Модуль 6. Функции

[Функции: определение def и их вызов 1](#_Toc151502905)

[Простые задания на функции 5](#_Toc151502906)

[Функции с произвольным числом параметров \*args 7](#_Toc151502907)

[Позиционные и именованные параметры 9](#_Toc151502908)

[Операторы \* и \*\* для упаковки и распаковки коллекций 11](#_Toc151502909)

[Рекурсивные функции 15](#_Toc151502910)

[Задания на рекурсивные функции 16](#_Toc151502911)

[Анонимные (lambda) функции 18](#_Toc151502912)

[Аргумент key для сортировки коллекций по ключу 21](#_Toc151502913)

[Функция filter 24](#_Toc151502914)

[Задания на функции с именованными, фактическими и формальными параметрами 25](#_Toc151502915)

[Изменение области видимости переменных 28](#_Toc151502916)

[Организация программы и функция main 29](#_Toc151502917)

[Подсказки типов Hinting и doc strings 29](#_Toc151502918)

[Функция как параметр функции 30](#_Toc151502919)

[Функция как тип 33](#_Toc151502920)

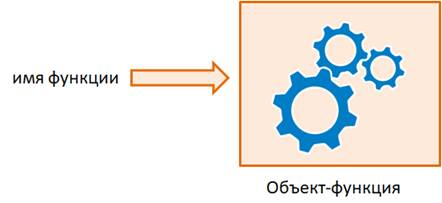
[Области видимости 34](#_Toc151502921)

[Прочее 37](#_Toc151502922)

[Домашнее задание 39](#_Toc151502923)

# Функции: определение def и их вызов

Этим занятием мы открываем с вами одну из ключевых тем в программировании – функции.



Если в интерактивном режиме напечатать имя print, то увидим, что оно ссылается на встроенную функцию. Чтобы активизировать программу, заложенную в объект-функцию, нужно поставить круглые скобки после имени функции:

print()

Теперь был исполнен определенный код и в консоль была выведена пустая строка. Эти круглые скобки являются оператором вызова функций.

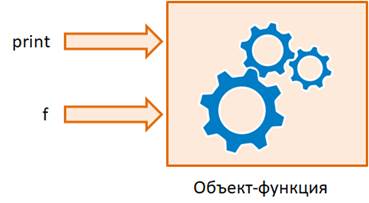
То есть, само по себе имя – это ссылка на функцию, а имя функции с круглыми скобками – это уже вызов функции. Вот этот момент нужно очень хорошо и четко понимать, чтобы уметь правильно обращаться с функциями при построении программ.

Итак, раз print – это ссылка на функцию, то что нам тогда мешает создать еще одну ссылку на этот же объект-функцию, через оператор присваивания:

f = print

И вызвать ее через второе имя f:

f("hello")



Как видите, все работает. Теперь функцию print можно вызывать и через f. А можно ли тогда исходному имени функции print присвоить другой объект, например, строку:

print = "это была функция print"

Да, и это нам никто не мешает делать. Теперь прежнюю функцию можно вызвать только через имя f:

f(print)

То есть, имя функции – это просто ссылка на объект-функцию и не более того. Конечно, заменять стандартные имена функций на другие, без острой необходимости не стоит.

А вначале лучше совсем этого не делать. Именно этого правила и следует придерживаться.

Хорошо, мы теперь знаем, как правильно воспринимать функцию в Python. Но зачем они нужны? Разве нельзя писать программы и без них? Конечно можно. И самые первые языки программирования их даже не имели.

Такое дублирование программного кода негативно влияет на надежность программ, возможность их быстрого редактирования, читабельности и т.п. В программирование это называется кратко:

DRY – Don’t Repeat Yourself

Как только, вы замечаете в программе повторение (дублирование) кода, значит, неверно строите ее архитектуру и структуру кода следует пересмотреть. Это очень грубая ошибка в программировании.

Так вот, чтобы нам устранить дублирование при отправке электронных писем, повторяющийся фрагмент можно поместить в функцию, а затем, вызывать ее в нужных местах программы. Это первое, для чего были введены функции в языки программирования – устранять дублирование кода.

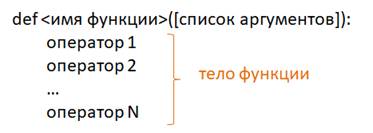
В Python существует множество стандартных, встроенных функций, но на все случаи жизни их не придумаешь. Поэтому, программист может сам создавать свои собственные по мере необходимости. Для этого используется следующий синтаксис:

def <имя функции>([список аргументов]):  
        оператор 1  
        оператор 2  
        …  
        оператор N

Здесь имя функции придумывается программистом подобно именам переменных и, так как функция – это определенное действие, то ее имя следует выбирать как глагол, например:

go, show, get, set и т.п.

Далее, идет набор операторов, которые образуют тело функции. Именно они начинают выполнятся при ее вызове.



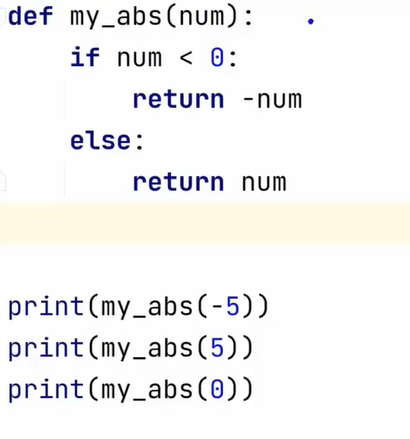
Функция должна быть черной коробочкой. В функции не должно быть интерактива. Ввода-вывода. Взаимодействия с пользователем. Если это конечно не специализированная функция для ввода-вывода! Черная коробка. Функция которая отправляет почту не должна спрашивать адрес у отправителя и каким способом она ее будет доставлять почтовыми голубями иди ездовыми собаками. Приняла и отправила.

**Пример 1**

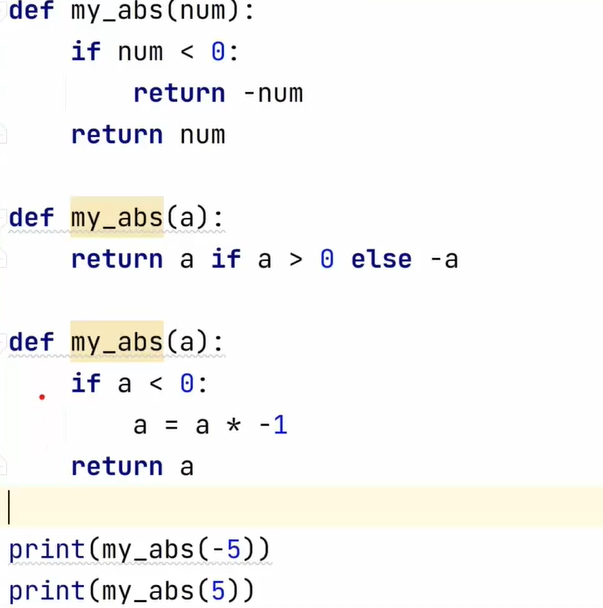
def sum2(a, b):  
 c = a + b  
 return c  
  
  
result = sum2(5, 2)  
print(result)

result = sum2(sum2(4, -2), 7) + 12 # Каков результат?  
  
