МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ

ФЕДЕРАЦИИ

Федеральное государственное автономное

образовательное учреждение высшего образования

«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Кафедра инфокоммуникаций

«Рекурсия в языке Python»

Отчет по лабораторной работе № 2.9

по дисциплине «Основы программной инженерии»

Выполнил студент группы ПИЖ-б-о-21-1

Халимендик Я. Д. « » 2022г.

Подпись студента_____

Работа защищена « »_____20__г.

Проверил Воронкин Р.А. _____
(подпись)

Ставрополь 2022

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

Ход работы:

1. Изучить теоретический материал работы.

2. Создать общедоступный репозиторий на GitHub, в котором будет использована лицензия IT и язык программирования Python.

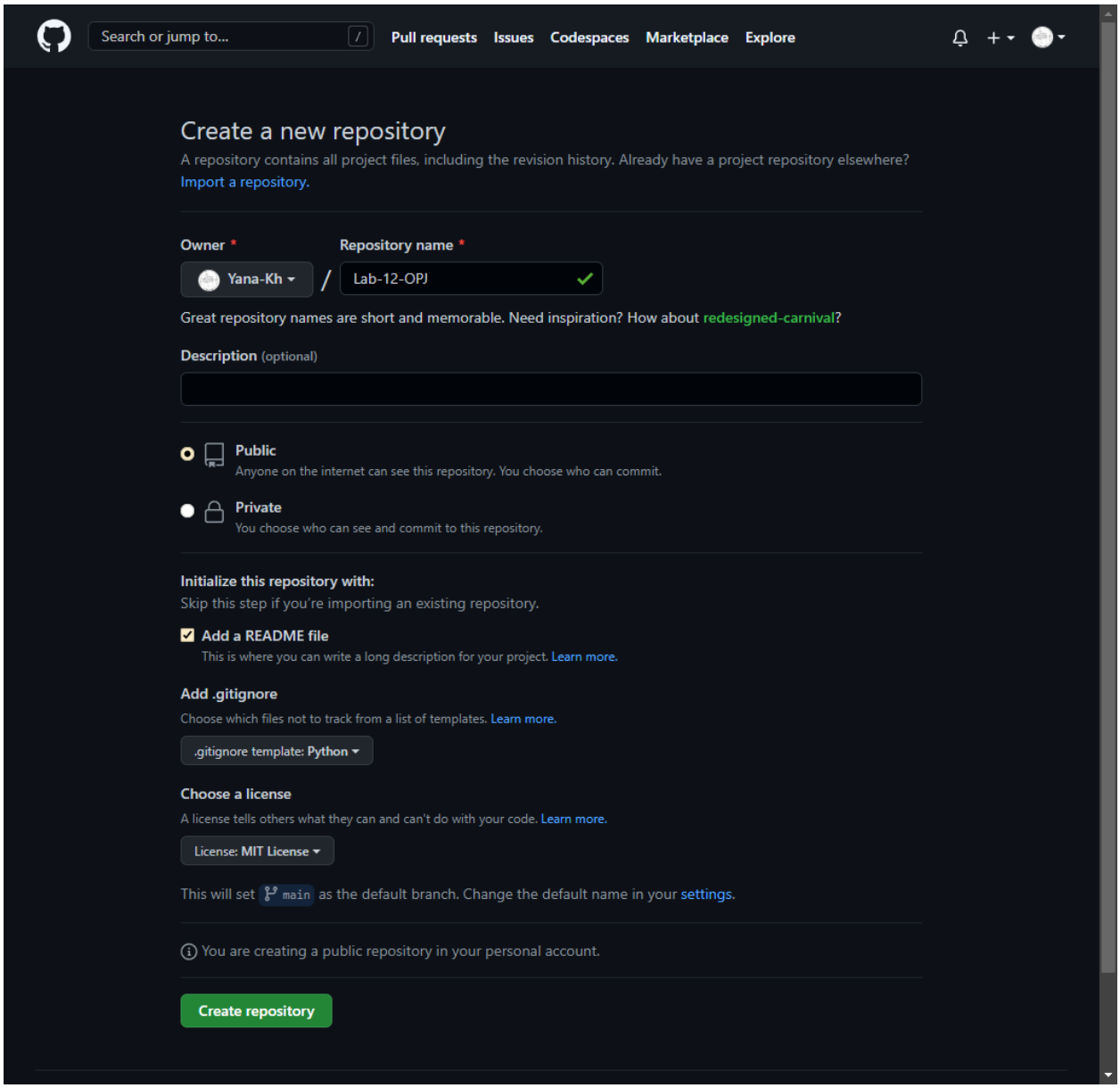

Рисунок 1 – Создание репозитория

3. Выполните клонирование созданного репозитория.

