

Рекурсия в Python

Информатика. Язык программирования Python в курсе информатики. Все классы.

Функции могут вызывать и другие функции, и даже вызывать сами 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)

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

Как это работает? Допустим, мы вызвали функцию factorial(4). Будет вызвана функция, у которой значение параметра n==4. Она проверит условие n==0, поскольку условие ложно, то будет выполнена инструкция  return n∗factorial(n−1). Но чтобы вычислить это значение, будет вызвана функция factorial(3), т. к. параметр n имеет значение, равное 4. Теперь в памяти будет находиться две функции factorial — одна со значением параметра n==4, а другая — n==3. При этом активна будет последняя функция.

Эта функция в свою очередь вызовет функцию factorial(2), та вызовет функцию factorial(1), затем factorial(0). В случае этой функции ничего более вызвано не будет, функция просто вернет значение 1, и управление вернется в функцию factorial(1). Та умножит значение n==1 на значение 1, которое вернула функция factorial(0), и вернет полученное произведение, равное 1. Управление вернется в функцию factorial(2), которая умножит n==2 на значение 1, которое вернула функция factorial(1) и вернет полученное произведение, равное 2. Функция factorial(3) вернет 3∗2==6, а функция factorial(4)вернет 4∗6==24.

Таблица последовательности, в которой будут вызываться функции, приведена ниже. Значения функции возвращают в порядке, обратном порядке их вызова, то есть сначала заканчивает работу функция factorial(0), затем factorial(1) и т. д.

|  |  |
| --- | --- |
| С какими параметрами вызвана функция | Какое значение вернула |
| factorial(4) | 4 \* 6 == 24 |
| factorial(3) | 3 \* 2 == 6 |
| factorial(2) | 2 \* 1 == 2 |
| factorial(1) | 1 \* 1 == 1 |
| factorial(0) | 1 |

При отладке программы всю последовательность вложенных вызовов рекуррентных функций можно изучить в окне «Call Stack» («стек вызовов») в режиме отладки среды Wing IDE. При этом значения локальных переменных будут отображаться в окне «Stack data», и для каждой вызываемой функции значения локальных переменных будут своими.

Для того, чтобы реализовать рекурсию нужно ответить на следующие вопросы:

1. Какой случай (для какого набора параметров) будет крайним (простым) и что функция возвращает в этом случае?
2. Как свести задачу для какого-то набора параметров (за исключением крайнего случая) к задаче, для другого набора параметров (как правило, с меньшими значениями)?

При этом программирование рекурсии выглядит так. Функция должна сначала проверить, не является ли переданный набор параметров простым (крайним) случаем. В этом случае функция должна вернуть значение (или выполнить действия), соответствующие простому случаю. Иначе функция должна вызвать себя рекурсивно для другого набора параметров, и на основе полученных значений вычислить значение, которое она должна вернуть.

Рассмотрим несколько примеров (на самом деле эти примеры являются «учебными» и все перечисленные задачи гораздо лучше решать при помощи циклов, а не рекурсией).

ВЫЧИСЛЕНИЕ СУММЫ НАТУРАЛЬНЫХ ЧИСЕЛ ОТ 1 ДО N

Если n==1, то сумма равна 1. Иначе сумма чисел от 1 до n равна сумме чисел от 1 до n−1, которую можно вычислить при помощи рекурсии плюс число n.

def sum(n):  
    if n == 1:  
        return 1  
    else:  
        return n + sum(n — 1)

ПРОВЕРКА СТРОКИ НА ПАЛИНДРОМНОСТЬ

Строка является палиндромом, если она одинаково читается как справа налево, так и слева направо. Напишем функцию IsPalindrome, которая возвращает значение типа bool в зависимости от того, является ли строка палиндромом.

Крайнее значение — пустая строка или строка из одного символа всегда палиндром. Рекурсивный переход — строка является палиндромом, если у нее совпадают первый и последний символ, а также строка, полученная удалением первого и последнего символа является палиндромом.

def IsPalindrome(S):  
    if len(S) <= 1:  
        return True  
    else:  
        return S[0] == S[-1] and IsPalindrome(S[1:-1])

СУММИРОВАНИЕ СПИСКА

Дан список чисел, необходимо просуммировать его.

