

# Функции в Python

Основы и немного больше

# Структура презентации

1. Что такое функция
2. Пример использования
3. Объявление функции в Python
4. Аргументы функции
5. Передача аргументов при вызове
6. Специальные случаи
7. Области видимости функции
8. Управление областями поиска переменной
9. Рекурсия и замыкание
10. Функция, как объект
11. Lambda-выражение
12. Армия функций
13. Общие рекомендации по написанию функций
14. Аннотации
15. Несколько встроенных средств функционального программирования

## Функция?

Функция — фрагмент программного кода (подпрограмма), к которому можно обратиться из другого места программы.

## Чем она полезна?

- Сократить дублирование кода.
- Упростить понимание программы.
- Увеличить переиспользование кода.
- Упростить поддержку.

# Пример

```
a = input('Input number of lines: ')
b = []
for i in range(int(a)):
    b.extend(input(f'Input line {i}: ').split())
b = [int(element) for element in b]

while True:
    result = 0
    for element in b:
        result += element

    if result % 2 != 0:
        a = input('Input number of lines: ')
        b = []
        for i in range(int(a)):
            b.extend(input(f'Input line {i}: ').split())
        b = [int(element) for element in b]
    else:
        print(f'Even sum equals {result}. Good luck!')
        break
```

=>

```
def get_user_input():
    number_of_lines = int(input('Input number of lines: '))
    return [
        int(number)
        for line_number in range(number_of_lines)
        for number in input(f'Input line {line_number}: ').split()
    ]

while True:
    user_input = get_user_input()
    result = sum(user_input)
    if result % 2 != 0:
        continue
    print(f'Even sum equals {result}. Good luck!')
    break
```

## Объявление функции/Инструкция def

```
def useful_function(first_argument, second_argument):  
    sum_result = first_argument + second_argument  
    mul_result = first_argument * second_argument  
  
    return sum_result, mul_result
```

<- заголовок

<- тело

Функция не существует, пока  
исполнение не дойдет до def

```
if x is None:  
    def foo(argument):  
        pass  
else:  
    def bar(argument):  
        pass
```

Функция может что-то возвращать. return  
осуществляет выход из функции

```
def bar(x):  
    return x  
  
a = bar(2)  
assert a == 2
```

```
def foo(argument):  
    return argument, argument + 1  
  
a = foo(1)  
assert a == (1, 2)
```

```
def fun():  
    pass  
  
a = fun()  
assert a is None
```

## Подробнее об аргументах

- Позиционные `def foo(a, b, c): # foo(1,2,3), foo(1, c=3, b=2)`
- По умолчанию `def foo(a, b=1): # foo(1), foo(1,2)`
- Переменное число позиционных аргументов `def foo(*args): # foo(1,2,3,4)`
- Ключевые аргументы `def foo(*, a, b=True) # foo(a=1), foo(a=1, b=False)`
- Обязательно позиционный (3.8+), переменное количество позиционных и ключевых  
`def foo(a, /, b, *args, **kwargs): # foo(1, b=2, valuable_argument=42)`

# Вызов функции

```
def foo(x, y=2, *args, a=3, b, **kwargs):  
    # do something here  
extra_args = (9, 10)  
extra_kwargs = {'k': 'a', 'm': 'n'}  
foo(1, 2, *extra_args, 3, 4, **extra_kwargs, b=5, c=7, d=8)  
    ||  
    v  
foo(1, 2, 9, 10, 3, 4, k='a', m='n', b=5, c=7, d=8)
```

## Порядок передачи аргументов при вызове

1. Расставляются позиционные
2. Ключевые присваиваются
3. Избыточные позиционные в **args**
4. Избыточные ключевые в **kwargs**
5. Недостающие значения заполняются значениями по умолчанию (если есть)

В итоге каждый аргумент должен получить ровно одно значение (\* и \*\* могут остаться пустыми).

```
foo(another_function(), *extra_args, **extra_kwargs)
```

`another_function` будет вызвана до вызова `foo`

# Специальные случаи

## Аргументы передаются присваиванием

Можно повлиять на изменяемый объект вызывающей области.

