

# Урок 2

# Циклы. Рекурсия. Функции

Циклы — многократное повторение однотипных действий. Рекурсивный перебор. Алгоритм Евклида. Решето Эратосфена — алгоритм определения простых чисел. Использование функций.

#### Введение

#### Циклы

Понятие цикла. Виды циклов

Алгоритмическое представление цикла

Цикл с предусловием

Цикл с постусловием

Арифметический цикл (цикл с параметром)

Рекурсивный перебор

Алгоритм Евклида

Решето Эратосфена – алгоритм определения простых чисел

Функция перевода десятичного числа в двоичный формат

Практическое задание

Дополнительные материалы

Используемая литература

# Введение

На прошлом уроке вы освоили основы алгоритмизации, изучили основные способы представления алгоритмов и их виды. Мы поняли, что алгоритмы, содержащие циклические конструкции, встречаются повсеместно. Поэтому важно понимать, как именно они работают.

Рассмотрим подробнее циклические алгоритмы, рекурсию и использование функций в алгоритмах.

# Циклы

В современном программировании без циклов невозможно обойтись. Бессмысленно прописывать несколько раз подряд одно и то же действие, когда можно воспользоваться одним оператором. Вместо записи как минимум 10-20 лишних строк кода, можно написать всего лишь одну или две необходимые.

## Понятие цикла. Виды циклов

Цикл — это структура, обеспечивающая многократное повторение одного действия или их совокупности.

В основе цикла всегда лежит проверка какого-либо условия. Пока заданное для цикла условие истинно – определенные в цикле действия будут повторяться вновь и вновь. Следует понимать, что действия (операции) внутри цикла должны влиять на переменные, используемые в условии цикла. Иначе он никогда не завершится: условие всегда будет истинно.

Действия, выполняемые внутри циклы, принято называть **телом цикла**. Этап очередного выполнения действия в цикле – это **итерация цикла**.

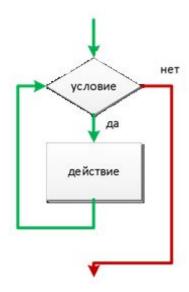
Проверка условия цикла может осуществляться до или после выполнения тела цикла. В зависимости от того, в какой момент происходит проверка условия, происходит разделение на две категории: циклы с предусловием и с постусловием.

## Алгоритмическое представление цикла

#### Цикл с предусловием

В цикле с предусловием сначала выполняется проверка условия цикла. И только в случае, если условие верно, программа переходит к выполнению тела цикла. После она снова проверяет условие, и если оно все еще истинно, программа вновь выполняет определенное действие в теле цикла. Как только условие перестает быть истинным, цикл завершается.

Принцип работы цикла с предусловием наглядно представлен на блок-схеме.



В языке Python к циклам с предусловием относятся циклы while и for.

Операторы **while** и **for** эквивалентны. Все, что можно записать с помощью одного из них, можно написать и с помощью второго. Но на практике в их использовании есть различие. Как правило, если количество повторений тела цикла известно заранее, то используется оператор **for**. Если неизвестно, то предпочтителен **while**.

#### Цикл с постусловием

В таком цикле сначала выполняются действия, образующие тело, а потом вычисляется логическое условие цикла. Если оно принимает значение истинности (true), то тело цикла выполняется повторно – и так до тех пор, пока условие не примет значение ложности (false). Основное отличие от циклов с предусловием заключается в том, что здесь тело цикла выполнится обязательно хотя бы один раз.

Принцип работы цикла с постусловием:



В языке Python цикла с постусловием, как такового, нет. Если такая конструкция необходима, то можно воспользоваться оператором цикла while в связке с оператором break (более подробно этот случай мы рассмотрим при изучении цикла while).

Иногда возникают такие задачи, когда необходимо как минимум один раз произвести вычисления и только после этого проверить, стоит ли продолжать расчеты или остановиться на полученном

значении. Например, когда необходимо посчитать какую-либо сумму или произведение с определенной точностью. В таких случаях при построении алгоритма стоит применять конструкцию цикла с постусловием.

Наиболее удачным бывает применение цикла с постусловием в задачах, где требуется выполнить действия: например, ввести число. После выполнения пользователем этих действий программа проверяет результат ввода. Если все в порядке, то вычислительный процесс идет дальше. Если нет, то программа сообщает пользователю о допущенной ошибке и возвращается на вход тех же данных. Так происходит до тех пор, пока пользователь не введет правильные данные.

