Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное образовательное учреждение
высшего образования
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Департамент перспективной инженерии

ОТЧЕТ
ПО ЛАБОРАТОРНОЙ РАБОТЕ №2
дисциплины «Объектно-ориентированное программирование»

Вариант №1

Выполнила:
Беседина Инга Олеговна
3 курс, группа ИВТ-б-о-22-1,
09.03.01 «Информатика и
вычислительная техника», очная
форма обучения

_____
(подпись)

Проверил:
Воронкин Р. А., канд. технических
наук, доцент, доцент департамента
цифровых, робототехнических систем
и электроники института
перспективной инженерии

_____
(подпись)

Отчет защищен с оценкой _____ Дата защиты_____

Ставрополь, 2024 г.

Тема: Перегрузка операторов в языке Python

Цель: Приобретение навыков по перегрузке операторов при написании программ с помощью языка программирования Python версии 3.x.

Ход работы

Пример 1: *Изменить класс Rational из примера 1 лабораторной работы 4.1, используя перегрузку операторов.*

Пример 1:** *Рациональная (несократимая) дробь представляется парой целых чисел (a, b), где a — числитель, b — знаменатель. Создать класс Rational для работы с рациональными дробями. Обязательно должны быть реализованы операции:*

- сложения add, $(a, b) + (c, d) = (ad + be, bd)$;
- вычитания sub, $(a, b) - (c, d) = (ad - be, bd)$;
- умножения mul, $(a, b) \times (c, d) = (ac, bd)$;
- деления div, $(a, b)/(c, d) = (ad, be)$;
- сравнения equal, greate, less.

Должна быть реализована приватная функция сокращения дроби reduce, которая обязательно

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-


class Rational:

    def __init__(self, a=0, b=1) -> None:
        a = int(a)
        b = int(b)

        if b == 0:
            raise ValueError("Illegal value of the denominator")

        self.__numerator = a
        self.__denominator = b

        self.__reduce()

    def __reduce(self):
        # Функция для нахождения наибольшего общего делителя
        def gcd(a, b):
            if a == 0:
                return b
            elif b == 0:
                return a
            elif a >= b:
```

```
            return gcd(a % b, b)
        else:
            return gcd(a, b % a)

    sign = 1
    if (self.__numerator > 0 and self.__denominator < 0) or \
            (self.__numerator < 0 and self.__denominator > 0):
        sign = -1

    a, b = abs(self.__numerator), abs(self.__denominator)
    c = gcd(a, b)

    self.__numerator = sign * (a // c)
    self.__denominator = b // c

# Клонировать дробь
def __clone(self):
    return Rational(self.__numerator, self.__denominator)

@property
def numerator(self):
    return self.__numerator

@numerator.setter
def numerator(self, value):
    self.__numerator = int(value)
    self.__reduce()

@property
def denominator(self):
    return self.__denominator

@denominator.setter
def denominator(self, value):
    value = int(value)
    if value == 0:
        raise ValueError("Illegal value of the denominator")

    self.__denominator = value
    self.__reduce()

