

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение

высшего образования

«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Институт цифрового развития

Кафедра информационных систем и технологий

Отчет по лабораторной работе №12.

Дисциплина: «Основы программной инженерии»

Выполнил:

Студент группы ПИЖ-б-о-22-1,

направление подготовки: 09.03.04

«Программная инженерия»

ФИО: Гуртовой Ярослав Дмитриевич

Проверил:

Воронкин Р. А.

Ставрополь 2023

Тема: Лабораторная работа 2.9. Рекурсия в языке Python.

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

Выполнение работы:

1. Изучил теоретический материал работы.
2. Создал репозиторий на git.hub.

Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)

Required fields are marked with an asterisk ().*

Owner * **Repository name ***

KingItProgger / lr-2.9

✓ lr-2.9 is available.

Great repository names are short and memorable. Need inspiration? How about [jubilant-octo-funicular](#) ?

Description (optional)

☒ **Public**
Anyone on the internet can see this repository. You choose who can commit.

☐ **Private**
You choose who can see and commit to this repository.

Initialize this repository with:

☒ **Add a README file**
This is where you can write a long description for your project. [Learn more about READMEs.](#)

Add .gitignore

.gitignore template: Python

Choose which files not to track from a list of templates. [Learn more about ignoring files.](#)

Choose a license

License: MIT License

A license tells others what they can and can't do with your code. [Learn more about licenses.](#)

This will set **main** as the default branch. Change the default name in your [settings](#).

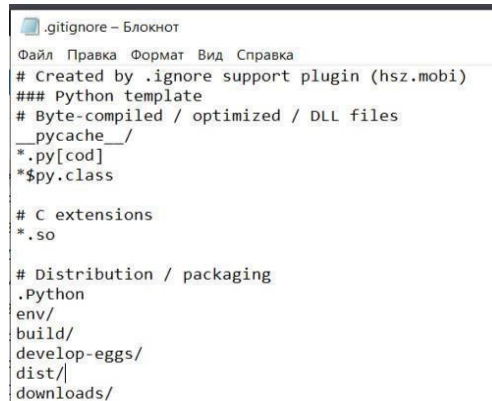
Рисунок 1 – создание репозитория

3. Клонировал репозиторий.

```
PS C:\Users\User\Desktop\учеба\Зсемак\змій\lr_2.9> git clone https://github.com/hsz/mobi
Cloning into 'lr-2.9'...
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 5 (delta 0), reused 0 (delta 0), pack-reused 0
Receiving objects: 100% (5/5), done.
PS C:\Users\User\Desktop\учеба\Зсемак\змій\lr_2.9>
```

Рисунок 2 – клонирование репозитория 4.

4. Дополнить файл gitignore необходимыми правилами.



```
.gitignore – Блокнот
Файл Правка Формат Вид Справка
# Created by .ignore support plugin (hsz.mobi)
### Python template
# Byte-compiled / optimized / DLL files
__pycache__/
*.py[cod]
*$py.class

# C extensions
*.so

# Distribution / packaging
.Python
env/
build/
develop-eggs/
dist/
downloads/
```

Рисунок 3 – .gitignore для IDE PyCharm

5. Организовать свой репозиторий в соответствии с моделью ветвления git-flow.

```
PS C:\Users\User\Desktop\учеба\Зсемак\змій\lr_2.9\lr-2.9> git checkout -b develop
Switched to a new branch 'develop'
PS C:\Users\User\Desktop\учеба\Зсемак\змій\lr_2.9\lr-2.9>
```

Рисунок 4 – создание ветки develop

6. 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 измениться скорость работы рекурсивных версий функций `factorial` и `fib` при использовании декоратора `lru_cache` ? Приведите в отчет и обоснуйте полученные результаты.