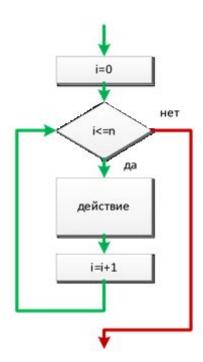
#### Арифметический цикл (цикл с параметром)

Под арифметическим циклом (циклом с параметром) подразумевается цикл с предусловием, количество итераций которого может быть вычислено или заранее известно. В арифметическом цикле обычно используется переменная (счетчик), которая определяет номер текущей итерации. На каждой итерации цикла значение счетчика должно изменяться (увеличиваться или уменьшаться), иначе цикл не завершит свою работу.

Цикл **for** – любимый оператор цикла у всех программистов. Разберемся, чем он заслужил такое расположение и каков порядок его исполнения в классическом варианте.

Выполнение оператора стартует с шагов, описанных в части «начальные действия». Эта часть **for** отрабатывается всегда, и при этом в точности — один раз. Далее вычисляется значение условия в основе цикла. Если условие имеет значение истины (true), то выполнение цикла продолжается; если false, то оператор цикла считается выполненным и управление передается следующему оператору за циклом.

Принцип работы арифметического цикла:



В языке Python к арифметическим циклам относятся for.

При написании алгоритма с использованием цикла нужно позаботиться о двух вещах:

- Мы должны быть уверены, что написанный нами цикл завершается за конечное число шагов;
- Каждый шаг должен приближать нас к решению поставленной задачи. Иными словами, мы должны гарантировать, что обработанная часть данных отвечает некоторому условию, которое остается неизменным на протяжении всего времени выполнения цикла и остается истинным после выхода из него.

## Рекурсивный перебор

Перейдем к более сложному понятию рекурсии.

Рекурсия встречается не только в различных сферах науки, но и в повседневной жизни. Она является особенно сильным средством в математических определениях.

Мощность рекурсии связана с тем, что она позволяет определить бесконечное множество объектов с помощью конечного высказывания. Точно так же бесконечные вычисления можно описать с помощью конечной рекурсивной функции, даже если она не содержит явных циклов.

Для многих программистов с самого начала сложно понять, что такое рекурсия и как она работает. Поэтому начнем с самых простых и жизненных примеров. Это сон во сне. Или динамический рисунок: художник пишет автопортрет, на котором он изображен рисующим самого себя.

На практике принцип работы рекурсии можно увидеть, если навести включенную web-камеру на экран монитора. Камера будет записывать изображение экрана компьютера, и выводить его же на этот экран. Получится что-то наподобие замкнутого цикла. В итоге вы будете наблюдать нечто вроде тоннеля.

Перейдем к реальным примерам использования и реализации рекурсии в программировании.

В современных алгоритмах ведущих ІТ-компаний применяется рекурсивный перебор: в алгоритме PageRank от Google и тИЦ от Яндекс.

Без функции в программировании не существует такого понятия, как рекурсия. Это функция, которая вызывает саму себя непосредственно (в своем теле) или косвенно (через другую функцию).

Рекурсия – так же, как цикл – это перебор. Все, что решается итеративно, можно решить рекурсивно, то есть с использованием рекурсивной функции.

Любой алгоритм, реализованный с помощью рекурсивного перебора, может быть представлен в итерационном виде и наоборот.

Возникает лишь вопрос о том, что будет работать эффективнее: алгоритм в итерационном виде или с рекурсивным перебором.

Для начала разберемся в определениях рекурсии и итерации:

- 1. Рекурсия это алгоритм обработки данных, при котором программа вызывает сама себя непосредственно или с помощью других программ;
- 2. Итерация это алгоритм обработки данных, при котором определенные действия повторяются многократно, не приводя при этом к рекурсивным вызовам программ.

Можно сделать вывод, что рекурсия и цикл взаимозаменяемы. Но выбирать реализацию следует, исходя из типа задачи и эффективности работы алгоритма.

Задача по приведению рекурсии к итерационному подходу аналогична и является симметричной.

Вывод: для каждого подхода существует свой тип задач, который определяется по конкретным требованиям.

