

Лабораторная работа №4

Выполнил Эсеналиев Арсен

ИВТ-б-о-21-1

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

1. Создал общедоступный репозиторий на GitHub с MIT

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

Owner * **Repository name ***

Arsen445 / LB2.9 ✓

Great repository names are short and memorable. Need inspiration? How about [literate-fiesta?](#)

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:
Skip this step if you're importing an existing repository.

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

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

.gitignore template: None ▾

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

License: MIT License ▾

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

You are creating a public repository in your personal account.

Create repository

2. Выполнил клонирование созданного репозитория.

```

C:\Users\GG_Force>d:
D:\>cd REP4
D:\REP4>git clone https://github.com/Arsen445/LB2.9.git
Cloning into 'LB2.9'...
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 4 (delta 0), reused 0 (delta 0), pack-reused 0
Receiving objects: 100% (4/4), done.
D:\REP4>

```

3. Организовал свой репозиторий в соответствии с моделью ветвления git-flow. (Перешел с главной main на develop)

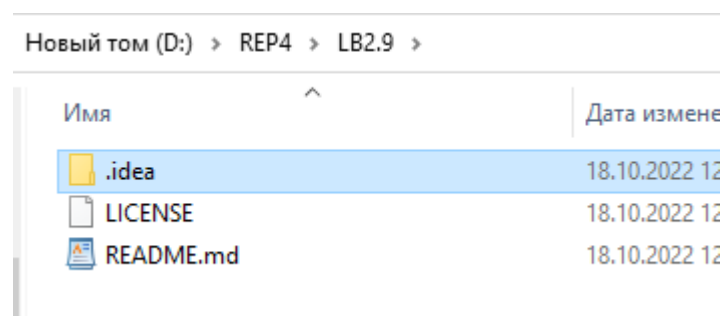
```

D:\REP4\LB2.9>git flow init
Which branch should be used for bringing forth production releases?
- main
Branch name for production releases: [main]
Branch name for "next release" development: [develop]

How to name your supporting branch prefixes?
Feature branches? [feature/]
Bugfix branches? [bugfix/]
Release branches? [release/]
Hotfix branches? [hotfix/]
Support branches? [support/]
Version tag prefix? []
Hooks and filters directory? [D:/REP4/LB2.9/.git/hooks]
D:\REP4\LB2.9>_

```

4. Создал проект PyCharm в папке репозитория.



5. Дополнил файл .gitignore необходимыми правилами для работы с IDE PyCharm.

```
# Created by https://www.toptal.com/developers/gitignore/api/python,pycharm
# Edit at https://www.toptal.com/developers/gitignore?templates=python,pycharm

### PyCharm ###
# Covers JetBrains IDEs: IntelliJ, RubyMine, PhpStorm, AppCode, PyCharm, CLion, Android Studio
# Reference: https://intellij-support.jetbrains.com/hc/en-us/articles/206544839

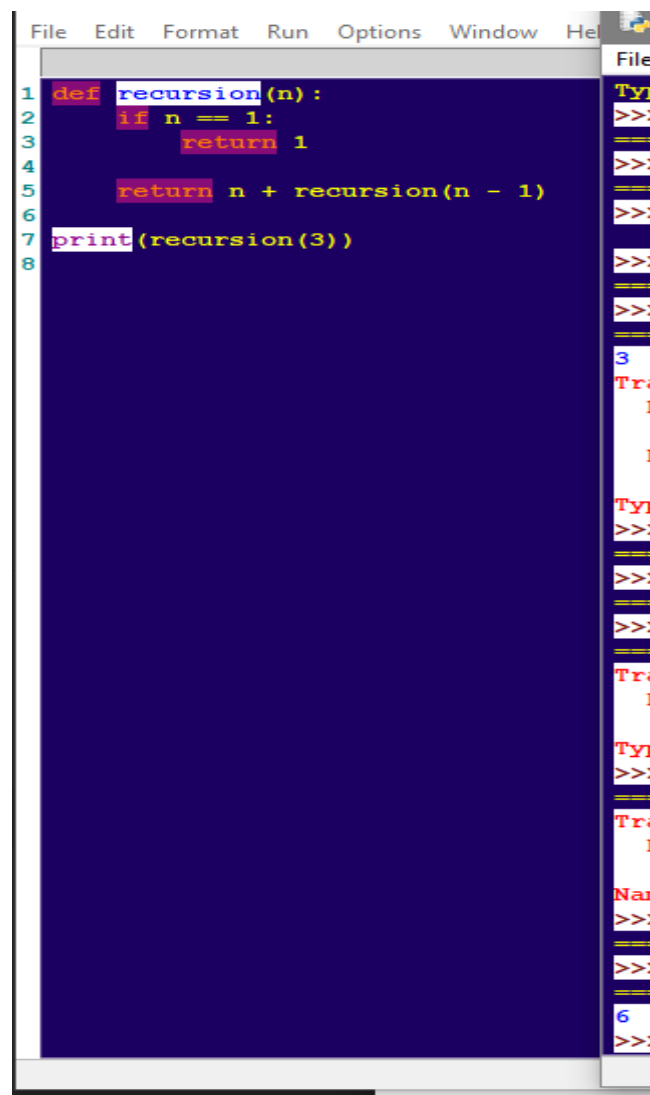
# User-specific stuff
.idea/**/workspace.xml
.idea/**/tasks.xml
.idea/**/usage.statistics.xml
.idea/**/dictionaries
.idea/**/shelf

# AWS User-specific
.idea/**/aws.xml

# Generated files
.idea/**/contentModel.xml

# Sensitive or high-churn files
.idea/**/dataSources/
```

