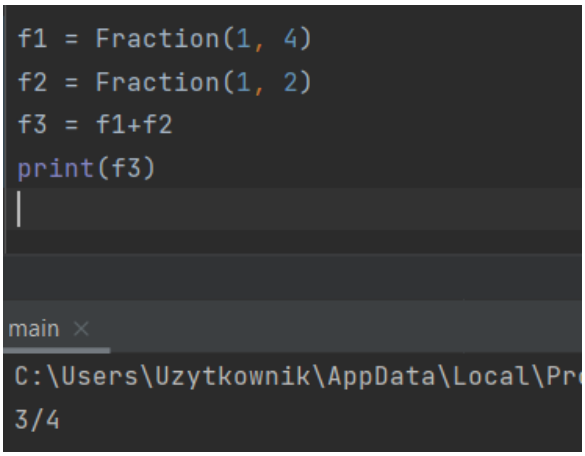
Julia Grzegorzewska, Wiktoria Fimińska
Link do repozytorium: https://github.com/grzesiaaa/Algorytmy_lista_1

RAPORT LISTA 1

Na początku opisałyśmy naszą klasę i jej atrybuty. Zastosowałyśmy wbudowane operatory *__add__*, *__sub__*, *__mul__* oraz *__truediv__* do zaimplementowania działań arytmetycznych. Poprzez nową zmienną *new_fraction* opisujemy ułamek po wykonaniu tych działań i wypisujemy ją. Poniżej przedstawiamy efekt:



W operatorze *__init__* sprawdzamy czy licznik oraz mianownik są liczbami całkowitymi - jeśli nie to wyskakuje błąd TypeError. Również zgłasza wyjątek ZeroDivisionError, gdy mianownik równa się 0.

```
def __init__(self, num: int, denom: int):
    if denom == 0:
        raise ZeroDivisionError("You can't divide by 0.")
    elif type(num) != int or type(denom) != int:
        raise TypeError("Numerator and denominator must be integers.")
```

```
f3 = Fraction(8, 0)

main
C:\Users\Uzytkownik\AppData\Local\Programs\Python\Python39\python.exe "C:/Users/Uzytkownik/Desktop/STUDIA/semest III/ALGORYTMY
Traceback (most recent call last):
  File "C:\Users\Uzytkownik\Desktop\STUDIA\semest III\ALGORYTMY\rozwiązania\Algorytmy_lista_1\main.py", line 134, in <module>
    f3 = Fraction(8, 0)
  File "C:\Users\Uzytkownik\Desktop\STUDIA\semest III\ALGORYTMY\rozwiązania\Algorytmy_lista_1\main.py", line 11, in __init__
    raise ZeroDivisionError("You can't divide by 0.")
ZeroDivisionError: You can't divide by 0.
```

Gdy mianownik jest liczbą ujemną minus naszego ułamka przechodzi na jego licznik. Gdy zarówno licznik, jak i mianownik są ujemne minusy są niwelowane.

```
if (self.num < 0 and self.denom < 0) or (self.num > 0 > self.denom):
    self.num *= -1
    self.denom *= -1
```

```
f1 = Fraction(-1, -4)
f2 = Fraction(1, -2)
print(f1)
print(f2)
```

```
main
C:\Users\Uzytkownik\AppData\Local\Programs\Python
1/4
-1/2
```

W naszym kodzie zastosowałyśmy też metodę, która modyfikuje i przechowuje ułamki w postaci nieskracalnej. Korzystałyśmy z biblioteki math, by znaleźć największy wspólny dzielnik (gcd) licznika i mianownika, a następnie podzieliłyśmy oba przez niego.

```
self.num = num
self.denom = denom
gcd = math.gcd(self.num, self.denom)
self.num //= gcd
self.denom //= gcd
```

Użyłyśmy operatora *__str__*, do wypisywania ułamków na ekran (gdy mianownik jest równy 1 to wypisujemy sam licznik, jeśli -1 to licznik pomnożony przez (-1)).

```
def __str__(self):
    """
    Return string representation of the fraction.
    """
    if self.denom == 1:
        return str(self.num)
    elif self.denom == -1:
        self.num *= -1
        return str(self.num)
    elif self.num == 0:
        return "0"
    return f"{self.num}/{self.denom}"
```

Aby porównywać ułamki skorzystałyśmy z operatorów *__lt__*, *__le__*, *__eq__* oraz *__ne__*. Nie stosowałyśmy metod *__gt__* oraz *__ge__*, ponieważ są one analogiczne do wcześniej wymienionych.

```
f1 = Fraction(1, 4)
f2 = Fraction(1, 2)
f3 = Fraction(8, 3)
print(f1 < f3)
print(f1 == f2)
print(f2 != f3)
```

```
main
C:\Users\Uzytkownik\AppData\Loc
True
False
True
```

Aby wypisywać mianownik lub licznik opisałyśmy metody returnujące każdy z nich.

```
def get_num(self):
    """
    Return numerator of the fraction.
    """
    return self.num

def get_den(self):
    """
    Return denominator of the fraction.
    """
    return self.denom
```

```
f3 = Fraction(8, 5)
print(f3.get_den())
print(f3.get_num())
```

```
main
C:\Users\Uzytkownik\AppData\Local\Pro
5
8
```