Крайний случай — пустой список, сумма чисел в нем равна 0. Иначе нужно вычислить сумму чисел в срезе списка без одного элемента и добавить значение этого элемента.

def Sum(A):  
    if len(A) == 0:  
        return 0  
    else:  
        return Sum(A[:-1]) + A[-1]

НАИБОЛЬШЕЕ ЗНАЧЕНИЕ В СПИСКЕ

Дан список чисел, необходимо найти наибольшее значение в нем.

Крайний случай — список из одного элемента, наибольшее значение в нем равно единственному элементу. Иначе нужно вычислить наибольшее значение в срезе списка без одного элемента и взять максимум из этого числа и значения этого элемента.

def Max(A):  
    if len(A) == 1:  
        return A[0]  
    else:  
        return max(Max(A[:-1]), A[-1])

ЧИСЛА ФИБОНАЧЧИ

Последовательность Фибоначчи задана так: F0=0, F1=1, при n>1 число Фибоначчи с номером n вычисляется как Fn=Fn−1+Fn−2. Для рекурсивного вычисления чисел Фибоначчи достаточно аккуратно запрограммировать эти соотношения:

def Fib(n):  
    if n <= 1:  
        return n  
    else:  
        return Fib(n — 1) + Fib(n — 2)

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.

БЫСТРОЕ ВОЗВЕДЕНИЕ В СТЕПЕНЬ

Одним из полезных применений рекурсии является алгоритм быстрого возведения в степень. Если вычислять степень числа anпри помощи простого цикла, то понадобится n−1 умножение. Но можно использовать рекуррентные соотношения:

an=an−1×a, при нечетном n

an=(an/2)2, при четном n.

Это позволяет записать алгоритм, который будет выполнять не более, чем 2∗log2n умножений:

def power(a, n):  
    if n == 0:  
        return 1  
    elif n % 2 == 1:  
        return power(a, n - 1) \* a  
    else:  
        return power(a, n // 2) \*\* 2

ХАНОЙСКИЕ БАШНИ

Другой классической задачей, решаемой при помощи рекурсии, является задача о Ханойских башнях. Головоломка “Ханойские башни” состоит из трех стержней, пронумерованных числами 1, 2, 3. На стержень 1 надета пирамидка из n дисков различного диаметра в порядке возрастания диаметра. Диски можно перекладывать с одного стержня на другой строго по одному, при этом диск нельзя класть на диск меньшего диаметра. Необходимо переложить всю пирамидку со стержня 1 на стержень 3 <b>за минимальное число перекладываний</b>.

Необходимо написать программу, которая для данного числа дисков n печатает последовательность перекладываний, необходимую для решения головоломки.

Сначала нужно подумать, как переложить пирамидку из n дисков с одного стержня на другой. Для этого нужно прежде всего перенести самый большой диск. Но чтобы перенести этот диск нужно всю пирамидку без этого диска, то есть пирамидку из n−1 диска перенести на третий стержень, затем перенести один самый большой диск, затем перенести пирамидку из n−1диска на тот стержень, на который переместили самый большой диск.

Напишем рекурсивную функцию   move(n, start, finish), которая печатает последовательность перекладываний, необходимых для перемещения пирамидки из n дисков со стержня номер start на стержень номер finish. Простой случай — n==1, в этом случае рекурсия не нужна и нужно просто переместить один диск.

def move(n, start, finish):  
    if n == 1:  
        print("Перенести диск 1 со стержня", start, "на стержень", finish)  
    else:  
        temp = 6 — start — finish # Вспомогательный колышек  
        move(n — 1, start, temp)  
        print("Перенести диск", n, "со стержня", start, "на стержень", finish)  
        move(n — 1, temp, finish)  
# Для решения головоломки из 10 дисков вызываем так:  
move(10, 1, 3)

Решение можно сделать проще, если понять, что крайний случай — это случай n==0, в этом случае для перемещения пирамидки из 0 дисков… просто ничего не нужно делать!

def move(n, start, finish):  
    if n > 0:  
        temp = 6 — start — finish # Вспомогательный колышек  
        move(n — 1, start, temp)  
        print("Перенести диск", n, "со стержня", start, "на стержень", finish)  
        move(n — 1, temp, finish)

ОГРАНИЧЕНИЕ НА ГЛУБИНУ РЕКУРСИИ

По умолчанию глубина рекурсии в языке Питон ограничена 1000 вызовов. Это ограничение можно поднять при помощи функции  setrecursionlimit из модуля system. Например, чтобы увеличить возможную глубину рекурсии до 10000 нужно в начале программы выполнить две инструкции:

import sys  
sys.setrecursionlimit(10000)

Следует учитывать, что глубину рекурсии нельзя увеличивать до очень большого значения, помимо ограничения, который устанавливается при помощи setrecursionlimit есть и ограничения операционной системы.