Как и у цикла, у рекурсии должно быть условие остановки (завершения работы). Его еще называют случаем остановки (базовым случаем). Иначе рекурсия, как и цикл, будет работать вечно. Это условие является тем случаем, к которому рекурсия идет. При каждом шаге вызывается рекурсивная функция — до тех пор, пока при следующем вызове не сработает условие остановки (базовое) и не произойдет возврат к последнему вызову функции.

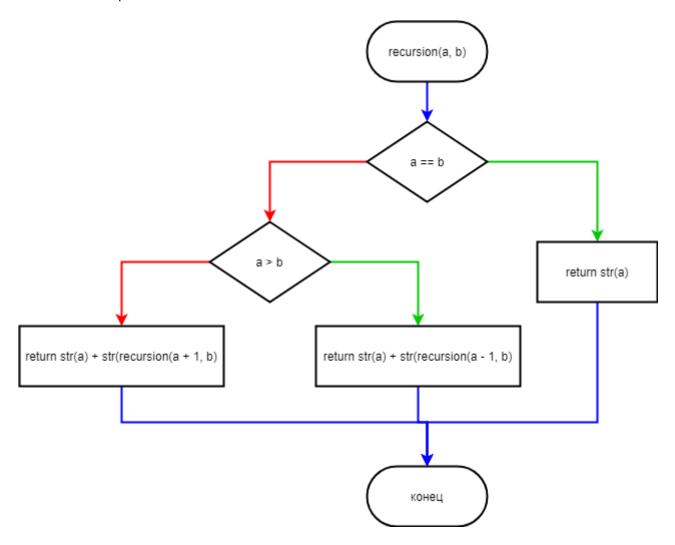
#### Основные шаги рекурсивной функции:

- Первый шаг. Необходимое условие для остановки (базовый случай);
- Второй шаг. Необходимое условие для продолжения или шаг рекурсии.

Рассмотрим рекурсию на практическом примере.

Даны два целых числа: A и B. Необходимо вывести все числа от A до B включительно, в порядке возрастания, если A < B, или в порядке убывания, если A > B.

#### Блок-схема алгоритма:



Программная реализация:

```
def recursion(a,b): # основное условие задачи

# Базовый случай

if a == b:
    return str(a)

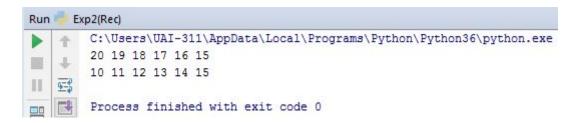
if a > b:
    # Шаг рекурсии / рекурсивное условие
    return str(a) + " " + str(recursion(a - 1, b))

else:
    # Шаг рекурсии / рекурсивное условие
    return str(a) + " " + str(recursion(a + 1, b))

print(recursion(20,15))

print(recursion(10,15))
```

#### Результат работы программы:



Основные особенности алгоритма, который допускает рекурсивное решение:

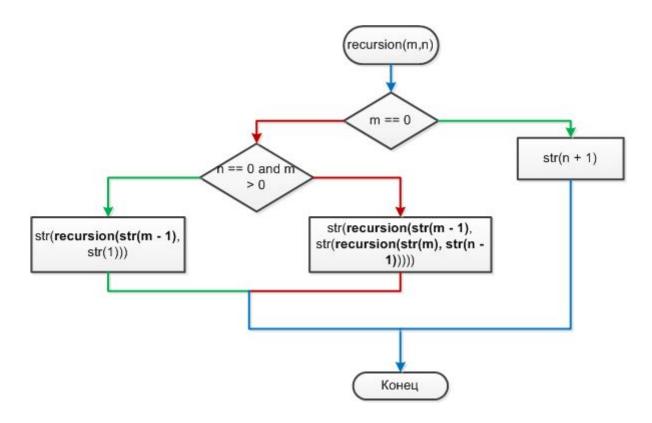
- есть последовательность действий, которую необходимо выполнить несколько раз;
- алгоритм использует в своих вычислениях данные, которые изменяются каждый раз;
- действия алгоритма изменяются только при определенном условии;
- у алгоритма есть условие завершения: рекурсивный алгоритм не бесконечен.

#### Реализация функции Аккермана с помощью рекурсии

В теории вычислимых функций важную роль играет функция Аккермана А(m,n). Она является простым примером всюду определенной вычислимой функции, которая не является просто рекурсивной. Она принимает два неотрицательных целых числа в качестве входных параметров и возвращает натуральное число.

