КУРСАВАЯ РАБОТА

[Название организации] | [Адрес организации]

ПОВТ-19

КуДЛАЕв АНДРЕЙ СЕРГЕЕВИЧ

* [Введение](https://pythonru.com/osnovy/funkcii-v-python#%D0%92%D0%B2%D0%B5%D0%B4%D0%B5%D0%BD%D0%B8%D0%B5)
  + [Определение](https://pythonru.com/osnovy/funkcii-v-python#%D0%9E%D0%BF%D1%80%D0%B5%D0%B4%D0%B5%D0%BB%D0%B5%D0%BD%D0%B8%D0%B5)
  + [Вызовы](https://pythonru.com/osnovy/funkcii-v-python#%D0%92%D1%8B%D0%B7%D0%BE%D0%B2%D1%8B)
  + [Еще](https://pythonru.com/osnovy/funkcii-v-python#%D0%95%D1%89%D0%B5)
* [Инструкция return](https://pythonru.com/osnovy/funkcii-v-python#%D0%98%D0%BD%D1%81%D1%82%D1%80%D1%83%D0%BA%D1%86%D0%B8%D1%8F-return)
  + [Возврат простого значения](https://pythonru.com/osnovy/funkcii-v-python#%D0%92%D0%BE%D0%B7%D0%B2%D1%80%D0%B0%D1%82-%D0%BF%D1%80%D0%BE%D1%81%D1%82%D0%BE%D0%B3%D0%BE-%D0%B7%D0%BD%D0%B0%D1%87%D0%B5%D0%BD%D0%B8%D1%8F)
  + [Возврат нескольких значений](https://pythonru.com/osnovy/funkcii-v-python#%D0%92%D0%BE%D0%B7%D0%B2%D1%80%D0%B0%D1%82-%D0%BD%D0%B5%D1%81%D0%BA%D0%BE%D0%BB%D1%8C%D0%BA%D0%B8%D1%85-%D0%B7%D0%BD%D0%B0%D1%87%D0%B5%D0%BD%D0%B8%D0%B9)
* [Аргументы и параметры](https://pythonru.com/osnovy/funkcii-v-python#%D0%90%D1%80%D0%B3%D1%83%D0%BC%D0%B5%D0%BD%D1%82%D1%8B-%D0%B8-%D0%BF%D0%B0%D1%80%D0%B0%D0%BC%D0%B5%D1%82%D1%80%D1%8B)
  + [Указание произвольного количества аргументов](https://pythonru.com/osnovy/funkcii-v-python#%D0%A3%D0%BA%D0%B0%D0%B7%D0%B0%D0%BD%D0%B8%D0%B5-%D0%BF%D1%80%D0%BE%D0%B8%D0%B7%D0%B2%D0%BE%D0%BB%D1%8C%D0%BD%D0%BE%D0%B3%D0%BE-%D0%BA%D0%BE%D0%BB%D0%B8%D1%87%D0%B5%D1%81%D1%82%D0%B2%D0%B0-%D0%B0%D1%80%D0%B3%D1%83%D0%BC%D0%B5%D0%BD%D1%82%D0%BE%D0%B2)
* [Документирование функции](https://pythonru.com/osnovy/funkcii-v-python#%D0%94%D0%BE%D0%BA%D1%83%D0%BC%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5-%D1%84%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D0%B8)
* [Методы, функции и атрибуты, связанные с объектами функции](https://pythonru.com/osnovy/funkcii-v-python#%D0%9C%D0%B5%D1%82%D0%BE%D0%B4%D1%8B-%D1%84%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D0%B8-%D0%B8-%D0%B0%D1%82%D1%80%D0%B8%D0%B1%D1%83%D1%82%D1%8B-%D1%81%D0%B2%D1%8F%D0%B7%D0%B0%D0%BD%D0%BD%D1%8B%D0%B5-%D1%81-%D0%BE%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%B0%D0%BC%D0%B8-%D)
* [Рекурсивные функции](https://pythonru.com/osnovy/funkcii-v-python#%D0%A0%D0%B5%D0%BA%D1%83%D1%80%D1%81%D0%B8%D0%B2%D0%BD%D1%8B%D0%B5-%D1%84%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D0%B8)
* [Глобальная переменная](https://pythonru.com/osnovy/funkcii-v-python#%D0%93%D0%BB%D0%BE%D0%B1%D0%B0%D0%BB%D1%8C%D0%BD%D0%B0%D1%8F-%D0%BF%D0%B5%D1%80%D0%B5%D0%BC%D0%B5%D0%BD%D0%BD%D0%B0%D1%8F)
* [Присвоение функции переменной](https://pythonru.com/osnovy/funkcii-v-python#%D0%9F%D1%80%D0%B8%D1%81%D0%B2%D0%BE%D0%B5%D0%BD%D0%B8%D0%B5-%D1%84%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D0%B8-%D0%BF%D0%B5%D1%80%D0%B5%D0%BC%D0%B5%D0%BD%D0%BD%D0%BE%D0%B9)
* [Анонимная функция: лямбда](https://pythonru.com/osnovy/funkcii-v-python#%D0%90%D0%BD%D0%BE%D0%BD%D0%B8%D0%BC%D0%BD%D0%B0%D1%8F-%D1%84%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D1%8F-%D0%BB%D1%8F%D0%BC%D0%B1%D0%B4%D0%B0)
* [Изменяемые аргументы по умолчанию](https://pythonru.com/osnovy/funkcii-v-python#%D0%98%D0%B7%D0%BC%D0%B5%D0%BD%D1%8F%D0%B5%D0%BC%D1%8B%D0%B5-%D0%B0%D1%80%D0%B3%D1%83%D0%BC%D0%B5%D0%BD%D1%82%D1%8B-%D0%BF%D0%BE-%D1%83%D0%BC%D0%BE%D0%BB%D1%87%D0%B0%D0%BD%D0%B8%D1%8E)

