Fraction` class with a `decimal_str` method. The current line of code is highlighted: `if digits[-1] >= 5:`. Below this, the code continues with `digits[-1] += 1 # Round up by incrementing last digit.`. The bottom window is the debugger, titled "Debug Doctest Fraction.decimal_str". It shows the stack trace and the value of variable `a` as 190000. The code editor has syntax highlighting and some annotations in yellow.

```
class Fraction:
    def decimal_str(self, max_frac: int = 100) -> str:
        digits: list = [] # A list for collecting digits.
        while (a != 0) and (len(digits) <= max_frac):
            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?
```

Debug Doctest Fraction.decimal_str

MainThread

Evaluate expression (Enter) or add a watch (Ctrl+Shift+Enter)

decimal_str, fraction_d

a = {int} 190000

b = {int} 20000

run, docrunner.py:1525

start, docrunner.py:13

self = {Fraction} <fraction_decimal_str_err.Fraction object at 0x7d37f0a0ed50>

a = {int} 18399

Debuggen in PyCharm



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

The screenshot shows the PyCharm IDE interface. The top navigation bar includes tabs for 'PC', 'PWI', 'program...', 'main', and 'Doctest Fraction.decimal_str'. The main area displays a Python file named 'fraction_decimal_str_err.py' with the following code:

```
class Fraction:
    def decimal_str(self, max_frac: int = 100) -> str:
        digits: list = [] # A list for collecting digits.
        while (a != 0) and (len(digits) <= max_frac):
            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?
```

The line of code being debugged is highlighted in blue: `if (a // b) >= 5: # Do we need to round up?`. The bottom panel shows the 'Threads & Variables' tab of the debugger, with the 'MainThread' selected. It lists local variables and their values:

- a = {int} 190000
- b = {int} 20000
- digits = {list: 4} [0, 9, 1, 9]
- max_frac = {int} 3
- negative = {bool} False
- self = {Fraction} <fraction_decimal_str_err.Fraction object at 0x7d37f0a0ed50>

Debuggen in PyCharm



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

The screenshot shows the PyCharm IDE interface. The top navigation bar includes tabs for 'PC', 'Program', 'main', and 'Doctest Fraction.decimal_str'. The main area is a code editor with the file 'fraction_decimal_str_err.py' open. The code defines a class 'Fraction' with a method 'decimal_str'. A specific line of code is highlighted with a blue selection bar:

```
    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.
```

The bottom part of the interface is a 'Threads & Variables' tool window titled 'MainThread'. It displays a list of current threads and variables. One variable, 'digits', is selected and shown in a expanded list:

- decimal_str, fraction_d
- <frame not available>
- _run, docrunner.py:10
- run, doctest.py:1525
- start, docrunner.py:13
- <module>, docrunner

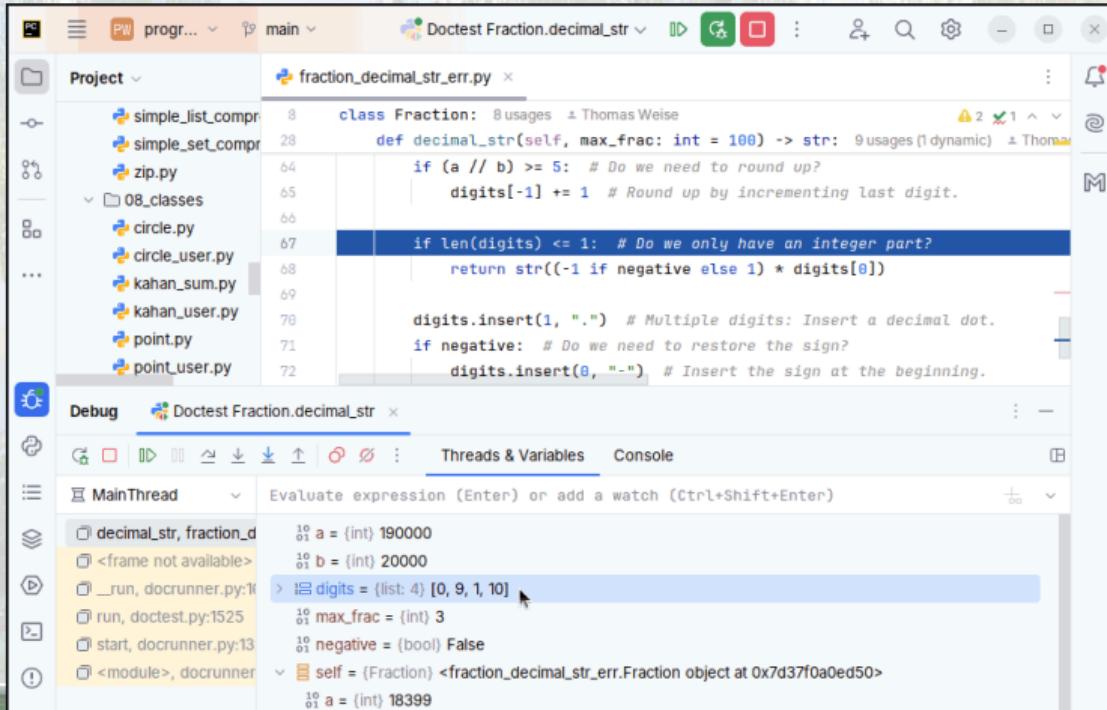
Evaluate expression (Enter) or add a watch (Ctrl+Shift+Enter)

```
10 a = {int} 190000  
10 b = {int} 20000  
> 10 digits = {list: 4} [0, 9, 1, 10]  
10 max_frac = {int} 3  
10 negative = {bool} False  
10 self = {Fraction} <fraction_decimal_str_err.Fraction object at 0x7d37f0a0ed50>  
10 a = {int} 18399
```

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.



The screenshot shows the PyCharm IDE interface. The top navigation bar includes tabs for 'PC', 'Program', 'main', and 'Doctest Fraction.decimal_str'. The main window displays a project structure with files like simple_list_compr, simple_set_compr, zip.py, and several files under the 08_classes folder (circle.py, circle_user.py, kahan_sum.py, kahan_user.py, point.py, point_user.py). The code editor shows a class Fraction with a method decimal_str. A specific line of code is highlighted in blue: 'if len(digits) <= 1: # Do we only have an integer part?'. Below the code editor, the 'Threads & Variables' tab is selected in the 'Debug' tool window. The variable 'digits' is listed with a value of [list: 4] [0, 9, 1, 10]. Other variables shown include a, b, max_frac, negative, and self.

```
class Fraction:
    def decimal_str(self, max_frac: int = 100) -> str:
        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.
```

Threads & Variables

```
a = {int: 190000}
b = {int: 20000}
max_frac = {int: 3}
negative = {bool: False}
self = {Fraction} <fraction_decimal_str_err.Fraction object at 0x7d37f0a0ed50>
a = {int: 18399}
```

Debuggen in PyCharm



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

A screenshot of the PyCharm IDE interface. The top navigation bar shows 'PC' and 'program...'. The main window has tabs for 'Project' and 'fraction_decimal_str_err.py'. The code editor shows a class Fraction with a method decimal_str. A specific line of code is highlighted: 'if len(digits) <= 1: # Do we only have an integer part?'. Below the code editor is a 'Threads & Variables' tab in the debugger tool window. The variable 'digits' is listed with a value of '[0, 9, 1, 10]'. Other variables shown include 'a' (190000), 'b' (20000), 'max_frac' (3), 'negative' (False), and 'self' (Fraction object).

```
class Fraction:
    def decimal_str(self, max_frac: int = 100) -> str:
        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.
```

Threads & Variables

Variable	Type	Value
a	int	190000
b	int	20000
max_frac	int	3
negative	bool	False
self	Fraction	<fraction_decimal_str_err.Fraction object at 0x7d37f0a0ed50>
a	int	18399

Debuggen in PyCharm



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

A screenshot of the PyCharm IDE interface. The top navigation bar shows "PC", "program...", "main", and "Doctest Fraction.decimal_str". The main window has a "Project" view on the left listing files like "simple_list_compr.py", "simple_set_compr.py", "zip.py", and several files under "08_classes" such as "circle.py", "circle_user.py", "kahan_sum.py", "kahan_user.py", "point.py", and "point_user.py". The code editor on the right shows a portion of "fraction_decimal_str_err.py":

```
class Fraction:
    def decimal_str(self, max_frac: int = 100) -> str:
        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.
```

The status bar at the bottom indicates "MainThread". The bottom panel shows the "Threads & Variables" tab is active, with a list of variables:

- decimal_str, fraction_d
- <frame not available>
- _run, docrunner.py:1025
- run, doctest.py:1525
- start, docrunner.py:13
- <module>, docrunner

A tooltip for the variable "digits" shows its value as `[list: 4] [0, 9, 1, 10]`. Other visible variable values include `a = {int} 190000`, `b = {int} 20000`, `max_frac = {int} 3`, `negative = {bool} False`, and `self = {Fraction} <fraction_decimal_str_err.Fraction object at 0x7d37f0a0ed50>`.

Debuggen in PyCharm



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

The screenshot shows the PyCharm IDE interface during a debugging session. The top window displays the code for `fraction_decimal_str_err.py`. A specific line of code is highlighted:

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

The bottom window shows the debugger's state. The current frame is `MainThread`. A tooltip indicates the value of the variable `digits`:

```
!> digits = [list: 4] [0, 9, 1, 10]
```

The variable `digits` is listed in the local variables pane, along with other variables like `a`, `b`, `max_frac`, `negative`, and `self`.

Debuggen in PyCharm



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

A screenshot of the PyCharm IDE interface. The top navigation bar shows 'PC' and 'programmierung' under 'File', and 'main' under 'Project'. The title bar says 'Doctest Fraction.decimal_str'. The main area shows a Python file 'fraction_decimal_str_err.py' with code related to fraction conversion. A specific line of code is highlighted: 'if len(digits) <= 1: # Do we only have an integer part?'. Below the code editor is a 'Threads & Variables' tab in the debugger tool window. The variable 'digits' is expanded, showing its value as '[0, 9, 1, 10]'. Other variables listed include 'a', 'b', 'max_frac', 'negative', and 'self'. The bottom status bar shows the path 'd:/doctests/fraction_decimal_str_err.py:10'.

Debuggen in PyCharm



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

A screenshot of the PyCharm IDE interface. The top navigation bar shows 'PC' and 'program...' with dropdown menus, followed by 'main' and other icons. The main window has a 'Project' view on the left listing files like 'simple_list_compr.py', 'simple_set_compr.py', 'zip.py', and several files under '08_classes'. The central editor area displays code for 'fraction_decimal_str_err.py'. A specific line of code is highlighted: 'if len(digits) <= 1: # Do we only have an integer part?'. Below the editor is a 'Threads & Variables' tab in the 'Debug' tool window. The variable pane shows a list of variables with their values. The variable 'digits' is selected, showing its value as '[list: 4] [0, 9, 1, 10]'. Other visible variables include 'a' (190000), 'b' (20000), 'max_frac' (3), 'negative' (False), and 'self' (a Fraction object).

```
class Fraction:
    def decimal_str(self, max_frac: int = 100) -> str:
        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.
```

```
a = {int} 190000
b = {int} 20000
max_frac = {int} 3
negative = {bool} False
self = {Fraction} <fraction_decimal_str_err.Fraction object at 0x7d37f0a0ed50>
a = {int} 18399
```

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.

A screenshot of the PyCharm IDE interface. The top window shows a Python file named 'fraction_decimal_str_err.py' with code related to fraction representation. A specific line of code is highlighted: 'if len(digits) <= 1: # Do we only have an integer part?'. Below this, the code continues with 'return str((-1 if negative else 1) * digits[0])', 'digits.insert(1, ".")', 'if negative:', and 'digits.insert(0, "-")'. The bottom window is a debugger tool titled 'Threads & Variables'. It shows a stack trace for 'MainThread' with several frames listed, including 'decimal_str', 'fraction_d', and frames from 'docrunner.py'. A tooltip or status bar at the bottom indicates the variable 'digits' has a value of '[list: 4] [0, 9, 1, 10]'.

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.

The screenshot shows the PyCharm IDE interface. The top bar displays the project name "Doctest Fraction.decimal_str" and the file "fraction_decimal_str_err.py". The code editor shows a class `Fraction` with a method `decimal_str`. A specific line of code is highlighted: `if len(digits) <= 1: # Do we only have an integer part?`. Below the code editor is the "Threads & Variables" tab of the debugger toolbar, which lists variables like `a`, `b`, and `digits`. The variable `digits` is currently selected, showing its value as `[0, 9, 1, 10]`.

```
class Fraction:
    def decimal_str(self, max_frac: int = 100) -> str:
        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.
```

```
decimal_str, fraction_d
<frame not available>
__run, docrunner.py:1025
run, doctest.py:1525
start, docrunner.py:13
<module>, docrunner
```

```
a = {int} 190000
b = {int} 20000
digits = {list: 4} [0, 9, 1, 10]
max_frac = {int} 3
negative = {bool} False
self = {Fraction} <fraction_decimal_str_err.Fraction object at 0x7d37f0a0ed50>
a = {int} 18399
```

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.