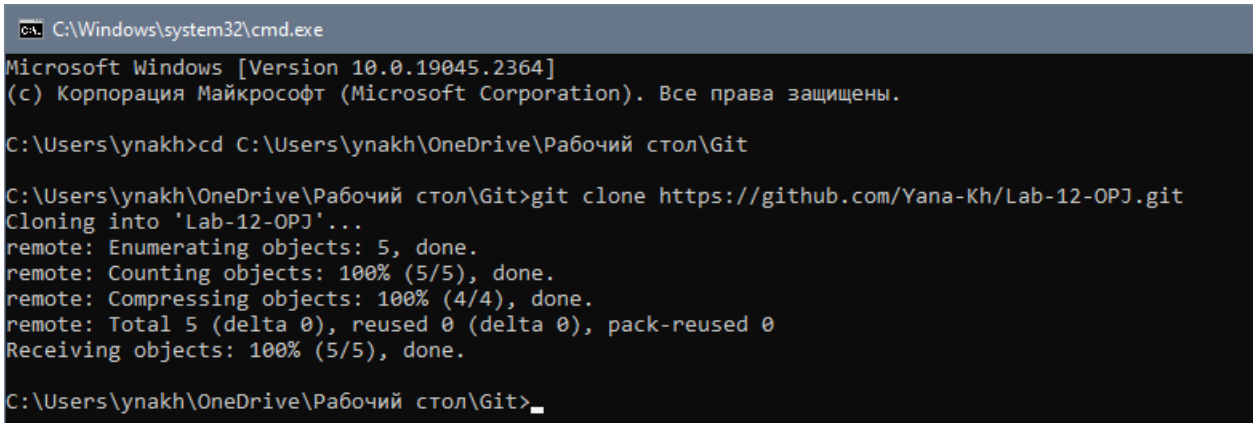

Рисунок 2 – Клонирование репозитория

4. Дополните файл .gitignore необходимыми правилами для работы с IDE PyCharm.

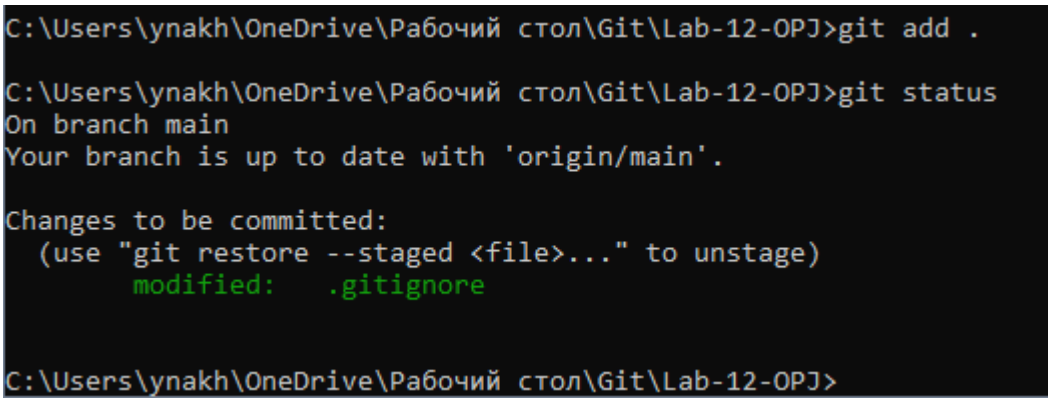

Рисунок 3 – Дополнение файла .gitignore

5. Организуйте свой репозиторий в соответствие с моделью ветвления git-flow.