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

from timeit import timeit
from functools import lru_cache

import sys

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

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

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

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

if __name__ == "__main__":
    sys.setrecursionlimit(5000)
    n = 35
    setup1 = """from __main__ import fib_recursion"""
    setup2 = """from __main__ import factorial_recursion"""
    timer = timeit(stmt=f'fib_recursion({n})', number=10, setup=setup1)
    print('Время выполнения рекурсивной функции с @lru_cache: ', {timer})
    timer = timeit(stmt=f'factorial_recursion({n})', number=10, setup=setup2)
    print('Время выполнения рекурсивной функции с @lru_cache: ', {timer})
```

Рисунок 5 – пример 1

7. 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.

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

import sys
from timeit import timeit

class TailRecurseException(Exception):
    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:
                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, nxt=1):
    if i == 0:
        return current
    else:
        return fib(i - 1, nxt, current + nxt)

```

```
if __name__ == '__main__':  
    n = 30  
    setup1 = """from __main__ import factorial"""  
    setup2 = """from __main__ import fib"""  
    timer = timeit(stmt=f'factorial({n})', number=10, setup=setup1)  
    print(f"Время выполнения функции factorial(): {timer}")  
    timer = timeit(stmt=f'fib({n})', number=10, setup=setup2)  
    print(f"Время выполнения функции fib(): {timer}")
```

Рисунок 6 – код программы

```
Время выполнения функции factorial(): 0.0005292999994708225  
Время выполнения функции fib(): 0.000487600002088584  
  
Process finished with exit code 0
```

Рисунок 7 – выполнение программы

Индивидуальное задание

8. Создайте рекурсивную функцию, печатающую все подмножества множества $\{1, 2, \dots, N\}$

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

def generate(n, t=[]):
    if n == 0:
        print(t)
        return
    generate(n - 1, t + [n])
    generate(n - 1, t)

if __name__ == '__main__':
    n = int(input("Введите значения n: "))
    generate(n)
```

Рисунок 8 – выполнение индивидуального задания

```
C:\Users\User\AppData\Local\Programs\Python\Python311\python.exe "C:/Users/User/Desktop/ОПИ/лр 2.9/my_task.py"
Введите значения n: 3
[3, 2, 1]
[3, 2]
[3, 1]
[3]
[2, 1]
[2]
[1]
[]

Process finished with exit code 0
```

Рисунок 9 – результат выполнения индивидуального задания

9. Зафиксировал все изменения в github в ветке develop.

```
PS C:\Users\User\Desktop\учеба\Зсемак\змії\lr_2.9\lr-2.9> git add .
warning: in the working copy of 'pycharm/.idea/inspectionProfiles/profiles_settings.xml'
the next time Git touches it
PS C:\Users\User\Desktop\учеба\Зсемак\змії\lr_2.9\lr-2.9> git commit -m"laba"
[develop f2a3a81] laba
 8 files changed, 153 insertions(+)
 create mode 100644 pycharm/.idea/.gitignore
 create mode 100644 pycharm/.idea/inspectionProfiles/profiles_settings.xml
 create mode 100644 pycharm/.idea/lr 2.9.iml
 create mode 100644 pycharm/.idea/misc.xml
 create mode 100644 pycharm/.idea/modules.xml
 create mode 100644 pycharm/fib.py
 create mode 100644 pycharm/my_task.py
 create mode 100644 pycharm/task2.py
PS C:\Users\User\Desktop\учеба\Зсемак\змії\lr_2.9\lr-2.9>
```

Рисунок 10 – фиксация изменений в ветку develop

10. Слил ветки.