The screenshot shows the PyCharm IDE interface. The top navigation bar includes tabs for 'PC', 'File', 'Edit', 'Run', 'View', 'Tools', 'Help', and icons for 'Project', 'File', 'Run', 'Edit', 'Tools', 'Help'. The title bar displays 'Doctest Fraction.decimal_str' and the file path 'main'. The main area is a code editor with the file 'fraction_decimal_str_err.py' open. The code defines a class 'Fraction' with a method 'decimal_str'. A specific line of code is highlighted: 'if len(digits) <= 1: # Do we only have an integer part?'. Below the code editor is a 'Threads & Variables' tab in the 'Debug' tool window. The 'Threads & Variables' tab shows the current thread 'MainThread' and a list of variables. One variable, 'digits', is selected and highlighted in blue, showing its value as '[list: 4] [0, 9, 1, 10]'. Other variables listed include 'a' (value 190000), 'b' (value 20000), 'max_frac' (value 3), 'negative' (value False), and 'self' (value Fraction object at 0x7d37f0a0ed50). The bottom of the screen shows a decorative background image of a lake and trees.

```
class Fraction:
    def decimal_str(self, max_frac: int = 100) -> str:
        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.
```

```
a = {int} 190000
b = {int} 20000
max_frac = {int} 3
negative = {bool} False
self = {Fraction} <fraction_decimal_str_err.Fraction object at 0x7d37f0a0ed50>
a = {int} 18399
```

Debuggen in PyCharm



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

A screenshot of the PyCharm IDE interface. The top navigation bar shows 'Doctest Fraction.decimal_str' and various tool icons. The main area is divided into two panes: 'Project' on the left and 'fractions_decimal_str_err.py' on the right. The code editor shows a class 'Fraction' with a method 'decimal_str'. A specific line of code is highlighted: 'if len(digits) <= 1: # Do we only have an integer part?'. Below the code editor is a 'Threads & Variables' tab in the 'Debug' tool window. The variable 'digits' is expanded, showing its value as '[0, 9, 1, 10]'. Other variables listed include 'a', 'b', 'max_frac', 'negative', and 'self'. The bottom status bar shows the current file path as 'fractions_decimal_str_err.py:10'.



Reparierte Methode



Fraction: decimal_str

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



```
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
>>> 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.



```
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
>>> 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.

```
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
>>> 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 endet die Schleife trotzdem.

```
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
>>> 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 endet 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.

```
>>> 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 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 an 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 an 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.

```
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
>>> 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 an 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.

```
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
>>> 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.

```
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
>>> 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.

```
1     """A module with mathematics routines."""
2
3     from math import isclose # Checks if two float numbers are similar.
4
5
6     def factorial(a: int) -> int: # 1 `int` parameter and `int` result
7         """
8             Compute the factorial of a positive integer `a`.
9
10            :param a: the number to compute the factorial of
11            :return: the factorial of `a`, i.e., `a!`.
12        """
13
14        product: int = 1 # Initialize `product` as `1`.
15        for i in range(2, a + 1): # `i` goes from `2` to `a`.
16            product *= i # Multiply `i` to the product.
17
18        return product # Return the product, which now is the factorial.
19
20
21     def sqrt(number: float) -> float:
22         """
23             Compute the square root of a given `number`.
24
25            :param number: The number to compute the square root of.
26            :return: A value `v` such that `v * v` is approximately `number`.
27        """
28
29        guess: float = 1.0      # This will hold the current guess.
30        old_guess: float = 0.0 # 0.0 is just a dummy value != guess.
31        while not isclose(old_guess, guess): # Repeat until no change.
32            old_guess = guess # The current guess becomes the old guess.
33            guess = 0.5 * (guess + number / guess) # The new guess.
34
35        return guess
```

Fraction: sqrt

- So, nun können wir das machen, was wir eigentlich machen wollten:
- Wir wollten Quadratwurzeln mit wahnsinniger Genauigkeit berechnen!

```
1     """A module with mathematics routines."""
2
3     from math import isclose # Checks if two float numbers are similar.
4
5
6     def factorial(a: int) -> int: # 1 `int` parameter and `int` result
7         """
8             Compute the factorial of a positive integer `a`.
9
10            :param a: the number to compute the factorial of
11            :return: the factorial of `a`, i.e., `a!`.
12        """
13
14        product: int = 1 # Initialize `product` as `1`.
15        for i in range(2, a + 1): # `i` goes from `2` to `a`.
16            product *= i # Multiply `i` to the product.
17
18        return product # Return the product, which now is the factorial.
19
20
21     def sqrt(number: float) -> float:
22         """
23             Compute the square root of a given `number`.
24
25            :param number: The number to compute the square root of.
26            :return: A value `v` such that `v * v` is approximately `number`.
27        """
28
29        guess: float = 1.0      # This will hold the current guess.
30        old_guess: float = 0.0 # 0.0 is just a dummy value != guess.
31        while not isclose(old_guess, guess): # Repeat until no change.
32            old_guess = guess # The current guess becomes the old guess.
33            guess = 0.5 * (guess + number / guess) # The new guess.
34
35        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.

```
1     """A module with mathematics routines."""
2
3     from math import isclose # Checks if two float numbers are similar.
4
5
6     def factorial(a: int) -> int: # 1 `int` parameter and `int` result
7         """
8             Compute the factorial of a positive integer `a`.
9
10            :param a: the number to compute the factorial of
11            :return: the factorial of `a`, i.e., `a!`.
12        """
13
14        product: int = 1 # Initialize `product` as `1`.
15        for i in range(2, a + 1): # `i` goes from `2` to `a`.
16            product *= i # Multiply `i` to the product.
17
18        return product # Return the product, which now is the factorial.
19
20
21     def sqrt(number: float) -> float:
22         """
23             Compute the square root of a given `number`.
24
25            :param number: The number to compute the square root of.
26            :return: A value `v` such that `v * v` is approximately `number`.
27        """
28
29        guess: float = 1.0      # This will hold the current guess.
30        old_guess: float = 0.0 # 0.0 is just a dummy value != guess.
31        while not isclose(old_guess, guess): # Repeat until no change.
32            old_guess = guess # The current guess becomes the old guess.
33            guess = 0.5 * (guess + number / guess) # The new guess.
34
35        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.

```
1     """A square root algorithm based on fractions."""
2
3     from fraction import ONE, ONE_HALF, ZERO, Fraction
4
5
6     def sqrt(number: Fraction, max_steps: int = 10) -> Fraction:
7         """
8             Compute the square root of a given :class:`Fraction`.
9
10            :param number: The rational number to compute the square root of.
11            :param max_steps: the maximum number of steps, defaults to `10`.
12            :return: A value `v` such that `v * v` is approximately `number`.
13
14            >>> sqrt(Fraction(2, 1)).decimal_str(750)
15            '1.4142135623730950488016887242096980785696718753769480731766797379\
16            90732478462107038850387534327641572735013846230912297024924836055850737\
17            21264412149709993583141322266592750559275579995050115278206057147010955\
18            99716059702745345968620147285174186408891986095523292304843087143214508\
19            39762603627995251407989687253396546331808829640620615258352395054745750\
20            287755996172983557522033753185701135437460340849884716038689997069900481\
21            5030544027790316454247823068492936918621580578463115966687130130156185\
22            68987237235288509264861249497715421833420428568606014682472077143585487\
23            4155657069677653720226485447015888016207584749226572260020855844665214\
24            58398893944370926591800311388246468157082630100594858704003186480342194\
25            89727829064104507263688131373985525611732204025'
26            >>> sqrt(Fraction(4, 1)).decimal_str()
27            '2'
28            >>> (ONE_HALF * (ONE + sqrt(Fraction(5, 1)))).decimal_str(420)
29            '1.6180339887498948482045868343656381177203091798057628621354486227\
30            05260462818902449707207204189391137484754088075386891752126633862223536\
31            93179318006076672635443338908659593958290563832266131992829026788067520\
32            8766892501711696207032210432162695486262963136144381497587012203408058\
33            8795445474924618569536486449241044320771344947049565846788509874339442\
34            21254487706647809158846074998871240076521705751797883416625624940758907'
35            """
36            if number < ZERO: # No negative numbers are permitted.
37                raise ArithmeticError(f"Cannot compute sqrt({number}).")
38            guess: Fraction = ONE # This will hold the current guess.
39            old_guess: Fraction = ZERO # 0.0 is just a dummy value != guess.
40            while old_guess != guess: # Repeat until nothing changes anymore.
41                old_guess = guess # The current guess becomes the old guess.
42                guess = ONE_HALF * (guess + number / guess) # The new guess.
43                max_steps -= 1 # Reduce the number of remaining steps.
44                if max_steps <= 0: # If we have exhausted the maximum steps...
45                    break # ...then we stop (and return the guess).
46            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.

```
1     """A square root algorithm based on fractions."""
2
3     from fraction import ONE, ONE_HALF, ZERO, Fraction
4
5
6     def sqrt(number: Fraction, max_steps: int = 10) -> Fraction:
7         """
8             Compute the square root of a given :class:`Fraction`.
9
10            :param number: The rational number to compute the square root of.
11            :param max_steps: the maximum number of steps, defaults to `10`.
12            :return: A value `v` such that `v * v` is approximately `number`.
13
14            >>> sqrt(Fraction(2, 1)).decimal_str(750)
15            '1.4142135623730950488016887242096980785696718753769480731766797379\
16            90732478462107038850387534327641572735013846230912297024924836055850737\
17            21264412149709993583141322266592750559275579995050115278206057147010955\
18            99716059702745345968620147285174186408891986095523292304843087143214508\
19            39762603627995251407989687253396546331808829640620615258352395054745750\
20            287755996172983557522033753185701135437460340849884716038689997069900481\
21            5030544027790316454247823068492936918621580578463115966687130130156185\
22            6898723723528850926486124949715421833420428568606014682472077143585487\
23            41556570696776537202264854470158588016207584749226572260020855844665214\
24            58398893944370926581900311388246468157082630100594858704003186480342194\
25            89727829064104507263688131373985525611732204025'
26            >>> sqrt(Fraction(4, 1)).decimal_str()
27            '2'
28            >>> (ONE_HALF * (ONE + sqrt(Fraction(5, 1)))).decimal_str(420)
29            '1.6180339887498948482045868343656381177203091798057628621354486227\
30            05260462818902449707207204189391137484754088075386891752126633862223536\
31            93179318006076672635443338908659593958290563832266131992829026788067520\
32            8766892501711696207032210432162695486262963136144381497587012203408058\
33            87954454749246185695364864494241044320771344947049565846788509874339442\
34            21254487706647809158846074998871240076521705751797883416625624940758907'
35            """
36            if number < ZERO: # No negative numbers are permitted.
37                raise ArithmeticError(f"Cannot compute sqrt({number}).")
38
39            guess: Fraction = ONE # This will hold the current guess.
40            old_guess: Fraction = ZERO # 0.0 is just a dummy value != guess.
41            while old_guess != guess: # Repeat until nothing changes anymore.
42                old_guess = guess # The current guess becomes the old guess.
43                guess = ONE_HALF * (guess + number / guess) # The new guess.
44                max_steps -= 1 # Reduce the number of remaining steps.
45                if max_steps <= 0: # If we have exhausted the maximum steps...
46                    break # ...then we stop (and return the guess).
47
48            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`.

```
1     """A square root algorithm based on fractions."""
2
3     from fraction import ONE, ONE_HALF, ZERO, Fraction
4
5
6     def sqrt(number: Fraction, max_steps: int = 10) -> Fraction:
7         """
8             Compute the square root of a given :class:`Fraction`.
9
10            :param number: The rational number to compute the square root of.
11            :param max_steps: the maximum number of steps, defaults to `10`.
12            :return: A value `v` such that `v * v` is approximately `number`.
13
14            >>> sqrt(Fraction(2, 1)).decimal_str(750)
15            '1.4142135623730950488016887242096980785696718753769480731766797379\
16            90732478462107038850387534327641572735013846230912297024924836055850737\
17            21264412149709993583141322266592750559275579995050115278206057147010955\
18            99716059702745345968620147285174186408891986095523292304843087143214508\
19            39762603627995251407989687253396546331808829640620615258352395054745750\
20            28775599617298355752203375318570113543746034084988471603868999706990481\
21            5030544027790316454247823068492936918621580578463115966687130130156185\
22            6898723723528850926486124949771542183342042856860614682472077143585487\
23            4155657069677653720226485447015888016207584749226572260020855844665214\
24            5839893944370926591800311388246468157082630100594858704003186480342194\
25            89727829064104507263688131373985525611732204025'
26            >>> sqrt(Fraction(4, 1)).decimal_str()
27            '2'
28            >>> (ONE_HALF * (ONE + sqrt(Fraction(5, 1)))).decimal_str(420)
29            '1.6180339887498948482045868343656381177203091798057628621354486227\
30            05260462818902449707207204189391137484754088075386891752126633862223536\
31            93179318006076672635443338908659593958290563832266131992829026788067520\
32            6766892501711696207032210432162695486262963136144381497587012203408058\
33            8795445474924618569536486449241044320771344947049565846788509874339442\
34            21254487706647809158846074998871240076521705751797883416625624940758907\
35            """
36            if number < ZERO: # No negative numbers are permitted.
37                raise ArithmeticError(f"Cannot compute sqrt({number}).")
38            guess: Fraction = ONE # This will hold the current guess.
39            old_guess: Fraction = ZERO # 0.0 is just a dummy value != guess.
40            while old_guess != guess: # Repeat until nothing changes anymore.
41                old_guess = guess # The current guess becomes the old guess.
42                guess = ONE_HALF * (guess + number / guess) # The new guess.
43                max_steps -= 1 # Reduce the number of remaining steps.
44                if max_steps <= 0: # If we have exhausted the maximum steps...
45                    break # ...then we stop (and return the guess).
46            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.

