* [ОБЗОР КУРСА](https://lyceum.yandex.ru/courses/123/groups/568)

[Урок Функции 5](https://lyceum.yandex.ru/courses/123/groups/568/lessons/644)

**Функции с переменным числом аргументов**

**План урока**

1

[Распаковка и запаковка значений](https://lyceum.yandex.ru/courses/123/groups/568/lessons/644/materials/1276#1)

2

[Аргументы по умолчанию](https://lyceum.yandex.ru/courses/123/groups/568/lessons/644/materials/1276#2)

3

[Именованные аргументы](https://lyceum.yandex.ru/courses/123/groups/568/lessons/644/materials/1276#3)

4

[Инструкция pass. Согласованность аргументов](https://lyceum.yandex.ru/courses/123/groups/568/lessons/644/materials/1276#4)

**Аннотация**

*В этом уроке мы научимся писать сложные функции, принимающие неопределенное число аргументов, и передавать в функцию именованные параметры. Попутно мы немного поговорим о том, как работает множественное присваивание.*

**1. Распаковка и запаковка значений**

В конце урока про возврат значений из функции, мы коснулись темы возврата нескольких значений и множественного присваивания получившихся значений нескольким переменным.

**def** get\_coordinates():

**return** 1, 2

x, y = get\_coordinates()

**print**(x) *# => 1*

**print**(y) *# => 2*

Хотя этот прием с присваиванием результата нескольким значениям часто используется именно в применении к функциям, на самом деле, никакого отношения к механике работы функций он не имеет. В приведенном примере функция просто возвращает кортеж, а всю дальнейшую работу делает механизм множественного присваивания, а точнее процедура «распаковывания». Вы с ней уже сталкивались, когда обсуждали кортежи. Сейчас мы посмотрим на возможности множественного присваивания внимательнее.

Итак, вы можете написать:

x, y = (1.5, 2.5)

В момент присваивания кортеж будет распакован, его первый элемент будет записан в x, второй — в y. Распаковать можно не только кортежи, но и списки: x, y = [1.5, 2.5] будет работать точно так же.

Если при множественном присваивании (когда слева больше одной переменной) число элементов слева и справа не совпадает, возникает ошибка времени исполнения. Вы это видели, когда разбирали запись трех координат в две переменные или двух — в три.

**Важно**

Есть способ собрать сразу **несколько значений в одну переменную**. Это делается при помощи звёздочки перед именем переменной:

x, y, \*rest = 1, 2, 3, 4, 5, 6

В этом случае в x будет записана единица, в y — 2, а в rest — список, состоящий из всех аргументов, которые не попали в обычные переменные. В данном случае rest будет равен [3, 4, 5, 6].

**Важно**

Учтите, что rest всегда будет списком, даже когда в него попадает лишь один элемент или даже ноль:

x, y, \*rest = 1, 2, 3

**print**(rest)

*# => [3]*

x, y, \*rest = 1, 2

**print**(rest)

*# => []*

x, y, z, \*rest = 1, 2

*# Ошибка выполнения*

**Важно**

Звездочка может быть только у одного аргумента, но не обязательно у последнего:

\*names, surname = 'Анна Мария Луиза Медичи'.split()

**print**(names) *# => ['Анна', 'Мария', 'Луиза']*

**print**(surname) *# => Медичи*

Также есть возможность **распаковывать вложенные списки**. Например

a, (b, c), d = [1, [2, 3], 4]

запишет в переменные a, b, c, d значения 1, 2, 3 и 4 соответственно.

**Важно**

Если вы хотите **распаковать единственное значение в кортеже**, то после имени переменной должна идти запятая:

a = (1,)

b, = (1,)

**print**(a) *# => (1,)*

**print**(b) *# => 1*

Помимо распаковывания есть и операция **запаковывания**. Она выполняется всегда, когда справа от знака равенства стоит больше одного значения. Например, можно написать: values = 1, 2, 3. Тогда значения в правой части автоматически будут запакованы в кортеж (1, 2, 3).

Запаковывание можно комбинировать с распаковыванием:

a, \*b = 1, 2, 3

**print**(a, b)

*# => 1 [2, 3]*

**Важно**

Общее правило такое: при любых нестандартных присваиваниях **сначала происходит запаковывание значений в правой части, а затем — распаковка их в переменные, стоящие в левой части**.

Вообще, лучший способ понять операции с запаковыванием и распаковыванием значений — поэкспериментировать с ними.

