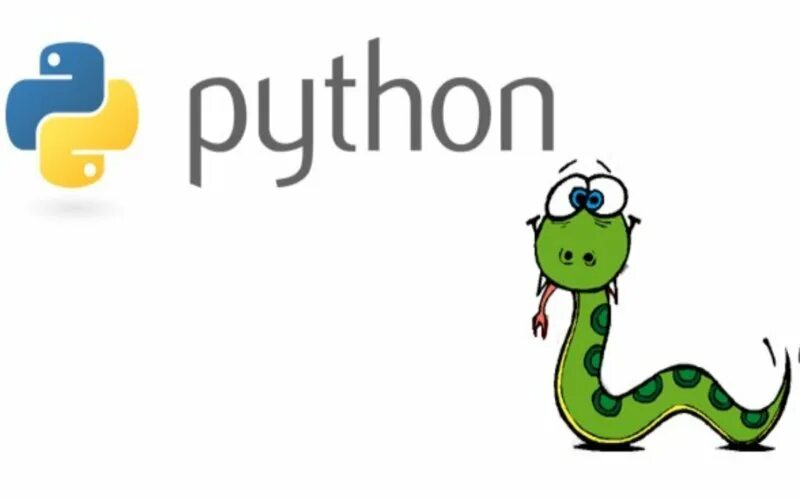
**ЯЗЫК ПРОГРАММИРОВАНИЯ**



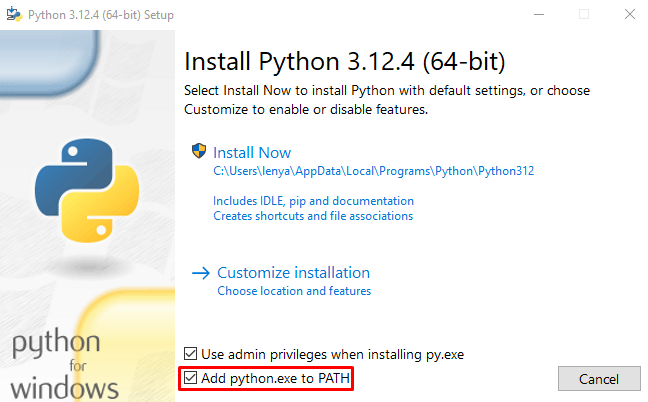
Установка Python на Windows

Python — это бесплатный и открытый язык программирования. Для его установки на Windows следуйте инструкции:

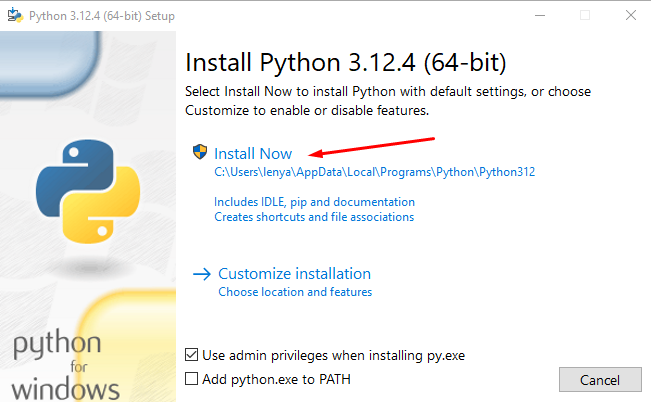
1. Перейдите по <https://www.python.org/downloads/> на официальный сайт в раздел **«Downloads»**
2. Скачайте актуальную версию Python, кликнув по жёлтой кнопке **«Download»**:



1. Запустите загрузившийся установщик
2. Установите галочку в чекбоксе **«Add Python to path»**:



1. Нажмите на кнопку **«Install Now»**:



1. Дождитесь завершения установки

Переменные и работа с данными

В процессе программирования нам постоянно приходится взаимодействовать с данными: обрабатывать их, производить вычисления и выполнять различные действия. Но прежде данные нужно где-то сохранить, и в этом нам помогут переменные.

## Переменные

Переменная — это именованная ссылка, ведущая на определенную ячейку оперативной памяти компьютера, в которой и хранится соответствующий объект. При создании переменной в неё записывается ссылка на объект, который и хранится где-то глубоко в оперативной памяти. Для любителей ассоциаций переменная в Python больше похожа на номерок в гардеробе, по значению которого определяется место хранения объекта.

Инициализация переменной

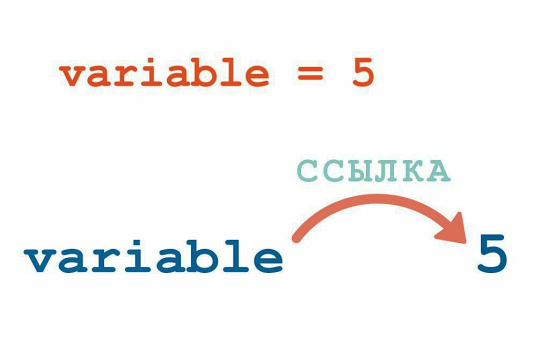
Для работы с переменной её сначала нужно объявить. Как мы уже выяснили, переменная имеет имя, за которым скрывается ссылка, ведущая на конкретный объект, потому для создания переменной необходимо придумать имя переменной и присвоить ей значение с помощью оператора присваивания, который обозначается в Python знаком =.

Общая структура инициализации (создания) переменной такова:

имя переменной = значение

Создадим переменную с именем variable и значением 5, являющимся целым числом:

variable = 5 # связываем переменную variable с числом 5



Текст после знака # называется комментарием. Комментарии в языке программирования — это поясняющие строки в программном коде, которые позволяют понять смысл написанного. Они пишутся для людей, но игнорируются компиляторами и интерпретаторами.

Имя переменной

Наименование переменной должно соответствовать определенным правилам:

1. Имя переменной может состоять только из латинских букв, цифр и знаков \_ нижнего подчеркивания
2. Имя переменной не может начинаться с цифр
3. Имя переменной не может совпадать с названием служебных слов, определенных в Python

*Примечание.* Служебных слов в Python очень много (например, in, for, while и т.д.), заучивать их отдельно не нужно, поскольку они запомнятся в процессе изучения языка.

Рекомендуется также по возможности давать переменным осмысленные наименования, что упрощает читаемость и отладку кода в будущем, особенно в работе с большими проектами. К хорошему тону следует приучаться заранее.

Рассмотрим корректные примеры инициализации переменной:

* num = 7
* age = 15
* string = "world" (здесь значением выступает строка, потому она обособляется кавычками)

И некорректные:

* 9num = 9 (имя начинается с цифры)
* variable name = "name" (имя содержит пробелы)
* in = 20 (имя совпадает со служебным словом in — о нем мы поговорим позже)
* "Leonid" = name (нарушена структура переменной: имя переменной = значение)

*Примечание:* Python — регистрозависимый язык, потому имена var и Var, например, считаются различными.

## Множественное и каскадное присваивание

В некоторых случаях нужно определить сразу несколько переменных, для чего удобно использовать *множественное присваивание*, которое поможет инициализировать произвольное количество переменных в одну строку. Предположим, нужно определить точку (𝑥,𝑦,𝑧) на трёхмерной оси координат, а 𝑥, 𝑦 и 𝑧 принимают различные значения. Тогда можно объявить каждую переменную на отдельной строке:

x = 5

y = 7

z = 3

А с помощью множественного присваивания можно переписать приведенный выше код так:

x, y, z = 5, 7, 3

При множественном присваивании имена переменных перечисляются через запятую, а после = указываются соответствующие им значения, причем количество имен должно строго соответствовать количеству значений.

Иногда возникает необходимость создать несколько переменных с одинаковыми значениями, тогда удобно использовать каскадное присваивание. Например, если x, y и z принимают значения, равные единице, можно объявить каждую переменную на отдельной строке:

x = 1

y = 1

z = 1

Благодаря каскадному присваиванию можно переписать этот код так:

x = y = z = 1

Объем кода сокращается, что делает его более компактным и удобочитаемым.

Ввод и вывод данных

Мы познакомились с переменными и изучили азы работы с ними. В нём данные мы задавали сами, но зачастую они заранее нам неизвестны и подаются в процессе выполнения программы, для такого ввода нам понадобится функция input(). А выводить данные на экран нам будет помогать функция print().

## Функция print()

Функция print() используется для вывода данных на экран. Она принимает на вход произвольное количество аргументов, которые передаются в скобках и перечисляются через запятую.

Примечание. Аргументы — это значения, которые передаются в функцию при её вызове (они указываются внутри скобок).

Например, этот код:

print(1048576)

выводит число

1048576

При выводе строк важно не забывать обрамлять их в двойные или одинарные кавычки.

Данный код:

print("One Shot")

выводит строку

One Shot

Обратите внимание на то, что при выводе кавычки не отображаются.

Помимо чисел и строк аргументами могут выступать и переменные.

Приведенный ниже код:

a = 10

b = 40

result = a + b

print(a)

print(b)

print(result)

выводит

10

40

50

## Вычисления внутри функции print()

Операции между переменными, числами и строками можно выполнять прямо внутри print() во время вывода.

Данный код:

x = 15

y = 5

print(x + y)

print(x - y)

print(50 - 20)

print(100 \* x)

print("Hello" + ", " + "World!")

выводит

20

10

30

1500

Hello, World!

## Множественный вывод

Поскольку функция print() принимает произвольное количество аргументов, она позволяет выводить сразу несколько элементов. Для этого нужно перечислить в скобках через запятую все аргументы, которые нужно вывести.

Например, этот код:

name = "Leonid"

surname = "Shastin"

print(name, surname)

выводит

Leonid Shastin

А этот код:

a = 50

b = 250

print(a, "+", b, "=", a + b)

выводит

50 + 250 = 300

Можно использовать вычисления внутри print() и множественный вывод одновременно, чтобы печатать данные на экран в наиболее удобном формате.

В будущих уроках мы иногда будем использовать термин «инструкция». Инструкция языка программирования — это команда или набор команд на языке программирования, посредством которой компьютеру сообщается о необходимости выполнения одного определенного шага по обработке порции информации. В коде выше a = 50, b = 250 и использование функции print() — три отдельные инструкции. Другими словами, любая программа на языке Python — набор инструкций.

## Параметры sep и end

Обратите внимание на то, что в примерах, приведенных выше, значения при множественном выводе печатаются через один пробел, а код, содержащий несколько команд print(), выводит каждое последующее значение с новой строки (в конце вывода внутри каждой команды происходит переход на новую строку). На самом деле функция print() помимо аргументов на вывод принимает ещё необязательные именованные параметры sep и end.

## Параметр sep

Параметр sep задает разделитель между значениями на выводе, по умолчанию он равен одному пробелу.

Так, эти две программы эквивалентны:

print(16, 32)

print(16, 32, sep=' ')

Они выводят

16 32

и разделяют 16 и 32 с помощью одного пробела.

Символ-разделитель можно поменять, для этого необходимо в функции print() после перечисления всех аргументов обратиться к параметру sep и присвоить ему нужное значение в виде строки.

Например, этот код:

x = 20

y = 30

print(x, y, sep="+")

print(x, y, sep="")

print(x, y, sep="-\_-")

print(x, y, 50, 70, sep="|")

выводит

20+30

2030

20-\_-30

20|30|50|70

## Параметр end

Параметр end задает значение, которое записывается в конец вывода. По умолчанию он равен символу перевода строки \n, именно поэтому при каждом новом print() вывод происходит уже с новой строки. Такое поведение можно изменить, для этого нужно в функции print() в конце обратиться к параметру end и присвоить ему нужное значение в виде строки.

Так, эти две программы эквивалентны:

print(100, end='\n')

print(200)

print(100)

print(200)

Они выводят указанные числа на разных строках

100

200

Данный код:

print(50, end=" ")

print(60)

print(70)

выводит

50 60

70

Перехода на новую строку после первого print() не происходит, поскольку стандартное поведение было изменено. Вместо пробела в качестве параметра end можно передавать любое другое строковое значение.

Существует также возможность комбинирования параметров sep и end внутри одного print().

Приведенный ниже код:

print(10, 20, 30, sep="|", end="!")

print(40, 50)

выводит

10|20|30!40 50

Арифметические операции над числами

В этом уроке мы рассмотрим все остальные арифметические операции, а также затронем вещественный тип данных, порядок выполнения операций в Python и скобочные выражения.

## Основные арифметические операции

В таблице ниже представлены 4 основных математических операции (+, -, \*, /).

| Оператор | Назначение |
| --- | --- |
| + | сложение |
| **-** | вычитание |
| \* | умножение |
| / | деление |

Приведённый ниже код:

x = 24

y = 5

print(x + y)

print(x - y)

print(x \* y)

print(x / y)

выводит

29

19

120

4.8

Обратите внимание на то, что в результате деления 24 на 5 получается число 4.8, содержащее дробную часть.

## Тип данных float

Числа с плавающей точкой (они же вещественные) принадлежат в Python к типу float. Эти числа представлены последовательностью десятичных цифр и точкой, являющейся разделителем между целой и дробной частью.

Примеры вещественных чисел: , , 4.84.8, π.

Приведенный ниже код:

result = 10 / 2

print(result)

print(type(result))

выводит

5.0

<class 'float'>

Отсюда следует важный вывод: при использовании операции деления (/) в Python результатом всегда будет являться число типа float, причем независимо от того, делится одно число на другое без остатка или нет.

Над вещественными числами, как и над целыми, можно производить различные арифметические операции.

Этот код:

a = 5.0

b = 2.5

print(a + b)

print(a \* b)

print(a - b)

print(a / b)

выводит

7.5

12.5

2.5

2.0

При выполнении различных операций с числами типа float на выходе всегда получается число того же типа.

Примечание. С помощью функции int() можно преобразовать вещественное число к целочисленному типу — в этом случае дробная часть будет отброшена. Например, int(52.6) вернёт 52.

## Дополнительные арифметические операции

Существует также 3 дополнительных операции (//, %, \*\*).

|  |  |
| --- | --- |
| **Оператор** | **Назначение** |
| // | целочисленное деление |
| % | остаток от деления |
| \*\* | возведение в степень |

## Целочисленное деление

Оператор целочисленного деления (//) отличается от обычного оператора деления (/) тем, что он округляет получившийся результат до ближайшего меньшего целого числа.

Приведенный ниже код:

print(15 // 2)

print(10 // 2)

print(3 // 7)

print(235 // 10)

выводит

7

5

0

23

В примерах выше мы работали с положительными числами. При целочисленном делении отрицательных чисел округление тоже осуществляется до ближайшего меньшего целого, а потому -8 // 3 вернёт −3−3.

Данный код:

print(-10 // 4)

print(-10 // -3)

print(15 // -6)

print(20 // -3)

выводит

-3

3

-3

-7

## Деление с остатком

Оператор деления с остатком (%) возвращает остаток от деления одного числа на другое. Если число делится нацело, тогда возвращается ноль.

Приведенный ниже код:

print(5 % 3)

print(10 % 6)

print(4 % 2)

print(22 % 10)

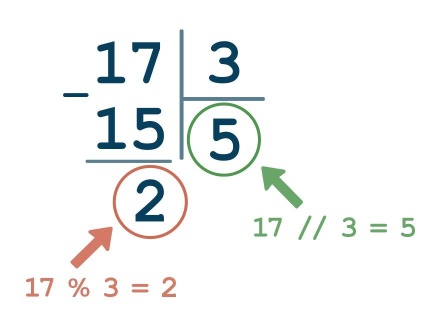
выводит

2

4

0

2



На картинке можно увидеть, что при целочисленном делении 17 на 3 результатом является число 5, а остаток при этом равен 2.

При делении отрицательных чисел с остатком важно помнить о логике целочисленного деления — округление работает вниз.

Данный код:

print(-7 % 3)

print(-10 % 7)

print(-22 % 10)

выводит

2

4

8

В этом примере ближайшим меньшим и кратным трём числом для −7 является −9, соответственно остаток равен 2 и вычисляется как разница между −7 и −9.

Оператор нахождения остатка от деления чаще всего используется в двух случаях:

* Для проверки чётности числа. Следующий код:

print(5 % 2)

print(16 % 2)

выводит

1

0

* Для выделения последней цифры числа. Следующий код:

print(14 % 10)

print(271 % 10)

print(7 % 10)

выводит

4

1

7

Обратите внимание на то, что для отрицательных чисел операция нахождения остатка будет давать иные результаты. Программа:

print(-14 % 10)

print(-271 % 10)

print(-7 % 10)

выводит

6

9

3

Для того чтобы правильно найти последнюю цифру у отрицательных чисел можно предварительно перевести их в положительные с помощью функции abs().

Функция abs(num) возвращает модуль числа num (для неотрицательных чисел возвращает это же число, а для отрицательных — противоположное по знаку число).

Программа:

print(abs(19) % 10)

print(abs(-271) % 10)

print(abs(-7) % 10)

выводит

9

1

7

Выделение последней цифры с помощью операции % часто используют вместе с целочисленным делением //, ведь последняя позволяет «удалить» последнюю цифру у числа.

В приведённом ниже коде:

a = 12345

last\_gidit = a % 10

a = a // 10

в переменную last\_digit записывается последняя цифра исходного числа (цифра 5), а далее переменная a перезаписывается на значение без выделенной цифры, то есть на 1234.

При поочерёдном использовании этих двух арифметических операций можно «достать» из числа все его цифры по очереди, начиная с последней. Только важно не забывать, что для отрицательных чисел нужно использовать модуль.

## Возведение в степень

Оператор возведения в степень (\*\*) возводит какое-то число в определенную степень.

Этот код:

print(2 \*\* 3)

print(5 \*\* 1)

print(10 \*\* 0)

print(-10 \*\* 2)

print(4 \*\* 0.5)

выводит

8

5

1

-100

2.0

Порядок выполнения нескольких операций возведения в степень подряд определяется справа налево.

Данный код:

print(2 \*\* 3 \*\* 2)

выводит

512

В этом примере сначала 3 возводится во 2 степень, а затем уже стоящая слева 2 возводится в 9 степень.

## Сокращенные арифметические операции

Благодаря сокращенным арифметическим операциям в Python объем кода зачастую можно сократить без потери смысла.