**Определение**

Вот пример простой функции:

def compute\_surface(radius):

from math import pi

return pi \* radius \* radius

Для определения функции нужно всего лишь написать ключевое слово def перед ее именем, а после — поставить двоеточие. Следом идет блок инструкций.

Последняя строка в блоке инструкций может начинаться с return, если нужно вернуть какое-то значение. Если инструкции return нет, тогда по умолчанию функция будет возвращать объект None. Как в этом примере:

i = 0

def increment():

global i

i += 1

Функция инкрементирует глобальную переменную i и возвращает None (по умолчанию).

### Вызовы

Для вызова функции, которая возвращает переменную, нужно ввести:

surface = compute\_surface(1.)

Для вызова функции, которая ничего не возвращает:

increment()

### Еще

Функцию можно записать в одну строку, если блок инструкций представляет собой простое выражение:

def sum(a, b): return a + b

Функции могут быть вложенными:

def func1(a, b):

def inner\_func(x):

return x\*x\*x

return inner\_func(a) + inner\_func(b)

Функции — это объекты, поэтому их можно присваивать переменным.

### Возврат простого значения

Аргументы можно использовать для изменения ввода и таким образом получать вывод функции. Но куда удобнее использовать инструкцию return, примеры которой уже встречались ранее. Если ее не написать, функция вернет значение None.

### Возврат нескольких значений

Пока что функция возвращала только одно значение или не возвращала ничего (объект None). А как насчет нескольких значений? Этого можно добиться с помощью массива. Технически, это все еще один объект. Например:

def stats(data):

"""данные должны быть списком"""

\_sum = sum(data) # обратите внимание на подчеркивание, чтобы избежать переименования встроенной функции sum

mean = \_sum / float(len(data)) # обратите внимание на использование функции float, чтобы избежать деления на целое число

variance = sum([(x-mean)\*\*2/len(data) for x in data])

return mean,variance # возвращаем x,y — кортеж!

m, v = stats([1, 2, 1])

**Аргументы и параметры**

В функции можно использовать неограниченное количество параметров, но число аргументов должно точно соответствовать параметрам. Эти параметры представляют собой позиционные аргументы. Также Python предоставляет возможность определять значения по умолчанию, которые можно задавать с помощью аргументов-ключевых слов.