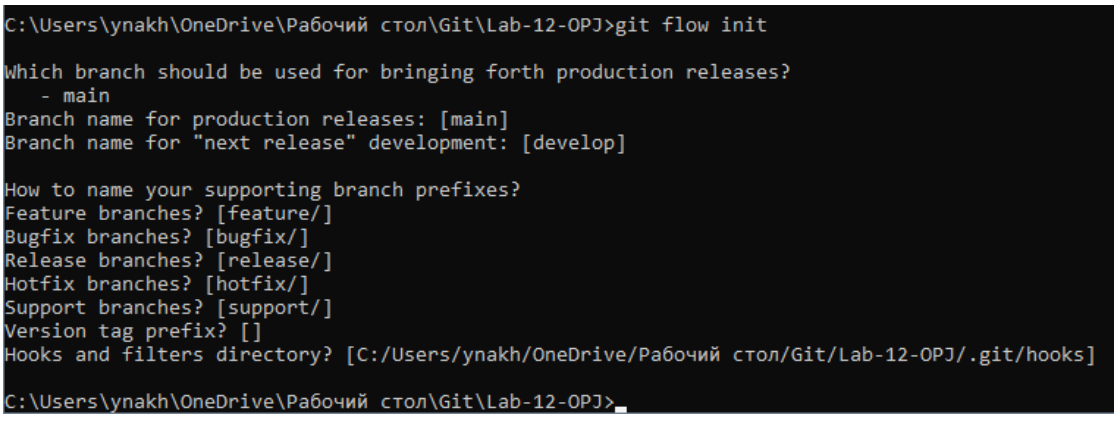

Рисунок 4 – Организация репозитория в соответствии с моделью git-flow

6. Создайте проект PyCharm в папке репозитория.

