

Тема 3.11 ФУНКЦИИ В PYTHON

Подпрограммы прежде всего необходимы в ситуации, когда в разных частях программы необходимо выполнять одни и те же действия несколько раз. В таком случае повторяемые операторы оформляются в виде функции или процедуры, к которой можно обращаться и вызывать ее выполнение из разных частей программы.

Функция – особым образом сгруппированный набор команд, которые выполняются последовательно, но воспринимаются как единое целое. При этом функция может возвращать (или не возвращать) свой результат.

Функции – это такие участки кода, которые изолированы от остальной программы и выполняются только тогда, когда вызываются. Вы уже встречались с функциями `sqrt()`, `len()` и `print()`. Они все обладают общим свойством: они могут принимать параметры (ноль, один или несколько), и они могут возвращать значение (хотя могут и не возвращать). Например, функция `sqrt()` принимает один параметр и возвращает значение (корень числа). Функция `print()` принимает переменное число параметров и ничего не возвращает.

```
def <имя функции>([аргументы]):  
    <тело функции>
```

После функции до кода, который находится вне функции, необходимо делать *отступ в две пустые строки* для повышения читаемости кода. Если у вас есть несколько функций в одном файле, между кодом одной и сигнатурой другой функции тоже надо оставлять две пустые строки.

Покажем, как написать функцию `factorial()`, которая принимает один параметр – число, и возвращает значение – факториал этого числа.

```
def factorial():  
    res = 1  
    for i in range(1, n + 1):  
        res *= i  
    return res  
print(factorial(3))  
print(factorial(5))
```

Функция задается один раз, вызываться может сколько угодно с разными значениями аргументов.

Далее идет тело функции, оформленное в виде блока, то есть с отступом. Внутри функции вычисляется значение факториала числа `n` и оно сохраняется в переменной `res`. Функция завершается инструкцией `return res`, которая завершает работу функции и возвращает значение переменной `res`.

Локальные и глобальные переменные

Переменные, создаваемые внутри функций, недоступны извне и существуют только внутри функции. Они называются локальными.

Создаваемые вне функции переменные могут быть доступны из функций. Они являются глобальными.

Внутри функции можно использовать переменные, объявленные вне этой функции

```
def f():  
    print(a)  
    a = 1  
f()
```

Здесь переменной `a` присваивается значение 1, и функция `f()` печатает это значение, несмотря на то, что до объявления функции `f` эта переменная не инициализируется. В момент вызова функции `f()` переменной `a` уже присвоено значение, поэтому функция `f()` может вывести его на экран.

Такие переменные (объявленные вне функции, но доступные внутри функции) называются *глобальными*.

```
def f():  
    a = 1  
f()  
print(a)
```

Получим ошибку `NameError: name 'a' is not defined`. Такие переменные, объявленные внутри функции, называются *локальными*. Эти переменные становятся недоступными после выхода из функции.

Рекурсия

Раньше мы рассматривали примеры, в которых функция может вызывать сама себя. Например, функция вычисления факториала. Хорошо известно, что $0! = 1$, $1! = 1$. А как вычислить величину $n!$ для большого n ? Если бы мы могли вычислить величину $(n-1)!$, то тогда мы легко вычислим $n!$, поскольку $n! = n \cdot (n-1)!$. Но как вычислить $(n-1)!$? Если бы мы вычислили $(n-2)!$, то мы сможем вычислить и $(n-1)! = (n-1) \cdot (n-2)!$. А как вычислить $(n-2)!$? Если бы... В конце концов, мы дойдем до величины $0!$, которая равна 1. Таким образом, для вычисления факториала мы можем использовать значение факториала для меньшего числа. Это можно сделать и в программе на Питоне:

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
print(factorial(5))
```

Подобный прием (вызов функцией самой себя) называется рекурсией, а сама функция называется рекурсивной.

Рекурсивные функции являются мощным механизмом в программировании. К сожалению, они не всегда эффективны. Также часто использование рекурсии приводит к ошибкам. Наиболее распространенная из таких ошибок – бесконечная рекурсия, когда цепочка вызовов функций никогда не завершается и продолжается, пока не кончится свободная память в компьютере. Пример бесконечной рекурсии приведен в эпиграфе к этому разделу. Две наиболее распространенные причины для бесконечной рекурсии:

- 1) Неправильное оформление выхода из рекурсии. Например, если мы в программе вычисления факториала забудем поставить проверку `if n == 0`, то `factorial(0)` вызовет `factorial(-1)`, тот вызовет `factorial(-2)` и т. д.
- 2) Рекурсивный вызов с неправильными параметрами. Например, если функция `factorial(n)` будет вызывать `factorial(n)`, то также получится бесконечная цепочка.

Поэтому при разработке рекурсивной функции необходимо прежде всего оформлять условия завершения рекурсии и думать, почему рекурсия когда-либо завершит работу.

Функция filter

Функция `filter` принимает критерий отбора элементов, а затем сам список элементов. Возвращает она список из элементов, удовлетворяющих критерию.

Чтобы этой функцией воспользоваться, нужно сообщить функции `filter` критерий, который говорит, брать элемент в результирующий список или нет. Давайте напишем простую функцию, которая проверяет, что слово длиннее шести букв, и затем отберем с ее помощью длинные слова.

```
def is_word_long(word):
    return len(word) > 6
words = ['В', 'новом', 'списке', 'останутся', 'только', 'длинные', 'слова']
for word in filter(is_word_long, words):
    print(word)
# => останутся
# => длинные
```

С методом `filter` вам не нужно вручную создавать и заполнять список, достаточно указать условие отбора.

Лямбда-функции

Часто в качестве аргумента для функций высшего порядка мы хотим использовать совсем простую функцию. Причем нередко такая функция нужна в программе только в одном месте, поэтому ей необязательно даже иметь имя.

Такие короткие безымянные (анонимные) функции можно создавать инструкцией

lambda <аргументы>: <выражение>.

Такая инструкция создаст функцию, принимающую указанный список аргументов и возвращающую результат вычисления выражения.

В языке Python тело лямбда-функции имеет ровно одно выражение. Скобки вокруг аргументов не пишутся, аргументы от выражения отделяет двоеточие.

Теперь мы можем записать функцию, проверяющую длину слова, следующим образом:

```
lambda word: len(word) > 6
```

И список длинных слов теперь извлечь очень просто:

```
long_words = list(filter(lambda word: len(word) > 6, words))
```

Лямбда-функция – полноценная функция. Ее можно использовать в составе любых конструкций. Например, созданную лямбда-функцию можно присвоить какой-либо переменной:

```
add = lambda x, y: x + y
```

```
add(3, 5) # => 8
```

```
add(1, add(2, 3)) # => 6
```

Функция map

Функция map преобразует каждый элемент списка по некоторому общему правилу и в результате создает список из преобразованных значений.

Функция, которую map принимает, – преобразование одного элемента. Зная, как преобразуется один элемент, map выполняет превращение целых списков.

Например, возьмем набор из чисел от 1 до 10 и применим к ним функцию возведения в квадрат.

```
list(map(lambda x: x ** 2, range(1, 10)))
```

```
# => [1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Преобразовать список слов в список длин слов мы можем следующим образом:

```
words = 'the quick brown fox jumps over the lazy dog'.split()
```

```
list(map(lambda word: len(word), words))
```

```
# => [3, 5, 5, 3, 5, 4, 3, 4, 3]
```

```
list(map(len, words))
```

```
# => [3, 5, 5, 3, 5, 4, 3, 4, 3]
```

Еще один пример – считывание списка чисел с клавиатуры:

```
numbers = list(map(float, input().split()))
```

Пример преобразования каждого слова в списке к верхнему регистру:

```
words = ['list', 'of', 'several', 'words']
```

```
list(map(lambda word: word.upper(), words))
```

```
# => ['LIST', 'OF', 'SEVERAL', 'WORDS']
```

```
list(map(str.upper, words))
```

```
# => ['LIST', 'OF', 'SEVERAL', 'WORDS']
```