МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ

РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное

образовательное учреждение высшего образования

«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Кафедра инфокоммуникаций

Отчет по лабораторной работе №2.9

Рекурсия в языке Python

по дисциплине «Технологии программирования и алгоритмизации»

Выполнил студент группы ИВТ-б-о-20-1

Хашиев Х.М. « »_____20__г.

Подпись студента_____

Работа защищена « »_____20__г.

Проверил Воронкин Р.А. _____

(подпись

Ставрополь 2021

Цель работы: приобретение навыков по работе с рекурсивными функциями при написании программ с помощью языка программирования Python версии 3.

Ход работы: Пример 1

https://github.com/Mirror-Shard/L2.9

1. Создал репозиторий на github с лицензией MIT, добавил .gitignore и выбрал язык Python.

2. Проработал пример из учебника:

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

# Эта программа показывает работу декоратора, который производит оптимизацию
# хвостового вызова. Он делает это, вызывая исключение, если оно является его
# прародителем, и перехватывает исключения, чтобы вызвать стек

import sys


class TailRecurseException:
    def __init__(self, args, kwargs):
        self.args = args
        self.kwargs = kwargs


def tail_call_optimized(g):
    """
    Эта программа показывает работу декоратора, который производит оптимизацию
    хвостового вызова. Он делает это, вызывая исключение, если оно является его
    прародителем, и перехватывает исключения, чтобы подделать оптимизацию хвоста.

    Эта функция не работает, если функция декоратора не использует хвостовой вызов.
    """

    def func(*args, **kwargs):

        f = sys._getframe()
        if f.f_back and f.f_back.f_back and f.f_back.f_back.f_code == f.f_code:
            raise TailRecurseException(args, kwargs)
        else:
            while True:
```

Рисунок 1 – Код примера(1)

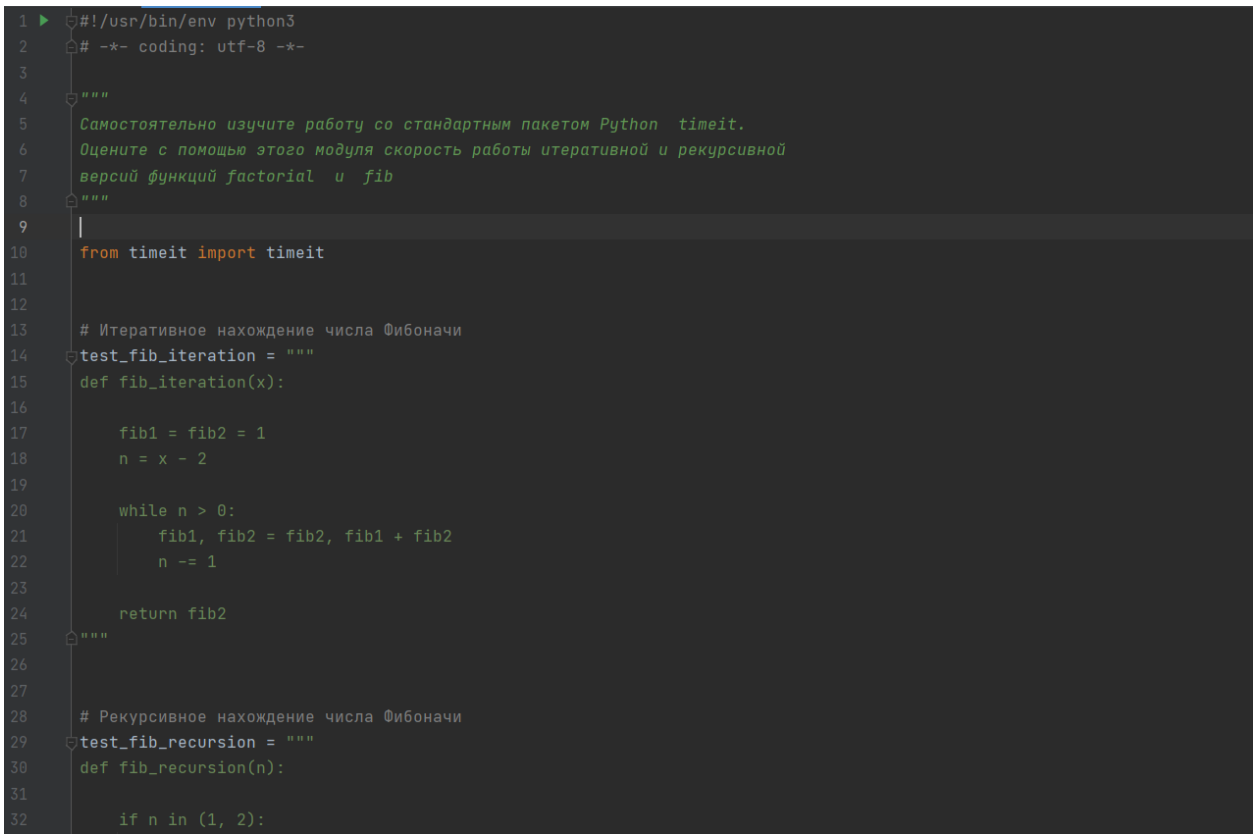
```
                    try:
                        return g(*args, **kwargs)
                    except TailRecurseException as e:
                        args = e.args
                        kwargs = e.kwargs