Поэтому условно можно сказать, что неизменяемые типы данных передаются по значению, а изменяемые – по ссылке

```
def foo(a):  
    a.append(1)  
    mutable_element = []  
    foo(mutable_element)  
    assert mutable_element == [1]
```

## Заголовок функции выполняется только однажды

Аргументы по умолчанию инициализируются только один раз

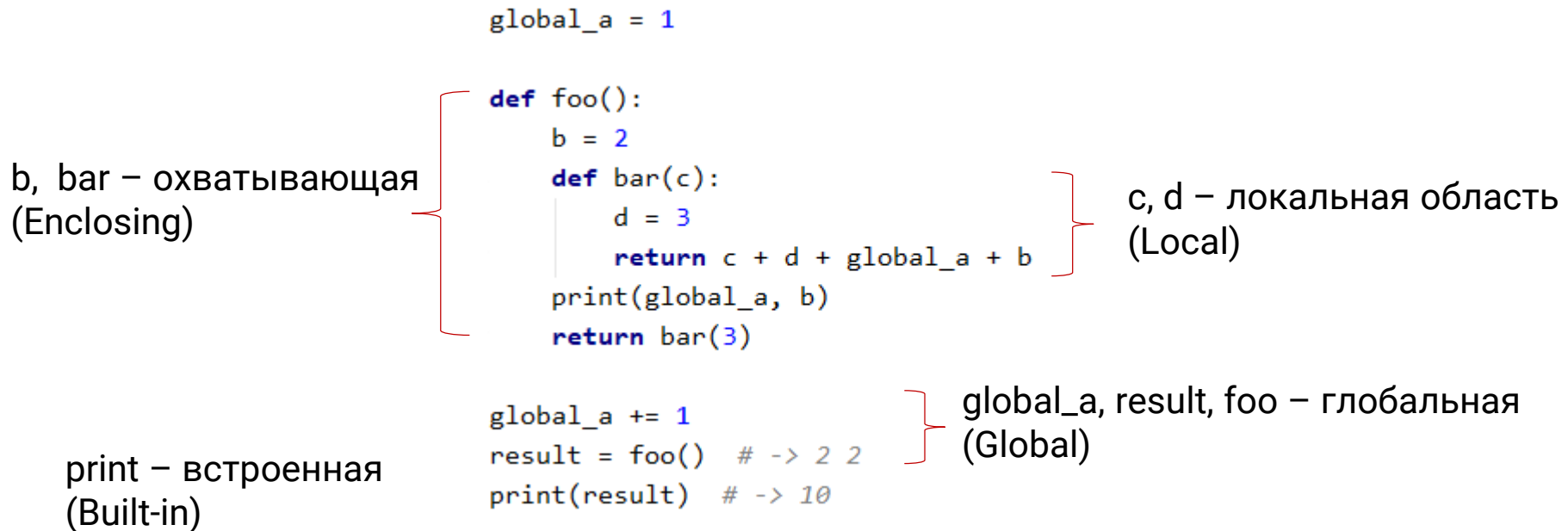
```
def bar(a=[]):  
    a.append(2)  
    return a  
bar()  
b = bar()  
assert b == [2, 2]
```



# Области видимости функции (правило LEGB)

Рассматривается относительно функции **bar**

Локальные области видимости создаются в момент вызова функции.



