**Рекурсивные и лямбда-функции**

В Python функцию можно вызывать саму из себя. Это называется **рекурсией**. В качестве примера рекурсивной функции я приведу вычисление числа в степени n (n – целое число).

def func\_rec(x, n):  
 if n == 0:  
 return 1  
 else:  
 return x \* func\_rec(x, n-1)  
  
  
print(func\_rec(2, 3))

Теперь подробнее разберемся как она работает. Для начала заметим, что



то есть, для вычисления значения на текущем n-м шаге достаточно взять значение на предыдущем n-1-м шаге и умножить его на x. Эта формула, по сути, отражает принцип рекурсии. В нашем случае она будет выглядеть так:

x\* func\_rec(x,n-1)

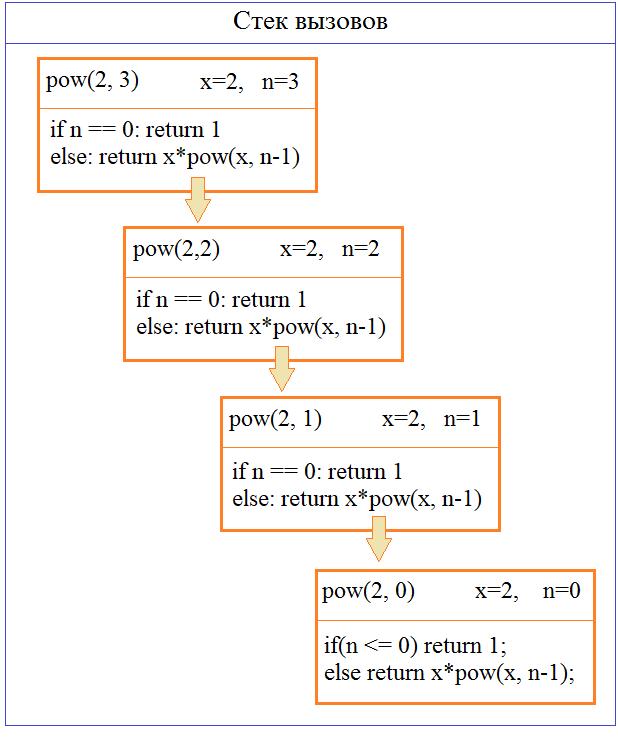
Далее, запуск функции осуществляется с аргументами 2 и 3: Она помещается в стек вызова функций, в котором хранится порядок вызова различных функций. Далее, выполняется тело функции. Проверяется первое условие. Оно оказывается ложным, так как 3 == 0 дает false. Поэтому идет переход на else и прежде, чем выполнить оператор return, снова вызывается та же функция func\_rec (2, 2).

Выполнение функции func\_rec (2, 3) останавливается, в стек помещается вторая функция func\_rec (2, 2) и она запускается. Снова проверяется первое условие, оно ложно, переходим по else к оператору return и опять вызываем func\_rec(2, 1).

Здесь снова все повторяется, в стек помещается очередной вызов функции и по условию вызывается следующая функция func\_rec (2, 0).

Теперь первое условие становится истинным и функция func\_rec (2,0) возвращает значение 1 и рекурсия не идет дальше вглубь – она останавливается. Функция func\_rec (2.0) завершается, она удаляется из стека вызовов и управление передается функции func\_rec (2, 1). Но она частично уже выполнена. Поэтому, берется значение 1 от func\_rec (2, 0), результат умножается на x=2 и величина 2 возвращается функцией func\_rec (2, 1).

Функция func\_rec (2,1) также удаляется из стека, управление переходит к вышестоящей функции func\_rec (2,2) и здесь мы уже имеем результат умножения x\*x, то есть, 2\*2 = 4. Далее, возвращаемся к самой верхней функции func\_rec (2,3), здесь 2\*4=8 и этот результат становится результатом вычисления рекурсивной функции.



**Подвиг**

Вводится целое положительное число N. Необходимо написать рекурсивную функцию, которая отображает на экране последовательность целых чисел от 1 до N (включительно). Каждое число выводится с новой строки.

В качестве параметра функция должна принимать одно числовое значение. То есть, иметь только один параметр.

**Анонимные или лямбда-функции**

В Python существуют так называемые анонимные функции (они же лямбда-функции). Что это такое? Давайте для начала рассмотрим такой пример:

**def** showElements(lst, func):

**for** x **in** lst:

**if** func(x):

**print**(x)

**def** \_\_odd(x):

**return** True **if** x%2 != 0 **else** False

a = [1,2,3,4,5,6,7]

showElements(a, \_\_odd)

У нас имеется функция showElements, которая отображает из списка lst только те элементы, для которых функция func возвращает значение True. Далее, мы создаем вспомогательную функцию \_\_odd, которая возвращает True для нечетных значений. И идет вызов showElements для списка a и селектора в виде функции \_\_odd.

