

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

Институт перспективной инженерии  
Департамент цифровых, робототехнических систем и электроники

**ОТЧЕТ**  
**ПО ЛАБОРАТОРНОЙ РАБОТЕ №2**  
**дисциплины**  
**«Объектно-ориентированное программирование»**  
**Вариант 1**

Выполнил:  
Бабенко Артём Тимофеевич  
3 курс, группа ИВТ-б-о-22-1,  
09.03.01 «Информатика и  
вычислительная техника»,  
направленность (профиль)  
«Программное обеспечение средств  
вычислительной техники и  
автоматизированных систем», очная  
форма обучения

---

(подпись)

Проверил:  
Ассистент департамента цифровых,  
робототехнических систем и  
электроники Богданов С.С

---

(подпись)

Отчет защищен с оценкой \_\_\_\_\_ Дата защиты \_\_\_\_\_

Ставрополь, 2024 г.

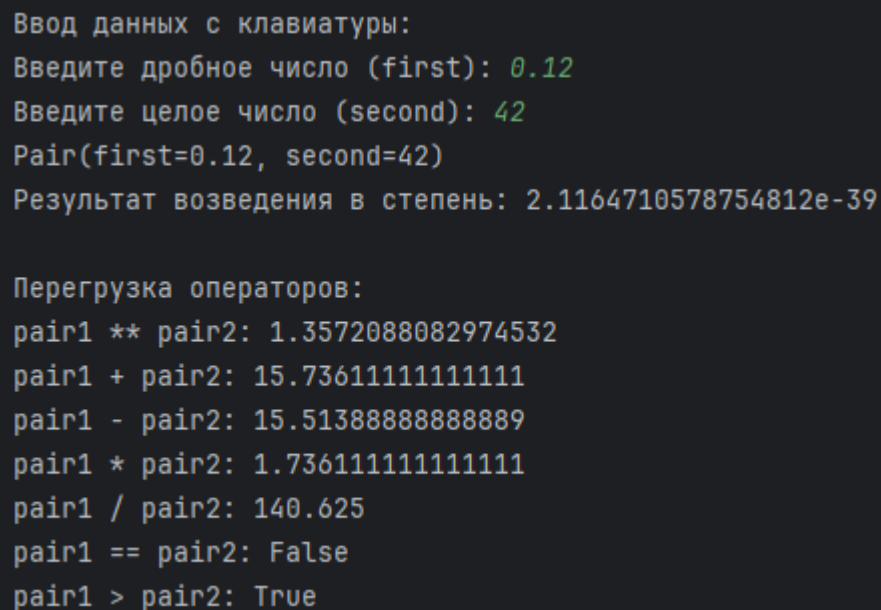
Тема: Перегрузка операторов в языке Python

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

Порядок выполнения работы:

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

Результат работы программы:



```
Ввод данных с клавиатуры:
Введите дробное число (first): 0.12
Введите целое число (second): 42
Pair(first=0.12, second=42)
Результат возведения в степень: 2.1164710578754812e-39

Перегрузка операторов:
pair1 ** pair2: 1.3572088082974532
pair1 + pair2: 15.736111111111111
pair1 - pair2: 15.513888888888889
pair1 * pair2: 1.7361111111111111
pair1 / pair2: 140.625
pair1 == pair2: False
pair1 > pair2: True
```

Рисунок 1 – Результат работы программы

Код программы:

```
class Pair:
    def __init__(self, first: float, second: int):
        """
        Инициализация объекта Pair.
        :param first: дробное число (float).
        :param second: целое число (int), показатель степени.
        """

    if not isinstance(first, (float, int)):
        raise ValueError("Поле 'first' должно быть дробным числом.")
    if not isinstance(second, int):
        raise ValueError("Поле 'second' должно быть целым числом.")
```

```
self.first = float(first) # Преобразуем в float для единообразия
self.second = second
```

```
def power(self):
    """
    Возведение числа first в степень second.
    :return: результат возведения в степень.
    """
    return self.first ** self.second
```

```
def read(self):
    """
    Ввод значений полей с клавиатуры.
    """
    try:
        self.first = float(input("Введите дробное число (first): "))
        self.second = int(input("Введите целое число (second): "))
    except ValueError:
        print("Ошибка ввода. Проверьте, что введены корректные значения.")
        raise
```

```
def display(self):
    """
    Вывод значений полей на экран.
    """
    print(f"Pair(first={self.first}, second={self.second})")
```

```
# Перегрузка оператора возведения в степень (**)
```

```
def __pow__(self, other):
    if isinstance(other, Pair):
        return self.power() ** other.power()
    elif isinstance(other, (int, float)):
        return self.power() ** other
    else:
```

```

        raise TypeError("Операнд должен быть числом или объектом класса Pair.")

# Перегрузка операторов сравнения
def __eq__(self, other):
    if isinstance(other, Pair):
        return self.power() == other.power()
    elif isinstance(other, (int, float)):
        return self.power() == other
    else:
        raise TypeError("Операнд должен быть числом или объектом класса Pair.")

def __ne__(self, other):
    return not self.__eq__(other)

def __lt__(self, other):
    if isinstance(other, Pair):
        return self.power() < other.power()
    elif isinstance(other, (int, float)):
        return self.power() < other
    else:
        raise TypeError("Операнд должен быть числом или объектом класса Pair.")

def __le__(self, other):
    return self.__lt__(other) or self.__eq__(other)

def __gt__(self, other):
    return not self.__le__(other)

def __ge__(self, other):
    return not self.__lt__(other)

# Перегрузка арифметических операторов
def __add__(self, other):
    if isinstance(other, Pair):
        return self.power() + other.power()