Техника запаковывания и распаковывания переменных со звездочкой используется не только в операциях присваивания. Похожий механизм применяется для написания функций, которые могут принимать переменное число аргументов. И синтаксис для этого используется похожий на тот, который используется в множественном присваивании. В списке аргументов функции, один из аргументов может быть помечен звездочкой — тогда в него попадут все значения на соответствующей позиции, которые еще не присвоены другим аргументам.

**Важно**

Но есть и отличие. При **множественном присваивании** переменная со звездочкой получает **список** значений. А когда **аргумент функции** указан со звездочкой, он получает **кортеж** значений.

Например, функция print принимает сколько угодно аргументов и даёт таким образом возможность выводить на экран неограниченное число значений.

Разработчики языка Python могли поступить иначе и сделать функцию, принимающую всегда ровно один аргумент-список, и выводить на экран элементы этого списка. С точки зрения функциональности результат был бы аналогичным, но такую функцию было бы не так удобно использовать.

Мы сделаем свою функцию для вычисления произведения всех аргументов.

**def** product(first, \*rest):

result = first

**for** value **in** rest:

result \*= value

**return** result

product(2,3,5,7)

*# => 210*

Эта функция принимает как минимум один аргумент — first. Это не позволяет вызвать функцию без аргументов, что было бы бессмысленно. Все аргументы, кроме первого, попадают в **кортеж rest**.

Не всем функциям необходимо произвольное число элементов. Бывает так, что функции требуется просто разное число аргументов. В этом случае можно поступить следующим образом: поставить звездочку, которая формально позволяет использовать любое число аргументов, а внутри функции вручную проверять, что число переданных элементов — правильное.

**Важно**

Звёздочку можно использовать не только для того, чтобы запаковать аргументы. Распаковать их тоже можно. Если при вызове функции вы поставите звёздочку перед переданным аргументом-списком, то список раскроется и как бы «потеряет границы». Элементы списка станут аргументами функции.

arr = ['cd', 'ef', 'gh']

*# Здесь мы передаем просто список как один аргумент*

**print**(arr) *# => ['cd', 'ef', 'gh']*

*# А здесь мы раскрыли список и*

*# функция print получила три отдельных аргумента*

**print**(\*arr) *# => cd ef gh*

*# Это аналогично вызову*

**print**('сd', 'ef', 'gh') *# => cd ef gh*

*# Раскрытие списка можно комбинировать с любыми аргументами*

**print**('ab', \*arr, 'yz') *# => ab cd ef gh yz*

*# При раскрытии может быть несколько аргументов со звездочкой*

**print**(\*arr, \*arr) *# => cd ef gh cd ef gh*

Такую технику применяют, когда вам надо передать в функцию заранее неизвестное число аргументов. Вы делаете отдельную переменную, хранящую список аргументов, а затем просто раскрываете её при помощи звёздочки. На следующем уроке нам придется написать несколько таких функций.

**2. Аргументы по умолчанию**

Бывает так, что какой-то параметр функции часто принимает одно и то же значение.

Например, хорошо известная вам функция int принимает два параметра: строка, которую нужно преобразовать в число, а также основание системы счисления. Это позволяет ей считывать числа в различных системах счисления, например, двоичное число 101 мы можем считать так:

int('101', 2) *# => 5*

Но чаще всего эта функция используется для считывания из строки чисел, записанных в десятичной системе счисления. Было бы неудобно каждый раз писать 10 вторым аргументом. На такой случай Python позволяет задавать некоторым аргументам значения по умолчанию. У функции int второй аргумент по умолчанию равен 10 и потому можно вызывать эту функцию с одним аргументом. Значение второго подставится автоматически.

**Важно**

Для того, чтобы определить аргумент по умолчанию, в списке аргументов функции достаточно после имени переменной написать знак равенства и нужное значение. Аргументы, имеющие значение по умолчанию, должны идти в конце, ведь иначе интерпретатор не смог бы понять, какой из аргументов указан, а какой пропущен (и значит, для него нужно использовать значение по умолчанию).

В качестве примера сделаем функцию, которая будет готовить бургеры с котлетами разного типа и по умолчанию добавлять туда помидоры, но не добавлять лук. Тогда функция приготовления будет выглядеть так:

**def** make\_burger(typeOfMeat, withOnion=False, withTomato=True):

**print**('Булочка')

**if** withOnion:

**print**('Луковые колечки')

**if** withTomato:

**print**('Ломтик помидора')

**print**('Котлета из', typeOfMeat)

**print**('Булочка')

Теперь команда make\_burger("свинина") будет делать бургер из свинины, в котором нет колечек и есть помидоры. Но если вам хочется поменять состав бургера, вы легко можете это сделать: make\_burger("свинина", True) сделает вам бургер и с луком, и с помидорами (они по-умолчанию включены), а make\_burger("свинина", False, False) сделает вам бургер, в котором кроме булочки и котлеты ничего нет.

