

## Вложенные функции

**Вложенной** называется функция, которая объявлена в теле другой функции.

Пример:

```
def outer():
    print('Called outer function')
    def inner():
        print('Called inner function')
    inner()

outer()
## Будет напечатано:
## Called outer function
## Called inner function
```

Любая функция скрывает, защищает (**инкапсулирует**) доступ к переменным и функциям, объявленным внутри неё. Причина в том, что переменные и функции существуют только в момент вызова самой функции. Как только функция завершается, все переменные и функции, которые были объявлены внутри неё, уничтожаются — просто удаляются из памяти. Для этого в *Python* применяется механизм сборки мусора (*Garbage Collection*).

```
def print_root(value, n=2):
    def root(value, n=2):
        result = value ** (1/n)
        return result
    res = root(value, n)
    print(f'Root of power {n} from {value} equals {res}')

print_root(81, 4)
## Будет напечатано:
## Root of power 4 from 81 equals 3.0

print(root(81, 4))
## Возникнет ошибка:
## NameError: name 'root' is not defined
```

```
print(res)
## Возникнет ошибка
## NameError: name 'res' is not defined
```

## Разрешение переменных. Области видимости

**Разрешение** — это процесс поиска интерпретатором объекта, который скрывается за названием переменной.

В *Python* существует **четыре типа переменных** в зависимости от их видимости. Порядок их разрешения будет идти от пункта 1 до пункта 4.

1. **Локальные переменные (local)** — это имена объектов, которые были объявлены в функции и используются непосредственно в ней.

В разряд локальных переменных также входят аргументы функции.

**Важно!** Эти объекты существуют только во время выполнения функции, в которой они были объявлены. После завершения работы функции они удаляются из оперативной памяти.

2. **Нелокальные переменные (nonlocal)** — это имена объектов, которые были объявлены во внешней функции относительно рассматриваемой функции.
3. **Глобальные переменные (global)** — это имена объектов, которые были объявлены в основном блоке программы (вне функций).
4. **Встроенные переменные (built-in)** — это имена объектов, которые встроены в функционал *Python* изначально. К ним относятся, например, функции `print`, `len`, структуры данных `list`, `dict`, `tuple` и другие.

### Встроенные переменные (Built-in)

Имена объектов, которые встроены в функционал *Python* изначально. К ним относятся, например, функции *print*, *len*, структуры данных *list*, *dict*, *tuple* и другие.

### Глобальные переменные (Global)

Имена объектов, которые были объявлены непосредственно в основном блоке программы (вне функций).

### Нелокальные переменные (Nonlocal)

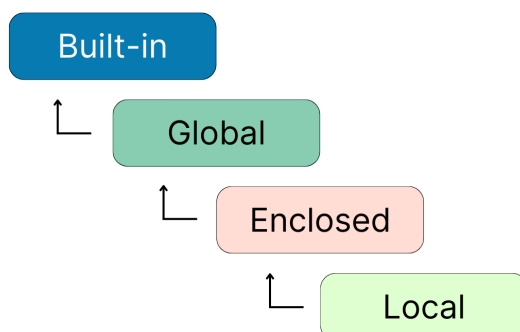
Имена объектов, которые были объявлены во внешней функции относительно рассматриваемой функции.

### Локальные переменные (Local)

Имена объектов, которые были объявлены в функции и используются непосредственно в ней. В разряд локальных переменных также входят аргументы функции.

Когда интерпретатор встречает в коде неизвестное имя (имя переменной или функции), он начинает искать имя в локальной области видимости, затем — в нелокальной, затем — в глобальной и наконец — во встроенной.

Часто нелокальную область видимости называют объемлющей (*enclosed*). Поэтому правило, по которому происходит поиск имени объекта среди областей видимости (разрешение), часто именуют правилом *LEGB* (*Local* → *Enclosed* → *Global* → *Built-in*).



