#### **Инструкция ветвления if**

Инструкция if является инструкцией управления, реализующая алгоритмическую структуру "ветвление". Синтаксически это выглядит так:

**if <логическое выражение>:**

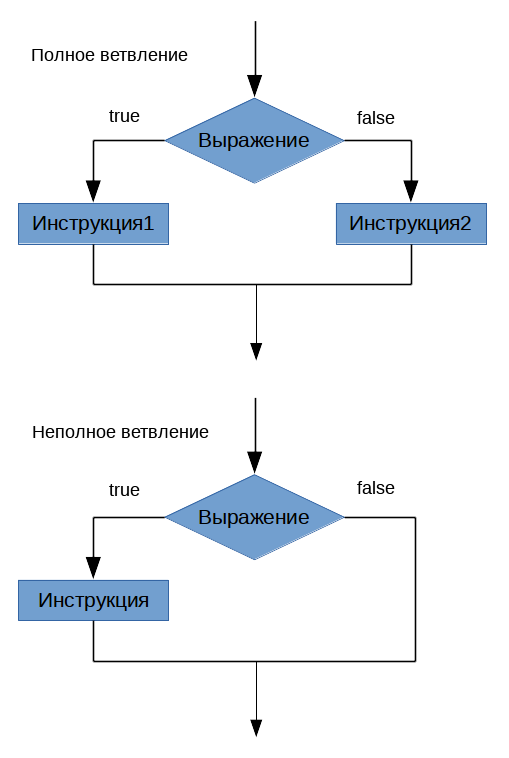
**<блог операций 1>**

**else:**

**<блог операций 2>**

Отступы (например, табуляция) обязательны в блоках if и else. Отступы должны быть одинаковой символьной длины, т. е. это либо табуляция, либо (например) 4 символа пробела. Для совместимости кода, не следует использовать в оформлении блоков смешанные символы (табуляции и пробела). Python имеет самобытный синтаксис в котором отступы - часть строгого синтаксиса. Нарушение этих правил приведет к синтаксической ошибке. Отступы являются ограничителями блока. Блоком называется инструкция или группа инструкций, которые выполняются как одна в инструкциях управления: if, for, while, def и др. Среды программирования, такие ка PyCharm, создают отступы в определённых участках кода автоматически.

Блок **else** является необязательным (неполное ветвление).

  
Постановка задачи.

**Программа 1** Неполное ветвление. Определить является ли число N кратным числу K.

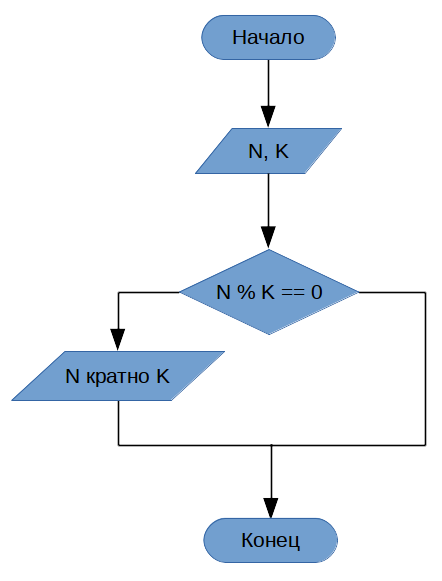
N = int(input("N = "))

K = int(input("K = "))

if N % K == 0:

print(N, "кратно", K)

Блок-схема к программе

  
**Программа 2** Полное ветвление. Введены два числа. Определить порядковый номер большего из них.

a = int(input("a = "))

b = int(input("b = "))

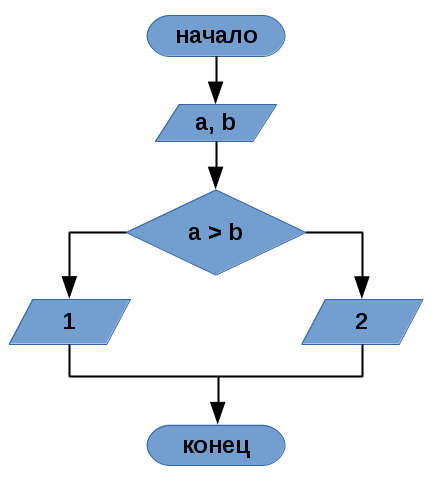
if a > b:

print("Число a больше b. Его порядковый номер 1")

else:

print("Число b больше или равно a. Его порядковый номер 2")

Блок-схема к программе



#### **Инструкции if, вложенные друг в друга**

Поскольку конструкция if else воспринимается как одна инструкция, то она может входить внутрь другой условной инструкции, создавая, таким образом, структуру вложенного ветвления. Для более удобного представления блоков вместо else if ... ("иначе если") используется сокращение elif. В более общем виде синтаксис условной инструкции принимает вид:

if <логическое выражение1>:

<блог операций 1>

elif <логическое выражение2>:

<блог операций 2>

elif <логическое выражение3>:

<блог операций 3>

…

else <логическое выражениеN>:

<блог операций N>

**Программа 3.** Дано целое число K. Вывести строку-описание оценки, соответствующей числу K (1 — «плохо», 2 — «неудовлетворительно», 3 — «удовлетворительно», 4 — «хорошо», 5 — «отлично»). Если K не лежит в диапазоне 1–5, то вывести строку «ошибка».

k = int(input("k = "))

if k == 1 : print("Плохо")

elif k == 2 : print("Неудовлетворительно")

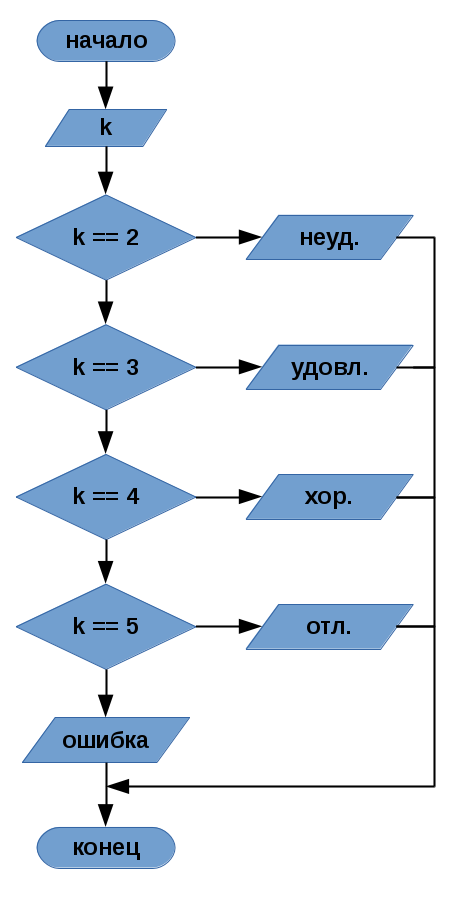
elif k == 3 : print("Удовлетворительно")

elif k == 4 : print("Хорошо")

elif k == 5 : print("Отлично")

else : print("Ошибка!")

Блок-схема к программе

  
**Программа 4.** Вложенное ветвление. Даны две переменные целого типа: A и B. Если их значения не равны, то присвоить каждой переменной большее из этих значений, а если равны, то присвоить переменным нулевые значения. Вывести новые значения переменных A и B.

a = int(input("a = "))

b = int(input("b = "))

if a != b :

if a > b:

b = a

print ("b= ",b)

else:

a = b

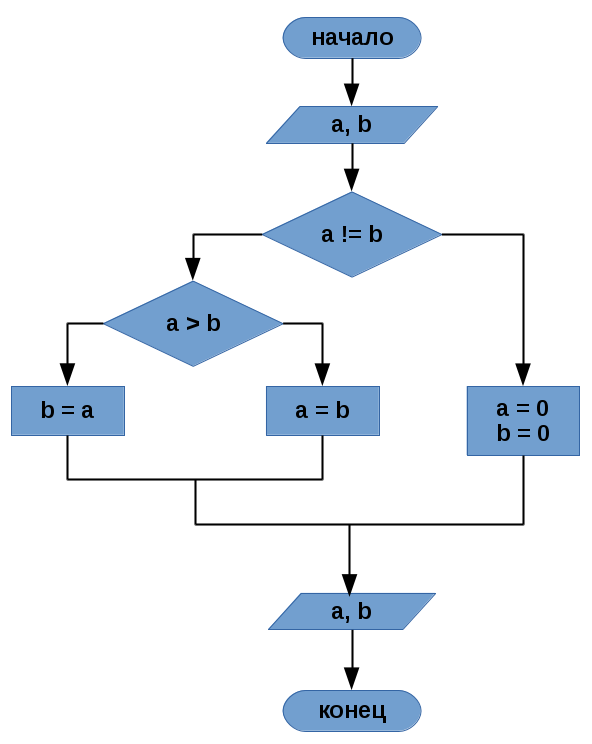
print ("a= ",a)

else:

a = b = 0

print ("a=b= ",a)

Блок-схема к программе 4

  
Хотя наличие отступов и делает структуру программы яснее, тем не менее, вложенные условия (nested conditional) очень быстро затрудняют чтение программы, поэтому большую глубину вложенности следует избегать, например, использовать несколько последовательных условных инструкций как в следующей программе.

**Программа 5** Каскадные условные инструкции. Даны три целых числа. Найти количество положительных чисел в исходном наборе.

a = int(input("a = "))

b = int(input("b = "))

c = int(input("c = "))

i = 0

if a > 0: i += 1

if b > 0: i += 1

if c > 0: i += 1

print("Количество положительных чисел равно", i)

#### **Тернарная операция**

В Python, как и в других языках программирования, существует альтернатива инструкции if - Тернарная операция. Строго говоря, в python нет тернарной операции. Это все та же инструкция управления, но представленная в необычной форме. Тернарная она называется потому, что в ней требуется наличие трех операндов:

a = <выражение1> if <логическое выражение> else

<выражение2>

В этой инструкции переменная a будет ссылаться на значение выражения выражение\_1, если логическое выражение истинно, в противном случае a будет ссылаться на значение выражение\_2.

Тернарная инструкция позволяет перефразировать инструкцию if в более лаконичной форме. Например, инструкцию:

if a % 2 == 0:

   i+=1

else:

   i\*=2

можно заменить на такую:

**i += 1 if not(a % 2) else i \* 2**

Если тернарная инструкция входит в более сложное выражение, то, ввиду старшинства приоритета арифметических операций, её следует заключать в круглые скобки.

Например:  
width = 100 + (10 if margin == 10 else 0)

Если бы не было скобок, то при значении margin не равным 10, значением width равнялось бы 0.

Тернарная инструкция имеет существенное отличие от обычной условной инструкции: последнюю нельзя применять в выражениях с "=".

##### **Задание**

1. Даны три действительных числа. Возвести в квадрат те из них, значения которых неотрицательны, и в четвертую степень - отрицательные
2. Даны две точки A(x1, y1) и B(x2, y2). Составить программу, определяющую, которая из точек находится ближе к началу координат.

**Циклы**

#### **Инструкция while**

Инструкция цикла while в языке Python имеет следующий синтаксис:

**while логическое выражение:**

**инструкции1**

**[else:**

**Инструкции2]**

Пока логическое выражение истинно, будут выполняются инструкции в теле цикла. Логическое выражение — это булевское выражение результатом вычисления которого является True или False. Вместо булевского выражения может быть любое выражение скалярного типа. Если результатом вычисления такого выражения является значение отличное от нуля, то это интерпретируется как «истина», в противном случае — как «ложь». Если логическое выражение приобретает ложное значение, то будет выполнен необязательный блок после else.  
Примечание. Блок else не будет выполняться, если цикл прерывается инструкцией break или continue (см. ниже). Обратите внимание, что блок else будет выполнятся и в том случае, если условие ложно изначально!  
**Программа**

i = 0

while i < 10:

i+=1

print(i, end=' ')

else:

print("\nЦикл звершил свою работу!" \

"\nКоличество итераций -", i)

Примечание. Для переноса части логической строки (в исходном коде программы) используется символ "\". После этого символа необходимо перейти на новую (физическую) строку, т. е. нажать на клавишу Enter.  
Сложность программирования такого типа цикла заключается в том, что разработчику необходимо продумывать изменение параметров в теле цикла так, чтобы эти изменения влияли на завершение цикла. Иначе говоря, результатом вычисления логического выражения должно стать, когда-нибудь, значение False. Неверно составленный программный код может привести к «зацикливанию» (т. е. невозможности выхода из цикла).  
В некоторых случаях требуется реализация «бесконечного цикла». Это можно сделать, например, таким образом:

**while True:**

**Инструкции**

Выход из «бесконечного» цикла можно реализовать с помощью инструкций if и break.

В отличие от других языков программирования (например, C++), блок в инструкции циклов и if не определяет область видимости данных. Переменные, с которыми связываются некоторые значения, как внутри блока, так и вне его — имеют глобальную видимость по отношению к программе (в данном модуле). Python по местоположению операции "=" связывает имя с конкретным пространством имен. Локальную область видимости имеют переменные (и другие объекты), которые определяются внутри блока функций — def. Об этом мы будем говорить позднее. Если требуется определить некоторый объект без инициализации применяется такая форма: var = None

#### **Накопители суммы и произведения**

В циклах часто приходится выполнять действия получения (накопления) суммы или произведения. Для этих целей используются накопители. Примером такого накопителя является счетчик, который увеличивает значение (считает шаги цикла) на единицу с каждой итерацией (программа 7.1). Для использования таких накопителей переменная (целого или вещественного типа) принимает начальное значение "0" для накопления суммы и начальное значение "1" для накопления произведения вне тела цикла (до начала цикла). Пример использования счетчика и накопителя произведения в следующей задаче.  
Постановка задачи: Вывести на экран все целые степени числа 2 от 1 до 10.  
**Программа**

i = 0

p = 1

while i < 10:

i+=1

p\*=2

print('{:d} в степени {:2d} = {:d}'.format(2, i, p))

Эту программу можно переписать с использованием операции побитового сдвига влево на 1 (ранее мы уже говорили об этой операции). Тогда инструкцию в строке 5 нужно переписать так:  
p = p << 1 или p <<= 1  
Циклы, возможно, не были столь полезны, если бы отсутствовала возможность обработки потока данных. В следующем классе задач в теле цикла используется инструкция if.  
Постановка задачи. Вывести ряд двузначных чисел, сумма разрядов которых кратна 3 и определить сумму членов этого ряда.  
**Программа**

i = 10

s = 0

while(i < 100):

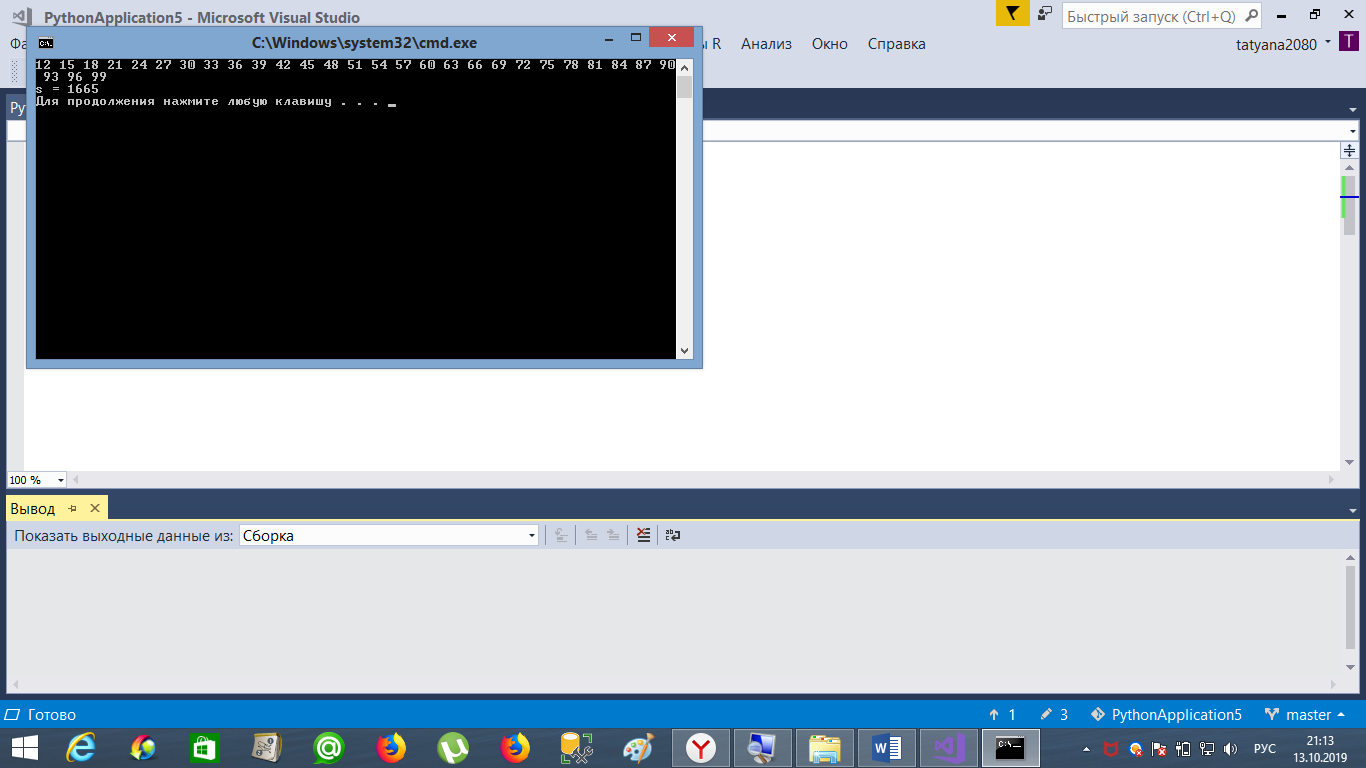
if (i // 10 + i % 10) % 3 == 0:

print(i, end = ' ')

s += i

i+=1

print("\ns =", s)



