Теоретический материал

Теоретический материал по теме "Функция" и "Рекурсия"

Сайт: Дистанционная подготовка
Курс: Д. П. Кириенко. Программирование на языке Python (школа 179 г. Москвы)
Book: Теоретический материал
Printed by: maung myo
Date: Воскресенье 4 Март 2018, 01:41

Table of Contents

Функции

Ранее была задача вычисления числа сочетаний из n элементов по k, для чего необходимо вычисление факториалов трех величин: n, k и n-k. Для этого можно сделать три цикла, что приводит к увеличению размера программы за счет трехкратного повторения похожего кода. Вместо этого лучше сделать одну *функцию*, вычисляющую факториал любого данного числа n и трижды использовать эту функцию в своей программе. Соответствующая функция может выглядеть так:

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

Этот текст должен идти 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 f. Функция завершается инструкцией return f, которая завершает работу функции и возвращает значение переменной f. 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, также в функциях, не возвращающих значения, инструкция return может отсутствовать.

Теперь мы можем использовать нашу функцию несколько раз. В этом примере мы трижды вызываем функцию factorial для вычисления трех факториалов: factorial(n), factorial(k), factorial(n-k).

```
n = int(input())
k = int(input())
print factorial(n) // (factorial(k) * factorial(n - k))
```

Мы также можем, например, объявить функцию binomial, которая принимает два целочисленных параметра n и k и вычисляет число сочетаний из n по k:

```
def binomial(n, k)
    return factorial(n) // (factorial(k) * factorial(n - k))
```

Тогда в нашей основной программе мы можем вызвать функцию binomial для нахождения числа сочетаний:

```
print binomial(n, k)
```

Вернемся к задаче нахождения наибольшего из двух или трех чисел. Функцию нахождения максимума из двух чисел можно написать так:

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

Теперь мы можем реализовать функцию max3, находящую максимум трех чисел:

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

Функция max3 дважды вызывает функцию max для двух чисел: сначала, чтобы найти максимум из a и b, потом чтобы найти максимум из этой величины и c.

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 выше функции f эта переменная не инициализируется. Но 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 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. То есть несмотря на то, что значение переменной a изменилось внутри функции, то 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). То есть 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, если внутри нее есть хотя бы одна инструкция, модифицирующая значение переменной (это может быть оператор =, += и

т.д., или использование этой переменной в качестве параметра цикла for, то эта переменная считается локальной и не может быть использована до инициализации. 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 функция должна поменять какую-то переменную, то как правило это лучше сделать, как значение, возвращаемое функцией.

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

```
return (a, b)
```

Тогда результат вызова функции тоже нужно присваивать кортежу:

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

Рекурсия

Эпиграф:

```
def ShortStory():
    print("У попа была собака, он ее любил.")
    print("Она съела кусок мяса, он ее убил,")
    print("В землю закопал и надпись написал:")
    ShortStory()
```

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 C++:

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

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 (об этом речь пойдет позже). 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 получиться бесконечная цепочка.

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