Данная функция имеет очень глубокий уровень рекурсии, поэтому часто используется для проверки компилятора на способность оптимизировать рекурсию.

Блок-схема функции Аккермана:



#### Программная реализация:

```
def recursion(m, n):

# Базовый случай

if m == 0:
    return n + 1

# Шаг рекурсии / рекурсивное условие

elif n == 0 and m > 0:
    return recursion(m - 1, 1)

# Шаг рекурсии / рекурсивное условие

else:
    return recursion(m - 1, recursion(m, n - 1))

print(recursion(0, 2))
```

#### Результат работы программы:

```
Run Exp2(Accerman)

C:\Users\UAI-311\AppData\Local\Programs\Python\Python36\python.exe

3

Process finished with exit code 0
```

## Алгоритм Евклида

Одним из классических примеров в алгоритмизации является алгоритм Евклида. Он применяется для нахождения наибольшего общего делителя (НОД) пары целых чисел.

Наибольший общий делитель (НОД) – это число, которое делит оба числа без остатка и так же делится само на любой другой делитель данных двух чисел.

#### Общий вид алгоритма Евклида

Имеется два целых числа а и b, не равные нулю. Последовательность чисел определена тем, что каждое r(k) – остаток от деления предыдущего числа на следующее, а предпоследнее делится на последнее нацело:

$$a = b*q(0) + r(1)$$

$$b = r(1)*q(1) + r(2)$$

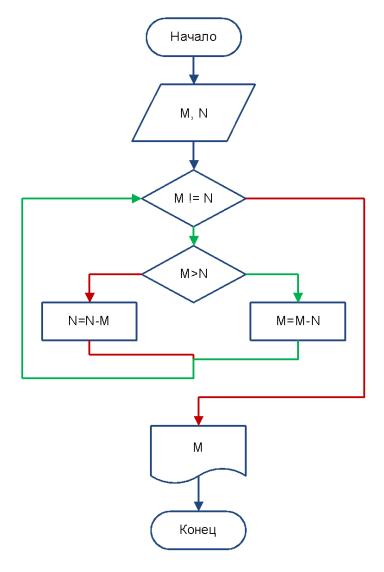
$$r(2) = r(2)*q(2) + r(3)$$

$$r(k-2) = r(k-1)*q(k-1) + r(k)$$

$$r(n-1) = r(n)*q(n)$$

Тогда НОД(a,b), наибольший общий делитель a и b, равен r(n) – последнему ненулевому члену этой последовательности.

Кроме всем известного метода, рассмотрим еще и наиболее **неэффективную версию алгоритма** – **метод последовательного вычитания**.



Описание алгоритма нахождения наибольшего общего делителя методом последовательного вычитания:

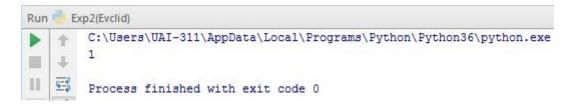
- 1. Шаг 1. Из большего числа вычитаем меньшее;
- 2. Шаг 2. Проверяем: если получается ноль, значит числа равны друг другу и являются наибольшим общим делителем. Следовательно, надо выйти из цикла;
- 3. Шаг 3. Если результат вычитания не равен нулю, то большее число меняем на результат вычитания;
- 4. Шаг 4. Переходим к шагу 1.

Реализация представленного выше алгоритма:

```
def gcd(a, b):
    while a != b:
        if a > b:
            a = a - b
        else:
            b = b - a
    print (a)

gcd(5, 8)
```

Результат работы программы:



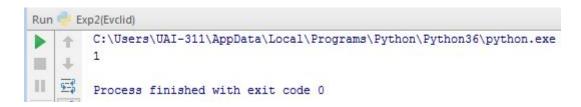
Теперь рассмотрим **наиболее эффективный алгоритм** реализации этого метода. Это **метод последовательного деления**.