Параметр — это имя в списке параметров в первой строке определения функции. Он получает свое значение при вызове. Аргумент — это реальное значение или ссылка на него, переданное функции при вызове. В этой функции:

def sum(x, y):

return x + y

x и y — это параметры, а в этой:

sum(1, 2)

1 и 2 — аргументы.

При определении функции параметры со значениями по умолчанию нужно указывать до позиционных аргументов:

def compute\_surface(radius, pi=3.14159):

return pi \* radius \* radius

Если использовать необязательный параметр, тогда все, что указаны справа, должны быть параметрами по умолчанию.

Выходит, что в следующем примере допущена ошибка:

def compute\_surface(radius=1, pi):

return pi \* radius \* radius

Для вызовов это работает похожим образом. Сначала нужно указывать все позиционные аргументы, а только потом необязательные:

S = compute\_surface(10, pi=3.14)

На самом деле, следующий вызов корректен (можно конкретно указывать имя позиционного аргумента), но этот способ не пользуется популярностью:

S = compute\_surface(radius=10, pi=3.14)

А этот вызов **некорректен:**

S = compute\_surface(pi=3.14, 10)

При вызове функции с аргументами по умолчанию можно указать один или несколько, и порядок не будет иметь значения:

def compute\_surface2(radius=1, pi=3.14159):

return pi \* radius \* radius

S = compute\_surface2(radius=1, pi=3.14)

S = compute\_surface2(pi=3.14, radius=10.)

S = compute\_surface2(radius=10.)

Можно не указывать ключевые слова, но тогда порядок имеет значение. Он должен соответствовать порядку параметров в определении:

S = compute\_surface2(10., 3.14)

S = compute\_surface2(10.)

Если ключевые слова не используются, тогда нужно указывать все аргументы:

def f(a=1,b=2, c=3):

return a + b + c

Второй аргумент можно пропустить:

f(1,,3)

Чтобы обойти эту проблему, можно использовать словарь:

params = {'a':10, 'b':20}

S = f(\*\*params)

Значение по умолчанию оценивается и сохраняется только один раз при определении функции (не при вызове). Следовательно, если значение по умолчанию — это изменяемый объект, например, список или словарь, он будет меняться каждый раз при вызове функции. Чтобы избежать такого поведения, инициализацию нужно проводить внутри функции или использовать неизменяемый объект:

def inplace(x, mutable=[]):

mutable.append(x)

return mutable

res = inplace(1)

res = inplace(2)

print(inplace(3))

[1, 2, 3]

def inplace(x, lst=None):

if lst is None: lst=[]

lst.append()

return lst

Еще один пример изменяемого объекта, значение которого поменялось при вызове:

def change\_list(seq):

seq[0] = 100

original = [0, 1, 2]

change\_list(original)

original

[100, 1, 2]

Дабы не допустить изменения оригинальной последовательности, нужно передать копию изменяемого объекта:

original = [0, 1, 2]

change\_list(original[:])

original

[0, 1, 2]

### Указание произвольного количества аргументов

#### **Позиционные аргументы**

Иногда количество позиционных аргументов может быть переменным. Примерами таких функций могут быть max() и min(). Синтаксис для определения таких функций следующий:

def func(pos\_params, \*args):

block statememt

При вызове функции нужно вводить команду следующим образом:

func(pos\_params, arg1, arg2, ...)

Python обрабатывает позиционные аргументы следующим образом: подставляет обычные позиционные аргументы слева направо, а затем помещает остальные позиционные аргументы в [кортеж](https://pythonru.com/uroki/kortezhi-tuple-uroki-po-python-dlja-nachinajushhih) (\*args), который можно использовать в функции.