print(result)

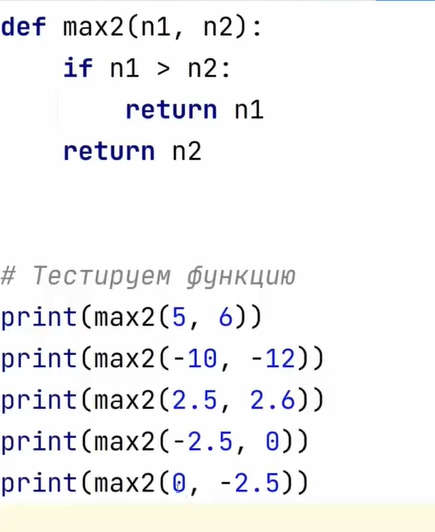
**Пример 2 Реализовать функцию получения модуля числа**



Как еще можно реализовать функцию abs



А что внутри у функции max2



## Простые задания на функции

**Задание 1. Паллиндром.** Необходимо определить, является ли это слово палиндромом (одинаково читается вперед и назад, например, АННА) с помощью функции. Если палиндром, на экран вывести True, иначе - False.

**Задание 2.** Объявите функцию с именем is\_triangle, которая принимает три стороны треугольника (целые числа) и проверяет, можно ли из переданных аргументов составить треугольник. (Напомню, что у любого треугольника длина третьей стороны всегда должна быть меньше суммы двух других). Если  проверка проходит, вернуть булево значение True, иначе - значение False.

Вызывать функцию не нужно, только задать.

**Входные данные**

3 4 5

**Выходные данные:**

True

def is\_triangle(a: int, b: int, c: int) -> bool:

""" Функция принимает три стороны треугольника и проверяет,

существует ли треугольник с такими сторонами """

a, b, c = sorted([a, b, c])

return c < a + b

**Задание 3.**Объявите функцию для проверки числа на нечетность (возвращается True, если переданное число нечетное и False, если число четное).

После объявления функции обработайте с её помощью список целых значений,И, используя генератор списков и созданную функцию, сформируйте список из нечетных значений на основе введенного исходного списка. Результат отобразите на экране командой:

print(\*lst)

где lst - сформированный список из нечетных значений.

**Входные данные**

st = [8, 11, -15, 3, 2, 10]

**Выходные данные:**

11 -15 3

def is\_odd(num: int) -> bool:

""" Функция проверяет число на нечётность """

return bool(num % 2)

lst = [x for x in nums if is\_odd(x)] # С помощью генератора списков и созданную функцию получаем новый список

print(\*lst)

**Задание 4.** Напишите функцию, которая проверяет корректность переданного ей email-адреса в виде строки. Будем полагать, что адрес верен, если он обязательно содержит символы '@' и '.', а все остальные символы могут принимать значения: 'a-z', 'A-Z', '0-9' и '\_'. Если email верен, то функция выводит ДА, иначе - НЕТ.

После объявления функции прочитайте (с помощью функции input) строку с email-адресом и вызовите функцию с этим аргументом.

**Входные данные**

sc\_lib@list.ru

**Выходные данные:**

ДА

chars = string.digits + string.ascii\_uppercase + string.ascii\_lowercase + '\_'  
print(chars)  
  
def checkEmail(email):  
 if '@' in email and '.' in email:  
 email = email.replace('@', '').replace('.', '')  
 for char in email:  
 if not char in chars:  
 break

else:  
 return True #print  
 return False #print  
 else:  
 return False #print  
  
  
print (checkEmail('sc\_lib@list.ru')

## Функции с произвольным числом параметров \*args

На этом занятии узнаем, как функциям можно передавать произвольное число аргументов. Где это используется, я думаю, вы понимаете? Например, известная нам функция sum (1,2,3,4,5) или max(1, 2, 3, -4)

def my\_sum(\*x):  
 s = 0  
 for i in x:  
 s += i  
 return s  
  
  
y = my\_sum(1, 2, 3, 4, 5)  
print(y)

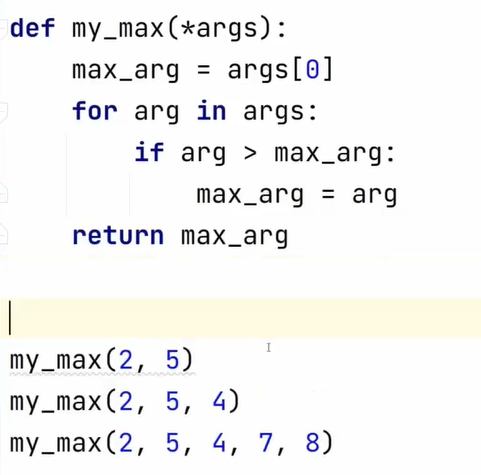
А как передать список или кортеж в нашу my\_sum

s = (1, 2, 3, 4, 5)  
y = my\_sum(\*s)  
print(y)

Или так

def my\_sum(x)

А как реализовать поиск максимально числа в последовательности



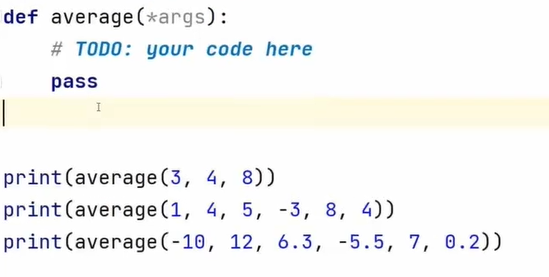
Задание. В стандартном Python отсутствует функция получения среднего числа последовательности.

a) Разработать такую функцию принимающую любую арифметическую последовательность и возвращающую среднее число

б) Реализовать округление результата с заданной точность(передаваемой также в функцию в виде именованного параметра)

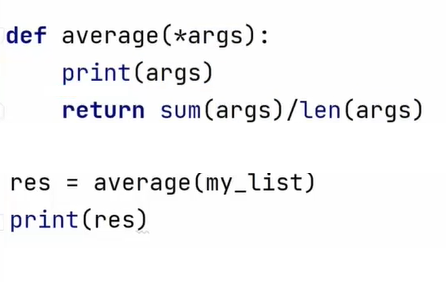
Решение

def average(\*args, accuracy=3):  
 s = sum(args) / len(args)   
 return round(s, accuracy)  
  
  
s = (1, 2, 3, 4, 5)  
  
y = average(\*s)  
print(y)



А сейчас

Проблема !



Решение



## Позиционные и именованные параметры

Передачу произвольных параметров \*args мы рассмотрели

Добавим энтропии в нашу функцию

def average(\*args, accuracy=3, indent = 10):  
 s = sum(args) / len(args)  
 s = str(round(s, accuracy))  
 s = s.rjust(indent, '\*')  
 return s

s = (1, 2, 3, 4, 5)  
  
y = average(\*s, 3, 10)  
print(y)

Работает!