#### **Использование в цикле while случайных чисел**

В некоторых задачах, в качестве "источника" данных, удобно применять генератор псевдо-случайных чисел. Реализация этого генератора находится в модуле random. Этот модуль содержит довольно много различных функций. В программе 4 используется функция randint. Эта функция имеет два аргумента randint(a, b), которые, собственно, определяют отрезок множества целых чисел из которого будут получены случайные числа: [a, b]. Более подробные сведения об использовании генератора случайных чисел мы дадим на следующем уроке.  
Используя генератор случайных чисел получить числовой ряд из промежутка от a до b включительно, в количестве 100 элементов. Определить, каких чисел больше - четных или нечетных в этом ряде? Если таких чисел равное количество, то вывести 0. Выведите числовую последовательность в виде прямоугольной таблицы - матрицы 10х10.  
**Программа**

from random import randint

a = int(input("a = "))

b = int(input("b = "))

i, j = 0, 0

while i < 100:

i += 1

k = randint(a, b)

print(k, end = ' ')

if not k % 2:

j += 1

if not i % 10:

print()

if j > 100 - j:

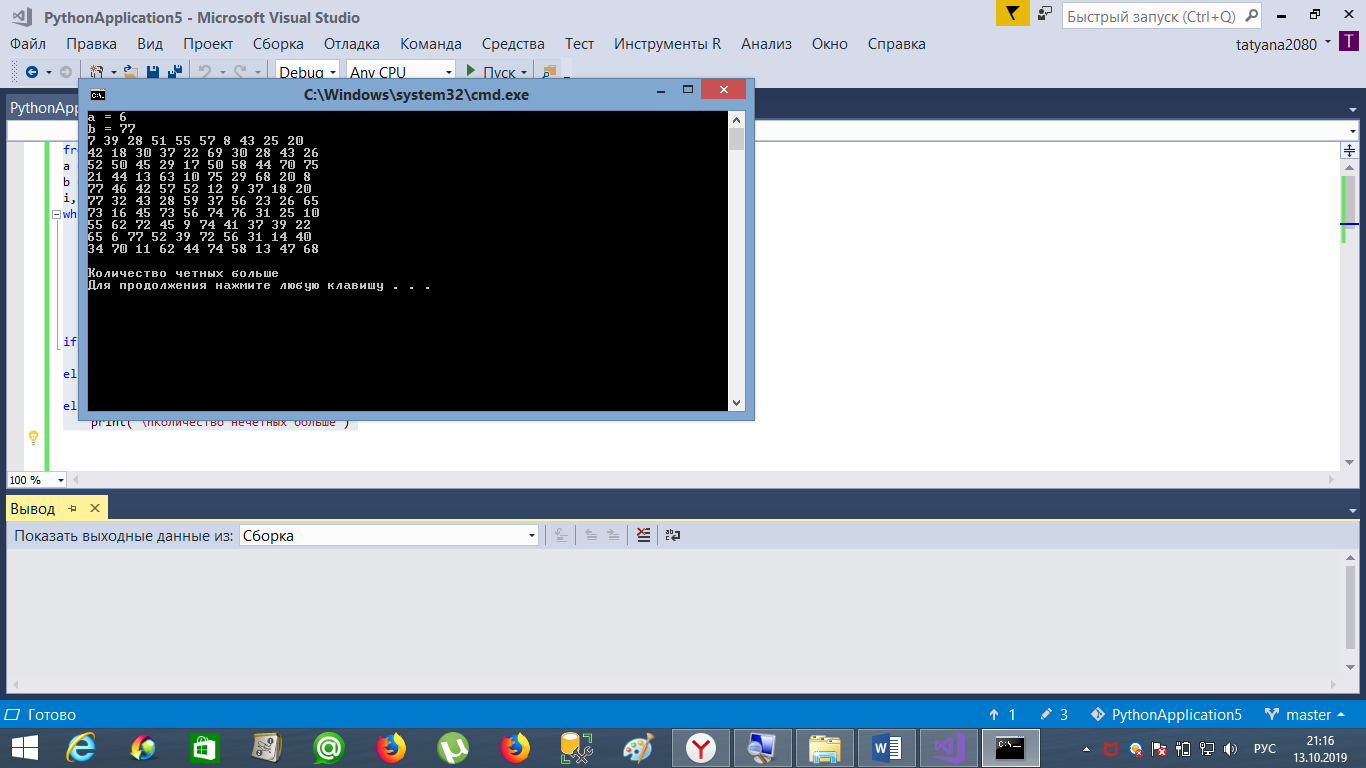
print("\nКоличество четных больше")

elif j == 50:

print("\n0")

else:

print("\nКоличество нечетных больше")



#### **Инструкции break и continue**

##### **break**

Существует класс задач в которых производится анализ цифр в числе. Рассмотрим подобный пример в программе 7.5.  
Дано число целое число n. Определить, имеются ли в числе две равные цифры, находящиеся в соседних разрядах. Например 10221, или 355, или 770.  
**Программа**

n = int(input("n = "))

f = False

while n // 10 != 0:

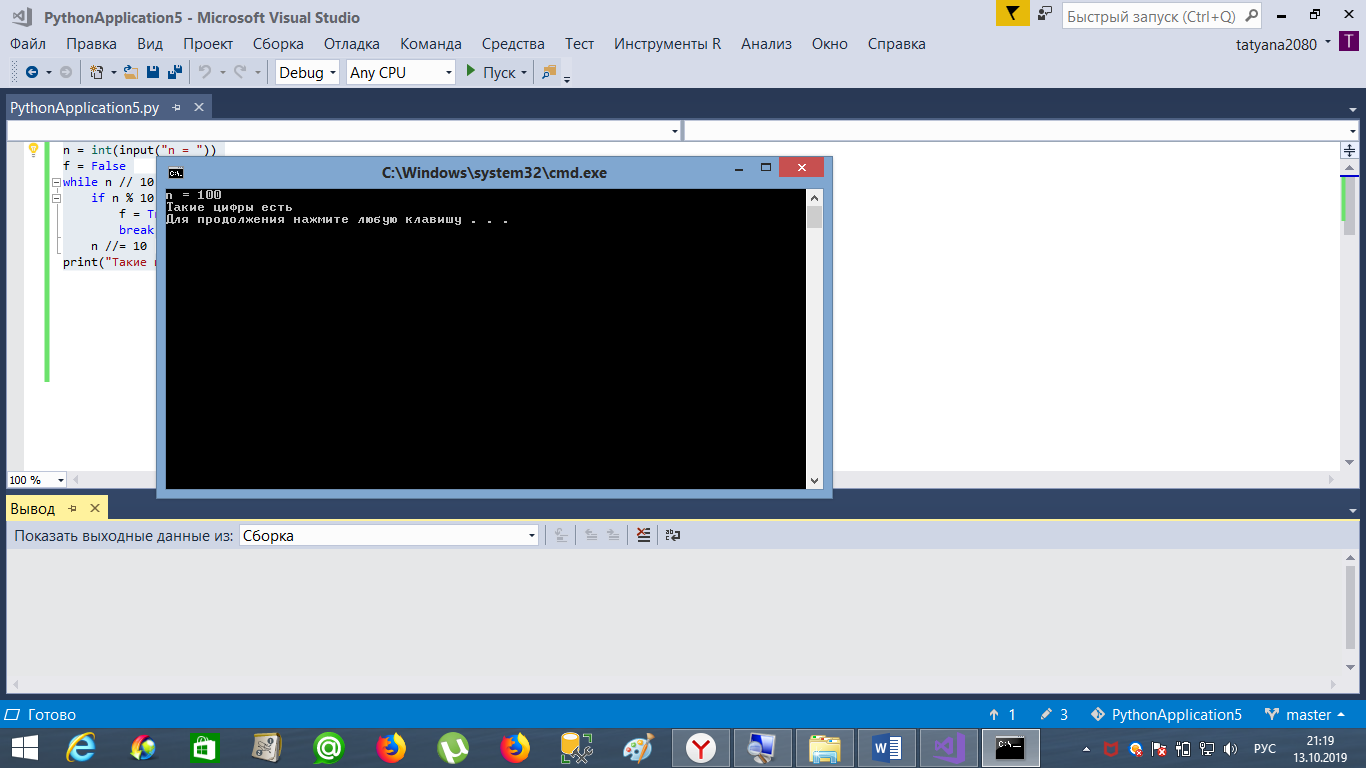
if n % 10 == n // 10 % 10:

f = True

break

n //= 10

print("Такие цифры есть" if f else "Таких чисел нет")



Примечание. Логические переменные применяются в качестве флагов, фиксирующих некоторое состояние объектов в программе. Их использование не обязательно, но выглядят они более элегантно и делают код более понятным.

В программе, чтобы прервать цикл (если найдены две одинаковые цифры), используется инструкция break. Инструкция break прерывает выполнение цикла while и for и передает управление программой инструкции, следующей за блоком цикла. Пример.  
Постановка задачи: Дано натуральное число n. Определить, является ли оно простым.

**Программа**

i = 2

k = True

n = int(input("n = "))

while i <= n / 2:

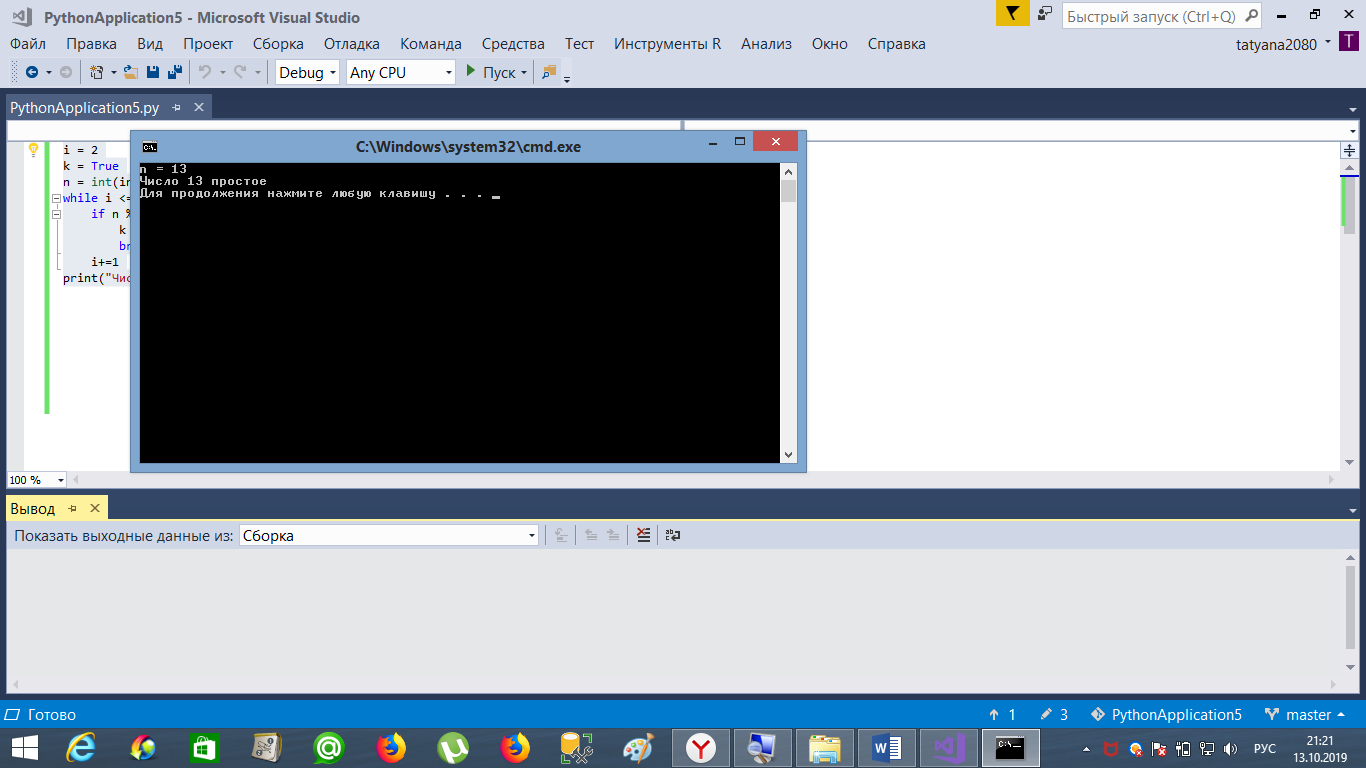
if n % i == 0:

k = False

break

i+=1

print("Число", n, ("простое" if k else "не является простым"))



Примечание. Если найден делитель числа, то нет необходимости просматривать все возможные делители числа, вплоть до граничного условия. Нужно немедленно прервать выполнение цикла.

##### **continue**

В отличие от break - continue прерывает выполнение только текущей итерации и передает выполнение на вычисление условия. При этом инструкции, следующие за continue до конца блока, игнорируются. Данная инструкция применяется не часто, но может быть использована, например, для контроля допустимости значений в выражениях тела цикла. Требуется вывести значения функции f(x) = 1 / x с шагом 1 (протабулировать) на отрезке от -20 до 20. Значение x == 0, в котором функция неопределена, должно быть пропущено.  
**Программа**

i = -20

while i <= 20:

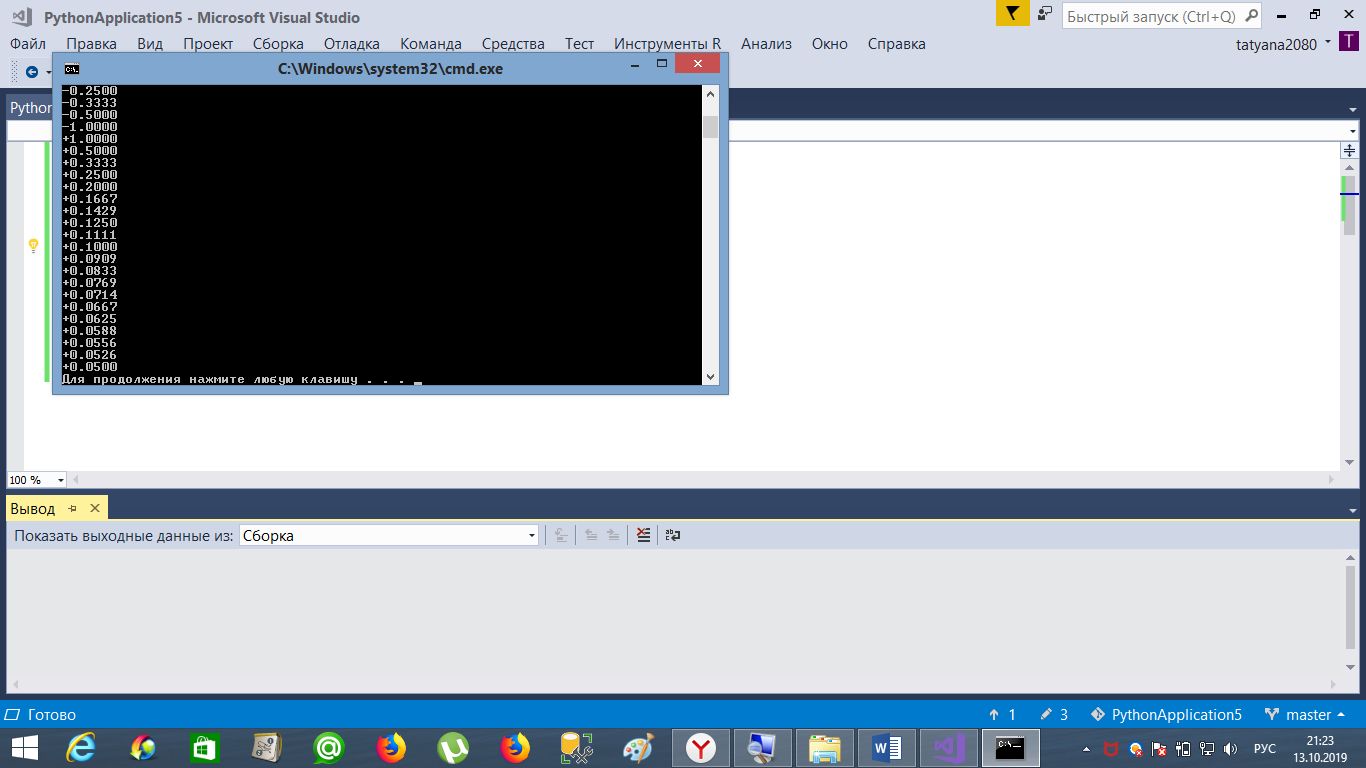
if i == 0:

i+=1

continue

print("{:+7.4f}".format(1/i))

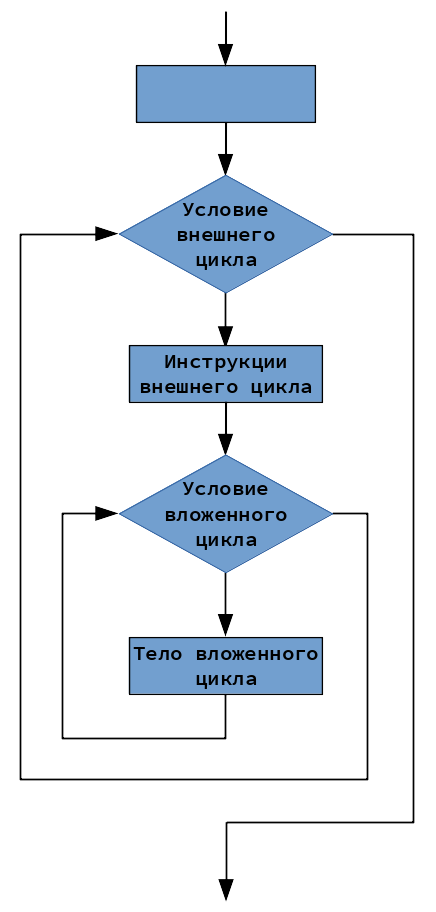
i+=1



Следует сказать, что многие разработчики стараются не использовать в циклах инструкции break и continue, так как вполне можно обойтись и без них. Однако, последние повышают читаемость программы.

