**Алиев Фарид Махир оглы**

**Выполнение лабораторных работ**

[**Лабораторная работа №1. Линейные конгруэнтные генераторы псевдослучайных последовательностей.** 1](#_Toc187172913)

[**Лабораторная работа №2. Сеть Фейстеля.** 3](#_Toc187172914)

[**Лабораторная работа №4. Алгоритм RC4.** 12](#_Toc187172915)

[**Лабораторная работа №5. Ассиметричные алгоритмы шифрования данных: алгоритм RSA.** 15](#_Toc187172916)

[**Лабораторная работа №6. Комбинирование симметричных и асимметричных алгоритмов.** 18](#_Toc187172917)

[**Лабораторная работа №7. Алгоритм N-хэш.** 25](#_Toc187172918)

[**Лабораторная работа №8. Алгоритм SHA.** 29](#_Toc187172919)

# **Лабораторная работа №1. Линейные конгруэнтные генераторы псевдослучайных последовательностей.**

Реализация генератора псевдослучайных чисел, код:

s1 = 1

s2 = 1

def modmult(a, b, c, m, s):

    global q

    q = s // a

    s = b \* (s - a \* q) - c \* q

    if s < 0:

        s += m

    return s

def comb\_lcg():

    global s1, s2

    s1 = modmult(53668, 40014, 12211, 2147483563, s1)

    s2 = modmult(52774, 40692, 3791, 2147483399, s2)

    z = s1 - s2

    if z < 1:

        z += 2147483562

    return z \* 4.656613e-10

def init\_lcg(init\_s1, init\_s2):

    global s1, s2

    if not (1 <= init\_s1 <= 2147483562):

        raise ValueError("s1 должен быть в диапазоне [1, 2147483562]")

    if not (1 <= init\_s2 <= 2147483398):

        raise ValueError("s2 должен быть в диапазоне [1, 2147483398]")

    s1 = init\_s1

    s2 = init\_s2

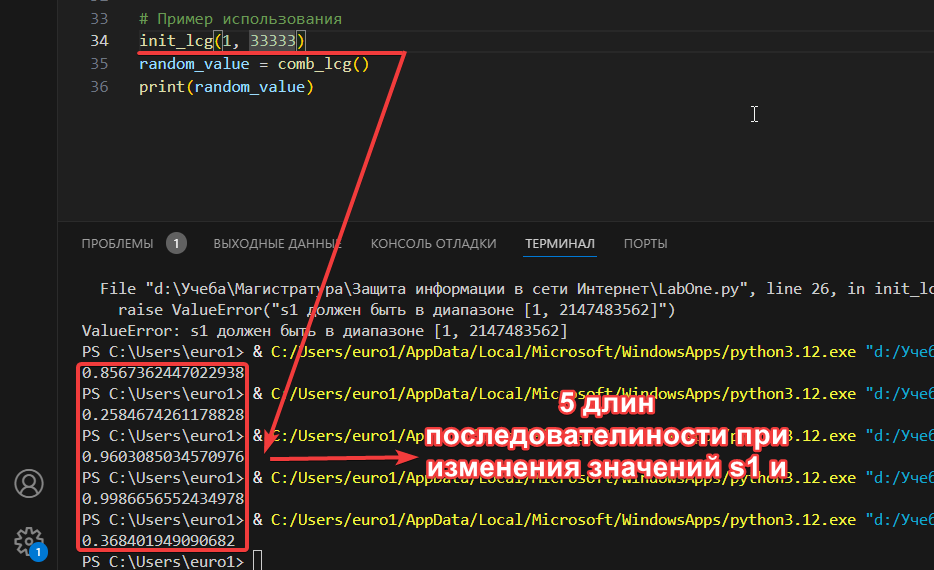
# входные значения для s1, s2

init\_lcg(3213, 123)

random\_value = comb\_lcg()

print(random\_value)

Результат выполнения:



Ответы на вопросы:

1. Особенности линейного конгруэнтного генератора: линейный конгруэнтный генератор (ЛКГ) — это один из простейших алгоритмов генерации псевдослучайных чисел. Его особенности включают математическую простоту, выраженную формулой \ (X\_{n+1} = (a \cdot X\_n + c) \mod m\), где \(a\), \(c\), и \(m\) — константы. Он легко реализуется, обладает предсказуемым поведением и периодом, который зависит от выбранных параметров. Однако его период может быть коротким, а статистические свойства иногда проявляют слабости (например, предсказуемость и коррелируемость результатов).

2. Для чего объединяются линейные конгруэнтные генераторы: объединение линейных конгруэнтных генераторов позволяет увеличить длину их периода, улучшить статистические свойства сгенерированных последовательностей и уменьшить корреляцию между результатами. При соединении нескольких ЛКГ с разными параметрами создается новый генератор, который обладает свойствами, не присущими отдельным генераторам, что ведет к более качественным псевдослучайным последовательностям, подходящим для численных методов, симуляций и других приложений.

3. Преимущества линейного конгруэнтного генератора: основные преимущества ЛКГ заключаются в его простоте и быстроте генерации. Он требует относительно небольшого объема памяти и несложной реализации. Также его скорость генерации очень высокая, что делает его подходящим для приложений, где требуется быстрое выполнение, таких как компьютерные игры и симуляции. Кроме того, при правильном выборе параметров он может обеспечивать приемлемые статистические характеристики в ряде задач.

4. Как вычисляются математическое ожидание и дисперсия: для линейного конгруэнтного генератора математическое ожидание \(E(X)\) может быть вычислено как \(E(X) = \frac{m - 1}{2}\) для диапазона от 0 до \(m - 1\). Дисперсия \ (Var(X)\) обычно рассчитывается по формуле \ (Var(X) = \frac{(m^2 - 1)}{12}\), что позволяет оценить разброс значений генератора.

# **Лабораторная работа №2. Сеть Фейстеля.**

Предварительно нужно создать файл с текстом (input), который нам нужно зашифровать, и ключ зашифрования (key), а также создать пустые файлы куда будет сохраняться расшифрованный и зашифрованный наш текст – в данном случае файлы называются encrypted и decrypted.

Реализация Сети Фейстеля, код:

import os

# Чтобы посмотреть в каком мы находимся каталоге

print("Текущий рабочий каталог:", os.getcwd())

def feistel\_encrypt(L0, R0, key, rounds, f):

    L, R = L0, R0

    for i in range(rounds):

        L, R = R, L ^ f(R, key[i])

    return L, R

def feistel\_decrypt(Ln, Rn, key, rounds, f):

    L, R = Ln, Rn

    for i in range(rounds - 1, -1, -1):

        L, R = R ^ f(L, key[i]), L

    return L, R

def unit\_function(vi):

    return vi

def multiplication\_function(vi, x):

    return vi \* x

def read\_file(filename):

    with open(filename, 'r') as file:

        content = file.read()

    return content

def write\_file(filename, content):

    with open(filename, 'w') as file:

        file.write(content)

def main():

    # Настройки пути файлов для дешифрования

    input\_text\_file = 'D:\Учеба\Магистратура\Защита информации в сети Интернет\Лаба 2\input.txt'

    key\_file = 'D:\Учеба\Магистратура\Защита информации в сети Интернет\Лаба 2\key.txt'

    encrypted\_file = 'D:\Учеба\Магистратура\Защита информации в сети Интернет\Лаба 2\encrypted.txt'

    decrypted\_file = 'D:\Учеба\Магистратура\Защита информации в сети Интернет\Лаба 2\decrypted.txt'

    # Чтение текста и ключа из файлов

    text = read\_file(input\_text\_file).strip()

    key = list(map(int, read\_file(key\_file).strip().split()))

    # Инициализация значений L0 и R0

    L0 = int.from\_bytes(text.encode(), 'big') >> (len(text) \* 4)

    R0 = int.from\_bytes(text.encode(), 'big') & ((1 << (len(text) \* 4)) - 1)

    # Выбор функции

    func\_type = input("Выберите функцию (1 - Единичная, 2 - Множение): ")

    if func\_type == '1':

        f = unit\_function

        def wrapped\_f(vi, ki):

            return f(vi)

    elif func\_type == '2':

        multiplier = int(input("Введите множитель: "))

        f = lambda vi, x: multiplication\_function(vi, multiplier)

        def wrapped\_f(vi, ki):

            return f(vi, ki)

    else:

        print("Неверный ввод.")

        return

    # Шифрование

    Ln, Rn = feistel\_encrypt(L0, R0, key, len(key), wrapped\_f)

    encrypted\_text = f"{Ln} {Rn}"

    write\_file(encrypted\_file, encrypted\_text)

    # Дешифрование

    L0\_dec, R0\_dec = feistel\_decrypt(Ln, Rn, key, len(key), wrapped\_f)

    decrypted\_text = (L0\_dec << (len(text) \* 4)) | R0\_dec

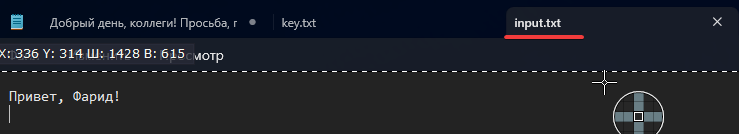
    decrypted\_text = decrypted\_text.to\_bytes((decrypted\_text.bit\_length() + 7) // 8, 'big').decode()

    write\_file(decrypted\_file, decrypted\_text)

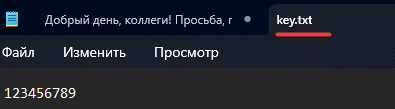
if \_\_name\_\_ == "\_\_main\_\_":

    main()

Текст для шифрования(input):



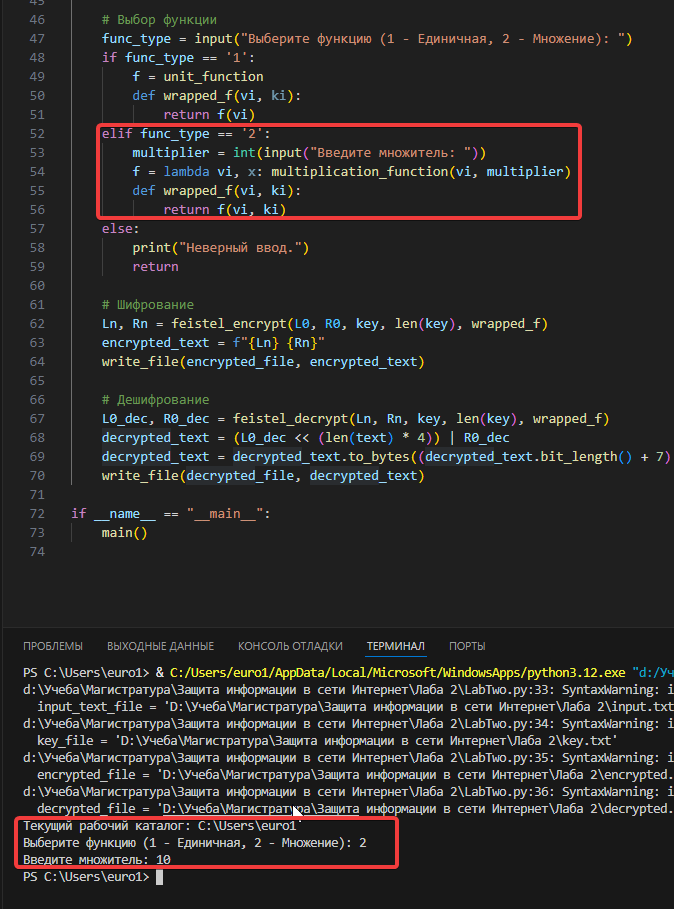
Ключ шифрования(txt):

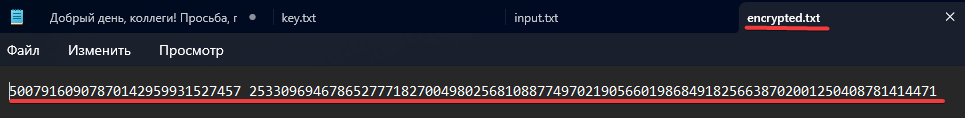


Результат выполнения:

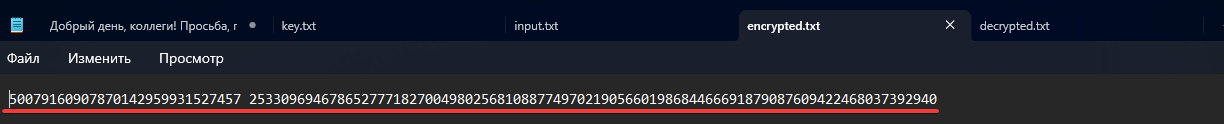
Зашифрованный текст(decrypted):

С помощью функции множителя:





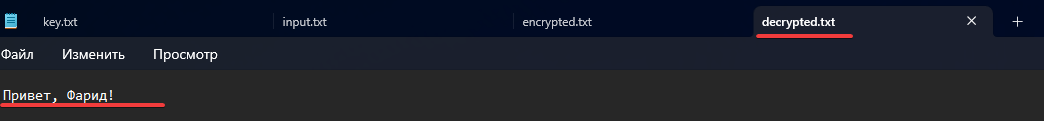
Единичная:



Также срабатывает условие обработчика ошибок, если ввести не тот способ, то упадет ошибка, что является корректным условием:



Расшифрованный текст(encrypted):



Ответы на вопросы:

1. В сети Фейстеля можно использовать различные образующие функции, включая:

- функцию замещения (S-функцию), которая заменяет входные биты на другие биты по заданным правилам.

- функцию расширения, которая изменяет размер входных данных, чтобы они соответствовали размеру ключа.

- линейные и нелинейные функции, такие как сложение по модулю 2, XOR, а также более сложные функции, включающие операции умножения или сложения с ключом.