Прямой и обратный ход рекурсии

Информатика. Язык программирования Python в курсе информатики. Все классы.

*Чтобы понять рекурсию, нужно понять рекурсию*.

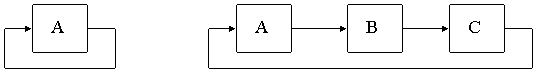
В программировании **рекурсия** — вызов функции (процедуры) из неё же самой, непосредственно (простая рекурсия) или через другие функции (сложная или косвенная рекурсия), например, функция A вызывает функцию B, а функция B — функцию A.

Количество вложенных вызовов функции или процедуры называется *глубиной* рекурсии.

Преимущество рекурсивного определения объекта заключается в том, что такое конечное определение теоретически способно описывать бесконечно большое число объектов. С помощью рекурсивной программы же возможно описать бесконечное вычисление, причём без явных повторений частей программы.

Программист разрабатывает программу, сводя исходную задачу к более простым. Среди этих задач может оказаться и первоначальная, но в упрощенной форме.

Например, для вычисления F(N) может понадобиться вычислить F(N−1). Иными словами, частью алгоритма вычисления функции будет вычисление этой же функции.



Пример рекурсивного алгоритма: N!=(N−1)!∗N, если N=0, то N!=1

Любое рекурсивное определение состоит из двух частей. Одна часть определяет понятие через него же, другая часть – через иные понятия.

Процедура является **рекурсивной**, если она обращается сама к себе прямо или косвенно (через другие процедуры).

Заметим, что при косвенном обращении все процедуры в цепочке – рекурсивные. Все сказанное о процедурах целиком относится и к функциям.

ПРИМЕР РЕКУРСИВНОЙ ФУНКЦИИ ВЫЧИСЛЕНИЯ ФАКТОРИАЛА НА PASCAL

function factorial(N: integer) : longint;  
begin  
   if N = 0 then  
       factorial := 1  
   else   
       factorial := factorial(N-1) \* N   
end;

ПРИМЕР РЕКУРСИВНОЙ ФУНКЦИИ ВЫЧИСЛЕНИЯ ФАКТОРИАЛА НА КУМИРЕ

алг  
  нач  
  цел n, ans  
  ввод n  
  f(n, ans)  
  вывод ans  
кон  
алг f (цел n, рез цел t)  
  нач  
  если n = 1 то  
    t := 1  
  иначе  
    f(n - 1, t)  
    t := t \* n  
  все  
кон

Рекурсия изнутри

Это может показаться удивительным, но самовызов функции/процедуры ничем не отличается от вызова другой функции/процедуры. Что происходит, если одна функция вызывает другую? В общих чертах следующее:

* в памяти размещаются параметры, передаваемые функции (но не параметры-переменные, т. к. они передаются по ссылке!);
* для внутренних переменных вызываемой функции также отводится новая область памяти (несмотря на совпадение их имен и типов с переменными вызывающей функции);
* запоминается адрес возврата в вызывающую функцию;
* управление передается вызванной функции.

Если функцию или процедуру вызвать повторно из другой функции/процедуры или из нее самой, будет выполняться тот же код, но работать он будет *с другими значениями параметров и внутренних переменных*. Это и дает возможность рекурсии.

Прямой и обратный ход рекурсии

Действия, выполняемые функцией до входа на следующий уровень рекурсии, называются выполняющимися **на прямом ходу рекурсии**, а действия, выполняемые по возврату с более глубокого уровня к текущему, – выполняющимися **на обратном ходу рекурсии**.

ДЕМОНСТРАЦИЯ ХОДА РЕКУРСИИ НА PASCAL

var glubina: Integer := 0;  
function factorial(N: integer) : longint;  
begin  
   {при входе в функцию увеличиваем глобальную переменную, которая считает глубину рекурсии}  
   glubina := glubina + 1;  
   Writeln('Прямой ход рекурсии. Глубина = ', glubina);  
   Result := 1;  
   if N > 0 then Result := factorial(N-1) \* N;  
   Writeln('Обратный ход рекурсии. Глубина = ', glubina);  
   {при входе из функции - уменьшаем эту глобальную переменную}  
   glubina := glubina - 1;  
end;  
begin  
  factorial(3);  
end.