Программная реализация представленного выше метода последовательного деления:

```
def gcd(a, b):
    if b == 0:
        return a
    else:
        return gcd(b, a % b)

print(gcd(5, 8))
```

Результат работы программы:

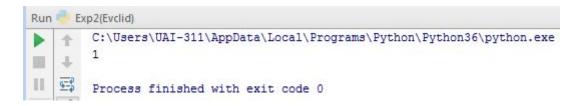


Также существует реализация этого метода в нерекурсивном виде:

```
def gcd(a, b):
    while b != 0:
        a, b = b, a % b
    return a

print(gcd(5, 8))
```

Результат работы программы:

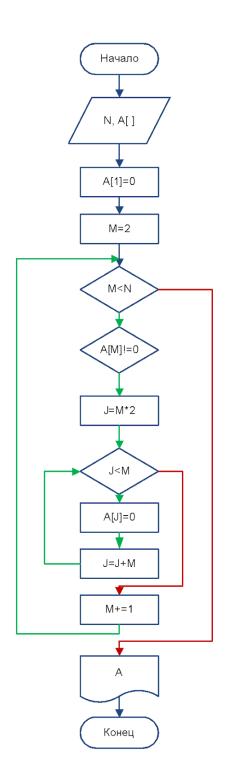


С точки зрения производительности **метод последовательного деления** предпочтительнее. В нем более эффективно работает стек вызовов рекурсии, то есть дополнительные память на хранение и время на доступ к ранее сохраненным данным.

## Решето Эратосфена – алгоритм определения простых чисел

Рассмотрим еще один классический пример алгоритмизации – «Решето Эратосфена». Это алгоритм нахождения простых чисел до заданного числа N. При его выполнении последовательно отделяются составные числа, кратные простым, начиная с 2.

Блок-схема алгоритма:



Программная реализация алгоритма «Решето Эратосфена»:

```
n = int(input("вывод простых чисел до числа ... "))
a = [0] * n  # создание массива с n количеством элементов
for i in range(n): # заполнение массива ...
   a[i] = i # значениями от 0 до n-1
# вторым элементом является единица, которую не считают простым числом
# забиваем ее нулем.
a[1] = 0
m = 2 # замена на 0 начинается с 3-го элемента (первые два уже нули)
while m < n: # перебор всех элементов до заданного числа
    if a[m] != 0: # если он не равен нулю, то
       j = m * 2 # увеличить в два раза (текущий элемент - простое число)
       while j < n:
           a[j] = 0 # заменить на 0
            j = j + m # перейти в позицию на m больше
# вывод простых чисел на экран (может быть реализован как угодно)
b = []
for i in a:
   if a[i] != 0:
       b.append(a[i])
del a
print(b)
```

Результат работы программы:

```
      Run
      Exp2(Ert)

      С:\Users\UAI-311\AppData\Local\Programs\Python\Python36\python.exe C:/Users/UAI-311/PycharmProjestson простых чисел до числа ... 90

      [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89]

      Ргосезя finished with exit code 0
```

«Решето Эратосфена» лежит в основе многих шифровальных алгоритмов, например, алгоритма шифрования RSA.

Еще один пример – работа компьютера с памятью. Между процессорной и оперативной памятью находится кэш-память. Работа с ней ведется намного быстрее, чем с оперативной памятью, но размеры ее ограничены. Например, при обработке большого массива данных процессор загружает их в кэш по частям. Сначала – первую партию данных, работает с ними, а после выгружает обратно. Эти действия повторяются до полной обработки данных.

Перенесем эту ситуацию на алгоритм решета: каждое простое число выводим из всего массива при проходе по нему. Процессор много раз будет загружать в кэш разные отрезки массива. Скорость при этом будет сильно теряться. Применение алгоритма «Решето Эратосфена» предлагает минимизировать затраты на копирование массива из одной памяти в другую.

### Функция перевода десятичного числа в двоичный формат

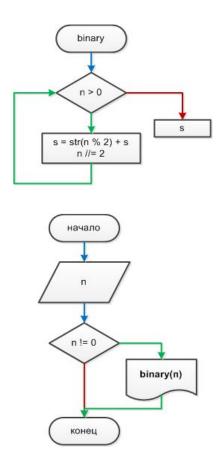
Рассмотрим алгоритм решения задачи, который может пригодиться вам для понимания работы различных систем счисления. Рассмотрим алгоритм перевода чисел десятичной системы счисления в двоичную. Для этого напишем отдельную функцию.

