

**МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ  
РОССИЙСКОЙ ФЕДЕРАЦИИ  
Федеральное государственное автономное  
образовательное учреждение высшего образования  
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»**

**Кафедра инфокоммуникаций**

**Основы кроссплатформенного программирования**

**Отчет по лабораторной работе №2.9**

Рекурсия в языке Python

Выполнил студент группы

ИВТ-б-о-21-1

Мальцев Н.А. « » \_\_\_\_\_ 20\_\_ г.

Подпись студента \_\_\_\_\_

Работа защищена « » \_\_\_\_\_ 20\_\_ г.

Проверил доцент

Кафедры инфокоммуникаций, старший  
преподаватель

Воронкин Р.А.

\_\_\_\_\_  
(подпись)

Ставрополь 2022

## Рекурсия в языке Python.

**Цель работы:** приобретение навыков по работе с рекурсивными функциями при написании программ с помощью языка программирования Python 3.10.

### Порядок выполнения работы:

#### 1. Проработка примеров:

Код программы:

```
#!/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(Exception):
    def __init__(self, args, kwargs):
        self.args = args
        self.kwargs = kwargs

def tail_call_optimized(g):
    """
    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:
                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)

"""
@tail_call_optimized
```

```
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))
# выводит большое число,
# но не доходит до лимита рекурсии
```

Результат работы программы:

```
"C:\Users\Николай Мальцев\AppData\Local\Programs\Python\Python310\python.exe"
2846259680917054518906413212119868890148051401702799230794179994274411
Process finished with exit code 0
```

Рисунок 1. Результат работы программы из примера 1

## 2. Выполнение задачи 1:

Код программы:

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

import timeit

code1 = '''
def factorial(n):
    if n == 0:
        return 1
    elif n == 1:
        return 1
    else:
        return n * factorial(n - 1)
'''

code2 = '''
def fib(n):
    if n == 0 or n == 1:
        return n
    else:
        return fib(n - 2) + fib(n - 1)
'''

code3 = '''
def factorial(n):
    product = 1
    while n > 1:
        product *= n
        n -= 1
    return product
'''
```

```

code4 = '''
def fib(n):
    a, b = 0, 1
    while n > 0:
        a, b = b, a + b
        n -= 1
    return a
'''

code5 = '''
from functools import lru_cache
@lru_cache
def factorial(n):
    if n == 0:
        return 1
    elif n == 1:
        return 1
    else:
        return n * factorial(n - 1)
'''

code6 = '''
from functools import lru_cache
@lru_cache
def fib(n):
    if n == 0 or n == 1:
        return n
    else:
        return fib(n - 2) + fib(n - 1)
'''

if __name__ == '__main__':
    print('Результат рекурсивного факториала:', timeit.timeit(setup=code1,
number=1000))
    print('Результат рекурсивного числа Фибоначи:',
timeit.timeit(setup=code2, number=1000))
    print('Результат итеративного факториала:', timeit.timeit(setup=code3,
number=1000))
    print('Результат итеративного числа Фибоначи:',
timeit.timeit(setup=code4, number=1000))
    print('Результат факториала с декоратором:', timeit.timeit(setup=code5,
number=1000))
    print('Результат числа Фибоначи с декоратором:',
timeit.timeit(setup=code6, number=1000))

```

Результат работы программы:

```

"C:\Users\Николай Мальцев\AppData\Local\Programs\Python\Python3
Результат рекурсивного факториала: 8.600007276982069e-06
Результат рекурсивного числа Фибоначи: 9.599985787644982e-06
Результат итеративного факториала: 9.400013368576765e-06
Результат итеративного числа Фибоначи: 9.399984264746308e-06
Результат факториала с декоратором: 9.200011845678091e-06
Результат числа Фибоначи с декоратором: 8.800008799880743e-06

```

Рисунок 2. Результат работы программы к заданию 1

## Листинг программы:

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

import timeit

code1 = '''
def factorial(n, acc=1):
    if n == 0:
        return acc
    return factorial(n-1, n*acc)
'''

code2 = '''
def fib(i, current = 0, next = 1):
    if i == 0:
        return current
    else:
        return fib(i - 1, next, current + next)
'''

code3 = '''
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
def factorial(n, acc=1):
    if n == 0:
        return acc
    return factorial(n-1, n*acc)
'''

code4 = '''
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
def fib(i, current = 0, next = 1):
    if i == 0:
        return current
    else:
        return fib(i - 1, next, current + next)
'''

if __name__ == '__main__':
    print('Результат рекурсивного факториала:', timeit.timeit(setup=code1,
number=1000))
    print('Результат рекурсивного числа Фибоначи:',
timeit.timeit(setup=code2, number=1000))
    print('Результат итеративного факториала:', timeit.timeit(setup=code3,
number=1000))
    print('Результат итеративного числа Фибоначи:',
timeit.timeit(setup=code4, number=1000))

