Шпаргалка: лямбда-функции

> 📌 Лямбда-функция — это одноразовая функция без имени. Её ещё называют анонимной функцией.

Она нужна, чтобы сделать что-то в коде один раз.

Пример. Нужно вернуть в код квадрат числа. С обычной функцией это займёт две строчки:

```
def square(a):
    return a*a       # возвести число a в квадрат
```

А с лямбда-функцией хватит и одной:

```
lambda a: a*a       # возвести число a в квадрат
```

Запись стала проще и короче. А ещё это экономит время — не нужно придумывать имя для функции, которую используешь один раз.

Как написать лямбда-функцию

Такие функции начинают с ключевого слова `lambda`. Их пишут в одну строчку: тело не отделяют отступами.

Чтобы создать лямбда-функцию, нужно:

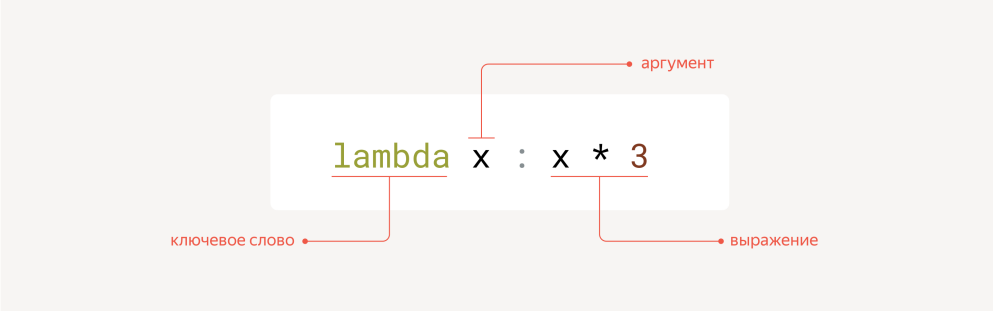
1. Поставить ключевое слово `lambda`;
2. Написать аргумент;
3. Поставить двоеточие;
4. Написать выражение.

Вот общая схема:

```
lambda аргумент(ы): выражение
```

Пример. Нужно умножить число на три.

```
lambda x: x * 3
```



Обрати внимание: в лямбда-функциях его не используют ключевое слово `return`.

💡 Лямбда-функция автоматически возвращает в код результат вычислений. Писать `return` не нужно.

Как вызвать лямбда-функцию

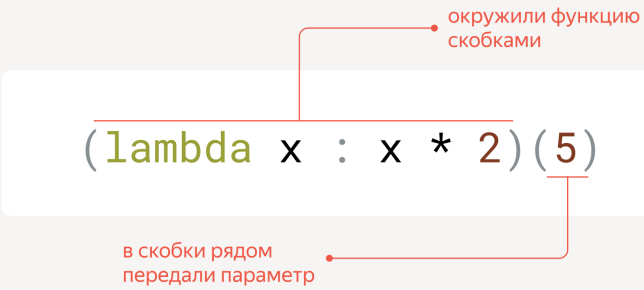
Обычную функцию сначала объявляют, потом вызывают. У лямбда-функции нет имени, поэтому и объявлять её не нужно. Такие функции сразу пишут с вызовом.

☝️ Лямбда-функцию не нужно объявлять — только вызвать.

Чтобы вызвать лямбда-функцию, нужно поставить круглые скобки и указать там весь её код. Рядом поставить ещё одни скобки и передать в них параметр.

Пример:

```
(lambda x : x * 2)(5)

# Результат: 10
```



Особенности функций

📌 Лямбда-функцию не нужно присваивать переменной

На самом деле, лямбда-функции можно дать имя. Для этого её присваивают переменной. Интерпретатор не выдаст ошибку, но делать так не стоит.

Руководство по написанию чистого кода — PEP8 — не советует давать имена анонимным функциям.

❌ **Вот так неправильно:**

```
# присвоили анонимную функцию переменной
multiply_by_two = lambda x: x * 2

# теперь её можно вызвать по имени
multiply_by_two()
```

Если хочешь использовать функцию с именем, создай обычную. Запись будет длиннее, но так правильнее.

✅ **Лучше написать так:**

```
# переписали код с обычной функцией
def multiply_by_two(x):
    return x * 2
```

📌 Аргументов — сколько угодно

Аргументы анонимных функций не отличаются от аргументов обычных. Их пишут без круглых скобок, в остальном — всё то же самое.