2. Процесс шифрования в сети Фейстеля происходит следующим образом: входные данные делятся на две равные части, называемые «L0» и «R0», в каждом раунде применяются комбинации функции и ключа (включая операции XOR) к одной из частей, и происходит обмен между ними. Этот процесс повторяется на протяжении определенного количества раундов, в ходе чего результаты обновляются до достижения финальной зашифрованной формы.

3. Процесс дешифрования в сети Фейстеля аналогичен процессу шифрования, но с перестановкой порядка использования ключей. В то время как шифрование принимает «L0» и «R0», дешифрование начинается с зашифрованных «Ln» и «Rn», и ключи обрабатываются в обратном порядке. Это достигается благодаря симметричности алгоритма, что позволяет легко выполнить обратные операции.

4. Величина блока разбиения определяется характеристиками шифруемых данных и алгоритма. Обычно она выбирается так, чтобы обеспечивать баланс между скоростью обработки и степенью безопасности. Стандартные размеры блоков для современных алгоритмов, таких как AES, составляют 128, 192 или 256 бит. Для сети Фейстеля могут быть выбраны другие размеры, например, 64 бита.

5. Свойство функции XOR, которое позволяет производить шифрование и дешифрование, заключается в том, что она является обратимой: A ⊕ B ⊕ B = A. Это означает, что, если мы применим операцию XOR дважды с одним и тем же числом, мы получим исходное значение.

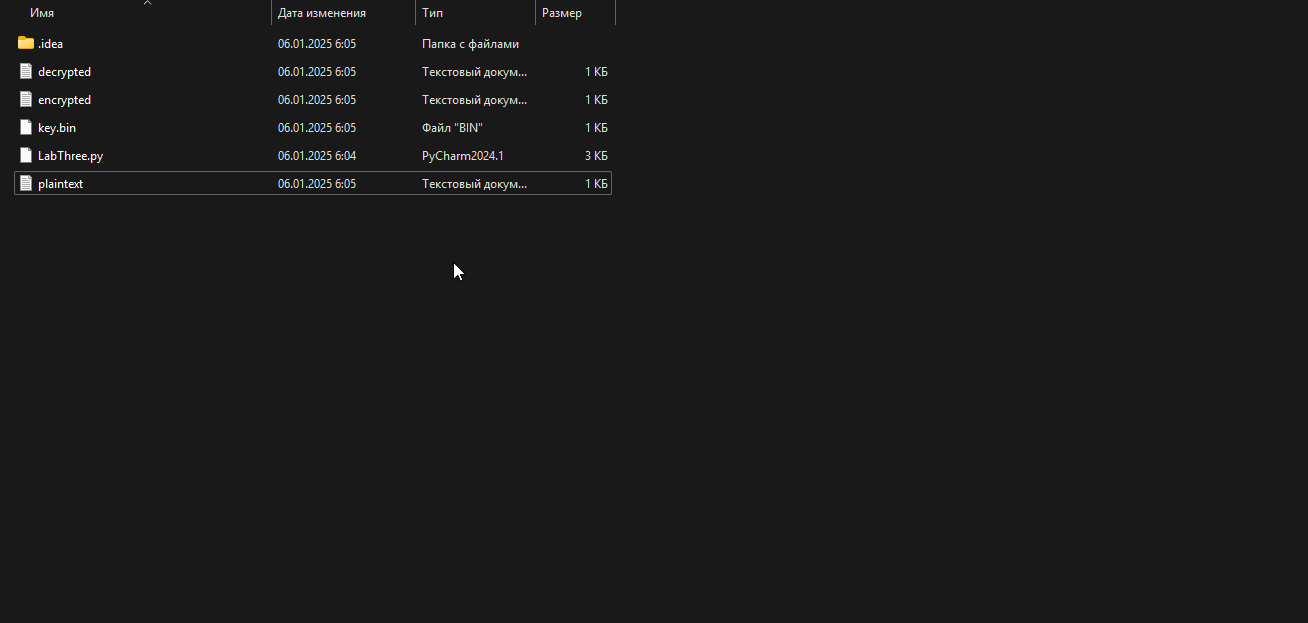
**Лабораторная работа №3. Алгоритм DES.**

Этот алгоритм также был разработан на языке Python, как и в предыдущие разы, и модифицирован так, чтобы файлы не создавались заранее (они будут созданы автоматически после выполнения функционала). Кроме того, зашифрованный текст вводится непосредственно во время работы программы шифрования (текст вводится в консоли). Также для реализации данного функционала была использована библиотека Crypto и способ шифрования DES для этого необходимо было установить с помощью команды: pip install pycryptodome.

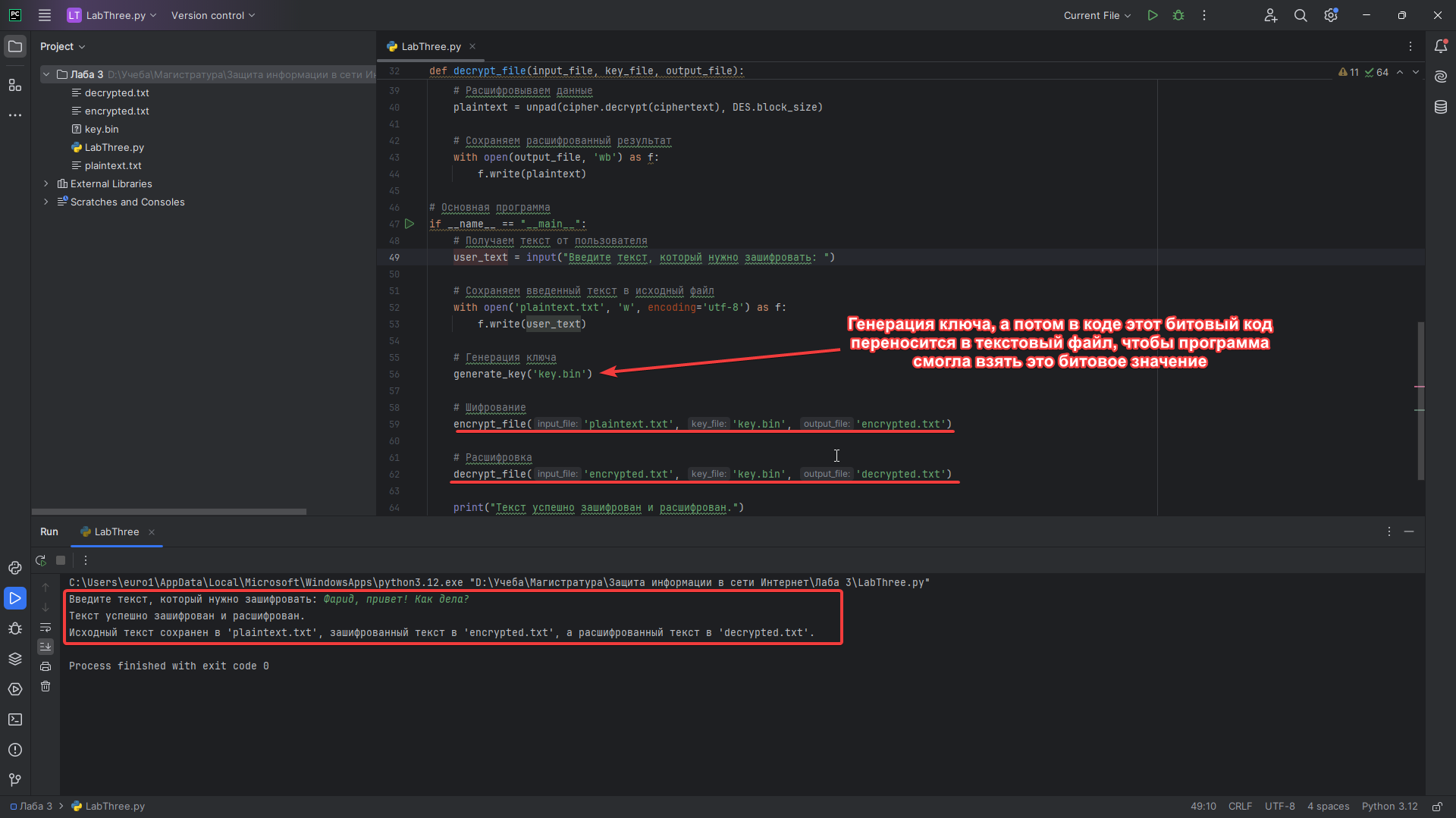
Реализованный алгоритм, код:

from Crypto.Cipher import DES  
from Crypto.Util.Padding import pad, unpad  
import os  
  
# Функция для генерации ключа  
def generate\_key(file\_path):  
 key = os.urandom(8) # Длина ключа DES: 8 байт  
 with open(file\_path, 'wb') as key\_file:  
 key\_file.write(key)  
  
# Загрузка ключа из файла  
def load\_key(file\_path):  
 with open(file\_path, 'rb') as key\_file:  
 return key\_file.read()  
  
# Шифрование файла  
def encrypt\_file(input\_file, key\_file, output\_file):  
 key = load\_key(key\_file)  
 cipher = DES.new(key, DES.MODE\_ECB) # Используем режим ECB  
  
 with open(input\_file, 'rb') as f: # Читаем файл в бинарном режиме  
 plaintext = f.read()  
  
 # Шифруем данные  
 ciphertext = cipher.encrypt(pad(plaintext, DES.block\_size))  
  
 # Сохраняем зашифрованный результат  
 with open(output\_file, 'wb') as f:  
 f.write(ciphertext)  
  
# Расшифровка файла  
def decrypt\_file(input\_file, key\_file, output\_file):  
 key = load\_key(key\_file)  
 cipher = DES.new(key, DES.MODE\_ECB)  
  
 with open(input\_file, 'rb') as f: # Читаем зашифрованные данные в бинарном режиме  
 ciphertext = f.read()  
  
 # Расшифровываем данные  
 plaintext = unpad(cipher.decrypt(ciphertext), DES.block\_size)  
  
 # Сохраняем расшифрованный результат  
 with open(output\_file, 'wb') as f:  
 f.write(plaintext)  
  
# Основная программа  
if \_\_name\_\_ == "\_\_main\_\_":  
 # Получаем текст от пользователя  
 user\_text = input("Введите текст, который нужно зашифровать: ")  
  
 # Сохраняем введенный текст в исходный файл  
 with open('plaintext.txt', 'w', encoding='utf-8') as f:  
 f.write(user\_text)  
  
 # Генерация ключа  
 generate\_key('key.bin')  
  
 # Шифрование  
 encrypt\_file('plaintext.txt', 'key.bin', 'encrypted.txt')  
  
 # Расшифровка  
 decrypt\_file('encrypted.txt', 'key.bin', 'decrypted.txt')  
  
 print("Текст успешно зашифрован и расшифрован.")  
 print("Исходный текст сохранен в 'plaintext.txt', зашифрованный текст в 'encrypted.txt', а расшифрованный текст в 'decrypted.txt'.")

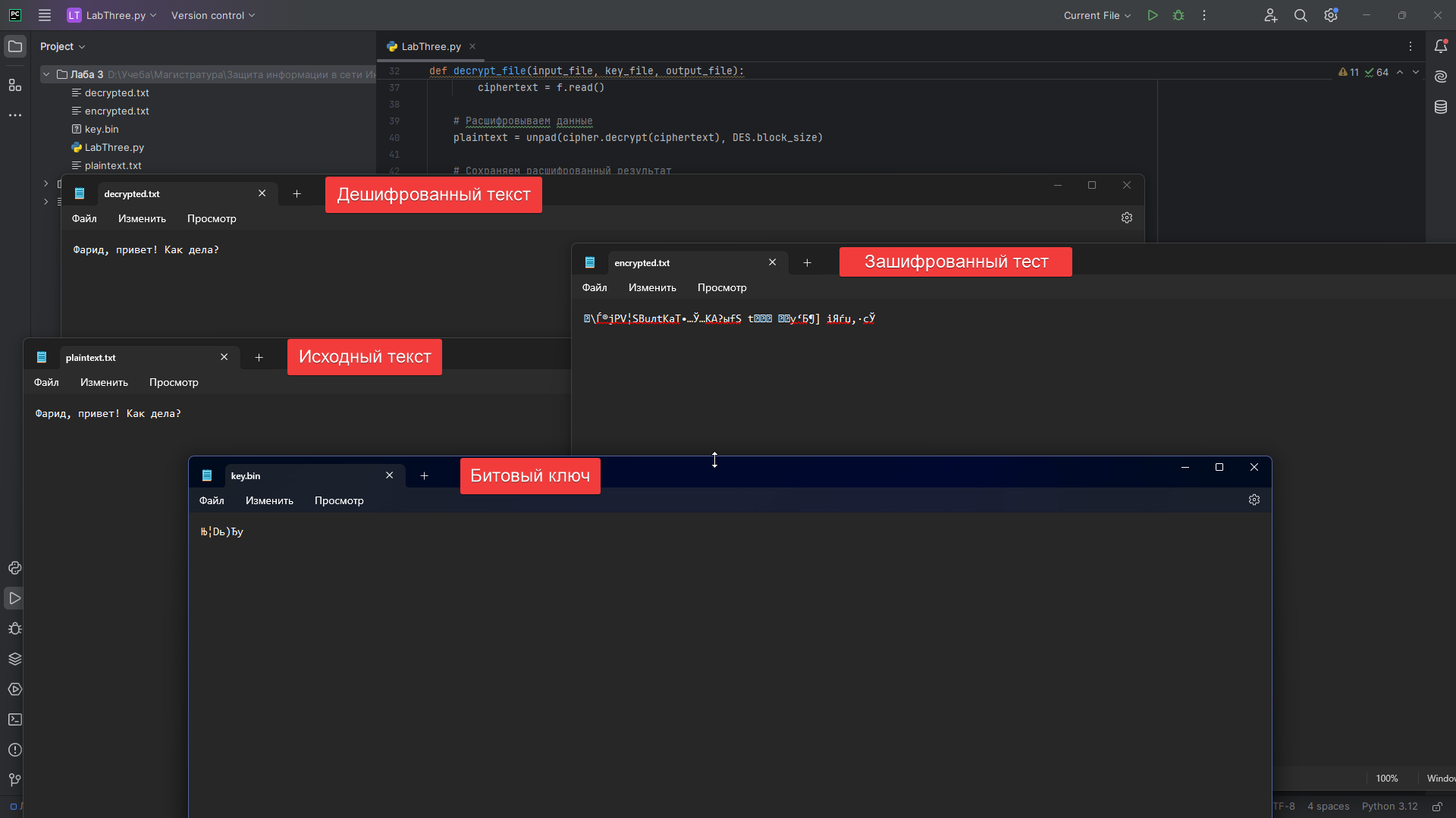
В результате сохраняются файлы в директории программы в зашифрованном и расшифрованном формате:



При запуске программы, пользователь вводит желаемый текст, который нужно зашифровать:



Создается текстовый файл с именем plaintext и потом идет само процесс шифрования и дешифрования:



Ответы на вопросы:

1. Величина блока открытого текста в алгоритме DES составляет 64 бита, а длина ключа — 56 бит.