```
1     """A square root algorithm based on fractions."""
2
3     from fraction import ONE, ONE_HALF, ZERO, Fraction
4
5
6     def sqrt(number: Fraction, max_steps: int = 10) -> Fraction:
7         """
8             Compute the square root of a given :class:`Fraction`.
9
10            :param number: The rational number to compute the square root of.
11            :param max_steps: the maximum number of steps, defaults to `10`.
12            :return: A value `v` such that `v * v` is approximately `number`.
13
14            >>> sqrt(Fraction(2, 1)).decimal_str(750)
15            '1.4142135623730950488016887242096980785696718753769480731766797379\
16            90732478462107038850387534327641572735013846230912297024924836055850737\
17            21264412149709993583141322266592750559275579995050115278206057147010955\
18            99716059702745345968620147285174186408891986095523292304843087143214508\
19            39762603627995251407989687253396546331808829640620615258352395054745750\
20            287755996172983557522033753185701135437460340849884716038689997069900481\
21            5030544027790316454247823068492936918621580578463115966687130130156185\
22            68987237235288509264861249497715421833420428568606014682472077143585487\
23            41556570696776537202264854470158588016207584749226572260020855844665214\
24            5839893944370926591800311388246468157082630100594858704003186480342194\
25            89727829064104507263688131373985525611732204025'
26            >>> sqrt(Fraction(4, 1)).decimal_str()
27            '2'
28            >>> (ONE_HALF * (ONE + sqrt(Fraction(5, 1)))).decimal_str(420)
29            '1.6180339887498948482045868343656381177203091798057628621354486227\
30            05260462818902449707207204189391137484754088075386891752126633862223536\
31            93179318006076672635443338908659593958290563832266131992829026788067520\
32            8766892501711696207032210432162695486262963136144381497587012203408058\
33            879544547492461869536486449241044320771344947049565846788509874339442\
34            21254487706647809158846074998871240076521705751797883416625624940758907\
35            """
36            if number < ZERO: # No negative numbers are permitted.
37                raise ArithmeticError(f"Cannot compute sqrt({number}).")
38            guess: Fraction = ONE # This will hold the current guess.
39            old_guess: Fraction = ZERO # 0.0 is just a dummy value != guess.
40            while old_guess != guess: # Repeat until nothing changes anymore.
41                old_guess = guess # The current guess becomes the old guess.
42                guess = ONE_HALF * (guess + number / guess) # The new guess.
43                max_steps -= 1 # Reduce the number of remaining steps.
44                if max_steps <= 0: # If we have exhausted the maximum steps...
45                    break # ...then we stop (and return the guess).
46            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.

```
1     """A square root algorithm based on fractions."""
2
3     from fraction import ONE, ONE_HALF, ZERO, Fraction
4
5
6     def sqrt(number: Fraction, max_steps: int = 10) -> Fraction:
7         """
8             Compute the square root of a given :class:`Fraction`.
9
10            :param number: The rational number to compute the square root of.
11            :param max_steps: the maximum number of steps, defaults to `10`.
12            :return: A value `v` such that `v * v` is approximately `number`.
13
14            >>> sqrt(Fraction(2, 1)).decimal_str(750)
15            '1.4142135623730950488016887242096980785696718753769480731766797379\
16            90732478462107038850387534327641572735013846230912297024924836055850737\
17            21264412149709993583141322266592750559275579995050115278206057147010955\
18            99716059702745345968620147285174186408891986095523292304843087143214508\
19            39762603627995251407989687253396546331808829640620615258352395054745750\
20            287755996172983557522033753185701135437460340849884716038689997069900481\
21            50305440277903164542478230684929369186215805784631115966687130130156185\
22            68987237235288509264861249497715421833420428568606014682472077143585487\
23            41556570696776537202264854470158588016207584749226572260020855844665214\
24            5839893944370926591800311388246468157082630100594858704003186480342194\
25            89727829064104507263688131373985525611732204025'
26            >>> sqrt(Fraction(4, 1)).decimal_str()
27            '2'
28            >>> (ONE_HALF * (ONE + sqrt(Fraction(5, 1)))).decimal_str(420)
29            '1.6180339887498948482045868343656381177203091798057628621354486227\
30            05260462818902449707207204189391137484754088075386891752126633862223536\
31            93179318006076672635443338908659593958290563832266131992829026788067520\
32            6766892501711696207032210432162695486262963136144381497587012203408058\
33            879544547492461869536486449241044320771344947049565846788509874339442\
34            21254487706647809158846074998871240076521705751797883416625624940758907\
35            """
36            if number < ZERO: # No negative numbers are permitted.
37                raise ArithmeticError(f"Cannot compute sqrt({number}).")
38
39            guess: Fraction = ONE # This will hold the current guess.
40            old_guess: Fraction = ZERO # 0.0 is just a dummy value != guess.
41            while old_guess != guess: # Repeat until nothing changes anymore.
42                old_guess = guess # The current guess becomes the old guess.
43                guess = ONE_HALF * (guess + number / guess) # The new guess.
44                max_steps -= 1 # Reduce the number of remaining steps.
45                if max_steps <= 0: # If we have exhausted the maximum steps...
46                    break # ...then we stop (and return the guess).
47
48            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.

```
1     """A square root algorithm based on fractions."""
2
3     from fraction import ONE, ONE_HALF, ZERO, Fraction
4
5
6     def sqrt(number: Fraction, max_steps: int = 10) -> Fraction:
7         """
8             Compute the square root of a given :class:`Fraction`.
9
10            :param number: The rational number to compute the square root of.
11            :param max_steps: the maximum number of steps, defaults to `10`.
12            :return: A value `v` such that `v * v` is approximately `number`.
13
14            >>> sqrt(Fraction(2, 1)).decimal_str(750)
15            '1.4142135623730950488016887242096980785696718753769480731766797379\
16            90732478462107038850387534327641572735013846230912297024924836055850737\
17            21264412149709993583141322266592750559275579995050115278206057147010955\
18            99716059702745345968620147285174186408891986095523292304843087143214508\
19            39762603627995251407989687253396546331808829640620615258352395054745750\
20            287755996172983557522033753185701135437460340849884716038689997069900481\
21            5030544027790316454247823068492936918621580578463115966687130130156185\
22            6898723723528850926486124949771542183342042856860614682472077143585487\
23            41556570696776537202264854470158588016207584749226572260020855844665214\
24            5839893944370926591800311388246468157082630100594858704003186480342194\
25            89727829064104507263688131373985525611732204025'
26            >>> sqrt(Fraction(4, 1)).decimal_str()
27            '2'
28            >>> (ONE_HALF * (ONE + sqrt(Fraction(5, 1)))).decimal_str(420)
29            '1.6180339887498948482045868343656381177203091798057628621354486227\
30            05260462818902449707207204189391137484754088075386891752126633862223536\
31            93179318006076672635443338908659593958290563832266131992829026788067520\
32            6766892501711696207032210432162695486262963136144381497587012203408058\
33            8795445474924618569536486449241044320771344947049565846788509874339442\
34            21254487706647809158846074998871240076521705751797883416625624940758907\
35            """
36            if number < ZERO: # No negative numbers are permitted.
37                raise ArithmeticError(f"Cannot compute sqrt({number}).")
38
39            guess: Fraction = ONE # This will hold the current guess.
40            old_guess: Fraction = ZERO # 0.0 is just a dummy value != guess.
41            while old_guess != guess: # Repeat until nothing changes anymore.
42                old_guess = guess # The current guess becomes the old guess.
43                guess = ONE_HALF * (guess + number / guess) # The new guess.
44                max_steps -= 1 # Reduce the number of remaining steps.
45                if max_steps <= 0: # If we have exhausted the maximum steps...
46                    break # ...then we stop (and return the guess).
47
48            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.

```
"""
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)
    '1.4142135623730950488016887242096980785696718753769480731766797379\
    90732478462107038850387534327641572735013846230912297024924836055850737\
    2126441214970999358314132226659275055927557999505011527820657147010955\
    99716059702745345968620147285174186408891986095523292304843087143214508\
    39762603627995251407989687253396546331808829640620615258352395054745750\
    28775599617298355752203375318570113543746034084988471603868999706990481\
    5030544027790316454247823068492936918621580578463115966687130130156185\
    68987237235288509264861249497715421833420428568606014682472077143585487\
    4155657069677653720226485447015888016207584749226572260020855844665214\
    58398893944370926591800311388246486157082630100594858704003186480342194\
    89727829064104507263688131373985525611732204025'
    >>> sqrt(Fraction(4, 1)).decimal_str()
    '2'
    >>> (ONE_HALF * (ONE + sqrt(Fraction(5, 1)))).decimal_str(420)
    '1.6180339887498948482045868343656381177203091798057628621354486227\
    05260462818902449707207204189391137484754088075386891752126633862223536\
    93179318006076672635443338908659593958290563832266131992829026788067520\
    8766892501711696207032210432162695486262963136144381497587012203408058\
    8795445474924618695364864494241044320771344947049565846788509874339442\
    21254487706647809158846074998871240076521705751797883416625624940758907'
    """

    if number < ZERO: # No negative numbers are permitted.
        raise ArithmeticError(f"Cannot compute 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.

```
"""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)
    '1.4142135623730950488016887242096980785696718753769480731766797379\
    90732478462107038850387534327641572735013846230912297024924836055850737\
    21264412149709993583141322266592750559275579995050115278206057147010955\
    99716059702745345968620147285174186408891986095523292304843087143214508\
    39762603627995251407989687253396546331808829640620615258352395054745750\
    287755996172983557522033753185701135437460340849884716038689997069900481\
    50305440277903164542478230684929369186215805784631115966687130130156185\
    6898723723528850926486124949771542183342042856860614682472077143585487\
    41556570696776537202264854470158588016207584749226572260020855844665214\
    5839893944370926591800311388246468157082630100594858704003186480342194\
    89727829064104507263688131373985525611732204025'
    >>> sqrt(Fraction(4, 1)).decimal_str()
    '2'
    >>> (ONE_HALF * (ONE + sqrt(Fraction(5, 1)))).decimal_str(420)
    '1.6180339887498948482045868343656381177203091798057628621354486227\
    05260462818902449707207204189391137484754088075386891752126633862223536\
    93179318006076672635443338908659593958290563832266131992829026788067520\
    8766892501711696207032210432162695486262963136144381497587012203408058\
    8795445474924618569536486449241044320771344947049565846788509874339442\
    21254487706647809158846074998871240076521705751797883416625624940758907'
    """
    if number < ZERO: # No negative numbers are permitted.
        raise ArithmeticError(f"Cannot compute 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.

```
1     """A square root algorithm based on fractions."""
2
3     from fraction import ONE, ONE_HALF, ZERO, Fraction
4
5
6     def sqrt(number: Fraction, max_steps: int = 10) -> Fraction:
7         """
8             Compute the square root of a given :class:`Fraction`.
9
10            :param number: The rational number to compute the square root of.
11            :param max_steps: the maximum number of steps, defaults to `10`.
12            :return: A value `v` such that `v * v` is approximately `number`.
13
14            >>> sqrt(Fraction(2, 1)).decimal_str(750)
15            '1.4142135623730950488016887242096980785696718753769480731766797379\
16            90732478462107038850387534327641572735013846230912297024924836055850737\
17            21264412149709993583141322266592750559275579995050115278206057147010955\
18            99716059702745345968620147285174186408891986095523292304843087143214508\
19            39762603627995251407989687253396546331808829640620615258352395054745750\
20            287755996172983557522033753185701135437460340849884716038689997069900481\
21            5030544027790316454247823068492936918621580578463115966687130130156185\
22            68987237235288509264861249497715421833420428568606014682472077143585487\
23            4155657069677653720226485447015886016207584749226572260020855844665214\
24            58398893944370926591800311388246468157082630100594858704003186480342194\
25            89727829064104507263688131373985525611732204025'
26            >>> sqrt(Fraction(4, 1)).decimal_str()
27            '2'
28            >>> (ONE_HALF * (ONE + sqrt(Fraction(5, 1)))).decimal_str(420)
29            '1.6180339887498948482045868343656381177203091798057628621354486227\
30            05260462818902449707207204189391137484754088075386891752126633862223536\
31            93179318006076672635443338908659593958290563832266131992829026788067520\
32            6766892501711696207032210432162695486262963136144381497587012203408058\
33            8795445474924618569536486449241044320771344947049565846788509874339442\
34            21254487706647809158846074998871240076521705751797883416625624940758907\
35            '''
36            if number < ZERO: # No negative numbers are permitted.
37                raise ArithmeticError(f"Cannot compute sqrt({number}).")
38            guess: Fraction = ONE # This will hold the current guess.
39            old_guess: Fraction = ZERO # 0.0 is just a dummy value != guess.
40            while old_guess != guess: # Repeat until nothing changes anymore.
41                old_guess = guess # The current guess becomes the old guess.
42                guess = ONE_HALF * (guess + number / guess) # The new guess.
43                max_steps -= 1 # Reduce the number of remaining steps.
44                if max_steps <= 0: # If we have exhausted the maximum steps...
45                    break # ...then we stop (and return the guess).
46            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.

