# Лабораторная работа №8
# Функции и рекурсия

## 1 Цель работы

Изучить функции и рекурсию и научиться применять полученные знания на практике.

## 2 Краткая теория

### 2.1 Функции

Напомним, что в математике факториал числа n определяется как n! = 1 · 2 · ... · n. Например, 5! = 1 · 2 · 3 · 4 · 5 = 120. Ясно, что факториал можно легко посчитать, воспользовавшись циклом for. Представим, что нам нужно в нашей программе вычислять факториал разных чисел несколько раз (или в разных местах кода). Конечно, можно написать вычисление факториала один раз, а затем используя Copy-Paste вставить его везде, где это будет нужно.

```
# вычислим 3!
res = 1
for i in range(1, 4):
    res *= i
print(res)

# вычислим 5!
res = 1
for i in range(1, 6):
    res *= i
print(res)
```

Однако, если мы ошибёмся один раз в начальном коде, то потом эта ошибка попадёт в код во все места, куда мы скопировали вычисление факториала. Да и вообще, код занимает больше места, чем мог бы. Чтобы избежать повторного написания одной и той же логики, в языках программирования существуют функции.

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.

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

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

```
print(factorial(3))
print(factorial(5))
```

Дадим несколько объяснений. Во-первых, код функции должен размещаться в начале программы, вернее, до того места, где мы захотим воспользоваться функцией factorial(). Первая строчка этого примера является описанием нашей функции. factorial – идентификатор, то есть имя нашей функции. После идентификатора в круглых скобках идет список параметров, которые получает наша функция. Список состоит из перечисленных через запятую идентификаторов параметров. В нашем случае список состоит из одной величины n. В конце строки ставится двоеточие.

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.

Инструкция return может встречаться в произвольном месте функции, ее исполнение завершает работу функции и возвращает указанное значение в место вызова. Если функция не возвращает значения, то инструкция return используется без возвращаемого значения. В функциях, которым не нужно возвращать значения, инструкция return может отсутствовать.

Приведём ещё один пример. Напишем функцию max(), которая принимает два числа и возвращает максимальное из них (на самом деле, такая функция уже встроена в Питон).

```
def max(a, b):
    if a > b:
        return a
    else:
        return b

print(max(3, 5))
print(max(5, 3))
print(max(int(input()), int(input())))
```

Теперь можно написать функцию max3(), которая принимает три числа и возвращает максимальное их них.

```
def max(a, b):
    if a > b:
        return a
    else:
        return b

def max3(a, b, c):
    return max(max(a, b), c)

print(max3(3, 5, 4))
```

Встроенная функция max() в Питоне может принимать переменное число аргументов и возвращать максимум из них. Приведём пример того, как такая функция может быть написана.

```
def max(*a):
    res = a[0]
```

```
    for val in a[1:]:
        if val > res:
            res = val
    return res

print(max(3, 5, 4))
```

Все переданные в эту функцию параметры соберутся в один кортеж с именем a, на что указывает звёздочка в строке объявления функции.

### 2.2 Локальные и глобальные переменные

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.

Но если инициализировать какую-то переменную внутри функции, использовать эту переменную вне функции не удастся. Например:

```
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.

Интересным получится результат, если попробовать изменить значение глобальной переменной внутри функции:

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

a = 0
f()
print(a)
```

Будут выведены числа 1 и 0. Несмотря на то, что значение переменной a изменилось внутри функции, вне функции оно осталось прежним! Это сделано в целях «защиты» глобальных переменных от случайного изменения из функции. Например, если функция будет вызвана из цикла по переменной i, а в этой функции будет использована переменная i также для организации цикла, то эти переменные должны быть различными. Если вы не поняли последнее предложение, то посмотрите на следующий код и подумайте, как бы он работал, если бы внутри функции изменялась переменная i.

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

for i in range(1, 6):
    print(i, '! = ', factorial(i), sep='')
```

Если бы глобальная переменная i изменялась внутри функции, то мы бы получили вот что:

```
5! = 1
5! = 2
5! = 6
5! = 24
5! = 120
```

Итак, если внутри функции модифицируется значение некоторой переменной, то переменная с таким именем становится локальной переменной, и ее модификация не приведет к изменению глобальной переменной с таким же именем.

Более формально: интерпретатор Питон считает переменную локальной для данной функции, если в её коде есть хотя бы одна инструкция, модифицирующая значение переменной, то эта переменная считается локальной и не может быть использована до инициализации. Инструкция, модифицирующая значение переменной – это операторы =, +=, а также использование переменной в качестве параметра цикла for. При этом даже если инструкция, модицифицирующая переменную никогда не будет выполнена, интерпретатор это проверить не может, и переменная все равно считается локальной. Пример:

```
def f():
    print(a)
    if False:
        a = 0

a = 1
f()
```

Возникает ошибка: UnboundLocalError: local variable 'a' referenced before assignment. А именно, в функции f() идентификатор a становится локальной переменной, т.к. в функции есть команда, модифицирующая переменную a, пусть даже никогда и не выполняющийся (но интерпретатор не может это отследить). Поэтому вывод переменной a приводит к обращению к неинициализированной локальной переменной.

Чтобы функция могла изменить значение глобальной переменной, необходимо объявить эту переменную внутри функции, как глобальную, при помощи ключевого слова global:

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

```
a = 0
f()
print(a)
```

В этом примере на экран будет выведено 1 1, так как переменная a объявлена, как глобальная, и ее изменение внутри функции приводит к тому, что и вне функции переменная будет доступна.

Тем не менее, лучше не изменять значения глобальных переменных внутри функции. Если ваша функция должна поменять какую-то переменную, пусть лучше она вернёт это значением, и вы сами при вызове функции явно присвоите в переменную это значение. Если следовать этим правилам, то функции получаются независимыми от кода, и их можно легко копировать из одной программы в другую.

Например, пусть ваша программа должна посчитать факториал вводимого числа, который вы потом захотите сохранить в переменной f. Вот как это не стоит делать:

```
def factorial(n):
    global f
    res = 1
    for i in range(2, n + 1):
        res *= i
    f = res

n = int(input())
factorial(n)
# дальше всякие действия с переменной f
```

Этот код написан плохо, потому что его трудно использовать ещё один раз. Если вам завтра понадобится в другой программе использовать функцию «факториал», то вы не сможете просто скопировать эту функцию отсюда и вставить в вашу новую программу. Вам придётся поменять то, как она возвращает посчитанное значение.

Гораздо лучше переписать этот пример так:

```
# начало куска кода, который можно копировать из программы в программу
def factorial(n):
    res = 1
    for i in range(2, n + 1):
        res *= i
    return res
# конец куска кода

n = int(input())
f = factorial(n)
# дальше всякие действия с переменной f
```

Если нужно, чтобы функция вернула не одно значение, а два или более, то для этого функция может вернуть список из двух или нескольких значений:

```
return [a, b]
```

Тогда результат вызова функции можно будет использовать во множественном присваивании:

```
n, m = f(a, b)
```

Тогда результат вызова функции можно будет использовать во множественном присваивании:

### 2.3 Рекурсия

Как мы видели выше, функция может вызывать другую функцию. Но функция также может вызывать и саму себя! Рассмотрим это на примере функции 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·(n-1)!. Но как вычислить (n-1)!? Если бы мы вычислили (n-2)!, то мы сможем вычисли и (n-1)!=(n-1)·(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.

## 3 Порядок выполнения работы

Получить задание для выполнения лабораторной работы (раздел 4) согласно своему варианту (по журналу). Разработать программу.

## 4 Задания для выполнения работы

**Задание 1. Длина отрезка.**

Даны четыре действительных числа: $x_1$, $y_1$, $x_2$, $y_2$. Напишите функцию distance(x1, y1, x2, y2), вычисляющая расстояние между точкой $(x_1, y_1)$ и $(x_2, y_2)$. Считайте четыре действительных числа и выведите результат работы этой функции.

Если вы не знаете, как решить эту задачу, то вы, возможно, не изучали в школе теорему Пифагора. Попробуйте прочитать о ней на Википедии.

**Задание 2. Отрицательная степень.**

Дано действительное положительное число $a$ и целое число $n$.

Вычислите $a^n$. Решение оформите в виде функции power(a, n).

Стандартной функцией возведения в степень пользоваться нельзя.

**Задание 3. Большие буквы.**

Напишите функцию capitalize(), которая принимает слово из маленьких латинских букв и возвращает его же, меняя первую букву на большую.

Например, print(capitalize('word')) должно печатать слово Word.

На вход подаётся строка, состоящая из слов, разделённых одним пробелом. Слова состоят из маленьких латинских букв. Напечатайте исходную строку, сделав так, чтобы каждое слово начиналось с большой буквы. При этом используйте вашу функцию capitalize().

Напомним, что в Питоне есть функция ord(), которая по символу возвращает его код в таблице ASCII, и функция chr(), которая по коду символа возвращает сам символ. Например, ord('a') == 97, chr(97) == 'a'.

**Задание 4. Возведение в степень.**

Дано действительное положительное число a и целое неотрицательное число $n$. Вычислите $a^n$ не используя циклы, возведение в степень через ** и функцию math.pow(), а используя рекуррентное соотношение $a^n = a \cdot a^{n-1}$.

Решение оформите в виде функции power(a, n).

**Задание 5. Разворот последовательности.**

Дана последовательность целых чисел, заканчивающаяся числом 0. Выведите эту последовательность в обратном порядке.

При решении этой задачи нельзя пользоваться массивами и прочими динамическими структурами данных. Рекурсия вам поможет.

**Задание 6. Числа Фибоначчи.**

Напишите функцию fib(n), которая по данному целому неотрицательному $n$ возвращает $n$-е число Фибоначчи. В этой задаче нельзя использовать циклы – используйте рекурсию.