ZADANIA DODATKOWE

W zadaniach dodatkowych postanowiłyśmy przenieść warunki, które wcześniej były w *__init__* do kolejnych linijek jako osobne funkcje, aby kod był bardziej czytelny.

```
def __init__(self, num, denom):
    self.num = num
    self.denom = denom
    self.if_fraction()
    if self.denom == 0:
        raise ZeroDivisionError("You can't divide by 0.")
    else:
        self.if_float()
        self.reduce()
        self.customize()

def reduce(self):
    gcd = math.gcd(self.num, self.denom)
    self.num //= gcd
    self.denom //= gcd

def customize(self):
    if (self.num < 0 and self.denom < 0) or (self.num > 0 > self.denom):
        self.num *= -1
        self.denom *= -1
```

Aby można było określać czy ułamki >1 mają być podawane w postaci niewłaściwej, czy mieszanej, korzystamy ze @staticmethod.

```
@staticmethod
def mixed(opt):
    if opt == "False":
        Fraction.type = "simple"
    elif opt == "True":
        Fraction.type = "mixed"
```

```
f = Fraction(6, 5)
f1 = Fraction(-10, 3)
f2 = Fraction(1, 5)
print(f)
print(f1)
print(f2)
Fraction.mixed("True")
print("_")
print(f)
print(f1)
print(f2)
```

```
dodatkowe
C:\Users\Uzytkownik\AppData\Local\Prograr
6/5
-10/3
1/5
_
1(1/5)
-3(1/3)
1/5
```

Potrzebne było też zaktualizowanie metody *__str__*, w której określałyśmy jak mają być wyświetalne te ułamki w zależności od wybranego typu.

```
def __str__(self):
    if self.denom == 1:
        return str(self.num)
    elif self.denom == -1:
        self.num *= -1
        return str(self.num)
    elif self.num == 0:
        return "0"
    elif Fraction.type == "simple":
        return f"{self.num}/{self.denom}"
    elif Fraction.type == "mixed":
        if self.num > 0 and (self.num < self.denom):
            return f"{self.num}/{self.denom}"
        elif self.num < 0 and abs(self.num) > self.denom:
            int_number = self.num // self.denom + 1
            return f"{int_number}({-(self.num - int_number * self.denom)}/{self.denom})"
        else:
            int_number = self.num // self.denom
            return f"{int_number}({self.num - int_number * self.denom}/{self.denom})"
```

Naszym pomysłem była też możliwość tworzenia ułamka z innego ułamka (np. „Fraction(Fraction(1, 2), Franction(3 ,4)”). Udało nam się to osiągnąć dzięki funkcji „if_fraction”, która tworzy z podanych wewnątrz klasy ułamków nowe „obiekty’ (floaty).

```
def if_fraction(self):
    if isinstance(self.num, Fraction):
        self.num = self.num.num / self.num.denom
    if isinstance(self.denom, Fraction):
        self.denom = self.denom.num / self.denom.denom
```

```
f = Fraction(Fraction(10, 4), 5)
f1 = Fraction(-10, Fraction(Fraction(1, 2), 10))
print(-f)
print(1 + f1)
```

```
dodatkowe
C:\Users\Uzytkownik\AppData\Local\Programs\Python\Py
-1/2
-199
```

Aby móc również stosować ułamki dziesiętne opisałyśmy funkcję „if_float”, która sprawdza czy podany licznik bądź mianownik są floatami i następnie rozdziela część całkowita od dziesiętnej. Później sprawdza ile najwięcej miejsc po przecinku mają i mnoży licznik bądź mianownik przez odpowiednią potęgę 10.

```
def if_float(self):
    if type(self.num) == float and type(self.denom) == float:
        n = len(str(self.num).split('.')[1])
        m = len(str(self.denom).split('.')[1])
        if n >= m:
            self.num *= 10 ** n
            self.num = int(self.num)
            self.denom *= 10 ** n
            self.denom = int(self.denom)
        else:
            self.num *= 10 ** m
            self.num = int(self.num)
            self.denom *= 10 ** m
            self.denom = int(self.denom)
    elif type(self.num) == float and type(self.denom) == int:
        n = len(str(self.num).split('.')[1])
        self.num *= 10 ** n
        self.num = int(self.num)
        self.denom *= 10 ** n
    elif type(self.num) == int and type(self.denom) == float:
        m = len(str(self.denom).split('.')[1])
        self.num *= 10 ** m
        self.num = int(self.num)
        self.denom *= 10 ** m
        self.denom = int(self.denom)
```

Potrzebna była też aktualizacja metod działań arytmetycznych, aby móc nie tylko dodawać inne ułamki, ale także liczby całkowite. Konieczne było także dodanie prawostronnych działań, które umożliwiały dodawanie np. „1+f1”.

Aby możliwa była negacja podanego ułamka zastosowałyśmy metodę *__neg__*.

```
def __neg__(self):
    return Fraction(-self.num, self.denom)
```

Pomocne linki:

https://www.kodolamacz.pl/blog/wyzwanie-python-4-programowanie-obiektowe/

😊