Вот так:

def add\_mean(x, \*data):

return x + sum(data)/float(len(data))

add\_mean(10,0,1,2,-1,0,-1,1,2)

10.5

Если лишние аргументы не указаны, значением по умолчанию будет пустой кортеж.

#### **Произвольное количество аргументов-ключевых слов**

Как и в случае с позиционными аргументами можно определять произвольное количество аргументов-ключевых слов следующим образом (в сочетании с произвольным числом необязательных аргументов из прошлого раздела):

def func(pos\_params, \*args, \*\*kwargs):

block statememt

При вызове функции нужно писать так:

func(pos\_params, kw1=arg1, kw2=arg2, ...)

Python обрабатывает аргументы-ключевые слова следующим образом: подставляет обычные позиционные аргументы слева направо, а затем помещает другие позиционные аргументы в кортеж (\*args), который можно использовать в функции (см. предыдущий раздел). В конце концов, он добавляет все лишние аргументы в словарь (\*\*kwargs), который сможет использовать функция.

Есть функция:

def print\_mean\_sequences(\*\*kwargs):

def mean(data):

return sum(data)/float(len(data))

for k, v in kwargs.items():

print k, mean(v)

print\_mean\_sequences(x=[1,2,3], y=[3,3,0])

y 2.0

x 2.0

Важно, что пользователь также может использовать словарь, но перед ним нужно ставить две звездочки (\*\*):

print\_mean\_sequences(\*\*{'x':[1,2,3], 'y':[3,3,0]})

y 2.0

x 2.0

Порядок вывода также не определен, потому что словарь не отсортирован.

## **Документирование функции**

Определим функцию:

def sum(s,y): return x + y

Если изучить ее, обнаружатся два скрытых метода (которые начинаются с двух знаков нижнего подчеркивания), среди которых есть \_\_doc\_\_. Он нужен для настройки документации функции. Документация в Python называется docstring и может быть объединена с функцией следующим образом:

def sum(x, y):

"""Первая срока - заголовок

Затем следует необязательная пустая строка и текст

документации.

"""

return x+y

Команда docstring должна быть первой инструкцией после объявления функции. Ее потом можно будет извлекать или дополнять:

print(sum.\_\_doc\_\_)

sum.\_\_doc\_\_ += "some additional text"

## **Методы, функции и атрибуты, связанные с объектами функции**

Если поискать доступные для функции атрибуты, то в списке окажутся следующие методы (в Python все является [объектом](https://pythonru.com/osnovy/klass-i-obekt-v-python) — даже функция):

sum.func\_closure sum.func\_defaults sum.func\_doc sum.func\_name

sum.func\_code sum.func\_dict sum.func\_globals

И несколько скрытых методов, функций и атрибутов. Например, можно получить имя функции или модуля, в котором она определена:

>>> sum.\_\_name\_\_

"sum"

>>> sum.\_\_module

"\_\_main\_\_"

Есть и другие. Вот те, которые не обсуждались:

sum.\_\_call\_\_ sum.\_\_delattr\_\_ sum.\_\_getattribute\_\_ sum.\_\_setattr\_\_

sum.\_\_class\_\_ sum.\_\_dict\_\_ sum.\_\_globals\_\_ sum.\_\_new\_\_ sum.\_\_sizeof\_\_

sum.\_\_closure\_\_ sum.\_\_hash\_\_ sum.\_\_reduce\_\_ sum.\_\_str\_\_

sum.\_\_code\_\_ sum.\_\_format\_\_ sum.\_\_init\_\_ sum.\_\_reduce\_ex\_\_ sum.\_\_subclasshook\_\_

sum.\_\_defaults\_\_ sum.\_\_get\_\_ sum.\_\_repr\_\_

## **Рекурсивные функции**

Рекурсия — это не особенность Python. Это общепринятая и часто используемая техника в Computer Science, когда функция вызывает сама себя. Самый известный пример — вычисление факториала n! = n \* n — 1 \* n -2 \* … 2 \*1. Зная, что 0! = 1, факториал можно записать следующим образом:

def factorial(n):

if n != 0:

return n \* factorial(n-1)

else:

return 1

Другой распространенный пример — определение последовательности Фибоначчи:

f(0) = 1

f(1) = 1

f(n) = f(n-1) + f(n-2)

