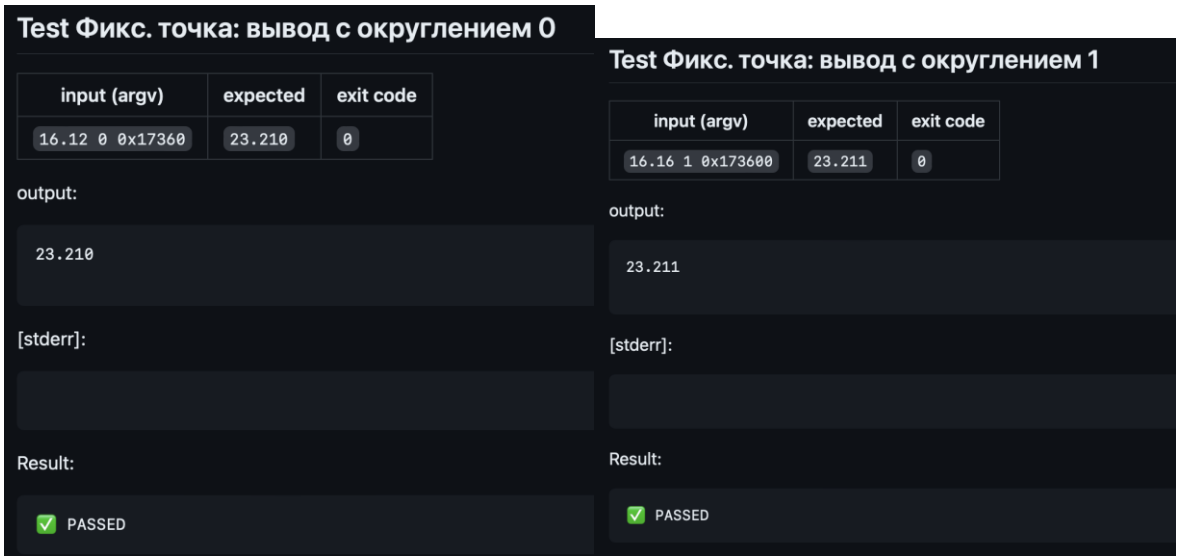
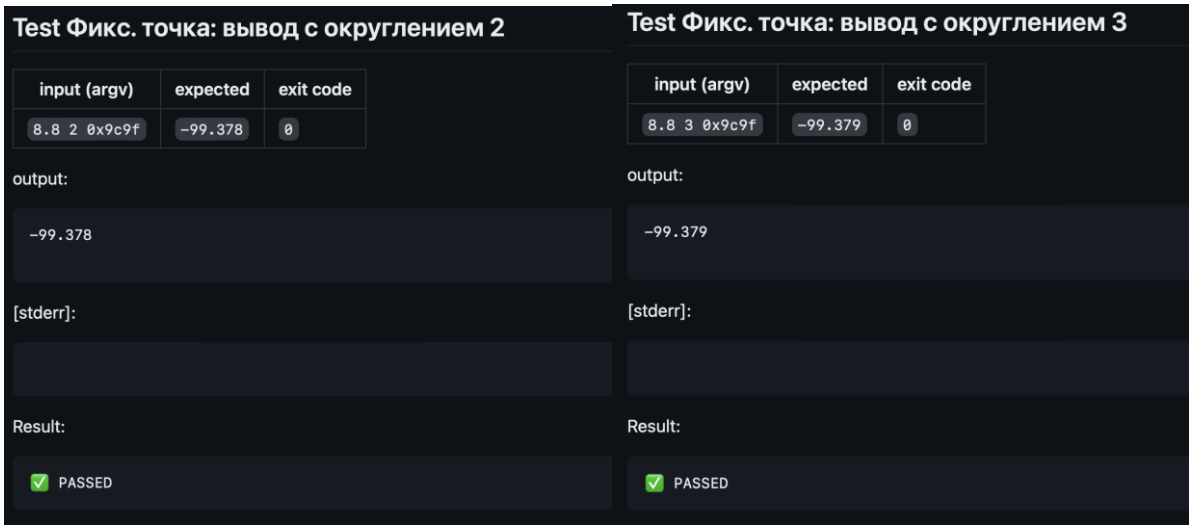
| | | |
|---|---|---|
| Лабораторная работа №1 | 22.Б05 | 2023 |
| Представление чисел | Назаров Матвей Антонович | |