Приведенный ниже код:

a = 1000

a = a + 2

a = a \* 3

a = a - 4

a = a // 5

a = a % 2

a = a / 1

можно переписать следующим образом:

a = 1000

a += 2

a \*= 3

a -= 4

a //= 5

a %= 2

a /= 1

Сокращенные операторы (+=, \*=, -=, /=, //=, %=) удобно использовать в тех случаях, когда приходится изменять значение какой-то конкретной переменной.

## Приоритет арифметических операций

Порядок выполнения операций в Python совпадает с привычным нам математическим порядком: наибольшим приоритетом обладает возведение степень, за ним следуют операции умножения и деления, а после уже операции сложения и вычитания.

Приведенный ниже код:

print(5 + 2 \*\* 2 \* 7)

print(3 \* 2 + 10)

выводит

33

16

## Скобочные выражения

Если в выражении присутствуют скобки, тогда они и определяют порядок выполнения операций.

Данный код:

x = 2 + 2 \* 4

y = (2 + 2) \* 4

print(x)

print(y)

выводит

10

16

Порядок выполнения всех операций, в том числе выражений в скобках, совпадает с математическим.

С использованием скобок в Python можно вычислять значения длинных и многосоставных выражений.

Например, данное математическое выражение

можно переписать в Python:

a = (5034 \*\* 2 + 26 \* (-70) - (9340 + 26 \* 12 \*\* 3)) / (570 + 12 \* 45)

print(a)

и получить на выводе 22779.34054054054.

Обратите внимание на то, что весь знаменатель дроби взят в отдельные скобки.

Условный оператор

При написании программ важно уметь задавать их различное поведение в зависимости от той или иной ситуации. В данном уроке мы поговорим о том, как это делать в языке Python.

## Ключевое слово if

Для принятия решения программа проверяет истинность написанного условия. В Python существуют различные способы проверки, но в каждом случае возможен лишь один из двух возможных исходов: истина (True) или ложь (False).

Проверка условия осуществляется при помощи ключевого слова if (if с английского языка – если). Рассмотрим следующую программу:

a = int(input())

if a > 50:

print('введённое число больше 50')

Данная программа сначала считывает число с клавиатуры. Если введённое число будет больше 50, то на экран будет выведен текст введённое число больше 50. Если же число будет меньше или равно 50, то программа ничего не выведет.

## Немного поговорим о синтаксисе языка.

Двоеточие в конце строки с использованием оператора if нужно для того, чтобы сообщить интерпретатору, что далее находится блок команд, который нужно выполнить при истинности условия.

Если условие истинно, то выполнится весь блок команд. А если ложно, то весь блок пропустится. Сам блок команд задаётся при помощи отступов.

В качестве отступа блока кода принято использовать 4 пробела.

## True и False как результат сравнения

Важно заметить, что в приведённой выше программе само выражение a > 50 принимает одно из значений: True или False, в чем легко убедиться на примере следующей программы:

a = 10

b = 20

print(a > 0)

print(b > 100)

Программа выведет:

True

False

Здесь переменная a действительно больше 00 и выражение a > 0 примет значение True. Во втором случае b > 100 является ложью и выражение примет значение False.

Значения True и False принадлежат типу данных bool. Других значений в этом типе данных не существует.

## Оператор else

Мы научились задавать выполнение блока команд, если условие истинно. Часто бывает необходимым задавать блок команд и при случае, когда условие ложно. Для этого используется ключевое слово else (else с английского языка – иначе).

Рассмотрим следующую программу:

a = int(input())

if a > 33:

print('Условие истинно')

print('Введённое число больше 33')

else:

print('Условие ложно')

print('Введённое число не больше 33')

Данная программа считывает число с клавиатуры, и если оно больше 33, то выводит две строки: Условие истинно и Введённое число больше 33. Иначе же выводит другие две строки: Условие ложно и Введённое число не больше 33.

## Операторы сравнения

На данном этапе нам нужно знать и уметь пользоваться шестью операторами сравнения. Большинство из них интуитивно понятны и встречались вам в других дисциплинах, например, в математике.

|  |  |
| --- | --- |
| Оператор | Пояснение |
| > | больше |
| < | меньше |
| >= | больше или равно |
| <= | меньше или равно |
| == | равно |
| != | не равно |

Важно знать, что приоритет операторов сравнения ниже, чем у арифметических операций.

В данном коде:

a = 10

b = 20

if a + b > 25:

print('Мы молодцы')

в строке с проверкой условия сначала выполнится вычисление a + b, а только затем результат сравнится с числом 25. В итоге на экран будет выведен текст Мы молодцы.

— Чтобы запомнить порядок написания операторов >= и <= просто проговорите их вслух. Они пишутся в том порядке, в котором вы их произносите.

— Не стоит путать оператор присваивания = и оператор сравнения ==. Первый используется для связывания переменной и какого-либо значения, второй – для сравнения двух значений.

Логические операторы

В [предыдущем уроке](https://stepik.org/lesson/1367896/step/1?unit=1383937) мы узнали, как задавать логику работы программы таким образом, чтобы выполнять только блоки кода, соответствующие заданному условию. Но что делать, если условий несколько? В Python для этого существуют логические операторы, которые позволяют создавать сложные условия. Под «сложными» подразумеваются условия, состоящие из нескольких простых.

Всего в языке Python существуют три логических оператора:

and — логическое умножение (читается как «И»)

or — логическое сложение (читается как «ИЛИ»)

not — логическое отрицание (читается как «НЕ»)

## Оператор and

Допустим, вы захотели приобрести новенький айфон. Для себя вы решили, что хотите телефон 14-ой версии или новее стоимостью не более 100000 рублей. Для этого вы зашли на популярный сайт техники SND. Давайте напишем программу, которая поможет нам с выбором:

model = int(input('Укажите модель айфона'))

cost = int(input('Укажите стоимость телефона'))

if model >= 14 and cost <= 100000:

print('Покупаем')

else:

print('Смотрим дальше')

В данной программе мы объединили два простых условия model >= 14 и cost <= 100000 с помощью логического оператора and. Этот оператор означает, что блок кода выполнится только в том случае, если выполнятся **оба** условия одновременно.

Рассмотрим таблицу истинности для логического оператора and. В этой таблице перечислены значения выражения a and b при всех возможных комбинациях значений a и b

|  |  |  |
| --- | --- | --- |
| **a** | **b** | **a and b** |
| False | False | False |
| False | True | False |
| True | False | False |
| True | True | True |

Можно заметить, что выражение a and b является истинным тогда и только тогда, когда все условия, объединённые логическим оператором and по отдельности являются истинными.

## Оператор or

После сдачи ЕГЭ у вас будет выбор, куда поступать. Допустим, в сумме вы набрали 250 баллов. Для обучения на бюджете вы рассматриваете направления, на которые проходной балл не превышает 250250. Также вы рассматриваете платную форму обучения со стоимостью, не превышающей 200000 рублей в год. Давайте напишем программу, которая позволит нам отобрать подходящие направления:

passing\_score = int(input('Введите проходной балл в вуз'))

cost\_education = int(input('Введите стоимость обучения в вузе'))

if passing\_score <= 250 or cost\_education <= 200000:

print('Направление подходит')

else:

print('Направление не подходит')

В данной программе мы объединили два простых условия passing\_score <= 250 и cost\_education <= 200000 с помощью логического оператора or. Этот оператор означает, что блок кода выполнится только в том случае, если выполнится **хотя бы одно** условие из указанных.

Рассмотрим таблицу истинности для логического оператора or. В этой таблице перечислены значения выражения a or b при всех возможных комбинациях значений a и b

|  |  |  |
| --- | --- | --- |
| **a** | **b** | **a or b** |
| False | False | False |
| False | True | True |
| True | False | True |
| True | True | True |

Можно заметить, что выражение a or b является истинным всегда, кроме случая, когда оба условия ложны.

Условия, объединённые логическими операторами or и and могут входить в условие if вместе и в любом количестве.

Рассмотрим следующую программу:

passing\_score = int(input('Введите проходной балл'))

cost\_education = int(input('Введите стоимость обучения'))

distance = int(input('Введите расстояние до вуза'))

city = input('Введите город, в котором находится вуз')

if passing score <= 250 or ((distance <= 10 or city == 'Москва') and cost\_education <= 200000):

print('Подходит')

else:

print('Не подходит)

Данная программа отбирает такие вузы, в которые либо проходной балл меньше или равен 250, либо вузы, стоимость обучения в которых не превышает 200000 рублей, притом во втором случае вуз должен находиться либо на расстоянии, не превышающем 10 километров, либо в Москве.

Обратите внимание, что сейчас порядок выполнения условий задан скобками. В конце урока мы поговорим о приоритете логических операций.

## Оператор not

Оператор not позволяет инвертировать (заменять на противоположный) результат выполнения логического выражения.

Рассмотрим пример:

a = int(input('Введите любое число'))

if not(a > 33):

print('a меньше или равен 33')

else:

print('a больше 33')

Такой код полностью эквивалентен коду:

a = int(input('Введите любое число'))

if a <= 33:

print('a меньше или равен 33')

else:

print('a больше 33')

Условие a > 33 было взято в скобки в первом примере, чтобы показать, что оператор not применяется ко всему выражению целиком, а не только к a.

Рассмотрим таблицу истинности для логического оператора not. В этой таблице перечислены значения выражения not a при всех возможных комбинациях значениях a

|  |  |
| --- | --- |
| **a** | **not a** |
| False | True |
| True | False |

Таким образом, оператор not ложное выражение делает истинным, а истинное — ложным.

## Приоритеты логических операций

Если в выражении используются несколько логических операторов, то приоритет следующий:

1. Сначала выполняется логическое отрицание not
2. Затем логическое умножение and
3. И в конце логическое сложение or

Если мы хотим поменять порядок выполнения, то, как и с математическими операторами, можно использовать скобки.

## Цепочки сравнений

При написании условий часто удобно использовать цепочки сравнений. Рассмотрим следующую программу:

a = int(input())

if 100 <= a < 1000:

print('Положительное трёхзначное число')

В данной программе мы считываем одно число и если оно находится в диапазоне от 100 (включительно) до 1000 (не включительно), то выводим соответствующий текст на экран.

На самом деле такой код полностью эквивалентен коду:

a = int(input())

if 100 <= a and a < 1000:

print('Положительное трёхзначное число')

То есть цепочки сравнений — это неявное использование логического оператора and.

Давайте представим, что у нас есть три числа, и мы хотим вывести на экран текст «Все числа различны», если ни одно число не совпадает с другим. И текст «Не все числа различны» в противном случае. Рассмотрим ещё одну программу, отражающую частую ошибку в логике начинающего программиста:

a = int(input())

b = int(input())

c = int(input())

if a != b != c:

print('Все числа различны')

else:

print('Не все числа различны')

На самом деле данная программа будет работать неверно. Например, при вводе чисел 55, 23 и 5 программа будет выдавать, что все числа различны, хотя это таковым не является. Давайте перепишем эту программу с явным использованием оператора and :

a = int(input())

b = int(input())

c = int(input())

if a != b and b != c:

print('Все числа различны')

else:

print('Не все числа различны')

Тут становится понятно, в чём заключается ошибка. В выражении a != b мы действительно проверяем, что первое число не равно второму. В выражении b != c проверяем, что второе число не равно третьему. Но не проверяем, что первое число не равно третьему.

Один из вариантов правильного кода будет выглядеть следующим образом:

a = int(input())

b = int(input())

c = int(input())

if a != b and b != c and a != c:

print('Все числа различны')

else:

print('Не все числа различны')

## Ещё немножко про приоритеты

В процессе изучения урока вполне закономерно мог возникнуть вопрос: «А в каком порядке будут выполняться действия в условии, если в нём есть и знаки математических операций, и знаки сравнения, и логические операторы?».

На самом деле в языке Python, впрочем, как и в любом другом языке программирования, чётко установлен приоритет выполнения тех или иных операций и операторов:

* Сначала выполняются арифметические операции
* Далее выполняются операторы сравнения
* Затем логические операторы

Рассмотрим следующую программу:

a = int(input())

b = int(input())

c = int(input())

if a + b > c or a \* b > 0:

print("Поговорим о приоритетах")

Таким образом, в данной программе сначала произойдет сравнение суммы a и b со значением c и произведение a и b с 0, а затем полученные результаты (каждый из которых равен True либо False) объединятся логическим оператором or.

Каскадные и вложенные операторы

## Вложенный оператор

В предыдущих уроках мы говорили, что при истинности выражения, записанного в условном операторе if, программа выполняет блок кода, отделённый отступом. На самом деле внутри этого блока кода можно писать любые инструкции Python. В том числе ещё одно условие if. Такое условие будет называться вложенным.

Рассмотрим следующую программу:

height = int(input('Введите ваш рост'))

if height > 160:

if height > 180:

print('Вы высокий')

else:

print('Вы среднего роста')

else:

print('Вы невысокий, но очень милый')

В данной программе сначала сравнивается рост со 160, и если условие не выполняется, то срабатывает блок с else, который печатает на экран «Вы невысокий, но очень милый». Если же условие выполняется, то рост теперь сравнивается со 180. И в зависимости от истинности этого условия выводится соответствующий текст на экран.

Обратите внимание на то, что каждый вложенный блок команд имеет отступ б**о**льший, чем предыдущий.

Также стоит отметить особенность работы else: этот оператор относится к ближайшему if, написанному на таком же уровне, что и сам else.

Рассмотрим программу, показывающую к какой возрастной категории взрослого населения относится проходящий данный курс пользователь:

age = int(input('Укажите свой возраст'))

if age >= 90:

print('Поздняя старость')

else:

if age >= 75:

print('Старость')

else:

if age >= 60:

print('Пожилой возраст')

else:

if age >= 45:

print('Зрелость')

else:

print('Молодость')

В данной задаче мы используем именно вложенный оператор, а не последовательные if, идущие друг за другом. Связано это с тем, что при последовательном написании возраст, например, 8383 года прошёл бы сразу по трём условиям (age >= 75; age >= 60; age >= 45) и вывелось бы три строчки с текстом, что является идейно неверным в данной задаче.

Можно заметить основной минус в таком подходе написания условий: если вложенность будет ещё больше, то код будет уходить всё правее и правее, что в конечном итоге уменьшит его читабельность.

Для избегания высокой вложенности существует **каскадный оператор.**

## ****Каскадный оператор****

Перепишем предыдущую задачу с использованием каскадного оператора:

age = int(input('Укажите свой возраст'))

if age >= 90:

print('Поздняя старость')

elif age >= 75:

print('Старость')

elif age >= 60:

print('Пожилой возраст')

elif age >= 45:

print('Зрелость')

else:

print('Молодость')

По этой программе можно заметить, что оператор elif задаёт такую же логику работы, как и else if в примере выше.

Общий принцип работы каскадного оператора if-elif-else таков:

* сначала проверяется условие из инструкции с оператором if
* если условие оказалось ложным, то переходим к условию, написанному в первом elif
* если и это условие оказалось ложным, то переходим к следующему elif и т.д.
* если условие в последнем elif оказалось ложным, то выполняется код из блока else

— Количество инструкций elif не ограничено. Последний оператор else, как и в конструкции if-else, является необязательным.

— Использование каскадного оператора необязательно, но рекомендуется при большой вложенности условий.

Циклы в Python

Цикл — это конструкция, которая позволяет многократно выполнять заданный набор инструкций.

В языке Python существуют две основные разновидности циклов:

* Циклы, выполняющиеся заданное количество раз (цикл for)
* Циклы, выполняющиеся до наступления какого-то события (цикл while)

## Цикл for

Предположим, наша задача — вывести на экран значение переменной 5 раз. Это можно сделать следующим образом:

a = 15

print(a)

print(a)

print(a)

print(a)

print(a)

Приведённый выше код можно переписать с использованием цикла for

a = 15

for i in range(5):

print(a)

Обе программы выводят

15

15

15

15

15

В ряде задач требуется большое количество раз повторять одни и те же операции, что удобно реализовать с помощью циклов.

Структура цикла for следующая:

for переменная цикла in range(количество повторений):

блок кода

Блок кода внутри цикла называется телом цикла.

Как и при написании условия if, двоеточие в конце строки сообщает о том, что дальше находится блок кода, отделённый отступом. Этот блок выполнится столько раз, сколько повторений будет указано в качестве аргумента в функции range().

Рассмотрим программу, которая считывает 4 числа и выводит их на экран:

for i in range(4):

a = int(input('Считываем очередное число'))

print(a)

При последовательном вводе чисел 10, 20, 15, 40 данная программа выведет

10

20

15

40

## Переменная цикла

В общей структуре цикла for у нас фигурировала переменная цикла. Давайте поговорим о ней подробнее.

Рассмотрим следующую программу:

for i in range(7):

print(i)

Данная программа выведет

0

1

2

3

4

5

6

Здесь функция range() генерирует последовательность целых чисел от 0 до n **не** включительно, где n — количество повторений цикла, которое мы указали в качестве аргумента, а переменная цикла i ссылается на эти значения по очереди. Притом можно заметить, что значение, на которое ссылается переменная i , меняется в зависимости от номера текущего повторения цикла (как правило, говорят — текущей итерации цикла). Таким образом, обращаясь к i, мы обращаемся к числам 0,1,2,3,4,5,6 по очереди.

Примечание 1. Использование значений переменной цикла не является обязательным, но удобно в некоторых программах.

Примечание 2. Для имени переменной цикла действуют все те же ограничения, что и для имени обычной переменной.

Примечание 3. Обратите внимание на то, что в данном примере ровно такое число повторений, какое передано в функцию range().

## Функция range с двумя аргументами

Если необходимо изменить диапазон значений, на который будет ссылаться переменная цикла (по умолчанию от 0 до указанного числа не включительно), то можно передавать в функцию range() два аргумента вместо одного. В этом случае необходимо указать стартовое и конечное значение для переменной.

Рассмотрим следующую программу:

for j in range(-3, 5):

print(j, '— очередная итерация')

Данная программа выведет

-3 — очередная итерация

-2 — очередная итерация

-1 — очередная итерация

0 — очередная итерация

1 — очередная итерация

2 — очередная итерация

3 — очередная итерация

4 — очередная итерация

Здесь переменная j ссылается на целые числа от −3 до 5 не включительно.

Например, если нам нужно вывести квадраты всех двузначных натуральных чисел, то можем сделать это следующим образом:

for number in range(10, 100):

print(number \*\* 2)

В этом случае будет выведено:

100

121

144

...

