**Тема урока: пользовательские функции**

1. Функции без параметров
2. Преимущества использования функций
3. Ключевое слово pass
4. Решение задач

**Аннотация.** Создание пользовательских функций без параметров.

**Функции**

В предыдущих уроках мы использовали встроенные в Python функции print(), input(), int(), str(), len() и многие другие. Пришло время начать писать свои собственные функции.

В самом начале курса вам было предложено решить задачу, в которой требовалось изобразить звездный прямоугольник размерами 5×75×7 (55 строк и 77 столбцов).

Наш первый вариант кода выглядел примерно так:

print('\*\*\*\*\*\*\*')

print('\*\*\*\*\*\*\*')

print('\*\*\*\*\*\*\*')

print('\*\*\*\*\*\*\*')

print('\*\*\*\*\*\*\*')

Далее мы изучили оператор умножения строки на число (оператор повторения) и написали бы код:

print('\*' \* 7)

print('\*' \* 7)

print('\*' \* 7)

print('\*' \* 7)

print('\*' \* 7)

Ну и наконец, мы изучили циклы, после чего код принял бы вид:

for \_ in range(5):

print('\*' \* 7)

А теперь представим, что таких прямоугольников нужно изобразить не один, а несколько, скажем, 3 штуки.

Тогда код программы будет иметь вид:

for \_ in range(5):

print('\*' \* 7)

print()

for \_ in range(5):

print('\*' \* 7)

print()

for \_ in range(5):

print('\*' \* 7)

Результатом выполнения такого кода будет:

\*\*\*\*\*\*\*

\*\*\*\*\*\*\*

\*\*\*\*\*\*\*

\*\*\*\*\*\*\*

\*\*\*\*\*\*\*

\*\*\*\*\*\*\*

\*\*\*\*\*\*\*

\*\*\*\*\*\*\*

\*\*\*\*\*\*\*

\*\*\*\*\*\*\*

\*\*\*\*\*\*\*

\*\*\*\*\*\*\*

\*\*\*\*\*\*\*

\*\*\*\*\*\*\*

\*\*\*\*\*\*\*

И хотя предыдущий код полностью решает поставленную задачу, он не лишен недостатков. Во-первых, он довольно громоздкий из-за **повторения части кода**, отвечающей за вывод прямоугольника. Во-вторых, если понадобится изменить размеры прямоугольника, придется менять их трижды, в каждой части кода, выводящей прямоугольник.

Вместо повторения кода для вывода прямоугольника, можно перенести  его в отдельную **функцию**и вызвать ее 3 раза.

Для создания функции пишем такой код:

def draw\_box():

for \_ in range(5):

print('\*' \* 7)

Когда функция создана, чтобы увидеть результат ее работы, надо вызвать ее по имени:

draw\_box()

Теперь, чтобы изобразить 3 прямоугольника, можно написать код:

draw\_box()

print()

draw\_box()

print()

draw\_box()

Код стал короче, читабельнее (за счет удачного названия функции), а главное, если потребуются иные размеры прямоугольника, достаточно будет изменить только саму функцию draw\_box().

**Именование функций**

Имена функциям назначаются точно так же, как переменным. Имя функции должно быть достаточно описательным, чтобы любой читающий ваш код, мог догадаться, что именно делает функция.

Python и тут требует соблюдения тех же правил, что при именовании переменных:

1. в имени функции используются только латинские буквы a-z, A-Z, цифры и символ нижнего подчеркивания (\_);
2. имя функции не может начинаться с цифры;
3. имя функции по возможности должно отражать ее назначение;
4. символы верхнего и нижнего регистра различаются.

**Помни:** Python — регистрочувствительный язык. Для именования переменных и функций принято использовать стиль **lower\_case\_with\_underscores**(слова из маленьких букв с подчеркиваниями).

Поскольку функции**выполняют действия**, большинство программистов предпочитает в именах функций **использовать** **глаголы**. Например:

* функцию, которая **рисует прямоугольник**, можно назвать draw\_box();
* функцию, которая **печатает чек**, можно назвать print\_check();
* функцию, которая вычисляет заработную плату до удержаний, можно назвать calculate\_gross\_рау().

Каждое из этих имен дает описание того, что функция делает.

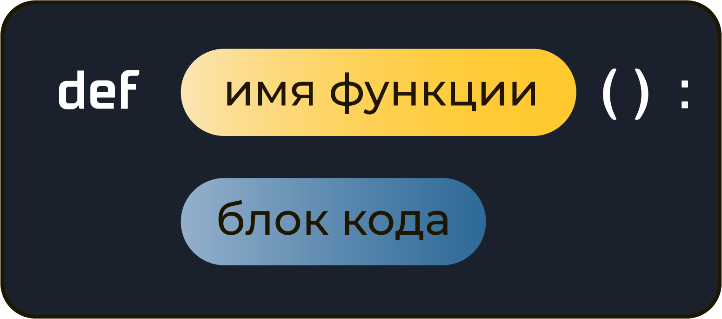
**Объявление функции**

Итак, **функция – отдельная, функционально независимая часть программы, выполняющая определенную задачу.**

Функции объявляются с помощью ключевого слова def (от англ. define – определять). За ключевым словом def следуют название функции, круглые скобки () и двоеточие :.

def название\_функции():

блок кода



   Первая строка объявления функции называется **заголовком функции**.

Со следующей строки идет блок кода – **тело функции**. Это набор инструкций, составляющих одно целое и выполняющихся каждый раз, когда вызывается функция.

Обратите внимание, что каждая строка в теле функции выделена отступом.

Для выделения строк блока кода отступом программисты Python обычно используют **четыре пробела** в соответствии со стандартом PEP 8.

Рассмотрим объявление функции:

def print\_message():

print('Я - Артур,')

print('король британцев. ')

Этот фрагмент кода определяет функцию с именем print\_message(). Тело ее состоит из двух инструкций, и вызов приведет к их исполнению.

**Вызов функции**

Для вызова функции пишут ее название и круглые скобки.

**Важно:** очень часто начинающие программисты забывают вызывать функцию. Помните, что объявление функции не вызывает ее.

# объявление функции

def print\_message():

print('Я - Артур,')

print('король британцев. ')

# вызов функции

print\_message()

**Примечания**

**Примечание 1.** Обратите внимание, что объявление пользовательской функции немного напоминает использование условного оператора if и циклов for, while.

**Примечание 2.** Объявление функции должно **предшествовать** ее вызову. Следующий программный код:

# вызов функции

print\_message()

# объявление функции

def print\_message():

print('Я - Артур,')

print('король британцев. ')

приведет к ошибке:

NameError: name 'print\_message' is not defined

**Примечание 3.** При объявлении функции следует убедиться, что каждая строка тела функции начинается с одинакового количества пробелов, иначе произойдет ошибка. Например, приведенное ниже определение функции приведет к ошибке:

def print\_greeting():

print('Доброе утро!')

print('Сегодня мы будем изучать функции.')

print('Это очень важная тема!')

**Примечание 4.** Иногда, при объявлении функции требуется сделать своего рода заглушку, чтобы функция ничего не выполняла. Тогда мы используем ключевое слово pass:

def do\_nothing():

pass

Мы объявили функцию с именем do\_nothing(). Тело такой функции содержит единственную строку кода, которая ничего не делает.

**Примечание 5.** Функции часто называют **подпрограммами**.

**Примечание 6.** Посвящается всем, кто забывает вызвать функцию и ожидает от нее результата:

## Звездный прямоугольник 1

Напишите функцию draw\_box(), которая выводит звездный прямоугольник с размерами 14×1014×10 в соответствии с образцом:

\*\*\*\*\*\*\*\*\*\*

\* \*

\* \*

\* \*

\* \*

\* \*

\* \*

\* \*

\* \*

\* \*

\* \*

\* \*

\* \*

\*\*\*\*\*\*\*\*\*\*

**Примечание.** Для вывода прямоугольника используйте цикл for.

# объявление функции

def draw\_box():

print('\*'\*10)

for i in range(12):

print('\*'+' '\*8+'\*')

print('\*'\*10)

draw\_box() # вызов функции

## Звездный треугольник 1

Напишите функцию draw\_triangle(), которая выводит звездный прямоугольный треугольник с катетами, равными 1010 в соответствии с образцом:

\*

\*\*

\*\*\*

\*\*\*\*

\*\*\*\*\*

\*\*\*\*\*\*

\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*

**Примечание.** Для вывода треугольника используйте цикл for.

# объявление функции

def draw\_triangle():

for i in range(1,11):

print(i\*'\*')

# основная программа

draw\_triangle() # вызов функции

## Звездный треугольник 🌶️

Напишите функцию draw\_triangle(), которая выводит звездный равнобедренный треугольник с основанием и высотой равными 1515 и 88 соответственно:

\*

\*\*\*

\*\*\*\*\*

\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Примечание 1.** Для вывода треугольника используйте цикл for.

**Примечание 2.** Справа от звездочек пробелов нет.

# объявление функции

def draw\_triangle():

m = 15

for i in range(1, m + 1, 2):

print(' ' \* ((m - i) // 2) + '\*' \* i)

# основная программа

draw\_triangle() # вызов функции

**Тема урока: функции с параметрами**

1. Функции с параметрами
2. Область видимости параметрической переменной
3. Решение задач

**Аннотация.** Создание пользовательских функций с параметрами.

**Функции с параметрами**

В предыдущем уроке мы определили функцию draw\_box(), которая выводит звездный прямоугольник с размерами 5×75×7:

def draw\_box():

for i in range(5):

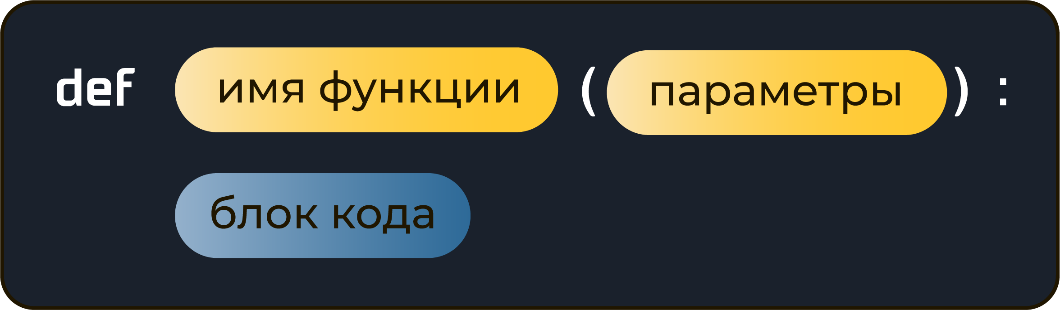
print('\*' \* 7)

Было бы намного удобнее, если бы функция draw\_box() выводила прямоугольник с произвольными размерами. И действительно, функции могут принимать входные параметры, что делает их более гибкими.