```
1     """A square root algorithm based on fractions."""
2
3     from fraction import ONE, ONE_HALF, ZERO, Fraction
4
5
6     def sqrt(number: Fraction, max_steps: int = 10) -> Fraction:
7         """
8             Compute the square root of a given :class:`Fraction`.
9
10            :param number: The rational number to compute the square root of.
11            :param max_steps: the maximum number of steps, defaults to `10`.
12            :return: A value `v` such that `v * v` is approximately `number`.
13
14            >>> sqrt(Fraction(2, 1)).decimal_str(750)
15            '1.4142135623730950488016887242096980785696718753769480731766797379\
16            90732478462107038850387534327641572735013846230912297024924836055850737\
17            21264412149709993583141322266592750559275579995050115278206057147010955\
18            99716059702745345968620147285174186408891986095523292304843087143214508\
19            39762603627995251407989687253396546331808829640620615258352395054745750\
20            287755996172983557522033753185701135437460340849884716038689997069900481\
21            5030544027790316454247823068492936918621580578463115966687130130156185\
22            6898723723528850926486124949715421833420428568606014682472077143585487\
23            415565706967765372022648544701585880616207584749226572260020855844665214\
24            5839893944370926591800311388246468157082630100594858704003186480342194\
25            89727829064104507263688131373985525611732204025'
26            >>> sqrt(Fraction(4, 1)).decimal_str()
27            '2'
28            >>> (ONE_HALF * (ONE + sqrt(Fraction(5, 1)))).decimal_str(420)
29            '1.6180339887498948482045868343656381177203091798057628621354486227\
30            05260462818902449707207204189391137484754088075386891752126633862223536\
31            93179318006076672635443338908659593958290563832266131992829026788067520\
32            7668982501711696207032210432162695486262963136144381497587012203408058\
33            8795445474924618569536486449241044320771344947049565846788509874339442\
34            21254487706647809158846074998871240076521705751797883416625624940758907\
35            '''
36            if number < ZERO: # No negative numbers are permitted.
37                raise ArithmeticError(f"Cannot compute sqrt({number}).")
38
39            guess: Fraction = ONE # This will hold the current guess.
40            old_guess: Fraction = ZERO # 0.0 is just a dummy value != guess.
41            while old_guess != guess: # Repeat until nothing changes anymore.
42                old_guess = guess # The current guess becomes the old guess.
43                guess = ONE_HALF * (guess + number / guess) # The new guess.
44                max_steps -= 1 # Reduce the number of remaining steps.
45                if max_steps <= 0: # If we have exhausted the maximum steps...
46                    break # ...then we stop (and return the guess).
47
48            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)
    '1.4142135623730950488016887242096980785696718753769480731766797379\
    90732478462107038850387534327641572735013846230912297024924836055850737\
    21264412149709993583141322266592750559275579995050115278206057147010955\
    99716059702745345968620147285174186408891986095523292304843087143214508\
    39762603627995251407989687253396546331808829640620615258352395054745750\
    287755996172983557522033753185701135437460340849884716038689997069900481\
    5030544027790316454247823068492936918621580578463115966687130130156185\
    6898723723528850926486124949771542183342042856860614682472077143585487\
    41556570696776537202264854470158588016207584749226572260020855844665214\
    5839893944370926591800311388246468157082630100594858704003186480342194\
    89727829064104507263688131373985525611732204025'
    >>> sqrt(Fraction(4, 1)).decimal_str()
    '2'
    >>> (ONE_HALF * (ONE + sqrt(Fraction(5, 1)))).decimal_str(420)
    '1.6180339887498948482045868343656381177203091798057628621354486227\
    05260462818902449707207204189391137484754088075386891752126633862223536\
    93179318006076672635443338908659593958290563832266131992829026788067520\
    3276689250171169620703221043216295486262963136144381497587012203408058\
    8795445474924618569536486449241044320771344947049565846788509874339442\
    21254487706647809158846074998871240076521705751797883416625624940758907'
    """

    if number < ZERO: # No negative numbers are permitted.
        raise ArithmeticError(f"Cannot compute 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.

```
1     """A square root algorithm based on fractions."""
2
3     from fraction import ONE, ONE_HALF, ZERO, Fraction
4
5
6     def sqrt(number: Fraction, max_steps: int = 10) -> Fraction:
7         """
8             Compute the square root of a given :class:`Fraction`.
9
10            :param number: The rational number to compute the square root of.
11            :param max_steps: the maximum number of steps, defaults to `10`.
12            :return: A value `v` such that `v * v` is approximately `number`.
13
14            >>> sqrt(Fraction(2, 1)).decimal_str(750)
15            '1.4142135623730950488016887242096980785696718753769480731766797379\
16            90732478462107038850387534327641572735013846230912297024924836055850737\
17            21264412149709993583141322266592750559275579995050115278206057147010955\
18            99716059702745345968620147285174186408891986095523292304843087143214508\
19            39762603627995251407989687253396546331808829640620615258352395054745750\
20            287755996172983557522033753185701135437460340849884716038689997069900481\
21            5030544027790316454247823068492936918621580578463115966687130130156185\
22            6898723723528850926486124949715421833420428568606014682472077143585487\
23            41556570696776537202264854470158588016207584749226572260020855844665214\
24            5839893944370926591800311388246468157082630100594858704003186480342194\
25            89727829064104507263688131373985525611732204025'
26            >>> sqrt(Fraction(4, 1)).decimal_str()
27            '2'
28            >>> (ONE_HALF * (ONE + sqrt(Fraction(5, 1)))).decimal_str(420)
29            '1.6180339887498948482045868343656381177203091798057628621354486227\
30            05260462818902449707207204189391137484754088075386891752126633862223536\
31            93179318006076672635443338908659593958290563832266131992829026788067520\
32            8766892501711696207032210432162695486262963136144381497587012203408058\
33            87954454749246185695364864494241044320771344947049565846788509874339442\
34            21254487706647809158846074998871240076521705751797883416625624940758907'
35            """
36            if number < ZERO: # No negative numbers are permitted.
37                raise ArithmeticError(f"Cannot compute sqrt({number}).")
38            guess: Fraction = ONE # This will hold the current guess.
39            old_guess: Fraction = ZERO # 0.0 is just a dummy value != guess.
40            while old_guess != guess: # Repeat until nothing changes anymore.
41                old_guess = guess # The current guess becomes the old guess.
42                guess = ONE_HALF * (guess + number / guess) # The new guess.
43                max_steps -= 1 # Reduce the number of remaining steps.
44                if max_steps <= 0: # If we have exhausted the maximum steps...
45                    break # ...then we stop (and return the guess).
46            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.

```
1     """A square root algorithm based on fractions."""
2
3     from fraction import ONE, ONE_HALF, ZERO, Fraction
4
5
6     def sqrt(number: Fraction, max_steps: int = 10) -> Fraction:
7         """
8             Compute the square root of a given :class:`Fraction`.
9
10            :param number: The rational number to compute the square root of.
11            :param max_steps: the maximum number of steps, defaults to `10`.
12            :return: A value `v` such that `v * v` is approximately `number`.
13
14            >>> sqrt(Fraction(2, 1)).decimal_str(750)
15            '1.4142135623730950488016887242096980785696718753769480731766797379\
16            90732478462107038850387534327641572735013846230912297024924836055850737\
17            21264412149709993583141322266592750559275579995050115278206057147010955\
18            99716059702745345968620147285174186408891986095523292304843087143214508\
19            39762603627995251407989687253396546331808829640620615258352395054745750\
20            28775599617298355752203375318570113543746034084988471603868999706990481\
21            50305440277903164542478230684929369186215805784631115966687130130156185\
22            68987237235288509264861249497715421833420428568606014682472077143585487\
23            41556570696776537202264854470158588016207584749226572260020855844665214\
24            5839893944370926591800311388246468157082630100594858704003186480342194\
25            89727829064104507263688131373985525611732204025'
26            >>> sqrt(Fraction(4, 1)).decimal_str()
27            '2'
28            >>> (ONE_HALF * (ONE + sqrt(Fraction(5, 1)))).decimal_str(420)
29            '1.6180339887498948482045868343656381177203091798057628621354486227\
30            05260462818902449707207204189391137484754088075386891752126633862223536\
31            93179318006076672635443338908659593958290563832266131992829026788067520\
32            8766892501711696207032210432162695486262963136144381497587012203408058\
33            87954454749246185695364864494241044320771344947049565846788509874339442\
34            21254487706647809158846074998871240076521705751797883416625624940758907\
35            """
36            if number < ZERO: # No negative numbers are permitted.
37                raise ArithmeticError(f"Cannot compute sqrt({number}).")
38
39            guess: Fraction = ONE # This will hold the current guess.
40            old_guess: Fraction = ZERO # 0.0 is just a dummy value != guess.
41            while old_guess != guess: # Repeat until nothing changes anymore.
42                old_guess = guess # The current guess becomes the old guess.
43                guess = ONE_HALF * (guess + number / guess) # The new guess.
44                max_steps -= 1 # Reduce the number of remaining steps.
45                if max_steps <= 0: # If we have exhausted the maximum steps...
46                    break # ...then we stop (and return the guess).
47
48            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.

```
1     """A square root algorithm based on fractions."""
2
3     from fraction import ONE, ONE_HALF, ZERO, Fraction
4
5
6     def sqrt(number: Fraction, max_steps: int = 10) -> Fraction:
7         """
8             Compute the square root of a given :class:`Fraction`.
9
10            :param number: The rational number to compute the square root of.
11            :param max_steps: the maximum number of steps, defaults to `10`.
12            :return: A value `v` such that `v * v` is approximately `number`.
13
14            >>> sqrt(Fraction(2, 1)).decimal_str(750)
15            '1.4142135623730950488016887242096980785696718753769480731766797379\
16            90732478462107038850387534327641572735013846230912297024924836055850737\
17            21264412149709993583141322266592750559275579995050115278206057147010955\
18            99716059702745345968620147285174186408891986095523292304843087143214508\
19            39762603627995251407989687253396546331808829640620615258352395054745750\
20            287755996172983557522033753185701135437460340849884716038689997069900481\
21            5030544027790316454247823068492936918621580578463115966687130130156185\
22            6898723723528850926486124949771542183342042856860614682472077143585487\
23            41556570696776537202264854470158588016207584749226572260020855844665214\
24            5839893944370926591800311388246468157082630100594858704003186480342194\
25            89727829064104507263688131373985525611732204025'
26            >>> sqrt(Fraction(4, 1)).decimal_str()
27            '2'
28            >>> (ONE_HALF * (ONE + sqrt(Fraction(5, 1)))).decimal_str(420)
29            '1.6180339887498948482045868343656381177203091798057628621354486227\
30            05260462818902449707207204189391137484754088075386891752126633862223536\
31            93179318006076672635443338908659593958290563832266131992829026788067520\
32            8766892501711696207032210432162695486262963136144381497587012203408058\
33            879544547492461869536486449241044320771344947049565846788509874339442\
34            21254487706647809158846074998871240076521705751797883416625624940758907\
35            '''
36            if number < ZERO: # No negative numbers are permitted.
37                raise ArithmeticError(f"Cannot compute sqrt({number}).")
38            guess: Fraction = ONE # This will hold the current guess.
39            old_guess: Fraction = ZERO # 0.0 is just a dummy value != guess.
40            while old_guess != guess: # Repeat until nothing changes anymore.
41                old_guess = guess # The current guess becomes the old guess.
42                guess = ONE_HALF * (guess + number / guess) # The new guess.
43                max_steps -= 1 # Reduce the number of remaining steps.
44                if max_steps <= 0: # If we have exhausted the maximum steps...
45                    break # ...then we stop (and return the guess).
46            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`.

```
"""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)
    '1.4142135623730950488016887242096980785696718753769480731766797379\
    90732478462107038850387534327641572735013846230912297024924836055850737\
    21264412149709993583141322266592750559275579995050115278206057147010955\
    99716059702745345968620147285174186408891986095523292304843087143214508\
    39762603627995251407989687253396546331808829640620615258352395054745750\
    28775599617298355752203375318570113543746034084988471603868999706990481\
    5030544027790316454247823068492936918621580578463115966687130130156185\
    68987237235288509264861249497715421833420428568606014682472077143585487\
    41556570696776537202264854470158588016207584749226572260020855844665214\
    5839893944370926591800311388246468157082630100594858704003186480342194\
    89727829064104507263688131373985525611732204025'
    >>> sqrt(Fraction(4, 1)).decimal_str()
    '2'
    >>> (ONE_HALF * (ONE + sqrt(Fraction(5, 1)))).decimal_str(420)
    '1.6180339887498948482045868343656381177203091798057628621354486227\
    05260462818902449707207204189391137484754088075386891752126633862223536\
    93179318006076672635443338908659593958290563832266131992829026788067520\
    82766892501711696207032210432162695486262963136144381497587012203408058\
    8795445474924618569536486449241044320771344947049565846788509874339442\
    21254487706647809158846074998871240076521705751797883416625624940758907'
    """

    if number < ZERO: # No negative numbers are permitted.
        raise ArithmeticError(f"Cannot compute 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`.

```
1     """A square root algorithm based on fractions."""
2
3     from fraction import ONE, ONE_HALF, ZERO, Fraction
4
5
6     def sqrt(number: Fraction, max_steps: int = 10) -> Fraction:
7         """
8             Compute the square root of a given :class:`Fraction`.
9
10            :param number: The rational number to compute the square root of.
11            :param max_steps: the maximum number of steps, defaults to `10`.
12            :return: A value `v` such that `v * v` is approximately `number`.
13
14            >>> sqrt(Fraction(2, 1)).decimal_str(750)
15            '1.4142135623730950488016887242096980785696718753769480731766797379\
16            90732478462107038850387534327641572735013846230912297024924836055850737\
17            21264412149709993583141322266592750559275579995050115278206057147010955\
18            99716059702745345968620147285174186408891986095523292304843087143214508\
19            39762603627995251407989687253396546331808829640620615258352395054745750\
20            28775599617298355752203375318570113543746034084988471603868999706990481\
21            5030544027790316454247823068492936918621580578463115966687130130156185\
22            6898723723528850926486124949771542183342042856860614682472077143585487\
23            41556570696776537202264854470158588016207584749226572260020855844665214\
24            5839893944370926591800311388246468157082630100594858704003186480342194\
25            89727829064104507263688131373985525611732204025'
26            >>> sqrt(Fraction(4, 1)).decimal_str()
27            '2'
28            >>> (ONE_HALF * (ONE + sqrt(Fraction(5, 1)))).decimal_str(420)
29            '1.6180339887498948482045868343656381177203091798057628621354486227\
30            05260462818902449707207204189391137484754088075386891752126633862223536\
31            93179318006076672635443338908659593958290563832266131992829026788067520\
32            8766892501711696207032210432162695486262963136144381497587012203408058\
33            8795445474924618569536486449241044320771344947049565846788509874339442\
34            21254487706647809158846074998871240076521705751797883416625624940758907\
35            """
36            if number < ZERO: # No negative numbers are permitted.
37                raise ArithmeticError(f"Cannot compute sqrt({number}).")
38            guess: Fraction = ONE # This will hold the current guess.
39            old_guess: Fraction = ZERO # 0.0 is just a dummy value != guess.
40            while old_guess != guess: # Repeat until nothing changes anymore.
41                old_guess = guess # The current guess becomes the old guess.
42                guess = ONE_HALF * (guess + number / guess) # The new guess.
43                max_steps -= 1 # Reduce the number of remaining steps.
44                if max_steps <= 0: # If we have exhausted the maximum steps...
45                    break # ...then we stop (and return the guess).
46            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...