9801

Примечание. Если второе число будет меньше или равно первого, то такой цикл выполнится ровно 0 раз.

В программе:

for t in range(34, 19):

print('Этот текст выведен не будет')

цикл не выполнится ни разу.

## Функция range с тремя аргументами

По умолчанию функция range() генерирует последовательность целых чисел с шагом 1. Этот шаг можно регулировать, если передать в функцию range() третий параметр. Следующий код:

for k in range(3, 20, 5):

print(k)

выведет

3

8

13

18

## Отрицательный шаг функции range

Во всех примерах выше значение, на которое ссылается переменная цикла, менялось от меньшего к большему. В ряде же задач удобно идти от большего числа к меньшему. В этом случае необходимо указать отрицательный шаг генерации.

Программа

for var in range(21, 9, -3):

print(var)

Выведет

21

18

15

12

Обратите внимание, что логика работы первых двух аргументов остаётся такой же: первый аргумент — левая граница, второй аргумент — правая граница не включительно, поэтому число 9 не будет выведено на экран.

## Оператор break

В ряде некоторых задач (в том числе из ЕГЭ) необходимо преждевременно осуществлять выход из цикла. Для этого используется оператор break. Когда интерпретатор доходит до инструкции, содержащей break, он выходит из цикла и переходит к дальнейшим строкам кода (при их наличии).

Рассмотрим пример:

for i in range(2, 10):

print(i)

if i > 5:

break

Данная программа выведет

2

3

4

5

6

То есть при i = 5 условие if i > 5 ещё не выполняется, а уже на следующем шаге при i = 6 оператор break прервёт выполнение цикла. Поэтому 6 — последнее число, которое выведет данная программа.

Примеры использования цикла for

Давайте рассмотрим популярные задачи с использованием циклов.

## Поиск количества с использованием счётчика

Очень часто при решении типовых задач приходится подсчитывать количество чисел из некоторой последовательности, которые удовлетворяют определенным условиям. Для этого удобно использовать переменную-счётчик, которая будет последовательно накапливать количество найденных чисел и в конце цикла выводить итоговый результат.

Например, данный код считает количество положительных чисел, оканчивающихся на 7, в диапазоне от −100 до 100:

count = 0

for num in range(-100, 100 + 1):

if num > 0 and num % 10 == 7:

count += 1

print(count)

И выводит

10

В этом коде сначала создается переменная-счетчик count с нулевым значением, а затем последовательно перебираются все числа из диапазона [−100;100], и каждый раз, когда выполняется условие if, значение count увеличивается на 11, что сигнализирует о том, что было найдено ещё одно очередное подходящее значение. И после выполнения цикла в count будет храниться итоговое количество найденных чисел.

! Обратите внимание на то, что для включения правой границы в перебор мы увеличиваем конечное значение на 11.

## Вычисления на числовом диапазоне

Нередко требуется обработать числа из определенного диапазона и произвести некоторые вычисления с ними. К подобного рода задачам можно отнести поиск суммы, произведения, среднего арифметического конкретных значений и прочее.

## Поиск суммы

Данный код:

s = 0

for num in range(50, 151):

if num % 12 == 5:

s += num

print(s)

находит сумму чисел из отрезка [50;150], остаток от деления на 12 которых равен 5.

В зависимости от задачи условие отбора может меняться, но общая реализация остается прежней.

## Поиск произведения

Данный код:

p = 1

for num in range(2, 11):

if num % 2 == 0:

p \*= num

print(p)

находит произведение чётных чисел из отрезка [2;10].

Обратите внимание на то, что начальное значение p равно единице.

Подобным образом можно вычислить факториал некоторого числа n.

Приведенный ниже код:

n = int(input())

factorial = 1

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

factorial \*= num

print(factorial)

при вводе числа 5 выведет 120.

Примечание. Факториал числа *n* — это произведение всех чисел от 1 до *n*.

## Поиск среднего арифметического

Приведенный ниже код:

s = count = 0

for num in range(103, 217):

if num % 5 == 0:

s += num

count += 1

average\_value = s / count

print(average\_value)

перебирает числа из отрезка [103;216] и находит среднее арифметическое всех чисел, удовлетворяющих условию. Для этого требуется сумму всех подходящих чисел разделить на их количество: в переменную s записывается сумма чисел, а в count — их количество.

Примечание. Можно отвечать и на другие вопросы, связанные с вычислением на числовых диапазонах. Выше приведены распространенные примеры.

## Поиск максимума и минимума

С помощью цикла можно обработать введённые с клавиатуры числа и найти максимальное и минимальное из них.

Пусть на вход подается *n* неотрицательных целых чисел, не превышающих 10000.

Данный код:

n = int(input())

max\_num = -1

min\_num = 10001

for \_ in range(n):

num = int(input())

if num > max\_num:

max\_num = num

if num < min\_num:

min\_num = num

print(min\_num, max\_num)

сначала создает переменные max\_num и min\_num, которые и будут использованы для сохранения максимального и минимального из введенных значений соответственно. Изначально max\_num принимает строго меньшее значение, чем минимальное из допустимых чисел. Это нужно для того, чтобы условие перезаписи максимума выполнилось независимо от величины введенного первого числа. Аналогично min\_num на старте принимает значение, большее максимального допустимого.

Пусть приведенной выше программе на вход будут подано число *n*=5, а затем 8,0,1056,8192,517. Составим таблицу пошагового выполнения программы и рассмотрим как меняются max\_num и min\_num на каждой итерации:

|  |  |  |  |
| --- | --- | --- | --- |
| **Номер итерации** | **Введенное число** | **min\_num** | **max\_num** |
| 1 | 8 | 8 | 8 |
| 2 | 0 | 0 | 8 |
| 3 | 1056 | 0 | 1056 |
| 4 | 8192 | 0 | 8192 |
| 5 | 517 | 0 | 8192 |

То есть на каждой итерации значения min\_num и max\_num будут перезаписываться в зависимости от величины введенного числа, а в конце цикла они будут содержать минимальное и максимальное из введенных значений: 0 и 8192 соответственно.

## Переменная цикла как порядковый номер элемента

Переменную цикла удобно использовать для отслеживания номера текущего элемента. Рассмотрим код:

for i in range(1, 6):

a = int(input())

print('Номер элемента —', i, 'Значение элемента —', a)

Данная программа после считывания очередного числа сразу же выводит его порядковый номер и значение.

Например, при последовательном вводе 10,20,−15,17,−4 на экран будет выведено

Номер элемента — 1 Значение элемента — 10

Номер элемента — 2 Значение элемента — 20

Номер элемента — 3 Значение элемента — -15

Номер элемента — 4 Значение элемента — 17

Номер элемента — 5 Значение элемента — -4

Цикл while

В предыдущих уроках мы говорили о цикле for, который выполнялся известное количество раз. Но в некоторых задачах это значение нам заранее неизвестно. Для их решения нам понадобится цикл while (while - "пока" в переводе с английского), который работает до тех пор, пока заданное условие истинно.

Структура цикла while следующая:

while условие:

блок кода

Рассмотрим программу, которая 5 раз выводит на экран Python для ЕГЭ:

for i in range(5):

print('Python для ЕГЭ')

С помощью цикла while можно задать ту же самую логику:

i = 0

while i < 5:

print("Python для ЕГЭ")

i += 1

Обратите внимание на то, что если в цикле for переменная i меняется автоматически, то в цикле while нам приходится прописывать эти изменения вручную.

Важно отметить, что любую программу с циклом for можно переписать с использованием цикла while, но не всегда это рационально.

Рассмотрим задачу:

*Необходимо считать с клавиатуры неизвестное количество чисел и найти их сумму, а признаком окончания ввода является 00.*

Программа для решения этой задачи может выглядеть так:

s = 0

num = int(input())

while num != 0:

s += num

num = int(input())

print(s)

В конструкции

while num != 0:

s += num

num = int(input())

блок команд внутри цикла выполняется до тех пор, пока условие истинно, в данном случае пока num не равен нулю:

1. сначала выполняется проверка условия num != 0
2. если условие истинно (т.е. num не равен нулю), к s прибавляется значение num
3. в num "считывается" очередное число
4. шаги 1-3 повторяются до тех пор, пока условие истинно (т.е. пока не будет введён 0)

Строка перед циклом

num = int(input())

необходима для считывания первого числа. Без неё переменная num не определена, и потому при выполнении условия num != 0 произойдет ошибка.

Обратите внимание на то, что количество итераций в цикле while будет зависеть от того, сколько раз выполнится условие.

Возможна ситуация, в которой цикл while не выполнится ни разу:

a = 10

while a > 20:

print("Этот текст никогда не выведется")

Здесь условие a > 20 не выполнится, поэтому заход в тело цикла не произойдет.

А в приведенном ниже случае цикл while будет выполняться бесконечно:

i = 0

while i < 10:

print(i)

поскольку i никак в теле цикла не меняется, и потому условие i < 10 будет всегда истинно.

Оператор break можно использовать и при работе с циклом while.

Приведённый ниже код:

num = 1

while num \*\* 2 < 50:

print(num \*\* 2)

num += 1

if num == 6:

break

выведет квадраты чисел от 1 до 5 включительно. Квадраты чисел 6 и 7 не будут выведены, поскольку оператор break досрочно прервет выполнение цикла при num = 6.

Вложенные циклы

В предыдущих уроках мы говорили, что в теле циклов for и while можно писать любые инструкции языка, в том числе разрешено использовать новые циклы. Такие циклы внутри других циклов будут называться вложенными.

Рассмотрим следующую программу:

for i in range(3):

for j in range(4):

print('i =', i, 'j =', j)

При запуске программы выполняется первая строка for i in range(3). В данный момент i = 0, и со следующей строки начинает выполняться тело этого цикла. В прошлых уроках мы узнали, что переход на следующую итерацию будет осуществлён, когда полностью выполнятся все инструкции тела цикла. В данном случае в теле цикла находится другой цикл. То есть пока не завершится вложенный цикл for j in range(4), переход на следующую итерацию внешнего цикла for i in range(3) осуществлён не будет. Таким образом при i = 0 переменная j примет значения 0,1,2,30,1,2,3 по очереди, при этом будет выполнена соответствующая функция print(). Далее будет осуществлён переход на новую итерацию внешнего цикла, переменная i станет равна единице, снова запустится цикл for j in range(4) и т.д. В конечном итоге на экран будет выведен следующий текст:

i = 0 j = 0

i = 0 j = 1

i = 0 j = 2

i = 0 j = 3

i = 1 j = 0

i = 1 j = 1

i = 1 j = 2

i = 1 j = 3

i = 2 j = 0

i = 2 j = 1

i = 2 j = 2

i = 2 j = 3

— Обратите внимание на то, что в случае вложенных циклов мы используем разные переменные цикла.

— Выполнение всех итераций вложенного цикла — всего лишь одна итерация цикла, который находится на уровень выше.

*Рассмотрим чуть более осмысленный пример*. Предположим, мы являемся товароведами в местном магазине «Шестёрочка». В конце дня мы с коллегой Галиной проводим ревизию остатков товара на складе. Галя нам говорит сначала названия отделов, затем перечисляет продукцию в каждом из них в формате «наименование товара — количество оставшейся продукции в у.е. (условных единицах)».

Наша задача — вывести количество оставшейся продукции для каждого отдела в формате «отдел — остаток товара». Притом заведомо известно, что всего отделов 4, в каждом из них по 3 товара.

Напишем программу для решения данной задачи:

n = 4

k = 3

for i in range(n):

name = input() # считываем название отдела

sum\_balance = 0 # остаток товара для очередного отдела

for j in range(k):

product = input() # считываем название товара

balance = int(input()) # считываем остаток товара

sum\_balance += balance # увеличиваем остаток товаров отдела

print(name, '-', sum\_balance) # выводим остаток для текущего отдела

В данной программе в теле цикла for i in range(n) мы считываем название отдела, при этом указываем, что остаток товара пока что равен нулю. Для этого отдела мы запускаем цикл для пересчёта остатка товара (сначала даётся наименование, затем количество. Поэтому два раза используется функция input()). В переменную sum\_balance мы прибавляем остаток каждого товара в данном отделе. Когда цикл с переменной j полностью выполнится, это означает, что мы посчитали все товары конкретного отдела, и можем выводить данные на экран. После выполнения функции print() осуществляется переход на следующую итерацию цикла с переменной i, снова вводится название отдела и (**важно**) снова задаётся sum\_balance = 0, потому что в новом отделе суммарный остаток пока что нулевой. Далее программа выполняется до своего логического завершения и постепенно выводятся остатки для каждого отдела в требуемом формате.

Обратите внимание на то, что переменная product никак не используется в выводе текста на экран, но она необходима, чтобы удовлетворить условия считывания данных.

*Рассмотрим пример, использующий б****о****льшую вложенность*. Представим, что после долгого изучения языка Python, вы отправились в десятичасовой перелёт в очень жаркие страны. Но из-за джетлага (синдрома смены часового пояса) вы забыли код от чемодана, в котором у вас все вещи (а главное — купальник и плавки).

Смоделируем подбор четырёхзначного шифра при помощи вложенных циклов:

for i in range(10):

for j in range(10):

for k in range(10):

for m in range(10):

print(i, j, k, m)

При выполнении первой строки фиксируется i = 0 и начинает выполняться тело цикла. В нём запускается новый цикл, фиксируется j = 0. В нём начинается цикл с k = 0 и наконец в нём запускается цикл, на первой итерации которого m = 0. Таким образом мы дошли до строки print(i, j, k, m) и вывели на экран 0 0 0 0. После вывода первого числа заканчивается первая итерация цикла for m in range(10). Теперь осуществляется переход на вторую итерацию, то есть m становится единицей (заметьте, что остальные переменные всё ещё равны нулю). По итогу вторым напечатанным числом станет 0 0 0 1. Затем выведутся коды 0 0 0 2, 0 0 0 3 и т.д. На последней итерации четвёртого цикла выведется 0 0 0 9. На данном этапе последний цикл завершается, то есть заканчивает выполняться блок команд, который находится внутри цикла for k in range(10). Следовательно, осуществляется переход на следующую итерацию этого цикла: k становится единицей и начинает выполняться тело этого цикла. В нём запускается цикл for m in range(10), который снова начинает выводить значения на экран: 0 0 1 0, 0 0 1 1, 0 0 1 2 ... 0 0 1 9. Завершается четвёртый цикл, k меняется на двойку и уже понятно, что будет дальше. В какой-то момент будет выведено 0 0 9 9. И только тогда завершается первая итерация цикла for j in range(10). Далее j становится единицей и всё начинается «с начала» для циклов с переменными k и m.

Таким образом данный код выведет на экран

0 0 0 0

0 0 0 1

0 0 0 2

0 0 0 3

0 0 0 4

0 0 0 5

0 0 0 6

0 0 0 7

0 0 0 8

0 0 0 9

0 0 1 0

0 0 1 1

.......

9 9 9 9

Рассмотрим пример построения части таблицы умножения с использованием вложенных циклов.

Этот код:

for i in range(10):

for j in range(7):

print(j, '\*', i, '=', j \* i, end=' ') # заменяем перенос строки на несколько пробелов

print() # добавляем перенос строки в конце очередной итерации внешнего цикла

выводит

0 \* 0 = 0 1 \* 0 = 0 2 \* 0 = 0 3 \* 0 = 0 4 \* 0 = 0 5 \* 0 = 0 6 \* 0 = 0

0 \* 1 = 0 1 \* 1 = 1 2 \* 1 = 2 3 \* 1 = 3 4 \* 1 = 4 5 \* 1 = 5 6 \* 1 = 6

0 \* 2 = 0 1 \* 2 = 2 2 \* 2 = 4 3 \* 2 = 6 4 \* 2 = 8 5 \* 2 = 10 6 \* 2 = 12

0 \* 3 = 0 1 \* 3 = 3 2 \* 3 = 6 3 \* 3 = 9 4 \* 3 = 12 5 \* 3 = 15 6 \* 3 = 18

0 \* 4 = 0 1 \* 4 = 4 2 \* 4 = 8 3 \* 4 = 12 4 \* 4 = 16 5 \* 4 = 20 6 \* 4 = 24

0 \* 5 = 0 1 \* 5 = 5 2 \* 5 = 10 3 \* 5 = 15 4 \* 5 = 20 5 \* 5 = 25 6 \* 5 = 30

0 \* 6 = 0 1 \* 6 = 6 2 \* 6 = 12 3 \* 6 = 18 4 \* 6 = 24 5 \* 6 = 30 6 \* 6 = 36

0 \* 7 = 0 1 \* 7 = 7 2 \* 7 = 14 3 \* 7 = 21 4 \* 7 = 28 5 \* 7 = 35 6 \* 7 = 42

0 \* 8 = 0 1 \* 8 = 8 2 \* 8 = 16 3 \* 8 = 24 4 \* 8 = 32 5 \* 8 = 40 6 \* 8 = 48

0 \* 9 = 0 1 \* 9 = 9 2 \* 9 = 18 3 \* 9 = 27 4 \* 9 = 36 5 \* 9 = 45 6 \* 9 = 54

## Оператор break

В прошлых уроках мы говорили, что оператор break нужен для преждевременного выхода из цикла. Во вложенных циклах логика его работы остаётся прежней. Только стоит отметить, что break прерывает ближайший цикл, в котором он расположен.

Рассмотрим пример:

for i in range(4):

for j in range(20):

print('i =', i, 'j =', j)

if j > 2:

break

При запуске данной программы получим

i = 0 j = 0

i = 0 j = 1

i = 0 j = 2

i = 0 j = 3

i = 1 j = 0

i = 1 j = 1

i = 1 j = 2

i = 1 j = 3

i = 2 j = 0

i = 2 j = 1

i = 2 j = 2

i = 2 j = 3

i = 3 j = 0

i = 3 j = 1

i = 3 j = 2

i = 3 j = 3

То есть при достижении j = 3 сначала выполнится инструкция print(), т.к. она находится до проверки условия, а далее условие станет истинным, break прервёт ближайший цикл (цикл с переменной j) и мы «досрочно» перейдём на следующую итерацию цикла с переменной i.