Первыми стоит указывать более важные аргументы (в нашем примере мы считаем, что класть или не класть лук — более важное решение, чем класть или не класть помидор). Если вы укажете только одно дополнительное значение, то оно будет присвоено первому аргументу по умолчанию, а второй аргумент так и останется со значением по умолчанию. Если укажете два значения, то значения будут присвоены обеим переменным.

**3. Именованные аргументы**

Еще одна проблема функций заключается в том, что программист вынужден помнить (или каждый раз узнавать в документации) порядок аргументов. В некоторых случаях тяжело угадать логичный порядок аргументов. Чтобы не запоминать эти малозначительные детали, можно передавать аргументы в функцию с указанием имени аргумента, в таком случае порядок аргументов не важен. Вы уже сталкивались с именованными аргументами. Встроенная функция **print** часто используется с несколькими такими параметрами: **sep=" "** - для разделения аргументов при выводе (по умолчанию — пробелами) и **end="\n"** для того, чтобы в конце добавлялся символ перевода строки. Так, нам не надо беспокоится о том, какой параметр sep или end указывать первым.

**Позиционные и именованные аргументы**

Аргументы, которые передаются без указания имён, называются **позиционными**, потому что функция по положению аргумента понимает, какому параметру он соответствует. Аргументы, которые передаются с именами, называются **именованными**.

Чтобы вашу функцию можно было вызывать, используя именованные аргументы, буквально ничего не нужно делать. Все функции, которые вы писали на предыдущих уроках, уже можно вызывать, передавая им именованные аргументы.

Если у функции есть аргументы, то при вызове можно использовать имена параметров, которые вы использовали в определении функции (исключение составляют списки аргументов неопределенной длины, в которых используются аргументы «со звездочкой»). Это ещё один повод давать аргументам значащие, а не однобуквенные имена. Можно вспомнить или догадаться, что функция **matrix\_has\_value** имеет параметры matrix и value, но совершенно невозможно будет вспомнить про имена параметров, такие как a, b или m, v.

**Важно**

Именованные аргументы можно использовать вместе со значениями по умолчанию. Например, мы можем вызвать нашу функцию для создания бургеров, передав ей нужные именованные аргументы, а остальные оставив значениями по умолчанию (так как мы используем именованные аргументы, нам теперь не важно, в каком порядке мы их определяли):

make\_burger(typeOfMeat='говядина', withTomato=False)

Именованные и позиционные аргументы не всегда хорошо друг с другом «ладят». При вызове функции позиционные аргументы должны обязательно идти перед именованными аргументами. Достаточно сложно сформулировать точные правила поведения аргументов функции при использовании одновременно аргументов со звездочкой и именованных аргументов. Чем запоминать точные правила в таких случаях лучше пользоваться здравым смыслом.

**Общие рекомендации:**

— Если вам приходится долго думать о том, как записать список аргументов, чтобы он работал корректно, лучше использовать более простую версию. Ведь код, который тяжело писать, с большой вероятностью будет тяжело читать.

— Если ваш вызов функции не работает, попробуйте прочитать его «глазами интерпретатора». Однозначно ли он читается или вы можете придумать несколько вариантов разложить переданные в вызове функции параметры по аргументам? Если вы можете трактовать код несколькими способами, то с большой вероятностью, интерпретатор столкнется с теми же трудностями. В ситуации, когда код неоднозначен, интерпретатор Python не пытается угадать, что программист имел ввиду, а сообщает об ошибке. Часто это считается синтаксической ошибкой, и ошибка возникает еще до того как программа начинает выполняться.

**4. Инструкция pass. Согласованность аргументов**

В языке Python есть эталонно бесполезная инструкция **pass**. Инструкция **pass** — это инструкция-заглушка, которая не делает ничего. Дело в том, что синтаксис языка Python не позволяет в некоторых местах обойтись без команд.

Например, не может быть функции с пустым телом. Ветвь условного оператора или тело цикла тоже должны выполнять какие-либо действия, но иногда программист хочет отложить их написание и ставит такую заглушку.

**if** game\_over:

**pass** *# To Do: написать вывод итогового результата*

Давайте теперь напишем функцию, которая при вызове не делает ничего. Наша первая попытка будет такой:

**def** nop():

**pass**

nop()

Было бы удобно, если можно было бы завести функцию, которая принимает любые аргументы и, игнорируя их все, не делает ничего. Для захвата произвольного числа параметров, воспользуемся аргументом со звёздочкой:

**def** nop(\*rest):

**pass**

nop()

nop("Любое", "сказанное", "вами слово", "будет проигнорировано")

nop(100500, None, [1, 2, 3, 4, 5])

**Задание** (необязательное):

Мы хотим, чтобы функция **nop** принимала такие же аргументы, как функция **print**. Тогда можно будет отключать вывод на экран, просто заменив **print** на **nop**.