2. Основные шаги алгоритма DES включают:

- Первичное перестановление (IP).

- Разделение блока на две половины.

- Применение 16 раундов (каждый из которых включает расширение, XOR с ключом, подстановку и перестановку).

- Объединение итоговых половин после всех раундов.

- Применение обратной перестановки (IP-1) для получения зашифрованного текста.

3. Процесс дешифрования в DES осуществляется аналогично шифрованию, но ключи используются в обратном порядке. Вместо применения ключа от первого раунда сначала используется ключ от последнего раунда, и так далее.

4. Ключ генерируется на основе первоначального 56-битного (или 64-битного с контрольными битами) ключа с помощью перестановок и выборок, чтобы получить 16 различных 48-битных ключей для каждого раунда шифрования.

5. В процессе шифрования используются стандартизированные таблицы, такие как таблица первоначальной перестановки (IP), таблицы расширения (E), таблицы подстановки (S-boxes) и таблица обратной перестановки (IP-1).

# **Лабораторная работа №4. Алгоритм RC4.**

По похожему принципу из лабораторной работы №3 был реализован шифратор на алгоритме RS4. Библиотека использовалась та же *Crypto*.

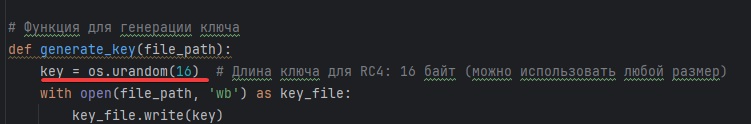
Код:

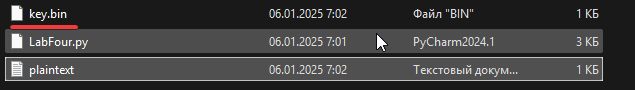
from Crypto.Cipher import ARC4  
import os  
  
# Функция для генерации ключа  
def generate\_key(file\_path):  
 key = os.urandom(16) # Длина ключа для RC4: 16 байт (можно использовать любой размер)  
 with open(file\_path, 'wb') as key\_file:  
 key\_file.write(key)  
  
# Загрузка ключа из файла  
def load\_key(file\_path):  
 with open(file\_path, 'rb') as key\_file:  
 return key\_file.read()  
  
# Шифрование файла  
def encrypt\_file(input\_file, key\_file, output\_file):  
 key = load\_key(key\_file)  
 cipher = ARC4.new(key)  
  
 with open(input\_file, 'rb') as f: # Читаем файл в бинарном режиме  
 plaintext = f.read()  
  
 # Шифруем данные  
 ciphertext = cipher.encrypt(plaintext)  
  
 # Сохраняем зашифрованный результат  
 with open(output\_file, 'wb') as f:  
 f.write(ciphertext)  
  
# Расшифровка файла  
def decrypt\_file(input\_file, key\_file, output\_file):  
 key = load\_key(key\_file)  
 cipher = ARC4.new(key)  
  
 with open(input\_file, 'rb') as f: # Читаем зашифрованные данные в бинарном режиме  
 ciphertext = f.read()  
  
 # Расшифровываем данные  
 plaintext = cipher.decrypt(ciphertext)  
  
 # Сохраняем расшифрованный результат  
 with open(output\_file, 'wb') as f:  
 f.write(plaintext)  
  
# Основная программа  
if \_\_name\_\_ == "\_\_main\_\_":  
 # Получаем текст от пользователя  
 user\_text = input("Введите текст, который нужно зашифровать: ")  
  
 # Сохраняем введенный текст в исходный файл  
 with open('plaintext.txt', 'w', encoding='utf-8') as f:  
 f.write(user\_text)  
  
 # Генерация ключа  
 generate\_key('key.bin')  
  
 # Шифрование  
 encrypt\_file('plaintext.txt', 'key.bin', 'encrypted.txt')  
  
 # Расшифровка  
 decrypt\_file('encrypted.txt', 'key.bin', 'decrypted.txt')  
  
 print("Текст успешно зашифрован и расшифрован.")  
 print("Исходный текст сохранен в 'plaintext.txt', зашифрованный текст в 'encrypted.txt', а расшифрованный текст в 'decrypted.txt'.")

После выполнения данной программы сохраняются также файлы в каталоге:

- plaintext (исходный текст);

- key (рандомный битовый 16-ти разрядный ключ);

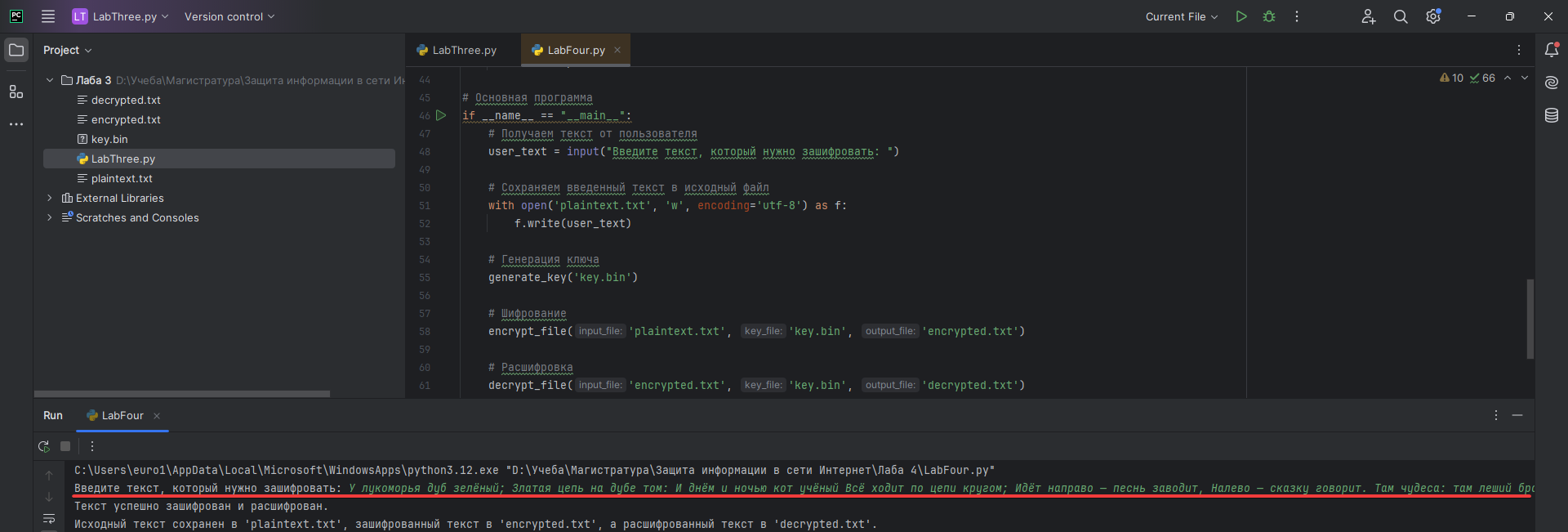




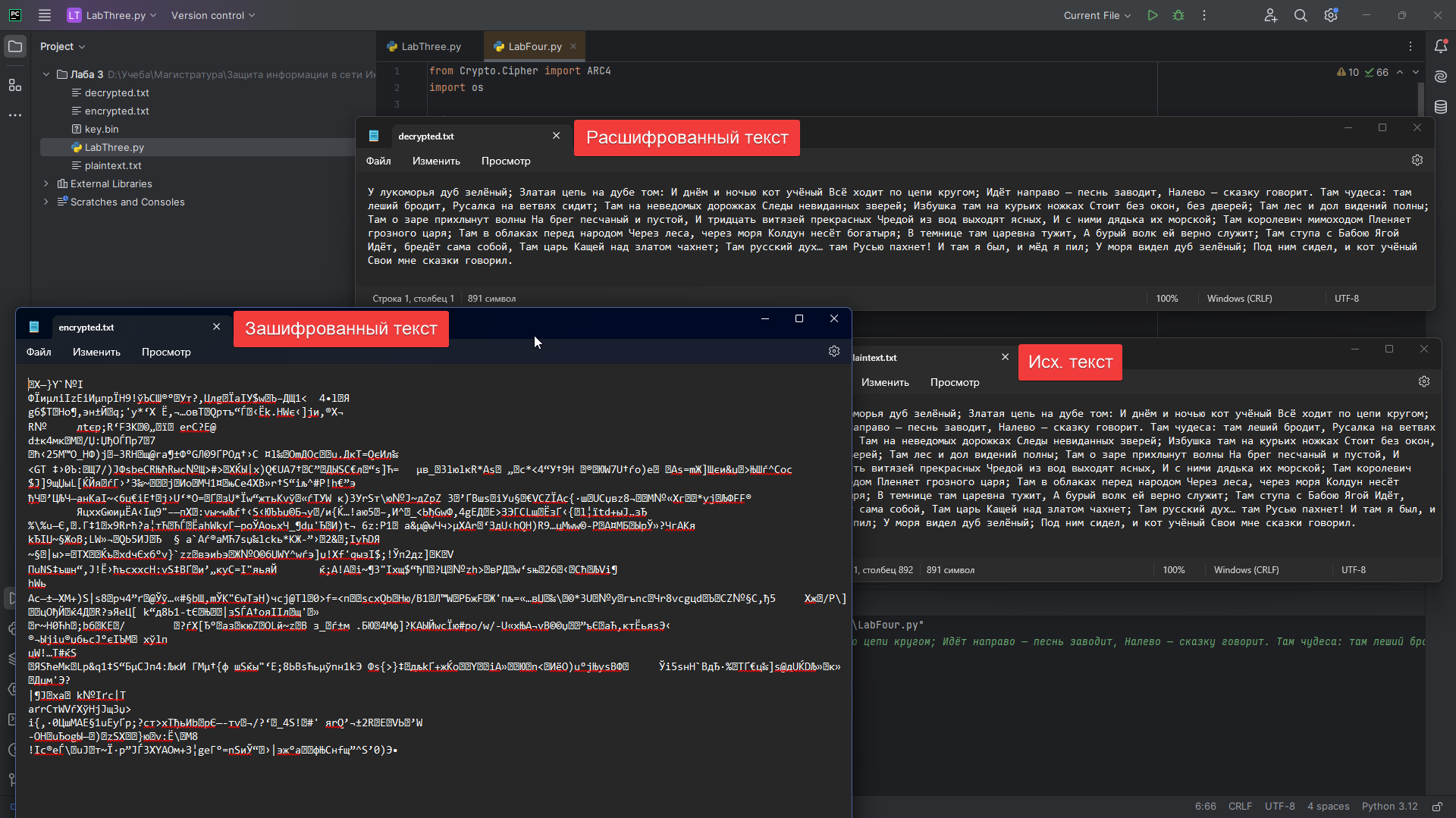
- decrypted (зашифрованный текст);

- encrypted (расшифрованный текст).

Во время запуска передается наш исходный текст, который необходимо зашифровать:



После передачи, текст шифруется алгоритмом RS4 можно убедится, открыв файл с зашифрованным текстом:



Ответы на вопросы:

1. Алгоритм шифрования RC4 состоит из следующих шагов:

1) Инициализация: создаётся массив состояний S, который инициализируется значениями от 0 до 255. Затем происходит перестановка массивов в зависимости от ключа, что обеспечивает неразрывную связь между ключом и состоянием S.

2) Генерация потока ключа: на основе текущего состояния S генерируются псевдослучайные байты, которые составляют поток ключа (гамму). Это достигается с помощью двунаправленного процесса, который перемешивает массив S и генерирует байты с использованием индексов из этого массива.

3) Шифрование (и дешифрование): Шифрование и дешифрование выполняется путем побитового XOR между входными данными и сгенерированным потоком ключа.

2. Дешифрование по алгоритму RC4 происходит идентично шифрованию. Поскольку RC4 является симметричным шифром, для дешифрования используется тот же процесс: входные данные XOR-ятся с тем же потоком ключа, который использовался для шифрования. Результат будет равен первоначальным данным.

3. Гамма в контексте RC4 представляет собой поток псевдослучайных байтов, который генерируется на основе ключа и состояния массива. Гамма используется в шифровании и дешифровании путем побитового XOR с открытым текстом или шифротекстом.

4. Потоковый шифр, такой как RC4, обладает следующими свойствами:

1) Быстрота работы: потоковые шифры обычно быстрее, чем блочные шифры, поскольку они обрабатывают данные по одному байту.

2) Симметричность: один и тот же алгоритм и ключ используются для шифрования и дешифрования.