```
PS C:\Users\User\Desktop\учеба\Зсемак\змії\lr_2.9\lr-2.9> git checkout main
Switched to branch 'main'
Your branch is up to date with 'origin/main'.
PS C:\Users\User\Desktop\учеба\Зсемак\змії\lr_2.9\lr-2.9> git merge develop
Updating d9d15f5..f2a3a81
Fast-forward
 pycharm/.idea/.gitignore           | 3 ++
 .../.idea/inspectionProfiles/profiles_settings.xml | 6 +++
 pycharm/.idea/lr 2.9.iml           | 8 +++
 pycharm/.idea/misc.xml             | 4 ++
 pycharm/.idea/modules.xml          | 8 +++
 pycharm/fib.py                     | 50 +++++
 pycharm/my_task.py                 | 13 +++++
 pycharm/task2.py                   | 61 +++++
 8 files changed, 153 insertions(+)
 create mode 100644 pycharm/.idea/.gitignore
 create mode 100644 pycharm/.idea/inspectionProfiles/profiles_settings.xml
 create mode 100644 pycharm/.idea/lr 2.9.iml
 create mode 100644 pycharm/.idea/misc.xml
 create mode 100644 pycharm/.idea/modules.xml
 create mode 100644 pycharm/fib.py
 create mode 100644 pycharm/my_task.py
 create mode 100644 pycharm/task2.py
PS C:\Users\User\Desktop\учеба\Зсемак\змії\lr_2.9\lr-2.9>
```

Рисунок 11 – сливание ветки develop в ветку main

Вывод: приобрел навыки по работе с рекурсивными функциями при написании программ с помощью языка программирования Python версии 3.x.

Контрольные вопросы:

1. Для чего нужна рекурсия?

Рекурсия — это прием в программировании, когда функция вызывает сама себя. Этот подход может быть использован для решения задач, которые могут быть разбиты на более простые подзадачи. Рекурсия часто приводит к более чистому и понятному коду, особенно в случаях, когда задача имеет структуру, поддерживающую деление на подзадачи.

2. Что называется базой рекурсии?

Утверждение, if $n == 0$: return 1

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

Стек программы (или стек вызовов) - это структура данных, используемая для управления вызовами функций в программе. Каждый раз, когда функция вызывается, информация о текущем состоянии функции (локальные переменные, адрес возврата и т. д.) сохраняется в стеке. Когда функция завершает выполнение, эта информация удаляется из стека, и управление возвращается к вызывающей функции. Основные операции со стеком - это "положить" (push) и "взять" (pop). Это относится к добавлению информации в стек при вызове функции и удалению ее при завершении функции. Вот как происходит использование стека программы при вызове функций:

- Когда функция вызывается, текущее состояние функции (локальные переменные, адрес возврата и т. д.) помещается в вершину стека.
- Текущая функция становится активной функцией, и управление передается в вызываемую функцию.
- Локальные переменные функции хранятся в том же фрейме стека, что и другая информация о состоянии функции.
- Эти переменные доступны только в пределах текущей функции.
- При завершении функции информация из вершины стека удаляется (pop), возвращая управление к вызывающей функции.
- Адрес возврата используется для определения, куда вернуть управление.
- Если функция вызывает саму себя (рекурсия), каждый вызов функции создает новый фрейм стека.
- Каждый уровень рекурсии имеет свои локальные переменные и адреса

возврата.

Стек вызовов предоставляет эффективный механизм управления выполнением программы, обеспечивая правильный порядок вызова и возврата функций. Он также играет важную роль в обработке исключений и управлении памятью.

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: Maximum Recursion Depth Exceeded`

6. Как изменить максимальную глубину рекурсии в языке Python?

Можно изменить предел глубины рекурсии с помощью вызова:
`sys.setrecursionlimit(limit)`

7. Каково назначение декоратора `lru_cache`?

Декоратор `lru_cache` (Least Recently Used Cache) предназначен для кэширования результатов выполнения функций с использованием стратегии "Наименее недавно использованный" (LRU). Он сохраняет результаты выполнения функции для предотвращения повторных вычислений, когда те же самые входные значения встречаются повторно.

Когда функция вызывается с определенными аргументами, `lru_cache` сохраняет результат выполнения функции в кэше. Если функция вызывается с теми же аргументами позже, она возвращает сохраненный результат, минуя фактическое выполнение функции. Это может существенно ускорить выполнение функций, требующих вычислений, которые могут быть дорогими по времени.

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

Хвостовая рекурсия - это форма рекурсии, при которой рекурсивный вызов является последней операцией в функции. Такой вызов расположен в "хвосте" функции, то есть не сопровождается последующими операциями возврата или обработки результатов. Важным свойством хвостовой рекурсии является то, что она может быть оптимизирована компилятором или интерпретатором, сокращая использование стека вызовов и уменьшая вероятность переполнения стека.

Оптимизация хвостовых вызовов, называемая также "оптимизацией хвостовой рекурсии" или "хвостовой рекурсивной оптимизацией", заключается в том, что компилятор или интерпретатор понимают, что после хвостового рекурсивного вызова нет необходимости сохранять текущий контекст выполнения (значения

переменных, адрес возврата и т. д.) на стеке вызовов