Функции с параметрами объявляются так же как функции без параметров, только с указанием в скобках:

def название\_функции(параметры):

блок кода



Давайте перепишем предыдущую версию функции draw\_box() так, чтобы она принимала параметры, задающие высоту и ширину прямоугольника:

def draw\_box(height, width): # функция принимает два параметра

for i in range(height):

print('\*' \* width)

Теперь наша функция draw\_box() принимает два целочисленных параметра height – высота прямоугольника и width – ширина прямоугольника, и для ее вызова нам нужно обязательно их указать.

Чтобы вывести звездный прямоугольник размерами 5 на 7 мы пишем код:

draw\_box(5, 7)

Результатом такого вызова функции draw\_box(5, 7) будет:

\*\*\*\*\*\*\*

\*\*\*\*\*\*\*

\*\*\*\*\*\*\*

\*\*\*\*\*\*\*

\*\*\*\*\*\*\*

Чтобы вывести прямоугольник размерами 10 на 15, мы пишем код:

draw\_box(10, 15)

Результатом такого вызова функции draw\_box(10, 15) будет:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Теперь с помощью нашей новой версии функции draw\_box() можем в одной программе выводить прямоугольники разных размеров. Следующий программный код:

draw\_box(3, 3)

print()

draw\_box(5, 5)

print()

draw\_box(4, 10)

выведет:

\*\*\*

\*\*\*

\*\*\*

\*\*\*\*\*

\*\*\*\*\*

\*\*\*\*\*

\*\*\*\*\*

\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*

На место параметров мы можем подставлять не только целочисленные константы, но и значения переменных. Следующий программный код:

n = 3

m = 9

draw\_box(n, m)

выведет:

\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*

**Еще примеры**

Напишем функцию print\_hello(n), которая принимает одно натуральное число �*n* и печатает слово Hello ровно �*n* раз.

def print\_hello(n):

print('Hello' \* n)

Следующий программный код:

print\_hello(3)

print\_hello(5)

times = 2

print\_hello(times)

выведет:

HelloHelloHello

HelloHelloHelloHelloHello

HelloHello

Функцию print\_hello() можно сделать более гибкой, если передавать в нее еще один параметр – текст для вывода:

def print\_text(txt, n):

print(txt \* n)

Следующий программный код:

print\_text('Hello', 5)

print\_text('A', 10)

выведет:

HelloHelloHelloHelloHello

AAAAAAAAAA

**Параметры VS аргументы**

**Аргумент** – это любая порция данных, которая передается в функцию, когда функция вызывается. **Параметр** – это переменная, которая получает аргумент, переданный в функцию.

Для функции draw\_box(height, width):

​def draw\_box(height, width):

for i in range(height):

print('\*' \* width)

параметрами являются переменные height и width.

В момент вызова функции draw\_box(height, width):

height = 10

draw\_box(height, 9)

аргументами являются height и 9.

    Параметры функций часто называют **параметрическими переменными**.

**Внесение изменений в параметры**

Когда аргумент передается в функцию, параметрическая переменная функции будет ссылаться на значение этого аргумента. Однако любые изменения, которые вносятся в параметрическую переменную, не будут влиять на аргумент.

Cледующий программный код:

def draw\_box(height, width):

height = 2

width = 10

for i in range(height):

print('\*' \* width)

n = 5

m = 7

draw\_box(n, m)

print(n, m)

выведет:

\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*

5 7

В теле функции вносятся изменения в значения параметрических переменных height и width, однако это никак не повлияло на значение переменных n и m из основной программы, которые передавались в качестве аргументов в функцию draw\_box().

**Примечание**

**Примечание 1.** Функции в Python могут принимать сколько угодно параметров.

**Примечание 2.** Иногда вместо параметров и аргументов говорят о **формальных параметрах** и **фактических параметрах**. Формальные параметры – переменные, которые мы пишем при описании функции. Фактические параметры – то, что реально подставляется при вызове функции.

**Звездный треугольник**

Напишите функцию draw\_triangle(fill, base), которая принимает два параметра:

* fill – символ заполнитель;
* base – величина основания равнобедренного треугольника;

а затем выводит его.

**Примечание.** Гарантируется, что основание треугольника – нечетное число.

# объявление функции

def draw\_triangle(fill, base):

for i in range((base+1)//2):

print(fill\*i)

for j in range((base+1)//2,0,-1):

print(fill\*j)

# считываем данные

fill = input()

base = int(input())

# вызываем функцию

draw\_triangle(fill, base)

**ФИО**

Напишите функцию print\_fio(name, surname, patronymic), которая принимает три параметра:

* name – имя человека;
* surname – фамилия человека;
* patronymic – отчество человека;

а затем выводит на печать ФИО человека.

**Примечание.** Предусмотрите тот факт, что все три буквы в ФИО должны иметь верхний регистр.

# объявление функции

def print\_fio(name, surname, patronymic):

b=name[:1].upper()

a=surname[:1].upper()

c=patronymic[:1].upper()

print(a,b,c,sep='')

# считываем данные

name, surname, patronymic = input(), input(), input()

# вызываем функцию

print\_fio(name, surname, patronymic)

## Сумма цифр

Напишите функцию print\_digit\_sum(), которая принимает одно целое число num и выводит на печать сумму его цифр

# объявление функции

def print\_digit\_sum(num):

m=0

while num>0:

m+=num%10

num//=10

print(m)

# считываем данные

n = int(input())

# вызываем функцию

print\_digit\_sum(n)

**Тема урока: локальные и глобальные переменные**

1. Локальные переменные
2. Область действия локальной переменной
3. Глобальные переменные
4. Глобальные константы
5. Решение задач

**Аннотация.** Локальные и глобальные переменные.

**Локальные переменные**

**Локальными** называются переменные, объявленные внутри функции и доступные только ей самой. Программный код за пределами функции к ним доступа не имеет.

Рассмотрим функцию print\_texas(), которая выводит информацию о количестве птиц, обитающих в Техасе.

def print\_texas():

birds = 5000

print('В Техасе обитает', birds, 'птиц.')

В теле функции мы создаем переменную birds, которой присваивается значение, равное 5000. Такая переменная является локальной для функции print\_texas(). Всякий раз, когда переменной внутри функции присваивается значение, в результате **создается локальная переменная**. Она принадлежит функции, в которой создается,  и к ней получает доступ только программный код этой функции.

Термин **“локальная”** указывает на то, что переменная может использоваться лишь в этом месте — внутри функции, в которой создается.

Если программный код одной функции попытается обратиться к локальной переменной, принадлежащей другой функции, произойдет ошибка.

Рассмотрим следующий программный код:

def print\_texas():

birds = 5000

print('В Техасе обитает', birds, 'птиц.')

def print\_california():

print('В Калифорнии обитает', birds, 'птиц.')

Функция print\_california() обращается к локальной переменной birds функции print\_texas(). Вызов функции print\_california(), приводит к ошибке:

NameError: name 'birds' is not defined

Локальные переменные скрыты от других функций, поэтому другие функции могут иметь собственные локальные переменные с тем же именем. Например,

def print\_texas():

birds = 5000

print('В Техасе обитает', birds, 'птиц.')

def print\_california():

birds = 9000

print('В Калифорнии обитает', birds, 'птиц.')

В каждой из этих двух функций есть локальная переменная с именем birds. Но они никогда не видны одновременно, так как находятся в разных функциях.

Когда выполняется функция print\_texas(), видима переменная birds, значение которой равно 5000. Когда выполняется функция print\_california(), видима переменная birds, значение которой равно 9000.

Разные функции могут иметь локальные переменные с одинаковыми именами, потому что они не видят локальных переменных друг друга.

**Область действия переменной**

Область действия переменной – часть программы, в которой можно к ней обращаться, та функция, где она создана. Переменная видима только программному коду в области ее действия. Никакая инструкция за пределами функции не может обращаться к такой переменной.

К локальной переменной не может обращаться программный код, который появляется внутри функции **до того**, как переменная была создана.

Например, если в функции print\_texas() поменять местами строки кода:

def print\_texas():

print('В Техасе обитает', birds, 'птиц.')

birds = 5000

то при вызове этой функции получим ошибку:

UnboundLocalError: local variable 'birds' referenced before assignment

Ошибка возникла в результате преждевременного обращения к еще не объявленной локальной переменной birds.

**Область действия параметрической переменной**

Область действия параметрической переменной — функция, в которой этот параметр используется. К параметрической переменной имеет доступ весь программный код этой функции.

Рассмотрим уже известную нам функцию:

def draw\_box(height, width):

for i in range(height):

print('\*' \* width)

Параметрические переменные тут height, width. Внутри функции объявляется одна локальная переменная i.

**Примечания**

**Примечание 1.** Параметрическая переменная тоже локальная.

**Примечание 2.** Память для локальных переменных выделяется на время исполнения данной функции в специальной области, называемой [стеком](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D0%B5%D0%BA). При завершении работы функции память освобождается, внутренние результаты работы функции не сохраняются от одного обращения к другому.

**Глобальные переменные**

**Глобальными** называются переменные, объявленные в основной программе и доступные как программе, так и всем ее функциям.