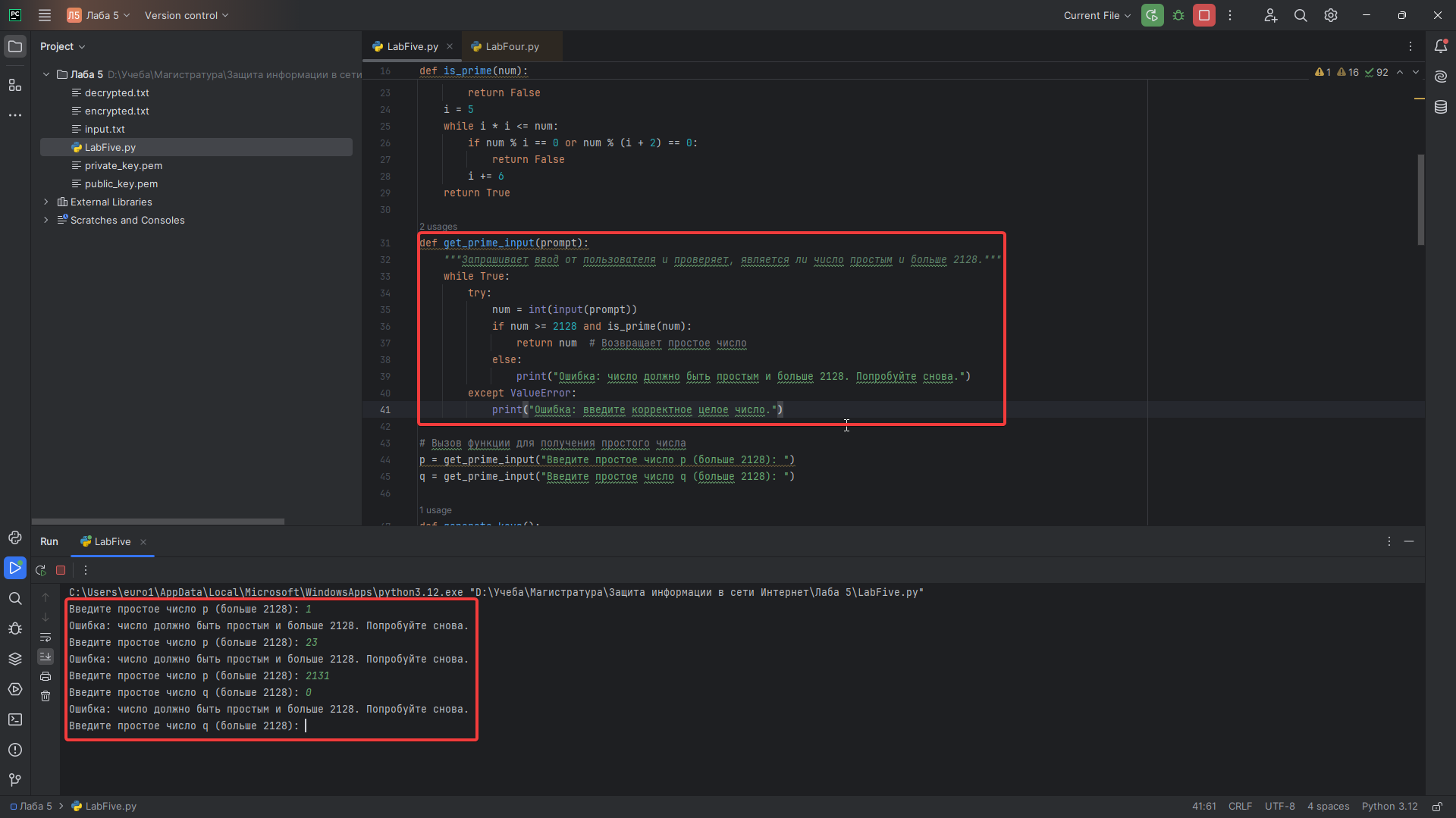
3) Простота реализации: потоковые шифры легко сочетаются с потоковыми протоколами, такими как TCP, и могут обрабатывать данные в реальном времени.

4) Независимость от размера блока: потоковые шифры могут работать с произвольными длинами данных, что делает их универсальными.

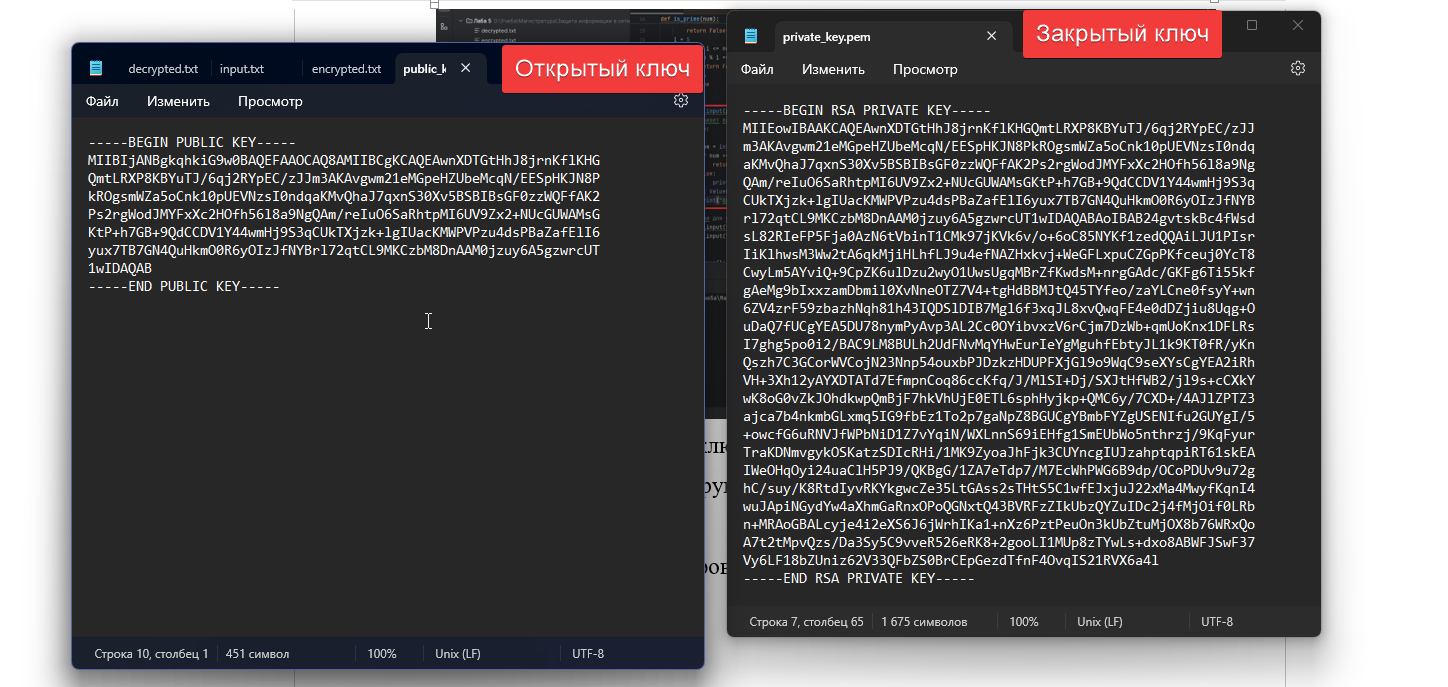
# **Лабораторная работа №5. Ассиметричные алгоритмы шифрования данных: алгоритм RSA.**

Была реализована программа, где: вычисляется открытый и закрытый ключ для алгоритма RSA:

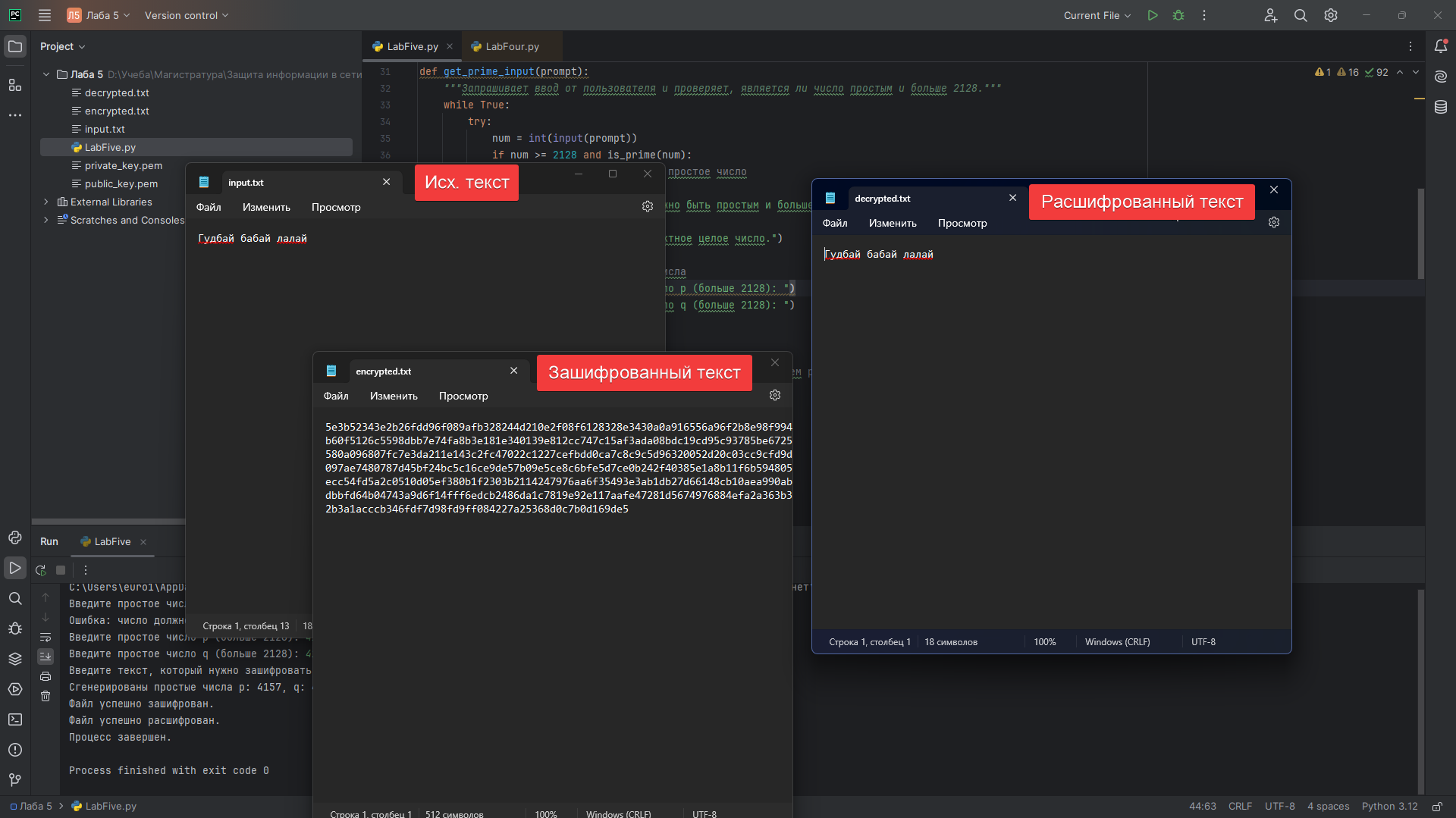
- числа p и q генерируются пользователем, была реализована валидация на введения значения числа p и q, числа должны быть больше значения 2128 и простыми, если числа не являются таковыми, то будет падать обработанная ошибка;



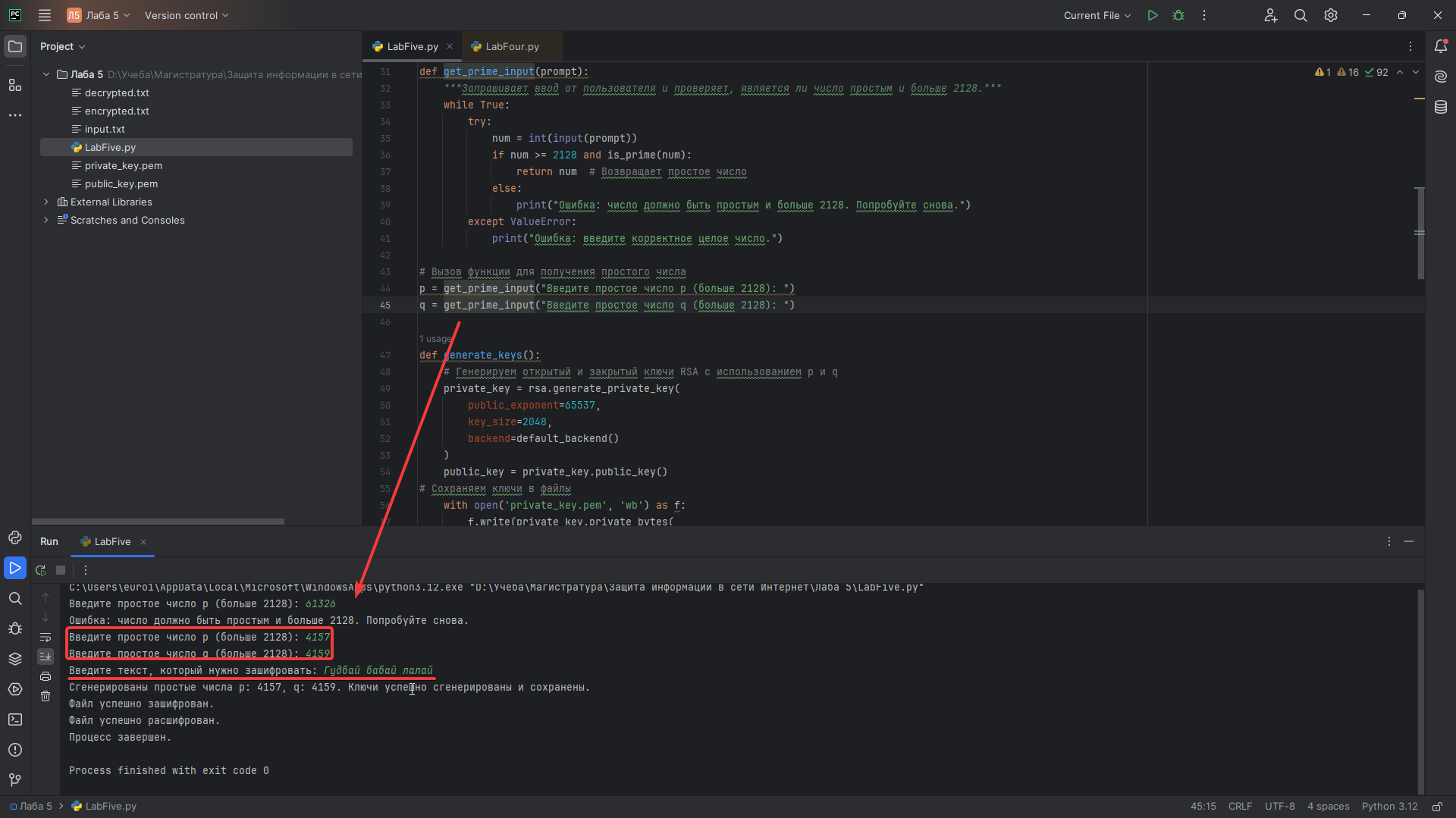
- сгенерированные ключи сохраняются в файлы: открытый ключ – в один файл, закрытый – в другой;



- исходный и зашифрованный тексты хранятся в файлах.