Но что если мы захотим изменить селектор и выбирать, например, все четные значения? Нужно создавать новую функцию и затем указывать ссылку на нее? Это как то не очень удобно. Вот здесь нам на помощь приходят лямбда-функции. Их можно объявить в любом месте программы Например, так:

r = **lambda** a,b: a+b

**print**( r(1,2) )

То есть, в анонимных функциях можно писать после двоеточий любой оператор, но только один. Несколько нельзя. И кроме того, результат работы этого оператора автоматически возвращается лямбда-функцией.

В нашем случае мы можем записать такую функцию сразу в качестве второго аргумента:

showElements(a, **lambda** x: True **if** x%2==0 **else** False)

Видите, как это удобно, понятно и наглядно. Мы сразу видим как будет выбирать значения функция showElements.

Если анонимная функция не принимает никаких аргументов, то она записывается так:

p = **lambda** : "hello world"

p()

**Подвиг 1.**

Вводится целое неотрицательное число n. Необходимо с помощью рекурсивной функции вычислить факториал числа n. Напомню, что факториал числа, равен: n! = 1 \* 2 \* 3 \*...\* n. Функция должна возвращать вычисленное значение.

**Подвиг 2.**

Вводится список целых чисел в одну строчку через пробел. Необходимо вычислить сумму этих введенных значений, используя рекурсивную функцию (для перебора элементов списка). Функция должна возвращать значение суммы. (Выводить на экран она ничего не должна).

Вызовите эту функцию и выведите вычисленное значение суммы на экран.

**Sample Input:** 8 11 -5 4 3

**Sample Output:** 21

**Полуподвиги**

- Объявите анонимную функцию с одним параметром для возведения числа в квадрат. Присвойте эту функцию переменной get\_sq.

- Объявите анонимную функцию с двумя параметрами для деления одного целого числа на другое. Если происходит деление на ноль, то функция должна возвращать значение None, иначе - результат деления.

- Объявите анонимную функцию для вычисления модуля числа (то есть, отрицательные числа нужно делать положительными).

**Области видимости переменных**

*Глобальная переменная – это переменная доступная в любом месте программы.*

Как можно ее задать? Очень просто – в начале текстового файла, например:

a = 5

N = (100, )

WIDTH, HEIGHT = (1000, 500)

Однако, использование глобальных переменных считается плохим стилем программирования. И допускается, разве что, задавать глобальные константы, используемые во всей программе. А имена констант записывать заглавными буквами, чтобы отличать их от обычных не глобальных (то есть, локальных) переменных.

*Локальные переменные – это переменные, объявленные внутри любого блока программы.*

Например:

**def** myFunc(b):

**for** x **in** range(b):

        n = x+1

**print**(n, end=" ")

myFunc(6)

Здесь у нас сразу три локальных переменных: b, n, x. Почему локальные? Дело в том, что они доступны только внутри функции myFunc и не существуют за ее пределами. Если мы попытаемся после функции выполнить операцию, например:

**print**(x)

то возникнет ошибка, что x не существует. А вот внутри функции все три переменные существуют:

**print**(b, n, x)

Ну, хорошо, а как мы можем работать с глобальной переменной внутри этой функции?

Более наглядный пример использования локальной переменной с тем же именем, что и глобальная, такой:

name = "Tom"

**def** say\_hi():

**print**("Hello", name)

**def** say\_bye():

    name = "Bob"

**print**("Good bye", name)

say\_hi()

say\_bye()

В консоли увидим:

Hello Tom  
Good bye Bob

То есть, первая функция использовала глобальную переменную name, т.к. никакой другой для нее не существовало, а вторая функция использовала локальную name, объявленную внутри нее. Этот пример показывает, что функция сначала пытается найти нужную переменную внутри собственной области видимости, и если не находит, то переходит на более высокий уровень, в данном случае – глобальной области.

Если же мы хотим поменять именно глобальную переменную, то в функции следует вначале это явно сказать: global a

тем самым мы указываем, что хотим работать с глобальной a, а не создавать локальную переменную. Теперь, при выполнении этой программы, мы видим измененное значение глобальной переменной a.

В Python имеется один интересный режим работы с локальными переменными с использованием ключевого слова **nonlocal**. Давайте предположим, что у нас имеется объявление одной функции внутри другой:

x = 0

**def** outer():

    x = 1

**def** inner():

        x = 2

**print**("inner:", x)

    inner()

**print**("outer:", x)

outer()

**print**("global:", x)

При запуске программы мы увидим ожидаемые результаты:

inner: 2  
outer: 1  
global: 0

А теперь внутри функции inner мы скажем, что хотим работать с переменной x, объявленной уровнем выше, то есть, внутри функции outer:

**nonlocal** x

Теперь строка x=2 будет означать изменение переменной x в функции outer и при запуске программы получим результаты:

inner: 2  
outer: 2  
global: 0

Но так можно делать только с локальными переменными. С глобальной работать не будет. Если мы пропишем строчку

**nonlocal** x

в функции outer, то возникнет ошибка, т.к. уровнем выше находится уже глобальная область. Здесь, вместо nonlocal следует уже использовать global:

**global** x

а nonlocal в inner убрать, иначе опять же получится ссылка на глобальную переменную.

Итак, на этом занятии мы с вами рассмотрели глобальную и локальную области видимости переменных и познакомились с ключевыми словами global и nonlocal.