## Основные логические операции

К основным логическим операциям относятся:

* инверсия (логическое "НЕ")
* конъюнкция (логическое "И")
* дизъюнкция (логическое "ИЛИ")
* импликация
* эквивалентность (логическое "РАВНО")

Каждую из них можно переписать на языке программирования Python с помощью операторов логики и сравнения.

|  |  |  |
| --- | --- | --- |
| **Логическая операция** | **Запись в АЛ** | **Запись в Python** |
| Инверсия | ¬ | not |
| Конъюнкция | ∧, & | and |
| Дизъюнкция | ∨ | or |
| Импликация | → | <= |
| Эквивалентность | ≣ | == |

## Приоритеты логических операций

Важно понимать, что порядок выполнения приведенных выше операций в АЛ и в Python отличается.

Приоритеты в алгебре логики:

1. инверсия
2. конъюнкция
3. дизъюнкция
4. импликация
5. эквивалентность

Приоритеты в Python:

1. <= и ==
2. not
3. and
4. or

Отсюда следует, что при записи логических функций в Python следует корректно переопределять приоритеты в соответствии с правилами, установленными в АЛ. Для этого используются скобки.

## Запись логических функций в Python

Рассмотрим несколько логических функций и сравним их запись в АЛ и на Python. Будем считать, что переменные (x, y, z, w), используемые в логической функции, в Python уже определены.

Пример №1. *F*=(*z*∨*y*)→(*x*≡*z*)

Запись на Python:

f = (z or y) <= (x == z)

Пример №2. *F*=(*y*→¬(*x*→*z*))∨*w*

Запись на Python:

f = (y <= (not(x <= z))) or w

Обратите внимание на то, что для отрицания скобки (𝑥→𝑧)(*x*→*z*) на Python требуется обернуть запись not(x <= z) в отдельные скобки и получить (not(x <= z)), что важно для верного соблюдения порядка выполнения операций — любые выражения с not следует оборачивать в дополнительные скобки.

Пример №3. *F*=¬(*x*∨*y*)∧¬*w*∨¬(*z*∨*w*)∧*y*

Запись на Python:

f = (not(x or y)) and (not w) or (not(z or w)) and y

Пример №4. *F*=((*x*→*w*)∧(¬*y*→*z*))→((*z*≡*x*)∨(*w*∧¬*y*))

Запись на Python:

f = ((x <= w) and ((not y) <= z) <= ((z == x) or (w and (not y))))

Пример №5. *F*=(*x*→*y*∧¬*z*)∨*w*

Запись на Python:

f = (x <= (y and (not z))) or w

Обратите внимание на то, что в этом примере при перезаписи на язык программирования часть *y*∧¬*z* важно взять в отдельные скобки, чтобы сначала выполнилась именно конъюнкция, ведь по умолчанию в Python оператор <= имеет больший приоритет, чем not.

Строковый тип данных

В программировании очень часто приходится работать со строками. Ранее мы уже говорили вкратце о строковом типе данных.

## Инициализация строк

Вспомним о том, что для создания переменной типа str необходимо облачить текст в двойные или одинарные кавычки:

string = "Yeeeeh, new string!"

Строка может быть также пустой:

empty\_string = ""

Или содержать всего один пробел:

space\_string = " "

При вводе данных через input() **всегда** возвращается строка. Данный код:

input\_string = input()

print(type(input\_string))

при вводе любого текста выводит

<class 'str'>

Для перехода на следующую строку используется метасимвол \n. Этот код:

text = "Я люблю Pascal -.-\nХа-ха, шучу, конечно Python!"

print(text)

выводит

Я люблю Pascal -.-

Ха-ха, шучу, конечно Python!

## Преобразование типов

С числами можно работать как со строками, для чего прежде их требуется преобразовать к строковому типу данных. Для этого используется функция str().

Данный код:

num = 5

print(type(num))

num = str(num)

print(type(num))

выводит

<class 'int'>

<class 'str'>

Строки тоже можно преобразовывать к типам данных int и float, но только в том случае, если они удовлетворяют соответствующему формату и состоят только из десятичных цифр (и плавающей точки для типа float).

Код:

a = "10"

print(40 + int(a))

b = "25.3"

print(float(b) + int(a))

выводит

50

35.3

Если попробовать привести к числовому формату неподходящую для этого строку, то это приведёт к ошибке, в которой будет сказано о невозможности перевода.

Программа:

s = 'This is stroka'

a = int(s)

Выведет

ValueError: invalid literal for int() with base 10: 'This is stroka'

## Операции над строками

Над строками, как и над числами, можно производить различные операции. Пришло время рассмотреть основные из них.

### Поиск длины

Количество символов, входящих в строку, называется её длиной. Для определения длины строки используется встроенная в Python функция len() («length» — длина в переводе с английского языка).

Приведённый ниже код:

name = "Deonis"

age = '49'

status = 'VIP'

regalia = ""

print(len(name))

print(len(age))

print(len(status))

print(len(regalia))

выводит

6

2

3

0

Функция len() возвращает число (тип данных **int**), над которым можно производить все свойственные числам операции.

Данный код:

name = "Deonis"

age = '49'

status = 'VIP'

regalia = ""

print(len(name) + len(age) + len(status) + len(regalia))

выводит 11 — суммарную длину всех заданных строк.

Обратите внимание на то, что пробел с точки зрения языка программирования — символ, который по свойствам ничем не отличается от других символов.

Программа:

s = 'Pro bel'

print(len(s))

выводит

7

### Конкатенация строк

Конкатенация — это операция соединения двух или более строк в одну. В Python она определяется оператором +.

Данный код:

name = "Denis"

surname = "Bakhtiev?"

print(name + " " + surname)

res = "Leonid" + " " + "Shastin" + "!"

print(res)

выводит

Denis Bakhtiev?

Leonid Shastin!

Важно заметить, что при конкатенации строк не происходит разделения через пробел или какой-либо другой символ.

Этот код:

print('Semyon' + 'Chaykin')

выводит

SemyonChaykin

Для склеивания элементов других типов данных в одну строку их сначала нужно преобразовать к типу str.

Приведенный ниже код:

x = 70

y = 30

print(x + y)

print(str(x) + str(y))

выводит

100

7030

### Дублирование строк

С помощью оператора \* в Python можно умножать строку на число, тем самым дублируя её определенное количество раз.

Данный код:

name = "Leo"

print(name \* 3)

print('some\_string' \* 0)

point = '100' \* 5

print(point)

выводит

LeoLeoLeo

100100100100100

Обратите внимание на то, что результатом умножения строки на ноль является пустая строка.

### Сравнение строк

Строки можно сравнивать между собой с помощью операторов == и !=.

Две строки считаются равными, если они имеют одинаковую длину и полностью идентичны друг другу. Сравнение строк происходит посимвольно, и если каждый символ в первой строке соответствует символу в другой строке, тогда строки считаются равными.

Данный код:

print('sunny' == "sunny")

print("Sunny" == 'sunny')

print('cold' != 'Cold')

выводит

True

False

True

Не забывайте о том, что Python — регистрозависимый язык, а потому одинаковые символы, записанные в разных регистрах, считаются разными.

### Оператор in

С помощью встроенного в Python оператора in можно определить вхождение одной строки в другую.

Этот код:

print("ab" in "aba")

print("leo" in "den")

print("zx" in "zyx")

print("ara" in "barabara")

выводит

True

False

False

True

Иногда удобно использовать оператор in в связке с not. В таком случае мы проверяем не наличие подстроки в строке, а её отсутствие.

Этот код:

print('1s2' not in 'ABCDE')

print('DF' not in 'QWDFRP')

выводит

True

False

Обратите внимание на то, что оператор in проверяет точное вхождение одной подстроки в другую, регистр при этом тоже имеет значение.

Данный код:

print('a' in 'AA')

print('var' in 'vaabr')

выводит

False

False

Индексация и срезы

В [прошлом уроке](https://stepik.org/lesson/1367919/step/1?unit=1383957) мы познакомились с основными операциями над строками. Пришла пора разобраться со структурой строк подробнее и научиться их более гибко обрабатывать.

## Индексация

Строка — это неизменяемая упорядоченная коллекция символов, каждый из которых имеет определенный номер (индекс).

Символы в строках нумеруются слева направо в порядке возрастания, начиная с нуля.

Данная строка

name = "Leonid"

состоит из 6 символов, имеющих индексы (номера) от 0 до 5:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Символ | L | e | o | n | i | d |
| Индекс | 0 | 1 | 2 | 3 | 4 | 5 |

А эта строка

status = "copyrightable"

состоит из 13 символов, которые индексируются так:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Символ | c | o | p | y | r | i | g | h | t | a | b | l | e |
| Индекс | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |

Первый символ любой строки всегда находится под индексом 0, а последний символ имеет индекс, на единицу меньший длины строки. В примерах выше в строках с длинами 6 и 13 последние символы имеют индексы 5 и 12 соответственно.

Каждый символ строки можно получить, обратившись к нему по его индексу. Для этого необходимо указать коллекцию, к элементу которой нужно обратиться, а рядом в квадратных скобках [] указать индекс элемента.

Этот код:

name = "Leonid"

print(name[0])

print(name[2])

print(name[5])

выводит

L

o

d

Нумерация символов в строке поддерживается также и в обратном порядке (справа налево), причем самый крайний символ будет иметь индекс, равный −1−1.

Символы этой строки при отрицательной индексации

name = "Leonid"

будут иметь следующие номера:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Символ | L | e | o | n | i | d |
| Индекс | -6 | -5 | -4 | -3 | -2 | -1 |

Последний символ строки в такой нумерации всегда можно получить по индексу −1, а первый символ — по индексу, равному длине строки с отрицательным знаком.

Данный код:

name = "Leonid"

print(name[-1])

print(name[-3])

print(name[-6])

выводит

d

n

L

Обратите внимание на то, что при попытке обратиться к несуществующему индексу произойдет ошибка.

Эта программа:

name = "Leonid"

print(name[6])

завершится с ошибкой

IndexError: string index out of range

по причине того, что в строке "Leonid" всего 6 символов, последний из которых имеет индекс, равный 5.

Примечание. К последнему индексу строки удобнее обращаться через отрицательную индексацию (−1−1), но ещё его можно вычислить универсально для строки s как len(s) - 1.

С символами можно работать как с отдельными строковыми объектами.

Например, код:

letters = "ABCDEF"

print(letters[0] + letters[2] + letters[-1])

выводит

ACF

## Обход строк

С помощью цикла for можно посимвольно «обойти» всю строку. Для перебора всех индексов достаточно передать в функцию range() длину строки, поскольку первый символ в строке имеет индекс 0, а последний — len(строка) - 1. Функция range(len(строка)) как раз генерирует необходимую последовательность целых чисел.

Данный код:

mood = "memories"

for i in range(len(mood)):

print(i, mood[i])

выведет

0 m

1 e

2 m

3 o

4 r

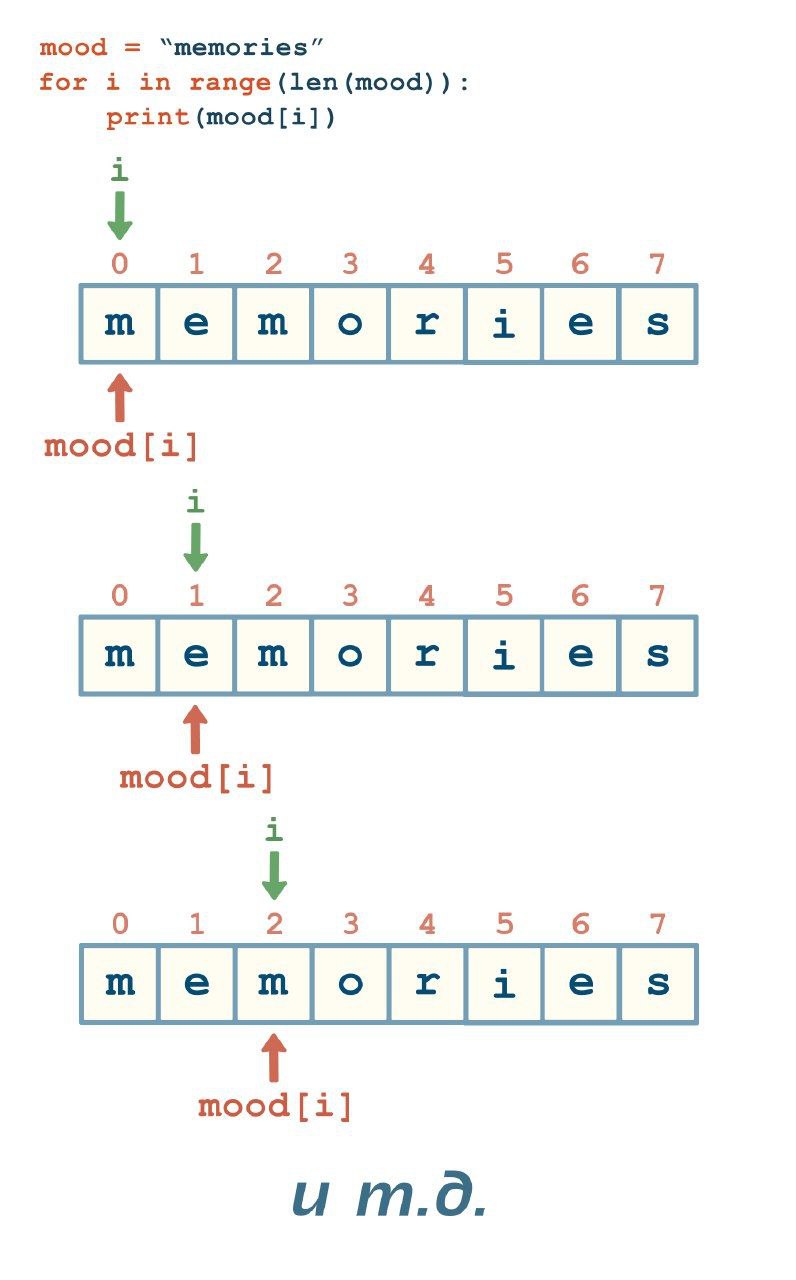
5 i

6 e

7 s

Здесь переменная i последовательно будет принимать значения индексов всех символов строки.

Ниже проиллюстрирован принцип обхода строки с использованием индексации:



Также цикл for позволяет посимвольно перебирать непосредственно элементы коллекций.

Этот код:

mood = "memories"

for sym in mood:

print(sym)

выведет

m

e

m

o

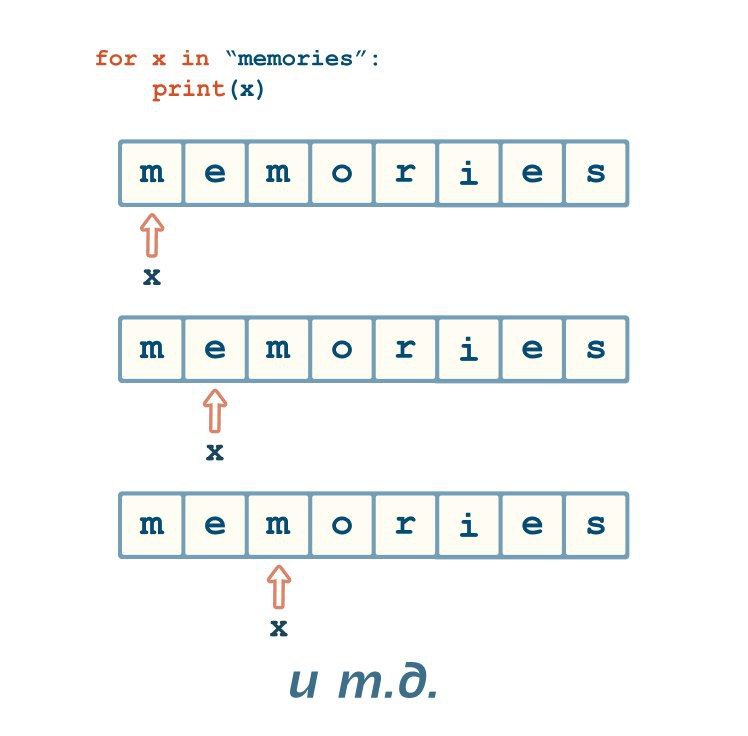
r

i

e

s

В нём переменная sym последовательно будет ссылаться на каждый символ строки от первого и до последнего. Отличие этого способа обхода строки от предыдущего заключается в том, что в первом случае есть возможность одновременно взаимодействовать как с символом, так и с его индексом.

Ниже проиллюстрирован принцип обхода строки без использования индексации:  


## Срезы

В некоторых случаях возникает необходимость получить целый фрагмент строки, состоящий из нескольких символов. Для решения этой задачи удобно использовать срезы, которые позволяют выделить сразу несколько идущих подряд символов.

Упрощённый синтаксис среза выглядит следующим образом:

string[start:stop]

Здесь start и stop задают соответственно начальный и конечный индексы строки. При таком срезе выделяются элементы строки от индекса start до индекса stop **не** включительно.

Рассмотрим на примере строки string = "Haphazard". Символы в ней индексируются так:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Индекс (+) | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| Символ | H | a | p | h | a | z | a | r | d |
| Индекс (-) | -9 | -8 | -7 | -6 | -5 | -4 | -3 | -2 | -1 |

Срез string[2:6] вернёт все символы, начиная с индекса 2 и заканчивая индексом 6 не включительно: phaz.

А срез string[0:4] вернёт первые четыре символа: Haph.

Также можно использовать отрицательную индексацию: срез string[-6:-2] вернёт haza.

Допустимо не указывать start среза, тогда будет взят фрагмент от начала строки и до значения stop. Срез string[:5] эквивалентен срезу string[0:5] и возвращает Hapha.

Аналогично можно не указывать stop, тогда будет взят фрагмент от значения start и до конца строки. Срез string[6:9] эквивалентен срезу string[6:] и возвращает ard.

Срезы никогда не приведут к ошибке индексации.

Данный код:

string = "Haphazard"

print(string[5:10000])

выведет

zard

Если же в указанном срезе не содержится ни одного символа, то выведется пустая строка.

Код:

string = "Haphazard"

print(string[5000:10000])

выведет пустую строку

Срез также допускает указание шага. Общий синтаксис среза выглядит так:

string[start:stop:step]

Здесь start и stop задают соответственно начальный и конечный индексы строки, а step определяет шаг выделения элементов. При таком срезе выделяются элементы строки от индекса start до индекса stop **не** включительно с шагом step.

Срез string[4:7:1] вернёт все символы, начиная с индекса 44 и заканчивая индексом 77 не включительно, с шагом 11: aza.