Рассмотрим следующий программный код:

birds = 5000 # глобальная переменная

def print\_texas():

print('В Техасе обитает', birds, 'птиц.')

def print\_california():

print('В Калифорнии обитает', birds, 'птиц.')

В самом начале программы создаем глобальную переменную birds, значение которой равно 5000. Далее описываем две функции, обращающиеся к глобальной переменной. Результатом выполнения следующего кода:

print\_texas()

print\_california()

будет:

В Техасе обитает 5000 птиц.

В Калифорнии обитает 5000 птиц.

Обмен информацией между основной программой и функциями осуществляется только с помощью параметров функций и глобальных переменных.

Функция может использовать любые глобальные переменные кроме имеющих те же имена, что и ее локальные переменные. Если в функции объявлена локальная переменная с тем же именем, что у одной из глобальных, то данная глобальная переменная становится недоступной в этой функции, и при указании идентификатора переменной произойдет обращение к локальной переменной функции, а не одноименной глобальной.

Рассмотрим следующий программный код:

birds = 5000 # глобальная переменная

def print\_texas():

birds = 1000 # локальная переменная

print('В Техасе обитает', birds, 'птиц.')

def print\_california():

birds = 7000 # локальная переменная

print('В Калифорнии обитает', birds, 'птиц.')

В самом начале программы мы создаем глобальную переменную birds, значение которой равно 5000. Далее мы описываем две функции, в которых создаются локальные переменные с таким же именем birds. Таким образом при обращении к переменной birds внутри функций будет происходить обращение именно к локальной переменной.

Результатом выполнения следующего кода:

print\_texas()

print\_california()

будет:

В Техасе обитает 1000 птиц.

В Калифорнии обитает 7000 птиц.

**Глобальные переменные – зло**

Большинство программистов согласны, что следует ограничить использование глобальных переменных либо не использовать их вообще. Причины следующие.

* **Глобальные переменные затрудняют отладку программы.** Значение глобальной переменной может быть изменено любой инструкцией в программном файле. Если обнаружится, что в глобальной переменной хранится неверное значение, то придется отыскать все инструкции, которые к ней обращаются, чтобы определить, откуда поступает плохое значение. В программе с тысячами строк кода сделать это непросто.
* **Функции, использующие глобальные переменные, обычно зависят от этих переменных.** Если возникнет необходимость применить такую функцию в другой программе, скорее всего придется эту функцию перепроектировать, чтобы она не опиралась на глобальную переменную.
* **Глобальные переменные затрудняют понимание программы.** Глобальная переменная может быть модифицирована любой инструкцией в программе. При необходимости разобраться в какой-то части программы, использующей глобальную переменную, придется узнать обо всех других частях программы, обращающихся к этой глобальной переменной.

В большинстве случаев следует создавать переменные локально и передавать их в качестве аргументов в функции, которым нужно к ним обратиться.

**Глобальные константы**

Хотя следует избегать использования глобальных переменных, в программе допускается применение глобальных констант. Глобальная константа – глобальное имя, ссылающееся на неизменное значение. Поскольку значение глобальной константы не может быть изменено во время исполнения программы, можно не беспокоиться о потенциальных опасностях, обычно связанных с использованием глобальных переменных.

Несмотря на то, что язык Python не позволяет создавать настоящие глобальные константы, их можно имитировать при помощи глобальных переменных. Если глобальная переменная не объявляется с использованием ключевого слова global внутри функции, то внутри этой функции вы никак не сможете изменить значение этой глобальной переменной.

Математический модуль math  определяет две глобальные переменные, math.pi и math.e, которым присвоены математические значения констант [*π*](https://ru.wikipedia.org/wiki/%D0%9F%D0%B8_(%D1%87%D0%B8%D1%81%D0%BB%D0%BE))= 3.14159265 и [*e*](https://ru.wikipedia.org/wiki/E_(%D1%87%D0%B8%D1%81%D0%BB%D0%BE))= 2.71828.

**Ключевое слово global**

Если нужно, чтобы инструкция внутри функции присваивала значение глобальной переменной, то требуется дополнительный шаг. В этом случае, глобальная переменная должна быть объявлена внутри функции.

Рассмотрим следующий программный код:

def print\_texas():

global birds

birds = 5000

print('В Техасе обитает', birds, 'птиц.')

def print\_california():

print('В Калифорнии обитает', birds, 'птиц.')

print\_texas()

print\_california()

 Результатом выполнения следующего кода:

print\_texas()

print\_california()

будет:

В Техасе обитает 5000 птиц.

В Калифорнии обитает 5000 птиц.

**Тема урока: функции с возвратом значения**

1. Функции с возвратом значения
2. Решение задач

**Аннотация.** Функции могут не только принимать и не принимать параметры, как в примерах предыдущих уроков, но и возвращать значения. Рассмотрим этот особый тип функций - с возвратом значения.

**Функция с возвратом значения**

Функция с возвратом значения похожа на функцию без возврата значения тем, что:

* это набор инструкций, выполняющий определенную задачу;
* когда нужно выполнить функцию, ее вызывают.

Однако, когда функция с возвратом значения завершается, она возвращает значение в ту часть программы, которая ее вызвала. Возвращаемое из функции значение используется как любое другое: оно может быть присвоено переменной, выведено на экран, использовано в математическом выражении (если это число) и т. д.

    Функция с возвратом значения возвращает значение обратно в ту часть программы, которая ее вызвала.

Мы уже сталкивались со многими функциями с возвратом значений:

* функция int() – преобразует строку к целому числу и возвращает его;
* функция float() – преобразует строку к вещественному числу и возвращает его;
* функция range() – возвращает последовательность целых чисел 0, 1, 2, ...;
* функция abs() – возвращает абсолютное значение числа (модуль числа);
* функция len() – возвращает длину строки или списка.

Функцию с возвратом значения пишут точно так же, как и без, но она должна иметь инструкцию return.

Вот общий формат определения функции с возвратом значения в Python:

def название\_функции():

блок кода

return выражение

В функции должна быть инструкция return, принимающая форму:

return выражение

Значение выражения, которое следует за ключевым словом return, будет отправлено в ту часть программы, которая вызвала функцию. Это может быть переменная либо выражение, к примеру, математическое.

  Функция с возвратом значения имеет инструкцию return, возвращающую значение в ту часть программы, которая ее вызвала.

При изучении вещественных чисел мы решали задачу о переводе градусов по шкале Фаренгейта в градусы по шкале Цельсия по формуле �=59(�− 32)*C*=95​(*F*− 32).

Напишем функцию, которая осуществляет перевод:

def convert\_to\_celsius(temp):

result = (5 / 9) \* (temp - 32)

return result

Задача этой функции — принять одно число temp в качестве аргумента – количество градусов по шкале Фаренгейта, и вернуть другое — количество градусов по шкале Цельсия.

Рассмотрим ее работу. Первая инструкция в блоке функции присваивает значение (5 / 9) \* (temp - 32) переменной result. Затем исполняется инструкция return, которая приводит к завершению исполнения функции и отправляет значение из переменной result, назад в ту часть программы, которая вызвала эту функцию.

# функция перевода градусов Фаренгейта в градусы Цельсия

def convert\_to\_celsius(temp):

result = (5 / 9) \* (temp - 32)

return result

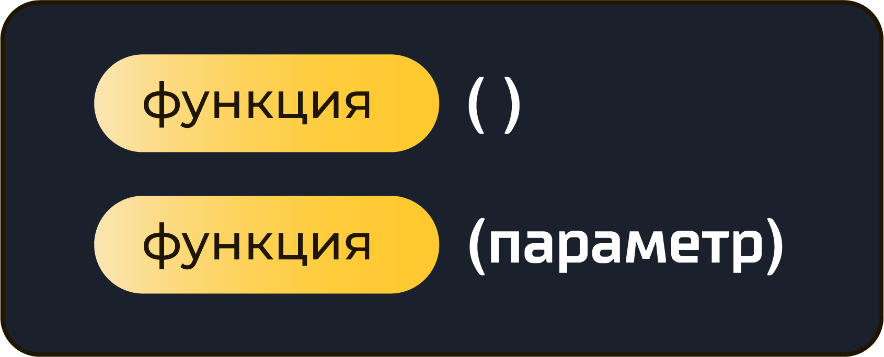
# основная программа

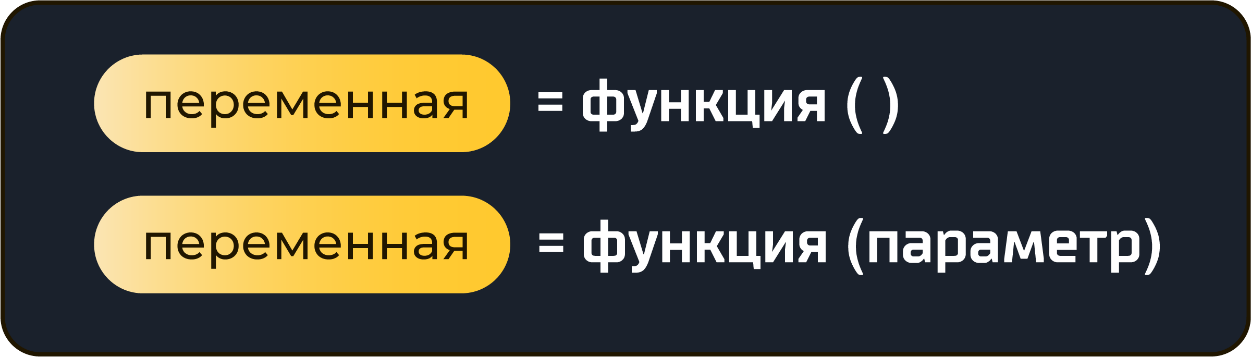
temp = float(input('Bвeдитe количество градусов по Фаренгейту: '))

celsius = convert\_to\_celsius(temp)

print(celsius) # градусы Цельсия

Основная программа получает от пользователя одно число – значение в градусах Фаренгейта, и вызывает функцию, передавая значение переменной temp в качестве аргумента. Значение, которое возвращается из функции convert\_to\_celsius, присваивается переменной celsius.