#### **Циклы while вложенные друг в друга**

Инструкция одного цикла while может находиться внутри инструкции другого цикла. Такая структура называется структурой вложенных циклов. Работа такой структуры совершается следующим образом: на одну итерацию внешнего цикла вложенный цикл совершает все возможные итерации. Нередко требуется изменять количество итераций вложенного цикла изменением параметра внешнего, иначе говоря, ставить в зависимость изменение параметра одного цикла от параметра другого.  
Блок-схема вложенных циклов while:

  
Решим классическую задачу. Дано целое n. Разложить число n на простые множители.  
**Программа**

n = int(input('n = '))

i = 2

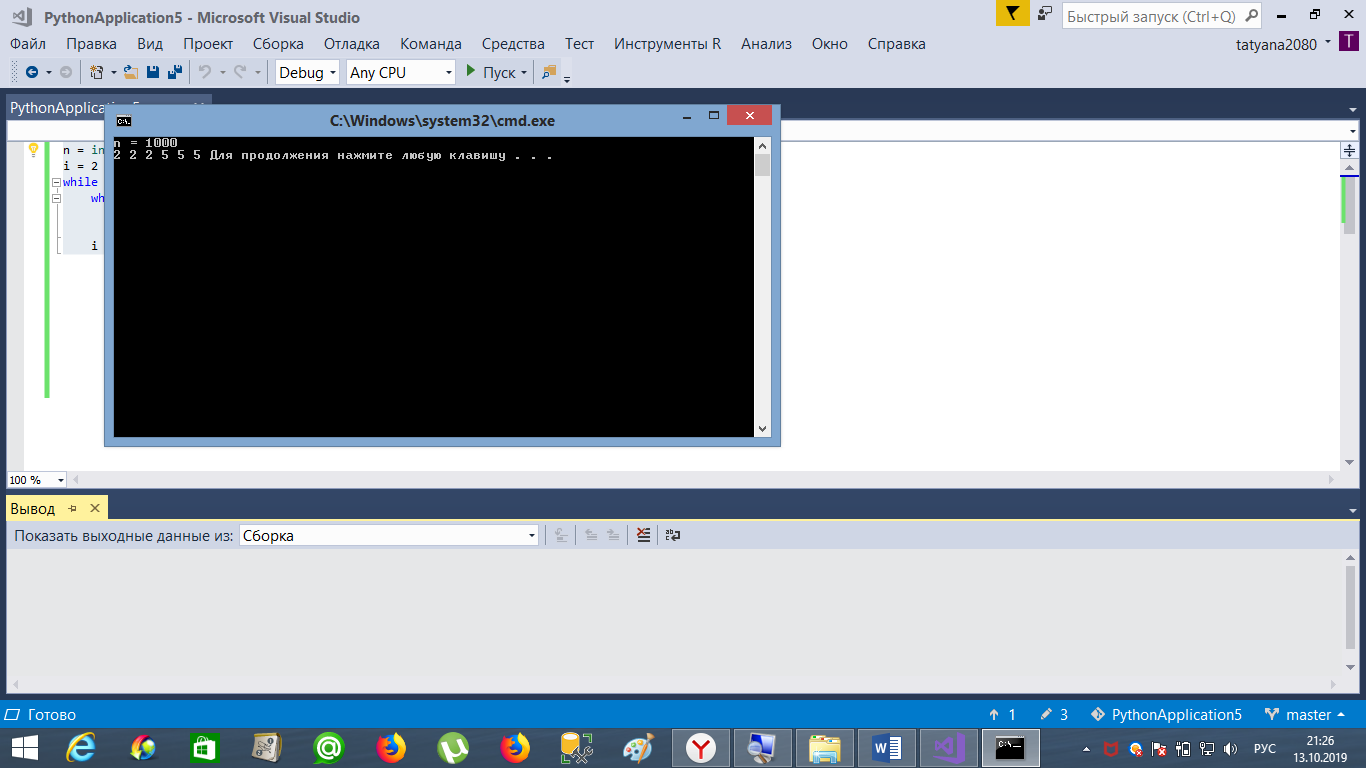
while i < n // 2:

while not n % i:

print(i, end=' ')

n //= i

i += 1



Идея алгоритма в следующем. Во внешнем цикле мы перебираем возможные делители числа. Во вешнем цикле мы делим число на делитель, пока это возможно.

##### **Задание**

1. Составить программу. Вводится натуральное число N. Найти сумму и произведение всех натуральных чисел от 1 до N
2. Составить программу. Вводятся натуральные число n, n > 10 и цифра k. Определить сумму цифр числа, а также есть ли цифра k в числе n.

Пример. Найти сумму четных цифр числа

m = int(input('Введите исходное число: '))

S = 0

while m > 0 :

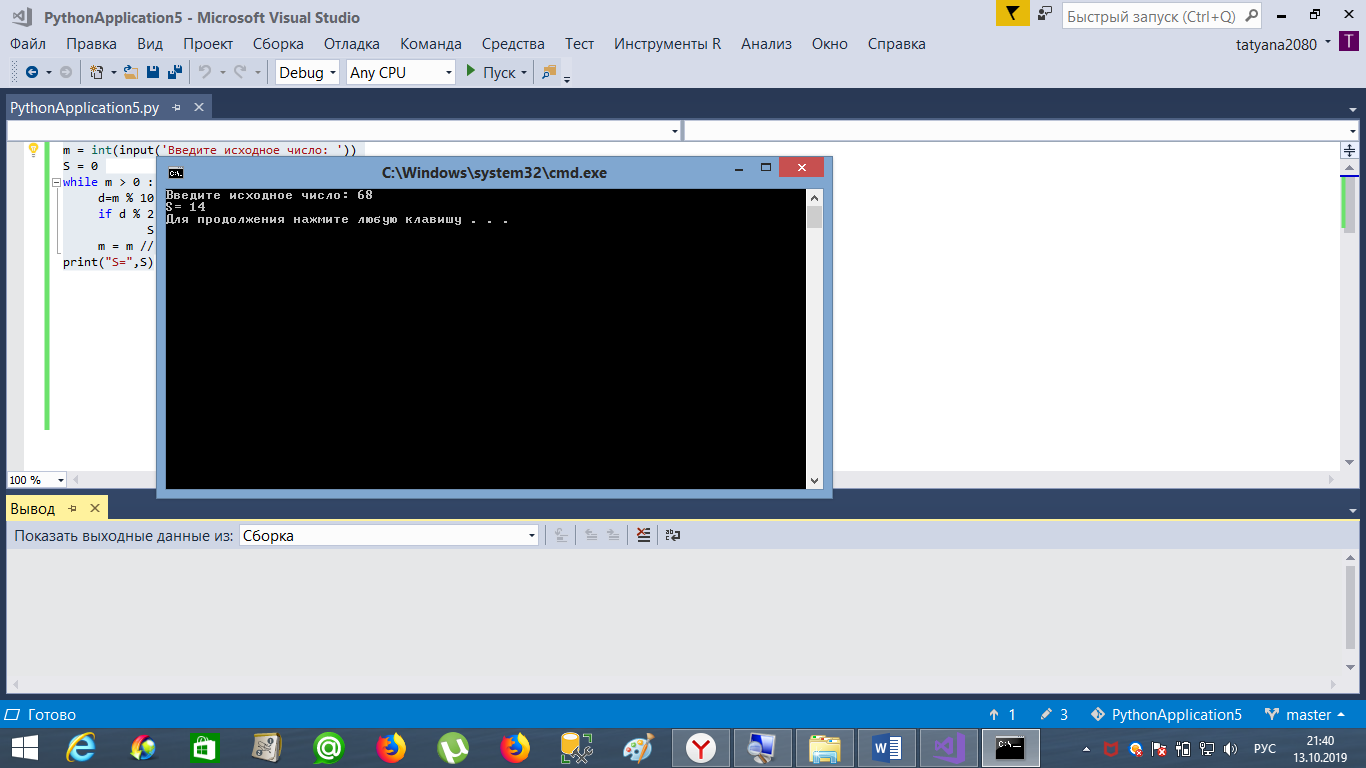
d=m % 10 # цифра числа

if d % 2 == 0: # если цифра четная, то суммирование

S += d

m = m // 10

print("S=",S)



Пример. Вычислить сумму первых десяти натуральных чисел.

1.

S = 0

for i in range (10) :

S = S + (i + 1)

print("S=",S) # 55

2.

S = 0

for i in range (1, 11) :

S = S + (i + 1)

print("S=",S) # 65

3.

S = 0

for i in range (10, 0,-1) :

S = S + (i + 1)

print("S=",S) # 65

4.

S = 0

i = 1

while i <= 10 :

S = S + i

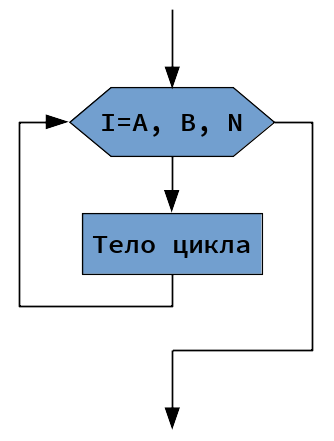
i+= 1

print("S=",S) # 55

#### **Цикл итераций по последовательности for**

Цикл for в python не является универсальным (по сравнению с подобными инструкциями в других языках, например, в С++). Но, по задумке создателей языка, он на это и не претендует, его задача в другом – он предназначен для итераций по последовательностям, таким как список (с которым мы познакомимся уже на следующем уроке). Отсюда его необычный синтаксис.

Тем не менее, цикл for в python реализует туже алгоритмическую структуру – цикл с параметром (или со счетчиком), что и реализации for в других языках. В виде блок схемы эта структура изображается так:

  
На этом рисунке: I – параметр (счетчик) цикла, A начальное значение параметра, B – его конечное значение, а N – шаг цикла (приращение счетчика). Особенность реализации цикла for в python в том, что в качестве модифицирующего выражения используется функция range(). Синтаксис цикла for:

**for target\_list in expression\_list**

**тело цикла**

**[else:**

**Выражение]**

Где target\_list – параметр цикла, а  expression\_list – имя последовательности (например, списка) или вызов функции range(). Также как и цикл while, for имеет необязательный блок else.  
Помимо обхода элементов коллекции, цикл for может применяться и в тех задачах, когда, прежде выполнения инструкций тела цикла, становится известным (или заранее определено) количество шагов этого цикла.

#### **Функция range()**

Формально range – это класс, создающий числовые целочисленные последовательности (арифметические прогрессии). Хотя range() и причисляют к коллекциям в python, но, фактически, значения элементов последовательности в памяти не сохраняются. Поскольку выражение с range() является всего-лишь правилом для создания определенной последовательности. Как превратить последовательность, созданную range().  
Функция может применяться с одним, двумя или тремя аргументами:

* range(A) – создается последовательность от 0 до A - 1 – [0, A-1];
* range(A, B) – создается последовательность от A до B – [A, B), B - не входит;
* range(A, B, N) – создается последовательность от A до B – [A, B) с шагом N, шаг может быть отрицательным;

Примеры показаны в программе ниже.  
**Программа**

for j in range(10):

print(j, end=' ')

print()

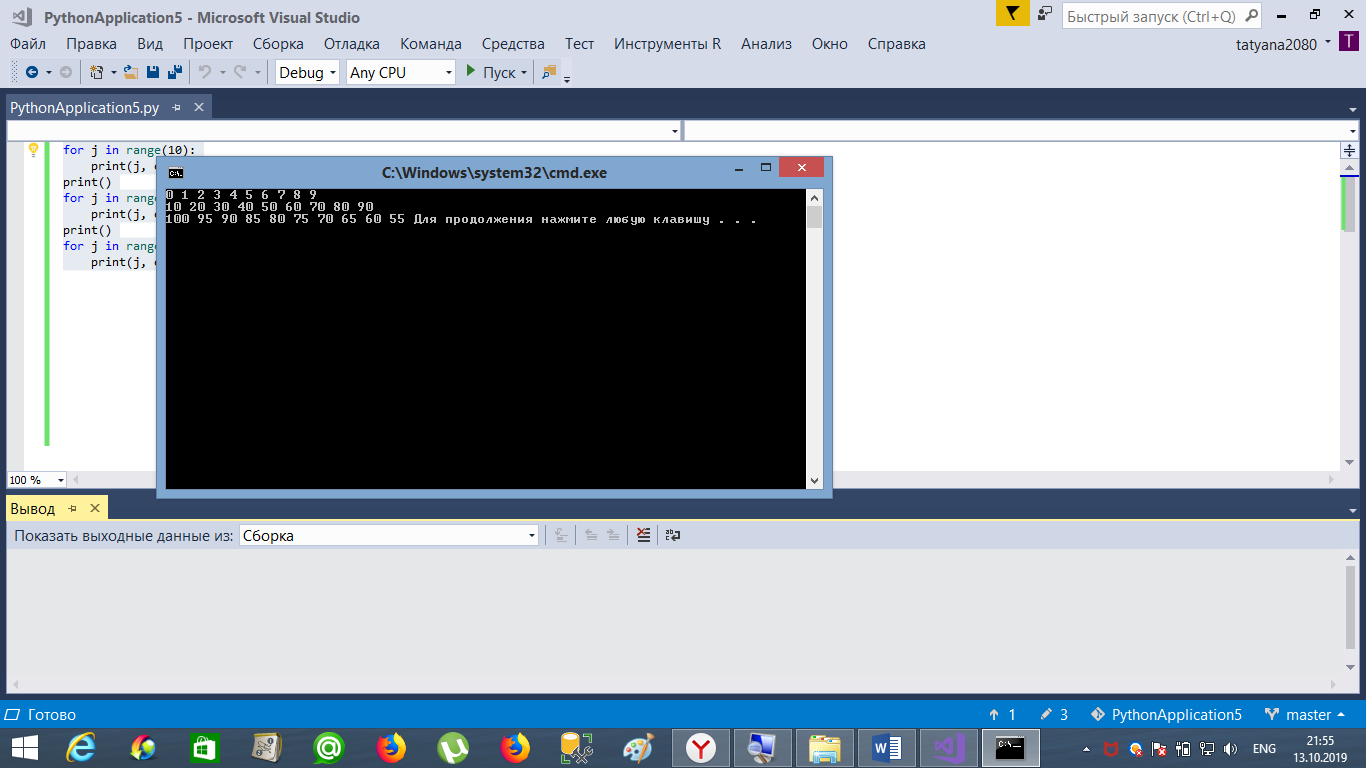
for j in range(10, 100, 10):

print(j, end=' ')

print()

for j in range(100, 50, -5):

print(j, end=' ')



Рассмотрим несколько задач по созданию рядов натуральных чисел.  
**Программа**  Создать последовательность нечетных чисел от 1 до n, включительно, если n нечетное.

n = int(input('n = '))