**Инструментарий и требования к работе:** Python (3.11.5)

**Ссылка на репозиторий: https://github.com/skkv-mkn/mkn-comp-arch-2023-fixed-floating-lomalovo**

**Результат работы на тестовых данных:**



## Test Фикс. точка: + с округлением 0

| input (argv) | expected | exit code |
|---|---|---|
| `8.8 0 0x9c9f + 0x1736` | `-76.167` | `0` |

output:

```
-76.167
```

[stderr]:

Result:

✅ PASSED

## Test Фикс. точка: + с округлением 1

| input (argv) | expected | exit code |
|---|---|---|
| `8.8 1 0xdc9f + 0xd736` | `-76.168` | `0` |

output:

```
-76.168
```

[stderr]:

Result:

✅ PASSED

## Test Фикс. точка: + с округлением 2

| input (argv) | expected | exit code |
|---|---|---|
| `16.16 2 0x6f7600 + 0x173600` | `134.672` | `0` |

output:

```
134.672
```

[stderr]:

Result:

✅ PASSED

## Test Фикс. точка: + с округлением 3

| input (argv) | expected | exit code |
|---|---|---|
| `16.16 3 0x6f7600 + 0x173600` | `134.671` | `0` |

output:

```
134.671
```

[stderr]:

Result:

✅ PASSED

## Test Плав. точка (single): * с округлением 0

| input (argv) | expected | exit code |
|---|---|---|
| `f 0 0x414587dd * 0x42ebf110` | `0x1.6c1b72p+10` | `0` |

output:

```
0x1.6c1b72p+10
```

[stderr]:

Result:

✅ PASSED

## Test Плав. точка (half): * с округлением 0

| input (argv) | expected | exit code |
|---|---|---|
| `h 0 0x4145 * 0x42eb` | `0x1.238p+3` | `0` |

output:

```
0x1.238p+3
```

[stderr]:

Result:

✅ PASSED


## Описание работы кода:

Я считаю, что сделал фиксированную точку без деления, со всеми способами округления + плавающую точку с умножением и округлением 0 + постарался реализовать все крайние случаи

Моя программа работает так: сначала я парсю аргументы командной строки, после этого вызываю основную функцию apply_operation

Внутри этой функции она обрабатывает number_format и в зависимости от типа числа действует по-разному:

### Фиксированная точка

Для фикс точки сначала я перевожу числа в 10ичную форму и парсю формат числа A.B. После этого я смотрю есть ли операция с числами, если есть, то выполняю ее с десятичной формой иначе просто оставляю первое число

Теперь перейдем к определению того, что получилось после операции. Я собираюсь хранить число так: целую часть я хочу хранить как int, а дробную как str, потому что дробная может начинаться с нуля. Переведем 10чную запись в двоичную и возьмем справа B знаков – получим дробную запись в двоичной, из оставшихся возьмем A правых – это будет запись целой части в формате дополнения до 2. После этого я определяю сначала целую часть в десятичной форме (для этого, если крайний левый битик не 0, то я вычитаю два раза нужную степень двойки, т.е. как бы заменяю 1 на -1 в двоичной записи)

```
if  len(whole_part) < whole_part_size:
        whole_part_dec = int(whole_part, 2)
else:
        whole_part_dec = int(whole_part, 2) - 2 ** whole_part_size *
int(whole_part[0])
```