[Рекурсивную функцию](https://pythonru.com/osnovy/rekursiya-python) можно записать так:

def fibbonacci(n):

if n >= 2:

else:

return 1

Важно, чтобы в ней было была конечная инструкция, иначе она никогда не закончится. Реализация вычисления факториала выше, например, не является надежной. Если указать отрицательное значение, функция будет вызывать себя бесконечно. Нужно написать так:

def factorial(n):

assert n > 0

if n != 0:

return n \* factorial(n-1)

else:

return 1

Важно!  
Рекурсия позволяет писать простые и элегантные функции, но это не гарантирует эффективность и высокую скорость исполнения.

Если рекурсия содержит баги (например, длится бесконечно), функции может не хватить памяти. Задать максимальное значение рекурсий можно с помощью модуля sys.

## **Глобальная переменная**

Вот уже знакомый пример с глобальной переменной:

i = 0

def increment():

global i

i += 1

Здесь функция увеличивает на 1 значение глобальной переменной i. Это способ изменять глобальную переменную, определенную вне функции. Без него функция не будет знать, что такое переменная i. Ключевое слово global можно вводить в любом месте, но переменную разрешается использовать только после ее объявления.

За редкими исключениями глобальные переменные лучше вообще не использовать.

## **Присвоение функции переменной**

С существующей функцией func синтаксис максимально простой:

variable = func

Переменным также можно присваивать встроенные функции. Таким образом позже есть возможность вызывать функцию другим именем. Такой подход называется непрямым вызовом функции.

Менять название переменной также разрешается:

def func(x): return x

a1 = func

a1(10)

10

a2 = a1

a2()

10

В этом примере a1, a2 и func имеют один и тот же id. Они ссылаются на один объект.

Практический пример — рефакторинг существующего кода. Например, есть функция sq, которая вычисляет квадрат значения:

def sq(x): return x\*x

Позже ее можно переименовать, используя более осмысленное имя. Первый вариант — просто сменить имя. Проблема в том, что если в другом месте кода используется sq, то этот участок не будет работать. Лучше просто добавить следующее выражение:

square = sq

Последний пример. Предположим, встроенная функция была переназначена:

dir = 3

Теперь к ней нельзя получить доступ, а это может стать проблемой. Чтобы вернуть ее обратно, нужно просто удалить переменную:

del dir

dir()

## **Анонимная функция: лямбда**

[Лямбда-функция](https://pythonru.com/uroki/lambda-funcija-uroki-po-python-dlja-nachinajushhih) — это короткая однострочная функция, которой даже не нужно имя давать. Такие выражения содержат лишь одну инструкцию, поэтому, например, if, for и while использовать нельзя. Их также можно присваивать переменным:

product = lambda x,y: x\*y

В отличие от функций, здесь не используется ключевое слово return. Результат работы и так возвращается.

С помощью type() можно проверить тип:

>>> type(product)

function

На практике эти функции редко используются. Это всего лишь элегантный способ записи, когда она содержит одну инструкцию.

power = lambda x=1, y=2: x\*\*y

square = power

square(5.)

25

power = lambda x,y,pow=2: x\*\*pow + y

[power(x,2, 3) for x in [0,1,2]]

[2, 3, 10]

## **Изменяемые аргументы по умолчанию**

>>> def foo(x=[]):

... x.append(1)

... print x

...

>>> foo()

[1]

>>> foo()

[1, 1]

>>> foo()

[1, 1, 1]

Вместо этого нужно использовать значение «не указано» и заменить на изменяемый объект по умолчанию:

>>> def foo(x=None):

... if x is None:

... x = []

... x.append(1)

... print x

>>> foo()

[1]

>>> foo()

[1]