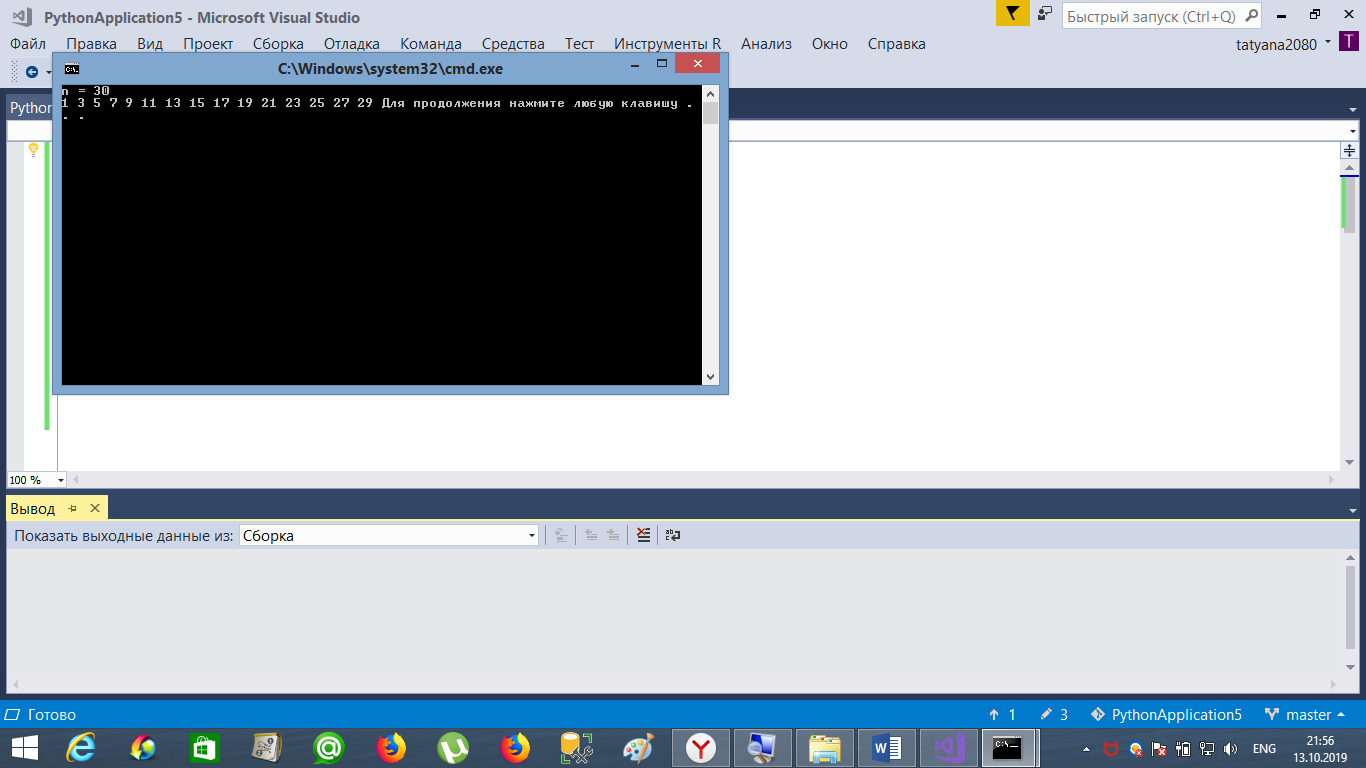
for j in range(1, n, 2):

print(j, end=' ')

else:

if n % 2:

print(n)



**Программа.** Вывести n чисел Фибоначчи. Числа Фибоначчи: f1 = 1, f2 = 1, fn = fn-1 + fn-2

f = None

f1, f2 = 1, 1

n = int(input('n = '))

print('1 =>', f1)

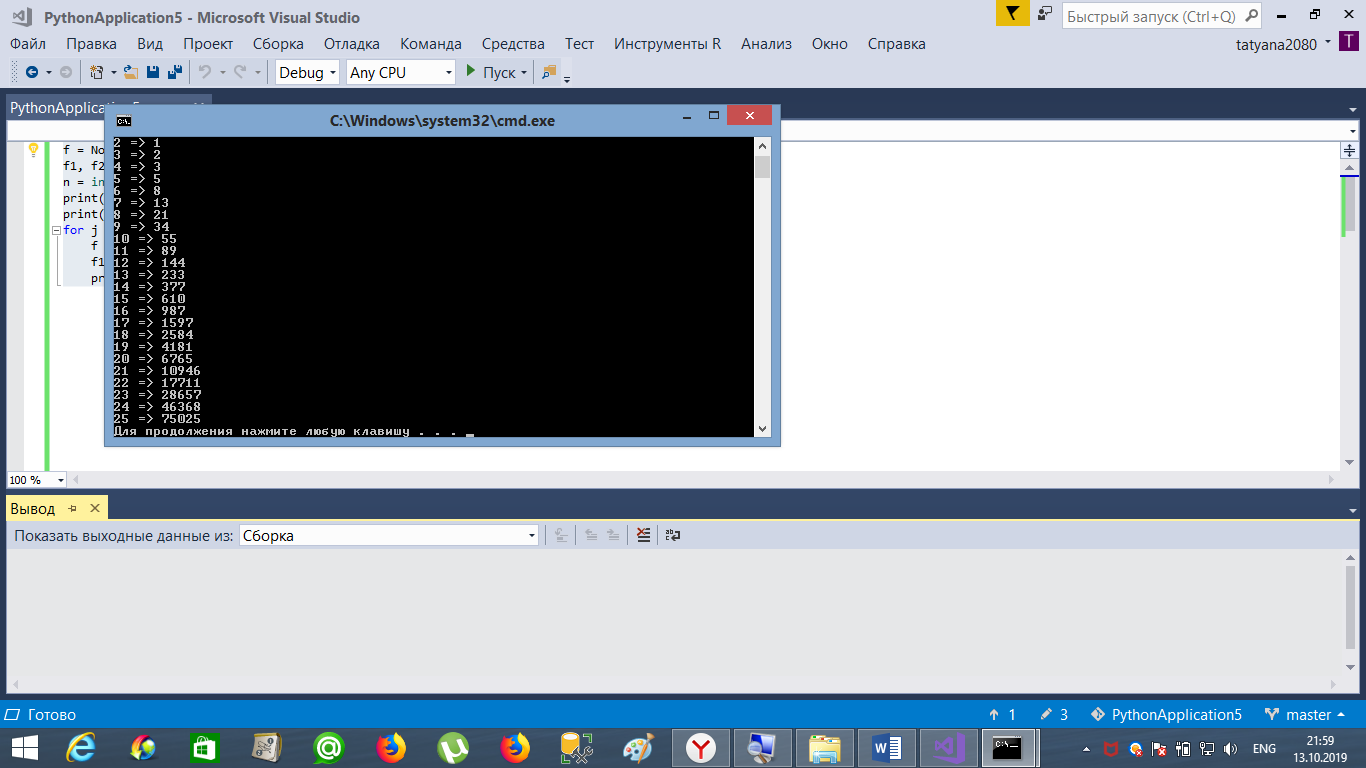
print('2 =>', f2)

for j in range(3, n + 1):

f = f1 + f2

f1, f2 = f2, f

print(j, '=>', f)



#### **Случайные числа**

Существует несколько классов задач (например, игровые программы, криптография, метод Монте-Карло, имитационное моделирование) для решения которых необходимо применять случайные числа. “Случайные числа”, на самом деле, не являются действительно случайными (поэтому их правильнее называть псевдослучайными), а подчиняются правилам математической теории вероятности и соответствующим функциям распределения из этой теории. В Python генератор случайных чисел реализован в модуле random.

Примечание. Генераторы псевдо-случайных чисел этого модуля не должны использоваться в целях обеспечения безопасности (криптография).

|  |  |
| --- | --- |
| Методы класса random | |
| Метод | **Описание** |
| random.seed(a=None, version=2) | Инициализация генератора, если параметры отсутствуют, то берется текущее системное время |
| random.randrange(start, stop[, step]) | Возвращает целое случайное число из интервала [start, stop) |
| random.randint(A, B) | Возвращает целое из отрезка [A, B] |
| random.random() | Возвращает случайное действительное число из интервала [0, 1). Используется без аргументов. |
| random.uniform(a, b) | Возвращает случайное действительное число N такое, что a <= N <= b для a <= b и b <= N <= a для b < a |
| random.choice(seq) | Случайный элемент из последовательности seq |
| random.shuffle(seq[, random]) | Перетасовывание элементов последовательности seq. Нельзя применять к неизменяемым коллекциям (например, строки) |
| random.sample(seq, k) | Возвращает список (новый, без замены) длиной k из последовательности seq, в котором элементы будут перетасованы |

**Программа.** С помощью модулей turtle и random построить гистограмму из 10 столбцов, в которой высота столба и цвет заливки столба определяются случайным образом.

import turtle

from random import randint, random

#################################################

# Установки окна и черепахи #

#################################################

window = turtle.Screen()

window.title('Гистограмма')

window.bgcolor('black')

window.setup(width=800, height=800)

t = turtle.Turtle()

t.speed(10)

t.hideturtle()

t.up()

t.goto(350, -350)

t.down()

t.color('red')

t.pensize(2)

t.bk(700) # Чертим ось x

t.lt(90)

t.up()

t.fd(1)

t.rt(90)

t.down()

t.pensize(1)

#################################################

# Установки гистограммы #

#################################################

l = 52 # Толщина столбца

m = 20 # Дистанция между столбцами

d = None # Подготовим высоту

#################################################

# Функция тик гистограммы #

#################################################

def tic():

R = random()

G = random()

B = random()

d = randint(10, 600)

t.color(R, G, B)

t.begin\_fill()

t.lt(90)

t.fd(d)

t.rt(90)

t.fd(l)

t.rt(90)

t.fd(d)

t.lt(90)

t.end\_fill()

t.up()

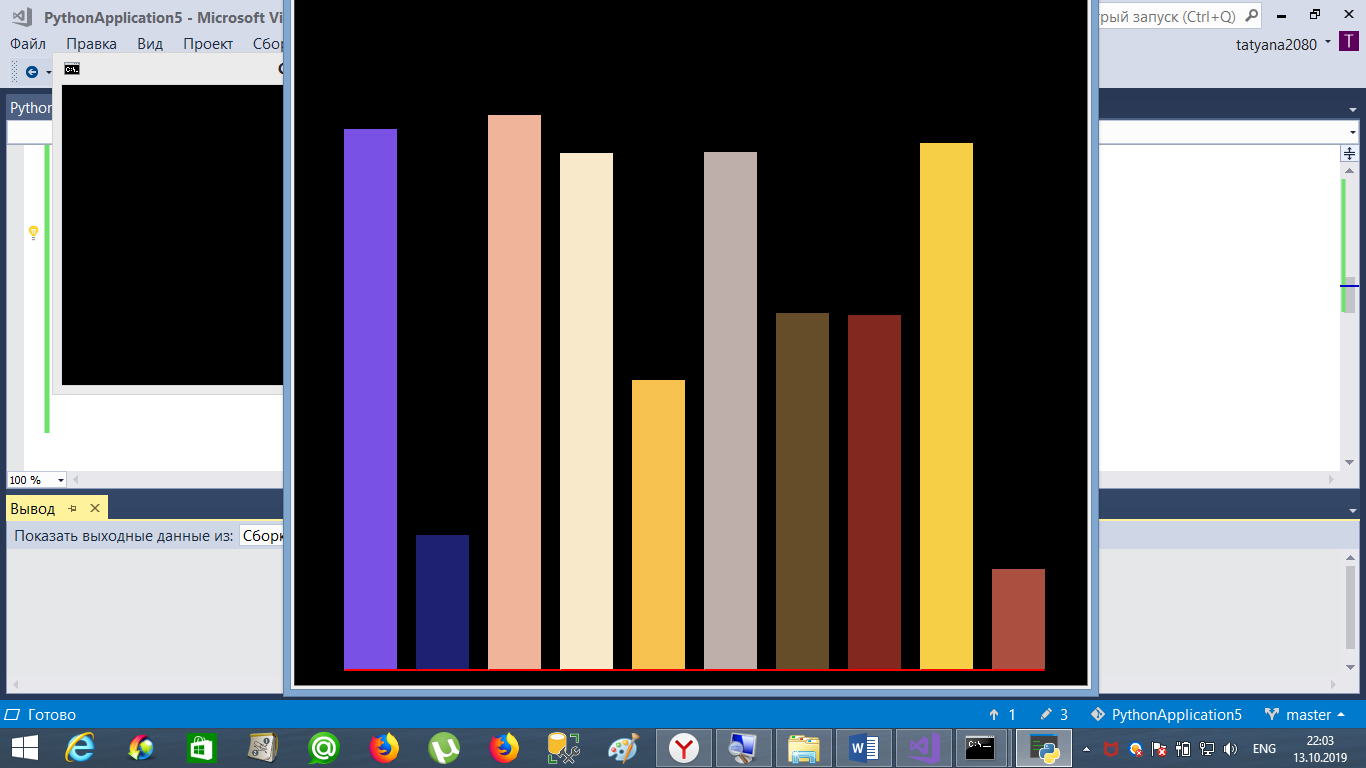
t.fd(m)

t.down()

for j in range(10):

tic()

window.exitonclick()

  
**Программа.** Разработать программу игры "Угадай число". Компьютер "загадывает" число, которое пользователь должен угадать за известное количество шагов.

from random import randint, seed

n = 10

a = None

seed()

# Компьютер загадал число:

k = randint(1, 100);

print('У вас', n, 'попыток')

for j in range(1, n + 1):

print('Попытка', j,

"\nКакое число загадал компьютер? -> ",

end='')

a = int(input())

if a > k:

print('много')

elif a < k:

print('мало')

else:

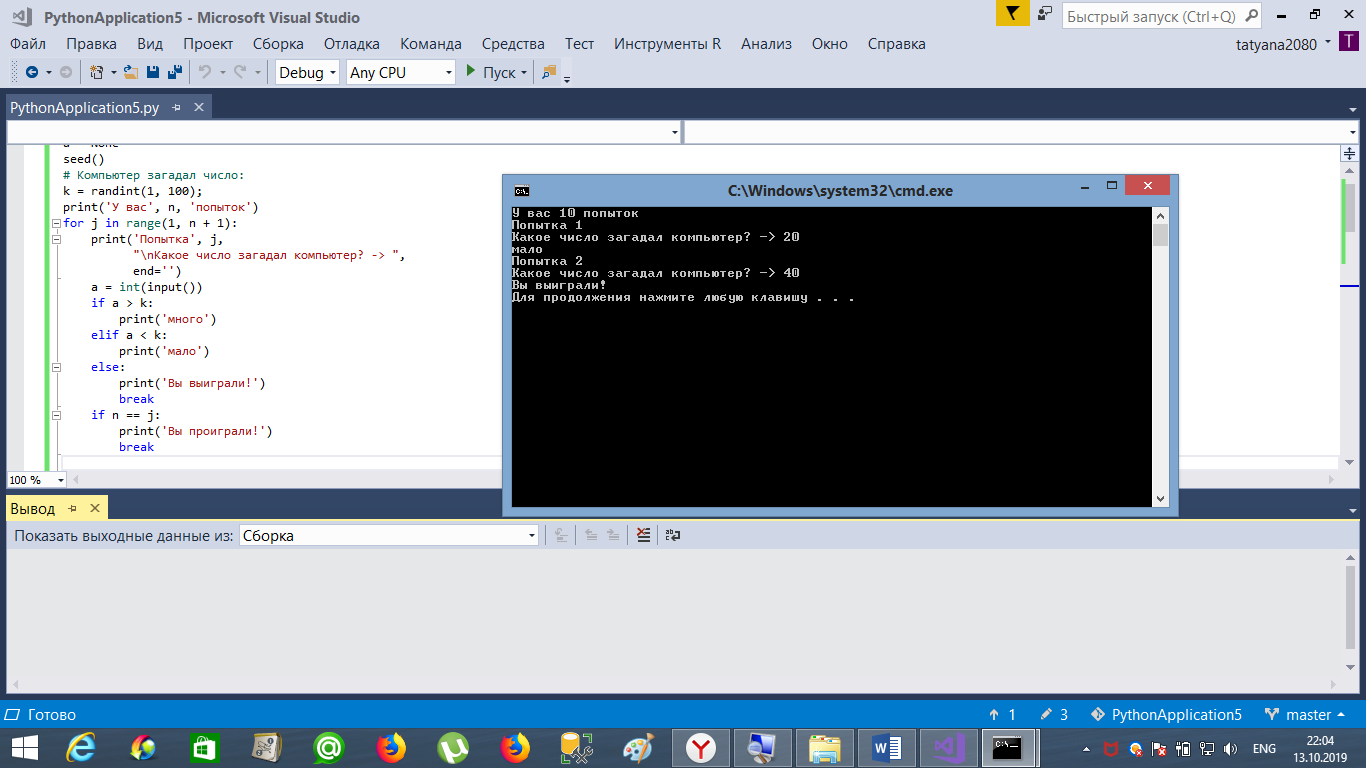
print('Вы выиграли!')

break

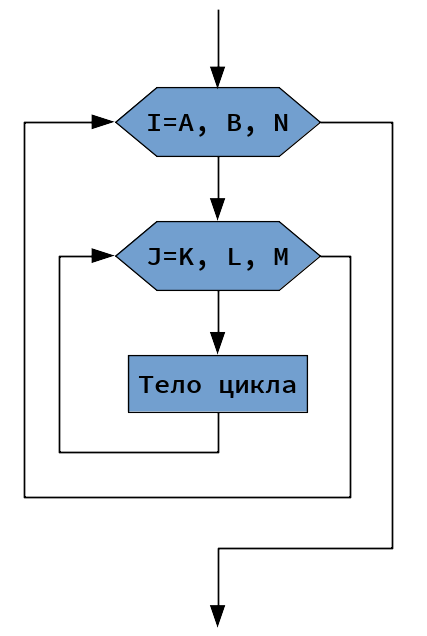
if n == j:

print('Вы проиграли!')

break



#### **Циклы for вложенные друг в друга**

Также как и циклы while, for могут быть вложенными друг в друга. Глубина вложенности может быть сколько угодно большая, но допускать этого не следует, поскольку с увеличением глубины вложенности сложность программы возрастает, и она становится плохо читаема. В блок-схеме такая структура изображается следующим образом:  
  
Рассмотрим класс таких задач.

