Функции

1. Определение и вызов

Функции – это многократно используемые фрагменты программы. При помощи функций можно объединить несколько инструкций в один блок, присвоить этому блоку имя и затем, обращаясь по имени этого блока, выполнить инструкции внутри него в любом месте программы необходимое число раз.

Пример:

```
print('hello')
print('world')
print('and everybody')
```

Это самая простая программа, которая может быть, если мы ее запустим, на экране увидим соответственно эти три сообщения.

Так вот, функция позволяет нам объединить несколько инструкций в одинн блок. Перед началом блока пишем ключевое слово def, сокращение от слова definition, переводится как определение. После этого вы должны указать имя блока или другими словами имя нашей функции.

Давайте в нашем примере назовем функцию sayHello:

```
def sayHello():
    print('hello')
    print('world')
    print('and everybody')
```

Отступами мы показываем какие инструкции будут входить в наш блок. Это называется определением функции. Само по себе определение ничего не делает, т.е. если мы на этом этапе запустим программу, то ничего не произойдет, потому что пока нигде в нашей программе мы не обращались к этому участку кода.

Для того, чтобы вызвать функцию нам необходимо выйти из блока тела функции и обратиться по имени к этой функции.

```
def sayHello():
    print('hello')
    print('world')
    print('and everybody')
sayHello()
```

Еще одну функцию создадим и назовем ее square.

```
def square(x):
    print('Квадрат числа', x, 'равен', x**2)


square(4)
```

И, например, давайте внутри цикла for попытаемся вызвать функцию.

```
def square(x):
    print('Квадрат числа', x, 'равен', x**2)
for x in range (1, 6):
    square(x)
```

Соответственно такой кусок выведет нам квадрат всех чисел в диапазоне от 1 до 5 включительно.

Внутри функции можно использовать и другие конструкции, условные операторы, циклы.

Допустим мы хотим написать функцию, которая будет проверять число на четность.

На вход она примет один аргумент – само число, которое нужно проверить.

```
def even(a):
    if a%2==0:
        print ('четное')
```

```
    else:
        print('нечетное')


for i in range (13, 20):
    even(i)
```

Ну и давайте просмотрим как писать цикл внутри функции. Для этого напишем функцию нахождения факториала.

```
def factorial(n):
    pr=1
    for i in range (2, n+1):
        pr*=i
    print(pr)
factorial(4)
```

Еще один интересный вариант создания функции. Для этого нам понадобится условный оператор.

```
if 5>1:
    def primer ():
        print('hello')
else:
    def primer ():
        print('HELLO')



primer()
```

В данном случае в зависимости от условия мы может определить одну и ту же функцию с разными выводами.

При этом если мы уберем условный оператор и оставим только два определения функции и запустим, у нас в конечном счете выведется то, что было записано последним, так как программа выполняется сверху вниз.

2. Области видимости. Глобальные и локальные переменные

```
def myFunc(b):
    for x in range(b):
        n=x+1
        print(n)
myFunc(6)
```

Так как функция – это отдельный блок программы, все переменные, объявленные внутри функции являются локальными, то есть в нашем примере это b, x, n. Все эти переменные доступны только внутри этой функции myFunc, за пределами которой мы не может обратиться к этим переменным. Стоит обратить внимание, что операторы цикла не образуют свои области видимости и все переменные объявленные внутри них доступными и за их пределами (в том же C++, Java это не так).

А как работать с глобальными переменными внутри функции.

То есть допустим мы изначально определили переменную a равную 5.

Потом внутри нашей функции переопределим ее значение на 10.

```
a=5
def myFunc(b):
    a=10
    for x in range(b):
        n=x+1
        print(n)
myFunc(6)
print(a)
```

Если мы запустим этот кусок результатом print(a) будет 5. Это происходит потому что, когда мы определяем внутри функции переменную a, создается новая локальная переменная с точно таким же именем, как у глобальной и она никак не влияет на значение глобальной переменной.

```
name ='Магомед'
def say_hi():
    print('hello', name)

def say_bye():
    #name='Патя'
    print('hello', name)

say_hi()
say_bye()
```

То есть поиск переменной осуществляется так: сначала переменная name ищется внутри функции, если она там не находится переходим в глобальную область и ищем переменную там.

Но, что делать если мы хотим внутри функции работать с глобальной переменной?

```
a=5
def myFunc(b):
    global a
    a=10
    for x in range(b):
        n=x+1
        print(n)
myFunc(6)
print(a)
```

Для этого прописываем внутри функции global a, это означает уже, что мы будем работать с переменной a объявленной выше. И если мы запустим этот кусок кода, то он нам уже выдаст a=10.

