

Функции часть I



Понятие функции

Функция в **Python** - объект, принимающий аргументы и возвращающий значение.

Вход в функцию - это передача ей аргументов - данных, полученных во внешней части программы. Получив данные, функция должна их как-то обработать: выполнить некоторые действия, вычислить какое-то значение. Выход из функции - значение, вычисленное блоком кода данной функции и передаваемое во внешнюю часть программы. Входные данные называют параметрами, а выходные - возвращаемым значением. Впрочем, функция может и не принимать никаких параметров. Что принимает в качестве параметров и что возвращает функция в результате своей работы, определяет программист.



Роль функции в программировании

1. Сокращение кода

Код, который повторяется можно перенести в функцию и использовать её тогда, когда нужно выполнить код, который находится внутри этой функции.

2. Логическое разделение программы

Мы можем выделить определённое сложное действие (например перемножение матриц) в отдельную функцию, чтобы оно не мешалось в коде, даже если используем её один раз за всё время выполнения программы.



Определение функции

```
# объявление функции my_function()
```

```
def my_function([параметр1, параметр2, ...]):  
    # тело функции
```

```
    # возвращаемое значение  
    return result # необязательно
```

```
# вызов функции  
my_function([аргумент1 , аргумент2, ...] )
```

```
type(my_function)  
<class 'function'>    - еще один тип в Python
```



Пример:

#Определение функции:

```
def summ(x, y):  
    result = x + y  
    return result
```

#ВЫЗОВ функции

```
a = 100
```

```
b = -50
```

```
answer = summ(a, b)
```

```
print(answer)
```

```
50
```



Вызов функции

Можно ли так вызвать функцию ?

```
summ(10)
```

```
def summ(x):  
    print(x)
```

NameError: name 'summ' is not defined



Что вернет функция без return:

#Определение функции:


```
def summ(x, y):  
    result = x + y
```

#ВЫЗОВ функции

```
answer = summ(100, -50)  
print(answer)      - ?
```

answer is None

True



Что вернет функция в отсуствии аргументов ?

#Определение функции:

```
def summ(x, y):  
    result = x + y  
    return result;
```

#вызов функции

```
answer = summ(100)
```

```
TypeError: summ() missing 1 required positional  
argument: 'y'
```




Параметры по умолчанию

Для некоторых параметров в функции можно указать значение по умолчанию, таким образом если для этого параметра не будет передано значение при вызове функции, то ему будет присвоено значение по умолчанию.

```
def premium(salary, percent=10):  
    p = salary * percent / 100  
    return p
```

```
result = premium(60000)  
print(result)
```

```
6000.0
```

```
print(premium(60000, 20))
```

```
12000.0
```



Пустая функция

```
def empty(var1, var2):  
    pass
```

```
result = empty(8, 10)  
print(result)
```

None



Именованные параметры


Иногда происходит такая ситуация, что функция требует большое количество аргументов. Пример:

```
def my_func(arg1, arg2, arg3, arg4, arg5, arg6):  
    pass # оператор, если кода нет
```

```
result = my_func(1, 2, 3, 4, 5, 6)
```

В такой ситуации код не всегда удобно читать и тяжело понять, какие переменные к каким параметрам относятся.

```
result = my_func(arg2=2, arg1=1, arg4=4, arg5=5,  
arg3=3, arg6=6)
```



Функция с неограниченным количеством позиционных аргументов


***args** – произвольное число позиционных аргументов

```
def manyargs(var1, *args):  
    print(type(args))  
    print(args)
```

```
result = manyargs(4, 9, 1, 3, 3, 1)
```

```
<class 'tuple'>
```

```
(9, 1, 3, 3, 1)
```



Функции с неограниченным количеством именованных аргументов

****kwargs** – произвольное число именованных аргументов.

```
def manykwargs(**kwargs):  
    print(type(kwargs))  
    print(kwargs)
```

```
manykwargs(name='Piter', age=20)
```

```
<class 'dict'>
```

```
{'name': 'Piter', 'age': 20}
```

Можно ли мне по другому назвать параметр ****kwargs** ?

Можно, только вас никто не поймет!



Все вместе.

***args** – произвольное число позиционных аргументов

****kwargs** – произвольное число именованных аргументов.

```
def many_all(var1, *args, **kwargs):  
    print(var1)  
    print(args)  
    print(kwargs)
```

```
many_all(10, 34, 77, name='Piter', age=20)
```

```
10
```

```
(34, 77)
```

```
{'name': 'Piter', 'age': 20}
```

Можно ли мне по другому назвать параметр ****kwargs** ?

Можно, только вас никто не поймет!



Scope (область видимости)

Область видимости указывает интерпретатору, когда наименование (или переменная) видима. Другими словами, область видимости определяет, когда и где вы можете использовать свои переменные, функции и т.д. Если вы попытаетесь использовать что-либо, что не является в вашей области видимости, вы получите ошибку `NameError`. Python содержит три разных типа области видимости:

- * Локальная область видимости
- * Глобальная область видимости
- * Нелокальная область видимости (была добавлена в Python 3)



Локальная область видимости

Локальная область видимости (`local`) — это блок кода или тело любой функции Python или лямбда-выражения. Эта область Python содержит имена, которые вы определяете внутри функции.

```
x = 100
```

```
def doubling(y):
```

```
    z = y*y
```

```
doubling(x)
```

```
print(z)
```

```
NameError: name 'z' is not defined
```




Глобальная область видимости

Глобальная область видимости(global). В Python есть ключевое слово `global`, которое позволяет изменять изнутри функции значение глобальной переменной. Оно записывается перед именем переменной, которая дальше внутри функции будет считаться глобальной.

```
x = 100
```

```
def doubling(y):
```

```
    global x
```

```
    x = y*y
```

```
doubling(x)
```

```
print(x)
```

```
10000
```



Нелокальная область видимости

В Python 3 было добавлено новое ключевое слово под названием **nonlocal**. С его помощью мы можем добавлять переопределение области во внутреннюю область.

```
def counter():  
    num = 0  
    def incrementer():  
        num += 1  
        return num  
    return incrementer
```

Если вы попыгаете запустить этот код, вы получите ошибку `UnboundLocalError`, так как переменная `num` ссылается прежде, чем она будет назначена в самой внутренней функции.



Нелокальная область видимости

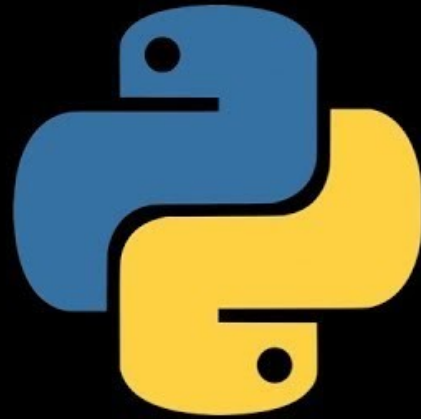
Добавим `nonlocal` в наш код:

```
def counter():  
    num = 0  
    def incrementer():  
        nonlocal num  
        num += 1  
        return num  
    return incrementer
```

```
inc = counter()
```

```
inc()
```

```
1
```



PYTHON

PROGRAMMING