# Привести дробь к строке.
def __str__(self):
    return f"{self.__numerator} / {self.__denominator}"
```

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

    # Привести дробь к вещественному значению.
    def __float__(self):
        return self.__numerator / self.__denominator

    # Привести дробь к логическому значению.
    def __bool__(self):
        return self.__numerator != 0

    # Сложение обыкновенных дробей.
    def __iadd__(self, rhs):  # +=
        if isinstance(rhs, Rational):
            a = self.numerator * rhs.denominator + \
                self.denominator * rhs.numerator
            b = self.denominator * rhs.denominator

            self.__numerator, self.__denominator = a, b
            self.__reduce()
            return self
        else:
            raise ValueError("Illegal type of the argument")

    def __add__(self, rhs):  # +
        return self.__clone().__iadd__(rhs)

    # Вычитание обыкновенных дробей.
    def __isub__(self, rhs):  # -=
        if isinstance(rhs, Rational):
            a = self.numerator * rhs.denominator - \
                self.denominator * rhs.numerator
            b = self.denominator * rhs.denominator

            self.__numerator, self.__denominator = a, b
            self.__reduce()
            return self
        else:
            raise ValueError("Illegal type of the argument")

    def __sub__(self, rhs):  # -
        return self.__clone().__isub__(rhs)

    # Умножение обыкновенных дробей.
    def __imul__(self, rhs):  # *=
```

```
        if isinstance(rhs, Rational):
            a = self.numerator * rhs.numerator
            b = self.denominator * rhs.denominator

            self.__numerator, self.__denominator = a, b
            self.__reduce()
            return self
        else:
            raise ValueError("Illegal type of the argument")

    def __mul__(self, rhs):
        return self.__clone().__imul__(rhs)

    # Деление обыкновенных дробей.
    def __itruediv__(self, rhs): # /=
        if isinstance(rhs, Rational):
            a = self.numerator * rhs.denominator
            b = self.denominator * rhs.numerator

            if b == 0:
                raise ValueError("Illegal value of the denominator")

            self.__numerator, self.__denominator = a, b
            self.__reduce()
            return self
        else:
            raise ValueError("Illegal type of the argument")

    def __truediv__(self, rhs): # /
        return self.__clone().__itruediv__(rhs)

    # Отношение обыкновенных дробей.
    def __eq__(self, rhs):  # ==
        if isinstance(rhs, Rational):
            return (self.numerator == rhs.numerator) and \
                (self.denominator == rhs.denominator)
        else:
            return False

    def __ne__(self, rhs):  # !=
        if isinstance(rhs, Rational):
            return not self.__eq__(rhs)
        else:
            return False
```

```
    def __gt__(self, rhs):  # >
        if isinstance(rhs, Rational):
            return self.__float__() > rhs.__float__()
        else:
            return False

    def __lt__(self, rhs):  # <
        if isinstance(rhs, Rational):
            return self.__float__() < rhs.__float__()
        else:
            return False

    def __ge__(self, rhs):  # >=
        if isinstance(rhs, Rational):
            return not self.__lt__(rhs)
        else:
            return False

    def __le__(self, rhs):  # <=
        if isinstance(rhs, Rational):
            return not self.__gt__(rhs)
        else:
            return False


if __name__ == '__main__':
    r1 = Rational(3, 4)
    print(f"r1 = {r1}")

    r2 = Rational(5, 6)
    print(f"r2 = {r2}")

    print(f"r1 + r2 = {r1 + r2}")
    print(f"r1 - r2 = {r1 - r2}")
    print(f"r1 * r2 = {r1 * r2}")
    print(f"r1 / r2 = {r1 / r2}")

    print(f"r1 == r2: {r1 == r2}")
    print(f"r1 != r2: {r1 != r2}")
    print(f"r1 > r2: {r1 > r2}")
    print(f"r1 < r2: {r1 < r2}")
    print(f"r1 >= r2: {r1 >= r2}")
    print(f"r1 <= r2: {r1 <= r2}")
```

```
r1 = 3 / 4
r2 = 5 / 6
r1 + r2 = 19 / 12
r1 - r2 = -1 / 12
r1 * r2 = 5 / 8
r1 / r2 = 9 / 10
r1 == r2: False
r1 != r2: True
r1 > r2: False
r1 < r2: True
r1 >= r2: False
r1 <= r2: True
```

Рисунок 1. Результат выполнения программы

Индивидуальные задания:

Задание 1

Выполнить индивидуальное задание 1 лабораторной работы 4.1, максимально задействовав имеющиеся в Python средства перегрузки операторов.

2. Поле first — дробное число; поле second — дробное число, показатель степени.

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-


class Exponentiation:

    def __init__(self, first=0.0, second=0.0):
        first = float(first)
        second = float(second)

        self.__number = first
        self.__exponent = second

        if first < 0:
            raise ValueError

    # Клонировать выражение
    def __clone(self):
        return Exponentiation(self.__number, self.__exponent)

    @property
    def number(self):
        return self.__number

    @number.setter
```

```
    def number(self, value):
        if value >= 0:
            self.__number = float(value)
        else:
            raise ValueError("Illegal value")

    @property
    def exponent(self):
        return self.__exponent

    @exponent.setter
    def exponent(self, value):
        self.__exponent = float(value)

    def power(self):
        return self.number**self.exponent

    # Привести выражение к строке
    def __str__(self):
        return f"{self.__number}^{self.__exponent}"

    def __repr__(self) -> str:
        return self.__str__()

    # Привести выражение к логическому значению.
    def __bool__(self):
        return self.__number != 0

    # перемножение степеней с одинаковыми основаниями
    def __imul__(self, rhs): # *=
        if isinstance(rhs, Exponentiation):
            a = self.exponent + rhs.exponent

            self.__exponent = a
            return self
        else:
            raise ValueError("Illegal type of the argument")

    def __mul__(self, rhs): # *
        return self.__clone().__imul__(rhs)