Если аргументов несколько, их перечисляют через запятую:

```
(lambda аргумент1, аргумент2, аргумент3: выражение)(параметр1, параметр2, параметр3)
```

Пример. Нужна функция, которая перемножит два числа. У неё два аргумента:

```
(lambda x, y: x * y)(3, 4)

# Результат: 12
```

Можно написать лямбда-функцию и без аргументов.

Пример. Лямбда-функция возвращает строку:

```
(lambda: 'Просто строка')()   # аргументов нет, поэтому скобки - пустые

# Результат: 'Просто строка'
```

На практике может пригодиться пустая функция. Её используют, если код ждёт пустое значение, но принимает только функцию. Выглядит она так:

```
(lambda: None)()
```

📌 Выражение — всегда одно

В лямбда-функции нельзя использовать больше одного выражения.

❗ Это строгое условие: лямбда-функция вычисляет одно выражение и возвращает одно значение.

Если написать два выражения или больше, Python выдаст ошибку.

Пример:

```
# в теле функции написали два выражения через запятую
lambda x: x + 1, x - 2

# получили ошибку:
# NameError: name 'x' is not defined
```

📌 Нельзя использовать утверждения

Лямбда-функции не поддерживают утверждения. Дело в том, что в Python не умеет обрабатывать утверждения, которые лежат внутри выражений.

Поэтому в лямбдах не работают ключевые слова:

- `return`;
- `pass`;
- `assert`;
- `raise`.

Они вызовут ошибку `SyntaxError`.

Пример:

```
lambda: pass    # так писать нельзя

# получим ошибку:
# SyntaxError: invalid syntax
```

📌 Можно использовать условия

Лямбда-функции пишут в одну строку. В них можно использовать условия `if-else`, но синтаксис будет немого отличаться от привычного:

```
lambda <аргумент(ы)> : <код, который выполнится, если условие верно> if <условие> else <код, который выполнится, если условие неверно>
```

Пример. Эта функция проверит, входит ли значение в диапазон от 5 до 20:

```
lambda x: True if (5 < x < 20) else False
```

Важный нюанс: лямбда-функция обязана вернуть значение. Она что-то получила, вычислила и передала обратно в код. Поэтому без `else` условие написать не получится — возникнет ошибка `SyntaxError`. Интерпретатор не поймёт, что делать, если условие не выполнено.

Функция `filter()`

Обычно лямбда-функции применяют в паре со встроенными функциями, которые работают с последовательностями: списками, строками и словарями.

Самые универсальные из них — `filter()` и `map()`.

> 📌 Функции `map()` и `filter()` — это **функции высшего порядка**. Это значит, они принимают другие функции в качестве аргумента.

Функция `filter()`

Она помогает отфильтровать последовательность: выбрать нужные элементы и отбросить лишние.

Синтаксис у неё такой:

```
filter(функция, последовательность)
```

Функция `filter()` берёт по очереди каждый элемент последовательности и передаёт его функции-аргументу. Та проверяет, соответствует он условию или нет:

- если да, функция-аргумент возвращает `True`. Тогда `filter()` возвращает этот элемент в код;
- если нет, функция возвращает `False`. Такой элемент программа отбрасывает.

Пример. Уберём из списка `items` элементы, в которых больше пяти символов.