6. Проработал пример лабораторной работы.



The screenshot shows a code editor window with a menu bar (File, Edit, Format, Run, Options, Window, Help) and a toolbar. The code is written in Python and defines a recursive function named `recursion`. The function takes an argument `n` and returns the sum of `n` and the result of `recursion(n - 1)` if `n` is not 1. If `n` is 1, it returns 1. The function is called with `recursion(3)` and the result is printed. The code is as follows:

```
1 def recursion(n):
2     if n == 1:
3         return 1
4     return n + recursion(n - 1)
5
6 print(recursion(3))
7
8
```

The right sidebar shows a file explorer with a tree view. The tree view shows a folder named `3` containing a file named `Tra`. The file `Tra` is selected, and its details are shown in the right pane. The details pane shows the file type as `Text` and the file size as `6`.

7. 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..

```
File Edit Format Run Options Window Help
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import timeit

time fact_rec = '''
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
'''

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

time fact_itr = '''
def factorial(n):
    s = 1
    for i in range(2, s+1):
        s *= i
    return s
'''

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

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

time fib_lru = '''
from functools import lru_cache
@lru_cache
def fib(n):
    if n == 0 or n == 1:
        return n
'''
```

8. 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.

```
Zadanie2.py - D:\REP4\LB2.9\Zadanie2.py (3.9.13)
File Edit Format Run Options Window Help

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

fib_s_intr = '''
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('Время вычисления факториала:', timeit.timeit(setup=fact_bez_intr, number=10))
    print('Время вычисления числа Фибоначи:', timeit.timeit(setup=fib_bez_intr, number=10))
    print('Время вычисления факториала с интроспекцией стека:', timeit.timeit(setup=fact_s_intr, number=10))
    print('Время вычисления числа Фибоначи с интроспекцией стека:', timeit.timeit(setup=fib_s_intr, number=10))
```

```
IDLE Shell 3.9.13
File Edit Shell Debug Options Window Help
Python 3.9.13 (tags/v3.9.13:6de2ca5, May 17 2022, 16:36:42) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: D:\REP4\LB2.9\Zadanie2.py =====
Результат факториала: 3.000000000086267e-07
Результат числа фибоначи: 3.000000000086267e-07
Результат факториала с интроспекцией стека: 2.9999999995311555e-07
Результат числа фибоначи с интроспекцией стека: 3.000000000086267e-07
>>>
===== RESTART: D:\REP4\LB2.9\Zadanie2.py =====
Время вычисления факториала: 3.000000000086267e-07
Время вычисления числа фибоначи: 4.999999999588667e-07
Время вычисления факториала с интроспекцией стека: 4.999999999588667e-07
Время вычисления числа фибоначи с интроспекцией стека: 3.000000000086267e-07
>>>
```