Функция **не может** изменить значение переменной, которая находится вне своей локальной области видимости. Вместо изменения значения глобальной переменной создаётся новая локальная переменная с тем же именем.

```
count = 10
def function():
    count = 100
function()
print(count)

## Будет выведено:
## 10
```

Однако функция может скорректировать объект изменяемого типа, находящийся за пределами её локальной области видимости, если изменит его содержимое.

```
words_list = ['foo', 'bar', 'baz']
def function():
    words_list[1] = 'quux'
function()
print(words_list)

## Будет выведено:
## ['foo', 'quux', 'baz']
```

## Изменение значений глобальных переменных. Объявление `global`

Если в коде функции происходит переопределение глобальной переменной, то необходимо просто указать, что та или иная переменная является глобальной. Для этого используются ключевое слово `global`:

```
global_count = 0

def add_item():
    global global_count
    global_count = global_count + 1

add_item()
print(global_count)

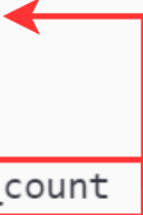
## Будет выведено:
```

```
## 1
```

Выражение `global` указывает на то, что, пока выполняется функция `add_item()`, ссылки на имя `global_count` будут вести к переменной `global_count`, объявленной в глобальной области видимости.

```
global_count = 0

def add_item():
    global global_count
    global_count = global_count + 1
```



Указываем, что `global_count` объявлена в глобальной области видимости

## Изменение значений нелокальных переменных. Объявление `nonlocal`

Если в коде функции происходит **переопределение** нелокальной переменной, то необходимо просто указать, что та или иная переменная является нелокальной. Для этого используются ключевое слово `nonlocal`:

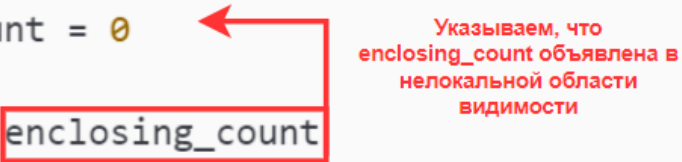
```
def outer():
    enclosing_count = 0
    def inner():
        nonlocal enclosing_count
        enclosing_count = enclosing_count + 1
    print(enclosing_count)
    inner()

outer()

## Будет напечатано:
## 1
```

После выражения `nonlocal enclosing_count`, когда `inner()` ссылается на `enclosing_count`, мы будем обращаться к `enclosing_count` в ближайшей объемлющей области, чье определение дано внутри `outer()`.

```
def outer():  
    enclosing_count = 0  
    def inner():  
        nonlocal enclosing_count  
        enclosing_count = enclosing_count + 1  
        print(enclosing_count)  
    inner()
```



Указываем, что  
enclosing\_count объявлена в  
нелокальной области  
видимости

## Изменение значений встроенных переменных

Python позволяет создавать переменные с именами идентичными именам встроенных в Python объектов.

Однако, делать этого крайне не рекомендуется, так как это может привести к ошибкам при дальнейших попытках программы обратиться к встроенному имени.

Пример (весь код является частью одной программы):

```
my_list = [1,4,5,7]  
len = len(my_list)  
print(len)  
## Будет выведено:  
## 4  
  
new_list = ['Ivan', 'Sergej', 'Maria']  
length = len(new_list)  
print(length)  
  
## Возникнет ошибка:  
## TypeError: 'int' object is not callable
```

## Определение рекурсии

**Рекурсия** в программировании — функция, которая вызывает саму себя в своем теле.

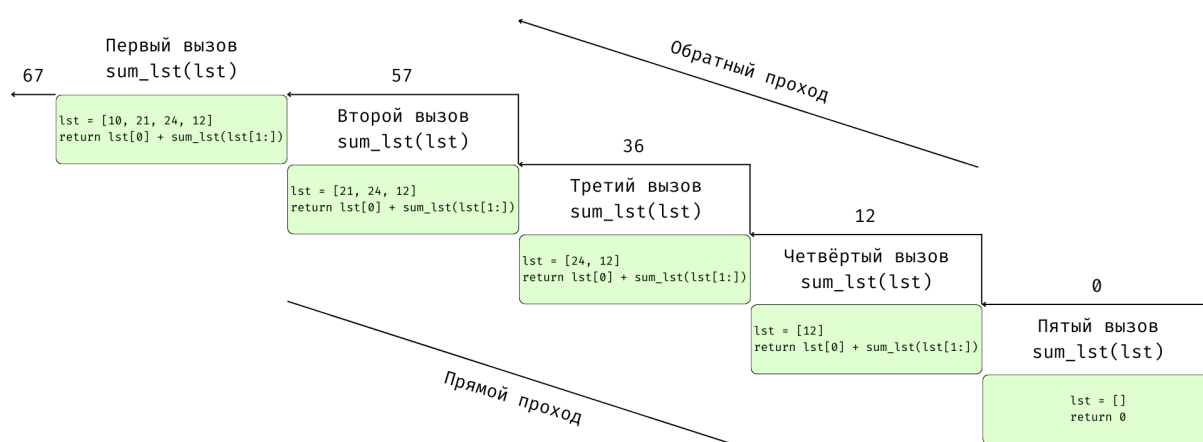
У рекурсивной функции должно быть прописано условие, при выполнении которого функция перестает вызывать сама себя. Такое условие будем называть **условием выхода из рекурсии** (его ещё называют базовым случаем).