        func.__doc__ = g.__doc__
        return func


@tail_call_optimized
def factorial(n, acc=1):
    """calculate a factorial"""
    if n == 0:
        return acc

    return factorial(n - 1, n * acc)


def fib(i, current=0, next=1):
    if i == 0:
        return current
    else:
        return fib(i - 1, next, current + next)


if __name__ == '__main__':
    print(factorial(10000))
    print(fib(10000))

```

Рисунок 2 – Код примера(2)

Задание 1

Самостоятельно изучите работу со стандартным пакетом Python timeit . Оцените с помощью этого модуля скорость работы итеративной и рекурсивной версий функций factorial и fib .

1. Код задания 1(порядок определения функций не имеет значения):



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

"""
Самостоятельно изучите работу со стандартным пакетом Python  timeit.
Оцените с помощью этого модуля скорость работы итеративной и рекурсивной
версий функций factorial  и  fib
"""

from timeit import timeit


# Итеративное нахождение числа Фибоначи
test_fib_iteration = """
def fib_iteration(x):

    fib1 = fib2 = 1
    n = x - 2

    while n > 0:
        fib1, fib2 = fib2, fib1 + fib2
        n -= 1

    return fib2
"""


# Рекурсивное нахождение числа Фибоначи
test_fib_recursion = """
def fib_recursion(n):

    if n in (1, 2):
```

Рисунок 4 – Код задания 1

2. Результат работы:

```
Время выполнения итеративной функции числа Фибоначи:  5.700000000004313e-08
Время выполнения рекурсивной функции числа Фибоначи:  6.600000000002437e-08
Время выполнения декоративной функции числа Фибоначи:  0.00010462699999999999
Время выполнения итеративной функции факториала:  4.7999999999992495e-08
Время выполнения рекурсивной функции факториала:  2.243999999999996e-06
Время выполнения декоративной функции факториала:  6.700000000005313e-08
```

Рисунок 5 – Результат работы второго задания

Задание 2

Самостоятельно проработайте пример с оптимизацией хвостовых вызовов в Python. С помощью пакета timeit оцените скорость работы функций factorial и fib с использованием интроспекции стека и без использования интроспекции стека. Приведите полученные результаты в отчет.

1. Код задания 2, часть первая:

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

from timeit import timeit


# Рекурсивное нахождение факториала
test_fac = """
def fac_recursion(n):

    if n == 0:
        return 1

    return fac_recursion(n - 1) * n
"""


# Рекурсивное нахождение числа Фибоначи
test_fib = """
def fib_recursion(n):

    if n in (1, 2):
        return 1

    return fib_recursion(n - 1) + fib_recursion(n - 2)
"""


