Lista 1

Maciej Karczewski

- Stworzyłem klase Fraction, która reprezentuje ułamiki zwykłe w postaci skróconej oraz je w takiej postaci przechowuje wraz z podstawowymi działaniami arytmerycznymi

```
    def __init__(self, num, den) :
        """Function creat and reduse fractions
        @parm num: (int) numerator
        @parm den: (int) denominator"""


        if type(num) == int and type(den) == int:
            if den == 0:
                raise ValueError("Denominator can't be 0")
            if den * num > 0:
                self.num = abs(num)
                self.den = abs(den)
            else :
                self.num = -abs(num)
                self.den = abs(den)

            # redusing fraction
            n = 2
            while n <= min(abs(self.num), abs(self.den)):
                if self.num % n == 0 and self.den % n == 0:
                    self.num = self.num // n
                    self.den = self.den // n
                else:
                    n += 1
        else :
            raise ValueError("Numerator and denominator must be intiger ")

    def __str__(self):
        if self.den == 1:
            return str(self.num)
        if self.mixed == True:
            return str(self.num // self.den) + " and " + str(self.num % self.den) + "/"
+str(self.den)
        return str(self.num) + '/' + str(self.den)

    def __add__(self, other):
        if type(other) != Fraction:
            other = Fraction(other, 1)

        return Fraction(self.num * other.den + other.num * self.den, self.den * other.den)

    def __radd__(self, other):
        return self + other

    def __sub__(self, other):
        if type(other) != Fraction:
            other = Fraction(other, 1)

        return Fraction(self.num * other.den - other.num * self.den, self.den * other.den)

    def __mul__(self,other):
        if type(other) != Fraction:
            return other * self
        return Fraction(self.num * other.num , self.den * other.den)

    def __rmul__(self, scalar):
        return Fraction(self.num * scalar, self.den)

    def __truediv__(self, other):
        if type(other) != Fraction:
            return Fraction(self.num, self.den * other)

        return Fraction(self.num * other.den , self.den * other.num)
```

In [2]:

```
f1 = Fraction(7,2)
f2 = Fraction(10,4)
print(f1)
print(f2)
```

```
7/2
5/2
```

In [3]:

```
f1 = Fraction(7,2)
f2 = Fraction(12,4)
f3 = Fraction(-2,5)
print(f1 - f2)
print(f1 * f2)
print(f1 / f2)
print(f1 * f3)
```

```
1/2
21/2
7/6
-7/5
```

- Gdy spróbuje się dać jako mianownik lub licznik liczbę inną niż typy Int wyskoczy błąd

In [4]:

```
Fraction.allow_float = False # o tym później
Fraction(2.1,1)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-4-3f9ae523161a> in <module>
      1 Fraction.allow_float = False # o tym później
----> 2 Fraction(2.1,1)

<ipython-input-1-d701155b7fe7> in __init__(self, num, den)
     42                     n += 1
     43         else :
---> 44             raise ValueError("Numerator and denominator must be intiger ")
     45 
     46     def get_num(self):

ValueError: Numerator and denominator must be intiger
```

In [5]:

```
Fraction.allow_float = False # o tym później
Fraction(2,1.5)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-5-ffc35d20ab23> in <module>
      1 Fraction.allow_float = False # o tym później
----> 2 Fraction(2,1.5)

<ipython-input-1-d701155b7fe7> in __init__(self, num, den)
     42                     n += 1
     43         else :
---> 44             raise ValueError("Numerator and denominator must be intiger ")
     45 
     46     def get_num(self):

ValueError: Numerator and denominator must be intiger
```

- Błąd wyskoczy także gdy spróbujemy wstawić do mianownika zero

In [6]:

```
Fraction(3,0)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-6-6ee63e0e64b3> in <module>
----> 1 Fraction(3,0)

<ipython-input-1-d701155b7fe7> in __init__(self, num, den)
     25         if type(num) == int and type(den) == int:
     26             if den == 0:
---> 27                 raise ValueError("Denominator can't be 0")
     28             if den * num > 0:
     29                 self.num = abs(num)

ValueError: Denominator can't be 0
```

- Dodałem możliwość porównania ułamków ze sobą (działa też dla ujemnych ułamków)

```
def __lt__(self, other):
      if type(other) != Fraction:
          if float(self) < other :
               return True
          else:
              return False

      if self.num / self.den < other.num / other.den :
          return True
      else:
          return False

  def __gt__(self, other):
      return not self < other

  def __eq__(self, other):
      if type(other) != Fraction:
          if math.isclose(float(self) , other) :
               return True
          else:
              return False

      if self.num == other.num and self.den == other.den:
          return True
      else:
          return False

  def __le__(self, other):
      if self.__lt__(other)  or self.__eq__(other):
          return True
      else:
          return False

  def __ge__(self, other):
      return  self > other or self == other
```

In [7]:

```
f1 = Fraction(2,3)
f2 = Fraction(5,2)
f3 = Fraction(4,-3)
print(f1 > f2)
print(f1 > f3)
print(f1 >= f2)
print(f1 < f2)
print(f1 <= f2)
print(f1 != f1)
print(f1 == f2)
```

```
False
True
False
True
True
False
False
```

- Stworzyłem metody get_num i get_den, które zwracają odpowiednio licznik i mianownik `def get_num(self):

  ```
   """Method return numerator"""
   return self.num
  ```

  def get_den(self):

  ```
   """Method return denominator"""
   return self.den
  ```

  `

In [8]:

```
f1 = Fraction(2,3)
print(f1.get_num())
print(f1.get_den())
```

```
2
3
```

Dodatkowo

- dodałem metode float która zwraca nam przybliżoną wartość tego ułamka

  ```
  def __float__(self):
      return self.num / self.den
  ```