### Пример №1 (сумма элементов в списке):

```
def sum_lst(lst):
    print(lst)
    if len(lst) == 0:
        return 0
    return lst[0] + sum_lst(lst[1:])
```

```
my_lst = [10, 21, 24, 12]
print(sum_lst(my_lst))
```

```
## Будет выведено:
## [10, 21, 24, 12]
## [21, 24, 12]
## [24, 12]
## [12]
## []
## 67
```

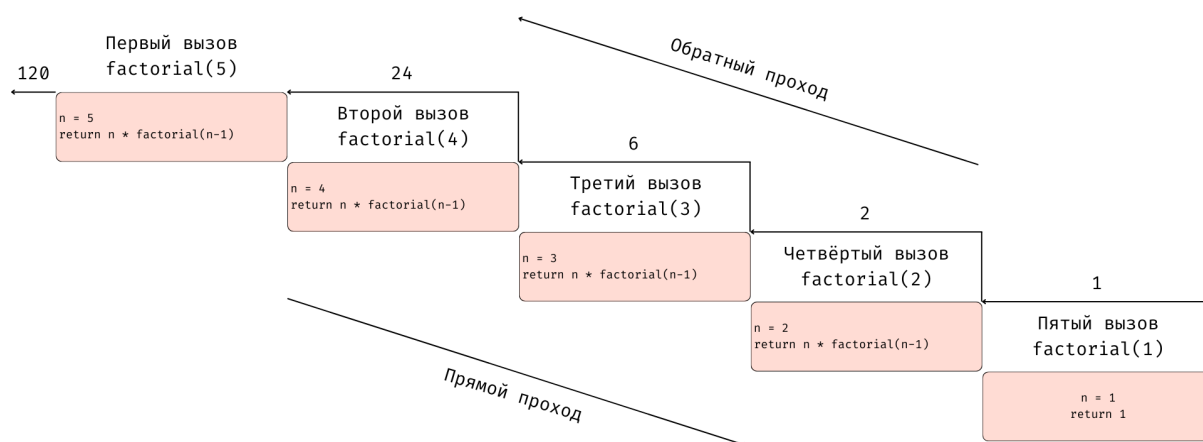


## Пример №2 (факториал):

```
def factorial(n):
    if n==0: return 1
    if n==1: return 1
    return n * factorial(n-1)

print(factorial(0))
print(factorial(3))
print(factorial(5))

## Будет напечатано:
## 1
## 6
## 120
```



## Стек вызова функций

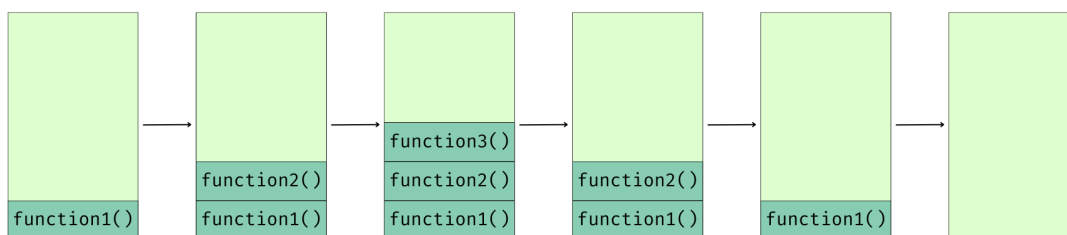
**Стек вызова функций** — структура данных, хранящая информацию об вызванных в программе функциях в виде адресов возврата функций.

