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

Институт цифрового развития
Кафедра инфокоммуникаций

«Рекурсия в языке Python»

ОТЧЕТ
по лабораторной работе №12
дисциплины
«Основы программной инженерии»

Выполнил:
Борсуков Владислав Олегович
2 курс, группа ПИЖ-б-о-21-1,
09.03.04 «Программная инженерия», направленность (профиль) «Разработка и сопровождение программного обеспечения», очная форма обучения

_____
(подпись)

Проверил:

_____
(подпись)

Отчет защищен с оценкой_____Дата защиты_____

Ставрополь, 2022 г.

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. Во сколько раз изменится скорость работы рекурсивных версий функций factorial и fib при использовании декоратора lru_cache? Приведите в отчет и обоснуйте полученные результаты.

```python
# -*- coding: utf-8 -*-

import timeit
from functools import lru_cache


def factorial_iter(n):
    product = 1
    while n > 1:
        product *= n
        n -= 1
    return product


def factorial_recurse(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return n * factorial_recurse(n - 1)


@lru_cache
def factorial_rec_lru(n):
    if n == 0:
        return 1
    elif n == 1:
        return 1
    else:
        return n * factorial_recurse(n - 1)


if __name__ == '__main__':
    print("Время, затраченное на итеративную версию")
    print(f'{timeit.timeit(lambda: factorial_iter(500), number=10000)},\n')
    print("Время, затраченное на рекурсивную версию")
    print(f'{timeit.timeit(lambda: factorial_recurse(500), number=10000)},\n')
    print("Время, затраченное на рекурсивную версию с lru_cache")
    print(timeit.timeit(lambda: factorial_rec_lru(500), number=10000))
```

Рисунок 1 – Код задания №1 (поиск факториала)

```
C:\Users\Borsukov\Desktop\LR12\PyCharm\venv\Scripts\python.exe
Время, затраченное на итеративную версию
0.8861406259820797,

Время, затраченное на рекурсивную версию
1.3383777830167674,

Время, затраченное на рекурсивную версию с lru_cache
0.002306718030013144
```

Рисунок 2 – Результат работы кода задания №1 (поиск факториала)

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

import timeit
from functools import lru_cache


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


def fib_recurse(n):
    if n == 0 or n == 1:
        return n
    else:
        return fib_recurse(n - 2) + fib_recurse(n - 1)


@lru_cache
def fib_rec_lru(n):
    if n == 0 or n == 1:
        return n
    else:
        return fib_rec_lru(n - 2) + fib_rec_lru(n - 1)


if __name__ == '__main__':
    print("Время, затраченное на итеративную версию")
    print(f'{timeit.timeit(lambda: fib_iter(15), number=10000)},\n')
    print("Время, затраченное на рекурсивную версию")
    print(f'{timeit.timeit(lambda: fib_recurse(15), number=10000)},\n')
    print("Время, затраченное на рекурсивную версию с lru_cache")
    print(timeit.timeit(lambda: fib_rec_lru(15), number=10000))
```

Рисунок 3 – Код задания №1 (числа Фибоначчи)

```
F:\GitLabы\Lab-2.9\venv\Scripts\python.exe F:\GitLabы\Lab-2.9\1_task_fib.py
Time for iterative version
0.006418199998734053,

Time for recurse version
1.2961012999985542,

Time for recurse_lru version
0.0011145000025862828
```

Рисунок 4 – Результат работы кода задания №1 (числа Фибоначчи)

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.

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

import timeit


class recursion(object):
    def __init__(self, func):
        self.func = func

    def __call__(self, *args, **kwargs):
        result = self.func(*args, **kwargs)
        while callable(result):
            result = result()
        return result

    def call(self, *args, **kwargs):
        return lambda: self.func(*args, **kwargs)


@recursion
def factorial_opt(n, acc=1):
    if n == 0:
        return acc
    return factorial(n - 1, n * acc)


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


if __name__ == '__main__':
    print("Время работы кода с использованием интроспекции")
    print(f'{timeit.timeit(lambda: factorial_opt(250), number=10000)}\n')
    print("Время работы кода без использования интроспекции")
    print(timeit.timeit(lambda: factorial(250), number=10000))
```

Рисунок 5 – Код задания №2

```
Время работы кода с использованием интроспекции
1.3844911579999462

Время работы кода без использования интроспекции
1.304327760999513
```

Рисунок 6 – Результат работы кода задания №2

Индивидуальное задание: Напишите рекурсивную функцию, проверяющую правильность расстановки скобок в строке. При правильной расстановке выполняются условия:

- количество открывающих и закрывающих скобок равно.
- внутри любой пары открывающая – соответствующая закрывающая скобка, скобки расставлены правильно.

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


def check_skobki(s, skobki=0):
    if skobki < 0:
        return False
    if s:
        if s[0] == '(':
            skobki += 1
        elif s[0] == ')':
            skobki -= 1
        return check_skobki(s[1:], skobki)
    return skobki == 0


if __name__ == '__main__':
    if check_skobki(input("Введите строку: ")):
        print("Скобки раставлены правильно")
    else:
        print("Скобки раставлены не правильно")
```

```
if __name__ == '__main__' › else
Run: Individual
C:\Users\Borsukov\Desktop\LR12\PyCharm\venv\Scri
Введите строку: )неправильные скобки(
Скобки раставлены не правильно
```

Рисунок 7 – Код и результат работы программы индивидуального задания

Контрольные вопросы

1. Для чего нужна рекурсия?

Функция может содержать вызов других функций. В том числе процедура может вызвать саму себя.

2. Что называется базой рекурсии?

У рекурсии, как и у математической индукции, есть база — аргументы, для которых значения функции определены

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

Максимальная глубина рекурсии ограничена движком JavaScript. Точно можно рассчитывать на 10000 вложенных вызовов, некоторые интерпретаторы допускают и больше, но для большинства из них 100000 вызовов – за пределами возможностей. Существуют автоматические оптимизации, помогающие избежать переполнения стека вызовов («оптимизация хвостовой рекурсии»), но они ещё не поддерживаются везде и работают только для простых случаев.

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

Функция sys. getrecursionlimit() возвращает текущее значение предела рекурсии, максимальную глубину стека интерпретатора Python.

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

Ошибка RunTimeError

6. Как изменить максимальную глубину рекурсии в языке Python?

С помощью функции setrecursionlimit() модуля sys

7. Каково назначение декоратора lru_cache ?

Декоратор @lru_cache() модуля functools оборачивает функцию с переданными в нее аргументами и запоминает возвращаемый результат соответствующий этим аргументам. Такое поведение может сэкономить время и ресурсы, когда дорогая или связанная с вводом/выводом функция периодически вызывается с одинаковыми аргументами.

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. Оптимизация хвостовой рекурсии выглядит так:

```
class recursion(object):
    "Can call other methods inside..."
    def __init__(self, func):
        self.func = func

    def __call__(self, *args, **kwargs):
        result = self.func(*args, **kwargs)
        while callable(result): result = result()
        return result

    def call(self, *args, **kwargs):
        return lambda: self.func(*args, **kwargs)


@recursion
def sum_natural(x, result=0):
    if x == 0:
        return result
    else:
        return sum_natural.call(x - 1, result + x)

# Даже такой вызов не заканчивается исключением
# RuntimeError: maximum recursion depth exceeded
print(sum_natural(1000000))
```