**Использование инструкции return по максимуму**

Взглянем еще раз на функцию convert\_to\_celsius():

def convert\_to\_celsius(temp):

result = (5 / 9) \* (temp - 32)

return result

Обратите внимание, что внутри этой функции происходят две вещи: во-первых, переменной result присваивается значение выражения (5 / 9) \* (temp - 32), и во-вторых, значение переменной result возвращается из функции. Эта функция хорошо справляется с поставленной перед ней задачей, но ее можно упростить. Поскольку инструкция return возвращает значение выражения, переменную result устраняем и переписываем функцию так:

def convert\_to\_celsius(temp):

return (5 / 9) \* (temp - 32)

Эта версия функции не сохраняет значение (5 / 9) \* (temp - 32) в отдельной переменной, а сразу возвращает значение выражения с помощью инструкции return. Делает то же, что и предыдущая версия, но за один шаг.

**Использование нескольких return**

В одной функции может быть сколько угодно инструкций return. Рассмотрим функцию convert\_grade(), которая переводит стобалльную оценку в пятибалльную:

def convert\_grade(grade):

if grade >= 90:

return 5

elif grade >= 80:

return 4

elif grade >= 70:

return 3

elif grade >= 60:

return 2

else:

return 1

# основная программа

grade = int(input('Введите вашу отметку по 100-балльной системе: '))

print(convert\_grade(grade))

В функции convert\_grade() используется 5 инструкций return. Каждая из них возвращает соответствующее значение и завершает работу функции.

Функцию convert\_grade() можно переписать с помощью одной инструкции return:

def convert\_grade(grade):

if grade >= 90:

result = 5

elif grade >= 80:

result = 4

elif grade >= 70:

result = 3

elif grade >= 60:

result = 2

else:

result = 1

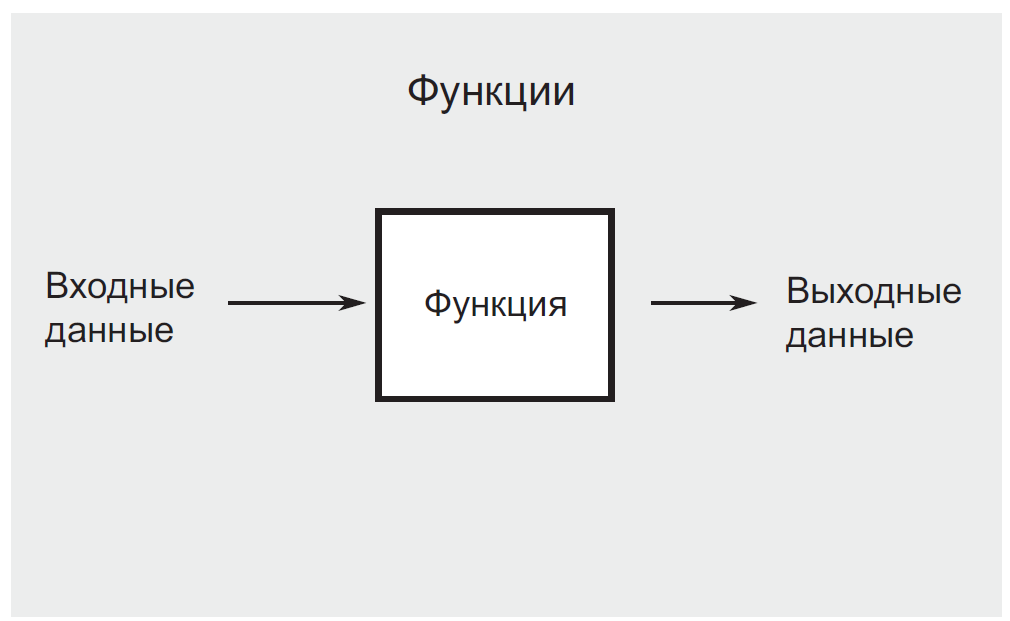
return result

**Примечания**

**Примечание 1.** Функции с возвратом значения предоставляют те же преимущества, что функции без возврата значения:

* упрощают программный код;
* уменьшают дублирование кода;
* упрощают тестирование кода;
* увеличивают скорость разработки;
* способствуют работе в команде.

**Примечание 2.** Графическая интерпретация работы функции с возвратом значения:



**Примечание 3.** result – хорошее название для переменной, значение которой возвращается из функции.

## Решение задач

**Задача 1.** Напишите функцию, которая возвращает длину гипотенузы прямоугольного треугольника по известным значениям его катетов.

**Решение.** Для нахождения длины гипотенузы, нам нужно применить теорему Пифагора: квадрат гипотенузы прямоугольного треугольника, равен сумме квадратов его катетов. Другими словами, если �,�*a*,*b* – длины катетов, а �*c* – длина гипотенузы, то имеет место равенство:�2=�2+�2⟹�=�2+�2*c*2=*a*2+*b*2⟹*c*=*a*2+*b*2​

Функция, вычисляющая длину гипотенузы, может иметь вид:

def compute\_hypotenuse(a, b):

c = (a \*\* 2 + b \*\* 2) \*\* 0.5

return c

 Для извлечения квадратного корня мы использовали оператор возведения в степень. Напомним, что результатом обоих выражений: math.sqrt(c) и c \*\* 0.5 является одно число �*c*​.

Следующий программный код:

print(compute\_hypotenuse(3, 4))

print(compute\_hypotenuse(5, 12))

print(compute\_hypotenuse(1, 1))

выведет:

5.0 # длина гипотенузы треугольника с катетами 3 и 4

13.0 # длина гипотенузы треугольника с катетами 5 и 12

1.4142135623730951 # длина гипотенузы треугольника с катетами 1 и 1

Если нужно передать программе числа, считанные с клавиатуры, то мы пишем следующий код:

x = int(input())

y = int(input())

hypotenuse = compute\_hypotenuse(x, y)

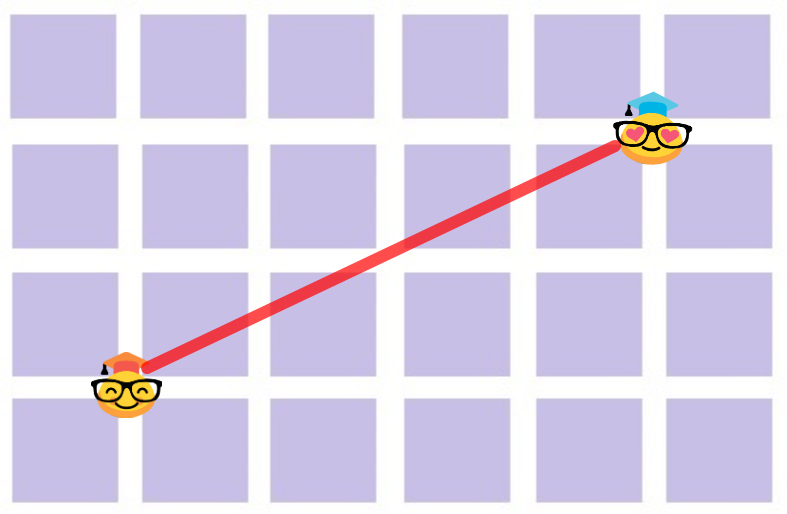
print(hypotenuse)

В модуле math имеется встроенная функция hypot(x, у) которая возвращает длину гипотенузы прямоугольного треугольника с катетами x и y.

Одним из основных преимуществ функций является возможность их повторного использования для решения похожих задач. Рассмотрим задачу нахождения расстояния между двумя точками.

**Задача 2.** Напишите функцию get\_distance(x1, y1, x2, y2), вычисляющую расстояние между точками (�1;�1)(*x*1​;*y*1​) и (�2;�2)(*x*2​;*y*2​).

**Решение.** Расстояние между двумя точками (�1;�1)(*x*1​;*y*1​) и (�2;�2)(*x*2​;*y*2​) определяется по формуле�=(�1−�2)2+(�1−�2)2*ρ*=(*x*1​−*x*2​)2+(*y*1​−*y*2​)2​.



Несложно заметить, что искомое расстояние – это длина гипотенузы прямоугольного треугольника с катетами равными ∣�1−�2∣∣*x*1​−*x*2​∣ и ∣�1−�2∣∣*y*1​−*y*2​∣.

Функция, вычисляющая расстояние между точками, может иметь вид:

def get\_distance(x1, y1, x2, y2):

return compute\_hypotenuse(x1 - x2, y1 - y2)

Для подсчета искомого расстояния мы используем уже созданную нами функцию compute\_hypotenuse передавая ей в качестве аргументов числа x1 - x2 и y1 - y2.

Основная программа имеет вид:

x1, y1 = float(input()), float(input()) # считываем координаты первой точки

x2, y2 = float(input()), float(input()) # считываем координаты второй точки

print(get\_distance(x1, y1, x2, y2)) # вычисляем и выводим расстояние между точками

**Задача 3.** Напишите функцию sum\_digits(n), принимающую в качестве аргумента натуральное число и возвращающую сумму его цифр.

**Решение.** Функция sum\_digits(n) может иметь вид:

def sum\_digits(n):

result = 0

while n > 0:

result += n % 10

n //= 10

return result

Основная программа имеет вид:

n = int(input())

print(sum\_digits(n)) # вычисляем и выводим сумму цифр считанного числа

**Задача 4.** Напишите функцию compute\_average(numbers), принимающую в качестве аргумента список чисел и возвращающую среднее значение элементов списка.

**Решение.** Для подсчета среднего значения элементов списка нужно вычислить сумму всех элементов и их количество, то есть использовать функции sum() и len(). Функция compute\_average(numbers) может иметь вид:

def compute\_average(numbers):

return sum(numbers) / len(numbers)

Основная программа имеет вид:

numbers = [1, 3, 5, 1, 6, 8, 10, 2]

print(compute\_average(numbers)) # вычисляем и выводим среднее значение элементов списка

Результатом работы такой программы будет число 4.5, которое и является средним значением.

## Конвертер километров

Напишите функцию convert\_to\_miles(km), которая принимает в качестве аргумента расстояние в километрах и возвращает расстояние в милях. Формула для преобразования: мили = километры \* 0.6214.