Если до начала объявления функции не была инициирована глобальная переменная, то уже внутри функции с помощью global мы ее создаем:

```
def myFunc(b):
    global a
    a=10
    for x in range(b):
        n=x+1
        print(n)
myFunc(6)
print(a)
```

В пайтон имеется один интересный режим работы с локальным переменными с использованием ключевого слова nonlocal. Что это такое? Давайте рассмотрим на таком примере:

```
#nonlocal
x=0 #глобальная
def outer():
   x=1 #внутри первой функции
   def inner():
      x=2 #внутри вложенной функции
      print('inner:', x)
   inner()
   print('outer:', x)


outer()
```

```
print('global:', x)
```

Здесь соответственно выведется три значения переменной x, каждая функция вывела свою переменную, которая была объявлена внутри ее области видимости.

Давайте теперь внутри функции inner пропишем такую конструкцию как nonlocal x. Что она будет означать? Это означает, что мы будем работать с переменной x, объявленной уровнем выше, т.е. внутри outer.

```
#nonlocal
x=0 #глобальная
def outer():
    x=1 #внутри первой функции
    def inner():
        nonlocal x
        x=2 #внутри вложенной функции
        print('inner:', x)
    inner()
    print('outer:', x)


outer()
print('global:', x)
```

Но так мы можем работать только с локальными переменными, с глобальными это так работать не будет, то есть если мы nonlocal сделаем внутри функции outer у нас вылезет ошибка, что то вроде привязка для нелокального x не обнаружена.

Вот что из себя представляют области видимости переменных в пайтон и вот так можно с ними работать, используя ключевые слова nonlocal и global.

3. Вложенные функции

Рассмотрим пример:

```
def colors():
    def print_red():
        print('red')

    def print_blue():
        print('blue')

    print_red()
    print_blue()

colors()
```

Здесь стоит сразу отметить, что функции print_red() и print_blue() мы объявили внутри другой функции colors() и, следовательно, они будут находиться внутри локальной области этой функции и значит вне пределов области видимости функции colors() вы вызвать print_red() или print_blue() не можете.

Как мы уже знаем внутри любой функции мы можем спокойно создавать переменные. Давайте в нашу функцию print_red() добавим переменную, в которую вынесем то, что у нее в принте и то же самое проделаем с print_blue():

```
def colors():
    def print_red():
        r='red'
        print(r)

    def print_blue():
        b='blue'
```

```
        print(b)

    print_red()
    print_blue()

colors()
```

То есть мы видим, что от этого наша программа не изменилась. Но теперь у нас с вами есть уже две локальные переменные. Локальные они относительно своей функции, то есть r относительно print_red(), a b относительно print_blue и работать мы с ними вне этих функций не можем, но при этом, если в самой функции colors() объявить какую-нибудь переменную, то с ней мы можем работать внутри print_red() и print_blue().

```
def colors():
    y='yellow'
    def print_red():
        r='red'
        print(r,y)

    def print_blue():
        b='blue'
        print(b,y)

    print_red()
    print_blue()

colors()
```

4. Рекурсия

В рекурсии мы имеем ситуацию, внутри которой повторяется вновь данная ситуация. В программировании рекурсия – это когда функция вызывает саму себя. Давайте на примере посмотрим, как это можно сделать.

```
def rec(x):
    print(x)
    rec(x)
rec(1)
```

Как видно при запуске, много раз выведется 1, а в какой-то момент выдается ошибка, что глубина рекурсии была достигнута, то есть в пайтоне есть ограничение на глубину вызовов внутри рекурсии.

Попытаемся примерно оценить сколько раз мы можем вызывать. Для этого в rec добавим к аргументу +1.

```
def rec(x):
    print(x)
    rec(x+1)
rec(1)
```

Как видно после запуска кода функция вызывается около 1000 с небольшим раз и потом выдает ошибку.

Отсюда нам нужно познакомиться с первой важной составляющей рекурсии. У рекурсии должен быть выход, то есть какое-то условие, по которому мы с вами должны перестать вызывать рекуррентную функцию. Давайте в наш код добавим такое условие.

```
def rec(x):
    if x<4:
        print(x)
        rec(x+1)
        print(x)
```

```
rec(1)
```

Теперь давайте приведем пример кода классической задачи на рекурсию, а это нахождение факториала:

```
def fact(x):
    if x==1:
        return 1
    return fact(x-1)*x
print(fact(4))
```