test_fac_optimized = """
class TailRecurseException:
    def __init__(self, args, kwargs):
        self.args = args
```

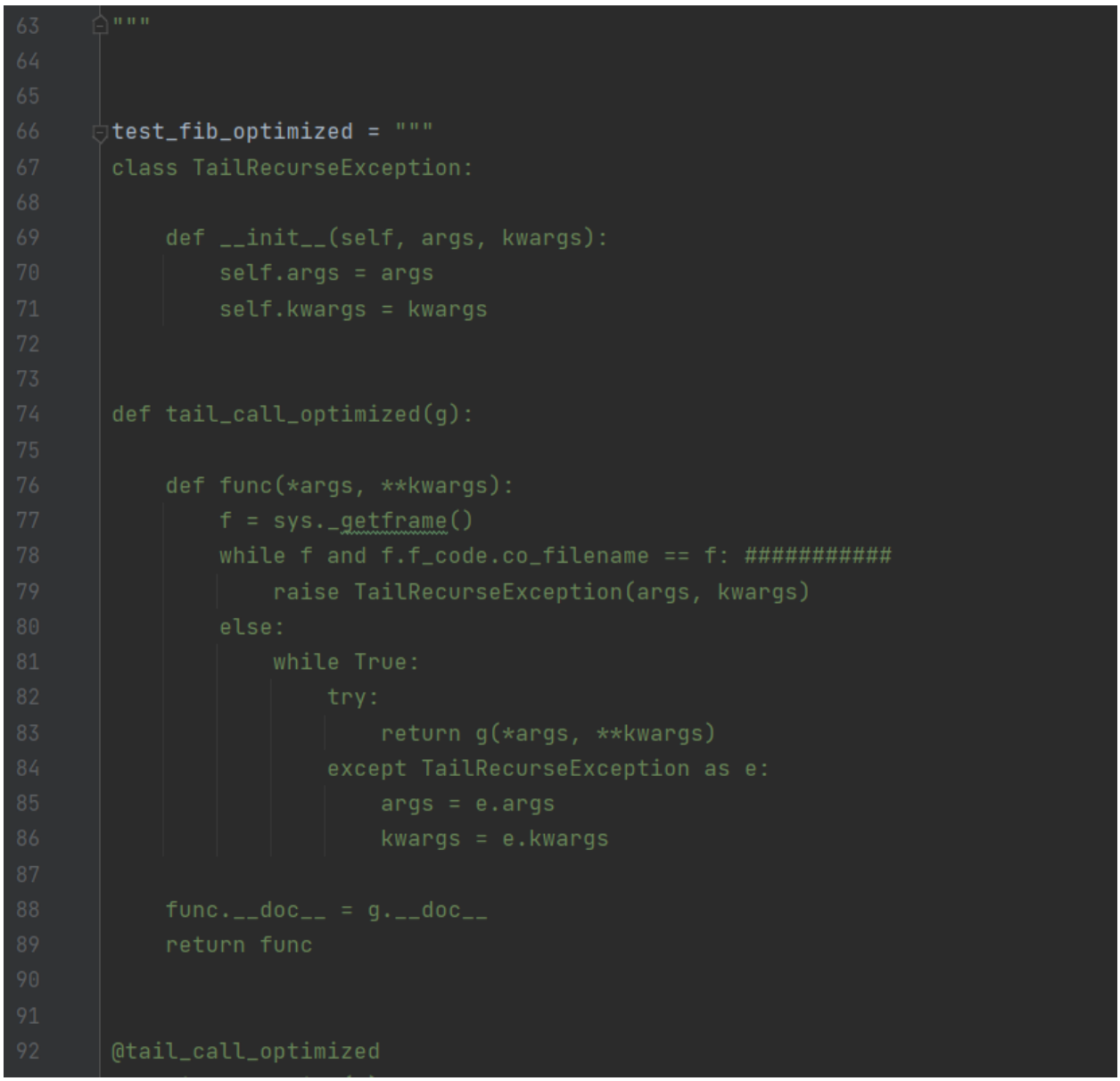
Рисунок 6 – Код задания 2, часть первая

```
        self.kwargs = kwargs


def tail_call_optimized(g):

    def func(*args, **kwargs):

        f = sys._getframe()

        while f and f.f_code.co_filename == f: #############
            raise TailRecurseException(args, kwargs)
        else:
            while True:
                try:
                    return g(*args, **kwargs)
                except TailRecurseException as e:
                    args = e.args
                    kwargs = e.kwargs

    func.__doc__ = g.__doc__
    return func


@tail_call_optimized
def fac_recursion(n):