```

```
elif isinstance(other, (int, float)):
    return self.power() + other
else:
    raise TypeError("Операнд должен быть числом или объектом класса Pair.")
```

```
def __sub__(self, other):
    if isinstance(other, Pair):
        return self.power() - other.power()
    elif isinstance(other, (int, float)):
        return self.power() - other
    else:
        raise TypeError("Операнд должен быть числом или объектом класса Pair.")
```

```
def __mul__(self, other):
    if isinstance(other, Pair):
        return self.power() * other.power()
    elif isinstance(other, (int, float)):
        return self.power() * other
    else:
        raise TypeError("Операнд должен быть числом или объектом класса Pair.")
```

```
def __truediv__(self, other):
    if isinstance(other, Pair):
        return self.power() / other.power()
    elif isinstance(other, (int, float)):
        return self.power() / other
    else:
        raise TypeError("Операнд должен быть числом или объектом класса Pair.")
```

```
def make_pair(first: float, second: int) -> Pair:
```

```
    """
```

```
    Создание объекта Pair с проверкой параметров.
```

```
    :param first: дробное число.
```

```
    :param second: целое число.
```

```

:return: объект Pair.
"""

try:
    return Pair(first, second)
except ValueError as e:
    print(f"Ошибка создания Pair: {e}")
    exit(1)

if __name__ == '__main__':
    # Демонстрация работы класса Pair
    print("Создание объекта Pair через конструктор:")
    pair1 = Pair(2.5, 3)
    pair1.display()
    print(f"Результат возведения в степень: {pair1.power()}")

    print("\nСоздание объекта Pair через функцию make_pair():")
    pair2 = make_pair(3.0, -2)
    pair2.display()
    print(f"Результат возведения в степень: {pair2.power()}")

    print("\nВвод данных с клавиатуры:")
    pair3 = Pair(0, 0) # Создаем пустой объект
    try:
        pair3.read()
        pair3.display()
        print(f"Результат возведения в степень: {pair3.power()}")
    except ValueError:
        print("Ошибка при вводе данных.")

    print("\nПерегрузка операторов:")
    print(f"pair1 ** pair2: {pair1 ** pair2}") # Возведение в степень
    print(f"pair1 + pair2: {pair1 + pair2}") # Сложение
    print(f"pair1 - pair2: {pair1 - pair2}") # Вычитание
    print(f"pair1 * pair2: {pair1 * pair2}") # Умножение

```

```
print(f"pair1 / pair2: {pair1 / pair2}") # Деление
print(f"pair1 == pair2: {pair1 == pair2}") # Сравнение
print(f"pair1 > pair2: {pair1 > pair2}") # Сравнение
```

Задание 2. . 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.

Результат работы программы:

```
bs1: 10101010
bs2: 11001100
AND (bs1 & bs2): 10001000
OR (bs1 | bs2): 11101110
XOR (bs1 ^ bs2): 01100110
NOT (~bs1): 01010101
Shift Left (bs1 << 2): 10101000
Shift Right (bs1 >> 2): 00101010
bs1[3]: 0
bs1[1:5]: [0, 1, 0, 1]
Size of bs1: 8
Count of set bits in bs1: 4
Initial bs3: 00000000
```