**Примечание.** Следующий программный код:

print(convert\_to\_miles(1))

print(convert\_to\_miles(5))

print(convert\_to\_miles(10))

должен выводить:

0.6214

3.107

6.214

# объявление функции

def convert\_to\_miles(km):

miles=km\*0.6214

return miles

# считываем данные

num = int(input())

# вызываем функцию

print(convert\_to\_miles(num))

## ****Количество дней****

Напишите функцию get\_days(month), которая принимает в качестве аргумента номер месяца и возвращает количество дней в данном месяце.

**Примечание 1.** Гарантируется, что передаваемый аргумент находится в диапазоне от 1 до 12.

**Примечание 2.** Считайте, что год является невисокосным.

**Примечание 3.** Следующий программный код:

print(get\_days(1))

print(get\_days(2))

print(get\_days(9))

должен выводить:

31

28

30

# объявление функции

def get\_days(month):

if month in [1,3,5,7,8,10,12]:

return 31

elif month in [4,6,9,11]:

return 30

else:

return 28

# считываем данные

num = int(input())

# вызываем функцию

print(get\_days(num))

## Делители 1

Напишите функцию get\_factors(num), принимающую в качестве аргумента натуральное число и возвращающую список всех делителей данного числа.

**Примечание.** Следующий программный код:

print(get\_factors(1))

print(get\_factors(5))

print(get\_factors(10))

должен выводить:

[1]

[1, 5]

[1, 2, 5, 10]

# объявление функции

def get\_factors(num):

s=[]

for i in range(1,num//2+1):

if num%i==0:

s.append(i)

s.append(num)

return s

# считываем данные

n = int(input())

# вызываем функцию

print(get\_factors(n))

## Делители 2

Напишите функцию number\_of\_factors(num), принимающую в качестве аргумента число и возвращающую количество делителей данного числа.

**Примечание 1.** Используйте уже реализованную функцию get\_factors(num) из предыдущей задачи.

**Примечание 2.** Следующий программный код:

print(number\_of\_factors(1))

print(number\_of\_factors(5))

print(number\_of\_factors(10))

должен выводить:

1

2

4

# объявление функции

def number\_of\_factors(num):

n=1

for i in range(1,num//2+1):

if num%i==0:

n+=1

return n

# считываем данные

n = int(input())

# вызываем функцию

print(number\_of\_factors(n))

## Найти всех

Напомним, что строковый метод find('a') возвращает местоположение первого вхождения символа a в строке. Проблема заключается в том, что данный метод не находит местоположение всех символов а.

Напишите функцию с именем find\_all(target, symbol), которая принимает два аргумента: строку target и символ symbol и возвращает список, содержащий все местоположения этого символа в строке.

**Примечание 1.** Если указанный символ не встречается в строке, то следует вернуть пустой список.

**Примечание 2.** Следующий программный код:

print(find\_all('abcdabcaaa', 'a'))

print(find\_all('abcadbcaaa', 'e'))

print(find\_all('abcadbcaaa', 'd'))

должен выводить:

[0, 4, 7, 8, 9]

[]

[4]

# объявление функции

def find\_all(target, symbol):

n=0

s=[]

for i in target:

if i==symbol:

s.append(n)

n+=1

return s

# считываем данные

s = input()

char = input()

# вызываем функцию

print(find\_all(s, char))

## Merge lists 1

Напишите функцию merge(list1, list2), которая принимает в качестве аргументов два отсортированных по возрастанию списка, состоящих из целых чисел, и объединяет их в один отсортированный список.

**Примечание 1.** Списки list1 и list2 могут иметь разную длину.

**Примечание 2.** Можно использовать списочный метод sort(), а можно обойтись и без него 😎.

**Примечание 3.** Следующий программный код:

print(merge([1, 2, 3], [5, 6, 7, 8]))

print(merge([1, 7, 10, 16], [5, 6, 13, 20]))

должен выводить:

[1, 2, 3, 5, 6, 7, 8]

[1, 5, 6, 7, 10, 13, 16, 20]

# объявление функции

def merge(list1, list2):

s=[]

s=list1+list2

s.sort()

return s

# считываем данные

numbers1 = [int(c) for c in input().split()]

numbers2 = [int(c) for c in input().split()]

# вызываем функцию

print(merge(numbers1, numbers2))

**Слияние двух отсортированных списков**

Слияние двух отсортированных списков в один — важная задача в информатике. Она естественно возникает при сортировке списков c использованием [сортировки слиянием](https://ru.wikipedia.org/wiki/%D0%A1%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0_%D1%81%D0%BB%D0%B8%D1%8F%D0%BD%D0%B8%D0%B5%D0%BC).

Пусть даны два отсортированных по возрастанию списка чисел list1 и list2:

list1 = [3, 10, 11, 12, 47, 57, 58, 63, 77, 79, 80, 95]

list2 = [0, 11, 12, 20, 24, 26, 47, 48, 53, 65, 70, 81, 84, 84, 90]

Простейшее решение задачи слияния списков использует списочный метод sort():

def merge(list1, list2):

    result = list1 + list2 # создаем результирующий список

    result.sort() # сортируем список встроенным методом sort()

    return result # возвращаем отсортированный список

list1 = [3, 10, 11, 12, 47, 57, 58, 63, 77, 79, 80, 95]

list2 = [0, 11, 12, 20, 24, 26, 47, 48, 53, 65, 70, 81, 84, 84, 90]

list3 = merge(list1, list2) # вызываем функцию слияния двух отсортированных списков

print(list3)

Результатом работы такого кода будет список:

[0, 3, 10, 11, 11, 12, 12, 20, 24, 26, 47, 47, 48, 53, 57, 58, 63, 65, 70, 77, 79, 80, 81, 84, 84, 90, 95]

И хотя функция merge() полностью справляется с своей задачей, она абсолютно не учитывает то, что два списка list1 и list2 уже отсортированы.

**Быстрое слияние двух отсортированных списков в один**

Пусть мы имеем два уже **отсортированных по возрастанию** списка list1 и list2.

Алгоритм быстрого слияния следующий:

1. Создаем численные указатели p1 = 0 и p2 = 0 на начала обоих списков list1 и list2 соответственно;
2. На каждом шаге берем меньший из двух элементов list1[p1] и list2[p2];
3. Записываем его в результирующий список;
4. Увеличиваем на 11 указатель на первый элемент списка (p1 или p2), из которого был взят элемент;
5. Когда один из начальных списков закончился, добавляем все оставшиеся элементы второго списка в результирующий список.

def quick\_merge(list1, list2):

result = []

p1 = 0 # указатель на первый элемент списка list1

p2 = 0 # указатель на первый элемент списка list2

while p1 < len(list1) and p2 < len(list2): # пока не закончился какой-нибудь из списков

if list1[p1] <= list2[p2]:

result.append(list1[p1])

p1 += 1

else:

result.append(list2[p2])

p2 += 1

if p1 < len(list1): # прицепление остатка

result += list1[p1:]

else: # иначе прицепляем остаток другого списка

result += list2[p2:]

return result

Следующий программный код:

list1 = [3, 10, 11, 12, 47, 57, 58, 63, 77, 79, 80, 95]

list2 = [0, 11, 12, 20, 24, 26, 47, 48, 53, 65, 70, 81, 84, 84, 90]

list3 = quick\_merge(list1, list2)

print(list3)

выведет:

[0, 3, 10, 11, 11, 12, 12, 20, 24, 26, 47, 47, 48, 53, 57, 58, 63, 65, 70, 77, 79, 80, 81, 84, 84, 90, 95]

## Merge lists 2

На вход программе подается число �*n*, а затем �*n* строк, содержащих целые числа **в порядке возрастания**. Из данных строк формируются списки чисел. Напишите программу, которая объединяет указанные списки в один отсортированный список с помощью функции quick\_merge(), а затем выводит его.

**Формат входных данных**  
На вход программе подается натуральное число �*n*, а затем �*n* строк, содержащих целые числа в порядке возрастания, разделенные символом пробела.

**Формат выходных данных**  
Программа должна вывести элементы окончательного отсортированного списка каждое через пробел.

def quick\_merge(list1, list2):

s = []

p1 = 0 # указатель на первый элемент списка list1

p2 = 0 # указатель на первый элемент списка list2

while p1 < len(list1) and p2 < len(list2): # пока не закончились оба списка

if list1[p1] <= list2[p2]:

s.append(list1[p1])

p1 += 1

else:

s.append(list2[p2])

p2 += 1

if p1 < len(list1): # прицепление остатка

s += list1[p1:]

else: # иначе прицепляем остаток другого списка

s += list2[p2:]

return s

tl = [] # Результирующий список

# Считываем количество списков для сортировки

for i in range(int(input())):

num = [int(c) for c in input().split()] # Считываем список чисел

tl = quick\_merge(tl, num) # Вызываем quick\_merge() для сортировки и объединения списков

# Выводим отсортированный результирующий список

print(\*tl)

**Тема урока: функции с возвратом значения**

1. Функции с возвратом значения
2. Решение задач

**Аннотация.** Кроме функций, возвращающих числовые значения, можно писать и возвращающие логические, строковые и другие значения.

**Возвращение булевых значений**

Python позволяет писать [булевы функции](https://ru.wikipedia.org/wiki/%D0%91%D1%83%D0%BB%D0%B5%D0%B2%D0%B0_%D1%84%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D1%8F), возвращающие либо истину (True), либо ложь (False). Булеву функцию можно применять для проверки условия, тогда значения True и False будут сигнализировать о его выполнении.

Булевы функции широко применяются для упрощения сложных условий, проверяемых в структурах принятия решения и структурах с повторением.  
Например, напишем программу, которая просит пользователя ввести число и определяет, четное оно или нечетное.

Это можно сделать так:

number = int(input())

if number % 2 == 0:

print('Это число четное. ')

else:

print('Это число нечетное.')