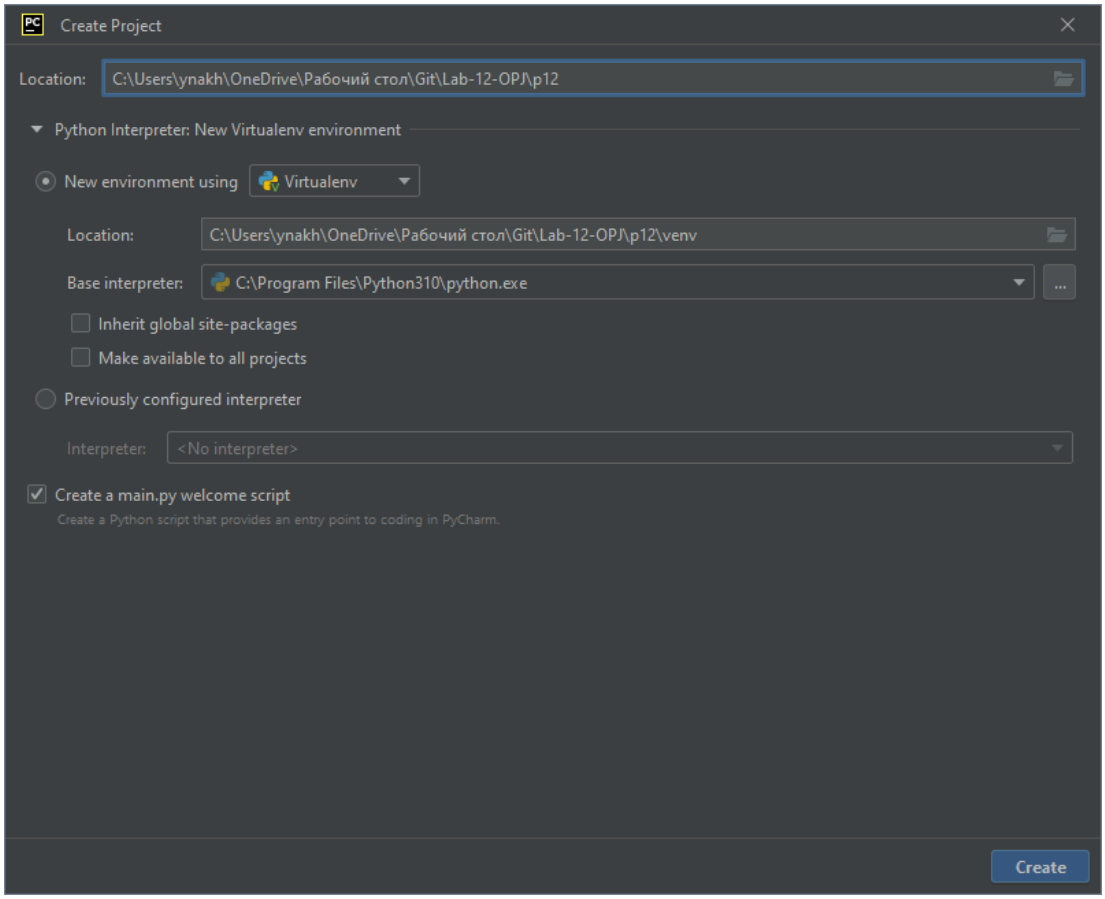

Рисунок 5 – Создание проекта PyCharm в папке репозитория

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.

Этот модуль предоставляет простой способ определения времени выполнения небольших фрагментов кода на Python. Он имеет как интерфейс командной строки, так и вызываемый. Это позволяет избежать ряда распространенных ловушек для измерения времени выполнения.

Модуль определяет три удобные функции и открытый класс.

Синтаксис:

timeit.timeit(stmt, setup,timer, number), где

- **stmt**: это код, для которого вы хотите измерить время выполнения. Значение по умолчанию - “pass”.
- **setup**: здесь будут детали настройки, которые необходимо выполнить перед stmt. Значение по умолчанию - “pass”.
- **timer**: это будет иметь значение таймера, timeit() уже имеет значение по умолчанию, и мы можем его игнорировать.
- **number**: stmt будет выполняться в соответствии с номером, указанным здесь. Значение по умолчанию - 1000000.

Для работы с timeit() нам нужно импортировать соответствующий модуль.

Важно, модулем timeit ваш код выполняется в другом пространстве имен. Таким образом, он не распознает функции, которые вы определили в своем глобальном пространстве имен. Для того, чтобы timeit распознавал ваши функции, вам необходимо импортировать его в то же пространство имен. Вы можете добиться этого, передав from __main__ import func_namек аргументу setup

Код для fib:

```
import timeit
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)


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


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

```
setup_code_1 = """
from __main__ import fib
n = 6
"""

setup_code_2 = """
from __main__ import fib_r
n = 6
"""

setup_code_3 = """
from __main__ import fib_w
n = 6
"""

if __name__ == "__main__":
    print("Рекурсивная функция:")
    print(timeit.timeit(stmt="fib_r(n)", setup=setup_code_2, number=10000))
    print("Итеративная функция:")
    print(timeit.timeit(stmt="fib_w(n)", setup=setup_code_3, number=10000))
    print("Рекурсивная функция с lru_cache:")
    print(timeit.timeit(stmt="fib(n)", setup=setup_code_1, number=10000))
```

```
Рекурсивная функция:
0.0033213999995496124
Итеративная функция:
0.0027098999998997897
Рекурсивная функция с lru_cache:
0.0003277999931015074
```

Рисунок 1 – Результат работы программы

Код для factorial:

```
import timeit
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)


def factorial_r(n):
    if n == 0:
        return 1
    elif n == 1:
        return 1
    else:
        return n * factorial(n - 1)
```

```python
def factorial_w(n):
    product = 1
    while n > 1:
        product *= n
        n -= 1
    return product


setup_code_1 = """
from __main__ import factorial
n = 6
"""

setup_code_2 = """
from __main__ import factorial_r
n = 6
"""

setup_code_3 = """
from __main__ import factorial_w
n = 6
"""

if __name__ == "__main__":
    print("Рекурсивная функция:")
    print(timeit.timeit(stmt="factorial_r(n)", setup=setup_code_2,
number=10000))
    print("Итеративная функция:")
    print(timeit.timeit(stmt="factorial_w(n)", setup=setup_code_3,
number=10000))
    print("Рекурсивная функция с lru_cache:")
    print(timeit.timeit(stmt="factorial(n)", setup=setup_code_1,
number=10000))
```