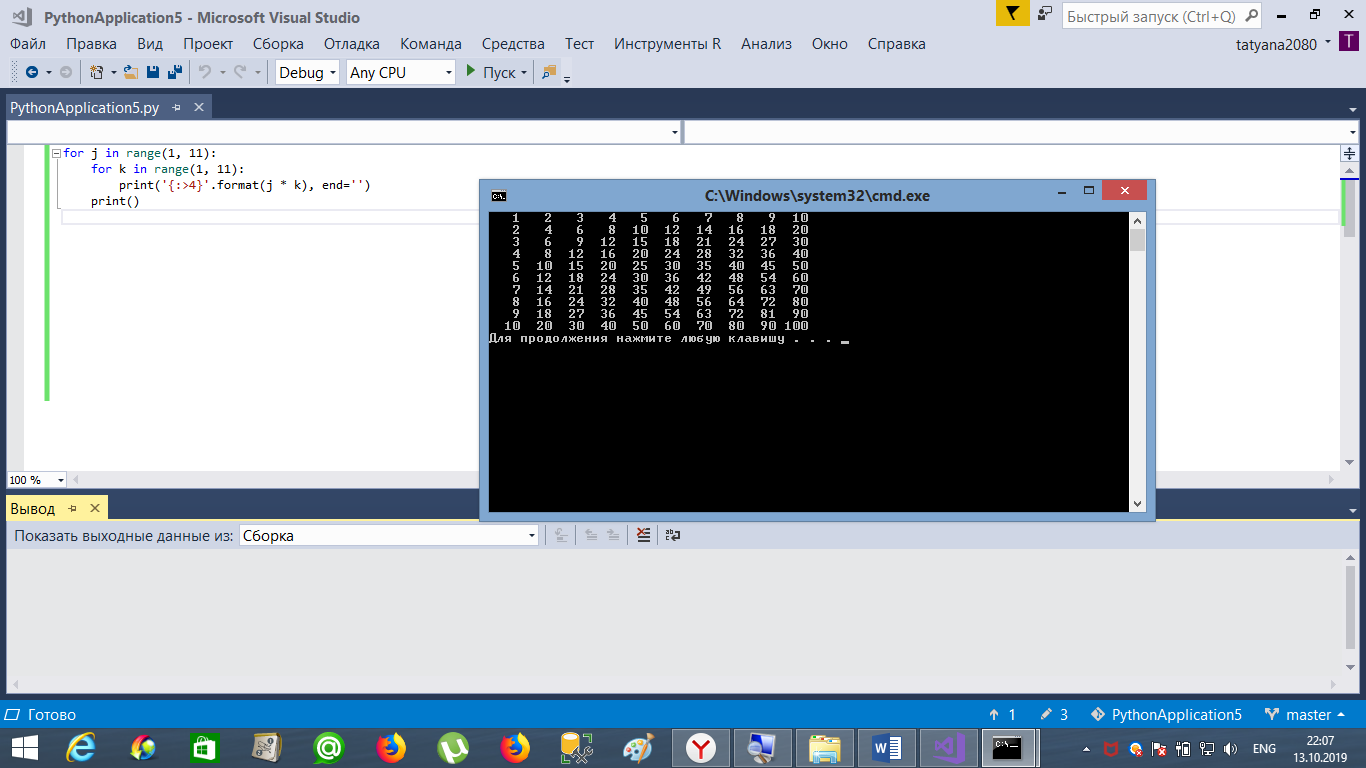
**Программа.** Вывести таблицу Пифагора (таблицу умножения в виде матрицы)

for j in range(1, 11):

for k in range(1, 11):

print('{:>4}'.format(j \* k), end='')

print()



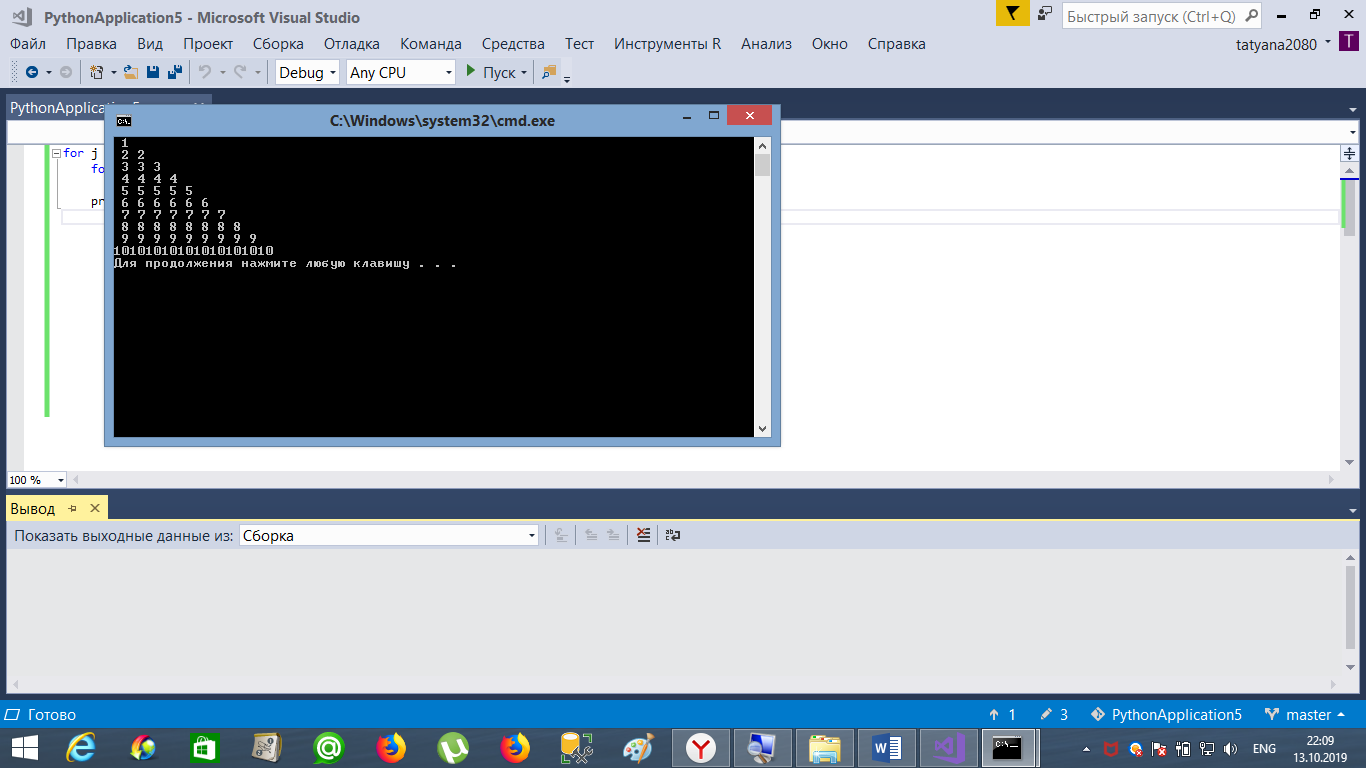
**Программа.** Составить программу выводящую числовой треугольник вида:

for j in range(1, 11):

for k in range(1, j + 1):

print('{:>2}'.format(j), end='')

print()



В программе мы поставили в зависимость количество шагов вложенного цикла от значения параметра внешнего.  
**Программа.** Дано целое n. Вывести все тройки чисел x, y и z для которых x2 + y2+ z2 = n

from math import sqrt

n = int(input('n = '))

m = int(sqrt(n)) + 1

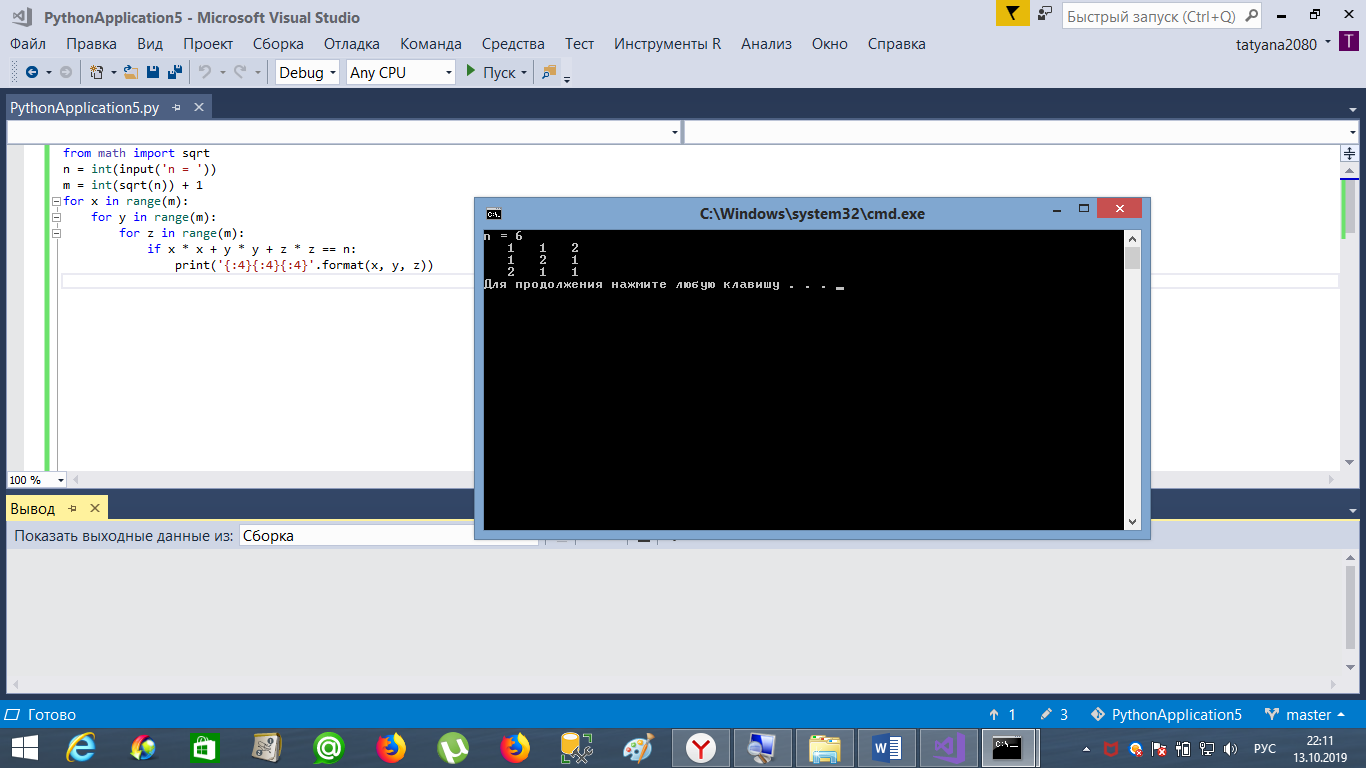
for x in range(m):

for y in range(m):

for z in range(m):

if x \* x + y \* y + z \* z == n:

print('{:4}{:4}{:4}'.format(x, y, z))



В заключение отметим, что один тип цикла (например, while) может быть вложен в цикл другого типа (for) и наоборот.  
**Программа.** Найти натуральное число от 1 до 10000 с максимальной суммой делителей

max = 0

for j in range(1, 10001):

s = 0

i = 2

while i < j // 2:

if j % i == 0:

s += i

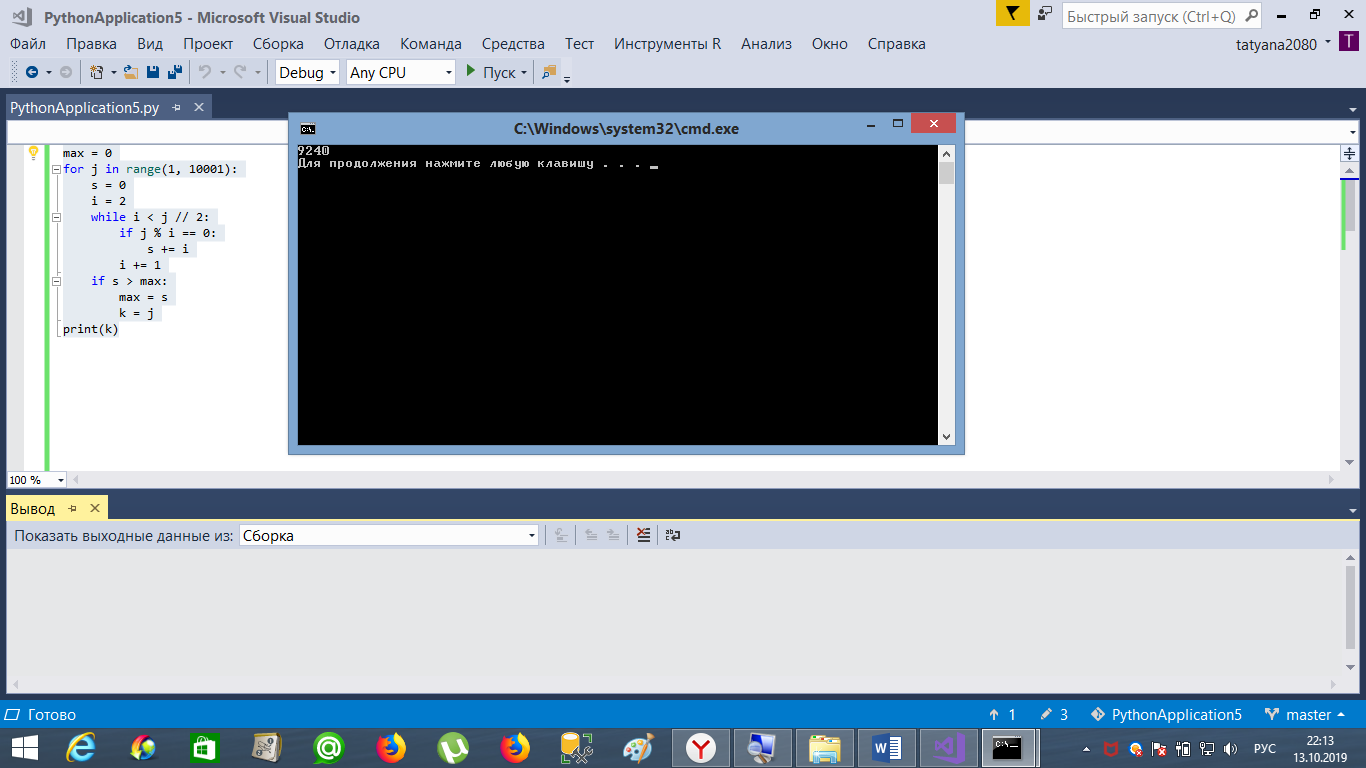
i += 1

if s > max:

max = s

k = j

print(k)



**Списки**

Напишите программу, на вход которой подаётся список чисел одной строкой. Программа должна для каждого элемента этого списка вывести сумму двух его соседей. Для элементов списка, являющихся крайними, одним из соседей считается элемент, находящий на противоположном конце этого списка. Например, если на вход подаётся список "1 3 5 6 10", то на выход ожидается список "13 6 9 15 7" (без кавычек.  
Если на вход пришло только одно число, надо вывести его же.  
Вывод должен содержать одну строку с числами нового списка, разделёнными пробелом.

a = [int(i) for i in input().split()]

if len(a) < 2:

print(' '.join([str(i) for i in a]))

else:

new = []