```
1     """A square root algorithm based on fractions."""
2
3     from fraction import ONE, ONE_HALF, ZERO, Fraction
4
5
6     def sqrt(number: Fraction, max_steps: int = 10) -> Fraction:
7         """
8             Compute the square root of a given :class:`Fraction`.
9
10            :param number: The rational number to compute the square root of.
11            :param max_steps: the maximum number of steps, defaults to `10`.
12            :return: A value `v` such that `v * v` is approximately `number`.
13
14            >>> sqrt(Fraction(2, 1)).decimal_str(750)
15            '1.4142135623730950488016887242096980785696718753769480731766797379\
16            90732478462107038850387534327641572735013846230912297024924836055850737\
17            21264412149709993583141322266592750559275579995050115278206057147010955\
18            99716059702745345968620147285174186408891986095523292304843087143214508\
19            39762603627995251407989687253396546331808829640620615258352395054745750\
20            287755996172983557522033753185701135437460340849884716038689997069900481\
21            5030544027790316454247823068492936918621580578463115966687130130156185\
22            6898723723528850926486124949771542183342042856860614682472077143585487\
23            41556570696776537202264854470158588016207584749226572260020855844665214\
24            5839893944370926591800311388246468157082630100594858704003186480342194\
25            89727829064104507263688131373985525611732204025'
26            >>> sqrt(Fraction(4, 1)).decimal_str()
27            '2'
28            >>> (ONE_HALF * (ONE + sqrt(Fraction(5, 1)))).decimal_str(420)
29            '1.6180339887498948482045868343656381177203091798057628621354486227\
30            05260462818902449707207204189391137484754088075386891752126633862223536\
31            93179318006076672635443338908659593958290563832266131992829026788067520\
32            876689250171169620703221043216295486262963136144381497587012203408058\
33            8795445474924618695364864494241044320771344947049565846788509874339442\
34            21254487706647809158846074998871240076521705751797883416625624940758907'
35            """
36            if number < ZERO: # No negative numbers are permitted.
37                raise ArithmeticError(f"Cannot compute sqrt({number}).")
38
39            guess: Fraction = ONE # This will hold the current guess.
40            old_guess: Fraction = ZERO # 0.0 is just a dummy value != guess.
41            while old_guess != guess: # Repeat until nothing changes anymore.
42                old_guess = guess # The current guess becomes the old guess.
43                guess = ONE_HALF * (guess + number / guess) # The new guess.
44                max_steps -= 1 # Reduce the number of remaining steps.
45                if max_steps <= 0: # If we have exhausted the maximum steps...
46                    break # ...then we stop (and return the guess).
47
48            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)
    '1.4142135623730950488016887242096980785696718753769480731766797379\
90732478462107038850387534327641572735013846230912297024924836055850737\
21264412149709993583141322266592750559275579995050115278206057147010955\
99716059702745345968620147285174186408891986095523292304843087143214508\
39762603627995251407989687253396546331808829640620615258352395054745750\
287755996172983557522033753185701135437460340849884716038689997069900481\
5030544027790316454247823068492936918621580578463115966687130130156185\
68987237235288509264861249497715421833420428568606014682472077143585487\
4155657069677653720226485447015888016207584749226572260020855844665214\
5839893944370926591800311388246468157082630100594858704003186480342194\
89727829064104507263688131373985525611732204025'
    >>> sqrt(Fraction(4, 1)).decimal_str()
    '2'
    >>> (ONE_HALF * (ONE + sqrt(Fraction(5, 1)))).decimal_str(420)
    '1.6180339887498948482045868343656381177203091798057628621354486227\
05260462818902449707207204189391137484754088075386891752126633862223536\
93179318006076672635443338908659593958290563832266131992829026788067520\
8766892501711696207032210432162695486262963136144381497587012203408058\
8795445474924618569536486449241044320771344947049565846788509874339442\
21254487706647809158846074998871240076521705751797883416625624940758907'
    """
    if number < ZERO: # No negative numbers are permitted.
        raise ArithmeticError(f"Cannot compute 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`.

```
1     """A square root algorithm based on fractions."""
2
3     from fraction import ONE, ONE_HALF, ZERO, Fraction
4
5
6     def sqrt(number: Fraction, max_steps: int = 10) -> Fraction:
7         """
8             Compute the square root of a given :class:`Fraction`.
9
10            :param number: The rational number to compute the square root of.
11            :param max_steps: the maximum number of steps, defaults to `10`.
12            :return: A value `v` such that `v * v` is approximately `number`.
13
14            >>> sqrt(Fraction(2, 1)).decimal_str(750)
15            '1.4142135623730950488016887242096980785696718753769480731766797379\
16            90732478462107038850387534327641572735013846230912297024924836055850737\
17            21264412149709993583141322266592750559275579995050115278206057147010955\
18            99716059702745345968620147285174186408891986095523292304843087143214508\
19            39762603627995251407989687253396546331808829640620615258352395054745750\
20            28775599617298355752203375318570113543746034084988471603868999706990481\
21            50305440277903164542478230684929369186215805784631115966687130130156185\
22            6898723723528850926486124949771542183342042856860614682472077143585487\
23            41556570696776537202264854470158588016207584749226572260020855844665214\
24            5839893944370926591800311388246468157082630100594858704003186480342194\
25            89727829064104507263688131373985525611732204025'
26            >>> sqrt(Fraction(4, 1)).decimal_str()
27            '2'
28            >>> (ONE_HALF * (ONE + sqrt(Fraction(5, 1)))).decimal_str(420)
29            '1.6180339887498948482045868343656381177203091798057628621354486227\
30            05260462818902449707207204189391137484754088075386891752126633862223536\
31            93179318006076672635443338908659593958290563832266131992829026788067520\
32            766892501711696207032210432162695486262963136144381497587012203408058\
33            8795445474924618569536486449241044320771344947049565846788509874339442\
34            21254487706647809158846074998871240076521705751797883416625624940758907\
35            """
36            if number < ZERO: # No negative numbers are permitted.
37                raise ArithmeticError(f"Cannot compute sqrt({number}).")
38            guess: Fraction = ONE # This will hold the current guess.
39            old_guess: Fraction = ZERO # 0.0 is just a dummy value != guess.
40            while old_guess != guess: # Repeat until nothing changes anymore.
41                old_guess = guess # The current guess becomes the old guess.
42                guess = ONE_HALF * (guess + number / guess) # The new guess.
43                max_steps -= 1 # Reduce the number of remaining steps.
44                if max_steps <= 0: # If we have exhausted the maximum steps...
45                    break # ...then we stop (and return the guess).
46            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.

```
"""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)
    '1.4142135623730950488016887242096980785696718753769480731766797379\
    90732478462107038850387534327641572735013846230912297024924836055850737\
    21264412149709993583141322266592750559275579995050115278206057147010955\
    99716059702745345968620147285174186408891986095523292304843087143214508\
    39762603627995251407989687253396546331808829640620615258352395054745750\
    287755996172983557522033753185701135437460340849884716038689997069900481\
    50305440277903164542478230684929369186215805784631115966687130130156185\
    6898723723528850926486124949771542183342042856860614682472077143585487\
    41556570696776537202264854470158588016207584749226572260020855844665214\
    5839893944370926591800311388246468157082630100594858704003186480342194\
    89727829064104507263688131373985525611732204025'
    >>> sqrt(Fraction(4, 1)).decimal_str()
    '2'
    >>> (ONE_HALF * (ONE + sqrt(Fraction(5, 1)))).decimal_str(420)
    '1.6180339887498948482045868343656381177203091798057628621354486227\
    05260462818902449707207204189391137484754088075386891752126633862223536\
    93179318006076672635443338908659593958290563832266131992829026788067520\
    8766892501711696207032210432162695486262963136144381497587012203408058\
    21254487706647809158846074998871240076521705751797883416625624940758907'
    """

    if number < ZERO: # No negative numbers are permitted.
        raise ArithmeticError(f"Cannot compute 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.

```
"""
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)
    '1.4142135623730950488016887242096980785696718753769480731766797379\
    90732478462107038850387534327641572735013846230912297024924836055850737\
    21264412149709993583141322266592750559275579995050115278206057147010955\
    99716059702745345968620147285174186408891986095523292304843087143214508\
    39762603627995251407989687253396546331808829640620615258352395054745750\
    28775599617298355752203375318570113543746034084988471603868999706990481\
    50305440277903164542478230684929369186215805784631115966687130130156185\
    6898723723528850926486124949771542183342042856860614682472077143585487\
    4155657069677653720226485447015888016207584749226572260020855844665214\
    5839893944370926591800311388246468157082630100594858704003186480342194\
    89727829064104507263688131373985525611732204025'
    >>> sqrt(Fraction(4, 1)).decimal_str()
    '2'
    >>> (ONE_HALF * (ONE + sqrt(Fraction(5, 1)))).decimal_str(420)
    '1.6180339887498948482045868343656381177203091798057628621354486227\
    05260462818902449707207204189391137484754088075386891752126633862223536\
    93179318006076672635443338908659593958290563832266131992829026788067520\
    327668925017116962070322210432162695486262963136144381497587012203408058\
    8795445474924618569536486449241044320771344947049565846788509874339442\
    21254487706647809158846074998871240076521705751797883416625624940758907'
    """

    if number < ZERO: # No negative numbers are permitted.
        raise ArithmeticError(f"Cannot compute 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.

```
"""
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)
    '1.4142135623730950488016887242096980785696718753769480731766797379\
    90732478462107038850387534327641572735013846230912297024924836055850737\
    21264412149709993583141322266592750559275579995050115278206057147010955\
    99716059702745345968620147285174186408891986095523292304843087143214508\
    39762603627995251407989687253396546331808829640620615258352395054745750\
    287755996172983557522033753185701135437460340849884716038689997069900481\
    50305440277903164542478230684929369186215805784631115966687130130156185\
    68987237235288509264861249497715421833420428568606014682472077143585487\
    41556570696776537202264854470158588016207584749226572260020855844665214\
    5839893944370926591800311388246468157082630100594858704003186480342194\
    89727829064104507263688131373985525611732204025'
    >>> sqrt(Fraction(4, 1)).decimal_str()
    '2'
    >>> (ONE_HALF * (ONE + sqrt(Fraction(5, 1)))).decimal_str(420)
    '1.6180339887498948482045868343656381177203091798057628621354486227\
    05260462818902449707207204189391137484754088075386891752126633862223536\
    93179318006076672635443338908659593958290563832266131992829026788067520\
    8766892501711696207032210432162695486262963136144381497587012203408058\
    879544547492461869536486449241044320771344947049565846788509874339442\
    21254487706647809158846074998871240076521705751797883416625624940758907'
    """

    if number < ZERO: # No negative numbers are permitted.
        raise ArithmeticError(f"Cannot compute 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.

```
"""
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)
    '1.4142135623730950488016887242096980785696718753769480731766797379\
90732478462107038850387534327641572735013846230912297024924836055850737\
21264412149709993583141322266592750559275579995050115278206057147010955\
99716059702745345968620147285174186408891986095523292304843087143214508\
39762603627995251407989687253396546331808829640620615258352395054745750\
28775996172983557522033753185701135437460340849884716038689997069900481\
50305440277903164542478230684929369186215805784631115966687130130156185\
68987237235288509264861249497715421833420428568606014682472077143585487\
41556570696776537202264854470158588616207584749226572260020855844665214\
5839893944370926591800311388246468157082630100594858704003186480342194\
89727829064104507263688131373985525611732204025'
    >>> sqrt(Fraction(4, 1)).decimal_str()
    '2'
    >>> (ONE_HALF * (ONE + sqrt(Fraction(5, 1)))).decimal_str(420)
    '1.6180339887498948482045868343656381177203091798057628621354486227\
05260462818902449707207204189391137484754088075386891752126633862223536\
93179318006076672635443338908659593958290563832266131992829026788067520\
8766892501711696207032210432162695486262963136144381497587012203408058\
8795445474924618695364864494241044320771344947049565846788509874339442\
21254487706647809158846074998871240076521705751797883416625624940758907'
    """

    if number < ZERO: # No negative numbers are permitted.
        raise ArithmeticError(f"Cannot compute 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.