у1 = average(\*s, indent=5, accuracy=4)

И так работает

А сейчас!

y2 = average(\*s, 3, 10)  
y3 = average(\*s, 5, accuracy=4)  
print(y2) # 4.0  
print(y3) # 3.3333

Что произошло? Почему 3.333

А теперь снова всё ОК или не ок

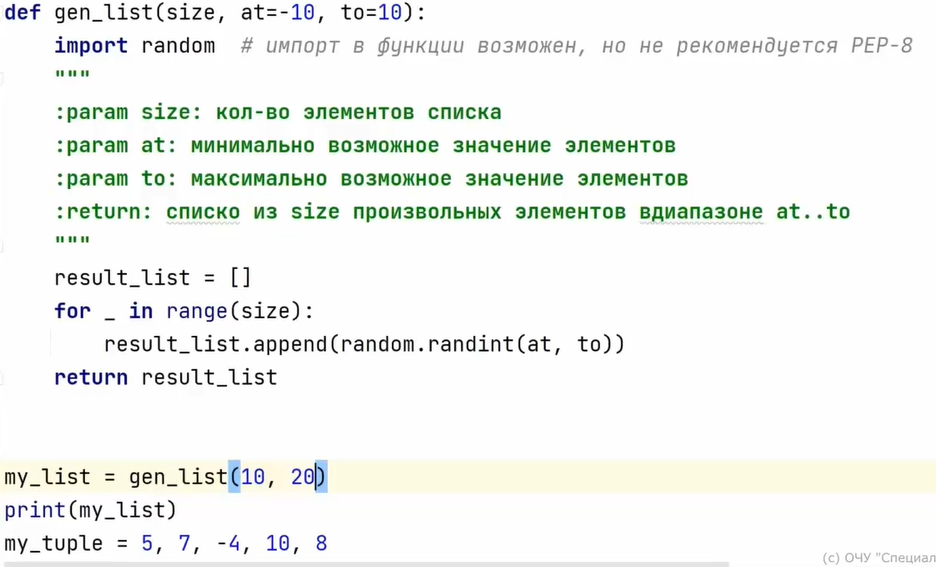
y3 = average(\*s, 5, indent=4)

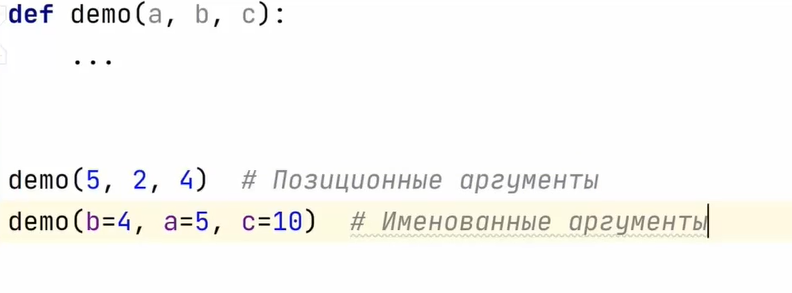
print(y3) # 3.333

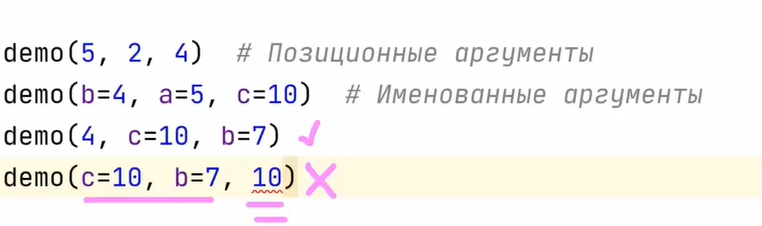
почти, чего не хватает

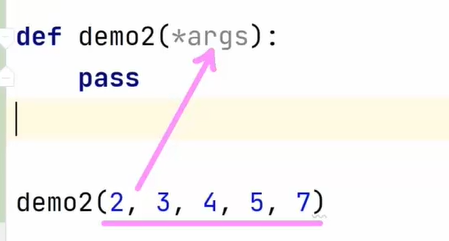
Разбираемся

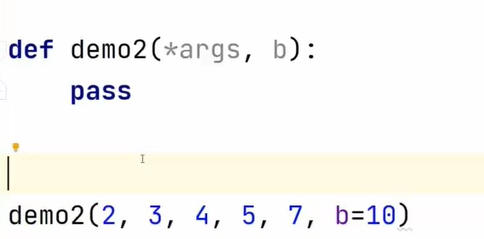
y = average(\*s) *# произвольные аргументы*y1 = average(\*s, 5, 10) *# произвольные и позиционные (он съел позиционные)*y2 = average(\*s, 5, accuracy=10) *#произвольные, позиционные, именованные*y3 = average(\*s, accuracy=5, indent=10) *# произвольные и именованные*y4 = average(\*s, accuracy=5, 10) *# ошибка*

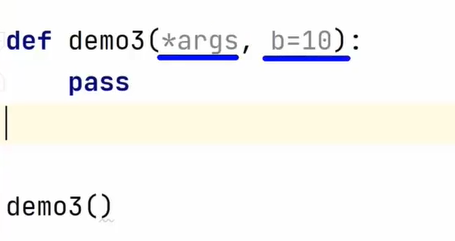












# Операторы \* и \*\* для упаковки и распаковки коллекций

Повторим об операторах \* и \*\*, которые мы затронули.

Одной из распространенных сфер, где применяются упаковка и распаковка - это параметры функций. Так, в определениях различных функций нередко можно увидеть, что они принимают такие параметры как \*args и \*\*kwargs.

Термины args и kwargs — это соглашения по программированию на Python, в реальности вместо них можно использовать любые именования. \*args представляет параметры, которые передаются по позиции. А \*\*kwargs означает параметры, которые передаются по имени. обозначает аргументы ключевого слова.

#правило

# args – кортеж всех позиционных аргументов (любое кол-во

# \*\* kwargs – словарь всех именнованных аргументов

# сначала позиционные, потом именнованные

Оператор \* применяется с любым итерируемым объектом (например, кортежем, списком и строками). Тогда как оператор \*\* можно использовать только со словарями.

\*args

Оператор \* позволяет передать в функцию несколько значений, и все они будут упакованы в кортеж:

def fun(\*args):

# обращаемся к первому элементу кортежа

print(args[0])

# выводим весь кортеж

print(args)

fun("Python", "C++", "Java", "C#")

Здесь функция fun принимает кортеж значений. При вызове мы можем передать ей различное количество значений. Так, в примере выше передается четыре строки, которые образуют кортеж. Консольный вывод программы:

Python

('Python', 'C++', 'Java', 'C#')

Оператор \*\*

Оператор \*\* упаковывает аргументы, переданные по имени, в словарь. Имена параметров служат ключами. Например, определим функцию, которая просто будет выводить все переданные параметры

def fun(\*\*kwargs):

print(kwargs) # выводим словарь на консоль

fun(name="Tom", age="38", company="Google")

fun(language="Python", version="3.11")

Консольный вывод программы:

{'name': 'Tom', 'age': '38', 'company': 'Google'}

{'language': 'Python', 'version': '3.11'}

Сочетание параметров

Параметры \*args и \*kwargs могут использоваться в функции вместе с другими параметрами. Например:

def sum(num1, num2, \*nums):

result=num1+num2

for n in nums:

result += n

return result

print(sum(1,2,3)) # 6

print(sum(1,2,3,4)) # 10

Устное программирование. Как решить такую задачу?