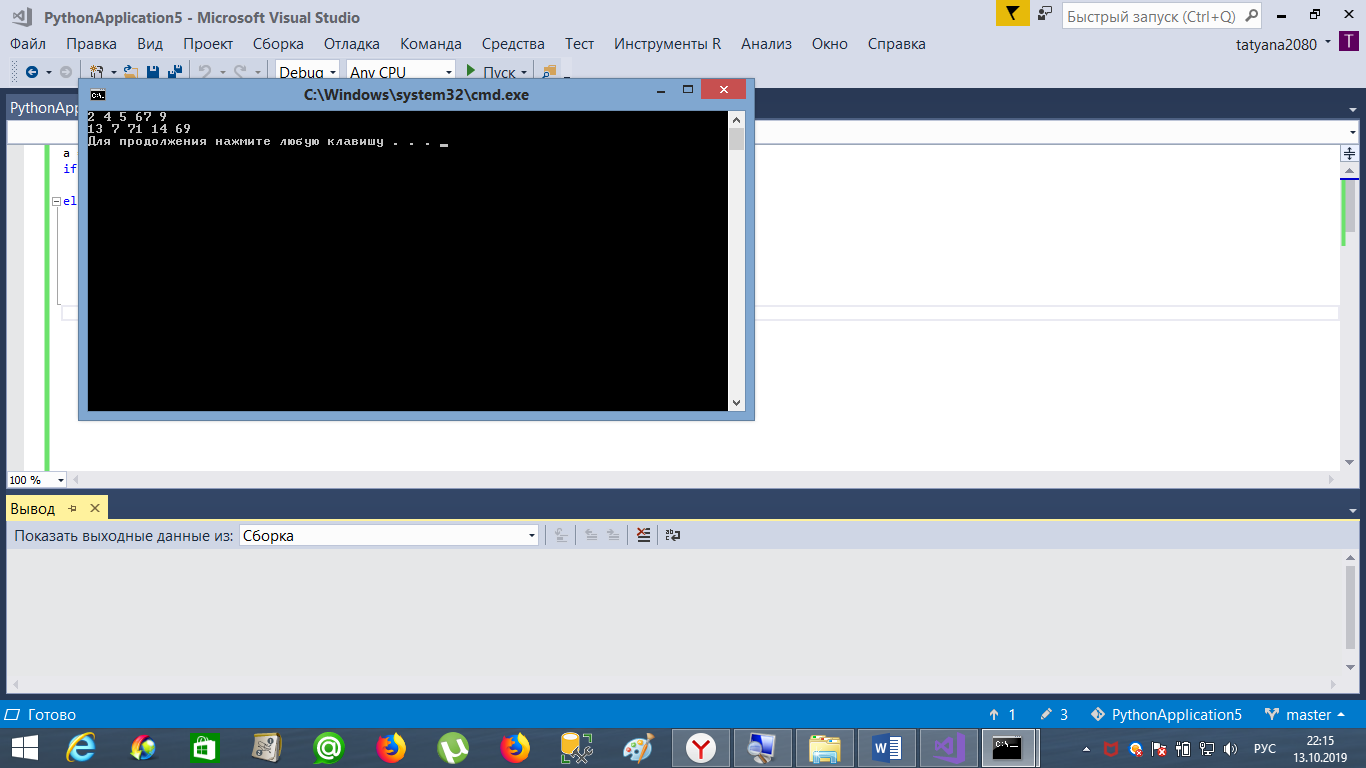
new.append(a[1] + a[len(a) - 1])

for i in range(1, len(a) - 1):

new.append(a[i-1] + a[i+1])

new.append(a[len(a) - 2] + a[0])

print(' '.join([str(i) for i in new]))



Напишите программу, которая выводит чётные числа из заданного списка и останавливается, если встречает число 237.

numbers = [

386, 462, 47, 418, 907, 344, 236, 375, 823, 566, 597, 978, 328, 615, 953, 345,

399, 162, 758, 219, 918, 237, 412, 566, 826, 248, 866, 950, 626, 949, 687, 217,

]

for x in numbers:

if x == 237:

break

elif x % 2 == 0:

print(x)

Напишите программу, которая принимает два списка и выводит все элементы первого, которых нет во втором.

set\_1 = set(['White', 'Black', 'Red'])

set\_2 = set(['Red', 'Green'])

print(set\_1 - set\_2)

Вы принимаете от пользователя последовательность чисел, разделённых запятой. Составьте список и кортеж с этими числами.

values = input('Введите числа через запятую: ')

ints\_as\_strings = values.split(',')

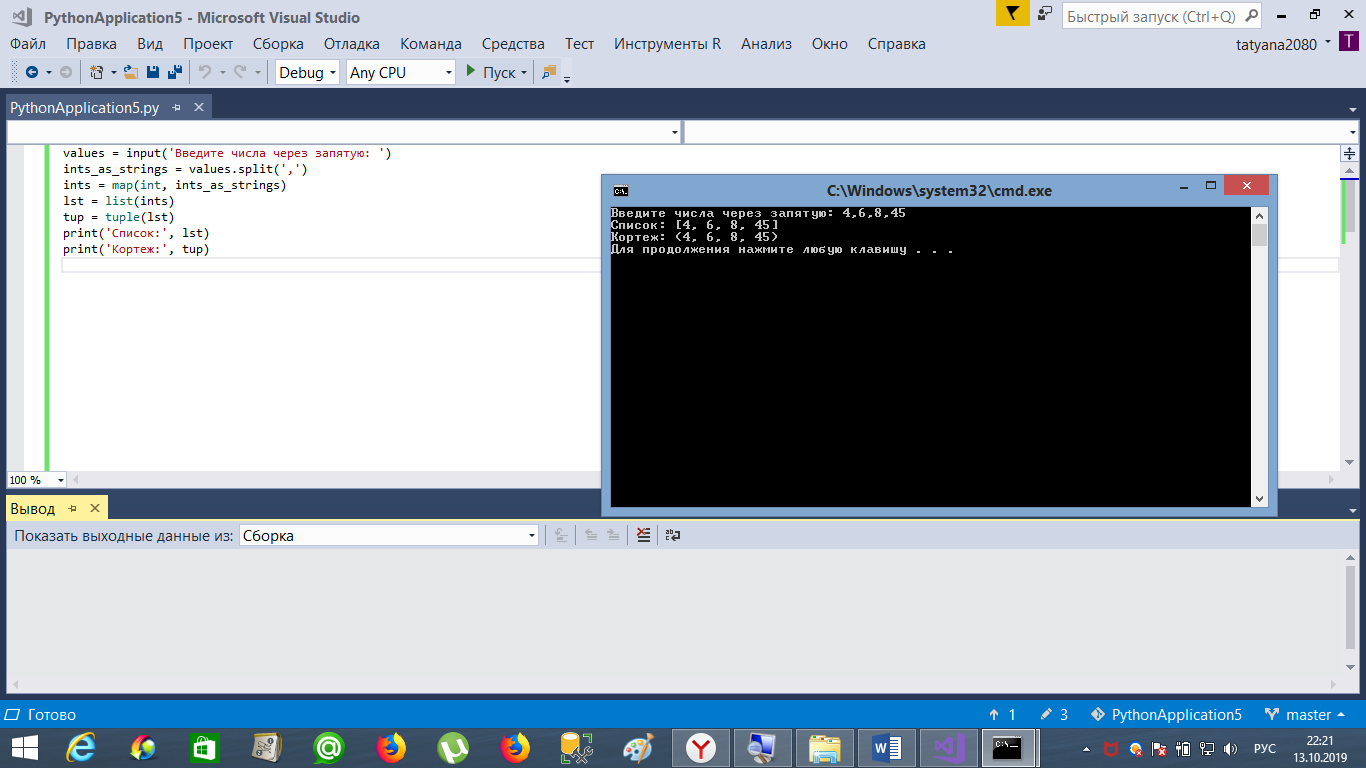
ints = map(int, ints\_as\_strings)

lst = list(ints)

tup = tuple(lst)

print('Список:', lst)

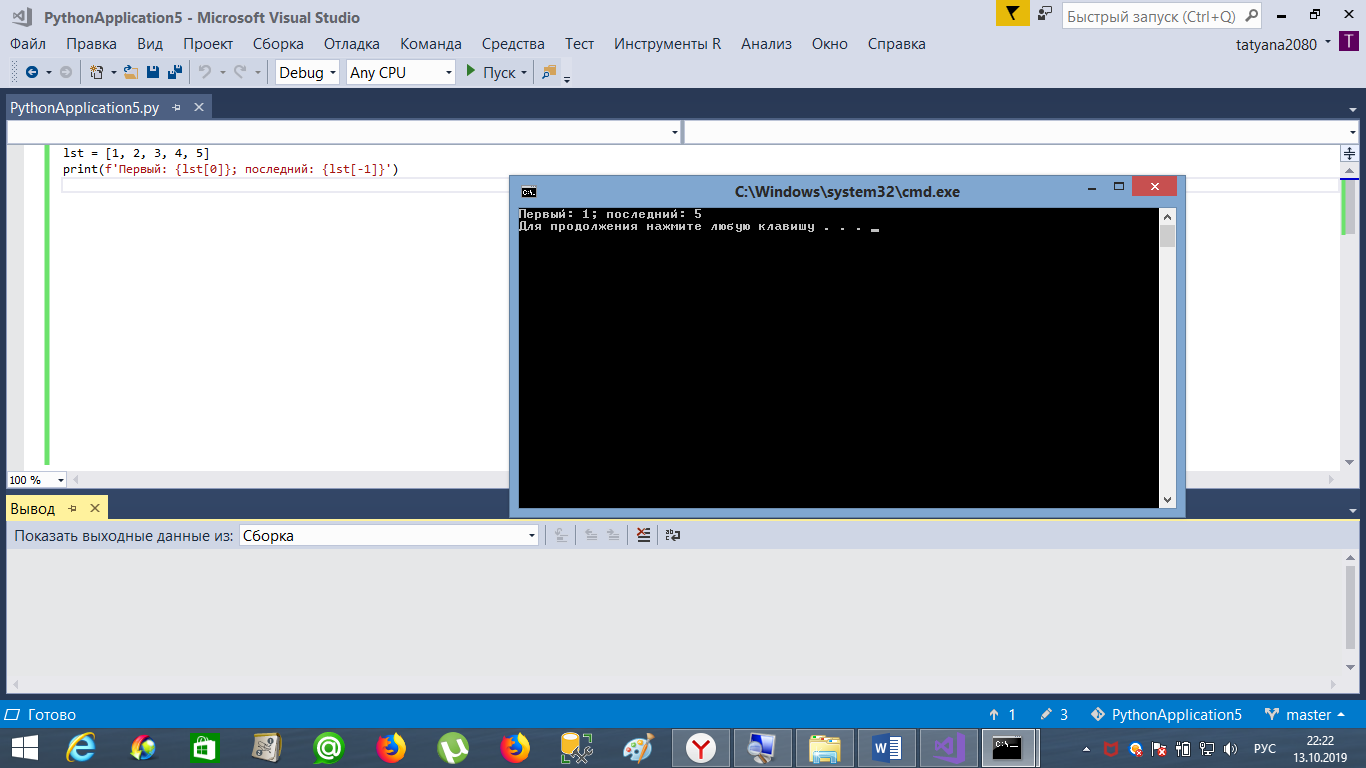
print('Кортеж:', tup)



Выведите первый и последний элемент списка.

lst = [1, 2, 3, 4, 5]

print(f'Первый: {lst[0]}; последний: {lst[-1]}')



Нужно отсортировать список, но при этом нужно сохранить знание о том, где стоял элемент до сортировки. Для этого можно применить такой трюк: заменим число x на кортеж (x, i), где i — номер элемента в списке. И после этого отсортируем.

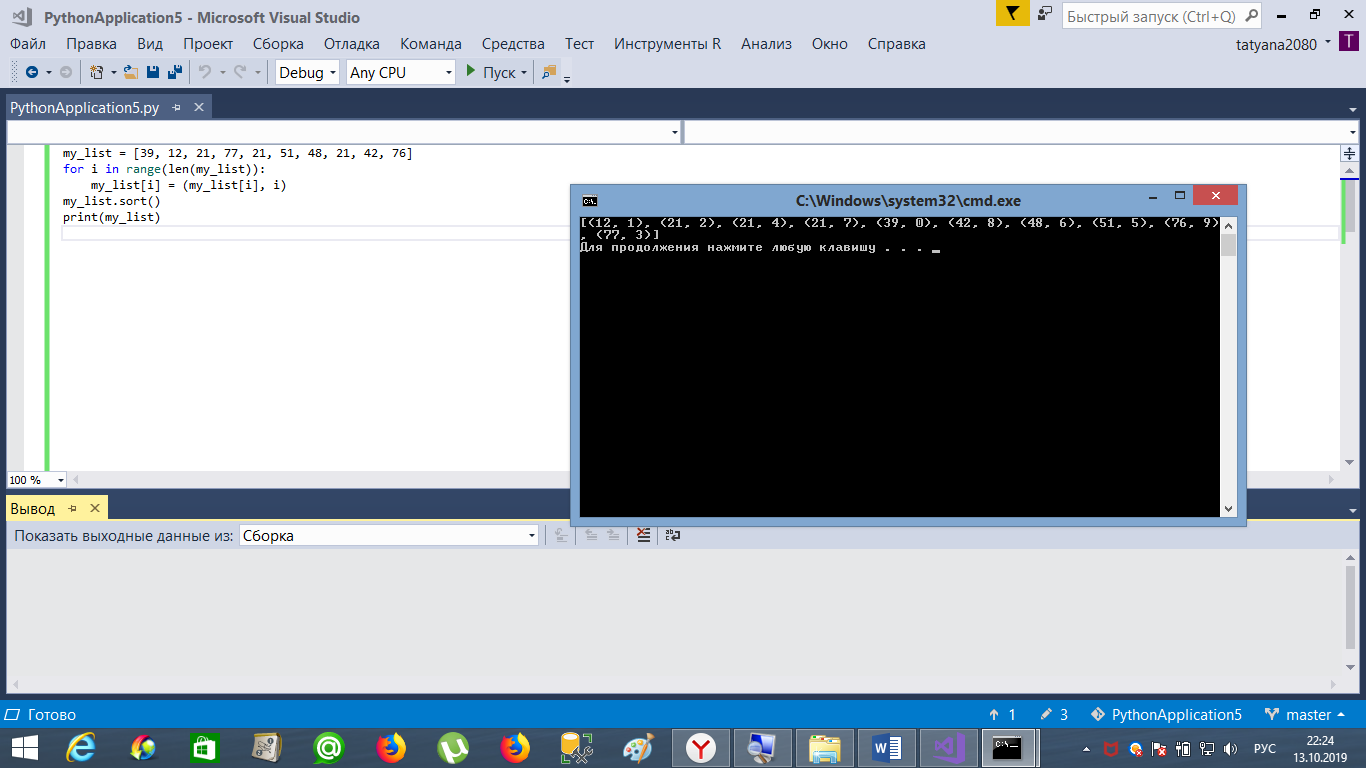
my\_list = [39, 12, 21, 77, 21, 51, 48, 21, 42, 76]

for i in range(len(my\_list)):

my\_list[i] = (my\_list[i], i)

my\_list.sort()

print(my\_list)



Благодаря такой операции мы знаем, что числа 21 стояли на 2, 4 и 7-й позиции в исходном списке, а наибольшее число 77 было на позиции 3. Всё это можно сделать короче с помощью генератора списков:

my\_list = [39, 12, 21, 77, 21, 51, 48, 21, 42, 76]

hacked\_list = sorted([(my\_list[i], i) **for** i **in** range(len(my\_list))])