Этот фрагмент кода будет легче понять, если написать булеву функцию is\_even(), которая принимает число в качестве аргумента и возвращает True, если оно четное, и False если нечетное.

def is\_even(number):

if number % 2 == 0:

return True

else:

return False

Теперь можно переписать инструкцию if-else основной программы так, чтобы она для определения четности переменной number вызывала функцию is\_even():

number = int(input())

if is\_even(number):

print('Это число четное. ')

else:

print('Это число нечетное.')

Так логику программы легче понять, а функцию можно вызывать в программе всякий раз, когда необходимо проверить четность числа.

**Использование булевых функций для валидации входных данных**

Булевы функции можно также использовать для ***упрощения сложного кода валидации входных данных***. Например в программе, предлагающей пользователю ввести номер модели изделия, где возможны только значения 100100, 200200 или 300300, можем написать такой код:

model = int(input())

while model != 100 and model != 200 and model != 300:

print('Допустимыми номерами моделей являются 100, 200 или 300.')

model = int(input())

Цикл валидации использует длинное составное булево выражение, и повторяется до тех пор, пока model не будет равняться 100100, 200200 или 300300.

Вместе с тем, цикл валидации можно упростить, написав булеву функцию проверки переменной model и вызывая ее в цикле. Напишем функцию is\_invalid(), которая принимает один параметр model и возвращает значение True, если модель недопустима и False в противном случае. Тогда цикл валидации можно переписать следующим образом:

while is\_invalid(model):

print('Допустимыми номерами моделей являются 100, 200 и 300.')

model = int(input())

После этого изменения цикл становится легче читать. Теперь вполне очевидно, что цикл повторяется до тех пор, пока номер модели недопустим. Приведенный ниже фрагмент кода показывает, как можно было бы написать функцию is\_invalid(). Она принимает номер модели в качестве аргумента, и если аргумент не равен 100100, 200200 или 300300, то эта функция возвращает True, говоря, что он недопустимый. В противном случае функция возвращает False.

def is\_invalid(model):

if model != 100 and model != 200 and model != 300:

return True

else:

return False

Создание функций, реализующих такую простую логику, — не всегда оптимальное решение, так как увеличивает размер кода и ведет к затратам времени на вызов функции и возврат обратно результата, что может сказаться негативно на производительности программы.

## Is Valid Triangle?

Напишите функцию is\_valid\_triangle(side1, side2, side3), которая принимает в качестве аргументов три натуральных числа, и возвращает значение True если существует невырожденный треугольник со сторонами side1, side2, side3 и False в противном случае.

**Примечание 1.** С данной [задачей](https://stepik.org/lesson/265083/step/13?unit=246031) мы уже сталкивались при изучении условного оператора.

**Примечание 2.** Следующий программный код:

print(is\_valid\_triangle(2, 2, 2))

print(is\_valid\_triangle(2, 3, 10))

print(is\_valid\_triangle(3, 4, 5))

должен выводить:

True

False

True

# объявление функции

def is\_valid\_triangle(side1, side2, side3):

side1, side2, side3=a,b,c

if a+b>c and b+c>a and a+c>b:

return True

else:

return False

# считываем данные

a, b, c = int(input()), int(input()), int(input())

# вызываем функцию

print(is\_valid\_triangle(a, b, c))

## Is a Number Prime? 🌶️

Напишите функцию is\_prime(num), которая принимает в качестве аргумента натуральное число и возвращает значение True если число является простым и False в противном случае.

**Примечание.** Следующий программный код:

print(is\_prime(1))

print(is\_prime(10))

print(is\_prime(17))

должен выводить:

False

False

True

# объявление функции

def is\_prime(num):

if num==1:

return False

for i in range(2,num):

if num%i==0:

return False

return True

# считываем данные

n = int(input())

# вызываем функцию

print(is\_prime(n))

## Next Prime 🌶️🌶️

Напишите функцию get\_next\_prime(num), которая принимает в качестве аргумента натуральное число num и возвращает первое простое число большее числа num.

**Примечание 1.** Используйте функцию is\_prime() из предыдущей задачи.

**Примечание 2.** Следующий программный код:

print(get\_next\_prime(6))

print(get\_next\_prime(7))

print(get\_next\_prime(14))

должен выводить:

7

11

17

# объявление функции

def get\_next\_prime(num):

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

count = 0

for j in range(1, num + i + 1):

if (num + i) % j == 0:

count += 1

if count == 2:

return num + i

# считываем данные

n = int(input())

# вызываем функцию

print(get\_next\_prime(n))

**Good password 🌶️**

Напишите функцию is\_password\_good(password), которая принимает в качестве аргумента строковое значение пароля password и возвращает значение True, если пароль является надежным и False - в противном случае.

Пароль является надежным если:

* его длина не менее 88 символов;
* он содержит как минимум одну заглавную букву (верхний регистр);
* он содержит как минимум одну строчную букву (нижний регистр);
* он содержит хотя бы одну цифру.

**Примечание.** Следующий программный код:

print(is\_password\_good('aabbCC11OP'))

print(is\_password\_good('abC1pu'))

должен выводить:

True

False

# объявление функции

def is\_password\_good(password):

if len(password)<8:

return False

fa,fb,fc=False,False,False

for i in password:

if i.isdigit(): # Если символ - цифра, устанавливаем флаг для цифры

fa= True

elif i.islower(): # Если символ - строчная буква, устанавливаем флаг для строчной буквы

fb= True

elif i.isupper(): # Если символ - заглавная буква, устанавливаем флаг для заглавной буквы

fc= True

return fa and fb and fc

# считываем данные

txt = input()

# вызываем функцию

print(is\_password\_good(txt))

## Ровно в одном

Напишите функцию is\_one\_away(word1, word2), которая принимает в качестве аргументов два слова word1 и word2 и возвращает значение True, если слова имеют одинаковую длину и отличаются ровно в одном символе и False  в противном случае.

**Примечание.** Следующий программный код:

print(is\_one\_away('bike', 'hike'))

print(is\_one\_away('water', 'wafer'))

print(is\_one\_away('abcd', 'abpo'))

print(is\_one\_away('abcd', 'abcde'))

должен выводить:

True

True

False

False

# объявление функции

def is\_one\_away(word1, word2):

if len(word1)!=len(word2):

return False

a=0

for i in range(len(word1)):

if word1[i]!=word2[i]:

a+=1

if a==1:

return True

else:

return False

# считываем данные

txt1 = input()

txt2 = input()

# вызываем функцию

print(is\_one\_away(txt1, txt2))

## Палиндром 🌶️

Напишите функцию is\_palindrome(text), которая принимает в качестве аргумента строку text и возвращает значение True если указанный текст является палиндромом и False в противном случае.

**Примечание 1.** Палиндром – это строка, которая читается одинаково в обоих направлениях

**Примечание 2.** При проверке считайте большие и маленькие буквы одинаковыми, а также игнорируйте пробелы, а также символы , . ! ? -.

**Примечание 3.** Следующий программный код:

print(is\_palindrome('А роза упала на лапу Азора.'))

print(is\_palindrome('Gabler Ruby - burrel bag!'))

print(is\_palindrome('BEEGEEK'))

должен выводить:

True

True

False

# объявление функции

def is\_palindrome(text):

for c in (',.!?- '):

text = text.replace(c, '')

text = text.lower()

a=0

s=text[::-1]

for i in range(len(text)):

if text[i]!=s[i]:

a+=1

if a==0:

return True

else:

return False

# считываем данные

txt = input()

# вызываем функцию

print(is\_palindrome(txt))

**BEEGEEK**

**BEEGEEK** наконец-то открыл свой банк, в котором используются специальные банкоматы с необычным паролем.

Действительный пароль BEEGEEK банка имеет вид a:b:c, где a, b и c – натуральные числа. Поскольку основатель **BEEGEEK** фанатеет от математики, то он решил:

* число a – должно быть палиндромом;
* число b – должно быть простым;
* число c – должно быть четным.

Напишите функцию is\_valid\_password(password), которая принимает в качестве аргумента строковое значение пароля password и возвращает значение True, если пароль является действительным паролем BEEGEEK банка и False - в противном случае.

**Примечание.** Следующий программный код:

print(is\_valid\_password('1221:101:22'))

print(is\_valid\_password('565:30:50'))

print(is\_valid\_password('112:7:9'))

print(is\_valid\_password('1221:101:22:22'))

должен выводить:

True

False

False

False

# Функция, которая проверяет, является ли число палиндромом

def is\_palindrome(num):

return str(num) == str(num)[::-1]

# Функция, которая проверяет, является ли число простым

def is\_prime(num):

if num == 1:

return False

for i in range(2, int(num \*\* 0.5) + 1):

if num % i == 0:

return False

return True

# Функция, которая проверяет, является ли число четным

def is\_even(num):

return num % 2 == 0

# Функция, которая проверяет валидность пароля

def is\_valid\_password(password):

l = password.split(":") # Разбиваем пароль на части, используя двоеточие как разделитель

# Проверяем, что пароль содержит три части

if len(l) == 3:

l = [int(el) for el in l] # Преобразуем каждую часть в целое число

a, b, c = l[0], l[1], l[2] # Присваиваем переменным значения из списка

# Проверяем, что первая часть (a) является палиндромом, вторая часть (b) - простым числом,

# а третья часть (c) - четным числом

return is\_palindrome(a) and is\_prime(b) and is\_even(c)

return False # Возвращаем False, если пароль не соответствует заданному формату

# считываем данные

psw = input()

# вызываем функцию

print(is\_valid\_password(psw))

## Правильная скобочная последовательность 🌶️

Напишите функцию is\_correct\_bracket(text), которая принимает в качестве аргумента непустую строку text, состоящую из символов ( и ) и возвращает значение True если поступившая на вход строка является правильной скобочной последовательностью и False в противном случае.

**Примечание 1.** Правильной скобочной последовательностью называется строка, состоящая только из символов ( и ), где каждой открывающей скобке найдется парная закрывающая скобка (при этом каждая открывающая скобка должна быть левее соответствующей ей закрывающей скобки).