Попробуйте «сломать» нашу функцию **nop**:

**def** nop(\*rest):

**pass**

То есть придумайте такой набор аргументов, который работает, если использовать функцию **print**, но выдает ошибку, если использовать функцию **nop**.

Объясните, почему эта команда ломает функцию **nop** и попробуйте поправить функцию **nop** известными вам средствами так, чтобы она не ломалась.

Чтобы написать функцию, которая игнорирует любой список аргументов, необходимо разрешить ей принимать произвольное число позиционных аргументов и произвольное число именованных аргументов:

**def** nop(\*rest, \*\*kwargs):

**pass**

nop(1, [2, 3], debug=True, file="debug.log")

**Важно**

Аргумент с двумя звёздочками, **\*\*kwargs** — это специальный аргумент, который может перехватить все «лишние» именованные аргументы, переданные в функцию. Лишними аргументами будут все именованные аргументы в команде вызова функции, для которых нет соответствующего параметра в определении функции.

Этот аргумент, как и аргумент с одной звёздочкой, захватывающий «лишние» позиционные аргументы, можно использовать в комбинации с обычными аргументами. Например, сделаем и вызовем функцию, распечатывающую информацию о пользователе:

**def** profile(name, surname, city, \*children, \*\*additional\_info):

**print**("Имя:", name)

**print**("Фамилия:", surname)

**print**("Город проживания:", city)

**if** len(children) > 0:

**print**("Дети:", ", ".join(children))

**print**(additional\_info)

profile("Сергей", "Михалков", "Москва", "Никита Михалков",

"Андрей Кончаловский", occupation="writer", diedIn=2009)

*# Имя: Сергей*

*# Фамилия: Михалков*

*# Город проживания: Москва*

*# Дети: Никита Михалков, Андрей Кончаловский*

*# {'diedIn': 2009, 'occupation': 'writer'}*

Как вы уже знаете, параметр **children** будет списком лишних позиционных аргументов. А вот **additional\_info** будет словарём лишних именованных аргументов. В последней строке мы распечатали переданный словарь, он выглядит так:

{'diedIn': 2009, 'occupation': 'writer'}

Вы уже знаете, что звёздочка может не только запаковывать аргументы, но и распаковывать их, если передать в функцию список со звёздочкой перед ним:

**print**(['Массив', 'из', 'четырех', 'аргументов'])

*# => ['Массив', 'из', 'четырех', 'аргументов']*

**print**(\*['Просто', 'три', 'аргумента'])

*# => Просто три аргумента*

Две звёздочки также позволяют не только запаковывать именованные аргументы в словарь, но и распаковывать словарь в набор именованных аргументов.

Часто в функции используется запаковывание, а затем распаковывание. Например, сделаем собственную функцию **perforated\_print**, которая будет делать всё то же самое, что функция **print**, но при этом будет печатать горизонтальную линию над и под распечатанным текстом. Мы хотим использовать все опции функции **print**, но не хотим их самостоятельно обрабатывать. Поэтому мы перехватываем все опции (sep, end и т.п.), переданные в нашу функцию **perforated\_print**, а затем передаем их без изменений в функцию **print**. С позиционными аргументами поступаем так же:

**def** perforated\_print(\*args, \*\*kwargs):

**print**(\*args,\*\*kwargs)

**print**('-' \* 20)

perforated\_print('Теперь текст выводится с линией перфорации.')

perforated\_print('И', 'можно', 'использовать', 'опции', end=':**\n**')

perforated\_print('end', 'sep', 'прочие', sep=', ', end='!**\n**')

Вы можете использовать распаковывание только что запакованных аргументов для того, чтобы усложнять поведение функции, подобно тому, как мы добавили черту к функции **print**. В дополнительных материалах вы узнаете, как создавать декораторы и модифицировать поведение уже существующих функций.

[Справка](https://yandex.ru/support/lyceum-students)

Исключительное право на учебную программу и все сопутствующие ей учебные материалы, доступные в рамках проекта «Яндекс.Лицей», принадлежат АНО ДПО «ШАД». Воспроизведение, копирование, распространение и иное использование программы и материалов допустимо только с предварительного письменного согласия АНО ДПО «ШАД».

© 2018 – 2020  ООО «[Яндекс](https://yandex.ru/)»

Чаты