    if n == 0:
        return 1

    return fac_recursion(n - 1) * n
```

Рисунок 7 – Код задания 2, часть вторая

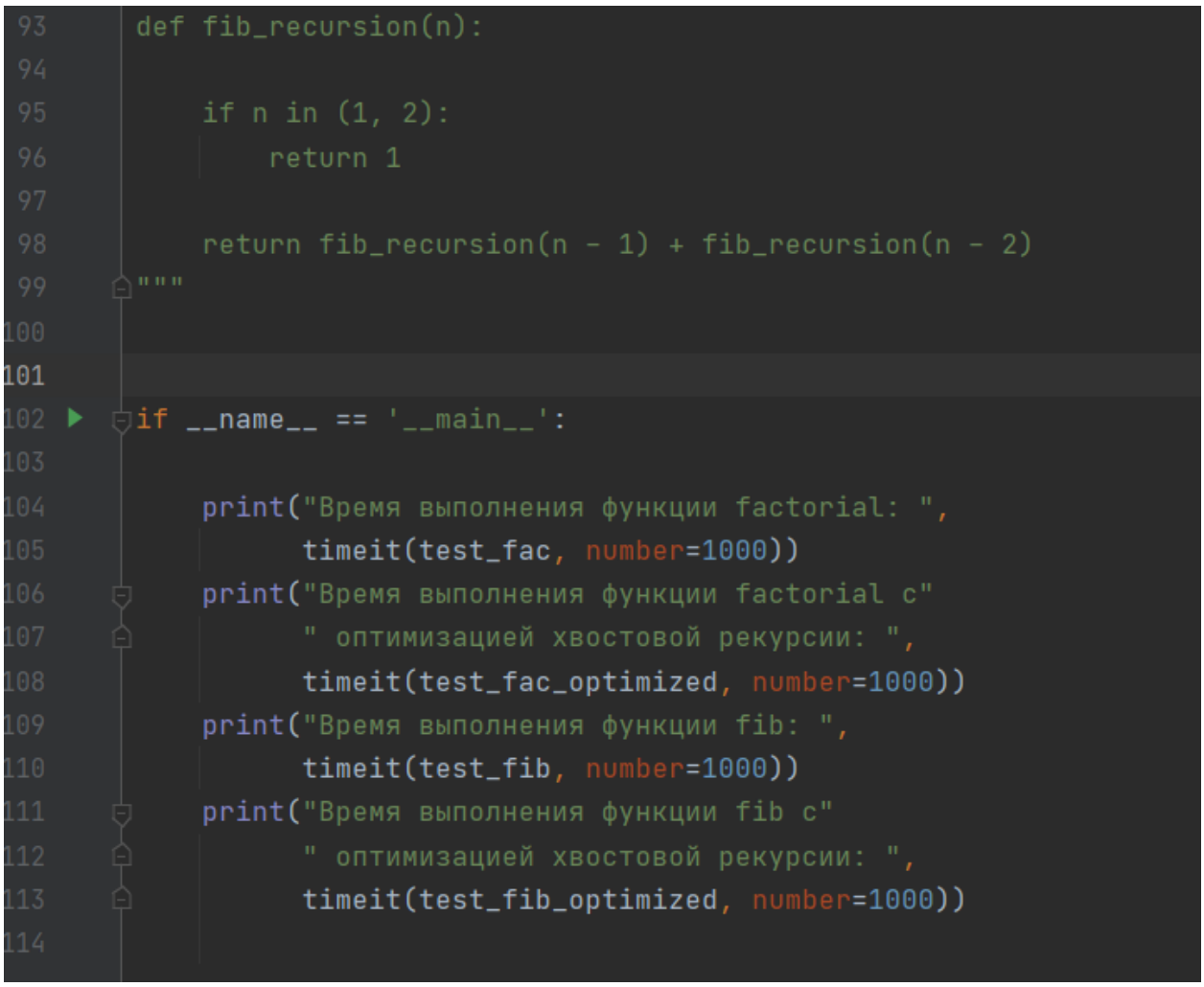
```
"""


test_fib_optimized = """
class TailRecurseException:

    def __init__(self, args, kwargs):
        self.args = args
        self.kwargs = kwargs


def tail_call_optimized(g):

    def func(*args, **kwargs):
        f = sys._getframe()
        while f and f.f_code.co_filename == f: ############
            raise TailRecurseException(args, kwargs)
        else:
            while True:
                try:
                    return g(*args, **kwargs)
                except TailRecurseException as e:
                    args = e.args
                    kwargs = e.kwargs

    func.__doc__ = g.__doc__
    return func


@tail_call_optimized
```

Рисунок 8 – Код задания 2, часть третья



```
"""


test_fib_optimized = """
class TailRecurseException:

    def __init__(self, args, kwargs):
        self.args = args
        self.kwargs = kwargs


def tail_call_optimized(g):

    def func(*args, **kwargs):
        f = sys._getframe()
        while f and f.f_code.co_filename == f: ############
            raise TailRecurseException(args, kwargs)
        else:
            while True:
                try:
                    return g(*args, **kwargs)
                except TailRecurseException as e:
                    args = e.args
                    kwargs = e.kwargs

    func.__doc__ = g.__doc__
    return func


@tail_call_optimized
```

Рисунок 9 – Код задания 2, часть четвёртая



```
def fib_recursion(n):

    if n in (1, 2):
        return 1

    return fib_recursion(n - 1) + fib_recursion(n - 2)
"""