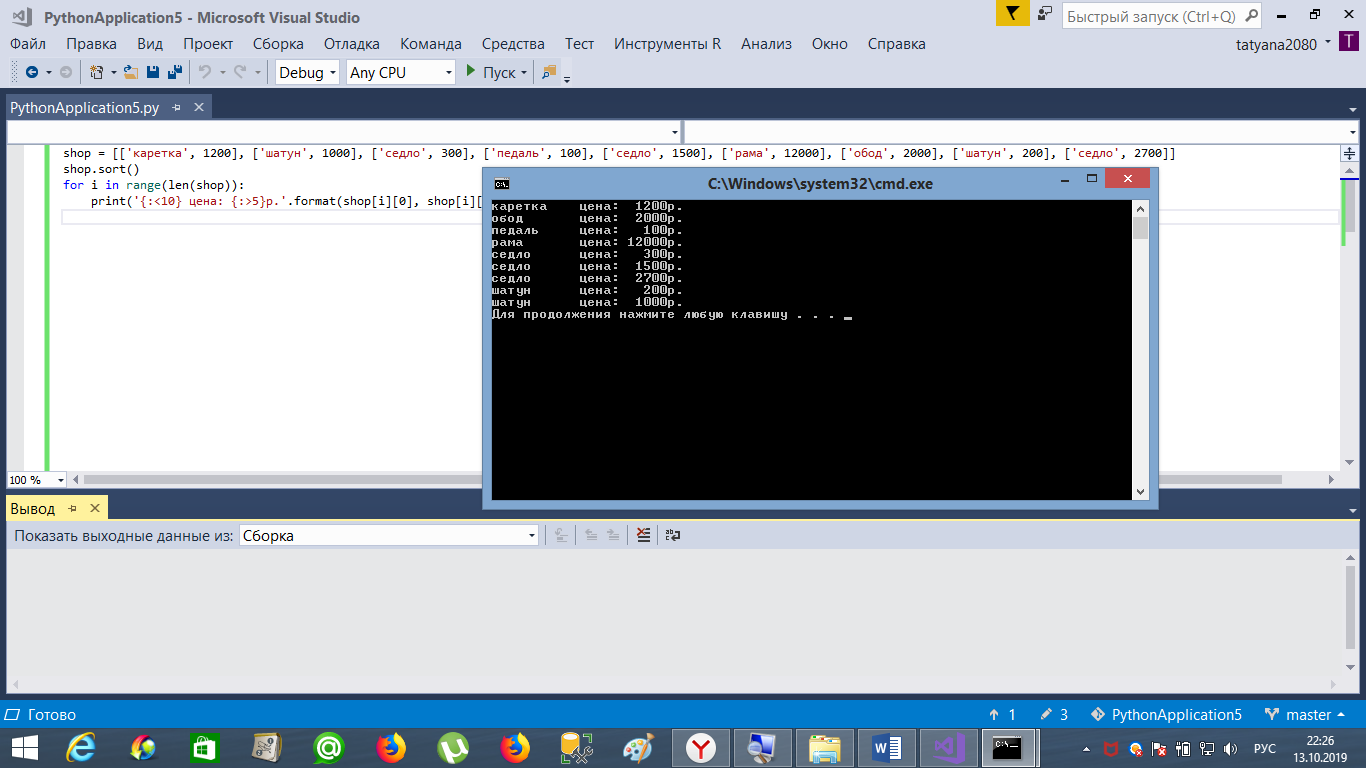
Есть список названий деталей и их стоимостей. Нам нужно отсортировать его сначала по названию деталей, а одинаковые детали по убыванию цены. Самая коротка реализация даст не совсем тот результат:

shop = [['каретка', 1200], ['шатун', 1000], ['седло', 300], ['педаль', 100], ['седло', 1500], ['рама', 12000], ['обод', 2000], ['шатун', 200], ['седло', 2700]]

shop.sort()

for i in range(len(shop)):

print('{:<10} цена: {:>5}р.'.format(shop[i][0], shop[i][1]))



Это можно исправить так:

def prepare\_item(item):

return (item[0], -item[1])

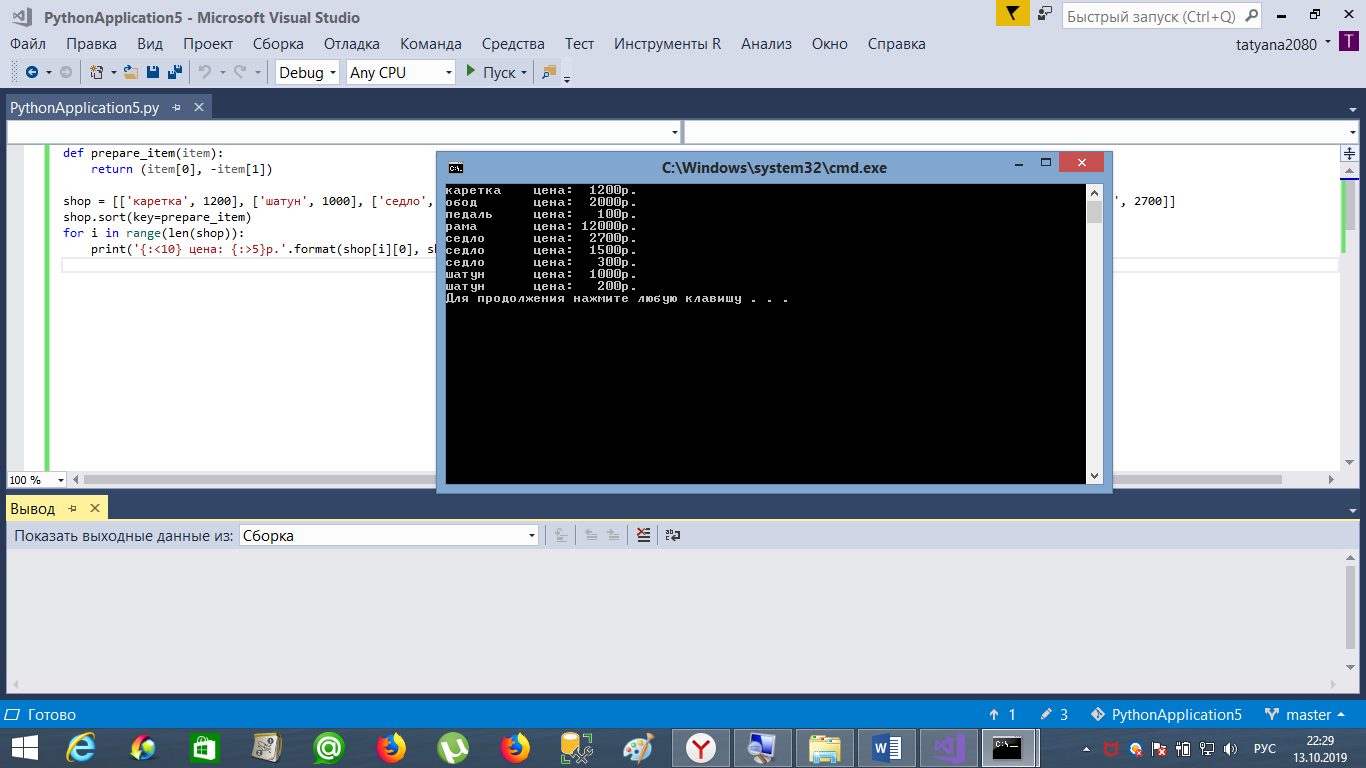
shop = [['каретка', 1200], ['шатун', 1000], ['седло', 300], ['педаль', 100], ['седло', 1500], ['рама', 12000], ['обод', 2000], ['шатун', 200], ['седло', 2700]]

shop.sort(key=prepare\_item)

for i in range(len(shop)):

print('{:<10} цена: {:>5}р.'.format(shop[i][0], shop[i][1]))

Что здесь произошло? Перед тем, как сравнивать два элемента списка к ним применялась функция prepare\_item, которая меняла знак у стоимости. В результате при одинаковов первом значении сортировка по второму происходила в обратном порядке.



Выполните обработку элементов прямоугольной матрицы A, имеющей N строк и M столбцов. Все элементы имеют целый тип. Дано целое число H. Определите, какие столбцы имеют хотя бы одно такое число, а какие не имеют.

from random import randint

n = int(input('Введите N: '))

m = int(input('Введите M: '))

h = int(input('Введите H: '))

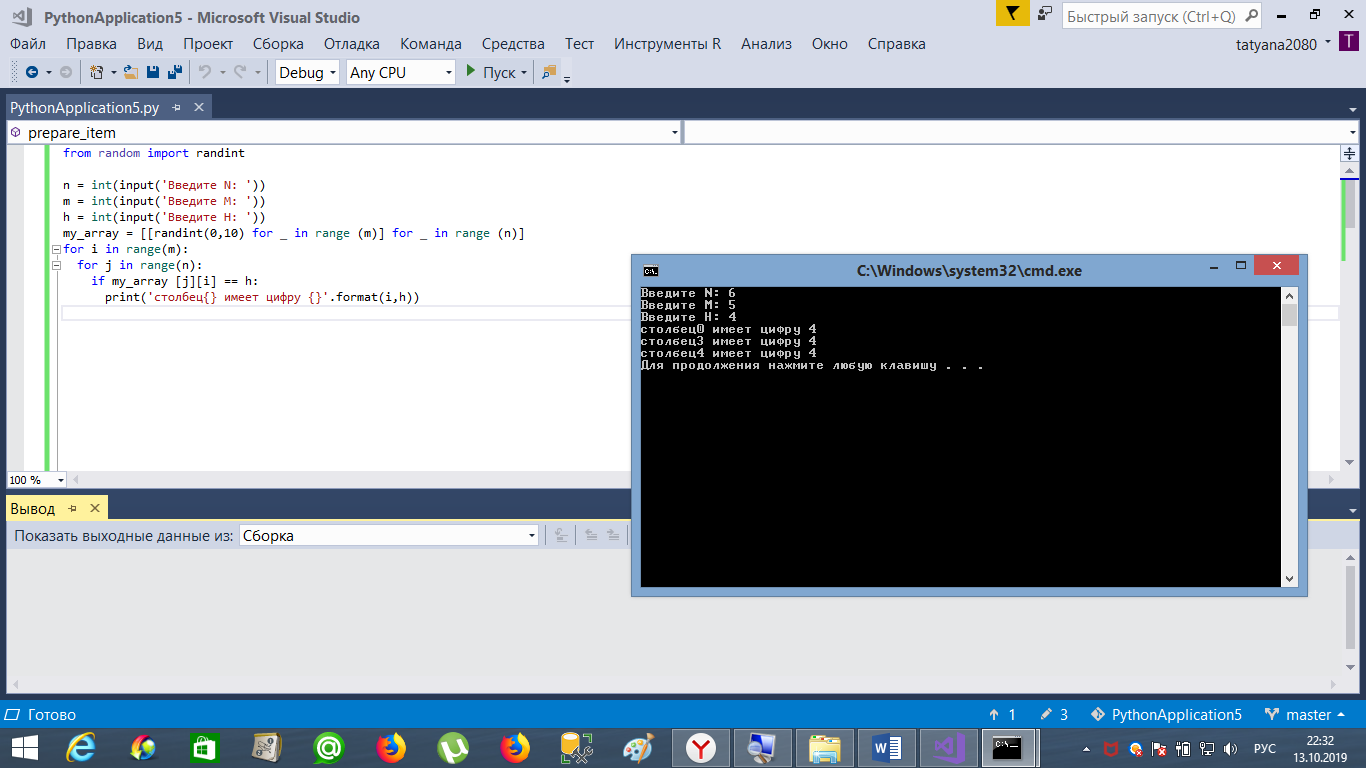
my\_array = [[randint(0,10) for \_ in range (m)] for \_ in range (n)]

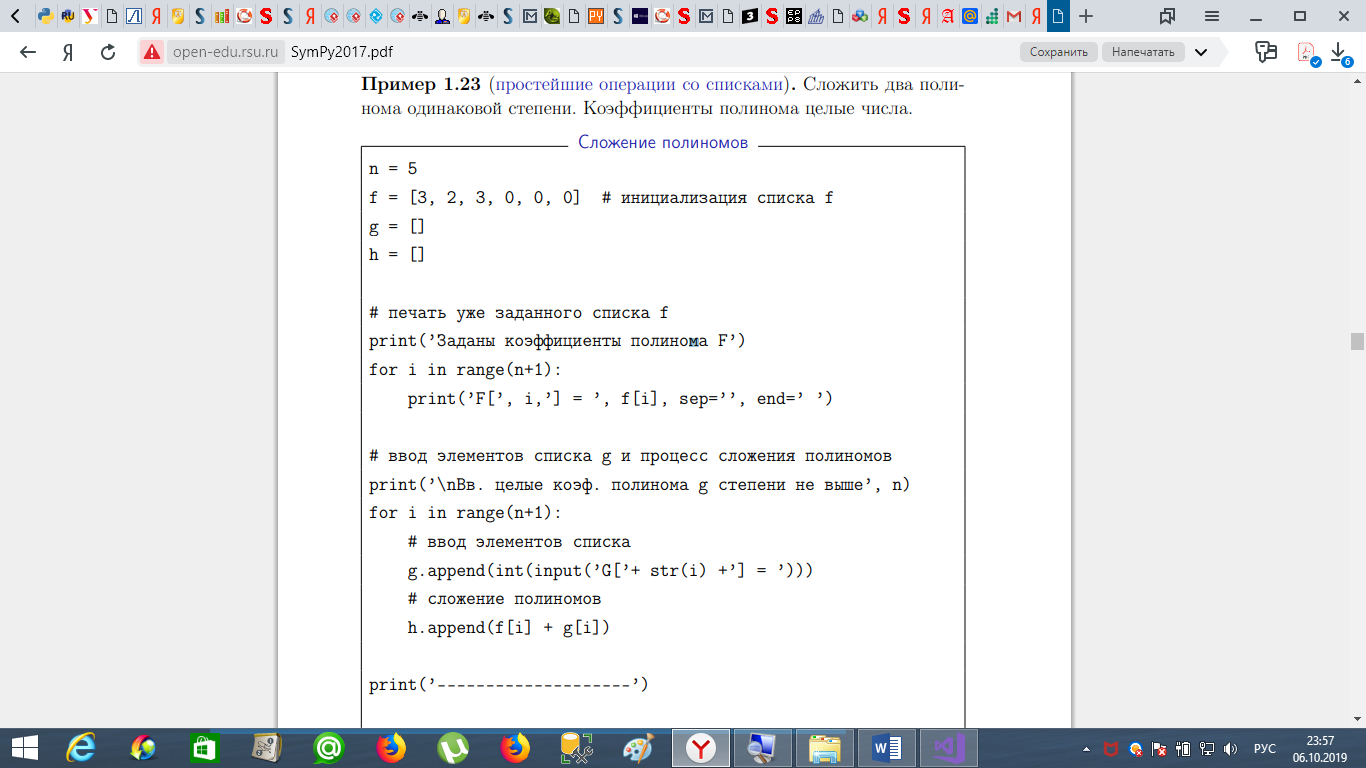
for i in range(m):

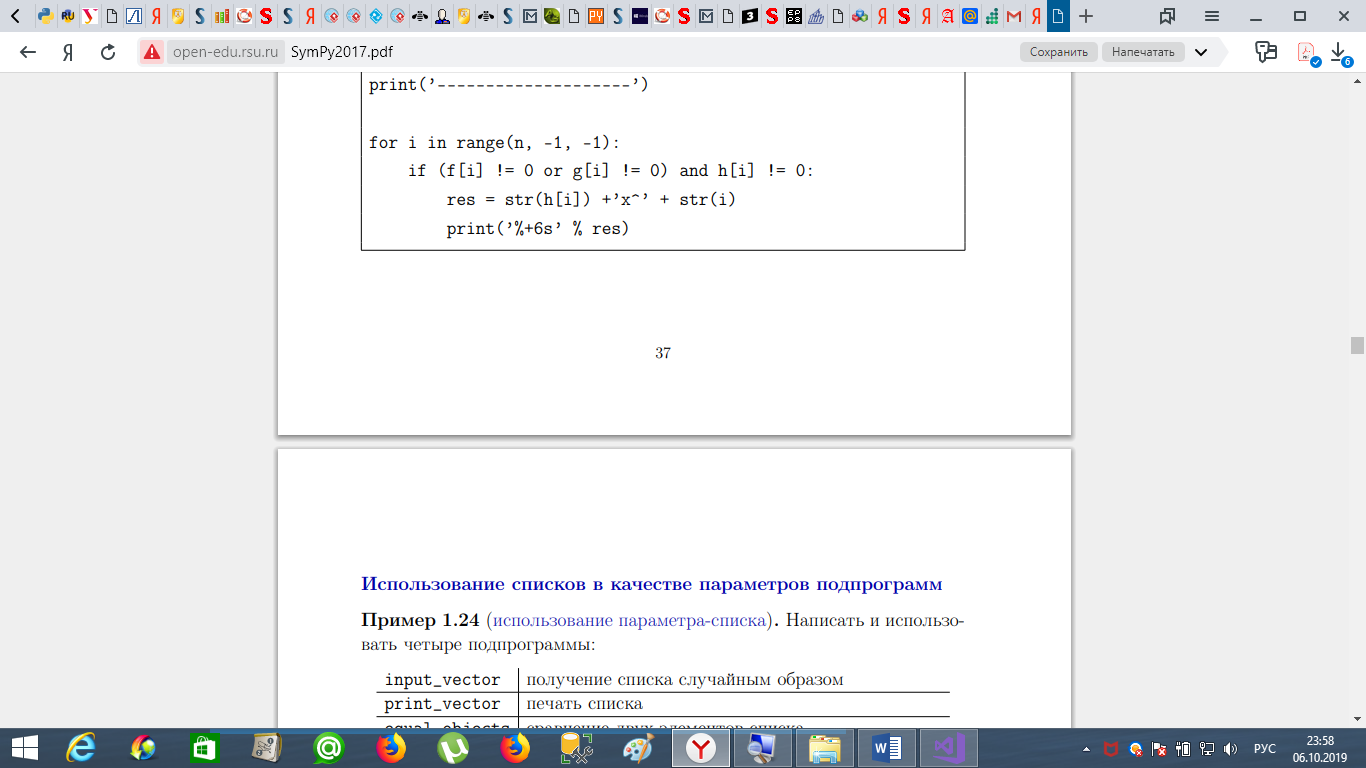
for j in range(n):

if my\_array [j][i] == h:

print('столбец{} имеет цифру {}'.format(i,h))







Поиск максимального (минимального) элемента в списке

# задаем список

mas = [3, 5, 67, -65, 34, 21]

# предположим, что максимальный элемент равен mas[0]

maximum = mas[0]

for i in range(1, len(mas)):

if mas[i] > maximum:

maximum = mas[i]