```
Рекурсивная функция:
0.0038047999842092395
Итеративная функция:
0.002179600007366389
Рекурсивная функция с lru_cache:
0.00033779998193494976
```

Рисунок 2 – Результат работы программы

Исходя из результатов мы видим, что рекурсивная функция выполняется медленее итеративной, при этом использование декоратора lru_cache позволяет сократить время работы рекурсивное функции в 10-11 раза.

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.

Код:

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

# Эта программа показыает 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
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



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


def fib(n):
    if n == 0 or n == 1:
```

```
        return n
    else:
        return fib(n - 2) + fib(n - 1)


@tail_call_optimized
def factorial_o(n, acc=1):
    if n == 0:
        return acc

    return factorial(n - 1, n * acc)


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


setup_code_1 = """
from __main__ import factorial
n = 100
"""

setup_code_2 = """
from __main__ import factorial_o
n = 100
"""

setup_code_3 = """
from __main__ import fib
n = 10
"""

setup_code_4 = """
from __main__ import fib_o
n = 10
"""

if __name__ == '__main__':
    print("Рекурсивная функция (fac):")
    print(timeit.timeit(stmt="factorial(n)", setup=setup_code_1,
number=10000))
    print("Оптимизированная функция (fac):")
    print(timeit.timeit(stmt="factorial_o(n)", setup=setup_code_2,
number=10000))
    print("Рекурсивная функция (fib):")
    print(timeit.timeit(stmt="fib(n)", setup=setup_code_3, number=10000))
    print("Оптимизированная функция (fib):")
    print(timeit.timeit(stmt="fib_o(n)", setup=setup_code_4, number=10000))
```

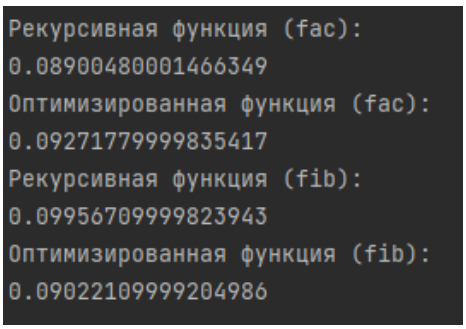

Рисунок 3 – Результат работы программы

Как мы можем увидеть, посмотрев на результат работы, сокращение времени выполнения после оптимизации есть, но не сильно большое, а иногда и вовсе нет.

9. Выполните индивидуальные задания. Приведите в отчете скриншоты работы программ решения индивидуального задания.

Вариант 32(4)

4. Создайте рекурсивную функцию, печатающую все возможные перестановки для целых чисел от 1 до N.

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

def permutations(line):
    """creating permutations"""
    if len(line) == 1:
        return [line]  # базовая рекурсия
    else:
        # создаем массив для записи перестановок
        all = []
        # первый элемент списка помещаем в a
        a = line[0]
        # все перестановки, для последовательности без 1-го эл
        per = permutations(line[1:])
        # перестановки
        for p in per:
            # использовала enumerate, чтоб не брать range(len)
            for i, item in enumerate(p):
                # создаем комбинации с разным положением a (1-ый эл.)
                tmp = p[0:i] + [a] + p[i:]
                all.append(tmp)
            all.append(p + [a])
        return all


if __name__ == "__main__":
    n = int(input("Enter n: "))
    print(permutations([i for i in range(1, n + 1)]))
```

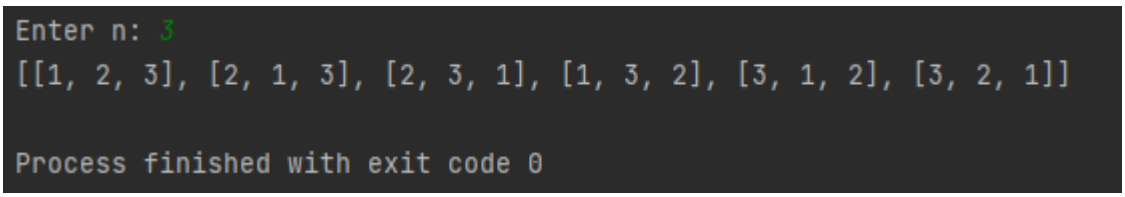

Рисунок 4 – Результат работы программы

10. Зафиксируйте сделанные изменения в репозитории.

Рисунок 5 – Фиксирование изменений в репозитории

Вопросы для защиты работы

1. Для чего нужна рекурсия?

2. Что называется базой рекурсии?

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

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

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

6. Как изменить максимальную глубину рекурсии в языке Python?

7. Каково назначение декоратора lru_cache ?

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