МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ

ФЕДЕРАЦИИ

Федеральное государственное автономное

образовательное учреждение высшего образования

«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Кафедра инфокоммуникаций

«Перегрузка операторов в языке Python»

Отчет по лабораторной работе № 4.2

по дисциплине «Основы программной инженерии»

Выполнил студент группы ПИЖ-б-о-21-1

Кучеренко С. Ю. _____« » 2023г.

Подпись студента_____

Работа защищена « »_____20__г.

Проверил Воронкин Р.А. _____

(подпись)

Ставрополь 2023

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

Ход работы:

1. Изучить теоретический материал работы.

2. Создать общедоступный репозиторий на GitHub, в котором будет использована лицензия MIT и язык программирования Python.

3. Выполните клонирование созданного репозитория.

4. Дополните файл .gitignore необходимыми правилами для работы с IDE PyCharm.

5. Организуйте свой репозиторий в соответствие с моделью ветвления git-flow.

6. Создайте проект PyCharm в папке репозитория.

7. Проработайте примеры лабораторной работы.

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


import math


class Vector2D:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __repr__(self):
        return 'Vector2D({}, {})'.format(self.x, self.y)

    def __str__(self):
        return '({}, {})'.format(self.x, self.y)

    def __add__(self, other):
        return Vector2D(self.x + other.x, self.y + other.y)

    def __iadd__(self, other):
        self.x += other.x
        self.y += other.y
        return self

    def __sub__(self, other):
        return Vector2D(self.x - other.x, self.y - other.y)

    def __isub__(self, other):
        self.x -= other.x
        self.y -= other.y
        return self

    def __abs__(self):
```

```
        return math.hypot(self.x, self.y)

    def __bool__(self):
        return self.x != 0 or self.y != 0

    def __neg__(self):
        return Vector2D(-self.x, -self.y)


if __name__ == '__main__':
    x = Vector2D(3, 4)
    print(x)
    print(abs(x))
    y = Vector2D(5, 6)
    print(y)
    print(x + y)
    print(x - y)
    print(-x)
    x += y
    print(x)
    print(bool(x))
    z = Vector2D(0, 0)
    print(bool(z))
    print(-z)
```

```
(3, 4)
5.0
(5, 6)
(8, 10)
(-2, -2)
(-3, -4)
(8, 10)
True
False
(0, 0)
```

Рисунок 1 – Результат выполнения программы

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


class Rational:
    def __init__(self, a=0, b=1):
        a = int(a)
        b = int(b)
        if b == 0:
            raise ValueError("Illegal value of the denominator")
        self.__numerator = a
        self.__denominator = b
        self.__reduce()

    # Сокращение дроби.
```

```
    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 > self.__denominator) or \
                (self.__numerator < 0 < self.__denominator):
            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):
        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

Process finished with exit code 0
```

Рисунок 2 – Результат выполнения программы

8. Выполните индивидуальные задания. Приведите в отчете скриншоты работы программ решения индивидуального задания.

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

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

Поле first — целое положительное число, часы; поле second — целое
положительное
число, минуты. Реализовать метод minutes() — приведение времени в минуты.
"""


class Conversion:
    def __init__(self, hours, minutes):
        self.hours = hours
        self.minutes = minutes

    # строковое представление времени
    def __str__(self):
        return f"{self.hours:02d}:{self.minutes:02d}"

    # сумма времен
    def __add__(self, other):
        total_minutes = self.minutes + other.minutes
        carry_hours = total_minutes // 60
        total_minutes %= 60
```

```
        total_hours = self.hours + other.hours + carry_hours
        return Conversion(total_hours, total_minutes)

    # разность времен
    def __sub__(self, other):
        total_minutes = (self.hours * 60 + self.minutes) - (other.hours * 60 +
other.minutes)
        if total_minutes < 0:
            raise ValueError("Результат отрицательный")
        total_hours = total_minutes // 60
        total_minutes %= 60
        return Conversion(total_hours, total_minutes)

    # сравнение времен
    def __lt__(self, other):
        return self.minutes + self.hours * 60 < other.minutes + other.hours *
60

    # время в минутах
    def get_minutes(self):
        return self.hours * 60 + self.minutes


if __name__ == "__main__":
    time1 = Conversion(2, 30)
    time2 = Conversion(1, 45)

    print(time1)
    print(time2)

    sum_time = time1 + time2
    print(sum_time)

    diff_time = time1 - time2
    print(diff_time)

    print(time1 < time2)