### print(maximum) Выборка элементов списка по критерию

*Задача: вывести номера элементов списка, которые больше своих соседей.*

import random

#Создадим список случайных чисел

mas = [random.randint(0, 100) for i in range(20)]

#Выведем его на экран

print(mas)

#Начиная со второго элемента массива mas[1] и до предпоследнего

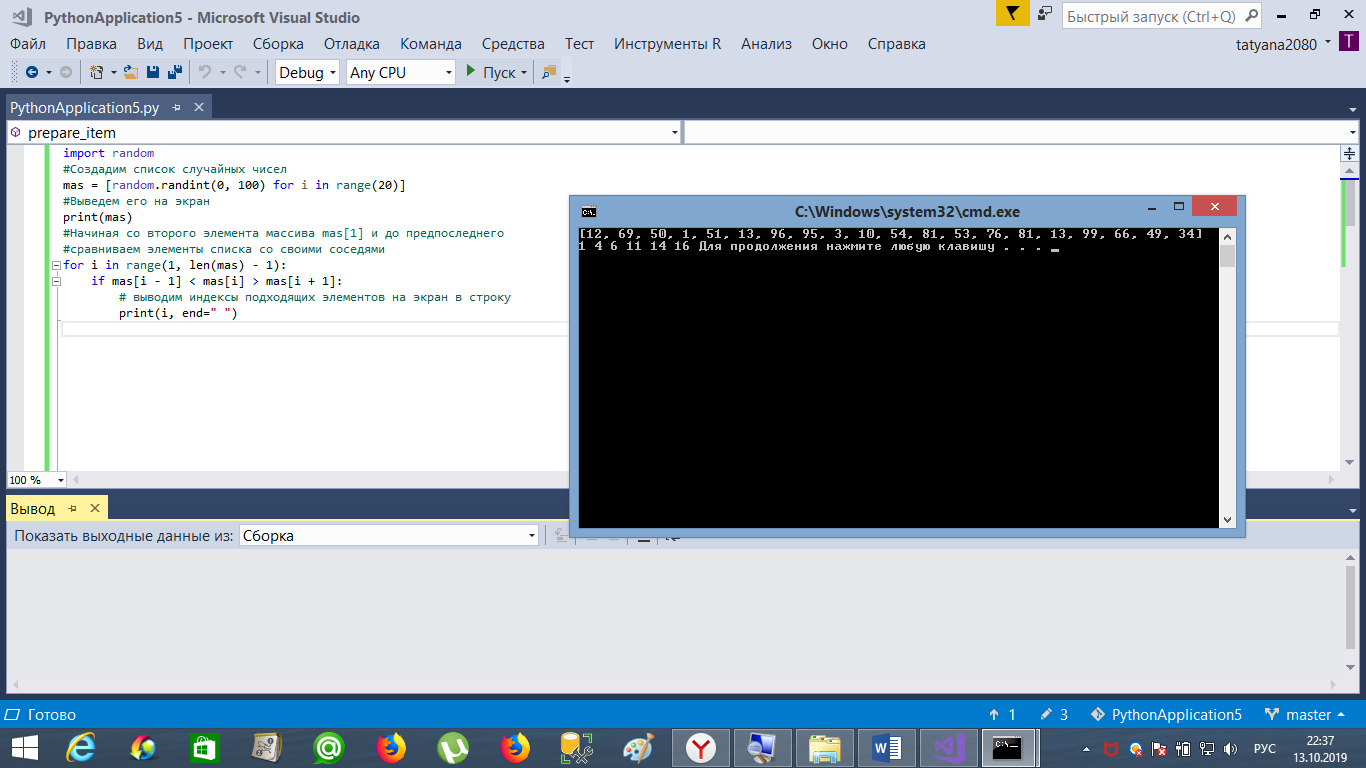
#сравниваем элементы списка со своими соседями

for i in range(1, len(mas) - 1):

if mas[i - 1] < mas[i] > mas[i + 1]:

# выводим индексы подходящих элементов на экран в строку

print(i, end=" ")



*Задача: Найти произведение элементов списка, кратных 6 и оканчивающихся на 8. Если таких элементов нет - сообщить об этом*

import random

#Создадим список случайных чисел

mas = [random.randint(0, 100) for i in range(20)]

#Выведем его на экран

print(mas)

#Так как ищем произведение - присвоим переменной значение 1

p = 1

for i in range(len(mas)):

if mas[i] % 6 == 0 and mas[i] % 10 == 8:

p \*= mas[i]

#Если произведение по-прежнему ноль - значит таких элементов нет

if p == 1:

print("Нет таких элементов")

else:

### print(p)

### 

### Реверс списка

Принцип алгоритма заключается в следующем: мы меняем местами 0-ый элемент с последним, 1-ый с предпоследним и д.т.

Итого количество таких обменов будет равно половине длины списка. Иначе элементы поменяются местами по-второму кругу и вернутся в первоначальное положение.

import random

#Создадим список случайных чисел

mas = [random.randint(0, 100) for i in range(20)]

#Выведем его на экран

print(mas)

l = len(mas)

for i in range(l//2):

mas[i], mas[l - 1 - i] = mas[l - 1 - i], mas[i]

### print(mas)

### Поиск заданного элемента в списке

# задаем список

mas = [3, 5, 67, -65, 34, 21]

# задаем искомое значение

point = 3

# вычисляем длину списка (количество элементов)

l = len(mas)

for i in range(l):

if mas[i] == point:

print("содержит")

break

else:

print("не содержит")

В цикле мы сравниваем каждый элемент списка с эталоном **point**

**Программа.** Заполнить двумерный массив размера 7х7 случайными числами и вывести его элементы

from random import randint

L = [[] \* 7 for i in range(7)]

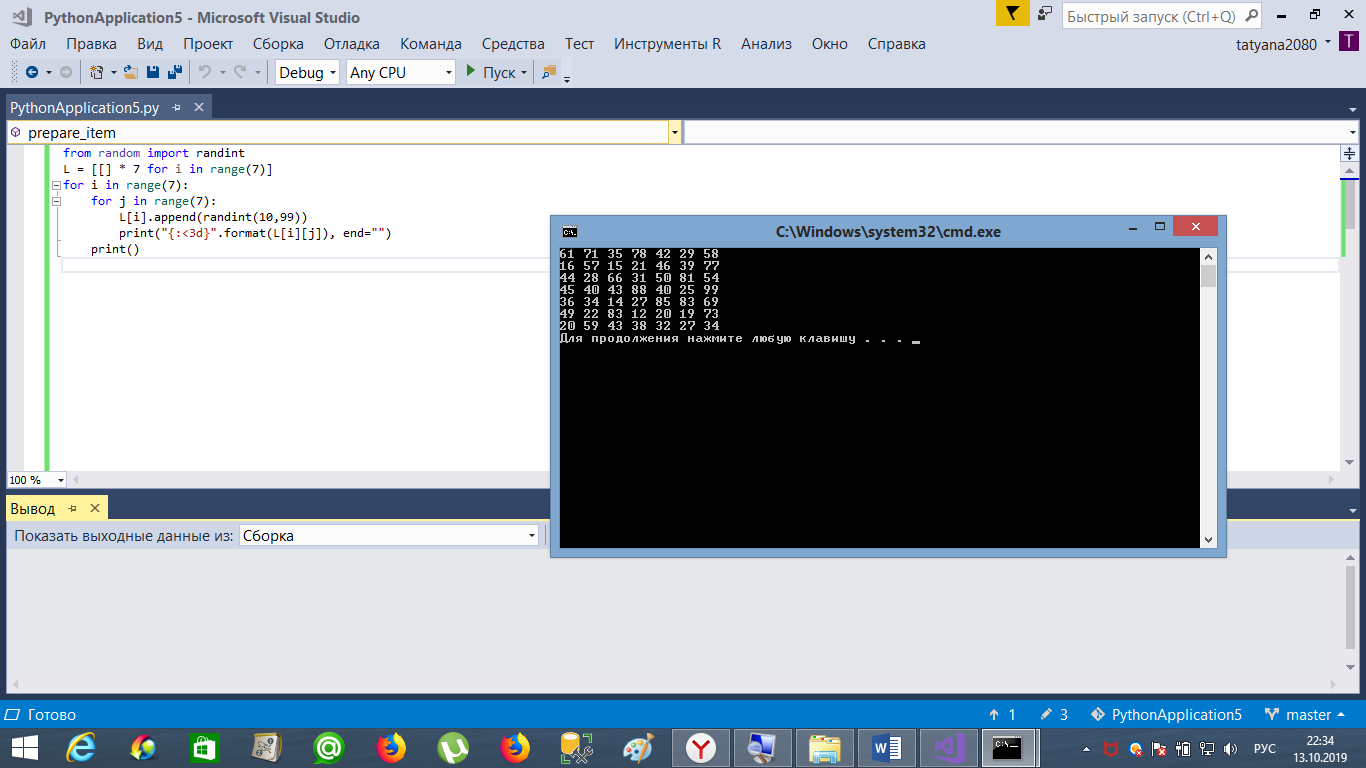
for i in range(7):

for j in range(7):

L[i].append(randint(10,99))

print("{:<3d}".format(L[i][j]), end="")

print()



Данная задача может быть решена более компактно, если использовать вложенные генераторы.

from random import randint

L = [[randint(10,99) for j in range(7)] for i in range(7)]

for i in range(7):

for j in range(7):

print("{:3d}".format(L[i][j]), end="")

#### print()

#### **Изменение списков с помощью срезов**

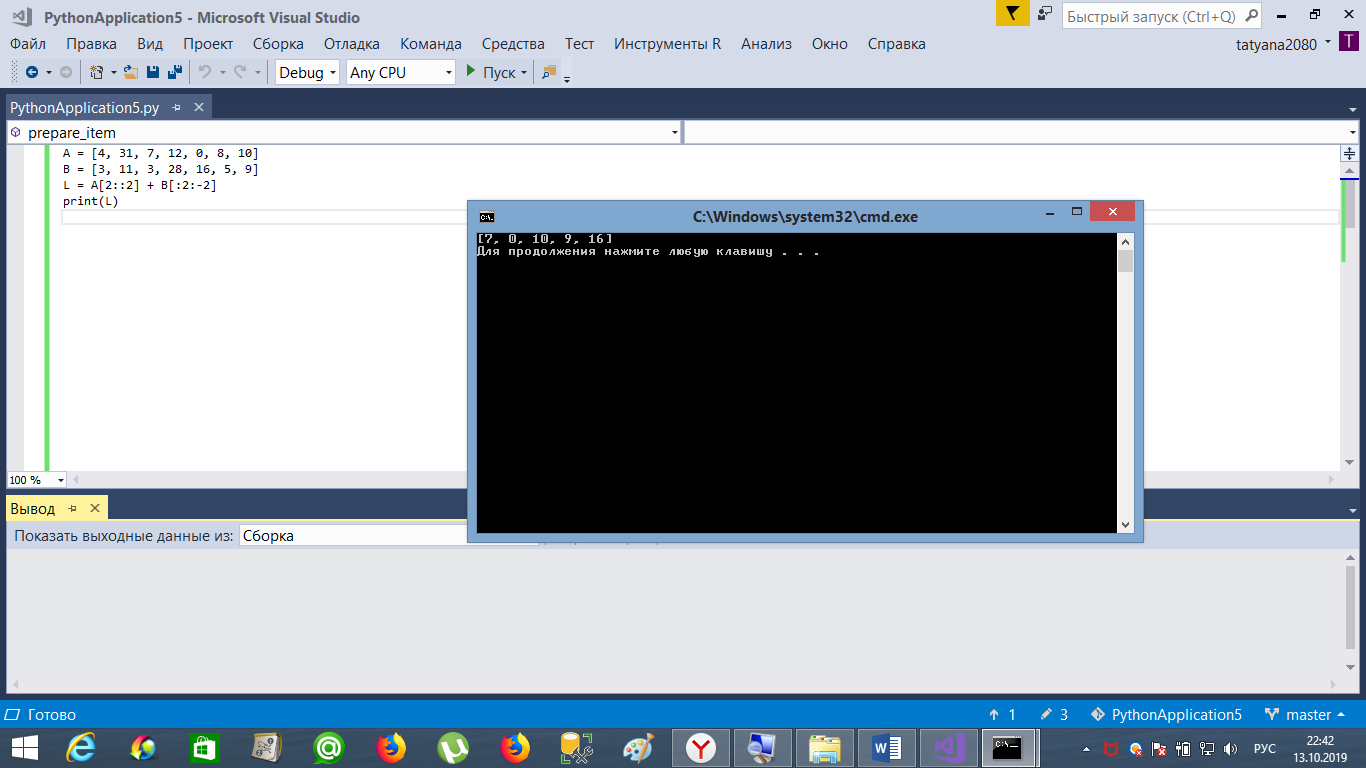
С помощью срезов можно выполнять операции изменения как исходного среза, так и любых других. Приведем примеры срезов с конкатенацией:

A = [4, 31, 7, 12, 0, 8, 10]

B = [3, 11, 3, 28, 16, 5, 9]

L = A[2::2] + B[:2:-2]

print(L)



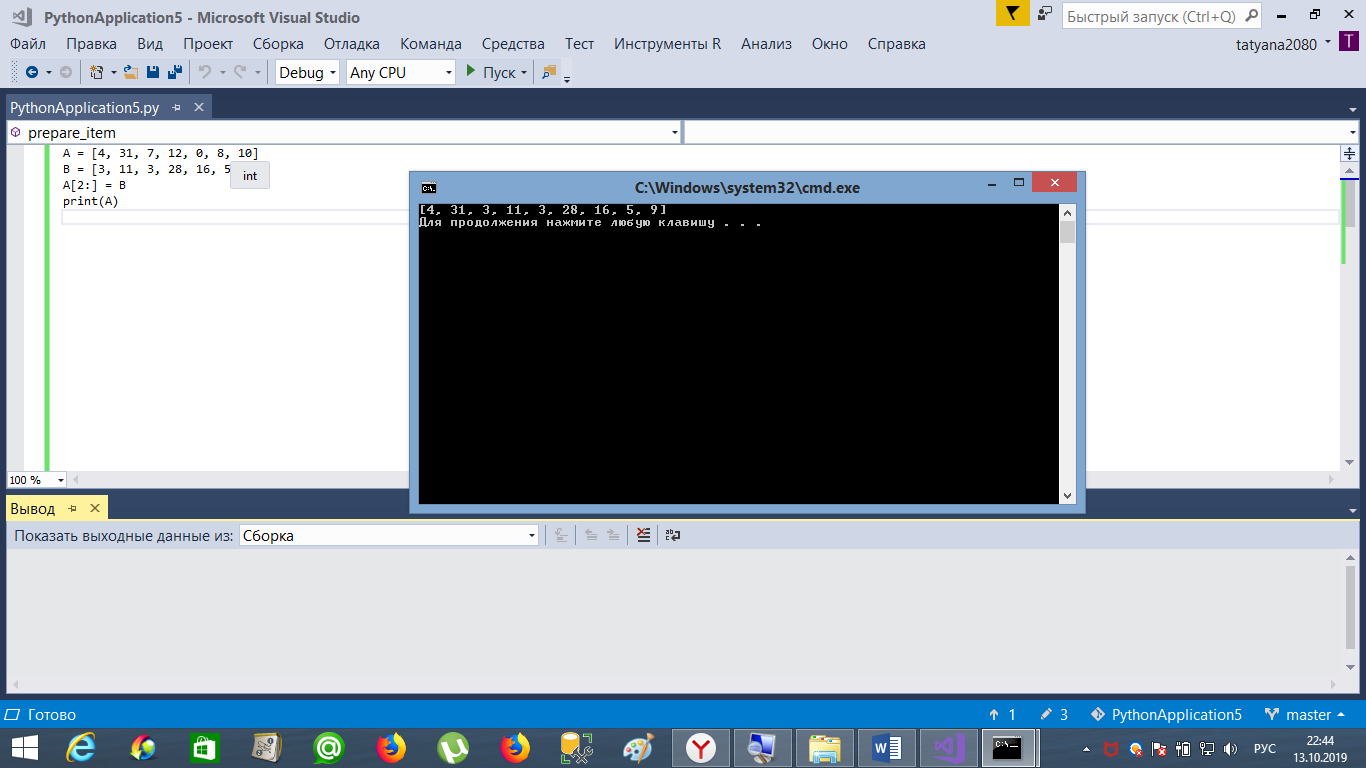
В следующей программе все элементы списка A, начиная со второго, будут заменены элементами списка B:

A = [4, 31, 7, 12, 0, 8, 10]

B = [3, 11, 3, 28, 16, 5, 9]

A[2:] = B

print(A)



С помощью операции del и срезов можно удалять в списках произвольные элементы.

A = [4, 31, 7, 12, 0, 8, 10]

B = [3, 11, 3, 28, 16, 5, 9]

del A[::2]

del B[-1:-5:-1]

print(A)

print(B)