В основе алгоритма будет выполняться бесконечный цикл, в котором:

- 1. Шаг 1. Вводим число десятичной системы счисления;
- 2. Шаг 2. Проверяем, не ввели ли ноль. Если число не равно нулю, то вызывается функция перевода его в двоичное представление. Затем вывод результата работы функции на экран;
- 3. Шаг 3. Если введен ноль, то прерываем цикл оператором break.

В итоге наша функция должна принимать десятичное число и возвращать его двоичное представление. Возврат будем выполнять в строковом виде. Переводим число по тому же алгоритму, что и вручную: последовательно находим остаток от деления на 2 последней цифры числа, после чего само число будет сокращаться путем деления его нацело на 2. Остаток от деления будет преобразован в строковый тип и присоединен в начало формируемой строки двоичного представления числа.

Блок-схема алгоритма задачи:

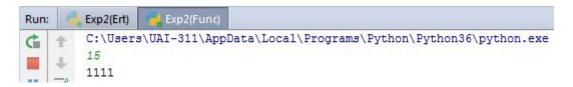


Программная реализация:

```
def binary(n):
    s = ''
    while n > 0:
        s = str(n % 2) + s
        n //= 2
    return s

while True:
    n = int(input())
    if n != 0:
        print(binary(n))
    else:
        break
```

#### Результат работы программы:



#### Подведем итоги:

- 1. Существует три варианта реализации циклических алгоритмов;
- 2. Рекурсия это хорошая вещь, но всегда надо анализировать задачу и выбирать наиболее эффективный алгоритм реализации;
- 3. Мы рассмотрели классические примеры функции в алгоритмизации, которые можно реализовать как рекурсивным, так итеративным перебором.

# Практическое задание

- 1. Написать программу, которая будет складывать, вычитать, умножать или делить два числа. Числа и знак операции вводятся пользователем. После выполнения вычисления программа не должна завершаться, а должна запрашивать новые данные для вычислений. Завершение программы должно выполняться при вводе символа '0' в качестве знака операции. Если пользователь вводит неверный знак (не '0', '+', '-', '\*', '/'), то программа должна сообщать ему об ошибке и снова запрашивать знак операции. Также сообщать пользователю о невозможности деления на ноль, если он ввел 0 в качестве делителя.
- 2. Посчитать четные и нечетные цифры введенного натурального числа. Например, если введено число 34560, то у него 3 четные цифры (4, 6 и 0) и 2 нечетные (3 и 5).
- 3. Сформировать из введенного числа обратное по порядку входящих в него цифр и вывести на экран. Например, если введено число 3486, то надо вывести число 6843.
- 4. Найти сумму n элементов следующего ряда чисел: 1 -0.5 0.25 -0.125 ...Количество элементов (n) вводится с клавиатуры.
- 5. Вывести на экран коды и символы таблицы ASCII, начиная с символа под номером 32 и заканчивая 127-м включительно. Вывод выполнить в табличной форме: по десять пар "код-символ" в каждой строке.
- 6. В программе генерируется случайное целое число от 0 до 100. Пользователь должен его отгадать не более чем за 10 попыток. После каждой неудачной попытки должно сообщаться

- больше или меньше введенное пользователем число, чем то, что загадано. Если за 10 попыток число не отгадано, то вывести загаданное число.
- 7. Напишите программу, доказывающую или проверяющую, что для множества натуральных чисел выполняется равенство: 1+2+...+n = n(n+1)/2, где n любое натуральное число.
- 8. Посчитать, сколько раз встречается определенная цифра в введенной последовательности чисел. Количество вводимых чисел и цифра, которую необходимо посчитать, задаются вводом с клавиатуры.
- 9. Среди натуральных чисел, которые были введены, найти наибольшее по сумме цифр. Вывести на экран это число и сумму его цифр.

#### Примечание ко всем задачам:

1. Постарайтесь решить задачи без использования массивов. Им будет посвящён следующий урок.

# Дополнительные материалы

1. <a href="http://www.intuit.ru/studies/courses/10/320/info">http://www.intuit.ru/studies/courses/10/320/info</a>

# Используемая литература

- 1. <a href="https://www.python.org">https://www.python.org</a>
- 2. <a href="http://www.intuit.ru/studies/courses/10/320/info">http://www.intuit.ru/studies/courses/10/320/info</a>
- 3. Марк Лутц. Изучаем Python, 4-е издание.