Данный код:

string = "Haphazard"

print(string[2:5:1])

print(string[0:4:1])

выводит

pha

Haph

По умолчанию start = 0, stop = len(строка), а step = 1.

Так, срез string[:4] полностью идентичен срезу string[0:4:1], а срез string[5:-1:1] полностью идентичен срезу string[5:-1:].

В некоторых случаях приходится брать каждый 2-й, 3-й или *n*-й элемент некоторой подстроки, тогда важна настройка шага.

Этот код:

string = "Haphazard"

print(string[2:6:2])

print(string[1:7:3])

print(string[0:-1:4])

выводит

pa

aa

Ha

Также можно использовать отрицательный шаг среза. В этом случае символы будут выводиться в обратном порядке.

Для строки string = 'Haphazard' срез string[4:1:-1] вернёт ahp.

Чаще всего срезы используются для взятия первых или последних n символов строки. Удобно запомнить, что:

* срез string[:n] возвращает первые n символов строки string
* срез string[-n:] возвращает последние n символов строки string
* срез string[n:] возвращает все символы строки, начиная с элемента с индексом n
* срез string[:-n] возвращает все символы от начала строки и до элемента с индексом -n - 1
* срез string[a:b] возвращает все символы строки, начиная с элемента с индексом a и заканчивая элементом с индексом b - 1

Ещё два интересных сценария использования срезов:

* срез string[:] возвращает полную копию строки, опуская определение значений start, stop и step
* срез string[::-1] возвращает перевернутую (задом наперед) строку

Данный код:

string = "Haphazard"

print(string[:])

print(string[::-1])

выводит

Haphazard

drazahpaH

## Изменение строк

При работе со строками важно помнить о том, что они относятся к неизменяемому типу данных, а потому изменить существующую строку нельзя. Попытка изменить какую-то часть строки приведет к ошибке.

Приведенный ниже код:

s = "hack"

s[0] = "b"

завершится с ошибкой

TypeError: 'str' object does not support item assignment

Та же ошибка возникнет при попытке заменить какую-то часть строки с помощью среза.

Однако для решения поставленной задачи можно создать новую строку, в процессе объявления которой просто заменить интересующую последовательность символов. Для этого удобно использовать срезы.

Данный код:

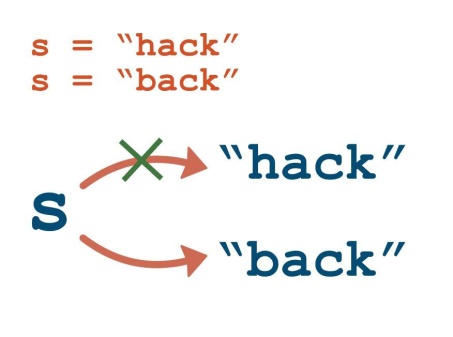
s = "hack"

s = "b" + s[1:]

print(s)

создает новую строку, присваивает ссылку на неё в переменную s и выводит

back



Важно понимать, что во второй строке рассмотренного выше кода не изменяется старая строка, а создается новый строковый объект.

Методы строк

Метод в Python — это специализированная функция, применяемая к определённым объектам. В данном уроке мы поговорим о часто используемых в ЕГЭ строковых методах.

Общий синтаксис применения метода к объекту таков:  
объект.метод(аргументы)  
  
В таблице ниже приведены популярные в ЕГЭ строковые методы:

|  |  |
| --- | --- |
| Название метода | Описание |
| strip() | удаляет пробельные символы в начале и в конце строки |
| count() | подсчитывает количество вхождений указанного объекта в строку |
| index() | находит первый индекс указанного элемента в строке |
| replace() | заменяет вхождения подстрок внутри строки |

### Метод strip()

Метод strip() возвращает копию исходной строки, удаляя слева и справа от неё пробелы и знаки переноса \n.

Данный код:

string = " Religion needed a miracle! \n"

print(string.strip())

удаляет пробелы и знаки переноса с левого и правого краев строки string и выводит

Religion needed a miracle!

### Метод count()

Метод count(substring) возвращает число вхождений указанной подстроки substring в исходную строку.

Этот код:

s1 = "BaAaaaAaabA"

print(s1.count('A'))

s2 = 'A pythonist loves a python :)'

print(s2.count("python"))

выводит

3

2

Важно понимать, что этот метод подсчитывает только непересекающиеся вхождения подстрок в исходную строку.

Приведенный ниже код:

string = "KOKOKOK"

print(string.count("KOK"))

выводит

2

несмотря на то, что в строку входят сразу 3 подстроки KOK. Такая логика работы связана как раз с пересечениями.

### Метод index()

Метод index(substring) возвращает индекс, с которого начинается первая найденная искомая подстрока

Данный код:

s = "bartender"

print(s.index("end"))

print(s.index("t"))

print(s.index("art"))

выводит

4

3

1

Особенностью работы этого метода заключается в том, что если искомой подстроки не найдется в исходной строке, тогда вернётся ошибка.

Этот код:

s = "bartender"

print(s.index("barr"))

завершится с ошибкой ValueError: substring not found.

В тех случаях, когда точно неизвестно, содержится ли искомая подстрока в исходной строке, можно воспользоваться проверкой с помощью уже известного нам оператора in.

Данный код:

s = "bartender"

if "barr" in s:

print(s.index("barr"))

else:

print('Подстрока не найдена')

проверяет, содержится ли подстрока barr в строке bartender и выводит

Подстрока не найдена

### Метод replace()

Метод replace(old, new, count) возвращает копию исходной строки, в которой вхождения old заменены на new.

Приведенный ниже код:

s1 = "mother"

s2 = "dislike"

s3 = "I love Pascal and only Pascal"

print(s1.replace("mot", "fut"))

print(s2.replace("like", "miss"))

print(s3.replace("Pascal", "Python"))

выводит

futher

dismiss

I love Python and only Python

Аргумент count является необязательным и позволяет задать максимальное количество выполняемых замен. По умолчанию, если count не задан, заменяются все вхождения.

Данный код:

s = "I love Pascal and only Pascal"

print(s.replace("Pascal", "Python", 1))

заменяет только первое вхождение Pascal в исходную строку и выводит

I love Python and only Pascal

Важно помнить о том, что строки являются неизменяемыми объектами, а потому строковые методы strip() и replace() не изменяют исходную строку, а возвращают новый строковый объект.

Данный код:

s1 = " ABC"

s1.strip()

print(s1)

s2 = "deed"

s2.replace("e", "o")

print(s2)

не изменяет строки s1 и s2 и выводит

ABC

deed

Для сохранения получившегося результата, возвращаемого методом, необходимо перезаписать его в какую-либо переменную.

Этот код:

s1 = " ABC"

s1 = s1.strip()

print(s1)

s2 = "deed"

s2 = s2.replace("e", "o")

print(s2)

перезаписывает возвращаемые методами новые строковые объекты в переменные s1 и s2 соответственно и выводит

ABC

dood

## Функции перевода между системами счисления

Для решения некоторых задач из ЕГЭ требуется переводить числа из одной системы счисления в другую. В этом уроке посмотрим как с помощью языка программирования можно осуществить такой перевод.

## Функции bin(), oct(), hex()

Для перевода числа из десятичной системы счисления в двоичную, восьмеричную и шестнадцатеричную используются соответственно функции bin(), oct() и hex().

### Функция bin()

Функция bin() принимает один аргумент — целое число и возвращает результат его перевода в двоичную систему счисления в **строковом** формате.

Код:

a = 20

b = bin(a)

print(b)

выводит

0b10100

где 10100 — результат перевода, 0b — префикс, обозначающий двоичную систему счисления.

Чтобы избавиться от префикса (первых двух символов) можно воспользоваться срезом.

Следующая программа:

a = 20

b = bin(a)

b = b[2:]

print(b)

выводит

10100

Как правило, строку с переводом и срезом объединяют и получают такой код:

a = 20

b = bin(a)[2:]

print(b)

Здесь важно заметить, что логика использования срезов не меняется. Во второй строке сначала выполнится функция bin(), которая вернёт строку со значением, а затем к ней применится срез.

### Функции oct() и hex()

Функция oct() принимает один аргумент — целое число и возвращает результат его перевода в восьмеричную систему счисления в **строковом** формате.

Функция hex() принимает один аргумент — целое число и возвращает результат его перевода в шестнадцатеричную систему счисления в **строковом** формате.

Программа:

a = 33

a\_8 = oct(a)

a\_16 = hex(a)

print(a\_8)

print(a\_16)

выводит

0o41

0x21

Результат выполнения функций oct() и hex() содержит префикс из двух символов 0o и 0x соответственно, от которых можно избавиться аналогичным образом, как мы поступали с функцией bin(), то есть используя срез.

## Функция int()

Ранее мы упоминали, что функцию int() можно использовать для перевода строкового значения в целочисленное. На самом же деле её функционал немного шире: функцию int() можно использовать, чтобы перевести число из допустимой системы счисления в десятичную. При этом первым аргументом указывается значение, которое мы переводим в **строковом** формате, а вторым — из какой системы счисления переводим.

Данный код:

a = '10100'

b = '41'

c = '21'

a\_10 = int(a, 2)

b\_10 = int(b, 8)

c\_10 = int(c, 16)

print(a\_10, b\_10, c\_10, sep='\n')

выводит

20

33

33

Разумеется, если нам дано число в иной системе счисления в формате int, то для перевода его нужно предварительно перевести в строковый формат, воспользовавшись функцией str(). Приведём пример кода, который осуществляет подобные действия:

a = 12345 # подразумевается, что это восьмеричное число

a = str(a)

b = int(a, 8)

print(b)

Далее рассмотрим примеры использования этих функций на задачах из ЕГЭ.

## Тип данных list

Ранее мы отдельно работали со строками и числами, но зачастую при решении различных задач необходимо хранить целые наборы данных, содержащие в себе сразу несколько объектов. Для этого используются различные структуры данных. В этом уроке мы поговорим об одной из них — о списках.

Список (он же тип данных list) — это упорядоченный набор элементов. В отличие от строк, списки являются изменяемыми коллекциями и могут содержать в себе не только символы, но и объекты других типов (int, str, float, list и т.д.).

## Инициализация списка

Пусть у нас есть набор тропических фруктов: апельсины, ананасы и бананы.

Сохранить информацию о них в Python без использования списка можно так:

fruit\_1 = "апельсин"

fruit\_2 = "банан"

fruit\_3 = "ананас"

А можно все эти строки сохранить в одном списке, для создания которого необходимо перечислить через запятую его элементы, заключив их в квадратные скобки:

fruits = ["апельсин", "банан", "ананас"]

В приведенном выше коде создается список fruits, содержащий 3 строковых объекта.

Нетрудно заметить, что, чем больше будет элементов, тем сложнее и неудобнее использовать отдельные переменные для их хранения и обработки. В таких случаях списки и приходят на помощь.

Списки позволяют хранить объекты разных типов:

strings = ["Leonid", "Denis"]

numbers = [50, 48, 94, 14]

mix = ["Leonid", 20, "Denis", 30, 52.37, "winners"]

Здесь список strings содержит строки, numbers — числа, а mix — объекты разных типов вперемешку.

Также внутри списка могут содержаться другие списки:

doublist = [6, [5, 7], ["banana", 10], "orange"]

В этом примере список doublist содержит и число 6, и строку "orange", и два других списка — [5, 7] и ["banana", 10].

Для создания пустого списка можно записать пустые квадратные скобки или воспользоваться функцией list():

empty\_list = []

empty\_list = list()

А вывести содержимое списка на экран можно с помощью все той же функции print().

Данный код:

empty\_list = []

fruits = ["orange", "apple", "banana"]

print(empty\_list)

print(fruits)

выводит

[]

['orange', 'apple', 'banana']

Также по аналогии с функциями int() и str(), переводящими в целочисленный и строковый тип данных соответственно, функция list() может преобразовывать коллекции (строки / множества) к типу данных list. При этом каждый элемент коллекции станет отдельным элементом списка.

Этот код:

string = "Leo"

setter = {"e", "L", 0}

print(list(string))

print(list(setter))

может вывести

['L', 'e', 'o']

[0, 'L', 'e']

Обратите внимание на то, что порядок элементов множества setter в получившемся списке может быть иным, как в примере выше. Это связано с тем, что элементы во множестве не упорядочены.

## Генераторы списка

В более сложных случаях используют генераторы списков – выражения, напоминающие цикл, с помощью которых заполняются элементы вновь созданного списка:

a = [i for i in range(10)]

Как вы знаете, цикл for i in range(10) перебирает все значения i от 0 до 9. Выражение перед словом for (в данном случае – i) – это то, что записывается в очередной элемент списка для каждого i. В приведенном

примере список заполняется значениями, которые последовательно при- нимает переменная i, то есть получим такой список:

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

То же самое можно получить, если использовать функцию list для того, чтобы создать список из данных, которые получаются с помощью функции range:

а = list (range(10))

Для заполнения списка квадратами этих чисел можно использовать такой генератор:

а = [i\*i for i in range(10)] #[0,1,4,9,16,25,36,49,64,81]

В конце записи генератора можно добавить условие отбора. В этом случае в список включаются лишь те из элементов, перебираемых в цикле, кото- рые удовлетворяют этому условию.

Например, следующий генератор составляет список из всех четных чисел в диапазоне от 0 до 9:

а = [i for i in range(10) if i%2 ==0] #[0,2,4,6,8]

Пусть word = 'Hello', numbers = [1, 14, 5, 9, 12], words = ['one', 'two', 'three', 'four', 'five', 'six'].

| Списочное выражение | Результирующий список |
| --- | --- |
| [0 for i in range(10)] | [0, 0, 0, 0, 0, 0, 0, 0, 0, 0] |
| [i \*\* 2 for i in range(1, 8)] | [1, 4, 9, 16, 25, 36, 49] |
| [i \* 10 for i in numbers] | [10, 140, 50, 90, 120] |
| [c \* 2 for c in word] | ['HH', 'ee', 'll', 'll', 'oo'] |
| [m[0] for m in words] | ['o', 't', 't', 'f', 'f', 's'] |
| [i for i in numbers if i < 10] | [1, 5, 9] |
| [m[0] for m in words if len(m) == 3] | ['o', 't', 's'] |

## Операции над списками

Рассмотрим основные операции, которые можно производить над списками.

### Поиск длины

Количество элементов, содержащихся в списке, называется его длиной. Для определения длины списка используется встроенная в Python функция len().

Приведённый ниже код:

squares = [1, 4, 9, 16, 25, 36, 49, 64]

months = ["December", "January", "March", "April", "May"]

print(len(squares))

print(len(months))

выводит

8

5

### Конкатенация списков

Конкатенация — это операция соединения двух или более списков в один. В Python она определяется оператором +.

Данный код:

pets = ["cat", "dog", "cow"]

predators = ["tiger", "lion", "lynx"]

animals = pets + predators

print(animals)

объединяет содержимое списков pets и predators в один список animals и выводит

['cat', 'dog', 'cow', 'tiger', 'lion', 'lynx']

### Дублирование списков

С помощью оператора \* в Python можно умножать список на число, тем самым дублируя его содержимое определенное количество раз.

Данный код:

numbers = [5, 7, 9]

print(numbers \* 3)

letters = ["a", "b", "c"]

letters \*= 2

print(letters)

non\_empty\_list = [1, 1, 1]

print(non\_empty\_list \* 0)

выводит

[5, 7, 9, 5, 7, 9, 5, 7, 9]

['a', 'b', 'c', 'a', 'b', 'c']

[]

Обратите внимание на то, что результатом умножения списка на ноль является пустой список.

### Сравнение списков

Списки можно сравнивать между собой с помощью операторов == и !=.

Два списка считаются равными, если они имеют одинаковую длину и полностью идентичны друг другу. Сравнение списков происходит поэлементно, и если каждый элемент в первом списке равен элементу в другом списке, тогда списки считаются равными.

Данный код:

print([5, 2, 3] == [5, 2, 3])

print([1, 1, 1] == [1, 1, 2])

print([6, 7] != [6, 7, 4, 9])

выводит

True

False

True

### Поиск минимума и максимума

С помощью встроенных в Python функций min() и max() можно получить минимальное и максимальное значения списка соответственно.

Данный код:

numbers = [9, 14, 2, 167, 311, 52, 1094]

print(max(numbers))

print(min(numbers))

выводит

1094

2

Обратите внимание на то, что если в списке будут присутствовать объекты разных типов, тогда попытка использовать функции min() и max() приведет к ошибке, потому что Python не знает как, например, сравнивать число со строкой.

Этот код:

elements = [5, 7, "Bugatti"]

print(max(elements))

завершится с ошибкой

TypeError: '>' not supported between instances of 'str' and 'int'

Функции max() и min() также можно использовать для поиска максимального и минимального значения среди чисел. Для этого нужно перечислить все сравниваемые значения через запятую в качестве аргументов функции.

Приведенный ниже код:

a = 10

b = 20

c = 7

print(min(a, b, c))

print(max(a, b, c))

выводит

7

20

### Поиск суммы

Для вычисления суммы всех элементов списка можно использовать функцию sum(), но прежде обязательно нужно убедиться в том, что все объекты, содержащиеся в списке, относятся к типам данных int или float.

Данный код:

measurements = [14, 15.5, 17.3, 2.7, 10, 30, 10.5]

print(sum(measurements))

выводит

100.0

### Оператор in

С помощью встроенного в Python оператора in можно определить вхождение какого-то элемента в список.

Этот код:

print(1 in [9, 6, 1, 8, 1, 4, 5, 1])

print(3 in [2, "string", 33])

выводит

True

False

Иногда удобно использовать оператор in в связке с not. В таком случае мы проверяем не наличие элемента в списке, а его отсутствие.

Этот код:

print("kekw" not in ["lol", "kekw", "lmao"])

print(7 not in [12, 13, 14])

выводит

False

True

Индексация и срезы

В [прошлом уроке](https://stepik.org/lesson/1367930/step/1?unit=1383968) мы познакомились с основными операциями над списками. Пришла пора разобраться со структурой списков подробнее и научиться их более гибко обрабатывать.

## Индексация

Список — это изменяемая упорядоченная коллекция элементов, каждый из которых имеет определенный номер (индекс).

Элементы в списках нумеруются слева направо в порядке возрастания, начиная с нуля.