Успешное выполнение программы:



Ответы на вопросы:

1. Алгоритм с открытым ключом — это криптографический алгоритм, использующий пару ключей: открытый ключ, который может быть известен всем, и закрытый ключ, который должен храниться в секрете. Открытый ключ служит для шифрования данных или проверки подписи, в то время как закрытый ключ используется для дешифрования данных или создания подписи.

2. Алгоритм RSA содержит три основных этапа: генерация ключей, шифрование и дешифрование.

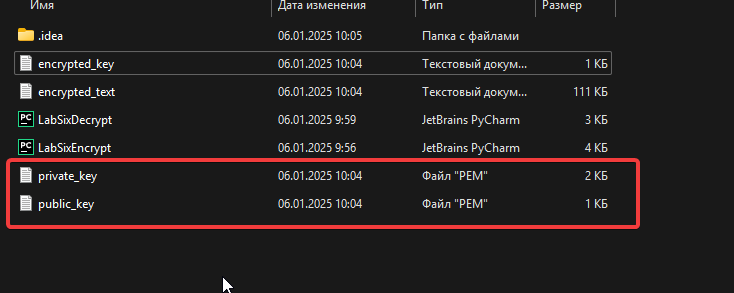
3. Вычисление ключей алгоритма RSA заключается в создании пары ключей — открытого и закрытого. Для этого выбираются два больших простых числа p и q.

4. Шифрование в алгоритме RSA происходит следующим образом: открытый ключ (n, e) используется для преобразования открытого текста M в зашифрованный текст С по формуле C \equiv M^e \mod n.

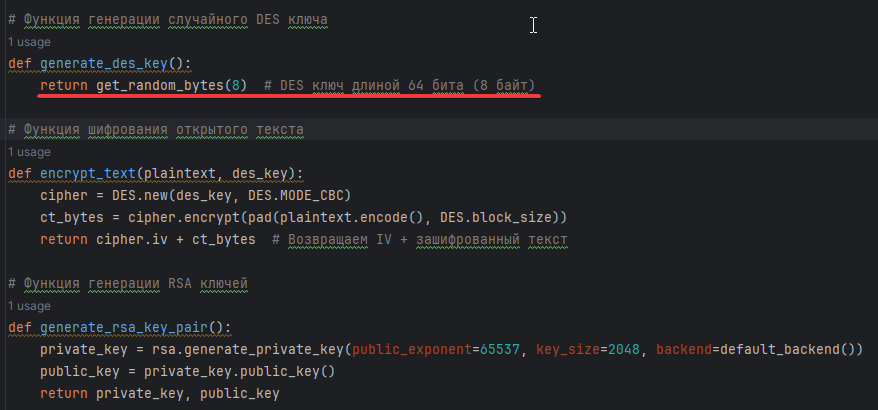
# **Лабораторная работа №6. Комбинирование симметричных и асимметричных алгоритмов.**

Программа реализована, так чтобы шифратор работал по принципу симметричного алгоритма DES открытый текст, а асимметричным RSA – ключ симметричного алгоритма;

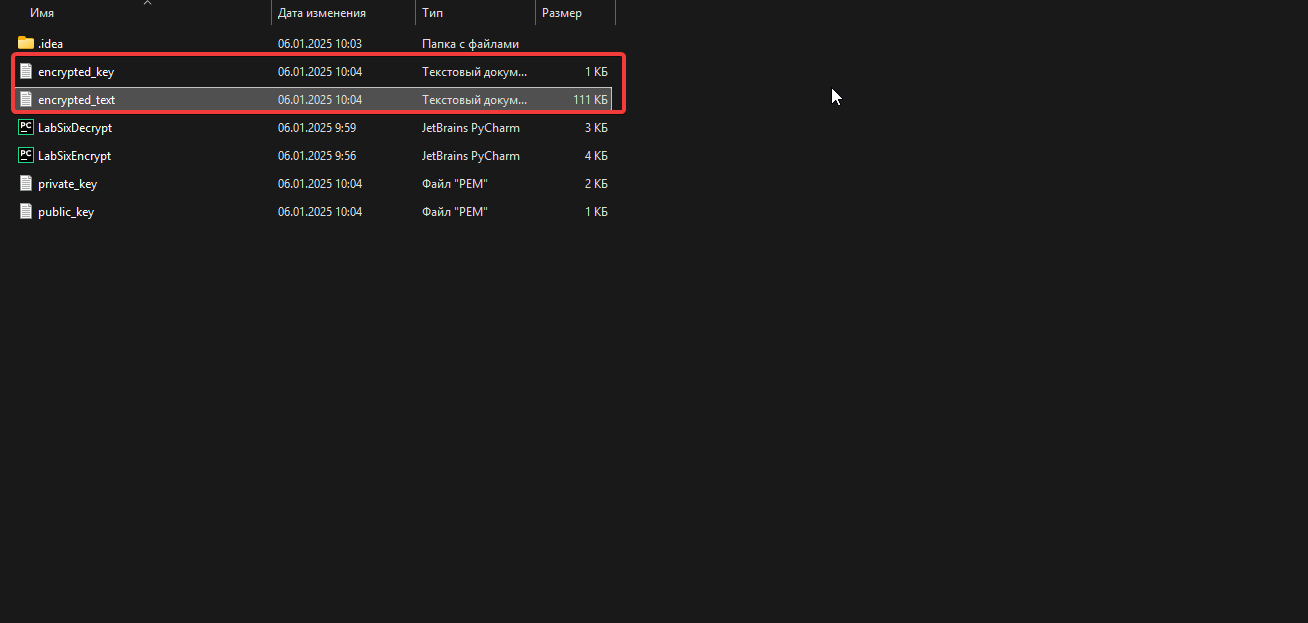
- шифруемый текст храниться в одном файле, открытый ключ для алгоритма RSA – в другом (текст, который будет зашифровываться вводится с клавиатуры и этот файл сохраняется в каталоге);



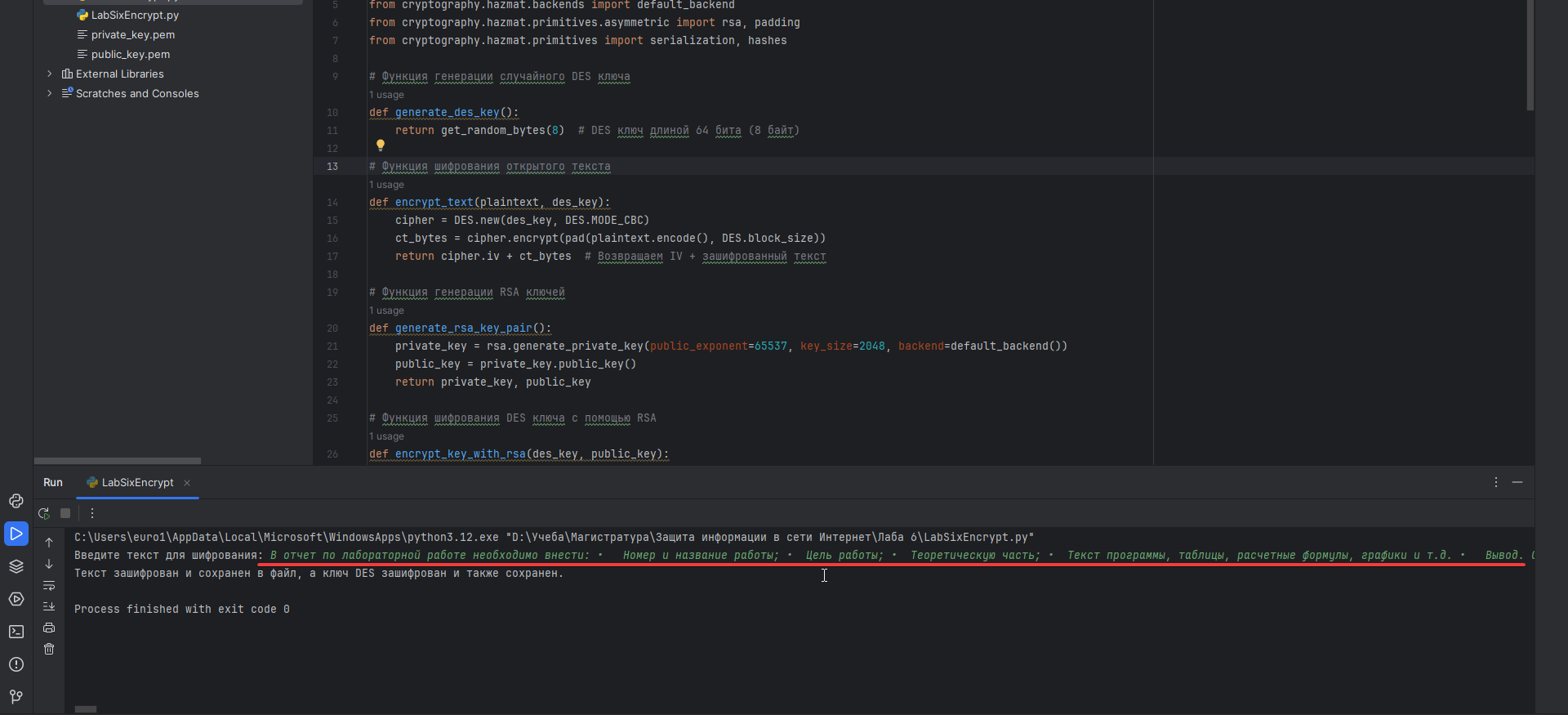
- ключ для симметричного алгоритма DES генериться случайным образом;



- зашифрованный текст храниться в одном файле, а зашифрованный асимметричным алгоритмом ключ симметричного алгоритма – в другом;



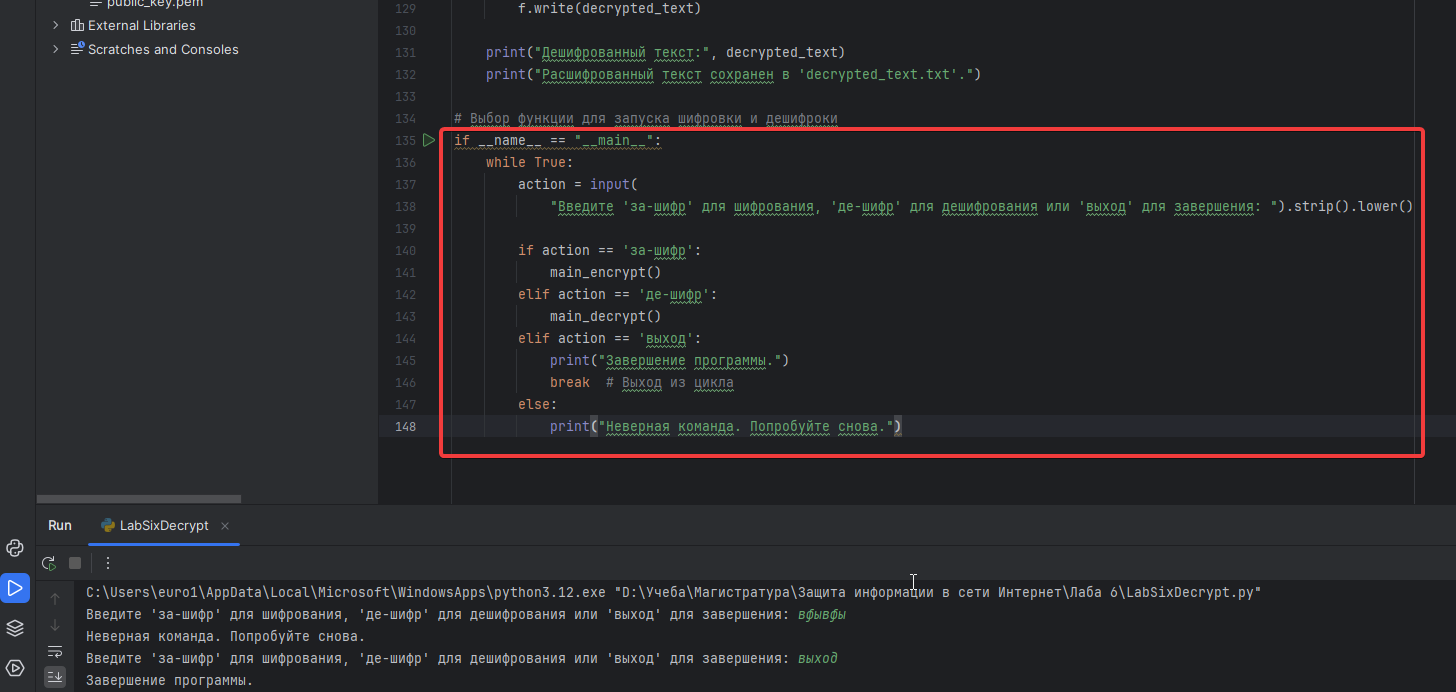
- программа работает с текстом произвольной длины.