In [9]:

```
f1 = Fraction(9,2)
print(float(f1))
```

```
4.5
```

- jeśli ułamek ma mianownik równy 1 to jest wypisywany tylko licznik

In [10]:

```
f1 = Fraction(-25, 5)
print(f1)
```

```
-5
```

- możemy wykonywać operacje arytmetyczne z Intami i porównywać z Intami lub floatami a jeśli zmienimy atrybut statyczny allow_float na True będziemy mogli także wykonywać operacje arytmetyczne także z floatami a same ułamki mogą być stworzone z liczb niecałkowitych

```
# Change float to ratio whe is allowed
if self.allow_float == True and type(num) in (float, int) and type(den) in (float, int):
    num = math.floor(num * 10 ** self.float_precision)
    den = math.floor(den * 10 ** self.float_precision)

    while num % 10 == 0 and den % 10 == 0:
        num = num // 10
        den = den // 10
```

In [11]:

```
Fraction.allow_float = True
f1 = Fraction(2.4,4)
f2 = Fraction(1.5,-3)
print(f1)
print(f2)
print(f1 > 3)
print(f1 > 0.1)
print(f2 + 2.5)
print(0.1 + f1)
print(f1 * 3)
print(3.14 * f2)
print(f1 / 2.73)
```

```
3/5
-1/2
False
True
2
7/10
9/5
-157/100
20/91
```

- Możemy też podnosić ułamek do potęgi

In [12]:

```
f1 = Fraction(7,14)
print(f1 ** 2)
print(f1 ** -2)
print(f1 ** f1)
```

```
1/4
4
5000/7071
```

- Jak tylko zmienimy atrybut statyczny mixed na True to wszystkie ułamki będą pokazywane w mieszanej postaci

In [13]:

```
Fraction.mixed = True
f1 = Fraction(8,3)
f2 = Fraction(5,2)
print(f1)
print(f2)
print(f1 + f2)
Fraction.mixed = False
```

```
2 and 2/3
2 and 1/2
5 and 1/6
```

- Możemy także zmienić precyzje ułamków z liczb zmiennoprzecinkowych zmieniając atrybut float_precision z 4 na inną liczę całkowitą ale im większa tym może wolniej działać o rząd wielkości

In [14]:

```
Fraction.float_precision = 4
print(Fraction(3.141592653, 2))
Fraction.float_precision = 7
print(Fraction(3.141592653, 2))
Fraction.float_precision = 4
```

```
6283/4000
15707963/10000000
```

Kod

https://github.com/maciejkar/Fraction_class.git

In [1]:

```
import math

class Fraction():
    """Class which represent fractions """

    mixed = False
    allow_float = False
    float_precision = 4

    def __init__(self, num, den) :
        """Function creat and reduse fractions
        @parm num: (int) numerator
        @parm den: (int) denominator"""

        # Change float to ratio whe is allowed
        if self.allow_float == True and type(num) in (float, int) and type(den) in (float, int):
            num = math.floor(num * 10 ** self.float_precision)
            den = math.floor(den * 10 ** self.float_precision)

            while num % 10 == 0 and den % 10 == 0:
                num = num // 10
                den = den // 10


        if type(num) == int and type(den) == int:
            if den == 0:
                raise ValueError("Denominator can't be 0")
            if den * num > 0:
                self.num = abs(num)
                self.den = abs(den)
            else :
                self.num = -abs(num)
                self.den = abs(den)

            # redusing fraction
            n = 2
            while n <= min(abs(self.num), abs(self.den)):
                if self.num % n == 0 and self.den % n == 0:
                    self.num = self.num // n
                    self.den = self.den // n
                else:
                    n += 1
        else :
            raise ValueError("Numerator and denominator must be intiger ")

    def get_num(self):
        """Method return numerator"""
        return self.num

    def get_den(self):
        """Method return denominator"""
        return self.den

    def __float__(self):
        return self.num / self.den

    def __str__(self):
        if self.den == 1:
            return str(self.num)
        if self.mixed == True:
            return str(self.num // self.den) + " and " + str(self.num % self.den) + "/" + str(self.den)
        return str(self.num) + '/' + str(self.den)

    def __add__(self, other):
        if type(other) != Fraction:
            other = Fraction(other, 1)

        return Fraction(self.num * other.den + other.num * self.den, self.den * other.den)

    def __radd__(self, other):
        return self + other

    def __sub__(self, other):
        if type(other) != Fraction:
            other = Fraction(other, 1)

        return Fraction(self.num * other.den - other.num * self.den, self.den * other.den)

    def __mul__(self,other):
        if type(other) != Fraction:
            return other * self
        return Fraction(self.num * other.num , self.den * other.den)

    def __rmul__(self, scalar):
        return Fraction(self.num * scalar, self.den)

    def __truediv__(self, other):
        if type(other) != Fraction:
            return Fraction(self.num, self.den * other)

        return Fraction(self.num * other.den , self.den * other.num)

    def __pow__(self, power):
        if type(power) != int:
            power = float(power)
        return Fraction(math.pow(self.num, power), math.pow(self.den, power))


    def __lt__(self, other):
        if type(other) != Fraction:
            if float(self) < other :
                 return True
            else:
                return False

        if self.num / self.den < other.num / other.den :
            return True
        else:
            return False

    def __gt__(self, other):
        return not self < other

    def __eq__(self, other):
        if type(other) != Fraction:
            if math.isclose(float(self) , other) :
                 return True
            else:
                return False

        if self.num == other.num and self.den == other.den:
            return True
        else:
            return False

    def __le__(self, other):
        if self.__lt__(other)  or self.__eq__(other):
            return True
        else:
            return False

    def __ge__(self, other):
        return  self > other or self == other
```

In [ ]:

```
```