Данный список

a = [10, 15, 13, -4, 0, 1]

состоит из 66 элементов, имеющих индексы (номера) от 0 до 5:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Элемент | 10 | 15 | 13 | -4 | 0 | 1 |
| Индекс | 0 | 1 | 2 | 3 | 4 | 5 |

А этот список

arr = [-5, 13, 'World', 14, 'Podgotovka', 'k', 'EGE', 'na', 100, [12, 14], [1, 2, 3, 4]]

состоит из 1111 элементов, которые индексируются так:

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Элемент | -5 | 13 | 'World' | 14 | 'Podgotovka' | 'k' | 'EGE' | 'na' | 100 | [12, 14] | [1, 2, 3, 4] |
| Индекс | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

Первый элемент любого списка всегда находится под индексом 00, а последний элемент имеет индекс, на единицу меньший длины списка. В примерах выше в списках с длинами 66 и 1111 последние элементы имеют индексы 55 и 1010 соответственно.

Обратите внимание на то, что под индексом 44 находится целиком строка 'Podgotovka', а под индексом 1010 — список [1, 2, 3, 4].

Каждый элемент списка можно получить, обратившись к нему по конкретному индексу. Для этого необходимо указать коллекцию, к элементу которой нужно обратиться, а рядом в квадратных скобках [] указать индекс элемента.

Этот код:

data = [15, 12345, 'string', 14, ['list', 123]]

print(data[0])

print(data[2])

print(data[4])

выводит

15

string

['list', 123]

Нумерация элементов в списке поддерживается также и в обратном порядке (справа налево), причем самый крайний элемент будет иметь индекс, равный −1−1.

Элементы этого списка при отрицательной индексации

data = [15, 12345, 'string', 14, ['list', 123]]

будут иметь следующие номера:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Элемент | 15 | 12345 | 'string' | 14 | ['list', 123] |
| Индекс | -5 | -4 | -3 | -2 | -1 |

Последний элемент списка в такой нумерации всегда можно получить по индексу −1−1, а первый элемент — по индексу, равному длине списка с отрицательным знаком.

Данный код:

data = [15, 12345, 'string', 14, ['list', 123]]

print(data[-1])

print(data[-3])

print(data[-4])

выводит

['list', 123]

'string'

12345

Обратите внимание на то, что при попытке обратиться к элементу под несуществующим индексом произойдет ошибка.

Эта программа:

data = [15, 12345, 'string', 14, ['list', 123]]

print(data[5])

завершится с ошибкой

IndexError: list index out of range

по причине того, что в списке data всего 55 элементов, последний из которых имеет индекс, равный 44.

Примечание. К последнему элементу списка удобно обращаться через отрицательную индексацию (−1−1), но ещё его можно вычислить универсально для списка arr как len(arr) - 1.

С элементами списка можно работать как с отдельными объектами в зависимости от их типа данных.

Например, код:

data = [123, 'Example', 456]

print(data[0] + data[2])

print(data[1][3])

выводит

579

m

Пояснение ко второму примеру: data[1] обращается к строке 'Example', далее происходит обращение к символу этой строки с индексом 3.

## Обход списков

С помощью цикла for можно поэлементно «обойти» весь список. Для перебора всех индексов достаточно передать в функцию range() длину списка, поскольку первый элемент в списке имеет индекс 0, а последний — len(список) - 1. Функция range(len(список)) как раз генерирует необходимую последовательность целых чисел.

Данный код:

my\_hobbies = ['computer games', 'tennis', 'sleep']

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

print(i, my\_hobbies[i])

выводит

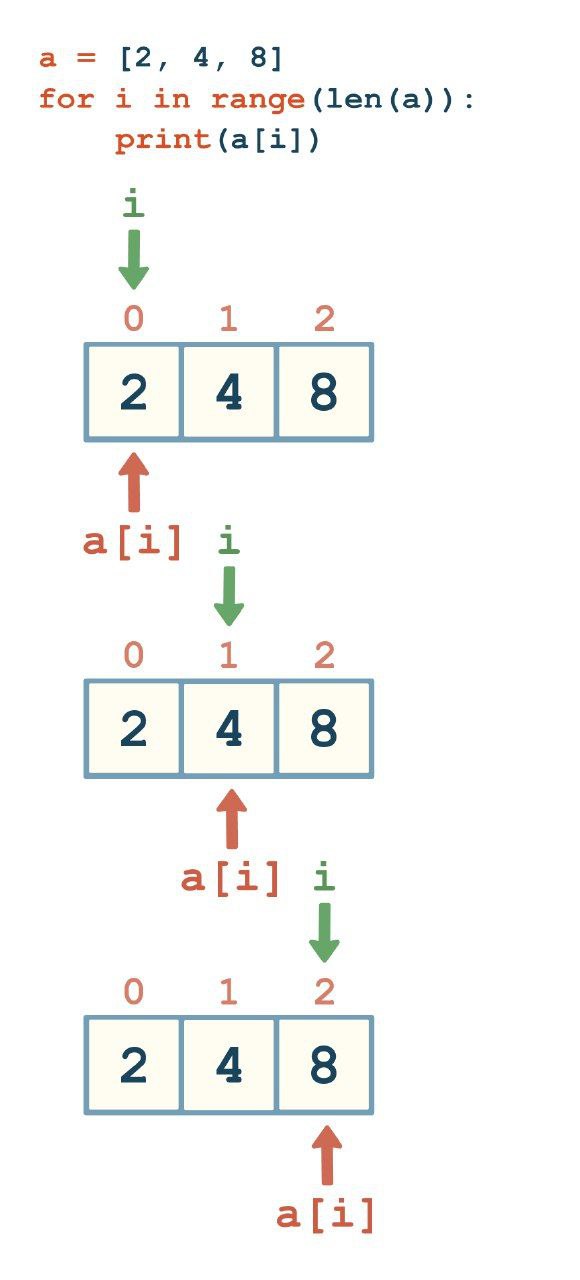
0 computer games

1 tennis

2 sleep

Здесь переменная i последовательно будет принимать значения индексов всех элементов списка.

Ниже проиллюстрирован принцип обхода списка с использованием индексации:



Также цикл for позволяет последовательно перебирать элементы коллекции.

Этот код:

random\_values = [4, 13, 'some text', [13, 12, 431, 'abcXYZ', 18], 11]

for element in random\_values:

print(element)

выводит

4

13

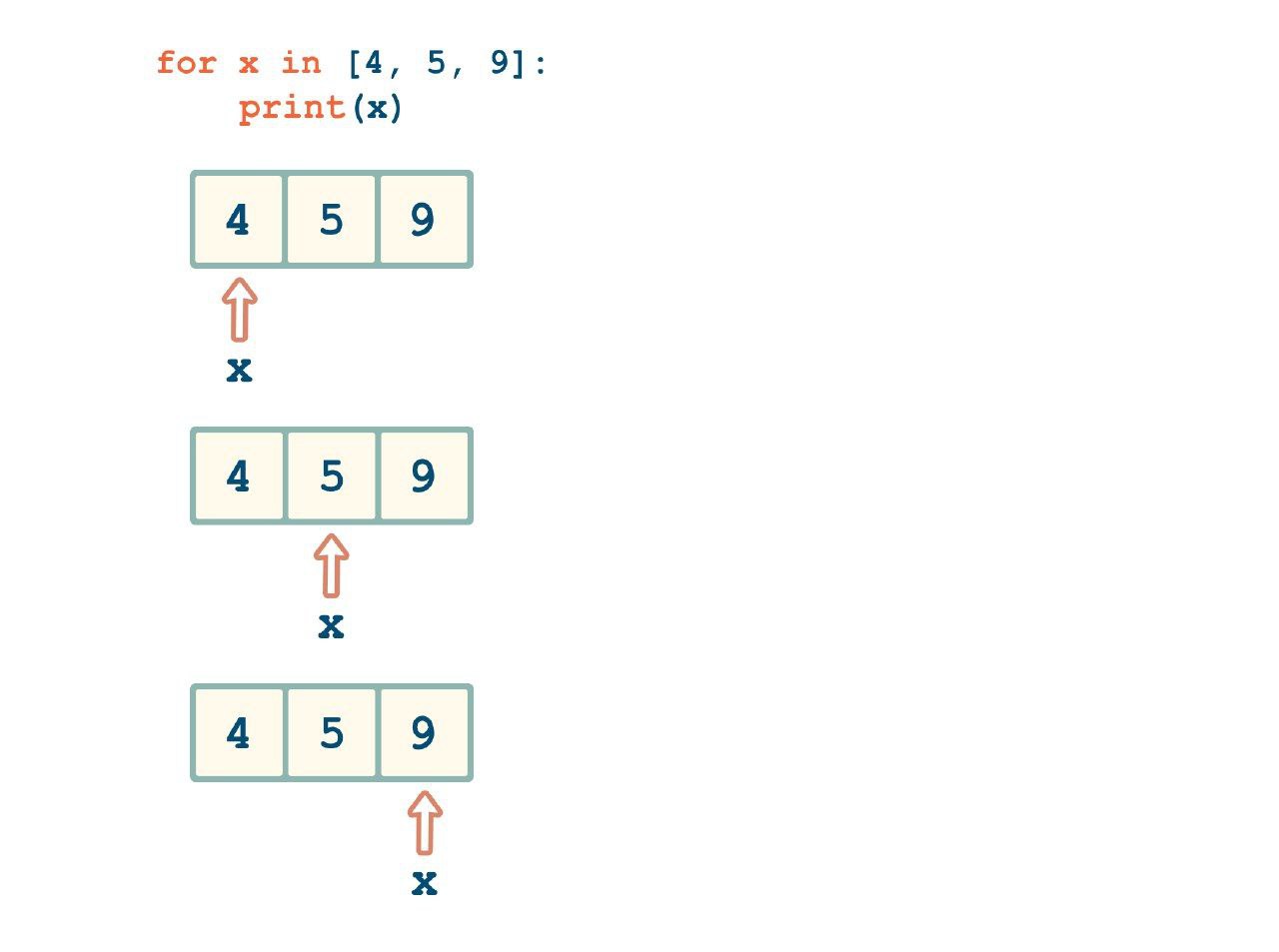
some text

[13, 12, 431, 'abcXYZ', 18]

11

В нём переменная element последовательно будет ссылаться на каждый элемент списка от первого и до последнего. Отличие этого способа обхода списка от предыдущего заключается в том, что в первом случае есть возможность одновременно взаимодействовать как с элементом, так и с его индексом.

Ниже проиллюстрирован принцип обхода строки без использования индексации:



## Срезы

В некоторых случаях возникает необходимость получить целый фрагмент списка, состоящий из нескольких элементов. Для решения этой задачи удобно использовать срезы, которые позволяют выделить сразу несколько идущих подряд элементов.

Упрощённый синтаксис среза выглядит следующим образом:

list[start:stop]

Здесь start и stop задают соответственно начальный и конечный индексы списка. При таком срезе выделяются элементы списка от индекса start до индекса stop **не** включительно.

Рассмотрим на примере списка persona = ['Denis', 'Bakhtiev', 33, 'gymnasium', 'id\_vk', 159, 260, 657]. Элементы в нем индексируются так:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Индекс (+) | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| Элемент | 'Denis' | 'Bakhtiev' | 33 | 'gymnasium' | 'id\_vk' | 159 | 260 | 657 |
| Индекс (-) | -8 | -7 | -6 | -5 | -4 | -3 | -2 | -1 |

Срез persona[4:8] вернёт все элементы, начиная с индекса 44 и заканчивая индексом 88 не включительно: ['id\_vk', 159, 260, 657].

А срез persona[0:4] вернёт первые четыре элемента: ['Denis', 'Bakhtiev', 33, 'gymnasium'].

Также можно использовать отрицательную индексацию: срез persona[-6:-4] вернёт [33, 'gymnasium'].

Как и при работе со строками, срезы в списках никогда не приведут к ошибке индексации.

Данный код:

persona = ['Denis', 'Bakhtiev', 33, 'gymnasium', 'id\_vk', 159, 260, 657]

print(persona[5:999999])

выводит

[159, 260, 657]

Если же в указанном срезе нет ни одного элемента, то выведется пустой список.

Код:

persona = ['Denis', 'Bakhtiev', 33, 'gymnasium', 'id\_vk', 159, 260, 657]

print(persona[5000:10000])

выводит

[]

Допустимо не указывать start среза, тогда будет взят фрагмент от начала списка и до значения stop. Срез persona[:5] эквивалентен срезу persona[0:5] и возвращает ['Denis', 'Bakhtiev', 33, 'gymnasium', 'id\_vk'].

Аналогично можно не указывать stop, тогда будет взят фрагмент от значения start и до конца списка. Срез persona[2:] эквивалентен срезу persona[2:8] и возвращает [33, 'gymnasium', 'id\_vk', 159, 260, 657].

Срез также допускает указание шага. Общий синтаксис среза выглядит так:

list[start:stop:step]

Здесь start и stop задают соответственно начальный и конечный индексы списка, а step определяет шаг выделения элементов. При таком срезе выделяются элементы списка от индекса start до индекса stop **не** включительно с шагом step.

Данный код:

data = [34, 11, 12, 2, 54, -2, -643, 11]

print(data[2:5:1])

print(data[0:4:1])

выводит

[12, 2, 54]

[34, 11, 12, 2]

По умолчанию start = 0, stop = len(список), а step = 1.

Так, срез data[:4] полностью идентичен срезу data[0:4:1], а срез data[5:-1:1] полностью идентичен срезу data[5:-1].

В некоторых случаях приходится брать каждый 2-й, 3-й или *n*-й элемент некоторого списка, тогда важна настройка шага.

Этот код:

data = [34, 11, 12, 2, 54, -2, -643, 11]

print(data[2:6:2])

print(data[1:7:3])

print(data[0:-1:4])

выводит

[12, 54]

[11, 54]

[34, 54]

Также можно использовать отрицательный шаг среза. В этом случае элементы будут выводиться в обратном порядке.

Данный код:

persona = ['Denis', 'Bakhtiev', 33, 'gymnasium', 'id\_vk', 159, 260, 657]

print(persona[3:1:-1])

print(persona[7:3:-1])

выводит

['gymnasium', 33]

[657, 260, 159, 'id\_vk']

Чаще всего срезы используются для взятия первых или последних n элементов списка.

Удобно запомнить, что:

* срез data[:n] возвращает первые n элементов списка data
* срез data[-n:] возвращает последние n элементов списка data
* срез data[n:] возвращает все элементы списка, начиная с элемента с индексом n
* срез data[:-n] возвращает все элементы от начала списка и до элемента с индексом -n - 1
* срез data[a:b] возвращает все элементы списка, начиная с элемента с индексом a и заканчивая элементом с индексом b - 1

Ещё два интересных сценария использования списков:

* срез data[:] возвращает поверхностную копию списка, опуская определение значений start, stop и step
* срез data[::-1] возвращает перевернутый (задом наперед) список.

Данный код:

data = [1, 2, 34, -14, 5]

print(data[:])

print(data[::-1])

выводит

[1, 2, 34, -14, 5]

[5, -14, 34, 2, 1]

## Изменение списков

При работе со списками важно помнить о том, что они относятся к изменяемому типу данных, а потому, в отличие от строк, списки изменять можно. Для того, чтобы изменить список необходимо по индексу обратиться к элементу, который мы хотим заменить, и указать новое значение.

Приведенный ниже код:

data = ['green', 'red', 13, 'yellow', 'white']

data[2] = 'BLACK'

print(data)

выводит

['green', 'red', 'BLACK', 'yellow', 'white']

Рассмотрим интересное проявление списков, связанное с их изменяемостью.

Данный код:

a = ['raz', 'dva', 'tri']

b = a

print(b)

a[0] = 1

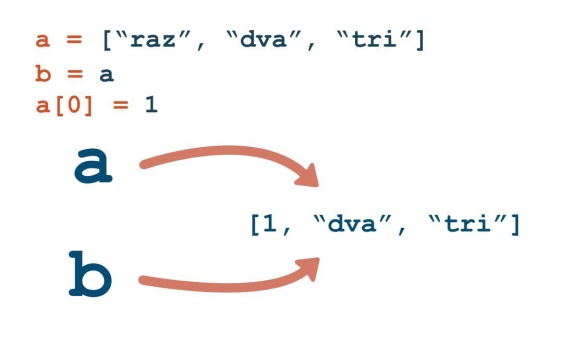
print(b)

выводит

['raz', 'dva', 'tri']

[1, 'dva', 'tri']

Несмотря на то, что переменную b мы вообще не трогали, а меняли список a, в конце программы при обращении к переменной b мы обращаемся к уже изменённому списку:



Для правильного создания копии списка можно воспользоваться срезом.  
Данный код:

a = ['raz', 'dva', 'tri']

b = a[:]

print(b)

a[0] = 1

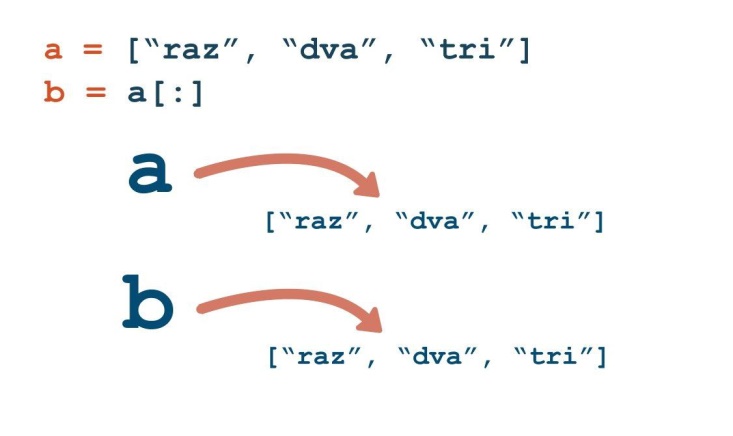
print(b)

выводит

['raz', 'dva', 'tri']

['raz', 'dva', 'tri']

Инструкция b = a[:] создаёт **новый** список со значениями, как у списка a, поэтому при изменении списка a список b не меняется:



## Методы списков

Ранее мы говорили, что метод в Python — это специализированная функция, применяемая к определённым объектам. Из таких объектов мы уже знаем строки и списки. Подробнее поговорим про списочные методы: что они делают и когда применяются.