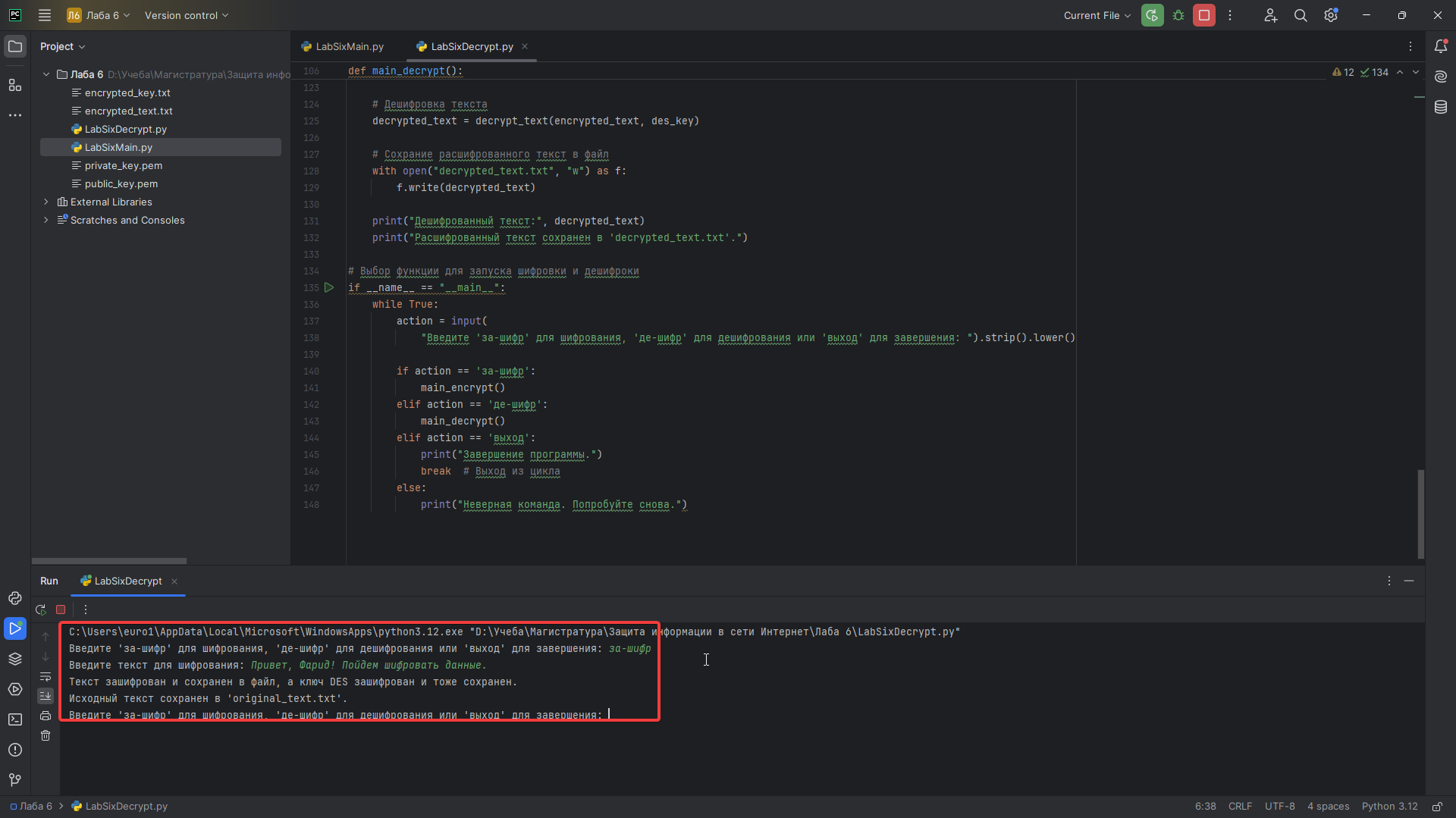
2. Реализовать приложение для дешифрования.

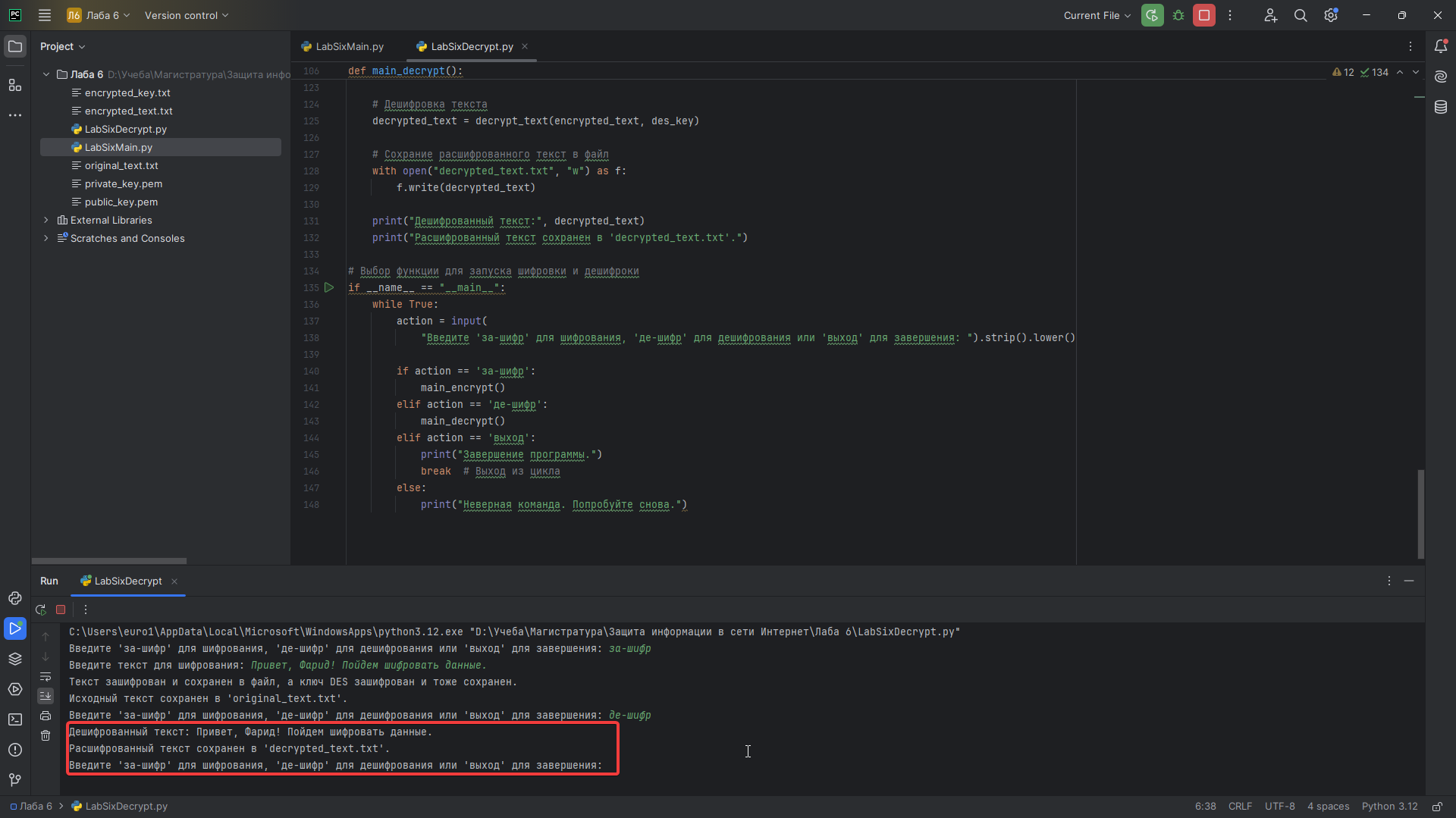
Зашифрованный текст должен храниться в одном файле, зашифрованный ключ симметричного алгоритма – в другом, а секретный ключ для алгоритма RSA – в третьем.

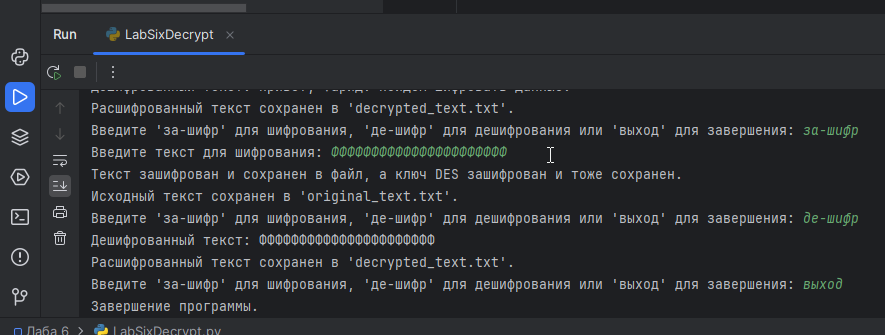
Ранее это было реализовано в двух отдельных файла зашифрования и дешифрования, но потом я упростил до одного и функцией вызова двух этих функций с помощью команд «за\_шифр» и «де\_шифр». И добавлена итерация пока не выполнится одно из условий:

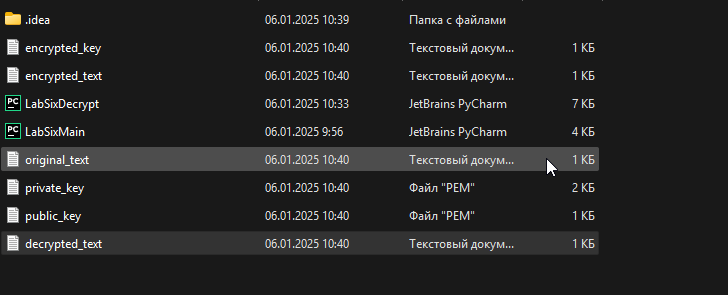


Функции зашифровки и расшифровки выполняются корректно:









Весь код:

import base64  
from Crypto.Cipher import DES  
from Crypto.Random import get\_random\_bytes  
from Crypto.Util.Padding import pad, unpad  
from cryptography.hazmat.backends import default\_backend  
from cryptography.hazmat.primitives.asymmetric import rsa, padding  
from cryptography.hazmat.primitives import serialization, hashes  
  
# Функция генерации случайного DES ключа  
def generate\_des\_key():  
 return get\_random\_bytes(8) # DES ключ длиной 64 бита (8 байт)  
  
# Функция шифрования открытого текста  
def encrypt\_text(plaintext, des\_key):  
 cipher = DES.new(des\_key, DES.MODE\_CBC)  
 ct\_bytes = cipher.encrypt(pad(plaintext.encode(), DES.block\_size))  
 return cipher.iv + ct\_bytes # Возвращаем IV + зашифрованный текст  
  
# Функция генерации RSA ключей  
def generate\_rsa\_key\_pair():  
 private\_key = rsa.generate\_private\_key(public\_exponent=65537, key\_size=2048, backend=default\_backend())  
 public\_key = private\_key.public\_key()  
 return private\_key, public\_key  
  
# Функция шифрования DES ключа с помощью RSA  
def encrypt\_key\_with\_rsa(des\_key, public\_key):  
 ciphertext = public\_key.encrypt(  
 des\_key,  
 padding.OAEP(  
 mgf=padding.MGF1(algorithm=hashes.SHA256()),  
 algorithm=hashes.SHA256(),  
 label=None  
 )  
 )  
 return ciphertext  
  
# Основная функция для шифрования  
def main\_encrypt():  
 # Получаем текст для шифрования  
 plaintext = input("Введите текст для шифрования: ")  
  
 # Генерируем ключ для DES  
 des\_key = generate\_des\_key()  
  
 # Шифруем текст  
 encrypted\_text = encrypt\_text(plaintext, des\_key)  
  
 # Генерируем пару ключей RSA  
 private\_key, public\_key = generate\_rsa\_key\_pair()  
  
 # Шифруем DES ключ с помощью RSA  
 encrypted\_des\_key = encrypt\_key\_with\_rsa(des\_key, public\_key)  
  
 # Кодируем зашифрованные данные в формате Base64 для текстового представления  
 encrypted\_text\_b64 = base64.b64encode(encrypted\_text)  
 encrypted\_des\_key\_b64 = base64.b64encode(encrypted\_des\_key)  
  
 # Сохранение зашифрованного текста и ключи в текстовые файлы  
 with open("encrypted\_text.txt", "wb") as f:  
 f.write(encrypted\_text\_b64)  
  
 with open("encrypted\_key.txt", "wb") as f:  
 f.write(encrypted\_des\_key\_b64)  
  
 # Сохранение открытого ключа RSA  
 with open("public\_key.pem", "wb") as f:  
 f.write(public\_key.public\_bytes(  
 encoding=serialization.Encoding.PEM,  
 format=serialization.PublicFormat.SubjectPublicKeyInfo  
 ))  
  
 # Сохранение закрытого ключа RSA в файле  
 with open("private\_key.pem", "wb") as f:  
 f.write(private\_key.private\_bytes(  
 encoding=serialization.Encoding.PEM,  
 format=serialization.PrivateFormat.TraditionalOpenSSL,  
 encryption\_algorithm=serialization.NoEncryption()  
 ))  
  
 # Сохранение в исходный текст в файл  
 with open("original\_text.txt", "w") as f:  
 f.write(plaintext)  
  
 print("Текст зашифрован и сохранен в файл, а ключ DES зашифрован и тоже сохранен.")  
 print("Исходный текст сохранен в 'original\_text.txt'.")  
  