В стеке действует правило *LIFO* (англ. *Last In, First Out*), «последний вошёл, первый вышел».



Ниже представлена схема работы стека вызова трёх функций, которые последовательно вызывают друг друга.

### Стек вызова функций



### Глубина рекурсии

Число единовременно ожидающих выполнения вызовов рекурсивной функции в стеке называется **глубиной рекурсии**.

Проще говоря, глубина рекурсии — это длина стека вызова.

Когда глубина рекурсии оказывается слишком большой (превышает максимальный размер стека функций), возникает ошибка `RecursionError`.

К сожалению, интерпретатор не может понять, есть ли условие выхода в коде функции, поэтому, когда в его стеке собирается слишком много вызовов той же самой функции, он возвращает ошибку. Это защита от бесконечного выполнения программы.

Этот порог индивидуален и может зависеть от сложности самой функции, версии *Python* и других настроек.

```
print(len(str(factorial(969))))
```

```
## Будет выведено:
```

```
## 2475
```

```
print(len(str(factorial(970))))
```

```
## Возникнет ошибка:
```

```
## RecursionError: maximum recursion depth exceeded in comparison
```

Глубину рекурсии можно увеличить самостоятельно, если для решения задачи крайне необходимо использовать рекурсию большей глубины.

Для этого нам необходимо импортировать модуль `sys`, в котором содержатся функции для управления вашей системой. Из него нам понадобится функция `setrecursionlimit()`, в аргументы которой нужно передать желаемую максимальную глубину рекурсии.

```
import sys
sys.setrecursionlimit(1000000000)
print(len(str(factorial(970))))

# Будет напечатано:
# 2478
```

## Цикл vs рекурсия

Важно учитывать, что рекурсии работают медленнее, чем аналогичные по сути циклы. К тому же существует ряд случаев, когда преобразование рекурсии в цикл затруднительно и неэффективно, например, реализация алгоритма дерева решений, быстрое преобразование Фурье или алгоритм *quicksort*. Однако, когда это оказывается возможным, цикл работает быстрее.

## Функция как объект

Функция, как и всё в *Python* (кроме ключевых слов, таких как `if`, `def`, `while` и т. д.), — это объект.

→ Во-первых, точно так же, как и любой другой объект (числа, строки, списки и т. д.), мы можем положить функцию в переменную.

```
p = print
p('Hello world!')
```

```
## Будет выведено:  
## Hello world!
```

- Во-вторых, функцию можно передать в качестве аргумента для другой функции.

```
def apply_func(func, x):  
    return func(x)  
print(apply_func(max, [1, 10, 35, 20, -1]))  
  
## Будет выведено:  
## 35
```

- В-третьих, у функции точно так же, как и любого другого объекта, есть свой тип данных — тип `function`.

```
print(type(apply_func))  
  
## Будет выведено:  
## <class 'function'>
```

**Важно!** Когда мы обращаемся с функцией как с объектом, мы не пишем скобки после имени функции. Они нужны только тогда, когда мы вызываем функцию и передаем в неё аргументы.

## Функция `map`

Функция `map()` позволяет преобразовать каждый элемент итерируемого объекта по заданной функции.

Аргументы функции `map()` следующие:

- первый — функция, которую необходимо применить к каждому элементу;
- второй — итерируемый объект (например, список).

Можно записать это в виде шаблона кода:

```
map(<имя_функции>, <итерируемый_объект>)
```

#### Пример №1:

```
number_list = [11, 12, 13, 14, 15, 16]
square_number_list = list(map(lambda x: x**3, number_list))
print(square_number_list)

## Будет выведено:
## [1331, 1728, 2197, 2744, 3375, 4096]
```

#### Пример №2:

```
def calculate_tax(salary):
    if salary < 1000:
        return salary * 0.05
    elif salary < 2000:
        return salary * 0.1
    else:
        return salary * 0.15

salaries = [1500, 2200, 3500, 1200]
taxes = list(map(calculate_tax, salaries))
print(taxes)

## Будет выведено:
## [150.0, 330.0, 525.0, 120.0]
```

#### Пример №3:

```
data = [('Amanda', 1.61, 51), ('Patricia', 1.65, 61), ('Marcos', 1.91, 101)]
map_func = lambda x: (*x, round(x[2] / (x[1]**2)))
updated_data = list(map(map_func, data))
```

```
print(updated_data)

## Будет выведено:
## [('Amanda', 1.61, 51, 19.7), ('Patricia', 1.65, 61, 22.4), ('Marcos',
1.91, 101, 27.7)]
```

## Функция filter

Функция `filter()` позволяет отфильтровать переданный ей итерируемый объект и оставить в нём только те элементы, которые удовлетворяют условию, заданному в виде функции.