```
1 """A square root algorithm based on fractions."""
2
3 from fraction import ONE, ONE_HALF, ZERO, Fraction
4
5
6 def sqrt(number: Fraction, max_steps: int = 10) -> Fraction:
7     """
8         Compute the square root of a given :class:`Fraction`.
9
10    :param number: The rational number to compute the square root of.
11    :param max_steps: the maximum number of steps, defaults to `10`.
12    :return: A value `v` such that `v * v` is approximately `number`.
13
14    >>> sqrt(Fraction(2, 1)).decimal_str(750)
15    '1.4142135623730950488016887242096980785696718753769480731766797379\
16    90732478462107038850387534327641572735013846230912297024924836055850737\
17    21264412149709993583141322266592750559275579995050115278206057147010955\
18    99716059702745345968620147285174186408891986095523292304843087143214508\
19    39762603627995251407989687253396546331808829640620615258352395054745750\
20    28775996172983557522033753185701135437460340849884716038689997069900481\
21    50305440277903164542478230684929369186215805784631115966687130130156185\
22    68987237235288509264861249497715421833420428568606014682472077143585487\
23    41556570696776537202264854470158588016207584749226572260020855844665214\
24    58398893944370926591800311388246468157082630100594858704003186480342194\
25    89727829064104507263688131373985525611732204025'
26    >>> sqrt(Fraction(4, 1)).decimal_str()
27    '2'
28    >>> (ONE_HALF * (ONE + sqrt(Fraction(5, 1)))).decimal_str(420)
29    '1.6180339887498948482045868343656381177203091798057628621354486227\
30    05260462818902449707207204189391137484754088075386891752126633862223536\
31    93179318006076672635443338908659593958290563832266131992829026788067520\
32    87668925017116962070322210432162695486262963136144381497587012203408058\
33    87954454749246185695364864449241044320771344947049565846788509874339442\
34    21254487706647809158846074998871240076521705751797883416625624940758907\
35    """
36    if number < ZERO: # No negative numbers are permitted.
37        raise ArithmeticError(f"Cannot computed sqrt({number}).")
38    guess: Fraction = ONE # This will hold the current guess.
39    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)`.

```
1 """A square root algorithm based on fractions."""
2
3 from fraction import ONE, ONE_HALF, ZERO, Fraction
4
5
6 def sqrt(number: Fraction, max_steps: int = 10) -> Fraction:
7     """
8         Compute the square root of a given :class:`Fraction`.
9
10    :param number: The rational number to compute the square root of.
11    :param max_steps: the maximum number of steps, defaults to `10`.
12    :return: A value `v` such that `v * v` is approximately `number`.
13
14    >>> sqrt(Fraction(2, 1)).decimal_str(750)
15    '1.4142135623730950488016887242096980785696718753769480731766797379\
16    90732478462107038850387534327641572735013846230912297024924836055850737\
17    21264412149709993583141322266592750559275579995050115278206057147010955\
18    99716059702745345968620147285174186408891986095523292304843087143214508\
19    39762603627995251407989687253396546331808829640620615258352395054745750\
20    28775996172983557522033753185701135437460340849884716038689997069900481\
21    50305440277903164542478230684929369186215805784631115966687130130156185\
22    68987237235288509264861249497715421833420428568606014682472077143585487\
23    41556570696776537202264854470158588016207584749226572260020855844665214\
24    58398893944370926591800311388246468157082630100594858704003186480342194\
25    89727829064104507263688131373985525611732204025'
26    >>> sqrt(Fraction(4, 1)).decimal_str()
27    '2'
28    >>> (ONE_HALF * (ONE + sqrt(Fraction(5, 1)))).decimal_str(420)
29    '1.6180339887498948482045868343656381177203091798057628621354486227\
30    05260462818902449707207204189391137484754088075386891752126633862223536\
31    9317931800607667263544338908659593958290563832266131992829026788067520\
32    87668925017116962070322210432162695486262963136144381497587012203408058\
33    87954454749246185695364864449241044320771344947049565846788509874339442\
34    21254487706647809158846074998871240076521705751797883416625624940758907\
35    """
36    if number < ZERO: # No negative numbers are permitted.
37        raise ArithmeticError(f"Cannot computed sqrt({number}).")
38    guess: Fraction = ONE # This will hold the current guess.
39    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.

```
1 """A square root algorithm based on fractions."""
2
3 from fraction import ONE, ONE_HALF, ZERO, Fraction
4
5
6 def sqrt(number: Fraction, max_steps: int = 10) -> Fraction:
7     """
8         Compute the square root of a given :class:`Fraction`.
9
10    :param number: The rational number to compute the square root of.
11    :param max_steps: the maximum number of steps, defaults to `10`.
12    :return: A value `v` such that `v * v` is approximately `number`.
13
14    >>> sqrt(Fraction(2, 1)).decimal_str(750)
15    '1.4142135623730950488016887242096980785696718753769480731766797379\
16    90732478462107038850387534327641572735013846230912297024924836055850737\
17    21264412149709993583141322266592750559275579995050115278206057147010955\
18    99716059702745345968620147285174186408891986095523292304843087143214508\
19    39762603627995251407989687253396546331808829640620615258352395054745750\
20    28775996172983557522033753185701135437460340849884716038689997069900481\
21    50305440277903164542478230684929369186215805784631115966687130130156185\
22    68987237235288509264861249497715421833420428568606014682472077143585487\
23    41556570696776537202264854470158588016207584749226572260020855844665214\
24    58398893944370926591800311388246468157082630100594858704003186480342194\
25    89727829064104507263688131373985525611732204025'
26    >>> sqrt(Fraction(4, 1)).decimal_str()
27    '2'
28    >>> (ONE_HALF * (ONE + sqrt(Fraction(5, 1)))).decimal_str(420)
29    '1.6180339887498948482045868343656381177203091798057628621354486227\
30    05260462818902449707207204189391137484754088075386891752126633862223536\
31    9317931800607667263544338908659593958290563832266131992829026788067520\
32    87668925017116962070322210432162695486262963136144381497587012203408058\
33    87954454749246185695364864449241044320771344947049565846788509874339442\
34    21254487706647809158846074998871240076521705751797883416625624940758907\
35    """
36    if number < ZERO: # No negative numbers are permitted.
37        raise ArithmeticError(f"Cannot computed sqrt({number}).")
38    guess: Fraction = ONE # This will hold the current guess.
39    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!

```
1 """A square root algorithm based on fractions."""
2
3 from fraction import ONE, ONE_HALF, ZERO, Fraction
4
5
6 def sqrt(number: Fraction, max_steps: int = 10) -> Fraction:
7     """
8         Compute the square root of a given :class:`Fraction`.
9
10    :param number: The rational number to compute the square root of.
11    :param max_steps: the maximum number of steps, defaults to `10`.
12    :return: A value `v` such that `v * v` is approximately `number`.
13
14    >>> sqrt(Fraction(2, 1)).decimal_str(750)
15    '1.4142135623730950488016887242096980785696718753769480731766797379\
16    90732478462107038850387534327641572735013846230912297024924836055850737\
17    21264412149709993583141322266592750559275579995050115278206057147010955\
18    99716059702745345968620147285174186408891986095523292304843087143214508\
19    39762603627995251407989687253396546331808829640620615258352395054745750\
20    28775996172983557522033753185701135437460340849884716038689997069900481\
21    50305440277903164542478230684929369186215805784631115966687130130156185\
22    68987237235288509264861249497715421833420428568606014682472077143585487\
23    41556570696776537202264854470158588016207584749226572260020855844665214\
24    58398893944370926591800311388246468157082630100594858704003186480342194\
25    89727829064104507263688131373985525611732204025'
26    >>> sqrt(Fraction(4, 1)).decimal_str()
27    '2'
28    >>> (ONE_HALF * (ONE + sqrt(Fraction(5, 1)))).decimal_str(420)
29    '1.6180339887498948482045868343656381177203091798057628621354486227\
30    05260462818902449707207204189391137484754088075386891752126633862223536\
31    9317931800607667263544338908659593958290563832266131992829026788067520\
32    87668925017116962070322210432162695486262963136144381497587012203408058\
33    87954454749246185695364864449241044320771344947049565846788509874339442\
34    21254487706647809158846074998871240076521705751797883416625624940758907\
35    """
36    if number < ZERO: # No negative numbers are permitted.
37        raise ArithmeticError(f"Cannot compute sqrt({number}).")
38    guess: Fraction = ONE # This will hold the current guess.
39    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 [19, 31].

```
1     """A square root algorithm based on fractions."""
2
3     from fraction import ONE, ONE_HALF, ZERO, Fraction
4
5
6     def sqrt(number: Fraction, max_steps: int = 10) -> Fraction:
7         """
8             Compute the square root of a given :class:`Fraction`.
9
10            :param number: The rational number to compute the square root of.
11            :param max_steps: the maximum number of steps, defaults to `10`.
12            :return: A value `v` such that `v * v` is approximately `number`.
13
14            >>> sqrt(Fraction(2, 1)).decimal_str(750)
15            '1.4142135623730950488016887242096980785696718753769480731766797379\
16            90732478462107038850387534327641572735013846230912297024924836055850737\
17            21264412149709993583141322266592750559275579995050115278206057147010955\
18            99716059702745345968620147285174186408891986095523292304843087143214508\
19            39762603627995251407989687253396546331808829640620615258352395054745750\
20            28775996172983557522033753185701135437460340849884716038689997069900481\
21            50305440277903164542478230684929369186215805784631115966687130130156185\
22            68987237235288509264861249497715421833420428568606014682472077143585487\
23            41556570696776537202264854470158588016207584749226572260020855844665214\
24            58398893944370926591800311388246468157082630100594858704003186480342194\
25            89727829064104507263688131373985525611732204025 \
26            >>> sqrt(Fraction(4, 1)).decimal_str()
27            '2'
28            >>> (ONE_HALF * (ONE + sqrt(Fraction(5, 1)))).decimal_str(420)
29            '1.6180339887498948482045868343656381177203091798057628621354486227\
30            05260462818902449707207204189391137484754088075386891752126633862223536\
31            9317931800607667263544338908659593958290563832266131992829026788067520\
32            87668925017116962070322210432162695486262963136144381497587012203408058\
33            87954454749246185695364864449241044320771344947049565846788509874339442\
34            21254487706647809158846074998871240076521705751797883416625624940758907\
35            """
36            if number < ZERO: # No negative numbers are permitted.
37                raise ArithmeticError(f"Cannot computed sqrt({number}).")
38            guess: Fraction = ONE           # This will hold the current guess.
39            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 [19, 31].
- Wenn unsere Funktion diese Zahl nach zehn Schritten nicht auf 700 Nachkommastellen genau liefert, wird dieser Doctest fehlschlagen!

```
1     """A square root algorithm based on fractions."""
2
3     from fraction import ONE, ONE_HALF, ZERO, Fraction
4
5
6     def sqrt(number: Fraction, max_steps: int = 10) -> Fraction:
7         """
8             Compute the square root of a given :class:`Fraction`.
9
10            :param number: The rational number to compute the square root of.
11            :param max_steps: the maximum number of steps, defaults to `10`.
12            :return: A value `v` such that `v * v` is approximately `number`.
13
14            >>> sqrt(Fraction(2, 1)).decimal_str(750)
15            '1.4142135623730950488016887242096980785696718753769480731766797379\
16            90732478462107038850387534327641572735013846230912297024924836055850737\
17            21264412149709993583141322266592750559275579995050115278206057147010955\
18            99716059702745345968620147285174186408891986095523292304843087143214508\
19            39762603627995251407989687253396546331808829640620615258352395054745750\
20            28775996172983557522033753185701135437460340849884716038689997069900481\
21            50305440277903164542478230684929369186215805784631115966687130130156185\
22            68987237235288509264861249497715421833420428568606014682472077143585487\
23            41556570696776537202264854470158588016207584749226572260020855844665214\
24            58398893944370926591800311388246468157082630100594858704003186480342194\
25            89727829064104507263688131373985525611732204025'
26            >>> sqrt(Fraction(4, 1)).decimal_str()
27            '2'
28            >>> (ONE_HALF * (ONE + sqrt(Fraction(5, 1)))).decimal_str(420)
29            '1.6180339887498948482045868343656381177203091798057628621354486227\
30            05260462818902449707207204189391137484754088075386891752126633862223536\
31            9317931800607667263544338908659593958290563832266131992829026788067520\
32            87668925017116962070322210432162695486262963136144381497587012203408058\
33            87954454749246185695364864449241044320771344947049565846788509874339442\
34            21254487706647809158846074998871240076521705751797883416625624940758907\
35            """
36            if number < ZERO: # No negative numbers are permitted.
37                raise ArithmeticError(f"Cannot computed sqrt({number}).")
38            guess: Fraction = ONE # This will hold the current guess.
39            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 [19, 31].
- 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.