# Функция дешифрования DES ключа  
def decrypt\_key\_with\_rsa(encrypted\_key, private\_key):  
 return private\_key.decrypt(  
 encrypted\_key,  
 padding.OAEP(  
 mgf=padding.MGF1(algorithm=hashes.SHA256()),  
 algorithm=hashes.SHA256(),  
 label=None  
 )  
 )  
  
# Функция дешифрования текста  
def decrypt\_text(encrypted\_text, des\_key):  
 iv = encrypted\_text[:8] # Первые 8 байт - это IV  
 cipher = DES.new(des\_key, DES.MODE\_CBC, iv)  
 decrypted\_padded\_text = cipher.decrypt(encrypted\_text[8:]) # Убираем IV  
 return unpad(decrypted\_padded\_text, DES.block\_size).decode()  
  
# Основная функция для дешифрования  
def main\_decrypt():  
 # Загружаем закрытый ключ RSA  
 with open("private\_key.pem", "rb") as f:  
 private\_key = serialization.load\_pem\_private\_key(f.read(), password=None)  
  
 # Прочтение зашифрованного ключа DES и декодирока из Base64  
 with open("encrypted\_key.txt", "rb") as f:  
 encrypted\_des\_key\_b64 = f.read()  
 encrypted\_des\_key = base64.b64decode(encrypted\_des\_key\_b64)  
  
 # Дешифровка ключа DES  
 des\_key = decrypt\_key\_with\_rsa(encrypted\_des\_key, private\_key)  
  
 # Прочтение зашифрованного текста и декодирока из Base64  
 with open("encrypted\_text.txt", "rb") as f:  
 encrypted\_text\_b64 = f.read()  
 encrypted\_text = base64.b64decode(encrypted\_text\_b64)  
  
 # Дешифровка текста  
 decrypted\_text = decrypt\_text(encrypted\_text, des\_key)  
  
 # Сохрание расшифрованного текст в файл  
 with open("decrypted\_text.txt", "w") as f:  
 f.write(decrypted\_text)  
  
 print("Дешифрованный текст:", decrypted\_text)  
 print("Расшифрованный текст сохранен в 'decrypted\_text.txt'.")  
  
# Выбор функции для запуска шифровки и дешифроки  
if \_\_name\_\_ == "\_\_main\_\_":  
 while True:  
 action = input(  
 "Введите 'за-шифр' для шифрования, 'де-шифр' для дешифрования или 'выход' для завершения: ").strip().lower()  
  
 if action == 'за-шифр':  
 main\_encrypt()  
 elif action == 'де-шифр':  
 main\_decrypt()  
 elif action == 'выход':  
 print("Завершение программы.")  
 break # Выход из цикла  
 else:  
 print("Неверная команда. Попробуйте снова.")

Ответы на вопросы:

1. Для чего и почему используют комбинированные криптоалгоритмы.

Комбинированные криптоалгоритмы используют для достижения баланса между безопасностью, эффективностью и удобством работы с криптографией. Эти алгоритмы объединяют сильные стороны как симметричных, так и асимметричных методов шифрования, обеспечивая надежную защиту данных. Комбинирование ключевых подходов позволяет использовать асимметричные алгоритмы для обмена ключами, а затем применять симметричные алгоритмы для шифрования самого сообщения. Это важно, поскольку:

1) Асимметричные алгоритмы медленнее, но обеспечивают безопасный обмен ключами.

2) Симметричные алгоритмы более быстры и эффективны для больших объемов данных.

2. В чём заключаются достоинства и недостатки асимметричных алгоритмов.

Достоинства:

1) Безопасный обмен ключами. Асимметричные алгоритмы позволяют безопасно обмениваться ключами без необходимости в предварительно установленном секретном ключе.

2) Цифровая подпись. Они обеспечивают возможность подписывать данные своими приватными ключами, что подтверждает подлинность и целостность сообщений.

Недостатки:

1) Низкая скорость. В сравнении с симметричными алгоритмами асимметричные алгоритмы обычно гораздо медленнее, что делает их менее подходящими для шифрования больших объемов данных.

2) Большие ключи. Для обеспечения необходимого уровня безопасности асимметричные алгоритмы требуют использования более длинных ключей.

3. В чём заключаются достоинства и недостатки симметричных алгоритмов.

Достоинства:

1) Высокая скорость. Симметричные алгоритмы быстрее, чем асимметричные, что делает их подходящими для шифрования больших объемов данных.

2) Простота реализации. Эти алгоритмы, как правило, проще в реализации и требуют меньшего объема вычислительных ресурсов.

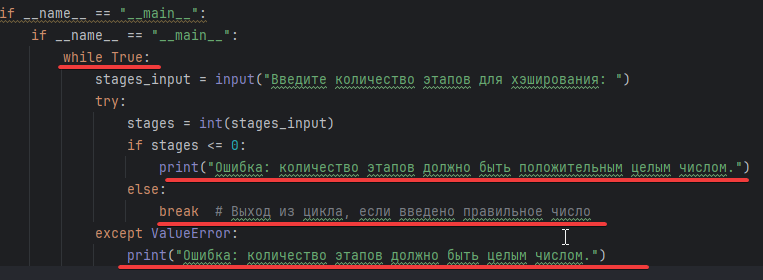
Недостатки:

1) Проблема распределения ключей. Чтобы два субъекта могли безопасно общаться, им необходимо каким-либо образом обменяться секретным ключом, что может быть проблематично.

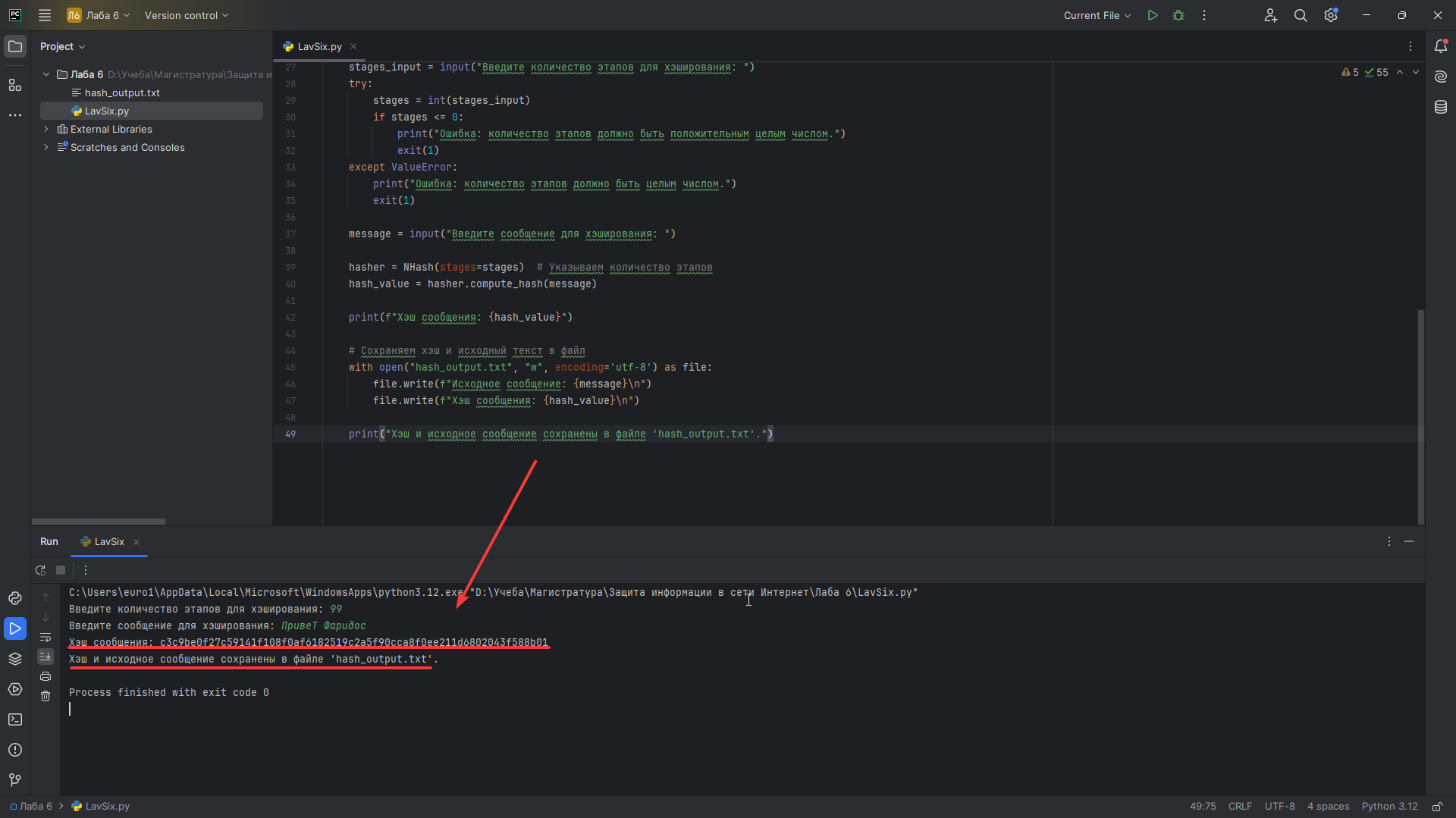
2) Угрозы безопасности. Если секретный ключ будет скомпрометирован, это приведет к потере конфиденциальности всех данных, зашифрованных с его использованием.

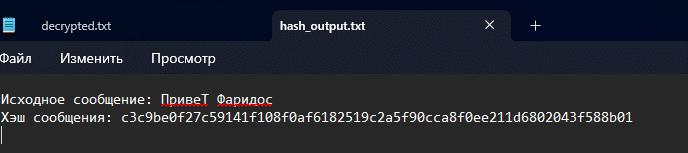
# **Лабораторная работа №7. Алгоритм N-хэш.**

1. Было реализована возможность шифрования несколько этапов (ввод этапов происходит с клавиатуры) и добавлена валидация на отрицательное и целое число.



2. Хэш сообщение и исходный сохраняется отдельно в файл (сделал в один, в падлу реализовывать в несколько, т.к. нужно несколько раз скринить 😊), а также выводится сообщение после выполнения программы и как называется файл.





Код:

import hashlib  
  
class NHash:  
 def \_\_init\_\_(self, stages=1):  
 *"""Инициализация N-hash с указанным количеством этапов."""* self.stages = stages  
  
 def compute\_hash(self, input\_data):  
 *"""Вычисляет хэш сообщения."""* result = input\_data  
  
 # Применяем хэширование на заданное количество этапов  
 for \_ in range(self.stages):  
 result = self.\_intermediate\_hash(result)  
  
 return result  
  
 def \_intermediate\_hash(self, text):  
 *"""Скрытый метод для промежуточного хэширования."""* # Используем SHA-256 как основной алгоритм  
 text\_bytes = text.encode('utf-8')  
 hash\_object = hashlib.sha256()  
 hash\_object.update(text\_bytes)  
 return hash\_object.hexdigest()  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 if \_\_name\_\_ == "\_\_main\_\_":  
 while True:  
 stages\_input = input("Введите количество этапов для хэширования: ")  
 try:  
 stages = int(stages\_input)  
 if stages <= 0:  
 print("Ошибка: количество этапов должно быть положительным целым числом.")  
 else:  
 break # Выход из цикла, если введено правильное число  
 except ValueError:  
 print("Ошибка: количество этапов должно быть целым числом.")  
  
 message = input("Введите сообщение для хэширования: ")  
  
 hasher = NHash(stages=stages) # Указываем количество этапов  
 hash\_value = hasher.compute\_hash(message)  
  
 print(f"Хэш сообщения: {hash\_value}")  
  
 # Сохраняем хэш и исходный текст в файл  
 with open("hash\_output.txt", "w", encoding='utf-8') as file:  
 file.write(f"Исходное сообщение: {message}\n")  
 file.write(f"Хэш сообщения: {hash\_value}\n")  
  
 print("Хэш и исходное сообщение сохранены в файле 'hash\_output.txt'.")

Ответы на вопросы:

1. Что из себя представляет хэш функция. Хэш-функция представляет собой алгоритм, который преобразует входные данные произвольного размера в фиксированный размер, обычно представляющий собой последовательность битов. Хэш-функции обладают рядом свойств: они являются односторонними, что означает, что обратное преобразование невозможно, и для разных входных данных они должны выдавать различные выходные данные (коллизии должно быть минимальное количество). Хэш-функции широко применяются в криптографии, компьютерной безопасности, а также для проверки целостности данных.

2. Пояснить один цикл работы алгоритма N-Hash. Один цикл работы алгоритма N-Hash включает в себя применение хэш-функции к текущему значению (обычно исходному сообщению или предыдущему хэшу) с целью получения нового хэша. Внутри этого цикла сообщение обрабатывается с использованием предварительно определенного алгоритма хэширования, например SHA-256. После выполнения хэширования полученный хэш становится входными данными для следующего цикла, если количество раундов превышает один. Процесс повторяется, пока не будет достигнуто необходимое количество раундов.

3. Как формируется образующая функция для алгоритма N-Hash. Образующая функция для алгоритма N-Hash формируется путем комбинирования нескольких характеристик, таких как выбор основных хэш-функций, которые будут использоваться на каждом этапе, а также структура и порядок обработки входных данных. Образующая функция может включать в себя дополнительные блоки криптографических операций, которые увеличивают стойкость к атакам и обеспечивают целостность данных на каждом этапе.