**Примечание 2.** Следующий программный код:

print(is\_correct\_bracket('()(()())'))

print(is\_correct\_bracket(')(())('))

должен выводить:

True

False

# объявление функции

def is\_correct\_bracket(text):

# Пока в строке есть подстрока '()', заменяем её на пустую строку

while '()' in text:

text = text.replace('()', '')

return not text # Возвращаем True, если вся строка опустела, что означает правильную последовательность

# считываем данные

txt = input()

# вызываем функцию

print(is\_correct\_bracket(txt))

## Змеиный регистр

Напишите функцию convert\_to\_python\_case(text), которая принимает в качестве аргумента строку в [«верблюжьем регистре»](https://ru.wikipedia.org/wiki/CamelCase) и преобразует его в [«змеиный регистр»](https://ru.wikipedia.org/wiki/Snake_case).

**Примечание 1.** Почитать подробнее о стилях именования можно [тут](https://ru.wikipedia.org/wiki/CamelCase).

**Примечание 2.** Следующий программный код:

print(convert\_to\_python\_case('ThisIsCamelCased'))

print(convert\_to\_python\_case('IsPrimeNumber'))

должен выводить:

this\_is\_camel\_cased

is\_prime\_number

# объявление функции

def convert\_to\_python\_case(text):

s = '' # Инициализируем пустую строку для результата

for i in text:

if i.isupper(): # Если текущий символ - заглавная буква

s += '\_' # Добавляем символ подчеркивания перед ней

s += i.lower() # Преобразуем текущий символ в нижний регистр и добавляем к результату

return s[1:] # Возвращаем строку, начиная с первого символа (без лишнего подчеркивания)

# считываем данные

txt = input()

# вызываем функцию

print(convert\_to\_python\_case(txt))

**Тема урока: функции с возвратом значения**

1. Функции с возвратом нескольких значений
2. Решение задач

**Аннотация.** Урок посвящен функциям с возвратом нескольких значений.

**Функции с возвратом нескольких значений**

В Python функции не ограничены возвратом всего одного значения. После инструкции return можно определить много выражений, разделенных запятыми:

return выражение 1, выражение 2, выражение 3 ...

Следующий программный код определяет функцию get\_powers(num), которая принимает в качестве аргумента число num и возвращает его квадрат, куб и четвертую степень.

def get\_powers(num):

return num\*\*2, num\*\*3, num\*\*4

Результатом выполнения следующего кода:

a, b, c = get\_powers(2)

print(a)

print(b)

print(c)

будет:

4

8

16

Возвращающие несколько значений функции – исключительная особенность языка Python. В большинстве языков программирования для возврата нескольких значений используется результирующий тип данных – список, который может содержать несколько значений.

Рассмотрим еще один пример. Пусть требуется написать функцию, которая находит точку пересечения двух **непараллельных**прямых ��+��=�*ax*+*by*=*e* и ��+��=�*cx*+*dy*=*f*. Другими словами требуется решить систему уравнений:

{��+��=���+��=�{*ax*+*by*=*ecx*+*dy*=*f*​Несложно прийти к решению этой системы:

{��+��=���+��=�⟺{�=�⋅�−�⋅��⋅�−�⋅��=�⋅�−�⋅��⋅�−�⋅�⎩⎨⎧​*ax*+*by*=*ecx*+*dy*=*f*​⟺⎩⎨⎧​*x*=*a*⋅*d*−*b*⋅*cd*⋅*e*−*b*⋅*f*​*y*=*a*⋅*d*−*b*⋅*ca*⋅*f*−*c*⋅*e*​​Программный код, решающий задачу, имеет вид:

def solve(a, b, c, d, e, f):

x = (d \* e - b \* f)/(a \* d - b \* c)

y = (a \* f - c \* e)/(a \* d - b \* c)

return x, y

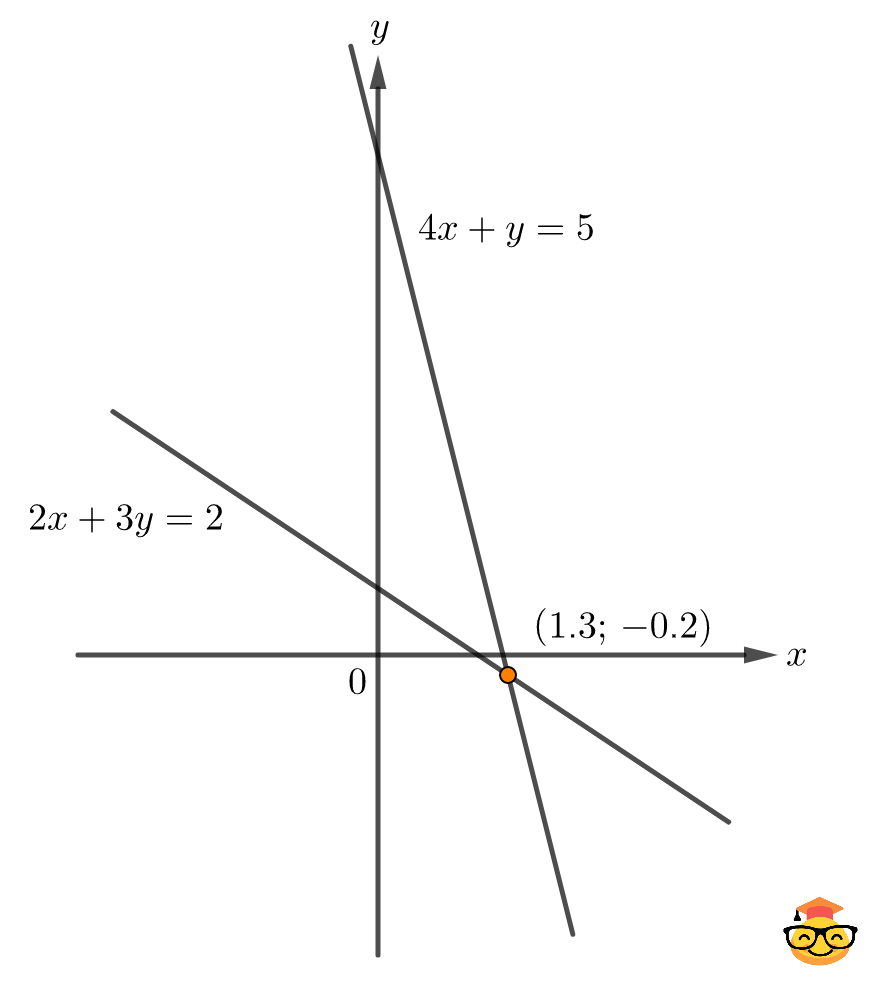
Следующий код:

xsol, ysol = solve(2, 3, 4, 1, 2, 5)

print('Решением системы являются числа', 'x =', xsol, 'y =', ysol)

выведет

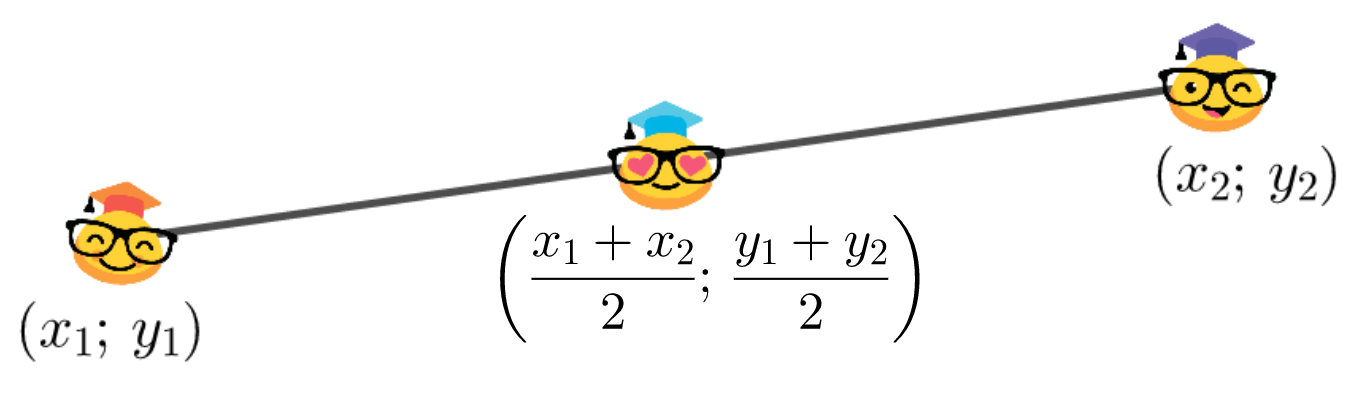
Решением системы являются числа x = 1.3 y = -0.2



## Середина отрезка

Напишите функцию get\_middle\_point(x1, y1, x2, y2), которая принимает в качестве аргументов координаты концов отрезка (�1;�1)(*x*1​;*y*1​) и (�2;�2)(*x*2​;*y*2​) и возвращает координаты точки являющейся серединой данного отрезка.

**Примечание 1.** Координаты середины отрезка с концами в точках (�1;�1)(*x*1​;*y*1​) и (�2;�2)(*x*2​;*y*2​) вычисляются по формуле:(�1+�22;�1+�22)(2*x*1​+*x*2​​;2*y*1​+*y*2​​)



**Примечание 2.** Следующий программный код:

print(get\_middle\_point(0, 0, 10, 0))

print(get\_middle\_point(1, 5, 8, 3))

должен выводить:

5.0 0.0

4.5 4.0

# объявление функции

def get\_middle\_point(x1, y1, x2, y2):

x=(x1+x2)/2

y=(y1+y2)/2

return x, y

# считываем данные

x\_1, y\_1 = int(input()), int(input())

x\_2, y\_2 = int(input()), int(input())

# вызываем функцию

x, y = get\_middle\_point(x\_1, y\_1, x\_2, y\_2)

print(x, y)

## Площадь и длина

Напишите функцию get\_circle(radius), которая принимает в качестве аргумента радиус окружности и возвращает два значения: длину окружности и площадь круга, ограниченного данной окружностью.