# Управление областями поиска переменной

## global

```
a = 3
def foo():
    a = 2
    def bar():
        global a
        a += 1
    bar()
    print(a) # -> 2
foo()
print(a) # -> 4
```

Если не найдена в глобальной области, то  
будет там создана

## nonlocal

```
a = 10
def foo():
    a = 2
    def bar():
        nonlocal a
        a += 1
    bar()
    print(a) # -> 3
foo()
print(a) # -> 10
```

Если не найдена в охватывающей  
области, то будет ошибка

# Рекурсия и замыкание

Рекурсия - это вызов функцией самой себя

```
def fibonacci(n):  
    if n == 1:  
        return 0  
    if n == 2:  
        return 1  
    return fibonacci(n-1) + fibonacci(n-2)  
print(fibonacci(10)) # -> 34
```

После завершения работы функции, ее локальные переменные уничтожаются, кроме тех, на которые остались ссылки, например во вложенной функции. Пример сохранения переменной в «замыкании» ниже.

```
def outer_function(a):  
    def inner_function(b):  
        return a * b  
    return inner_function  
mul_5 = outer_function(5)  
a = mul_5(11)  
assert a == 55
```

# Функция – это объект

Для полного списка атрибутов смотрите **dir(foo)**

```
def foo(a,b, *args):  
    return (a, b) + args  
  
bar = foo  
a = bar(1,2,3,4)  
print(a) # -> (1, 2, 3, 4)  
bar.depth = 10  
print(foo.depth) # -> 10  
print(type(bar)) # -> <class 'function'>  
print(foo.__name__) # -> foo  
print(foo.__code__)  
# -> <code object foo at 0x0000011111111111, file "/path/to/file/functions.py", line 22>
```

# Lambda-выражения

Компактный способ создания анонимной функции выражением. В теле может содержать только одно выражение, которое будет возвращаться, как результат работы функции.

```
def mul(a, b):  
    return a * b
```

```
mul = lambda a, b: a * b
```

```
def sum(a, b=4):  
    return a + b
```

```
sum = lambda a, b=4: a + b
```

# Армия функций

Нужно держать в уме, что создание локальных областей видимости происходит во время вызова функции

```
army = {}  
for i in range(5):  
    army[i] = lambda a: i + a  
del i  
print(army[0](10)) # -> NameError: name 'i' is not defined
```

```
army = {}  
for i in range(5):  
    army[i] = lambda a: i + a  
print(army[0](10)) # -> 14  
print(army[1](10)) # -> 14
```

Можно все усугубить, удалив `i` до вызова функций. Пользуйтесь `del` с осторожностью.

Один из способов решения проблемы ->

```
army[i] = lambda a, i=i: i + a
```

# Общие рекомендации по написанию функций

- Пользуйтесь аргументами для ввода и return для вывода, сокращайте количество побочных эффектов
- Избегайте использование глобальных переменных
- Старайтесь не менять изменяемые аргументы, переданные в функцию, кроме случаев, когда это явно подразумевается
- Каждая функция должна иметь одну задачу, которую можно выразить одним коротким предложением
- Маленькие функции проще читать и понимать
- Избегайте проверки на типы данных аргументов

# Аннотирование функций

```
def foo(a:int, b:'description'=3, *, c:tuple) -> dict:
    result = {}
    for i in range(a + b):
        result[i] = c
    return result
```

```
print(foo.__annotations__)
# {'a': <class 'int'>, 'b': 'description', 'c': <class 'tuple'>, 'return': <class 'dict'>}
foo(1,2,c=3)
```



# Встроенные функциональные средства

```
a = [0, 1, 2, 3, 0, 2]

b = list(map(lambda element: element * 3, a))
print(b) # [0, 3, 6, 9, 0, 6]

b = list(filter(lambda element: element > 1, a))
print(b) # [2, 3, 2]
```

## Встроенный модуль functools

```
from functools import reduce

b = reduce(lambda x, y: x + y, a, 1)
print(b) # 9

b = reduce(lambda x, y: x * y, filter(None, a), 1)
print(b) # 12
```


```
from functools import partial

def foo(a, b, *, c, d):
    return a, b, c, d

bar = partial(foo, 1, c=3, d=4)
b = bar(2, c=5)
print(b) # (1, 2, 5, 4)
```

enumerate, zip и остальные полезные функции изучайте самостоятельно

**Спасибо за  
внимание**

 **artezio\_software**

 **info@artezio.com**

**[www.artezio.com](http://www.artezio.com)**