Рисунок 2 – Результат работы программы

Код программы:

```
class BitString:
```

```
    MAX_SIZE = 100 # Максимально возможный размер битовой строки
```

```
    def __init__(self, size=MAX_SIZE, initial_value=None):
```

```
        """
```

```
        Конструктор класса.
```

```
        :param size: Размер битовой строки (не более MAX_SIZE).
```

```
        :param initial_value: Начальное значение битовой строки (список или строка).
```

```

"""
if size <= 0 or size > self.MAX_SIZE:
    raise ValueError(f"Размер должен быть в диапазоне от 1 до
{self.MAX_SIZE}")

self.size_ = size # Максимальный размер для данного объекта
self.bits = [0] * size # Инициализация списка нулями
self.count = 0 # Текущее количество установленных битов

if initial_value is not None:
    if isinstance(initial_value, str): # Если начальное значение - строка
        if len(initial_value) > size:
            raise ValueError("Длина строки превышает заданный размер")
        for i, char in enumerate(initial_value):
            if char not in ('0', '1'):
                raise ValueError("Строка должна содержать только символы '0' и '1'")
            self.bits[i] = int(char)
            if char == '1':
                self.count += 1
    elif isinstance(initial_value, list): # Если начальное значение - список
        if len(initial_value) > size:
            raise ValueError("Длина списка превышает заданный размер")
        for i, bit in enumerate(initial_value):
            if bit not in (0, 1):
                raise ValueError("Список должен содержать только значения 0 и 1")
            self.bits[i] = bit
            if bit == 1:
                self.count += 1
    else:
        raise TypeError("Начальное значение должно быть строкой или списком")

def size(self):
    """Возвращает максимальный размер битовой строки."""
    return self.size_

```



```

def __len__(self):
    """Возвращает текущее количество установленных битов."""
    return self.count

def __getitem__(self, index):
    """Перегрузка оператора индексирования []."""
    if isinstance(index, int):
        if index < 0 or index >= self.size_:
            raise IndexError("Индекс вне допустимого диапазона")
        return self.bits[index]
    elif isinstance(index, slice):
        return self.bits[index]
    else:
        raise TypeError("Индекс должен быть целым числом или срезом")

def __and__(self, other):
    """Операция побитового AND."""
    if not isinstance(other, BitString):
        raise TypeError("Операнд должен быть объектом класса BitString")
    if self.size_ != other.size_:
        raise ValueError("Битовые строки должны иметь одинаковый размер")

    result = BitString(self.size_)
    for i in range(self.size_):
        result.bits[i] = self.bits[i] & other.bits[i]
        if result.bits[i] == 1:
            result.count += 1
    return result

def __or__(self, other):
    """Операция побитового OR."""
    if not isinstance(other, BitString):
        raise TypeError("Операнд должен быть объектом класса BitString")
    if self.size_ != other.size_:
        raise ValueError("Битовые строки должны иметь одинаковый размер")

```

```
result = BitString(self.size_)
for i in range(self.size_):
    result.bits[i] = self.bits[i] | other.bits[i]
    if result.bits[i] == 1:
        result.count += 1
return result
```

```
def __xor__(self, other):
    """Операция побитового XOR."""
    if not isinstance(other, BitString):
        raise TypeError("Операнд должен быть объектом класса BitString")
    if self.size_ != other.size_:
        raise ValueError("Битовые строки должны иметь одинаковый размер")
```

```
result = BitString(self.size_)
for i in range(self.size_):
    result.bits[i] = self.bits[i] ^ other.bits[i]
    if result.bits[i] == 1:
        result.count += 1
return result
```

```
def __invert__(self):
    """Операция побитового NOT."""
    result = BitString(self.size_)
    for i in range(self.size_):
        result.bits[i] = 1 - self.bits[i]
        if result.bits[i] == 1:
            result.count += 1
    return result
```

```
def shift_left(self, n):
    """Сдвиг влево на n бит."""
    if n < 0:
        raise ValueError("Количество сдвигов должно быть неотрицательным")
```

```
result = BitString(self.size_)
result.bits = self.bits[n:] + [0] * min(n, self.size_)
result.count = sum(result.bits)
return result
```

```
def shift_right(self, n):
    """Сдвиг вправо на n бит."""
    if n < 0:
        raise ValueError("Количество сдвигов должно быть неотрицательным")
    result = BitString(self.size_)
    result.bits = [0] * min(n, self.size_) + self.bits[:self.size_ - n]
    result.count = sum(result.bits)
    return result
```

```
def __str__(self):
    """Строковое представление битовой строки."""
    return ".join(map(str, self.bits))
```

```
# Создание объектов BitString
```

```
bs1 = BitString(8, "10101010") # Битовая строка: 10101010
```

```
bs2 = BitString(8, "11001100") # Битовая строка: 11001100
```

```
# Вывод начальных битовых строк
```

```
print("bs1:", bs1) # Вывод: 10101010
```

```
print("bs2:", bs2) # Вывод: 11001100
```

```
# Побитовые операции
```

```
and_result = bs1 & bs2 # Побитовое AND
```

```
or_result = bs1 | bs2 # Побитовое OR
```

```
xor_result = bs1 ^ bs2 # Побитовое XOR
```

```
not_result = ~bs1 # Побитовое NOT
```

```
print("AND (bs1 & bs2):", and_result) # Вывод: 10001000
```

```
print("OR (bs1 | bs2):", or_result) # Вывод: 11101110
```

```
print("XOR (bs1 ^ bs2):", xor_result) # Вывод: 01100110
```

```

print("NOT (~bs1):", not_result)    # Вывод: 01010101

# Сдвиги
shift_left_result = bs1.shift_left(2) # Сдвиг влево на 2 бита
shift_right_result = bs1.shift_right(2) # Сдвиг вправо на 2 бита

print("Shift Left (bs1 << 2):", shift_left_result) # Вывод: 10101000
print("Shift Right (bs1 >> 2):", shift_right_result) # Вывод: 00101010

# Индексирование
print("bs1[3]:", bs1[3]) # Вывод: 0 (четвертый бит)
print("bs1[1:5]:", bs1[1:5]) # Вывод: [0, 1, 0, 1] (срез)

# Размер и длина
print("Size of bs1:", bs1.size()) # Вывод: 8 (максимальный размер)
print("Count of set bits in bs1:", len(bs1)) # Вывод: 4 (количество единиц)

# Создание пустой битовой строки и заполнение её значениями
bs3 = BitString(8) # Пустая битовая строка: 00000000
print("Initial bs3:", bs3) # Вывод: 00000000

# Установка значений через индексирование
bs3[0] = 1
bs3[2] = 1
bs3[4] = 1
bs3[6] = 1
print("Modified bs3:", bs3) # Вывод: 10101010
print("Count of set bits in bs3:", len(bs3)) # Вывод: 4 (количество единиц)