Синтаксис применения метода к списку:

список.метод(аргументы)

В таблице ниже приведены популярные в ЕГЭ методы списков:

|  |  |
| --- | --- |
| Название метода | Описание |
| append() | добавляет новый элемент в список |
| count() | подсчитывает количество вхождений указанного элемента в список |
| index() | находит индекс указанного элемента в списке |
| remove() | удаляет элемент из списка |
| sort() | сортирует список |
| reverse() | переворачивает список |

### Метод append()

Метод append(object) добавляет в конец списка указанный объект object.

Данный код:

data = [13, 14, 7]

data.append(10)

print(data)

выводит

[13, 14, 7, 10]

Обратите внимание на вторую строчку кода. Здесь не требуется переприсваивание списка, как необходимо делать со строками, потому что списки являются изменяемыми, и метод append() реально меняет существующий список.

### Метод count()

Метод count(value) возвращает число вхождений указанного значения value в указанный список.

Этот код:

arr = [1, 2, 3, 4, 1, 5, '1']

num\_1 = arr.count(1) # ищем числовые единички

str\_1 = arr.count('1') # ищем строковые единички

num\_6 = arr.count(6) # ищем числовые шестёрки

print(num\_1)

print(str\_1)

print(num\_6)

выводит

2

1

0

Обратите внимание на то, что поиск переданного элемента выполняется с учетом его типа данных.

### Метод index()

Метод index(value) возвращает индекс, на котором находится значение value в указанном списке. Если таких значений несколько, то возвращается индекс первого слева элемента.

Данный код:

arr = [1, 2, 3, 4, 1, 5, '1']

print(arr.index('1'))

print(arr.index(1))

print(arr.index(4))

выводит

6

0

3

Особенность работы этого метода заключается в том, что если искомое значение не найдется в списке, то вернётся ошибка.

Этот код:

arr = [1, 2, 3, 4, 1, 5, '1']

print(arr.index(15))

завершится с ошибкой

ValueError: 15 is not in list

В тех случаях, когда точно неизвестно, содержится ли искомый элемент в списке, можно воспользоваться проверкой с помощью уже известного нам оператора in.

Данный код:

arr = [1, 2, 3, 4, 1, 5, '1']

if 15 in arr:

print(arr.index(15))

else:

print('Искомого числа в списке нет')

проверяет, содержится ли число 15 в списке arr и выводит

Искомого числа в списке нет

### Метод remove()

Метод remove(value) удаляет из списка переданное значение value. Если таких значений в списке несколько, то удалится первое слева значение.

Приведенный ниже код:

arr = [1, 2, 3, 4, 1, 5, '1']

arr.remove(2) # удаляем число 2

print(arr)

arr.remove('1') # удаляем строку '1'

print(arr)

arr.remove(1) # удаляем число 1

print(arr)

выводит

[1, 3, 4, 1, 5, '1']

[1, 3, 4, 1, 5]

[3, 4, 1, 5]

Обратите внимание на то, что если попробовать удалить элемент, которого нет в списке, то программа завершится с ошибкой.

Данный код:

arr = [1, 2, 3, 4, 1, 5, '1']

arr.remove(10)

print(arr)

выводит

ValueError: list.remove(x): x not in list

Если необходимо удалить все вхождения какого-то элемента, притом неизвестно, сколько раз он встречается в списке, то можно воспользоваться циклом while. Данный код:

a = [1, 2, 1, 3, 2, 1, 1, 1, 4, 3, 2, 1]

while 1 in a:

a.remove(1)

print(a)

выводит

[2, 3, 2, 4, 3, 2]

Приведённый выше код также правильно отработает, если переданного значения value не будет в списке вообще.

### Метод sort()

Метод sort() сортирует по возрастанию элементы исходного списка. Его удобно применять к спискам, которые заполнены именно числовыми значениями.

Приведенный ниже код:

a = [123, 432, 231, 7, -4, 98]

a.sort()

print(a)

отсортирует элементы списка a и выведет

[-4, 7, 98, 123, 231, 432]

Если необходимо отсортировать элементы списка по убыванию, то можно использовать необязательный параметр reverse со значением True.

Код:

a = [123, 432, 231, 7, -4, 98]

a.sort(reverse=True)

print(a)

выводит

[432, 231, 123, 98, 7, -4]

Обратите внимание на то, что если не все данные в списке будут приведены к числовому значению, то программа завершится с ошибкой. Данный код:

a = [1, '4', 2]

a.sort()

print(a)

выдаст ошибку

TypeError: '<' not supported between instances of 'str' and 'int'

### Метод reverse()

Метод reverse() переворачивает список, к которому он применяется.

Приведенный ниже код:

a = [123, 432, 231, 7, -4, 98]

a.reverse()

print(a)

перевернёт список a и выведет:

[98, -4, 7, 231, 432, 123]

## Особенность работы методов

Важно помнить о том, что списки являются изменяемыми объектами, а потому списочные методы append(), remove() , sort() и reverse() изменяют исходный список, в отличие от любых строковых методов.

В данном коде:

array = [7, 5, 2, 10]

print(array)

array.append(666)

print(array)

array.remove(5)

print(array)

array.sort()

print(array)

array.reverse()

print(array)

каждый применённый метод изменяет список, на который ссылается переменная array. На выводе получится

[7, 5, 2, 10]

[7, 5, 2, 10, 666]

[7, 2, 10, 666]

[2, 7, 10, 666]

[666, 10, 7, 2]

Таким образом, перезаписывать переменную array после применения каждого из этих методов не нужно.

Если всё-таки попробовать перезаписать значение, то можно получить «неожиданный» вывод.

Программа:

a = [1, 2, 3]

a = a.append(4)

print(a)

выводит

None

На самом деле такой вывод абсолютно логичный. О том, почему так получается, мы поговорим в модуле «Функции» данного курса.

Работа с файлами

Для решения некоторых задач из ЕГЭ по информатике требуется владеть основами работы с файлами. В этом уроке мы научимся открывать и закрывать текстовые файлы и обрабатывать хранящиеся в них данные на Python.

## Файлы с расширением txt

Файл txt — это простейший текстовый файл, который хранит обычный текст.

Так выглядит текстовый файл на Windows 10:



Приведенный выше файл имеет имя — Python и расширение — .txt.

Название любого файла состоит из его имени и расширения. Расширение файла указывает на его формат. Например, расширения .jpg или .png говорят о том, что файл является изображением, а расширение .mp3 указывает на формат аудио. В рамках ЕГЭ мы будем работать только с текстовыми файлами с расширением .txt.

Расширение не всегда отображается в названии, это зависит от настроек операционной системы, однако оно всегда существует и определяет формат конкретного файла.

## Открытие файла

Функция open(file) по умолчанию создает файловый объект для чтения данных по указанному пути и имени file.

Путь к файлу передается в качестве аргумента функции open().

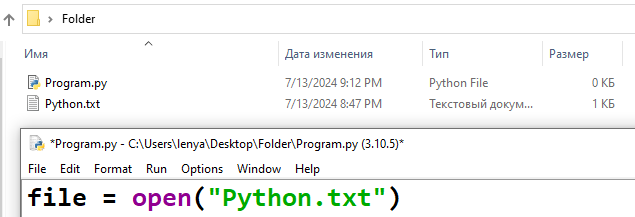
Если искомый файл находится в той же папке, где и наша программа с расширением .py (из которой мы открываем файл), тогда достаточно в функцию open() передать только название файла (в формате имя.расширение).

Этот код:

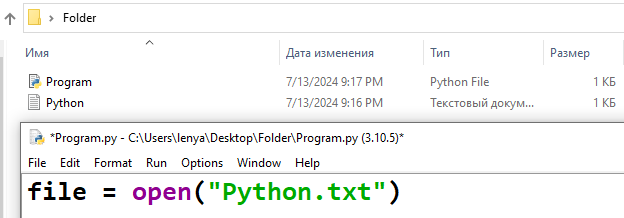
file = open("Python.txt")

открывает на чтение файл Python.txt и записывает ссылку на него в переменную file.

При этом сам текстовый файл лежит в одной папке с «питоновским»:



Если в настройках операционной системы отключены расширения файлов, тогда их имена будут выглядеть иначе:

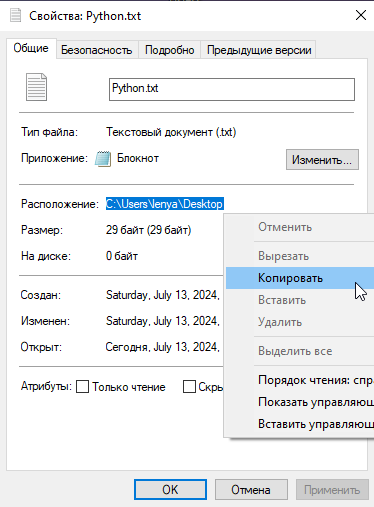


При этом расширение файла .txt всё равно нужно указывать, иначе программа завершится с ошибкой.

Но если бы искомый файл находился в другой папке, тогда данная программа завершилась бы с ошибкой:

FileNotFoundError: [Errno 2] No such file or directory: 'Python.txt'

В таком случае нужно указывать полный путь к файлу, для чего можно открыть его свойства и скопировать расположение папки, в которой он находится:



После чего останется дописать к получившемуся пути имя файла и получить C:\Users\lenya\Desktop\Python.txt, что и будет являться полным путем к файлу.

Символ \ в языке Python имеет особый функционал, поэтому использовать его в путях к файлу не рекомендуется. Один из примеров, когда использование обратного слэша приводит к «неожиданному результату»: допустим, что полный путь к файлу — C:\Users\new\_folder\Python.txt. Здесь после каталога Users следуют символы \ и n, которые прочитаются как единый символ \n, в связи с этим осуществится перенос текста на новую строку. Вместо \ допустимо использовать обычный слэш /, с которым проблем уже не возникнет.

Данный код:

file = open("C:/Users/lenya/Desktop/Python.txt")

открывает на чтение файл Python.txt и записывает ссылку на него в переменную file, при этом сам текстовый файл лежит в другой папке.

## Чтение из файла

Существуют разные способы чтения данных из файла, рассмотрим самые удобные из них (в рамках ЕГЭ).

### Метод readline()

Метод readline() считывает одну строку из файла вместе со знаком переноса \n, который и указывает на конец строки.

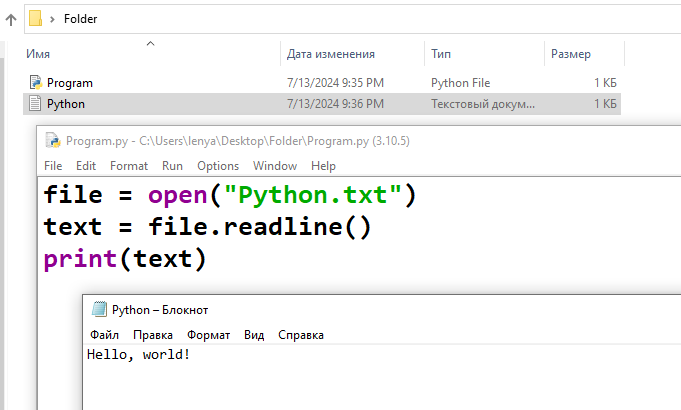
Этот код:

file = open("Python.txt")

text = file.readline()

print(text)

открывает текстовый файл Python.txt



и считывает из него строку текста, записывая её в переменную text, и выводит

Hello, world!

Метод readline() удобно использовать тогда, когда количество содержащихся в файле строк точно известно, поскольку операцию считывания можно применить определенное количество раз с помощью цикла for до тех пор, пока не будут прочитаны все строки. При этом очередное применение метода readline() считывает следующую необработанную строку и никогда не возвращается к уже прочитанным строкам.

Данный код:

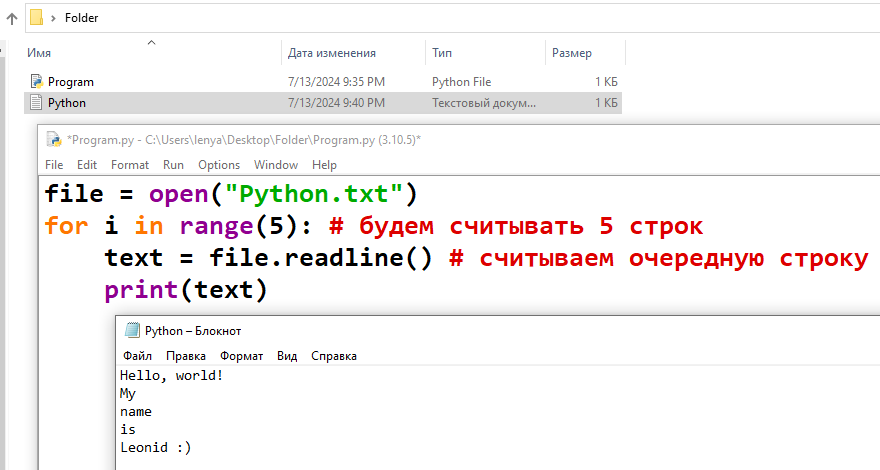
file = open("Python.txt")

for i in range(5): # будем считывать 5 строк

text = file.readline() # считываем очередную строку

print(text)

считывает все строки из файла



и выводит

Hello, world!

My

name

is

Leonid :)

А если известно, что в файле содержится всего одна строка, прочитать её можно так:

line = open("file\_name.txt").readline()

В приведенном выше коде переменная line будет ссылаться сразу на прочитанную строку.

### Построчное считывание с помощью цикла for

Можно использовать построчное считывание с помощью цикла for, которое особенно удобно тогда, когда заранее неизвестно количество обрабатываемых строк. При таком подходе последовательно будут считываться строки, пока не будет достигнут конец файла.

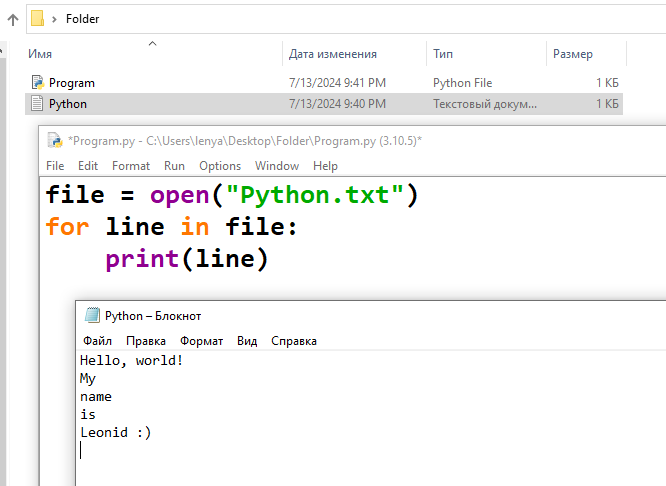
Этот код:

file = open("Python.txt")

for line in file:

print(line)

открывает файл



и последовательно выводит из него все строки

Hello, world!

My

name

is

Leonid :)

В приведенной выше программе переменная line последовательно ссылается на каждую очередную строчку из файла. В самом конце (после цикла for) она будет ссылаться на последнюю строчку.

### Сохранение прочитанных данных

Зачастую нужно не просто прочитать строки из файла, но еще и где-то их сохранить для дальнейшей работы.

Данный код:

file = open("Python.txt")

data = []

for line in file:

data.append(line)

print(data)

считывает все строки из файла, сохраняет их в список data и выводит

['Hello, world!\n', 'My\n', 'name\n', 'is\n', 'Leonid :)']

Обратите внимание на то, что в конце каждой прочитанной строки содержится знак переноса \n, избавиться от которого можно с помощью уже известного нам метода strip().

При чтении чисел из файла важно не забыть преобразовать их к целочисленному типу данных, ведь изначально они считываются как строки.

Этот код:

file = open("numbers.txt")

data = []

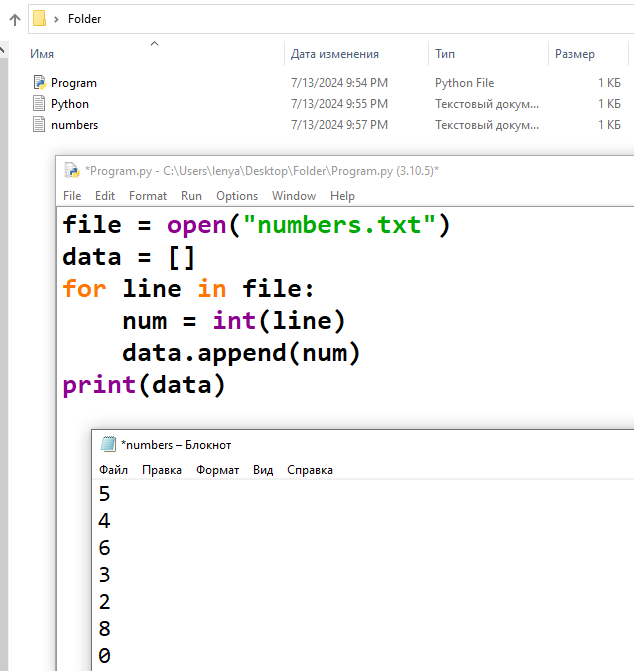
for line in file:

num = int(line)

data.append(num)

print(data)

последовательно считывает все числа из файла numbers.txt.



и сохраняет их в список data, прежде преобразуя к целому типу с помощью функции int(). И выводит:

[5, 4, 6, 3, 2, 8, 0]

С сохраненным списком чисел теперь можно спокойно работать. Например, можно посчитать среднее арифметическое всех прочитанных из файла чисел. Этот код:

file = open("numbers.txt")

data = []

for line in file:

num = int(line)

data.append(num)

s = sum(data)

c = len(data)

avg = s / c

print(avg)

Находит сумму и количество сохраненных чисел, вычисляет их среднее арифметическое и выводит:

4.0

Обратите внимание на то, что при преобразовании строки к целому числу с помощью функции int() использование метода strip() необязательно. Например, int('27\n') преобразует строку в число 27.

### Закрытие файла

После чтения данных из файла рекомендуется его закрыть для высвобождения памяти и сохранения данных.

Для этого используется метод close().

Данный код:

file = open("numbers.txt")

data = []

for line in file:

num = int(line)

data.append(num)

file.close() # закрываем файл

сначала открывает файл и сохраняет все данные из него в список data, а затем закрывает его с помощью метода close().

После закрытия файла можно работать со всеми прочитанными данными, ведь они уже сохранены в список data.

