Brüche bearbeiten mit dem TTGO

Manchmal ist es angebracht, nicht mit Dezimalzahlen, sondern mit Brüchen zu arbeiten. Für diesen Zweck bietet Python die Klasse **Fraction** an. Leider steht diese - zumindest in meiner Firmware-Version - in Micropython nicht zur Verfügung. Glücklicherweise ist es nicht allzu aufwendig, eine solche Klasse zu programmieren. Nebenbei kann man beim Studium des Quellcodes vielleicht auch einiges dazulernen.

Beginnen wir mit der **Definition des Konstruktors:** Damit werden lediglich der Zähler *z* und der Nenner *n* zu einem Objekt zusammengefasst.

```
class Fraction():

    def __init__(self, z, n):
        self.z = z          # Zähler
        self.n = n          # Nenner
```

Mit der folgenden Methode kann man ein Fraction-Objekt in eine Zeichenkette der Form "z/n" umwandeln und damit auf dem Terminal ausgeben.

```
    def __str__(self):          # Fraction Object > String
        return str(self.z) + '/' + str(self.n)
```

Nun können wir schon ein Fraction-Objekt erzeugen und auch ausgeben:

```
b0 = Fraction(126,777)                # Bruch (Fraction-Objekt)
erzeugen
print(b0.__str__())
```

Nach dem Starten des Programms sehen wir im Terminal:

```
>>> %Run -c $EDITOR_CONTENT
126/777
```

Nun gilt es, Methoden zum Rechnen zu erstellen. Beginnen wir mit der **Addition:**

```
    def __add__(self, other):       # Addition
        z = self.z*other.n + other.z*self.n
        n = self.n * other.n
        return Fraction(z,n)
```

Der Code für die weiteren Rechenarten Subtraktion, Multiplikation und Division wird analog gebildet. Sie finden ihn auch weiter unten im Rahmen des Gesamtprogramms.

Wir wenden und jetzt lieber dem **Kürzen** zu. Sicherlich ist Ihnen schon aufgefallen, dass ich bei dem Code für die Addition nicht das kleinste gemeinsame Vielfache (kgV) der Nenner benutzt habe, sondern als gemeinsamen Nenner einfach das Produkt der beiden Nenner gewählt habe. Das ist recht einfach, hat aber den Nachteil, dass das Ergebnis (meist) nicht in gekürzter Form vorliegt. Einen Bruch können wir mit im Folgenden dargestellten **reduce**-Methode **kürzen;** dazu benutzen wir die Methode **gcd** (Greatest Common Divisor = größter gemeinsamer Teiler, kurz ggT):

```
    def gcd(self): # ggT (Greatest Common Divisor)
        a = abs(self.z)
        b = abs(self.n)
        while b:
            a, b = b, a%b
        return a

    def reduce(self): # Kürzen
        __gcd = self.gcd()
        self.z = self.z // __gcd
        self.n = self.n // __gcd
        if self.z < 0 and self.n <0:
            self.z = - self.z
            self.n = - self.n
        return self
```

Die Methode gcd greift auf den Euklidischen Algorithmus zurück (https://de.wikipedia.org/wiki/Euklidischer_Algorithmus); etwas transparenter wird die dort benutzte while-Schleife in der folgenden ausführlicheren (gleichwertigen) Form:

```
while b != 0:
    h = a % b   #   Rest der Division von a durch b
    a = b
    b = h
```

In der Kurzform nutzen wir auch aus, dass der Wert b = 0 von while als False gedeutet wird.

Jetzt können wir die Rechenfähigkeit des Programms einmal testen. Die Aufgabe lautet: 2/15 + 3/10; das Ergebnis soll in gekürzter Form angegeben werden. Der zugehörige Code lautet:

```
b1 = Fraction(2,15)
b2 = Fraction(3,10)

s = b1.__add__(b2)    # s. u.
s = s.reduce()        # Der Ergebnisbruch wird gekürzt.
print(b1.__str__() + ' + ' + b2.__str__() + ' = ' + s.__str__())
```

Auf dem Terminal sehen wir dann:

```
>>> %Run -c $EDITOR_CONTENT
2/15 + 3/10 = 13/30
```

Erwas gewöhnungsbedürftig ist die Asymmetrie bei der Addition: Der erste Summand ist hier der Besitzer der Methode __add__, der zweite Summand ein Parameter von __add__. Viel intuitiver wäre die Schreibweise

```
s = b1 + b2
```

Und tatsächlich ist diese Schreibweise auch möglich. Taucht zwischen zwei Objekten nämlich ein Plus-Zeichen auf, sucht Micropython automatisch im Quelltext nach eine Methode mit dem Namen __add__, setzt b1 und b2 entsprechend dort ein und übergibt den Rückgabewert an s. Die vereinfachende Schreibweise würde demnach nicht funktionieren, wenn wir unsere Additionsmethode z. B. __plus__ genannt hätten.