Предлагаю с конца

**def** f(a1, a2, \*args, \*\*kwargs):  
 print(a1, a2, args, kwargs, sep=**'\n'**)  
  
  
f(**'a1'**, **"a2"**, **"b3"**, **'b4'**, [**'b5'**, **'b6'**], c1=**'1'**, c2=**'2'**, c3=(1, 2, 3), c4=[5, 6, 7])

Результат:

a1

a2

('b3', 'b4', ['b5', 'b6'])

{'c1': '1', 'c2': '2', 'c3': (1, 2, 3), 'c4': [5, 6, 7]}

Мы знаем, что \* и \*\* позволяют упаковывать аргументы в кортеж и словарь. Но их можно использовать не только в объявлении функций, но и при работе с разными коллекциями. Например, если взять кортеж из двух значений:

x, y = (1, 2)

то его можно распаковать в две переменные. Но, если мы пропишем там больше значений, например, четыре:

x, y = (1, 2, 3, 4)

то уже получим ошибку, так как элементов четыре, а переменных всего две. Но, используя оператор \*, мы можем упаковать оставшиеся значения во вторую переменную:

x, \*y = (1, 2, 3, 4)

или, в первую, без разницы:

\*x, y = (1, 2, 3, 4)

То же самое можно проделывать и со списками:

x, \*y = [1, "a", True, 4]

и строками:

\*x, y, z = "Hello Python!"

И вообще с любыми итерируемыми объектами. То есть, оператор \* упаковывает оставшиеся значения в список. Правда, мы не можем упаковывать уже упакованные данные, например, так:

\*y = 1, 2, 3

произойдет ошибка, так кортеж нельзя упаковать. но вот так:

x, \*y = 1, 2, 3

уже будет работать.

Этот же оператор может выполнять и обратную операцию – распаковывать коллекции в набор данных. Пусть у нас имеется список:

a = [1, 2, 3]

И на его основе мы хотим сформировать кортеж. Если просто записать переменную в круглых скобках:

(a,)

то увидим кортеж со списком внутри. Но, если прописать оператор \* перед списком:

(\*a,)

то произойдет распаковка его элементов и список превратится в кортеж. То же самое можно сделать и при вызове функций.

Закрепление \*args, \*\* kwargs

def get\_average(student, \*оценки):  
 print(f"{student} средний балл {sum(оценки) / len(оценки)}")  
  
  
get\_average("Мария", 4, 4, 5, 4, 5)  
  
  
def print\_date(name = 'Студент', \*\*данные):  
 print(name, \*данные.values())  
   
  
  
print\_date(name = 'Максим',age = 30,job = 'техлид' )

# Рекурсивные функции

Речь будет идти о рекурсивных функциях. То есть, о функциях, которые вызывают сами себя.

На первый взгляд может показаться невероятным, что функция может вызывать саму себя. Но это так. И, кстати, так можно делать во всех современных языках программирования, а не только в Python. Поэтому, изучая рекурсивные функции, вы, фактически, знакомитесь с  фундаментальным материалом, который применим и в других языках программирования.

Чтобы это все лучше улеглось, приведу два классических примера использования рекуррентных функций. Первый – это вычисление факториала натурального числа n. Можно заметить, что:

n! = (n-1)! \* n

и так далее:

(n-1)! = (n-2)! \* (n-1)

…

2! = 1 \* 2

1! = 1

Чем нам это может помочь? Если ввести функцию:

fact(n) = n!

То этот факториал можно вычислить по рекурсии:

fact(n) = n \* fact(n-1)

Именно ее мы сейчас и реализуем:

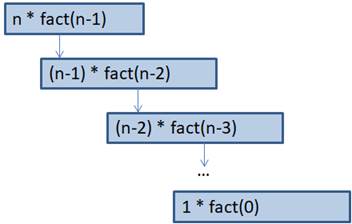
def fact(n):

    if n <= 0:

        return 1

    else:

        return n \* fact(n-1)



Вначале проверяем, если параметр n меньше или равен нулю, то возвращаем единицу, тогда fact(0) будет равен 1. А иначе идем по рекурсии, уменьшая значение n, то есть, мы будем устремляться к нулю. В итоге, у нас будет формироваться последовательность:

n \* (n-1) \* (n-2) \* … \* 1

Запустим эту функцию при n = 6 и видим в консоли значение 720, что верно:

res = fact(6)

print(res)

## Задания на рекурсивные функции

Задание 1. Алгоритм вычисления значения функции F(n), где n  — натуральное число, задан следующими соотношениями:

F(1) = 1;

F(n) = F(n − 1) + n при n>1

Чему равно значение функции F(5), F(30)

Ответ: 465.

def F(n):

if n == 1:

return 1

if n > 1:

return F(n - 1) + n

print(F(30))

Задание 2. Последовательность чисел Фибоначчи задается рекуррентным соотношением:

F(1) = 1

F(2) = 1

F(n) = F(n–2) + F(n–1), при n >2, где n – натуральное число.

Чему равно восьмое число в последовательности Фибоначчи?

В ответе запишите только натуральное число.

def F(n):

if n == 1:

return 1

if n == 2:

return 1

if n > 2:

return F(n-2) + F(n-1)

print(F(8))

21

Задача 2. (Сделать на основе первой задачи) Последовательность чисел трибоначчи задается рекуррентным соотношением:

F(1) = 0

F(2) = 1

F(3) = 1

F(n) = F(n–3) + F(n–2) + F(n–1), при n >3, где n – натуральное число.

Чему равно девятое число в последовательности трибоначчи?

В ответе запишите только натуральное число.

def F(n):

if n == 1:

return 0

if n == 2:

return 1

if n == 3:

return 1

if n > 3:

return F(n-3) + F(n-2) + F(n-1)

print(F(9))

44

Алгоритм вычисления значения функции F(n), где n  — целое неотрицательное число, задан следующими соотношениями:

F(0) = 0;

F(n) = F(n / 2), если n > 0 и при этом чётно;

F(n) = 1 + F(n − 1), если n нечётно.

Сколько существует таких чисел n, что 1 ≤ n ≤ 1000 и F(n)  =  3?

def f(n):

if n == 0:

return 0

if n > 0 and n % 2 == 0:

return f(n / 2)

if n % 2 != 0:

return 1 + f(n - 1)

k = 0

for n in range(1, 1001):

if f(n) == 3:

k += 1

print(k)

120

# Анонимные (lambda) функции

Это занятие будет посвящено, так называемым, анонимным или, их еще называют, lambda функциям. Что это за функции и почему их называют анонимными? Сейчас вы все узнаете!