    # деление степеней с одинаковыми основаниями
    def __itruediv__(self, rhs): # /=
        if isinstance(rhs, Exponentiation):
            a = self.exponent - rhs.exponent
```

```
            self.__exponent = a
            return self
        else:
            raise ValueError("Illegal type of the argument")

    def __truediv__(self, rhs): # /
        return self.__clone().__itruediv__(rhs)

    def __eq__(self, rhs):  # ==
        if isinstance(rhs, Exponentiation):
            return self.power() == rhs.power()
        else:
            return False

    def __ne__(self, rhs):  # !=
        if isinstance(rhs, Exponentiation):
            return not self.__eq__(rhs)
        else:
            return False

    def __gt__(self, rhs):  # >
        if isinstance(rhs, Exponentiation):
            return self.power() > rhs.power()
        else:
            return False

    def __lt__(self, rhs):  # <
        if isinstance(rhs, Exponentiation):
            return self.power() < rhs.power()
        else:
            return False

    def __ge__(self, rhs):  # >=
        if isinstance(rhs, Exponentiation):
            return not self.__lt__(rhs)
        else:
            return False

    def __le__(self, rhs):  # <=
        if isinstance(rhs, Exponentiation):
            return not self.__gt__(rhs)
        else:
            return False
```

```
if __name__ == '__main__':
    e1 = Exponentiation(0.5, 0.2)
    print(f"e1 = {e1}")

    e2 = Exponentiation(0.5, 1.3)
    print(f"e2 = {e2}")

    print(f"e1 * e2 = {e1 * e2}")
    print(f"e1 / e2 = {e1 / e2}")


    print(f"e1 == e2: {e1 == e2}")
    print(f"e1 != e2: {e1 != e2}")
    print(f"e1 > e2: {e1 > e2}")
    print(f"e1 < e2: {e1 < e2}")
    print(f"e1 >= e2: {e1 >= e2}")
    print(f"e1 <= e2: {e1 <= e2}")
```