```
1     """A square root algorithm based on fractions."""
2
3     from fraction import ONE, ONE_HALF, ZERO, Fraction
4
5
6     def sqrt(number: Fraction, max_steps: int = 10) -> Fraction:
7         """
8             Compute the square root of a given :class:`Fraction`.
9
10            :param number: The rational number to compute the square root of.
11            :param max_steps: the maximum number of steps, defaults to `10`.
12            :return: A value `v` such that `v * v` is approximately `number`.
13
14            >>> sqrt(Fraction(2, 1)).decimal_str(750)
15            '1.4142135623730950488016887242096980785696718753769480731766797379\
16            90732478462107038850387534327641572735013846230912297024924836055850737\
17            21264412149709993583141322266592750559275579995050115278206057147010955\
18            99716059702745345968620147285174186408891986095523292304843087143214508\
19            39762603627995251407989687253396546331808829640620615258352395054745750\
20            28775996172983557522033753185701135437460340849884716038689997069900481\
21            50305440277903164542478230684929369186215805784631115966687130130156185\
22            68987237235288509264861249497715421833420428568606014682472077143585487\
23            41556570696776537202264854470158588016207584749226572260020855844665214\
24            58398893944370926591800311388246468157082630100594858704003186480342194\
25            89727829064104507263688131373985525611732204025'
26            >>> sqrt(Fraction(4, 1)).decimal_str()
27            '2'
28            >>> (ONE_HALF * (ONE + sqrt(Fraction(5, 1)))).decimal_str(420)
29            '1.6180339887498948482045868343656381177203091798057628621354486227\
30            05260462818902449707207204189391137484754088075386891752126633862223536\
31            9317931800607667263544338908659593958290563832266131992829026788067520\
32            87668925017116962070322210432162695486262963136144381497587012203408058\
33            87954454749246185695364864449241044320771344947049565846788509874339442\
34            21254487706647809158846074998871240076521705751797883416625624940758907\
35            """
36            if number < ZERO: # No negative numbers are permitted.
37                raise ArithmeticError(f"Cannot computed sqrt({number}).")
38            guess: Fraction = ONE # This will hold the current guess.
39            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 [19, 31].
- 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.

```
1     """A square root algorithm based on fractions."""
2
3     from fraction import ONE, ONE_HALF, ZERO, Fraction
4
5
6     def sqrt(number: Fraction, max_steps: int = 10) -> Fraction:
7         """
8             Compute the square root of a given :class:`Fraction`.
9
10            :param number: The rational number to compute the square root of.
11            :param max_steps: the maximum number of steps, defaults to `10`.
12            :return: A value `v` such that `v * v` is approximately `number`.
13
14            >>> sqrt(Fraction(2, 1)).decimal_str(750)
15            '1.4142135623730950488016887242096980785696718753769480731766797379\
16            90732478462107038850387534327641572735013846230912297024924836055850737\
17            21264412149709993583141322266592750559275579995050115278206057147010955\
18            99716059702745345968620147285174186408891986095523292304843087143214508\
19            39762603627995251407989687253396546331808829640620615258352395054745750\
20            28775996172983557522033753185701135437460340849884716038689997069900481\
21            50305440277903164542478230684929369186215805784631115966687130130156185\
22            68987237235288509264861249497715421833420428568606014682472077143585487\
23            41556570696776537202264854470158588016207584749226572260020855844665214\
24            58398893944370926591800311388246468157082630100594858704003186480342194\
25            89727829064104507263688131373985525611732204025'
26            >>> sqrt(Fraction(4, 1)).decimal_str()
27            '2'
28            >>> (ONE_HALF * (ONE + sqrt(Fraction(5, 1)))).decimal_str(420)
29            '1.6180339887498948482045868343656381177203091798057628621354486227\
30            05260462818902449707207204189391137484754088075386891752126633862223536\
31            9317931800607667263544338908659593958290563832266131992829026788067520\
32            87668925017116962070322210432162695486262963136144381497587012203408058\
33            87954454749246185695364864449241044320771344947049565846788509874339442\
34            21254487706647809158846074998871240076521705751797883416625624940758907\
35            """
36            if number < ZERO: # No negative numbers are permitted.
37                raise ArithmeticError(f"Cannot computed sqrt({number}).")
38            guess: Fraction = ONE # This will hold the current guess.
39            old_guess: Fraction = ZERO # 0.0 is just a dummy value != guess.
```

sqrt: Doctests

- Wir holen uns den korrekten Wert von [19, 31].
- 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.

```
1     """A square root algorithm based on fractions."""
2
3     from fraction import ONE, ONE_HALF, ZERO, Fraction
4
5
6     def sqrt(number: Fraction, max_steps: int = 10) -> Fraction:
7         """
8             Compute the square root of a given :class:`Fraction`.
9
10            :param number: The rational number to compute the square root of.
11            :param max_steps: the maximum number of steps, defaults to `10`.
12            :return: A value `v` such that `v * v` is approximately `number`.
13
14            >>> sqrt(Fraction(2, 1)).decimal_str(750)
15            '1.4142135623730950488016887242096980785696718753769480731766797379\
16            90732478462107038850387534327641572735013846230912297024924836055850737\
17            21264412149709993583141322266592750559275579995050115278206057147010955\
18            99716059702745345968620147285174186408891986095523292304843087143214508\
19            39762603627995251407989687253396546331808829640620615258352395054745750\
20            28775996172983557522033753185701135437460340849884716038689997069900481\
21            50305440277903164542478230684929369186215805784631115966687130130156185\
22            68987237235288509264861249497715421833420428568606014682472077143585487\
23            41556570696776537202264854470158588016207584749226572260020855844665214\
24            58398893944370926591800311388246468157082630100594858704003186480342194\
25            89727829064104507263688131373985525611732204025'
26            >>> sqrt(Fraction(4, 1)).decimal_str()
27            '2'
28            >>> (ONE_HALF * (ONE + sqrt(Fraction(5, 1)))).decimal_str(420)
29            '1.6180339887498948482045868343656381177203091798057628621354486227\
30            05260462818902449707207204189391137484754088075386891752126633862223536\
31            9317931800607667263544338908659593958290563832266131992829026788067520\
32            87668925017116962070322210432162695486262963136144381497587012203408058\
33            87954454749246185695364864449241044320771344947049565846788509874339442\
34            21254487706647809158846074998871240076521705751797883416625624940758907\
35            """
36            if number < ZERO: # No negative numbers are permitted.
37                raise ArithmeticError(f"Cannot computed sqrt({number}).")
38            guess: Fraction = ONE # This will hold the current guess.
39            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.

```
1 """A square root algorithm based on fractions."""
2
3 from fraction import ONE, ONE_HALF, ZERO, Fraction
4
5
6 def sqrt(number: Fraction, max_steps: int = 10) -> Fraction:
7     """
8         Compute the square root of a given :class:`Fraction`.
9
10    :param number: The rational number to compute the square root of.
11    :param max_steps: the maximum number of steps, defaults to `10`.
12    :return: A value `v` such that `v * v` is approximately `number`.
13
14    >>> sqrt(Fraction(2, 1)).decimal_str(750)
15    '1.4142135623730950488016887242096980785696718753769480731766797379\
16    90732478462107038850387534327641572735013846230912297024924836055850737\
17    21264412149709993583141322266592750559275579995050115278206057147010955\
18    99716059702745345968620147285174186408891986095523292304843087143214508\
19    39762603627995251407989687253396546331808829640620615258352395054745750\
20    28775996172983557522033753185701135437460340849884716038689997069900481\
21    50305440277903164542478230684929369186215805784631115966687130130156185\
22    68987237235288509264861249497715421833420428568606014682472077143585487\
23    41556570696776537202264854470158588016207584749226572260020855844665214\
24    58398893944370926591800311388246468157082630100594858704003186480342194\
25    89727829064104507263688131373985525611732204025'
26    >>> sqrt(Fraction(4, 1)).decimal_str()
27    '2'
28    >>> (ONE_HALF * (ONE + sqrt(Fraction(5, 1)))).decimal_str(420)
29    '1.6180339887498948482045868343656381177203091798057628621354486227\
30    05260462818902449707207204189391137484754088075386891752126633862223536\
31    9317931800607667263544338908659593958290563832266131992829026788067520\
32    87668925017116962070322210432162695486262963136144381497587012203408058\
33    87954454749246185695364864449241044320771344947049565846788509874339442\
34    21254487706647809158846074998871240076521705751797883416625624940758907\
35    """
36    if number < ZERO: # No negative numbers are permitted.
37        raise ArithmeticError(f"Cannot computed sqrt({number}).")
38    guess: Fraction = ONE # This will hold the current guess.
39    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 $\phi^{4,8,30}$ berechnen.

```
1     """A square root algorithm based on fractions."""
2
3     from fraction import ONE, ONE_HALF, ZERO, Fraction
4
5
6     def sqrt(number: Fraction, max_steps: int = 10) -> Fraction:
7         """
8             Compute the square root of a given :class:`Fraction`.
9
10            :param number: The rational number to compute the square root of.
11            :param max_steps: the maximum number of steps, defaults to `10`.
12            :return: A value `v` such that `v * v` is approximately `number`.
13
14            >>> sqrt(Fraction(2, 1)).decimal_str(750)
15            '1.4142135623730950488016887242096980785696718753769480731766797379\
16            90732478462107038850387534327641572735013846230912297024924836055850737\
17            21264412149709993583141322266592750559275579995050115278206057147010955\
18            99716059702745345968620147285174186408891986095523292304843087143214508\
19            39762603627995251407989687253396546331808829640620615258352395054745750\
20            28775996172983557522033753185701135437460340849884716038689997069900481\
21            50305440277903164542478230684929369186215805784631115966687130130156185\
22            68987237235288509264861249497715421833420428568606014682472077143585487\
23            41556570696776537202264854470158588016207584749226572260020855844665214\
24            58398893944370926591800311388246468157082630100594858704003186480342194\
25            89727829064104507263688131373985525611732204025'
26            >>> sqrt(Fraction(4, 1)).decimal_str()
27            '2'
28            >>> (ONE_HALF * (ONE + sqrt(Fraction(5, 1)))).decimal_str(420)
29            '1.6180339887498948482045868343656381177203091798057628621354486227\
30            05260462818902449707207204189391137484754088075386891752126633862223536\
31            9317931800607667263544338908659593958290563832266131992829026788067520\
32            87668925017116962070322210432162695486262963136144381497587012203408058\
33            87954454749246185695364864449241044320771344947049565846788509874339442\
34            21254487706647809158846074998871240076521705751797883416625624940758907\
35            """
36            if number < ZERO: # No negative numbers are permitted.
37                raise ArithmeticError(f"Cannot compute sqrt({number}).")
38            guess: Fraction = ONE # This will hold the current guess.
39            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 $\phi^{4,8,30}$ berechnen.
- Dieser entspricht $\frac{1+\sqrt{5}}{2}$.

```
1     """A square root algorithm based on fractions."""
2
3     from fraction import ONE, ONE_HALF, ZERO, Fraction
4
5
6     def sqrt(number: Fraction, max_steps: int = 10) -> Fraction:
7         """
8             Compute the square root of a given :class:`Fraction`.
9
10            :param number: The rational number to compute the square root of.
11            :param max_steps: the maximum number of steps, defaults to `10`.
12            :return: A value `v` such that `v * v` is approximately `number`.
13
14            >>> sqrt(Fraction(2, 1)).decimal_str(750)
15            '1.4142135623730950488016887242096980785696718753769480731766797379\
16            90732478462107038850387534327641572735013846230912297024924836055850737\
17            21264412149709993583141322266592750559275579995050115278206057147010955\
18            99716059702745345968620147285174186408891986095523292304843087143214508\
19            39762603627995251407989687253396546331808829640620615258352395054745750\
20            28775996172983557522033753185701135437460340849884716038689997069900481\
21            50305440277903164542478230684929369186215805784631115966687130130156185\
22            68987237235288509264861249497715421833420428568606014682472077143585487\
23            41556570696776537202264854470158588016207584749226572260020855844665214\
24            58398893944370926591800311388246468157082630100594858704003186480342194\
25            89727829064104507263688131373985525611732204025'
26            >>> sqrt(Fraction(4, 1)).decimal_str()
27            '2'
28            >>> (ONE_HALF * (ONE + sqrt(Fraction(5, 1)))).decimal_str(420)
29            '1.6180339887498948482045868343656381177203091798057628621354486227\
30            05260462818902449707207204189391137484754088075386891752126633862223536\
31            93179318006076672635443338908659593958290563832266131992829026788067520\
32            87668925017116962070322210432162695486262963136144381497587012203408058\
33            87954454749246185695364864449241044320771344947049565846788509874339442\
34            21254487706647809158846074998871240076521705751797883416625624940758907\
35            """
36            if number < ZERO: # No negative numbers are permitted.
37                raise ArithmeticError(f"Cannot compute sqrt({number}).")
38            guess: Fraction = ONE # This will hold the current guess.
39            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 $\phi^{4,8,30}$ berechnen.
- Dieser entspricht $\frac{1+\sqrt{5}}{2}$.
- Wir können das als $\text{ONE_HALF} * (\text{ONE} + \text{sqrt}(\text{Fraction}(5, 1)))$ schreiben.

```
1     """A square root algorithm based on fractions."""
2
3     from fraction import ONE, ONE_HALF, ZERO, Fraction
4
5
6     def sqrt(number: Fraction, max_steps: int = 10) -> Fraction:
7         """
8             Compute the square root of a given :class:`Fraction`.
9
10            :param number: The rational number to compute the square root of.
11            :param max_steps: the maximum number of steps, defaults to `10`.
12            :return: A value `v` such that `v * v` is approximately `number`.
13
14            >>> sqrt(Fraction(2, 1)).decimal_str(750)
15            '1.4142135623730950488016887242096980785696718753769480731766797379\
16            90732478462107038850387534327641572735013846230912297024924836055850737\
17            21264412149709993583141322266592750559275579995050115278206057147010955\
18            99716059702745345968620147285174186408891986095523292304843087143214508\
19            39762603627995251407989687253396546331808829640620615258352395054745750\
20            28775996172983557522033753185701135437460340849884716038689997069900481\
21            50305440277903164542478230684929369186215805784631115966687130130156185\
22            68987237235288509264861249497715421833420428568606014682472077143585487\
23            41556570696776537202264854470158588016207584749226572260020855844665214\
24            58398893944370926591800311388246468157082630100594858704003186480342194\
25            89727829064104507263688131373985525611732204025'
26            >>> sqrt(Fraction(4, 1)).decimal_str()
27            '2'
28            >>> (ONE_HALF * (ONE + sqrt(Fraction(5, 1)))).decimal_str(420)
29            '1.6180339887498948482045868343656381177203091798057628621354486227\
30            05260462818902449707207204189391137484754088075386891752126633862223536\
31            93179318006076672635443338908659593958290563832266131992829026788067520\
32            87668925017116962070322210432162695486262963136144381497587012203408058\
33            87954454749246185695364864449241044320771344947049565846788509874339442\
34            21254487706647809158846074998871240076521705751797883416625624940758907\
35            """
36            if number < ZERO: # No negative numbers are permitted.
37                raise ArithmeticError(f"Cannot compute sqrt({number}).")
38            guess: Fraction = ONE # This will hold the current guess.
39            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 $\phi^{4,8,30}$ berechnen.
- Dieser entspricht $\frac{1+\sqrt{5}}{2}$.
- Wir können das als $\text{ONE_HALF} * (\text{ONE} + \text{sqrt}(\text{Fraction}(5, 1)))$ schreiben.
- Im Doctest erwarten wir, dass das Ergebnis auf 420 Nachkommastellen (die wir von [9] nehmen) genau ist.