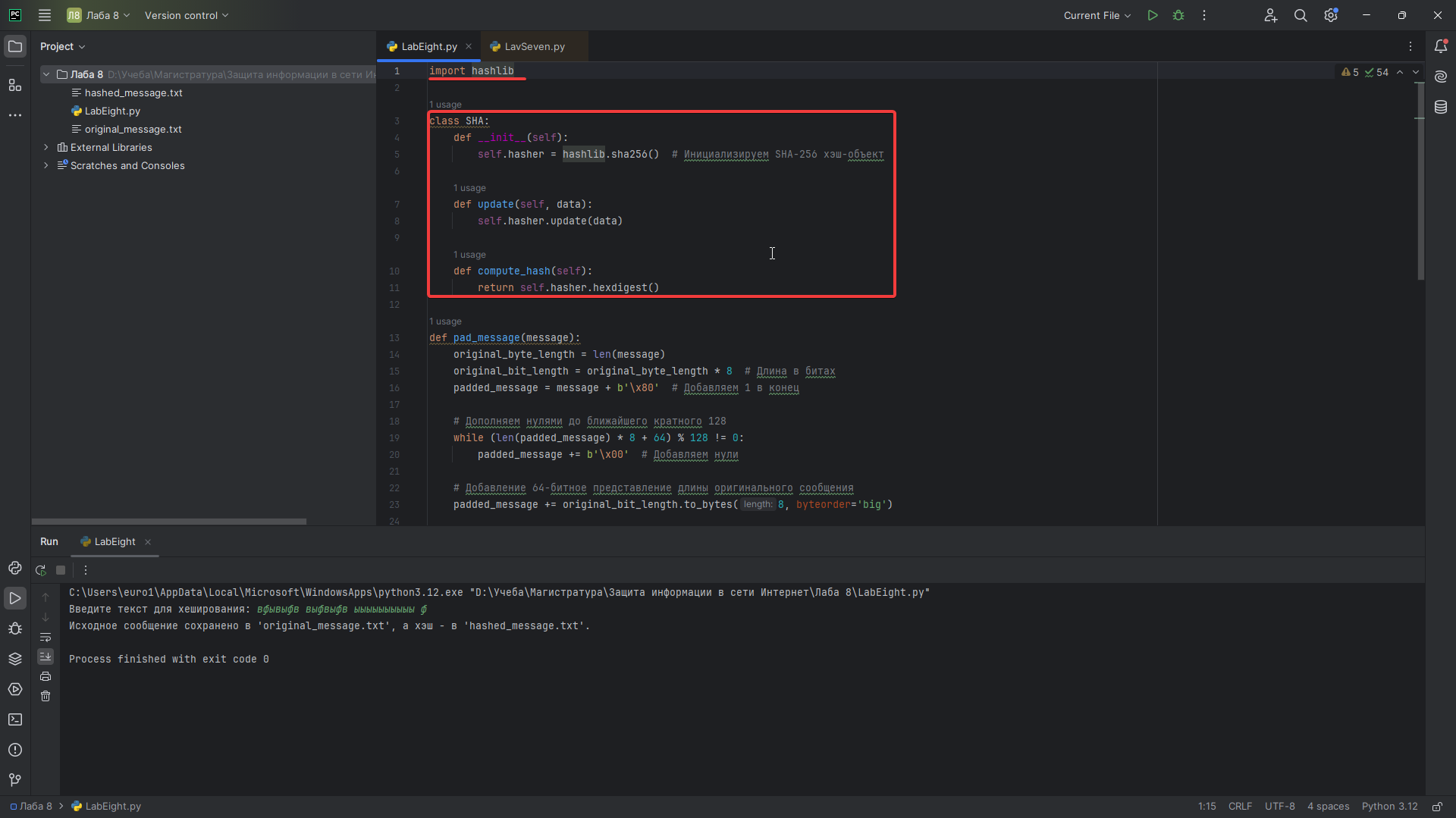
4. Какое минимальное количество раундов рекомендуется использовать в алгоритме N-Hash. Минимальное количество раундов, которое рекомендуется использовать в алгоритме N-Hash, обычно составляет несколько (например, 3 или 4), но в зависимости от конкретной реализации и требований безопасности, это значение может варьироваться. Рекомендуется проводить тестирование и оценку стойкости хэш-функции на разных значениях раундов для оптимизации безопасности.

5. На какие блоки делится открытое сообщение во время N-Hash шифрования. Открытое сообщение во время N-Hash шифрования делится на фиксированные блоки определенного размера. Размер этих блоков зависит от выбранного алгоритма хэширования и установленного размера блока (например, 512 или 1024 бита). Обычно на каждом этапе алгоритм работает с одним из таких блоков, вычисляя хэш на его основе и затем сочетая результаты обработки.

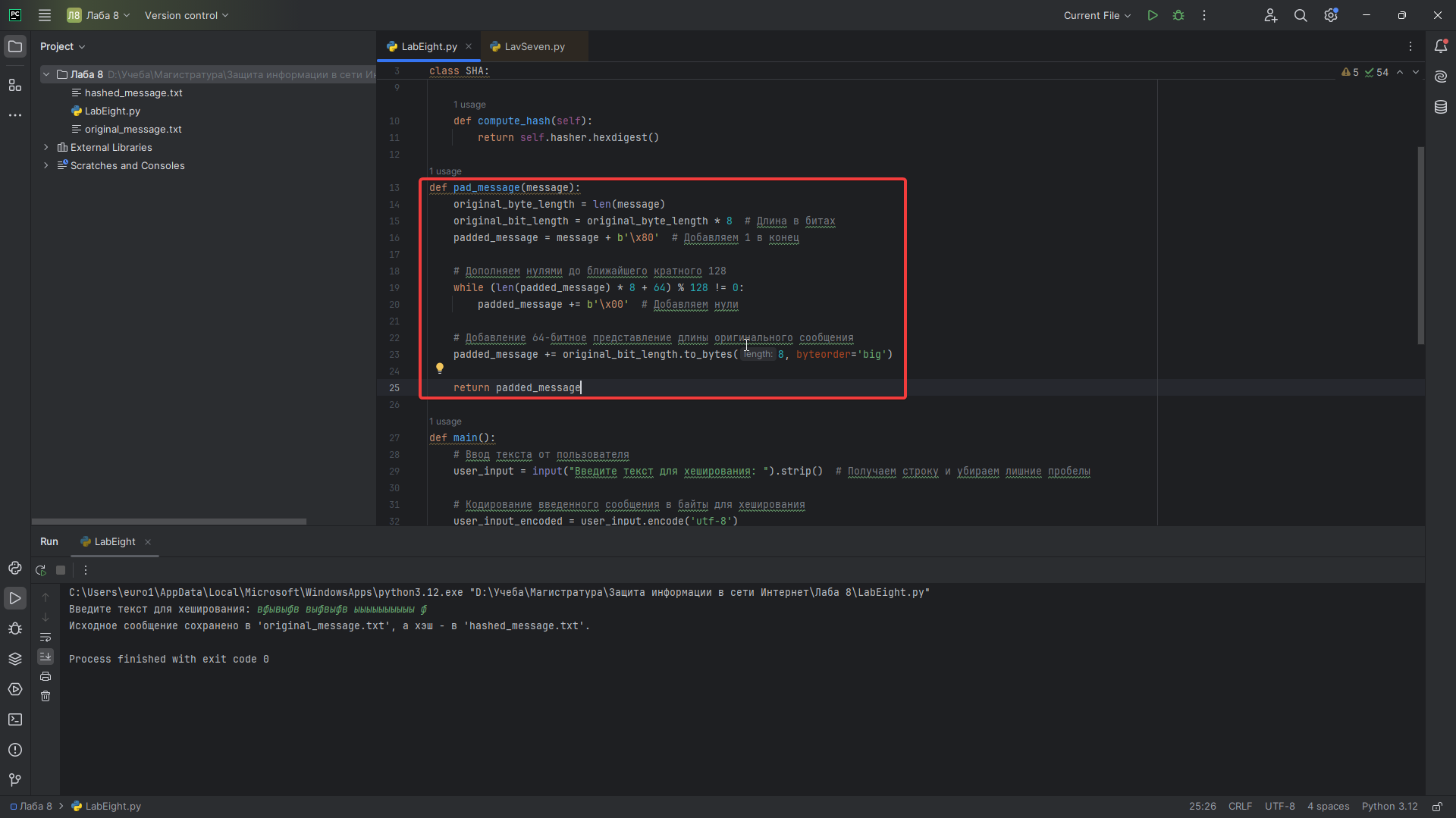
P.S. Была пропущена лабораторная №6 и данная лабораторная работа зафиксирована на скриншотах как шестая, но в дальнейшем исправил на LabSeven и сделана шестая лабораторная).

# **Лабораторная работа №8. Алгоритм SHA.**

Шифратор работает по принципу алгоритма SHA, благодаря импортированию библиотеки *hashlib.*

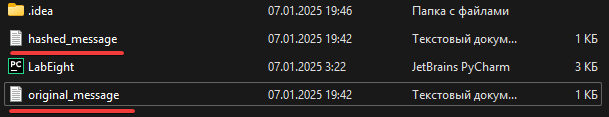


Как указано в примечании к лабораторной, необходимо дополнение единицы и какое-то кол-во нулей, сколько необходимо, если сообщение не кратно 128 битам, логика была реализована, следующим способом:



Ввод текста, который нужно зашифровать выполняется с клавиатуры пользователем – после ввода текста, выходит сообщение о зашифровании текста и сохранении в файлах «original\_message.txt» (исх. сообщение), «hashed\_message.txt» (хеш).

Реализовано отдельно в файлах, т.к. при совмещении ориг. текста и хеша блокнот видит все символы в ANSII (почему не ясно), если открывать в ворде, то все успешно отображалось, но все равно реализовал сохранение отдельно в файлы.



Весь код шифратора:

import hashlib  
  
class SHA:  
 def \_\_init\_\_(self):  
 self.hasher = hashlib.sha256() # Инициализируем SHA-256 хэш-объект  
  
 def update(self, data):  
 self.hasher.update(data)  
  
 def compute\_hash(self):  
 return self.hasher.hexdigest()  
  
def pad\_message(message):  
 original\_byte\_length = len(message)  
 original\_bit\_length = original\_byte\_length \* 8 # Длина в битах  
 padded\_message = message + b'\x80' # Добавляем 1 в конец  
  
 # Дополняем нулями до ближайшего кратного 128  
 while (len(padded\_message) \* 8 + 64) % 128 != 0:  
 padded\_message += b'\x00' # Добавляем нули  
  
 # Добавление 64-битное представление длины оригинального сообщения  
 padded\_message += original\_bit\_length.to\_bytes(8, byteorder='big')  
  
 return padded\_message  
  
def main():  
 # Ввод текста от пользователя  
 user\_input = input("Введите текст для хеширования: ").strip() # Получаем строку и убираем лишние пробелы  
  
 # Кодирование введенного сообщения в байты для хеширования  
 user\_input\_encoded = user\_input.encode('utf-8')  
  
 padded\_message = pad\_message(user\_input\_encoded)  
  
 # Создаем экземпляр класса SHA и вычисляем хэш  
 sha\_instance = SHA()  
 sha\_instance.update(padded\_message)  
 hash\_value = sha\_instance.compute\_hash()  
  
 # Сохраняем исходное сообщение в отдельный файл  
 with open("original\_message.txt", 'w', encoding='utf-8') as file:  
 file.write(user\_input) # Сохраняем исходное сообщение  
  
 # Сохраняем хэш в отдельный файл  
 with open("hashed\_message.txt", 'w', encoding='utf-8') as file:  
 file.write(hash\_value) # Сохраняем хэш отдельно  
  
 print("Исходное сообщение сохранено в 'original\_message.txt', а хэш - в 'hashed\_message.txt'.")  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 main()

1. Размер блока для алгоритма SHA составляет 512 бит (64 байта). Это означает, что данные, входящие в алгоритм, разбиваются на блоки размером 512 бит, если общая длина сообщения превышает 512 бит.

2. Процедура формирования хеш-образа в SHA включает несколько этапов:

Подготовка: исходное сообщение разбивается на блоки по 512 бит. Если длина сообщения не кратна 512, к сообщению добавляются биты для его выравнивания до нужной длины. Применяется добавление '1' бита, за которым следуют нули, а затем добавляется длина исходного сообщения в битах, закодированная в последних 64 битах блока.

Инициализация: устанавливаются начальные значения для пятнадцати постоянных (H0, H1, H2, H3, H4).

Обработка блоков: каждый 512-битный блок обрабатывается с использованием алгоритма, который включает в себя несколько итераций, манипуляций и функций.

Вывод: после обработки всех блоков конечный хеш-образ создается путем конкатенации значений H0 до H4.

3. Схема итерации алгоритма SHA-1 включает три основных этапа обработки каждого 512-битного блока. Соединение: изначально блок разделяется на 16 слов по 32 бита, каждые 32 бита из блока становятся одним словом.

Расширение: эти 16 слов расширяются до 80 слов, используя операции XOR и побитовые сдвиги.

Основной цикл: каждый из 80 итерационных шагов использует одну из четырех различных функций, основанных на значениях текущих слов, и добавляет текущий хеш и константы для получения новых значений.

4. Алгоритм SHA-1 основан на концепции хеширования, использующей свойства криптографической устойчивости, такие как письменная «односторонняя функция». Он применяет последовательность побитовых операций, логических операций, сдвигов и дополнительных констант для создания 160-битного (20 байт) хеш-значения.

5. Образующая функция, применяемая в SHA-1, комбинирует следующие операции: AND, OR, XOR, а также сдвиги (как циклические, так и обычные). Основные функции, используемые на каждом шаге, имеют вид:

F0 = (B AND C) OR (NOT B AND D)

F1 = B XOR C XOR D

F2 = (B AND C) OR (B AND D) OR (C AND D)

F3 = B XOR C XOR D

Эти функции используются на различных этапах, чтобы их результат был в конечном итоге увеличен с каждым новым шагом.