Но для начала вспомним как работает функция map

s = 'Hello Python from Russia'  
  
res = s.split()  
  
print(res)

#Результат

['Hello', 'Python', 'from', 'Russia']

А как получить длины слов

res = list(map(len, s.split()))  
  
print(res)

А сейчас?

def my\_len2(x):return len(x)

res = list(map(my\_len2, s.split()))

А можно сделать вот так

my\_len1 = lambda string: len(string)

res = list(map(my\_len1, s.split()))

Или даже так

res = list(map(lambda x: len(x), s.split()))

Что происходит, посмотрим на второй пример:

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

res = list(map(lambda x: x \*\* 2, s))  
  
print(res)

Результат

[1, 4, 9, 16, 25]

Похожий результат дает генератор!

res2 = [x \*\* 2 for x in s]

А давайте вынесем нашу ламбаду

lmd = lambda x: x \*\* 2  
  
res1 = list(map(lmd, s))

понятно как работает? Закрепим!

s = ['Hello', 'Python', 'from', 'Russia']  
  
  
  
my\_len1 = lambda string: len(string)  
  
def my\_len2(x):return len(x)  
  
res1 = list(map(my\_len1, s))  
res2 = list(map(my\_len2, s))  
  
print(res1)  
print(res2)

Лямбда функция определяется по очень простому синтаксису:

lambda param\_1, param\_2, …: команда

и записывается в программе, как оператор. Например, вот так можно определить лямбда-функцию для сложения двух значений:

lambda a, b: a + b

У нее два параметра a и b, а затем, через двоеточие написано, что с ними нужно сделать. Полученное значение будет автоматически возвращено этой функцией. И, обратите внимание, у этой функции нет имени, поэтому она и называется анонимной. Но как тогда ее вызывать? Для этого объект-функцию, который создает лямбда-функция, нужно присвоить какой-либо переменной:

s = lambda a, b: a + b

И уже через нее вызывать саму функцию:

s(1, 2)

В чем особенность такого определения функции? Зачем ее придумали и почему бы не пользоваться обычными функциями? У нее есть одно принципиальное отличие от ранее рассматриваемых нами функций – она может быть записана как элемент любой конструкции языка Python. Например, прямо как элемент списка:

a = [4, 5, lambda: print("lambda"), 7, 8]

Мы здесь описываем лямбда-функцию и сразу же передаем ее в список. С обычными функциями так не получится. Они должны быть объявлены заранее и только потом мы могли бы передать ссылку на нее в список. Если мы сейчас обратимся к третьему элементу этого списка:

a[2]

то увидим ссылку на объект-функцию. И, как мы уже знаем, чтобы ее запустить на выполнение нужно прописать оператор – круглые скобки:

a[2]()

И, действительно, функция была выполнена и в консоли отобразилось сообщение.

Однако, у таких функций есть одно существенное ограничение – в них можно прописать только одну конструкцию языка Python, то есть, выполнить только одну какую-либо команду. Также нельзя объявлять анонимные функции в несколько строк:

lambda a:

    print(a)

вызовет синтаксическую ошибку. Наконец, в лямбда-функциях нельзя использовать оператор присваивания:

lambda a: a = 1

Это также вызовет синтаксическую ошибку. Здесь мы можем лишь создавать новый объект на основе входных параметров (или глобальных, общих переменных программы):

s = lambda a: a + 1

s(1)

p = lambda: "hello python"

p()

либо просто возвращать ссылки на уже существующие объекты:

lambda a: a

Хотя лямбда-выражения позволяют немного сократить определения функций, тем не менее они ограничены тем, что они могут выполнять только одно выражение.

Однако они могут быть довольно удобны в тех случаях, когда необходимо использовать функцию для передачи в качестве параметра или возвращения в другой функции.

## Аргумент key для сортировки коллекций по ключу

На этом занятии мы увидим, как можно управлять алгоритмом сортировки с помощью специального параметра key, который имеется у метода sort() и функции sorted().

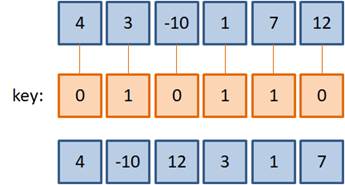
По умолчанию сортировка коллекции выполняется по значениям ее элементов. Например:

a = [4, 3, -10, 1, 7, 12]

b = sorted(a)

print(b)

Но мы можем вместо этих значений указать другие, которые будут использованы для сортировки элементов. Например, вычислить показатель четности значений. Тогда последовательность будет выстроена по возрастанию этих ключей. В результате увидим сначала четные значения, а потом – нечетные.



Чтобы выполнить такую манипуляцию в функции sorted() прописывается аргумент key и ему присваивается ссылка на функцию, которая будет формировать альтернативное значение элемента, то есть, ключ:

b = sorted(a, key=is\_odd)

А саму функцию можно определить, следующим образом:

def is\_odd(x):

    return x % 2

Здесь аргумент x – это текущее значение элемента коллекции, а то, что она возвращает, становится значением соответствующего ключа. То есть, для четных значений будем иметь 0, а для нечетных – 1. После запуска программы видим искомый результат сортировки.

Конечно, для простых функций, обычно, в аргументе key записывают лямбда-функцию. В нашем примере она будет выглядеть, следующим образом:

b = sorted(a, key=lambda x: x % 2)

Как видите, все достаточно просто.

Тот же самый аргумент key можно указывать и в методе sort() для списка:

a.sort(key=lambda x: x % 2)

Он здесь работает абсолютно также, как и в функции sorted().

Я, думаю, что вы в целом поняли, как использовать аргумент key для управления сортировкой. И здесь, как всегда, извечный вопрос – зачем это надо? Давайте я приведу несколько более практичных примеров. Предположим, что у нас имеется список городов:

lst = ["Москва", "Тверь", "Смоленск", "Псков", "Рязань"]

И требуется их выстроить по длине. Для этого воспользуемся функцией sorted() и в аргументе key укажем стандартную функцию len:

print( sorted(lst, key=len) )

получим следующий результат:

['Тверь', 'Псков', 'Москва', 'Рязань', 'Смоленск']

Или можно сделать сортировку по последнему символу слова:

print( sorted(lst, key=lambda x: x[-1]) )

['Москва', 'Псков', 'Смоленск', 'Тверь', 'Рязань']

Или только по первому:

print( sorted(lst, key=lambda x: x[0]) )

['Москва', 'Псков', 'Рязань', 'Смоленск', 'Тверь']

И так далее.

Аргумент key часто используют для сортировки сложных структур данных. Допустим, у нас имеется вот такой кортеж, содержащий вложенные кортежи с информацией по книгам:

books = (

    ("Евгений Онегин", "Пушкин А.С.", 200),

    ("Муму", "Тургенев И.С.", 250),

    ("Мастер и Маргарита", "Булгаков М.А.", 500),

    ("Мертвые души", "Гоголь Н.В.", 190)

)

И нам нужно его отсортировать по цене (последнее значение). Очевидно, это можно сделать так:

print( sorted(books, key=lambda x: x[2]) )

На выходе получим отсортированный список:

[('Мертвые души', 'Гоголь Н.В.', 190), ('Евгений Онегин', 'Пушкин А.С.', 200), ('Муму', 'Тургенев И.С.', 250), ('Мастер и Маргарита', 'Булгаков М.А.', 500)]

Вот так используется аргумент key для управления сортировкой элементов произвольных коллекций данных. И теперь вы знаете, как его применять в своих программах.