```
1 """A square root algorithm based on fractions."""
2
3 from fraction import ONE, ONE_HALF, ZERO, Fraction
4
5
6 def sqrt(number: Fraction, max_steps: int = 10) -> Fraction:
7     """
8         Compute the square root of a given :class:`Fraction`.
9
10    :param number: The rational number to compute the square root of.
11    :param max_steps: the maximum number of steps, defaults to `10`.
12    :return: A value `v` such that `v * v` is approximately `number`.
13
14    >>> sqrt(Fraction(2, 1)).decimal_str(750)
15    '1.4142135623730950488016887242096980785696718753769480731766797379\
16    90732478462107038850387534327641572735013846230912297024924836055850737\
17    21264412149709993583141322266592750559275579995050115278206057147010955\
18    99716059702745345968620147285174186408891986095523292304843087143214508\
19    39762603627995251407989687253396546331808829640620615258352395054745750\
20    28775996172983557522033753185701135437460340849884716038689997069900481\
21    50305440277903164542478230684929369186215805784631115966687130130156185\
22    68987237235288509264861249497715421833420428568606014682472077143585487\
23    41556570696776537202264854470158588016207584749226572260020855844665214\
24    58398893944370926591800311388246468157082630100594858704003186480342194\
25    89727829064104507263688131373985525611732204025'
26    >>> sqrt(Fraction(4, 1)).decimal_str()
27    '2'
28    >>> (ONE_HALF * (ONE + sqrt(Fraction(5, 1)))).decimal_str(420)
29    '1.6180339887498948482045868343656381177203091798057628621354486227\
30    05260462818902449707207204189391137484754088075386891752126633862223536\
31    93179318006076672635443338908659593958290563832266131992829026788067520\
32    87668925017116962070322210432162695486262963136144381497587012203408058\
33    87954454749246185695364864449241044320771344947049565846788509874339442\
34    21254487706647809158846074998871240076521705751797883416625624940758907\
35    """
36    if number < ZERO: # No negative numbers are permitted.
37        raise ArithmeticError(f"Cannot compute sqrt({number}).")
38    guess: Fraction = ONE # This will hold the current guess.
39    old_guess: Fraction = ZERO # 0.0 is just a dummy value != guess.
```

sqrt: Doctests

- Zu guter Letzt wollen wir den Goldenen Schnitt $\phi^{4,8,30}$ berechnen.
- Dieser entspricht $\frac{1+\sqrt{5}}{2}$.
- Wir können das als $\text{ONE_HALF} * (\text{ONE} + \text{sqrt}(\text{Fraction}(5, 1)))$ schreiben.
- Im Doctest erwarten wir, dass das Ergebnis auf 420 Nachkommastellen (die wir von [9] nehmen) genau ist.
- Wir führen die Doctests mit pytest aus.

```
1 """A square root algorithm based on fractions."""
2
3 from fraction import ONE, ONE_HALF, ZERO, Fraction
4
5
6 def sqrt(number: Fraction, max_steps: int = 10) -> Fraction:
7     """
8         Compute the square root of a given :class:`Fraction`.
9
10        :param number: The rational number to compute the square root of.
11        :param max_steps: the maximum number of steps, defaults to `10`.
12        :return: A value `v` such that `v * v` is approximately `number`.
13
14    >>> sqrt(Fraction(2, 1)).decimal_str(750)
15    '1.4142135623730950488016887242096980785696718753769480731766797379\
16    90732478462107038850387534327641572735013846230912297024924836055850737\
17    21264412149709993583141322266592750559275579995050115278206057147010955\
18    99716059702745345968620147285174186408891986095523292304843087143214508\
19    39762603627995251407989687253396546331808829640620615258352395054745750\
20    28775996172983557522033753185701135437460340849884716038689997069900481\
21    50305440277903164542478230684929369186215805784631115966687130130156185\
22    68987237235288509264861249497715421833420428568606014682472077143585487\
23    41556570696776537202264854470158588016207584749226572260020855844665214\
24    58398893944370926591800311388246468157082630100594858704003186480342194\
25    89727829064104507263688131373985525611732204025'
26    >>> sqrt(Fraction(4, 1)).decimal_str()
27    '2'
28    >>> (ONE_HALF * (ONE + sqrt(Fraction(5, 1)))).decimal_str(420)
29    '1.6180339887498948482045868343656381177203091798057628621354486227\
30    05260462818902449707207204189391137484754088075386891752126633862223536\
31    9317931800607667263544338908659593958290563832266131992829026788067520\
32    87668925017116962070322210432162695486262963136144381497587012203408058\
33    87954454749246185695364864449241044320771344947049565846788509874339442\
34    21254487706647809158846074998871240076521705751797883416625624940758907\
35    """
36    if number < ZERO: # No negative numbers are permitted.
37        raise ArithmeticError(f"Cannot computed sqrt({number}).")
38    guess: Fraction = ONE # This will hold the current guess.
39    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 [9] 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
5 fraction_sqrt.py .
6
7 ===== 1 passed in 0.02s
  ↪ =====
8 # pytest 9.0.2 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 [9] nehmen) genau ist.
- Wir führen die Doctests mit pytest aus.
- Sie sind alle erfolgreich!
- Wir haben nun Grundschulmathematik in eine Klasse in Python gegossen.

```
1 $ pytest --timeout=10 --no-header --tb=short --doctest-modules
  ↗ fraction_sqrt.py
=====
2 ===== test session starts
  ↗ =====
3 collected 1 item
4
5 fraction_sqrt.py .
6
7 ===== 1 passed in 0.02s
  ↗ =====
8 # pytest 9.0.2 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 [9] 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.

```
1 $ pytest --timeout=10 --no-header --tb=short --doctest-modules
  ↪ fraction_sqrt.py
=====
2 ===== test session starts
  ↪ =====
3 collected 1 item
4
5 fraction_sqrt.py .
6
7 ===== 1 passed in 0.02s
  ↪ =====
8 # pytest 9.0.2 with pytest-timeout 2.4.0 succeeded with exit code 0.
```

[100%]



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.

```
1 $ pytest --timeout=10 --no-header --tb=short --doctest-modules
  ↗ fraction_sqrt.py
=====
2 ===== test session starts
  ↗ =====
3 collected 1 item
4
5 fraction_sqrt.py .
6
7 ===== 1 passed in 0.02s
  ↗ =====
8 # pytest 9.0.2 with pytest-timeout 2.4.0 succeeded with exit code 0.
```

[100%]



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 .

=====
           1 passed in 0.02s
  ↪ =====
# pytest 9.0.2 with pytest-timeout 2.4.0 succeeded with exit code 0.
[100%]
```



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, 349).
- [2] Kent L. Beck. *JUnit Pocket Guide*. Sebastopol, CA, USA: O'Reilly Media, Inc., Sep. 2004. ISBN: 978-0-596-00743-0 (siehe S. 351).
- [3] 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. 350).
- [4] 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, 351).
- [5] 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. 349).
- [6] 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. 350).
- [7] "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. 349).
- [8] Euclid of Alexandria (*Εύκλειδης*). *Euclid's Elements of Geometry* ($\Sigma\tauοιχεῖα$). *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, 351).
- [9] 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).
- [10] 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. 349).

References II



- [11] John Hunt. *A Beginner's Guide to Python 3 Programming*. 2. Aufl. Undergraduate Topics in Computer Science (UTICS). Cham, Switzerland: Springer, 2023. ISBN: 978-3-031-35121-1. doi:10.1007/978-3-031-35122-8 (siehe S. 350).
- [12] 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. 349).
- [13] 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. 350).
- [14] 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. 350).
- [15] Kent D. Lee und Steve Hubbard. *Data Structures and Algorithms with Python*. Undergraduate Topics in Computer Science (UTICS). Cham, Switzerland: Springer, 2015. ISBN: 978-3-319-13071-2. doi:10.1007/978-3-319-13072-9 (siehe S. 350).
- [16] 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. 350).
- [17] Mark Lutz. *Learning Python*. 6. Aufl. Sebastopol, CA, USA: O'Reilly Media, Inc., März 2025. ISBN: 978-1-0981-7130-8 (siehe S. 350).
- [18] 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. 350).
- [19] 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).
- [20] 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. 351).

References III



- [21] 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. 350).
- [22] 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. 351).
- [23] 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. 350, 351).
- [24] 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. 349).
- [25] 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, 349).
- [26] 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. 350).
- [27] 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. 351).
- [28] 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. 351).
- [29] Anna Skoulikari. *Learning Git*. Sebastopol, CA, USA: O'Reilly Media, Inc., Mai 2023. ISBN: 978-1-0981-3391-7 (siehe S. 349).
- [30] 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, 351).

References IV



- [31] 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).
- [32] 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. 349, 350).
- [33] 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).
- [34] 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. 351).
- [35] 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. 349, 351).
- [36] 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. 350).
- [37] 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. 350).
- [38] 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. 349).
- [39] 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), 2024–2025. URL: <https://thomasweise.github.io/programmingWithPython> (besucht am 2025-01-05) (siehe S. 349, 350).

References V



- [40] Kevin Wilson. *Python Made Easy*. Birmingham, England, UK: Packt Publishing Ltd, Aug. 2024. ISBN: 978-1-83664-615-0 (siehe S. 138–147, 349, 350).
- [41] Martin Yanev. *PyCharm Productivity and Debugging Techniques*. Birmingham, England, UK: Packt Publishing Ltd, Okt. 2022. ISBN: 978-1-83763-244-2 (siehe S. 350).



Glossary (in English) I

- 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.
- 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^{1,25,40}. Learn more about debugging in³⁹.
- 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 functions¹⁰. They must be delimited by `"""..."""`^{10,38}.
- 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`⁷ or tools such as `pytest`¹². Their output is compared to the text following the snippet in the docstring. If the output matches this text, the test succeeds. Otherwise it fails.
- Git** is a distributed Version Control Systems (VCS) which allows multiple users to work on the same code while preserving the history of the code changes^{29,35}. Learn more at <https://git-scm.com>.
- GitHub** is a website where software projects can be hosted and managed via the Git VCS^{24,35}. 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^{5,32}. Learn more at <https://www.apple.com/ios>.
- iPadOS** is the operating system that powers Apple iPads⁵. Learn more at <https://www.apple.com/ipados>.

Glossary (in English) II



macOS or Mac OS is the operating system that powers Apple Mac(intosh) computers^{26,32}. Learn more at <https://www.apple.com/macos>.

modulo division is, in Python, done by the operator `%` that computes the remainder of a division. $15 \% 6$ gives us `[3]`.

Mypy is a static type checking tool for Python¹⁶ that makes use of type hints. Learn more at <https://github.com/python/mypy> and in³⁹.

numerator The number a of a fraction $\frac{a}{b} \in \mathbb{Q}$ is called the *numerator*.

PyCharm is the convenient Python IDE that we recommend for this course^{36,40,41}. It comes in a free community 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^{6,13,21,23,40}. Learn more at <https://pytest.org>.

Python The Python programming language^{11,15,17,39}, i.e., what you will learn about in our book³⁹. Learn more at <https://python.org>.

signature The signature of a function refers to the parameters and their types, the return type, and the exceptions that the function can raise¹⁸. In Python, the function `signature` of the module `inspect` provides some information about the signature of a function³.

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^{14,37}. 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.



Glossary (in English) III

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^{2,20,22,23,27,34}. 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 code³⁵. 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.

Xcode offers the tools for developing, testing, and distributing applications as well as an IDE for Apple platforms such as macOS and iOS²⁸.

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^{4,8}. The golden ratio is approximately $\phi \approx 1.618\ 033\ 988\ 749\ 894\ 848\ 204\ 586\ 834$ ³⁰. Represented as `float` in Python, its value is `1.618033988749895`.

\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., $\dots, -3, -2, -1, 0, 1, 2, 3, \dots$, and so on. It holds that $\mathbb{Z} \subset \mathbb{R}$.