В рамках ЕГЭ не приходится изменять содержимое файлов или работать с несколькими файлами одновременно, потому закрывать файл необязательно.

Функции

**Функция** в программировании (или подпрограмма) — фрагмент программного кода, к которому можно обратиться из другого места программы. Под функциями скрывается блок кода.

Рассмотрим фрагмент программы, в котором осуществляется поиск максимального числа в списке:

# arr — какой-то заданный список чисел

max\_element = arr[0]

for x in arr:

if x > max\_element:

max\_element = x

В данной программе переменная max\_element после цикла будет ссылаться на самый большой элемент списка arr.

Примерно такой код и выполняется, когда вызывают функцию max() и передают ей в качестве аргумента список. То есть все функции языка Python — не какие-то волшебные инструменты для того или иного действия, но под каждой из них скрывается заранее написанная программа, скрытая от пользователя.

Ранее мы пользовались готовыми (встроенными) функциями, в этом же уроке мы научимся писать свои собственные.

## Функции с параметрами

Общий вид функции в Python:

def название\_функции(параметры):

тело функции

return значение

def — ключевое слово, которое определяет пользовательскую функцию.

Название функции — любое допустимое имя в Python. На него распространяются те же ограничения, что и на имена переменных.

Параметры — переменные, которые принимают данные, переданные при вызове функции.

Тело функции — блок кода, отвечающий за логику работы функции.

return — ключевое слово, которое показывает какое значение подставится на место вызова функции.

Перепишем функцию max() с учётом показанного выше синтаксиса:

def custom\_max(arr):

max\_element = arr[0]

for x in arr:

if x > max\_element:

max\_element = x

return max\_element

В данной программе мы создали функцию custom\_max, которая на вход принимает аргумент arr (мы подразумеваем, что это список). Эта функция вычисляет максимальный элемент списка. Если же вызвать функцию custom\_max(), то на место вызова подставится посчитанное значение max\_element.

В программе:

def custom\_max(arr):

max\_element = arr[0]

for elem in arr:

if elem > max\_element:

max\_element = elem

return max\_element

a = [10, 15, 7, -11, 4]

b = custom\_max(a)

print(b)

на экран будет выведено 15

В данной программе в строке b = custom\_max(a) вызывается написанная функция, которая принимает список a. Далее находится максимум, и в итоге переменная max\_element будет ссылаться на число 15. Это значение указано в инструкции с return, поэтому число 15 и подставится на место вызова функции. Другими словами описанная строка примет вид: b = 15.

Обратите внимание на то, что описание функции и вызов функции — разные вещи. После инструкции def идёт описание логики работы функции при её вызове (все известные нам функции: max(), len(), print() и т.д. тоже имеют заранее описанный принцип работы). Другими словами, функция не будет работать, пока её не вызовут. Вызов же самописной функции ничем не отличается от вызова встроенных функций: указание названия функции со скобками, содержащими аргументы.

## Параметры и аргументы

Пришло время поговорить о таких понятиях как параметры и аргументы и об их отличиях.

Рассмотрим код:

def custom\_max(arr):

max\_element = arr[0]

for elem in arr:

if elem > max\_element:

max\_element = elem

return max\_element

a = [10, 15, 7, -11, 4]

b = custom\_max(a)

print(b)

Аргументы — данные, которые передаются в функцию при её вызове. В примере выше — это список a.

Параметры — это переменные, которые принимают переданные в функцию аргументы. В примеры выше параметром является переменная arr. То есть при вызове функции custom\_max(a) переменная arr ссылается на список a. Если передать этой функции какой-то другой список, то переменная arr будет ссылаться уже на него.

Обратите внимание на то, что названия аргументов и параметров могут как совпадать, так и различаться, и это не влияет на работу программы. Рекомендуется называть их по-разному.

Рассмотрим следующий код:

def summator(a, b, c):

d = a + b + c

return d

x = 10

y = 20

z = 5

print(summator(3, 5, 8))

print(summator(x, y, z))

print(summator(11, z, x))

В данном коде параметрами являются a, b и с. При первом вызове функции аргументами являются 3, 5, 8; при втором — x, y, z и при третьем — 11, z, x. То есть какие бы значения мы не передали в функцию summator при её вызове, они будут записаны в параметры a, b, с соответственно.

Рассмотрим код:

def calculate(a, b, c):

d = a \* b + c

return d

a = 10

b = 20

c = 5

print(calculate(c, 7, a))

В данном коде при вызове функции calculate(c, 7, a) параметр a примет значение переменной c, то есть 5; параметр b примет значение числа 7, а параметр c примет значение переменной a, то есть 10. В итоге на экран выведется значение выражения 5 \* 7 + 10, то есть число 45. В данном примере название некоторых переменных совпадает с названием параметров функции calculate, но это неважно.

Количество переданных аргументов должно совпадать с количеством параметров функции. Данный код:

def function(a, b):

d = a \* b

return d

print(function(1, 2, 3))

завершится с ошибкой и выведет

TypeError: function() takes 2 positional arguments but 3 were given

Аналогично произойдёт ошибка, если аргументов будет меньше, чем параметров.

## Оператор return

Как мы уже знаем, значение, написанное после оператора return будет подставлено на место вызова функции. На самом деле это значение может быть чем угодно: числом, строкой, списком, другой или даже этой же функцией (подробнее о последнем случае поговорим в следующем уроке). Рассмотрим пример функции, которая принимает один аргумент — натуральное число и находит все его делители:

def dividers(n):

arr = []

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

if n % i == 0:

arr.append(i)

return arr

print(dividers(1))

print(dividers(7))

print(dividers(24))

Данный код выводит все делители для чисел 1, 7 и 24. Для каждого с новой строки. Притом сам результат работы функции — список делителей:

[1]

[1, 7]

[1, 2, 3, 4, 6, 8, 12, 24]

Важная особенность использования оператора return: выполнение инструкции с этим оператором завершает функцию.

Рассмотрим пример:

def f(a):

return a + a

return a \* a

print(f(5))

print(f(8))

в данном коде инструкция return a \* a никогда не выполнится. Поэтому на экран будут выведены числа 10 и 16.

Допустимо использовать несколько операторов return. Как правило это делается в сочетании с условным оператором if-else или в некоторых других случаях, например, связанных с циклами.

Рассмотрим функцию, которая определяет, является ли натуральное число простым:

def is\_prime(n):

if n == 1:

return False

for i in range(2, n):

if n % i == 0:

return False

return True

print(dividers(1))

print(dividers(7))

print(dividers(24))

Обратите внимание на то, что диапазон перебора чисел в цикле определён от 2 до *n* **не** включительно. Данный код работает по следующему принципу: если найдётся хоть один делитель из отрезка [2; 𝑛 −1], значит число непростое, выполнится инструкция return False, которая завершит работу функции. Если же в цикле не произойдёт выход из функции, то вернётся значение True. Отдельно прописан случай для 𝑛=1*n*=1, поскольку это число не является простым по определению.

Важно заметить, что допустимо не указывать инструкцию с оператором return. В таком случае функция всё равно вернёт значение, установленное по умолчанию.

Код:

def function(a, b):

d = a + b

print(function(3, 5))

выводит

None

Такие функции имеют смысл, как правило, при работе с изменяемыми объектами, но в рамках ЕГЭ не используются.

В Python наравне с int, str, list и прочими типами данных существует тип данных NoneType, единственным объектом которого является значение None.

## Особенности написания и использования функций

В начале данного урока мы сказали, а затем убедились, что функции — отдельные подпрограммы. В связи с этим для правильной работы с функциями необходимо понимать некоторые их особенности. Одна из них — невозможность обратиться к переменным функции из основной программы.

Рассмотрим код:

def square(n):

a = n \*\* 2

return a

number = int(input())

f = square(number)

print(f)

print(a)

В данной программе считывается число с клавиатуры, затем вызывается функция square(), которая возводит в квадрат переданное число и возвращает полученный результат, который, в свою очередь, записывается в переменную f. Таким образом при вводе с клавиатуры числа 15 в f запишется 225 и выведется инструкцией print(f) на экран. До этого момента программа отработает без ошибок. Но при попытке исполнения команды print(a) на экран будет выведено: NameError: name 'a' is not defined. Данная ошибка говорит о том, что переменной с именем a не существует в нашей программе, несмотря на то, что только что была вызвана функция, которая содержит переменную с таким именем. Дело здесь в области видимости переменных, о которой мы не будем подробно говорить в данном курсе. Сейчас важно просто запомнить, что никогда не нужно обращаться к переменным внутри функций из основной программы.

## Применение функций

Как видно из примеров выше, функции удобны, когда нам нужно один и тот же код применить для разных начальных условий. Например, если нужно определить, являются ли простыми множество разных чисел. Разумеется, можно каждый раз писать код для нахождения всех делителей, но тогда итоговая программа получится очень громоздкой и тяжело читаемой.

Рассмотрим пример использования функции в следующей задаче №17 из ЕГЭ:

*В файле содержится последовательность целых чисел. Её элементы могут принимать целые значения от 1 до 100 000 включительно. Определите количество троек элементов последовательности, в которых хотя бы один элемент имеет чётную сумму цифр. В ответе запишите количество найденных троек. В данной задаче под тройкой подразумевается три идущих подряд элемента последовательности.*

Код для решения данной задачи:

def sum\_dig(n): # функция для определения суммы цифр числа

summ = 0 # переменная для суммы цифр числа

for digit in str(n): # проходим по каждой цифре числа

summ += int(digit) # прибавляем значение очередной цифры

return summ # возвращаем результат

f = open('17.txt') # открываем файл

data = []

for s in f:

data.append(int(s)) # добавляем в список очередное число из файла

count = 0 # счётчик количества подходящих троек

for i in range(len(data) - 2):

summ\_1 = sum\_dig(data[i]) # сумма цифр для первого элемента тройки

summ\_2 = sum\_dig(data[i + 1]) # сумма цифр для второго элемента тройки

summ\_3 = sum\_dig(data[i + 2]) # сумма цифр для третьего элемента тройки

if summ\_1 % 2 == 0 or summ\_2 % 2 == 0 or summ\_3 % 2 == 0: # проверяем выполнение условия задачи

count += 1 # увеличиваем счётчик

print(count)

Рекурсивные функции

**Рекурсия** — определение, описание, изображение какого-либо объекта или процесса внутри самого этого объекта или процесса, то есть ситуация, когда объект является частью самого себя.

В [прошлом уроке](https://stepik.org/lesson/1367938/step/1?unit=1383980) мы говорили о том, что после ключевого слова return можно указывать любое значение. Это значение подставится на место вызова функции. Если функция в инструкции с return вызывает саму себя, то она называется рекурсивной.

Рассмотрим пример рекурсивной функции для вычисления факториала числа (факториалом числа 𝑛*n* называют произведение всех натуральных чисел от 1 до *n*). Для этого нам потребуются следующие рассуждения:

равенство *n*!=1∗2∗3∗...∗(*n*−1)∗*n* можно переписать в виде: *n*!=(*n*−1)!∗*n*, то есть зная значение (*n*−1)! мы легко сможем найти значение *n*!. В свою очередь значение (𝑛−1)!(*n*−1)! просто находится, если знать (*n*−2)!, которое легко находится, зная (*n*−3)!, и так далее. Таким образом, зная факториал предыдущего числа мы за одно действие находим факториал текущего. Находя факториал для всё меньшего числа мы в конечном итоге придём к значению 1!, которое равно 1.

Напишем рекурсивную функцию для реализации данного алгоритма:

def factorial(n):

if n == 1:

return 1

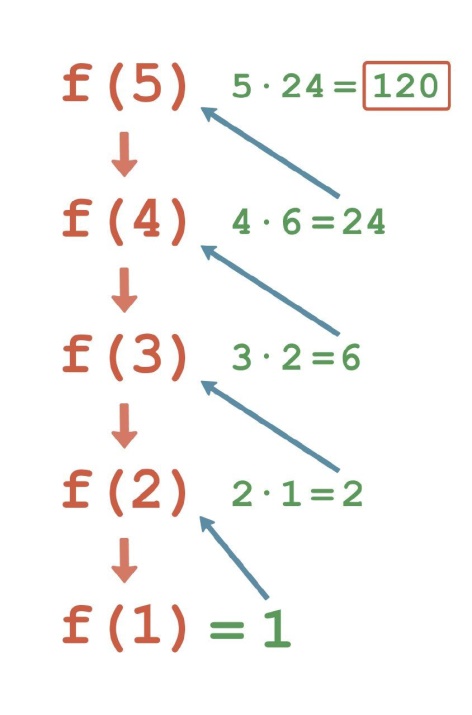
else:

return n \* factorial(n - 1)

Здесь при вызове, например, функции factorial(5) произойдёт вызов функции factorial(4), которая вызовет функцию factorial(3), которая вызовет функцию factorial(2), которая вызовет функцию factorial(1), которая вернёт единицу для последней из функций, а дальше по обратной цепочке будут подсчитаны значения факториалов для чисел от 2 до 5.

С учётом того, что return — выход из функции, оператор else в приведённой выше программе необязателен.

На картинке ниже вы можете увидеть последовательность вызовов функции и возврат результатов:



Каждый раз, вызывая внутри функции эту же функцию, мы как бы переходим на новый уровень. Количество таких вложенных вызовов без возврата значения называют глубиной рекурсии. Для примера выше глубина рекурсии равна 44.

Напишем ещё одну рекурсивную функцию. В данном случае она будет возводить переданное число в указанную степень.

def power(num, pow):

if pow == 1:

return num

return num \* power(num, pow - 1)

print(power(2, 10))

print(power(3, 5))

Для нахождения числа в степени 𝑛*n* мы это число умножаем на это же число в степени *n*−1. Например, 54=5∗53. При достижении единичной степени возвращаем само число. Для примера с пятёркой: 51=5.

Приведённый выше код выводит

1024

243

Обратите внимание на то, что если не прописать выход из рекурсивной функции, то функция будет вызывать саму себя «бесконечно». На самом деле это не совсем так, потому что интерпретатор рано или поздно остановит выполнение программы и выдаст ошибку.

Приведённый выше алгоритм не является полным аналогом операции возведения в степень \*\*, например, он не будет работать при отрицательных степенях, а потому показан лишь в учебных целях.

## Особенность рекурсии в Python

Напишем рекурсивную функцию, которая находит сумму натуральных чисел от 1 до *n* (по аналогии с факториалом):

def summatorial(n):

if n == 1:

return 1

else:

return n + summatorial(n - 1)

print(summatorial(5))

print(summatorial(250))

При передаче этой функции аргументов 55 и 250250 мы получим соответствующие результаты:

15

31375

Но при попытке вызова summatorial(1500), на экран выведется ошибка:

RecursionError: maximum recursion depth exceeded in comparison

которая гласит, что превышена максимальная установленная глубина рекурсии. Дело в том, что значение максимальной глубины рекурсии по умолчанию установлено на уровне 10001000, поэтому при попытке вызова рекурсивной функции с глубиной больше, чем 10001000 (иногда 10241024), на экран выведется соответствующая ошибка. Существует несколько способов решения описанной выше проблемы. Их мы обсудим при подробном изучении задания №16№16 в отдельном курсе.

Работа с модулями

В Python существует огромное количество реализованных функций, которые упрощают процесс решения различных задач. Мы уже знакомы с некоторыми функциями, которые доступны сразу: max(), len(), sum() и т.д. Но многие другие полезные функции хранятся в отдельных модулях, которые сначала нужно импортировать в текущую программу для дальнейшего использования.

Модуль — это такой же «питоновский» файл с расширением .py, содержащий код, включающий в себя различные функции и константы. По сути модуль представляет собой библиотеку функций, которые можно использовать в других программах, для чего их нужно прежде импортировать.

## Модуль math

Модуль math является одним из самых полезных и популярных в Python, а содержащиеся в нем функции зачастую удобно использовать в том числе и для решения задач из ЕГЭ по информатике.

Некоторые полезные функции из модуля math:

|  |  |
| --- | --- |
| **Функция** | **Назначение** |
| ceil | округление числа вверх до ближайшего целого |
| prod | вычисление произведения элементов коллекции |
| factorial | вычисление факториала числа |
| sqrt | вычисления квадратного корня из числа |

С полным перечнем функций, входящих в модуль math, можно ознакомиться <https://docs.python.org/3/library/math.html>.

## Подключение модулей

Для использования функций какого-либо модуля прежде нужно его подключить в текущую программу. Сделать это можно несколькими способами:

1. Такой импорт

import имя\_модуля

подключает модуль <имя\_модуля> в текущую программу полностью и позволяет использовать его функции через следующий синтаксис:

имя\_модуля.имя\_функции()

Данный код:

import math

num = 5

factorial\_num = math.factorial(num)

print(factorial\_num)

вычисляет факториал числа 5 с помощью функции factorial(), импортированной из модуля math, и выводит 120.

1. А таким образом

from имя\_модуля import имя\_функции

можно импортировать не весь модуль, а только определенную функцию из модуля для дальнейшего использования. При таком способе указывать имя модуля при обращении к функции уже не нужно.

Этот код:

from math import sqrt

num = 4

sqrt\_num = sqrt(num)

print(sqrt\_num)

вычисляет квадратный корень числа 4 и выводит 2.0.

Обратите внимание на то, что доступны только перечисленные при импорте функции.  
Данный код:

from math import sqrt

num = 4

factorial\_num = factorial(num)

print(factorial\_num)

завершится с ошибкой и выведет

NameError: name 'factorial' is not defined

Допустимо импортировать также сразу несколько функций, перечисляя их наименования через запятую:

from math import factorial, ceil, prod

1. Сразу все функции из модуля можно импортировать с помощью \*:

from math import \*

num = 6

print(factorial(num))

nums = [5, 2, 10]

print(prod(nums))

float\_num = 56.473

print(ceil(float\_num))

Приведенный выше код использует сразу 33 функции (factorial(), prod() и ceil()) и выводит

720

100

57

Примечание *№1*. Функция prod() принимает на вход коллекцию элементов типа int или float (например, список чисел) и вычисляет их произведение.

Примечание *№2*. Функция ceil() принимает на вход одно число и всегда округляет его вверх до ближайшего целого.

Этот код:

from math import ceil

print(ceil(12))

print(ceil(12.01))

print(ceil(12.5))

print(ceil(12.999))

выводит

12

13

13

13