Für die anderen Rechenoperationen haben wir die zugehörigen Methoden ebenfalls so benannt, dass die Rechenzeichen -, * und / benutzt werden können. Auch der Methodenbezeichner __str__ für die Konvertierung eines Fraction-Objekts in eine Zeichenkette wurde ganz bewusst gewählt: Taucht ein solches Objekt nun in einem print-Befehl auf, so wird dieses Objekt automatisch mit Hilfe dieser Methode konvertiert.

Das gesamte Programm (Definition unserer Fraction-Klasse und Testbeispiele) sieht dann so aus:

```
class Fraction():

    def __init__(self, z, n):
        self.z = z          # Zähler
        self.n = n          # Nenner

    def __str__(self): # Fraction Object > String
        return str(self.z) + '/' + str(self.n)

    def gcd(self): # ggT (Greatest Common Divisor)
        a = abs(self.z)
        b = abs(self.n)
        while b:
            a, b = b, a%b
        return a

    def reduce(self): # Kürzen
        __gcd = self.gcd()
        self.z = self.z // __gcd
        self.n = self.n // __gcd
        if self.z < 0 and self.n <0:
            self.z = - self.z
            self.n = - self.n
        return self

    def __add__(self, other): # Addition
        z = self.z*other.n + other.z*self.n
        n = self.n * other.n
        return Fraction(z,n) # ggf. schon hier kürzen

    def __sub__(self, other): # Subtraktion
        z = self.z*other.n - other.z*self.n
        n = self.n * other.n
        return Fraction(z,n) # ggf. schon hier kürzen

    def __mul__(self, other): # Multplikation
        z = self.z * other.z
        n = self.n * other.n
        return Fraction(z,n) # ggf. schon hier kürzen

    def __truediv__(self, other): # Division
        z = self.z * other.n
        n = self.n * other.z
        return Fraction(z,n) # ggf. schon hier kürzen

# Testen...

b0 = Fraction(126,777)                  # Bruch (Fraction-Objekt) erzeugen
print(b0.__str__(), ' = ', end = '')    # b0 in String konvertieren und zusammen mit
                                        # '=' ausgeben
b0 = b0.reduce()                        # kürzen
print(b0)                               # Als Argument von print werden die
                                        # Fraction-Objekte als String ausgegeben
```

```
b1 = Fraction(2,15)
b2 = Fraction(3,10)

s = b1 + b2                             # entspricht s = b1.__add__(b2)
s = s.reduce()
print(b1.__str__() + ' + ' + b2.__str__() + ' = ' + s.__str__())

p = b1 * b2                             # entspricht p = b1.__mul__(b2)
p = p.reduce()
print(b1, '*', b2, '=', p)              # oder...
print('%s * %s = %s'  % (b1, b2, p))

d = b1 - b2                             # entspricht d = b1.__sub__(b2)
d = d.reduce()
print('%s - %s = %s'  % (b1, b2, d))

q = b1 / b2                             # entspricht q = b1.__div__(b2)
q = q.reduce()
print('%s : %s = %s'  % (b1, b2, q))
```

Im Terminal werden folgende Ergebnisse angezeigt:

```
>>> %Run -c $EDITOR_CONTENT
126/777 = 6/37
2/15 + 3/10 = 13/30
2/15 * 3/10 = 1/25
2/15 * 3/10 = 1/25
2/15 - 3/10 = -1/6
2/15 : 3/10 = 4/9
```

Im Übrigen können Sie auch mit negative Brüchen arbeiten, z. B. mit Fraction(-3, 7). Das steht für -3/7.

Im Anhang des Beitrags befindet sich das Modul fraction.py für die Fraction-Klasse. Um es einzusetzen, muss dieses Modul in den Flash-Speicher geladen werden, die Fraction-Klasse importiert werden: *from fraction import Fraction*.

Dieses Modul besitzt übrigens zusätzlich die Methode **mN_str**. Damit kann ein Bruch in eine gemischte Zahl (Zeichenkette) umgewandelt werden.