**Примечание 1.** Длина окружности и площадь круга радиуса �*r* вычисляются по формулам:

С=2��,�=��2.С=2*πr*,*S*=*πr*2.**Примечание 2.** Для числа �*π* используйте глобальную константу из модуля math.

**Примечание 3.** Следующий программный код:

print(get\_circle(1))

print(get\_circle(1.5))

должен выводить:

6.283185307179586 3.141592653589793

9.42477796076938 7.0685834705770345

# объявление функции

def get\_circle(radius):

from math import pi

c=2\*pi\*radius

s=pi\*radius\*\*2

return c, s

# считываем данные

r = float(input())

# вызываем функцию

length, square = get\_circle(r)

print(length, square)

## Корни уравнения 🌶️🌶️

Напишите функцию solve(a, b, c), которая принимает в качестве аргументов три целых числа a, b, c – коэффициенты квадратного уравнения ��2+��+�=0*ax*2+*bx*+*c*=0 и возвращает его корни в порядке возрастания.

**Примечание 1.** С подобной [задачей](https://stepik.org/lesson/265110/step/7?unit=246058) мы уже сталкивались.

**Примечание 2.** Гарантируется, что квадратное уравнение имеет хотя бы один корень.

**Примечание 3.** Следующий программный код:

print(solve(1, -4, -5))

print(solve(-2, 7, -5))

print(solve(1, 2, 1))

должен выводить:

-1.0 5.0

1.0 2.5

-1.0 -1.0

# объявление функции

def solve(a, b, c):

d = b \*\* 2 - 4 \* a \* c # Вычисляем дискриминант

# Вычисляем два корня уравнения

x1 = (-b - d \*\* 0.5) / (2 \* a)

x2 = (-b + d \*\* 0.5) / (2 \* a)

return min(x1, x2), max(x1, x2) # Возвращаем корни в порядке возрастания

# считываем данные

a, b, c = int(input()), int(input()), int(input())

# вызываем функцию

x1, x2 = solve(a, b, c)

print(x1, x2)

**Преимущества использования функций**

Разбивая программы на функции получаем:

* **Более простой код.** Разбитый на функции код программы проще и легче для понимания.  Несколько небольших функций намного легче читать, чем одну длинную последовательность инструкций;
* **Повторное использование кода.** Функции позволяют избежать многократного повторения кода в программе. Если какая-то операция в программе выполняется в нескольких местах, то можно один раз написать для нее функцию и затем исполнять ее, когда понадобится.
* **Более простое тестирование.**  Когда каждая задача в программе содержится в собственной функции, программисты могут индивидуально протестировать каждую функцию и определить, выполняет ли она свою задачу правильно.
* **Более быстрая разработка.** Предположим, что программист или команда программистов разрабатывает много программ. Они обнаруживают общие задачи у разных программ, например выяснение имени пользователя и пароля, вывод текущего времени. Каждый раз писать программный код для этих задач не имеет смысла. Для часто встречающихся задач пишут функции, и включают в состав любой нуждающейся в них программы.
* **Упрощение командной работы.** Когда программа разрабатывается как набор функций, разным программистам можно поручить написание отдельных функций.

**Что выделять в функции?**

В функцию можно выделить любой законченный фрагмент программы. Можно ориентироваться на рекомендации:

* Когда несколько раз пишете в программе одну и ту же последовательность команд, необходимость введения функции приобретает характер острой внутренней потребности;
* Иногда обилие мелочей заслоняет главное и полезно убрать в функцию подробности, скрывающие смысл основной программы;
* Полезно разбить длинную программу на составные части, как книгу разбивают на главы, при этом основная программа становится похожей на оглавление;
* Сложные частные алгоритмы бывает полезно отладить отдельно в небольших тестирующих программах. Включить их в основную программу будет легко, если они оформлены в виде функций. Например, функций сортировок;
* Сделанное хорошо в одной программе, хочется перенести в новые. Для повторного использования лучше сразу выделять в программе полезные алгоритмы в отдельные функции, а функции собирать в пакеты.

## Биномиальный коэффициент 🌶️

Напишите функцию compute\_binom(n, k), которая принимает в качестве аргументов два натуральных числа n и k и возвращает значение [биномиального коэффициента](https://ru.wikipedia.org/wiki/%D0%91%D0%B8%D0%BD%D0%BE%D0%BC%D0%B8%D0%B0%D0%BB%D1%8C%D0%BD%D1%8B%D0%B9_%D0%BA%D0%BE%D1%8D%D1%84%D1%84%D0%B8%D1%86%D0%B8%D0%B5%D0%BD%D1%82), равного �!�!(�−�)!*k*!(*n*−*k*)!*n*!​.

**Примечание 1.** Факториалом натурального числа �*n*, называется произведение всех натуральных чисел от 11 до �*n*, то есть �!=1⋅2⋅3⋅…⋅�*n*!=1⋅2⋅3⋅…⋅*n*

**Примечание 2.** Реализуйте вспомогательную функцию factorial(n), вычисляющую факториал числа, или воспользуйтесь уже готовой функцией из модуля math. Обратите внимание, что функция compute\_binom(n, k) должна возвращать целое число.

# объявление функции

def compute\_binom(n, k):

from math import factorial

b=int(factorial(n)/(factorial(k)\*factorial(n-k)))

return b

# считываем данные

n = int(input())

k = int(input())

# вызываем функцию

print(compute\_binom(n, k))

# объявление функции факториала def factorial(m): c = 1 for i in range(1, m+1): c \*= i return c # объявление функции бинома def compute\_binom(n, k): return (factorial(n)/(factorial(k)\*(factorial(n-k)))) # считываем данные n = int(input()) k = int(input()) # вызываем функцию print(int(compute\_binom(n, k)))

## Число словами 🌶️

Напишите функцию number\_to\_words(num), которая принимает в качестве аргумента натуральное число num и возвращает его словесное описание на русском языке.

**Примечание 1.** Считайте, что число 1≤��� ≤991≤*num* ≤99.

**Примечание 2.** Следующий программный код:

print(number\_to\_words(7))

print(number\_to\_words(85))

должен выводить:

семь

восемьдесят пять

# объявление функции

def number\_to\_words(num):

list\_1 = ['один', 'два', 'три', 'четыре', 'пять', 'шесть', 'семь', 'восемь', 'девять', 'десять']

list\_11 = ['одиннадцать', 'двенадцать', 'тринадцать', 'четырнадцать', 'пятнадцать', 'шестнадцать', 'семнадцать', 'восемнадцать', 'девятнадцать']

list\_21 = ['двадцать', 'тридцать', 'сорок', 'пятьдесят', 'шестьдесят', 'семьдесят', 'восемьдесят', 'девяносто']

if num < 11:

return list\_1[num-1]

elif 10 < num <20:

return list\_11[num-11]

else:

if num % 10 == 0:

return list\_21[num//10-2]

else:

return list\_21[num // 10 - 2] + ' ' + list\_1[num%10-1]

# считываем данные

n = int(input())

# вызываем функцию

print(number\_to\_words(n))

## Искомый месяц

Напишите функцию get\_month(language, number), которая принимает на вход два аргумента language – язык ru или en и number – номер месяца (от 1 до 12) и возвращает название месяца на русском или английском языке.

**Примечание.** Следующий программный код:

print(get\_month('ru', 1))

print(get\_month('ru', 12))

print(get\_month('en', 1))

print(get\_month('en', 10))

должен выводить:

январь

декабрь

january

october

# объявление функции

def get\_month(language, number):

en = ['','january', 'february', 'march', 'april', 'may', 'june', 'july', 'august', 'september', 'october', 'november', 'december']

ru = ['', 'январь', 'февраль', 'март', 'апрель', 'май', 'июнь', 'июль', 'август', 'сентябрь', 'октябрь', 'ноябрь', 'декабрь']

if language=='ru':

return ru[number]

else:

return en[number]

# считываем данные

lan = input()

num = int(input())

# вызываем функцию

print(get\_month(lan, num))

## Магические даты

Магическая дата – это дата, когда день, умноженный на месяц, равен числу, образованному последними двумя цифрами года.

Напишите функцию is\_magic(date), которая принимает в качестве аргумента строковое представление корректной даты и возвращает значение True, если дата является магической и False - в противном случае.

**Примечание.** Следующий программный код:

print(is\_magic('10.06.1960'))

print(is\_magic('11.06.1960'))

должен выводить:

True

False

# объявление функции

def is\_magic(date):

d=int(date[:2])

m=int(date[3:5])

y=int(date[8:])

if d\*m==y:

return True

else:

return False

# считываем данные

date = input()

# вызываем функцию

print(is\_magic(date))

# объявление функции def is\_magic(date): seq = date.split(".") day, month, year = int(seq[0]), int(seq[1]), int(seq[2]) return day \* month == year % 100 # считываем данные date = input() # вызываем функцию print(is\_magic(date))

## Панграммы

Панграмма – это фраза, содержащая в себе все буквы алфавита. Обычно панграммы используют для презентации шрифтов, чтобы можно было в одной фразе рассмотреть все [глифы](https://ru.wikipedia.org/wiki/%D0%93%D0%BB%D0%B8%D1%84).

Напишите функцию, is\_pangram(text) которая принимает в качестве аргумента строку текста на английском языке и возвращает значение True если текст является панграммой и False в противном случае.

**Примечание 1.** Гарантируется, что введенная строка содержит только буквы английского алфавита и пробелы.

**Примечание 2.** Следующий программный код:

print(is\_pangram('Jackdaws love my big sphinx of quartz'))

print(is\_pangram('The jay pig fox zebra and my wolves quack'))

print(is\_pangram('Hello world'))

должен выводить:

True

True

False

# объявление функции

def is\_pangram(text):

if len(set(text.lower()))==27:

return True

else:

return False

# считываем данные

text = input()

# вызываем функцию

print(is\_pangram(text))

# объявление функции def is\_pangram(text): text = text.lower() for i in range(ord("a"), ord("z") + 1): if chr(i) not in text: return False return True # считываем данные text = input() # вызываем функцию print(is\_pangram(text))