Рассмотрим еще один пример

На складе лежат мешки различного веса и стоимости. Вес за мешок и стоимость записаны на как натуральные числа с списке. Отсортировать мешки по стоимости за кг, при равной стоимости первыми идут мешки с наименьшим весом

S = [  
 [47, 470],  
 [50, 600],  
 [60, 480],  
 [45, 540],  
 [30, 300]  
]

S0 = [(x[1]/x[0], x[0], x[1]) for x in S]  
print(S0)  
S0.sort(key=lambda x: (x[0], x[2]))  
print(S0)

## Функция filter

S.sort( key = lambda x: (x[1]/x[0], -x[1]) )

numbers = [1, 2, 4, 5, 7, 8, 10, 11]

# функция, которая проверяет числа

def filter\_odd\_num(in\_num):

if(in\_num % 2) == 0:

return True

else:

return False

# Применение filter() для удаления нечетных чисел

out\_filter = filter(filter\_odd\_num, numbers)

print("Тип объекта out\_filter: ", type(out\_filter))

print("Отфильтрованный список: ", list(out\_filter))

**Задача 2.** Отфильтровать стоп-слова (союзы и предлоги) из строки.

# Строка со стоп-словами

stop\_list = ["в", "и", "по", "за", "на", ".", ",", "г", ":"]

string='''

Сервис по поиску работы и сотрудников

HeadHunter опубликовал подборку

высокооплачиваемых вакансий в России за ноябрь 2024 года

в Москве : на первых строчках IT-архитекторы и техлиды

'''

string = string.split()

res = filter(lambda s: s not in stop\_list, string)

res = ' '.join(res)

Задача на key

На складе лежат пакеты с углём различного веса и стоимости. Вес и стоимость записаны на каждом пакете как натуральные числа. Для транспортировки отбираются K пакетов с самой низкой ценой угля за единицу веса; при равной стоимости за единицу веса выбираются пакеты с большим весом. По заданной информации о пакетах с углём и количестве транспортируемых пакетов определите суммарный вес угля в отправленных пакетах и стоимость самого тяжёлого отправленного пакета.

Входные данные

В первой строке через пробел записаны числа N - количество пакетов на складе (натуральное число, не превышающее 1000) и K – количество пакетов на отправку (натуральное число, не превосходящее 100). В каждой из последующих N строк через пробел записаны два числа – вес и стоимость каждого пакета.

Запишите в ответе два числа – сначала суммарный вес угля в отправленных пакетах, затем стоимость самого тяжёлого отправленного пакета.

Исходные данные:

10 4

47 470

50 600

60 480

45 540

30 300

15 180

70 560

30 360

91 910

40 320

При таких исходных данных самая низкая стоимость угля в пакетах весом 60, 70, 40; затем – у пакетов весом 91, 30, 47.

Поэтому наибольший возможный вес к отправке равен 70+60+40+91 = 261, а стоимость самого тяжёлого отправленного пакета равна 910.

## Задания на функции с именованными, фактическими и формальными параметрами

**Задание 2.**Объявите функцию с именем get\_rect\_value, которая принимает два аргумента (два числа) и еще один формальный параметр type с начальным значением 0. Если параметр type равен нулю, то функция должна возвращать периметр прямоугольника, а иначе - его площадь.

def get\_rect\_value(a:int, b:int, type=0)->int:

'''

принимает два аргумента (два числа) и еще один

формальный параметр type с начальным значением 0.

Если параметр type равен нулю, то функция должна

возвращать периметр прямоугольника,

а иначе - его площадь.

:param a:

:param b:

:param type:

:return:

'''

if type == 0:

return a \* 2 + b \* 2

else:

return a \* b

**Задание 3.** Объявите функцию с именем check\_password, которая принимает аргумент - строку (пароль) и имеет один формальный параметр chars с начальным значением в виде строки "$%!?@#". Функция должна проверять: есть ли в пароле хотя бы один символ из chars и что длина пароля не менее 8 символов. Если проверка проходит, то функция возвращает True, иначе - False.

P. S. Вызывать функцию не нужно, только объявить.

**Входные данные**

12345678!

**Выходные данные:**

True

def check\_password(st:str, chars='$%!?@#')->bool:

'''

принимает аргумент - строку (пароль) и имеет один формальный параметр chars

с начальным значением в виде строки "$%!?@#". Функция должна проверять: есть

ли в пароле хотя бы один символ из chars и что длина пароля не менее 8 символов.

Если проверка проходит, то функция возвращает True, иначе - False.

:param st: строка пароля

:param chars: '$%!?@#'

:return: True or False

'''

return all([any([i in st for i in chars]), len(st) > 7])

**Задание 5. Объявите функцию, которая принимает строку и заключает ее в указанный тег. Тег определяется формальным параметров tag с начальным значением в виде строки "h1". Например, мы передаем строку "Hello Python" и заключаем в тег "h1". На выходе должны получить строку (без кавычек):**

**"<h1>Hello Python</h1>"**

То есть, сначала открывается тег <h1>, а в конце строки - закрывается </h1>. И так для любых указанных тегов.

После объявления функции прочитайте (с помощью функции input) строку и дважды вызовите функцию (с выводом результата ее работы на экран):

- первый раз только со строкой  
- второй раз со строкой и именованным аргументом tag со значением 'div'.

**Входные данные**

Работаем с функциями

**Выходные данные:**

<h1>Работаем с функциями</h1>

<div>Работаем с функциями</div>

def tg(st: str, tag='h1')->str:

'''

принимает строку и заключает ее в указанный тег.

Тег определяется формальным параметров tag с начальным значением в виде строки "h1".

:param st: входящая строка

:param tag: таг

:return: <tag>строка</tag>

'''

return f'<{tag}>{st}</{tag}>'

st = input()

print(tg(st))

print(tg(st, tag='div'))

# Функция как тип

В Python функция фактически представляет отдельный тип. Так мы можем присвоить переменной какую-нибудь функцию и затем, используя переменную, вызывать данную функцию. Например,

def say\_hello(): print("Hello")  
  
def say\_goodbye(): print("Good Bye")  
  
  
  
message = say\_hello  
message() *# Hello*message = say\_goodbye  
message() *# Good Bye*

В данном случае переменной message присваивается одна из функций. Сначала ей передается функция say\_hello():

message = say\_hello

После этого переменная message будет указывать на данную функцию, то есть фактически представлять функцию say\_hello. А это значит, что мы можем вызывать переменную message как обычную функцию:

message() # Hello

Фактически это приведет к выполнению функции say\_hello, и на консоль будет выведена строка "Hello". Затем подобным образом мы можем передать переменной message другую функцию и вызвать ее.

Подобным образом можно через переменную вызывать функцию с параметрами и возвращать ее результат:

def sum(a, b): return a + b  
def multiply(a, b): return a \* b  
  
  
operation = sum  
result = operation(5, 6)  
print(result) *# 11*operation = multiply  
print(operation(5, 6))

# Функция как параметр функции

Поскольку функция в Python может представлять такое же значение как строка или число, соответственно мы можем передать ее в качестве параметра в другую функцию. Например, определим функцию, которая выводит на консоль результат некоторой операции:

def do(a, b, op):  
 return op(a, b)  
  
  
def sum(a, b):  
 return a + b  
  
  
def multiply(a, b):  
 return a \* b  
  
  
print(do(5,5,sum))  
print(do(5,5,multiply))

В данном случае функция do\_operation имеет три параметра, причем третий параметр, как предполагается, будет представлять функцию, которая принимает два параметра и возвращает некоторый результат. Иными словами третий параметр - operation представляет некоторую операцию, но на момент определения функции do\_operation мы точно не знаем, что это будет за операция. Мы только знаем, что она принимает два параметр и возвращает какой-то результат, который потом выводится на консоль.