    print(time1.get_minutes())
    print(time2.get_minutes())
```

```
02:30
01:45
04:15
00:45
False
150
105

Process finished with exit code 0
```

Рисунок 3 – Результат выполнения программы

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

"""
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.

Карточка иностранного слова представляет собой словарейу, содержащую
иностранное
слово и его перевод. Для моделирования электронного словаря иностранных слов
реализовать класс Dictionary. Данный класс имеет поле-название словаря и
содержит
список словарей WordCard, представляющих собой карточки иностранного слова.
Название
словаря задается при создании нового словаря, но должна быть предоставлена
возможность его изменения во время работы. Карточки добавляются в словарь и
удаляются
из него. Реализовать поиск определенного слова как отдельный метод. Аргументом
операции индексирования должно быть иностранное слово. В словаре не должно
быть
карточек-дублей. Реализовать операции объединения, пересечения и вычитания
словарей.
При реализации должен создаваться новый словарь, а исходные словари не должны
изменяться. При объединении новый словарь должен содержать без повторений все
слова,
содержащиеся в обоих словарях-операндах. При пересечении новый словарь должен
состоять только из тех слов, которые имеются в обоих словарях-операндах. При
вычитании
новый словарь должен содержать слова первого словаря-операнда, отсутствующие
во
втором.
"""


class WordCard:
    def __init__(self, foreign_word, translation):
        self.foreign_word = foreign_word
        self.translation = translation

    def __str__(self):
        return f"{self.foreign_word}: {self.translation}"


class Dictionary:
    def __init__(self, name):
        self.name = name
        self.cards = []

    def __str__(self):
        return f"Dictionary: {self.name}"

    # перегрузка оператора индексирования []
    def __getitem__(self, foreign_word):
        for card in self.cards:
```

```
            if card.foreign_word == foreign_word:
                return card
        raise KeyError(f"Word '{foreign_word}' not found in the dictionary.")

    # метод для добавления новой карточки в словарь
    def add_card(self, card):
        if card.foreign_word not in [c.foreign_word for c in self.cards]:
            self.cards.append(card)

    # метод для удаления карточки из словаря по иностранному слову
    def remove_card(self, foreign_word):
        self.cards = [card for card in self.cards if card.foreign_word !=
foreign_word]

    # метод для поиска определенного слова в словаре
    def search_word(self, foreign_word):
        for card in self.cards:
            if card.foreign_word == foreign_word:
                return card
        return None

    # метод для объединения словарей
    def union(self, other_dict):
        new_dict = Dictionary(f"{self.name} + {other_dict.name}")
        new_dict.cards = self.cards.copy()
        for card in other_dict.cards:
            if card.foreign_word not in [c.foreign_word for c in
new_dict.cards]:
                new_dict.cards.append(card)
        return new_dict

    # метод добавления совпадающих слов из 1 словаря во 2
    def intersection(self, other_dict):
        new_dict = Dictionary(f"{self.name} ∩ {other_dict.name}")
        for card in self.cards:
            if card.foreign_word in [c.foreign_word for c in
other_dict.cards]:
                new_dict.cards.append(card)
        return new_dict

    # метод добавления несовпадающих слов из 1 словаря во 2
    def difference(self, other_dict):
        new_dict = Dictionary(f"{self.name} - {other_dict.name}")
        for card in self.cards:
            if card.foreign_word not in [c.foreign_word for c in
other_dict.cards]:
                new_dict.cards.append(card)
        return new_dict

if __name__ == "__main__":
    card1 = WordCard("hello", "привет")
    card2 = WordCard("goodbye", "пока")
    card3 = WordCard("cat", "кошка")
    card4 = WordCard("dog", "собака")

    dict1 = Dictionary("English-Russian Dictionary")
    dict1.add_card(card1)
    dict1.add_card(card2)

    dict2 = Dictionary("Russian-English Dictionary")
    dict2.add_card(WordCard("привет", "hello"))
    dict2.add_card(WordCard("пока", "goodbye"))

    print(dict1["hello"])
```

```python
dict1.remove_card("goodbye")

print(dict1.search_word("goodbye"))

dict3 = dict1.union(dict2)
for card in dict3.cards:
    print(card)

dict4 = dict1.intersection(dict2)
for card in dict4.cards:
    print(card)

dict5 = dict1.difference(dict2)
for card in dict5.cards:
    print(card)
```

```
hello: привет
None
hello: привет
привет: hello
пока: goodbye
hello: привет

Process finished with exit code 0
```

Рисунок 4 – Результат выполнения программы

9. Зафиксируйте сделанные изменения в репозитории.
10. Выполните слияние ветки для разработки с веткой main / master.
11. Отправьте сделанные изменения на сервер GitHub.

Контрольные вопросы:

1. Какие средства существуют в Python для перегрузки операций?

Перегрузка операторов — один из способов реализации полиморфизма, когда мы можем задать свою реализацию какого-либо метода в своём классе.

Например, у нас есть два класса:

```
class A:
    def go(self):
        print('Go, A!')

class B(A):
    def go(self, name):
        print('Go, {}!'.format(name))
```

В данном примере класс *B* наследует класс *A*, но переопределяет метод *go*, поэтому он имеет мало общего с аналогичным методом класса *A*.

Однако в python имеются методы, которые, как правило, не вызываются напрямую, а вызываются встроенными функциями или операторами.

Например, метод `__init__` перегружает конструктор класса. Конструктор - создание экземпляра класса.

2. Какие существуют методы для перегрузки арифметических операций и операций отношения в языке Python?

- `__add__(self, other)` - сложение. x + y вызывает `x.__add__(y)`.
- `__sub__(self, other)` - вычитание (x - y).
- `__mul__(self, other)` - умножение (x * y).
- `__truediv__(self, other)` - деление (x / y).
- `__floordiv__(self, other)` - целочисленное деление (x // y).
- `__mod__(self, other)` - остаток от деления (x % y).
- `__divmod__(self, other)` - частное и остаток (divmod(x, y)).
- `__pow__(self, other[, modulo])` - возведение в степень (`x ** y`, `pow(x, y[, modulo])`).
- `__lshift__(self, other)` - битовый сдвиг влево (x << y).
- `__rshift__(self, other)` - битовый сдвиг вправо (x >> y).
- `__and__(self, other)` - битовое И (x & y).
- `__xor__(self, other)` - битовое ИСКЛЮЧАЮЩЕЕ ИЛИ (x ^ y).
- `__or__(self, other)` - битовое ИЛИ (x | y).

3. В каких случаях будут вызваны следующие методы: __add__ , __iadd__ и __radd__ ? Приведите примеры.

Например, операция x + y будет сначала пытаться вызвать `x.__add__(y)`, и только в том случае, если это не получилось, будет пытаться вызвать `y.__radd__(x)`. Аналогично для остальных методов.