9. Индивидуальное задание

13. Напишите программу вычисления функции Аккермана для всех неотрицательных целых аргументов m и n :

$$A(m, n) = \begin{cases} A(0, n) = n + 1 \\ A(m, 0) = A(m - 1, 1), & m \\ A(m, n) = A(m - 1, A(m, n - 1)), & m, n > 0. \end{cases} \quad (3)$$

```
ind.py - D:\REP4\LB2.9\ind.py (3.9.13)
File Edit Format Run Options Window Help
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

if __name__ == '__main__':
    def akkerman(m, n):
        if m == 0 and n > 0:
            return n + 1
        elif m > 0 and n == 0:
            return akkerman(m - 1, 1)
        elif m > 0 and n > 0:
            return akkerman(m - 1, akkerman(m, n - 1))
        else:
            return None

    print(akkerman(2, 3))
```

10. Зафиксируйте сделанные изменения в репозитории.

```
Git CMD
--exec <receive-pack> receive pack program
-u, --set-upstream set upstream for git pu
--progress force progress reportin
--prune prune locally removed r
--no-verify bypass pre-push hook
--follow-tags push missing but releva
--signed[=(yes|no|if-asked)] GPG sign the push
--atomic request atomic transact
-o, --push-option <server-specific> option to transmit
-4, --ipv4 use IPv4 addresses only
-6, --ipv6 use IPv6 addresses only

D:\REP4\LB2.9>git push --all
Enumerating objects: 9, done.
Counting objects: 100% (9/9), done.
Delta compression using up to 8 threads
Compressing objects: 100% (8/8), done.
Writing objects: 100% (8/8), 536.14 KiB | 26.81 M
Total 8 (delta 0), reused 0 (delta 0), pack-reuse
To https://github.com/Arsen445/LB2.9.git
ade708c..087222a develop -> develop

D:\REP4\LB2.9>

nothing added to commit but untracked files present
D:\REP4\LB2.9>git add .
D:\REP4\LB2.9>git status
On branch develop
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    new file:   Zadanie2.py
    new file:   ind.py
    new file:   prim1.py
    new file:   zadanie1.py
    new file:   "~$\320\233\320\2214 .docx"
    new file:   "\320\233\320\2214 .docx"

D:\REP4\LB2.9>git commit -m "last"2
```

11. Выполните слияние ветки для разработки с веткой main/master.

```
D:\REP4\LB2.9>git merge develop
Already up to date.

D:\REP4\LB2.9>
```

Контрольные вопросы:

1. Для чего нужна рекурсия?

В программировании рекурсия — вызов функции (процедуры) из неё же самой, непосредственно (простая рекурсия) или через другие функции (сложная или косвенная рекурсия). Рекурсивная программа позволяет описать повторяющееся или даже потенциально бесконечное вычисление, причём без явных повторений частей программы и использования циклов.

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 — это структура данных, в которой элементы хранятся в порядке поступления. Стек хранит последовательность данных. Связаны данные так: каждый элемент указывает на тот, который нужно использовать следующим. Это линейная связь — данные идут друг за другом и нужно брать их по очереди. Из середины стека брать нельзя. Главный принцип работы стека — данные, которые попали в стек недавно, используются первыми. Чем раньше попал — тем позже используется. После использования элемент стека исчезает, и верхним становится следующий элемент.

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

Функция `sys.getrecursionlimit()` возвращает текущее значение предела рекурсии, максимальную глубину стека интерпретатора Python. Этот предел предотвращает бесконечную рекурсию от переполнения стека языка C и сбоя Python. Это значение может быть установлено с помощью `sys`.

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 `RunTime`.

6. Как изменить максимальную глубину рекурсии в языке Python?

С помощью `sys.setrecursionlimit(число)`.

7. Каково назначение декоратора `lru_cache`?

Функция `lru_cache` предназначена для мемоизации (предотвращения повторных вычислений), т. е. кэширует результат в памяти. Полезный инструмент, который уменьшает количество лишних вычислений.

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. Типовой механизм реализации вызова функции основан на сохранении адреса возврата, параметров и локальных переменных функции в стеке и выглядит следующим образом:

1. В точке вызова в стек помещаются параметры, передаваемые функции, и адрес возврата.
2. Вызываемая функция в ходе работы размещает в стеке собственные локальные переменные.
3. По завершении вычислений функция очищает стек от своих локальных переменных, записывает результат (обычно — в один из регистров процессора).
4. Команда возврата из функции считывает из стека адрес возврата и выполняет переход по этому адресу. Либо непосредственно перед, либо сразу после возврата из функции стек очищается от параметров.