Аргументы функции `filter()` следующие:

- первый — функция, которая должна возвращать `True`, если условие выполнено, иначе — `False`;
- второй — итерируемый объект, с которым производится действие.

Можно записать это в виде шаблона кода:

```
filter(<имя_функции>, <итерируемый_объект>)
```

### Пример №1:

```
words_list = ["We're", 'in', 'a', 'small', 'village', 'near', 'Chicago',
'My', "cousin's", 'getting', 'married.']
even_list = filter(lambda x: len(x) % 2 == 0, words_list)
print(list(even_list))

## Будет выведено:
## ['in', 'near', 'My', "cousin's", 'married.']
```

### Пример №2:

```
data = [
    ("FPW-2.0_D", "Бонус: Тренажер по HTML", 10, 100, 10),
```

```
("FPW-2.0", "Бонус: Тренажер по JavaScript", 9.2, 70, 18),
("FPW-2.0_D", "Бонус: Тренажер по React", 8.5, 66.67, 68),
("FPW-2.0", "Бонусный: IT в современном мире", 8.64, 53.74, 856),
("FPW-2.0", "Бонусный: Введение", 8.73, 56.24, 745),
("FPW-2.0", "Бонус: D1. Знакомство с Django (NEW)", 9.76, 95.24, 21),
("FPW-2.0_D", "Бонус: D2. Модели (NEW)", 9.44, 77.78, 18)
]

def filter_module(module):
    code, name, avg_votes, nessa, count = module
    cond_1 = code == "FPW-2.0"
    cond_2 = nessa >= 70
    cond_3 = count > 50
    return cond_1 and cond_2 and cond_3

filtered_data = list(filter(filter_module, data))
print(filtered_data)

## Будет выведено:
## [('FPW-2.0', 'Бонус: Тренажер по JavaScript', 9.2, 70, 180),
('FPW-2.0', 'Бонусный: IT в современном мире', 8.64, 83.74, 856)]
```

## Конвейеры из map и filter

В некоторых случаях необходимо выполнить сразу несколько действий с итерируемыми объектами. Например, вы хотите сначала преобразовать данные, а затем отфильтровать их. Такие преобразования называются **конвейерными**.

Для построения конвейерных преобразований необязательно каждый раз после применения `map()` или `filter()` получать список элементов. Объекты `map` и `filter` можно подставлять в эти же функции `map()` и `filter()`.

### Пример №1:

```
words_list = ["We're", 'in', 'a', 'small', 'village', 'near', 'Chicago',
'My', "cousin's", 'getting', 'married.']
filtered_words = filter(lambda x: len(x) >= 5, words_list)
```

```
count_a = map(lambda x: (x, x.lower().count('a')), filtered_words)

print(list(count_a))

## Будет выведено:
## [("We're", 0), ('small', 1), ('village', 1), ('Chicago', 1),
('cousin's', 0), ('getting', 0), ('married.', 1)]
```

### Пример №2

```
data = [
    ('Amanda', 1.61, 51),
    ('Patricia', 1.65, 61),
    ('Marcos', 1.91, 101),
    ('Andrey', 1.79, 61),
    ('Nikos', 1.57, 78),
    ('Felicia', 1.63, 56),
    ('Lubov', 1.53, 34)
]

map_func = lambda x: (*x, x[2]/(x[1]**2))
updated_data = map(map_func, data)

filter_func = lambda x: 18.5 <= x[3] <= 25
filtered_data = filter(filter_func, updated_data)

print(list(filtered_data))

## Будет выведено:
## [('Amanda', 1.61, 51, 19.7), ('Patricia', 1.65, 61, 22.4), ('Andrey',
1.79, 61, 19.0), ('Felicia', 1.63, 56, 21.1)]
```