```
e1 = 0.5^0.2
e2 = 0.5^1.3
e1 * e2 = 0.5^1.5
e1 / e2 = 0.5^-1.1
e1 == e2: False
e1 != e2: True
e1 > e2: True
e1 < e2: False
e1 >= e2: True
e1 <= e2: False
```

Рисунок 2. Результат выполнения программы

Дополнительно к требуемым в заданиях операциям перегрузить операцию индексирования []. Максимально возможный размер списка задать константой. В отдельном поле size должно храниться максимальное для данного объекта количество элементов списка; реализовать метод size(), возвращающий установленную длину. Если количество элементов списка изменяется во время работы, определить в классе поле count. Первоначальные значения size и count устанавливаются конструктором.

В тех задачах, где возможно, реализовать конструктор инициализации строкой.

1. Создать класс BitString для работы с битовыми строками не более чем из 100 бит. Битовая строка должна быть представлена списком типа int, каждый элемент которого принимает значение 0 или 1. Реальный размер списка задается как аргумент конструктора инициализации. Должны быть реализованы все традиционные операции для работы с битовыми строками: and, or, xor, not. Реализовать сдвиг влево и сдвиг вправо на заданное количество битов.

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
```

```
class BitString:
    MAX_SIZE = 100  # Максимально возможный размер битовой строки

    def __init__(self, bit_string: str = '0'):
        """Инициализация битовой строки."""
        if len(bit_string) > self.MAX_SIZE:
            raise ValueError(f"Битовая строка не может превышать {
                               self.MAX_SIZE} бит.")

        # Преобразуем строку в список бит
        self.bits = [int(bit) for bit in bit_string if bit in '01']
        self.count = len(self.bits)

    def __getitem__(self, index: int) -> int:
        """Индексирование для получения бита по индексу."""
        if index < 0 or index >= self.count:
            raise IndexError("Индекс вне диапазона.")
        return self.bits[index]

    def __setitem__(self, index: int, value: int):
        """Индексирование для установки бита по индексу."""
        if index < 0 or index >= self.count:
            raise IndexError("Индекс вне диапазона.")
        if value not in (0, 1):
            raise ValueError("Значение должно быть 0 или 1.")

        self.bits[index] = value

    def size(self) -> int:
        """Возвращает текущий размер битовой строки."""
        return self.count

    def and_op(self, other):
        """Операция AND с другой битовой строкой."""
        min_size = min(self.count, other.count)
        result = [self.bits[i] & other.bits[i] for i in range(min_size)]
        return BitString(''.join(map(str, result)))

    def or_op(self, other):
        """Операция OR с другой битовой строкой."""
        min_size = min(self.count, other.count)
        result = [self.bits[i] | other.bits[i] for i in range(min_size)]
        return BitString(''.join(map(str, result)))
```

```
    def xor_op(self, other):
        """Операция XOR с другой битовой строкой."""
        min_size = min(self.count, other.count)
        result = [self.bits[i] ^ other.bits[i] for i in range(min_size)]
        return BitString(''.join(map(str, result)))

    def not_op(self):
        """Операция NOT для текущей битовой строки."""
        result = [1 - bit for bit in self.bits]
        return BitString(''.join(map(str, result)))

    def shift_left(self, n: int):
        """Сдвиг влево на n битов."""
        if n < 0:
            raise ValueError(
                "Количество битов для сдвига не должно быть
отрицательным.")

        new_bits = self.bits[n:] + [0] * n
        return BitString(''.join(map(str, new_bits)))

    def shift_right(self, n: int):
        """Сдвиг вправо на n битов."""
        if n < 0:
            raise ValueError(
                "Количество битов для сдвига не должно быть
отрицательным.")

        new_bits = [0] * n + self.bits[:-
                                       n] if n < self.count else [0] *
self.count
        return BitString(''.join(map(str, new_bits)))

    def __repr__(self) -> str:
        """Строковое представление битовой строки."""
        return ''.join(map(str, self.bits))


if __name__ == '__main__':
    bit_str1 = BitString('1101')
    bit_str2 = BitString('1011')

    print("Битовая строка 1:", bit_str1)
    print("Битовая строка 2:", bit_str2)
```

```
# Операции
print("AND:", bit_str1.and_op(bit_str2))
print("OR:", bit_str1.or_op(bit_str2))
print("XOR:", bit_str1.xor_op(bit_str2))
print("NOT Битовой строки 1:", bit_str1.not_op())

# Сдвиги
print("Сдвиг влево на 2:", bit_str1.shift_left(2))
print("Сдвиг вправо на 2:", bit_str1.shift_right(2))

# Индексирование
print("Бит по индексу 2 в Битовой строке 1:", bit_str1[2])
bit_str1[2] = 0
print("После изменения бита по индексу 2:", bit_str1)
```

```
Битовая строка 1: 1101
Битовая строка 2: 1011
AND: 1001
OR: 1111
XOR: 0110
NOT Битовой строки 1: 0010
Сдвиг влево на 2: 0100
Сдвиг вправо на 2: 0011
Бит по индексу 2 в Битовой строке 1: 0
После изменения бита по индексу 2: 1101
```

Рисунок 3. Результат выполнения программы

Контрольные вопросы:

1. В Python перегрузка операций осуществляется с помощью специальных методов, известных как "магические методы" или "методы с двойным подчеркиванием". Эти методы позволяют определять поведение объектов при использовании стандартных операторов (например, +, -, *, == и т.д.)

2. Арифметические операции:

__add__(self, other) — для +

__sub__(self, other) — для -

__mul__(self, other) — для *

__truediv__(self, other) — для /

__floordiv__(self, other) — для //

__mod__(self, other) — для %

__pow__(self, other) — для **

Операции отношения:

__eq__(self, other) — для ==

__ne__(self, other) — для !=

__lt__(self, other) — для <

__le__(self, other) — для <=

__gt__(self, other) — для >

__ge__(self, other) — для >=

3. Вызов методов: add, iadd и radd:

__add__: вызывается при использовании оператора +;

__iadd__: вызывается при использовании оператора +=

__radd__: вызывается, если левый операнд не поддерживает операцию и правый операнд имеет метод __add__

4. __new__: это метод класса, который отвечает за создание нового экземпляра класса. Он вызывается перед методом __init__ и должен возвращать новый объект. __init__: это метод экземпляра, который инициализирует уже созданный объект. Он не создает объект, а настраивает его

5. __str__: предназначен для создания "читаемого" представления объекта. Этот метод вызывается функцией str() или при использовании функции print(). __repr__: предназначен для создания "официального" представления объекта, которое должно быть как можно более точным и полным. Этот метод вызывается функцией repr() и в интерактивной оболочке Python.

Вывод: В ходе выполнения лабораторной работы были приобретены навыки по перегрузке операторов при написании программ с помощью языка программирования Python версии 3.x.