```

Результат работы программы:

```

"C:\Users\Николай Мальцев\AppData\Local\Programs\Python\Python
Результат рекурсивного факториала: 9.500014130026102e-06
Результат рекурсивного числа Фибоначи: 9.499985026195645e-06
Результат итеративного факториала: 9.499985026195645e-06
Результат итеративного числа Фибоначи: 9.600014891475439e-06

Process finished with exit code 0

```

Рисунок 3. Результат работы программы к заданию 2

Вывод: с использованием декоратора `iru_cache` рекурсивная функция выполняется примерно в 3 раза быстрее, а итеративная в 2.

#### 4. Выполнение индивидуального задания (вариант 11):

Задача: решить индивидуальное задание лабораторной работы 2.6, оформив каждую команду в виде отдельной функции.

Листинг программы:

```

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

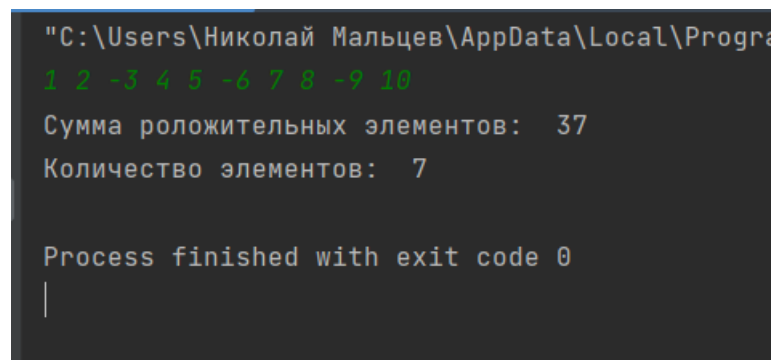
"""
Задан список положительных чисел, признаком конца которых служит
отрицательное
число. Используя рекурсию, подсчитать количество чисел и их сумму.
"""

```

```
def negative_num(s, sm, cnt):
    if not s:
        return sm, cnt
    if s[0] >= 0:
        sm += s[0]
        cnt += 1
    return negative_num(s[1:], sm, cnt)

if __name__ == '__main__':
    count = 0
    summ = 0
    A = list(map(int, input().split()))
    a = negative_num(A, summ, count)
    print("Сумма роложительных элементов: ", a[0], "\nКоличество элементов: ", a[1])
```

Результат выполнения программы:



```
"C:\Users\Николай Мальцев\AppData\Local\Programs\Python\Python38-64\python.exe"
1 2 -3 4 5 -6 7 8 -9 10
Сумма роложительных элементов: 37
Количество элементов: 7

Process finished with exit code 0
|
```

Рисунок 4. Результат выполнения программы к индивидуальному заданию

## Ответы на вопросы:

### 1. Для чего нужна рекурсия?

Рекурсивна программа применяется при повторяющихся или бесконечных вычислениях. Самоё распространённое применение рекурсии: вычисление факториала числа.

### 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 функции?

Стек – это способ хранения данных в памяти. Использовать можно только те данные, которые попали в стек последними. Чтобы использовать

второй элемент стека необходимо извлечь первый, чтобы использовать третий – необходимо извлечь первый и второй и т.д.

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

С помощью функции `sys.getrecursionlimit()`.

#### **5. Что произойдёт если число рекурсивных вызовов превысит максимальную глубину рекурсии в языке Python?**

Возникает исключение `RunTime`. То есть происходит ошибка выполнения программы.

#### **6. Как изменить максимальную глубину рекурсии в языке Python?**

С помощью функции `sys.setrecursionlimit()`, где в качестве аргумента функции указывается нужная глубина рекурсии.

#### **7. Каково назначение декоратора `lru_cache`?**

По сути, данный декоратор уменьшает количество повторяющихся вычислений, что немного ускоряет время работы программы (в некоторых случаях в разы).

#### **8. Что такое хвостовая рекурсия? Как проводится оптимизация хвостовой рекурсии?**

Хвостовая рекурсия — это рекурсивная функция, в которой функция вызывает себя в конце («хвосте») функции, в которой после возврата рекурсивного вызова не выполняется никаких вычислений. Многие компиляторы оптимизируют изменение рекурсивного вызова на хвостовой рекурсивный или итеративный вызов.

**Вывод:** в ходе работы были изучены рекурсивные функции в языке программирования Python 3.10.