if __name__ == '__main__':

    print("Время выполнения функции factorial: ",
          timeit(test_fac, number=1000))
    print("Время выполнения функции factorial с"
          " оптимизацией хвостовой рекурсии: ",
          timeit(test_fac_optimized, number=1000))
    print("Время выполнения функции fib: ",
          timeit(test_fib, number=1000))
    print("Время выполнения функции fib с"
          " оптимизацией хвостовой рекурсии: ",
          timeit(test_fib_optimized, number=1000))
```

Рисунок 10 – Код задания 2, часть 4

Индивидуальное задание

Создайте функцию, подсчитывающую сумму элементов массива по следующему алгоритму:

- массив делится пополам, подсчитываются и складываются суммы элементов в каждой половине. Сумма элементов в половине массива подсчитывается по тому же алгоритму, то есть снова путем деления пополам. Деления происходят, пока в получившихся кусках массива не окажется по одному элементу и вычисление суммы, соответственно, не станет тривиальным.

1. Создал рекурсивную функцию, которая складывает все элементы массива:

```python
from math import fsum


# Подсчитывает сумму всех элементов
def rec(massive):

    # Если в массиве два элемента
    if len(massive) == 2:
        return fsum(massive)

    # Если при дальнейшем делении массива останется 1 элемент
    elif len(massive) == 3:
        return fsum(massive)

    # Если в массиве больше 3 элементов
    else:
        # Делит массив на две части
        n = len(massive) // 2
        a = massive[:n]
        b = massive[n:]

        # Складывает части массива
        return rec(a) + rec(b)


if __name__ == '__main__':

    print("Введите числа в массив через пробел: ")
    mas = list(map(float, input().split()))
    print(rec(mas))
```

Рисунок 11 – Код индивидуального задания

2. Результат работы:

```
Введите числа в массив через пробел:
1 2 3 4 5 6 7 8 9 10
55.0
```

Рисунок 12 – Код индивидуального задания, часть 1

Контрольные вопросы:

1. Для чего нужна рекурсия?

Функция может содержать вызов других функций. В том числе процедура может вызвать саму себя. Никакого парадокса здесь нет – компьютер лишь последовательно выполняет встретившиеся ему в программе команды и, если встречается вызов процедуры, просто начинает выполнять эту функцию. Без разницы, какая функция дала команду это делать.

2. Что называется базой рекурсии?

База рекурсии – это такие аргументы функции, которые делают задачу настолько простой, что решение не требует дальнейших вложенных вызовов.

3. Самостоятельно изучите что является стеком программы. Как используется стек программы при вызове функций?

Стек в Python – это линейная структура данных, в которой данные расположены объектами друг над другом. Он хранит данные в режиме LIFO (Last in First Out). Данные хранятся в том же порядке, в каком на кухне тарелки располагаются одна над другой. Мы всегда выбираем последнюю тарелку из стопки тарелок. В стеке новый элемент вставляется с одного конца, и элемент может быть удален только с этого конца.

4. Как получить текущее значение максимальной глубины рекурсии в языке Python?

Чтобы проверить текущие параметры лимита, нужно запустить: sys.getrecursionlimit().

5. Что произойдет если число рекурсивных вызовов превысит максимальную глубину рекурсии в языке Python?

Существует предел глубины возможной рекурсии, который зависит от реализации Python. Когда предел достигнут, возникает исключение RuntimeError.

6. Как изменить максимальную глубину рекурсии в языке Python?

Изменить максимальную глубину рекурсии можно с помощью sys.setrecursionlimit().

7. Каково назначение декоратора lru_cache ?

Декоратор lru_cache является полезным инструментом, который можно использовать для уменьшения количества лишних вычислений. Декоратор оборачивает функцию с переданными в нее аргументами и запоминает возвращаемый результат, соответствующий этим аргументам.

8. Что такое хвостовая рекурсия? Как проводится оптимизация хвостовых вызовов?

Хвостовая рекурсия — частный случай рекурсии, при котором любой рекурсивный вызов является последней операцией перед возвратом из функции. Подобный вид рекурсии примечателен тем, что может быть легко заменён на итерацию путём формальной и гарантированно корректной перестройки кода функции. Оптимизация хвостовой рекурсии путём преобразования её в плоскую итерацию реализована во многих оптимизирующих компиляторах. В некоторых функциональных языках программирования спецификация гарантирует обязательную оптимизацию хвостовой рекурсии.

Вывод: приобретение навыков по работе с функциями при написании программ с помощью языка программирования Python версии 3.