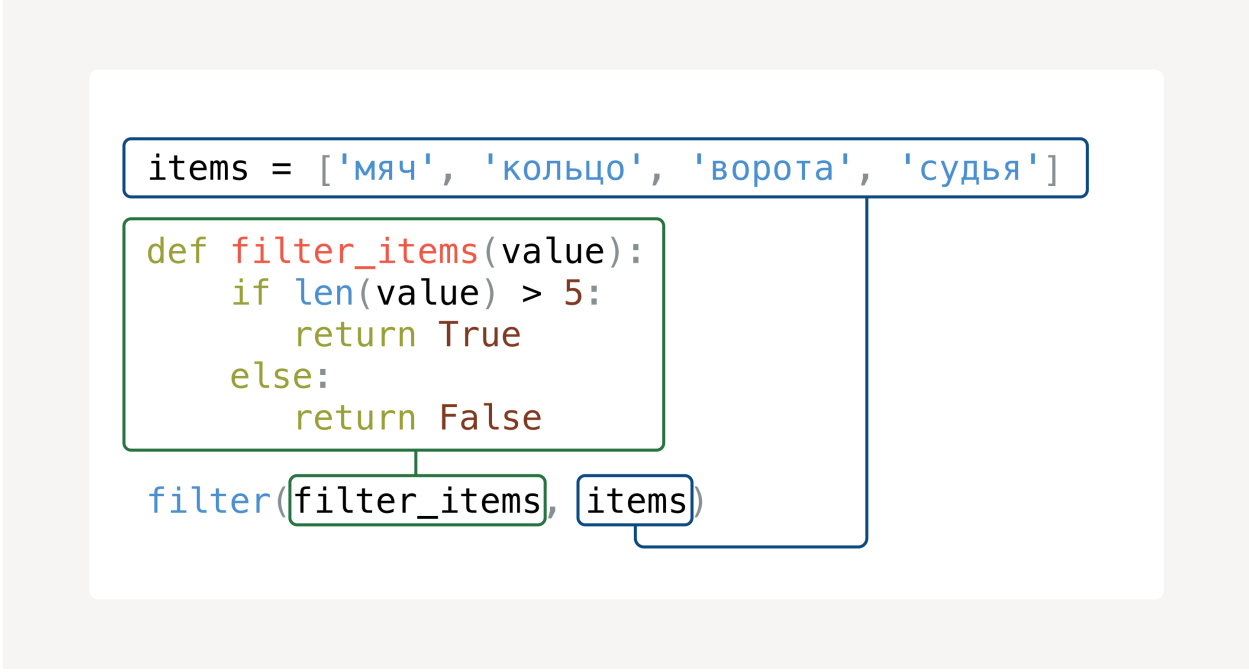
```
items = ['мяч', 'кольцо', 'ворота', 'судья']
```

Чтобы написать инструкцию для `filter()` с обычной функцией, придётся использовать условную конструкцию. Получится так:

```
def filter_items(value):
    if len(value) > 5:     # если длина value больше пяти
        return True        # верни True
    else:
        return False       # иначе верни False
```

Все аргументы для `filter()` готовы. Осталось передать их функции.

```
(filter(filter_items, items)
# ['кольцо', 'ворота']
```



А вот такой же код с лямбда-функцией. Он уместился в две строчки:

```
items = ['мяч', 'кольцо', 'ворота', 'судья']

filter(lambda x: len(x) > 5, items))
# ['кольцо', 'ворота']
```

Чтобы прочитать программу, тебе не нужно переключаться между разными блоками кода. Все инструкции идут друг за другом в одной строке.

Ещё пример. Есть список `a`. Нужно перебрать его и оставить только чётные числа.

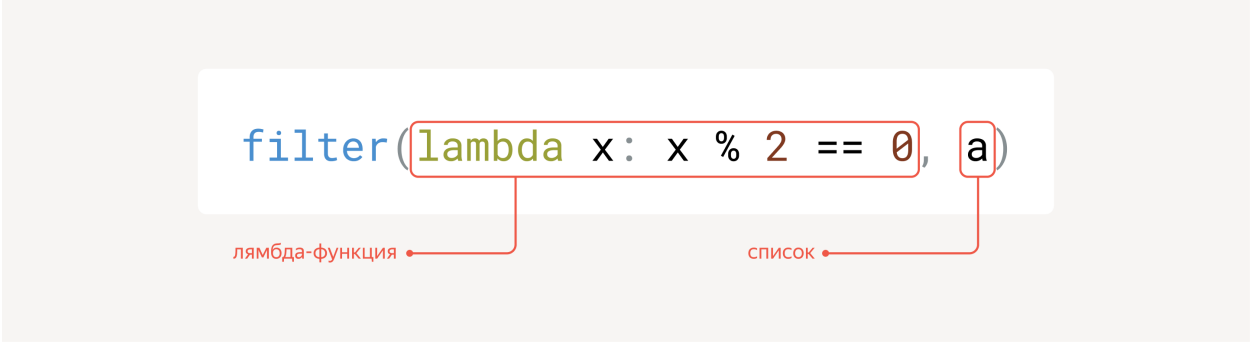
```
a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Лямбда-функция должна проверить, что число делится на два без остатка. В ней будет простое логическое выражение:

```
lambda x: x % 2 == 0   # остаток от деления x на два равен нулю
```

Лямбда-функцию сразу передают в `filter()`. Вторым аргументом будет имя списка — `a`. Получится вот так:

```
a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

filter(lambda x: x % 2 == 0, a)
```



Как напечатать результат

Если напечатать сейчас результат функции, он будет непонятным:

```
a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
b = filter(lambda x: x % 2 == 0, a)

print(b)
# Выведет: <filter object at 0x7fedb4daa880>
```

Python ничего не вывел. Он просто сослался на ячейку памяти, в которую записал результат.

Чтобы результат `filter()` стал читаемым, нужно превратить его в последовательность. С этим поможет ещё одна функция — `list()`. Она составит список элементов, которые прошли фильтр.

Пример:

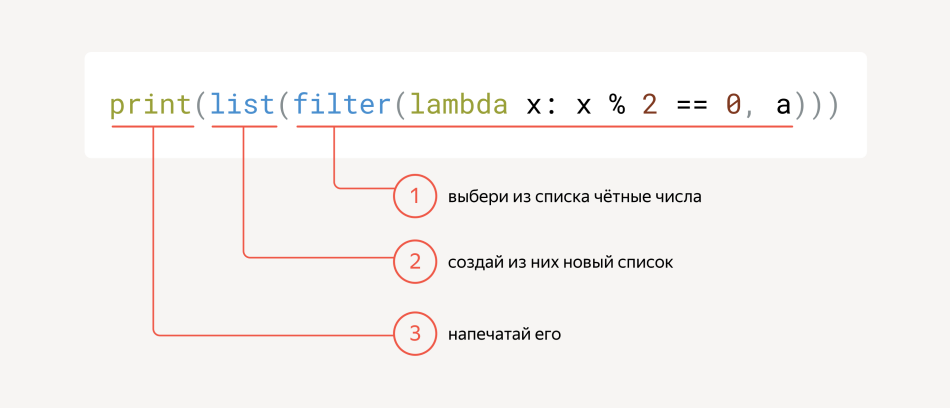
```
b = filter(lambda x: x % 2 == 0, a)
c = list(b)

print(c)
# Вывод: [2, 4, 6, 8, 10]
```

💡 Используй функции `list()` и `dict()` с `filter()` и `map()`. Они превратят результаты в последовательность — список или словарь.

Короткая запись

И даже этот код получится сократить. Можно объединить действия: сразу добавить `list()` и вызвать `print()`.

```
print(list(filter(lambda x: x % 2 == 0, a)))
```



Функция `map()`

Функция `map()` берёт функцию и применяет её к каждому элементу последовательности. Это полезно, когда нужно что-то сделать с каждым элементом. Например, сократить на два или умножить на пять.

Синтаксис у `map()` такой же, как у `filter()`. Сначала в неё передают функцию, затем — последовательность:

```
filter(функция, последовательность)
```

Пример. Есть список чисел `numbers`. Нужно возвести каждое число в квадрат:

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

# присвоили результат map() переменной squares
squares = map(lambda x: x ** 2, numbers)

# напечатали squares в виде списка
print(list(squares))
# Вывод: [1, 4, 9, 16, 25]
```

Функция `map()` передала каждый элемент `numbers` на вход лямбда-функции:

Зачем нужны лямбда-функции

Главная задача лямбда-функций — сократить и упростить код. Они заменяют обычные функции там, где можно что-то написать короче или вычислить быстрее.

Лямбда-функции делают код нагляднее и проще для чтения. Поэтому их используют с функциями `filter()` и `map()`.

С обычной функцией

```
items = ['мяч', 'кольцо', 'ворота', 'судья']

def filter_items(value):
    if len(value) > 5:
        return True
    else:
        return False

print(list(filter(filter_items, items)))
```

С лямбда-функцией

```
items = ['мяч', 'кольцо', 'ворота', 'судья']

print(list(filter(lambda x: len(x) > 5,
items)))
```

Лямбда-функции в автоматизации

Автоматизаторы часто пользуются лямбда-функциями при обработке данных.

Пример. Нужно преобразовать список строк в список чисел. Так с ними удобнее будет работать. Это можно сделать с помощью лямбда-функции, `map()` и `list()`.

```
string_list = ['1', '2', '3', '4', '5']

list(map(lambda x: int(x), string_list))

# Результат: [1, 2, 3, 4, 5]
```