При вызове функции do\_operation мы сможем передать в качестве третьего параметра другую функцию, например, функцию sum:

def do(a, b, op):

То есть в данном случае параметр operation фактически будет представлять функцию sum и будет возвращать сумму дву чисел.

Затем аналогичным образов в вызов функции do\_operation можно передать третьему параметру другую функцию - multiply, которая выполнит умножение чисел:

print(do(5,5,multiply))

Таким образом, более гибкие по функциональности функции, которые через параметры принимают другие функции.

Функция как результат функции

Также одна функция в Python может возвращать другую функцию. Например, определим функцию, которая в зависимости от значения параметра возвращает ту или иную операцию:

def sum(a, b): return a + b  
  
  
def subtract(a, b): return a - b  
  
  
def multiply(a, b): return a \* b  
  
  
def select\_operation(choice):  
 if choice == 1:  
 return sum  
 elif choice == 2:  
 return subtract  
 else:  
 return multiply  
  
  
operation = select\_operation(1) *# operation = sum*print(operation(10, 6)) *# 16*operation = select\_operation(2) *# operation = subtract*print(operation(10, 6)) *# 4*operation = select\_operation(3) *# operation = multiply*print(operation(10, 6)) *# 60*

В данном случае функция select\_operation в зависимости от значения параметра choice возвращает одну из трех функций - sum, subtract и multiply. Затем мы мы можем получить результат функции select\_operation в переменную operation:

operation = select\_operation(1)

Так, в данном случае в функцию select\_operation передается число 1, соответственно она будет возвращать функцию sum. Поэтому переменная operation фактически будет указывать на функцию sum, которая выполняет сложение двух чисел:

print(operation(10, 6))

# 16 - фактически равно sum(10, 6)

## Функция reduce

Последняя функция из нашей тройки — reduce() (говорят "свертка"), который используется для агрегации данных. Под агрегацией понимается операция, вычисляющая значение, зависящее от всего набора данных. С помощью функции reduce можно последовательно применить операции к элементам списка, чтобы получить единственное значение.

def add(x, y):

return x + y

def multiply(x, y):

return x \* y

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

result = reduce(add, numbers, 0)

print(result)

# 15

result = reduce(multiply, numbers, 1)

print(result)

Внутри reduce можно использовать и лямбда функции

Аналогичным образом можно получить и выполнить другие функции.

Например, передача лямбда-выражения в качестве параметра:

def do\_operation(a, b, operation):  
 return operation(a, b)  
  
  
  
x1 = do\_operation(5, 4, lambda a, b: a + b)  
x2 = do\_operation(5, 4, lambda a, b: a \* b)  
  
print(x1, x2)

Бессмысленный пример, но это основа для понимания декораторов и замыканий функций в дальнейшем!

В данном случае нам нет необходимости определять функции, чтобы передать их в качестве параметра, как в прошлом примере.

То же самое касается и возвращение лямбда-выражений из функций:

def select\_operation(choice):  
 if choice == 1:  
 return lambda a, b: a + b  
 elif choice == 2:  
 return lambda a, b: a - b  
 else:  
 return lambda a, b: a \* b  
  
  
operation = select\_operation(1) *# operation = sum*print(help(operation))

Результат:

<lambda> lambda a, b

Теперь в перемнной operation лежит lambda функция

print(operation(10, 6)) *# 16*operation = select\_operation(2) *# operation = subtract*print(operation(10, 6)) *# 4*operation = select\_operation(3) *# operation = multiply*print(operation(10, 6)) *# 60*

# Изменение области видимости переменных

Если же мы хотим изменить в локальной функции глобальную переменную, а не определить локальную, то необходимо использовать ключевое слово global:

name = "Илья"  
  
def say\_hi():  
 name = "Иван" *# скрываем значение глобальной переменной* print("Привет,", name)  
  
def say\_bye():  
 print("Пока,", name)  
  
  
say\_hi()  
say\_bye()

Привет, Иван

Пока, Илья

def say\_hi():  
 global name  
 name = "Иван" *# скрываем значение глобальной переменной* print("Привет,", name)

Результат:

Привет, Иван

Пока, Иван

# Организация программы и функция main

В программе может быть определено множество функций. И чтобы всех их упорядочить, одним из способов их организации является добавление специальной функции (обычно называется main), в которой потом уже вызываются другие функции:

def main():

    say\_hello()

    say\_goodbye()

def say\_hello():

    print("Hello")

def say\_goodbye():

    print("Good Bye")

# Вызов функции main

main()

Итак, мы узнали, что это такое и зачем они нужны, как объявляются функции и как они вызываются с разным числом аргументов. Для закрепления этого материала пройдите практические задания и переходите к следующему уроку, где мы продолжим изучение этой темы.

# Подсказки типов Hinting и doc strings

Первые упоминания о подсказках типов в языке программирования Python появились в базе Python Enhancement Proposals (PEP-483). Такие подсказки нужны для улучшения статического анализа кода и автодополнения редакторами, что помогает снизить риски появления багов в коде.

def get\_square(number: int) -> float:  
 return number \*\* 0.5  
  
res = get\_square(25)  
print(res)

или так

def get\_square(number: int) -> tuple[int, float]:  
 return number, number \*\* 0.5

или так

def get\_square(number: int) -> list:  
 return [number, number \*\* 0.5]

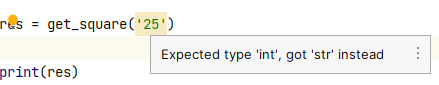
и даже так

def get\_square(number: int) -> str:

return [number, number \*\* 0.5]

Это всего лишь подсказка для программистов!

И теперь, если мы попробуем передать строку вместо числа, умный Pycharm напишет – я жду число, а не строку!



И еще одна подсказка!

help(print)

def get\_square(number: int) -> str:  
 *"""  
 описание функции  
 """*

print(help(get\_square))

для следующего урока.

# Области видимости

#Существует 4-области видимости LEGB

# Local -> # Encloded -> Global -> Builtin

# Локальная -> # Закрытая -> Глобальная -> Встроенная

Пример в консоли

sum

<built-in function sum>

print

<built-in function print>

А когда мы написали

print = 5

то функция print теперь живет в глобальной области видимости (в виде переменной) и пайтон её там легко находит. Переменную находит, а функцию нет.

print (5)

Результат

TypeError: 'int' object is not callable

Удалить print из глобальной области видимости можно с помощью оператора del.

А заглянуть в глобальную область видимости можно с помощью функции globals()

print = 5  
a = (1, 2, 3)  
b = {1: '1', 2: '2'}  
pprint.pprint(globals())

Результат:

**'a': (1, 2, 3),**

**'b': {1: '1', 2: '2'},**

**'pprint': <module 'pprint' from 'C:\\Users\\User\\AppData\\Local\\Programs\\Python\\Python311\\Lib\\pprint.py'>,**

**'print': 5,**

**Какую структуру данных возвращает** globals?

К областям видимости мы будем возвращаться постоянно!