Теперь преобразую дробную часть, тут заметим, что мы хотим получить запись правее точки, для этого на нужно сначала поперемножать битики на 2^-n с соответственным n, а потом умножить на 10 в степени количества знаков, тогда вынесем 5 в степени количества знаков, а двойку в степени количества знаков занесем внутрь, тогда понимаем, что на самом деле нужная нам запись числа это просто степень 5 умножить на перевод записи из 2 в 10, но обычной а не дробной.

```
if fractional_part[0] == "0":
   fractional_part = "1" + fractional_part[1:]
   fr_before_format = str(int(fractional_part, 2) * 5 ** fractional_part_size)
   fractional_part_str = str(int(fr_before_format[0]) - 5) + fr_before_format[1:]
else:
   fractional_part_str = str(int(fractional_part, 2) * 5 ** fractional_part_size)
```

Тут я обрабатываю один важный момент: если дробная часть начиналась с 0, то чтобы не потерять эти 0 при применении функции int, я первый битик заменяю на 1

После того как я получил десятичную запись, обращаюсь к функции округления:

(с умножением и делением действую похоже, просто округляя дважды, сначала результат умножения, а потом до 3ех знаков)

## Функция округления для фиксированной

```
        def rounding_for_fix_point(round_type: str, whole_part: int,
fractional_part: str, numb_after_point: int) -> str:
```

Последний из аргументов этой функции это количество знаков после запятой, которое мы хотим оставить

Вначале я объявляю, что для того, чтобы внутри функции хранить десятичную запись дробной части, я допишу слева 1 и превращу в int, это сделано, чтобы не потерять нули

Сначала проверяем случай, когда нам надо округлить в месте, правее которого все нули. В этом случае любое округление это просто отбрасывание знаков.

После этого делю все на два основных случая:

1)целая часть >0

2)целая часть <0

В зависимости от этого округление выполняется по-разному

В начале каждого из случаев я определяю, как будет выглядеть два варианта округления, а потом, в зависимости от типа округления, я определяю какой из них вернуть.

Благодаря тому, что округление я разбил на два случая, 0, 2 и 3 способ округления (к нулю и к бесконечностям) даются нам бесплатно, без необходимости смотреть на число.

А вот с округлением к ближайшему четному придется посмотреть на последнюю цифру и в зависимости от этого обработать вывод.

Еще тут стоит упомянуть то, как я разобрал случай с <0, ведь для того, чтобы записать ответ в этом случае нам надо вычесть дробную часть из 1:

round_down = whole + "." + str(3 * (10 ** (len(fractional_part))) - fractional_part_dec)[1:][:numb_after_point]

Это можно пояснить на примере: пусть наша запись это 0123 (например у числа 88.0123) У меня я храню ее как integer равный 10123, а теперь, для того, чтобы получить запись вычтенного из 1 нашего числа, при этом тоже начинающейся с 1, я вычитаю из 30000 10123, тогда как раз получим то, что хотим (случай с дробной частью равной 0000000 я тоже обрабатываю и действую отдельно)

## Плавающая точка

Теперь рассмотрим плавающую точку, сначала преобразую 16 в 2 и дописываю 0 слева. По этому определяю знак, экспоненту и мантиссу. Знак – первый бит, экспоненту восстанавливаю по форме со сдвигом. Далее мантиссу дополняю до 12 и 24 бит соответственно, чтобы при дальнейшем их преобразовании в 16ричную систему получить 3 и 6 символов соответственно. Теперь рассматриваю варианты разных операций, первым идет отсутствие операции. В этом случае сначала проверяю число на

денормализированное и на 0, если все ок, то return результат с округлением 0 (отбрасываю знак)

Теперь рассматриваю умножение, умножаю мантиссы как инты и смотрю, сколько знаков вылезло за запятую (обрабатываю ситуацию, когда после умножения количество символов до запятой возросло, например 1.1 * 1.1 = 10.01 тогда нам надо определять это как 1.001). В зависимости от этого обрабатываю экспоненту (увеличиваю или уменьшаю), после этого округляю к 0

В делении рассмотрел крайние случаи, это деление на 0 и на +-inf