```

Ответы на контрольные вопросы:

1. Какие средства существуют в Python для перегрузки операций?

Для перегрузки операций существуют специальные методы, задающие работу соответствующих операторов.

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 ( $x // y$ ).

`__lt__(self, other)` -  $x < y$  вызывает `x.__lt__(y)`.

`__le__(self, other)` -  $x \leq y$  вызывает `x.__le__(y)`.

`__eq__(self, other)` -  $x == y$  вызывает `x.__eq__(y)`.

`__ne__(self, other)` -  $x != y$  вызывает `x.__ne__(y)`.

`__gt__(self, other)` -  $x > y$  вызывает `x.__gt__(y)`.

`__ge__(self, other)` -  $x \geq y$  вызывает `x.__ge__(y)`.

`__iadd__(self, other)` -  $+=$ .

`__isub__(self, other)` -  $-=$ .

`__imul__(self, other)` -  $*=$ .

`__itruediv__(self, other)` -

`__ifloordiv__(self, other)` -

`__imod__(self, other)` -  $\% =$ .

`__ipow__(self, other[, modulo])` -

`__lshift__(self, other)` -  $<< =$ .

`__rshift__(self, other)` -  $>> =$ .

`__iand__(self, other)` -  $\& =$ .

`__ixor__(self, other)` -  $\wedge =$ .

`__ior__(self, other)` -  $| =$ .

`__neg__(self)` - унарный  $-$ . `y.__radd__(x)`.

`__pos__(self)` - унарный  $+$ .

`__abs__(self)` - модуль (`abs()`).

`__invert__(self)` - инверсия ( $\sim$ ).

`__complex__(self)` - приведение к `complex`.

`__int__(self)` - приведение к `int`. `__float__(self)` - приведение к `float`.  
`__round__(self[, n])` - округление.

`__radd__(self, other),`

`__rsub__(self, other),`

`__rmul__(self, other),`

`__rtruediv__(self, other) ,`

`__rfloordiv__(self, other),`

`__rmod__(self, other),`

`__rdivmod__(self, other),`

`__rpow__(self, other),`

`__rlshift__(self, other),`

`__rrshift__(self, other) ,`

`__rand__(self, other) ,`

`__rxor__(self, other) ,`

`__ror__(self, other)`

`__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 (`modulo`)).

`__lshift__(self, other)` - битовый сдвиг влево (`x << y`). `item in x ** y`,  
`pow(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.

`__add__(self, other)` вызывается при использовании оператора `+` . (`a + b`)

`__iadd__(self, other)` вызывается при использовании оператора `+=` . (`a +=`

b)

`__radd__(self, other)` делает то же самое, что и арифметические операторы, перечисленные выше, но для аргументов, находящихся справа, и только в случае, если для левого операнда не определён соответствующий метод.

4. Для каких целей предназначен метод `__new__` ? Чем он отличается от метода `__init__` ?

`__new__(cls[, ...])` — управляет созданием экземпляра. В качестве обязательного аргумента принимает класс (не путать с экземпляром). Должен возвращать экземпляр класса для его последующей его передачи методу

`__init__` . `__init__(self[, ...])` - конструктор.

5. Чем отличаются методы `__str__` и `__repr__` .

`__repr__(self)` - вызывается встроенной функцией `repr`; возвращает "сырые" данные, используемые для внутреннего представления в python.

`__str__(self)` - вызывается функциями `str`, `print` и `format`. Возвращает строковое представление объекта.

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