А в локальную заглянем:

pprint.pprint(locals ())

Важно правило:

# Давать разные название переменных для разных областей видимости/

Рассмотрим очередной антипример:

def a(b):  
 c = b \*\* 2  
  
  
d = 25  
e = a(d)  
  
print(a, b, c, d, e)

Как вам? Что напечатает принт? Какие переменные находятся в локальной области видимости?

Как переделать код, чтобы понять, что вообще происходит

def get\_square(number):  
 return number \*\* 2  
  
var = 25  
res\_get\_square = get\_square(var)

Посмотрим на локальную область видимости:

def get\_square(number):  
 print(locals())

Еще один эксперимент:

global\_var\_1 = 100  
  
def get\_square(number):  
 print (global\_var\_1)  
 *# print(global\_var\_2)  
 # print(global\_var\_3)* global\_var\_2 = 2000  
 unknown\_var = 500  
 return number \*\* 2  
  
  
  
var = 25  
global\_var\_2 = 200  
res\_get\_square = get\_square(var)  
global\_var\_3 = 300  
*# print(unknown\_var)*print(global\_var\_2)

Каков результат?

А сейчас посмотрим

name = "Илья"  
  
def say\_hi():  
 name = "Иван" *# скрываем значение глобальной переменной* print("Привет,", name)  
  
def say\_bye():  
 print("Пока,", name)  
  
  
say\_hi()  
say\_bye()

Привет, Иван

Пока, Илья

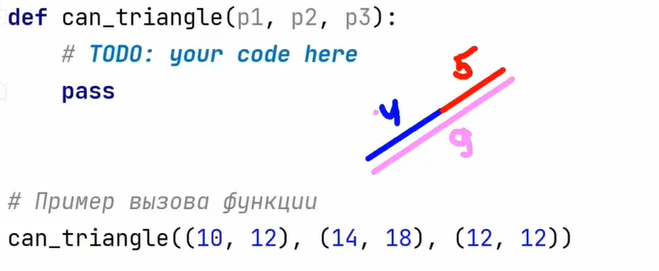
В отличие от глобальных переменных локальная переменная определяется внутри функции и доступна только из этой функции, то есть имеет локальную область видимости.

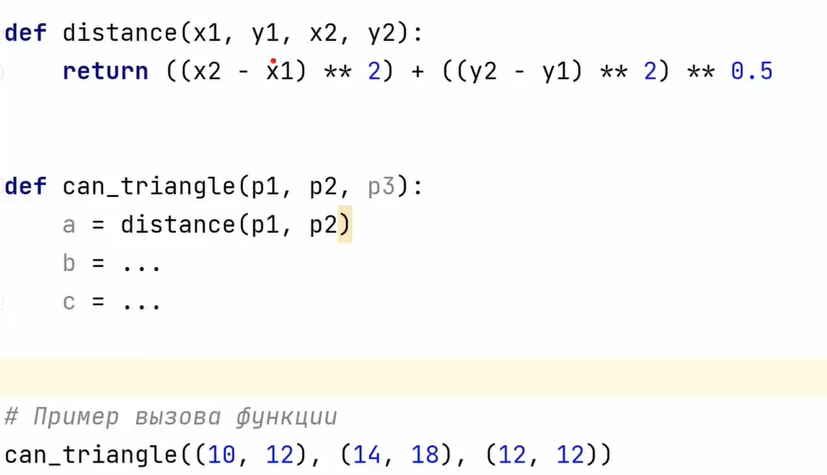
В данном случае в каждой из двух функций определяется локальная переменная name.

И хотя эти переменные называются одинаково, но тем не менее это две разных переменных, каждая из которых доступна только в рамках своей функции.

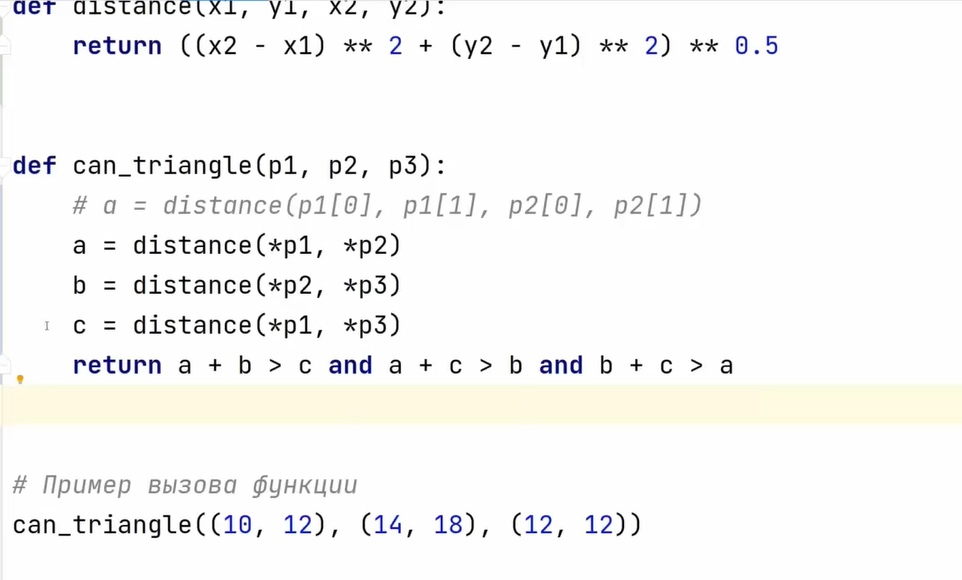
Также в функции say\_hi() определена переменная surname, которая также является локальной, поэтому в функции say\_bye() мы ее использовать не сможем.

# Прочее





Условие существование треугольника



# Домашнее задание

**Задача 1.** Лишний символ. Робот отправляет сообщения. Есть только две проблемы — иногда символы в каждом сообщении перемешиваются, и всегда добавляется ровно один лишний символ. Напишите программу, которая принимает две строки — исходное корректное сообщение и поврежденное доставленное, — и находит лишний символ.

Формат ввода:

Строка А — исходное сообщение.

Строка Б — доставленное сообщение.

#Пример ввода:

スーパーハカー

スーパーハッカー

Формат вывода:

Лишний символ в доставленном сообщении.

#Пример вывода:

ッ

a, b = input(), input()  
for character in a:  
 b = b.replace(character, '', 1)  
print(b)

**Задача 3.** Написать программу, чтобы посчитать площадь квартиры, состоящей из комнат rooms. Попробуйте применить map и reduce

rooms = [

    {"name": "кухня", "length": 6, "width": 4},

    {"name": "комната 1", "length": 5.5, "width": 4.5},

    {"name": "комната 2", "length": 5, "width": 4},

    {"name": "комната 3", "length": 7, "width": 6.3},

**from** functools **import** reduce

*#==============================================================================*

**def** rooms\_sqr(x):

**return** x["length"] \* x["width"]

*#==============================================================================*

**def** union(a, b):

**return** a + b

*#==============================================================================*

rooms = [

    {"name": "кухня", "length": 6, "width": 4},

    {"name": "комната 1", "length": 5.5, "width": 4.5},

    {"name": "комната 2", "length": 5, "width": 4},

    {"name": "комната 3", "length": 7, "width": 6.3},

]

**print**( reduce( union, list( map( rooms\_sqr, rooms ) ) ) )