При запуске этой программы, будет выведено:

Прямой ход рекурсии. Глубина = 1  
Прямой ход рекурсии. Глубина = 2  
Прямой ход рекурсии. Глубина = 3  
Прямой ход рекурсии. Глубина = 4  
Обратный ход рекурсии. Глубина = 4  
Обратный ход рекурсии. Глубина = 3  
Обратный ход рекурсии. Глубина = 2  
О

Алгоритм Евклида: Python

Информатика. Язык программирования Python в курсе информатики. Все классы.

Рассмотрим задачу нахождения наибольшего общего делителя двух натуральных чисел. Постановка задачи — даны два натуральных числа a и b, необходимо найти такое наибольшее число d, которое является делителем каждого из этих чисел.

Задачу можно решать разными способами. Наивный алгоритм — будем перебирать все числа от 1 до min(a,b), будем проверять делимость чисел a и b на проверяемое число, выберем наибольшее число, на которое делятся и a, и b. Такой алгоритм будет имеет сложность O(min(a,b)).

Можно оптимизировать перебор, например, перебирать не все числа до min(a,b), а только делители одного из чисел (a или b). Это можно организовать перебирая числа до min(a,b), если использовать идеи, аналогичные оптимизации алгоритма проверки числа на простоту, то есть алгоритм будет иметь сложность O(min(a,b)).

Гораздо более эффективно можно решить эту же задачу, если использовать **алгоритм Евклида**. Он основан на следующем свойстве (обозначим наибольший общий делитель чисел a и b как НОД(a, b)):

НОД(a, b) = НОД(a - b, b)

Свойство легко доказать, если заметить, что любой общий делитель чисел a и b также является общим делителем чисел a - b и b, и наоборот, т. е. для пар чисел (a, b) и (a - b, b) совпадают множества общих делителей, и, значит, и наибольший общий делитель.

Это позволяет написать рекурсивный алгоритм (название **gcd** есть аббревиатура от *Greatest Common Divisor*):

def gcd(a, b):  
    if a == 0 or b == 0:   
         return max(a, b)  
else:  
    if a > b:  
        return gcd(a - b, b)  
    else:  
        return gcd(a, b - a)

Этот алгоритм использует указанные рекуррентные соотношения до тех пор, пока одно из чисел не станет равно 0 (иногда используют другое условие окончания алгоритма — числа a и b станут равны), в этом случае возвращается значение другого числа.

Этот же алгоритм можно записать при помощи цикла while, без рекурсии:

def gcd(a, b):   
    while a != 0 and b != 0:  
        if a > b:   
            a = a - b  
        else:   
            b = b — a   
    return max(a, b)

Записанный в таком виде алгоритм неэффективен, так как многократно использует вычитания. Например, если взять a=10∗∗9+2, b=2, то алгоритм выполнит 500 миллионов вычитаний для получения вполне очевидного результата. Чтобы оптимизировать алгоритм необходимо понять, что многократное вычитание из большего числа меньшего закончится на числе, которое является остатком от деления двух первоначальных чисел. То есть в каждом из этих алгоритмов можно заменить операцию вычитания на взятие остатка:

def gcd(a, b):  
    if a == 0 or b == 0:   
        return max(a, b)  
    else:  
        if a > b:   
            return gcd(a % b, b)  
        else:  
            return gcd(a, b % a)

  или

def gcd(a, b):   
    while a != 0 and b != 0:  
        if a > b:  
            a = a % b   
        else:  
            b = b % a   
    return max(a, b)

Но алгоритм Евклида можно записать и еще проще. Давайте будем считать, что a⩾b. Тогда после замены a=a%b результат этой операции станет меньше чем b, то есть теперь a<b. Чтобы вернуть свойство a⩾b нужно поменять a и b местами, что удобно делать при помощи кортежей:

def gcd(a, b):   
    if b == 0:   
        return a  
    else:  
        return gcd(b, a % b)

  или

def gcd(a, b):   
    while b != 0:   
        a, b = b, a % b   
    return a

Таким образом, в этом алгоритме поддерживается два инварианта: a⩾b и НОДНОД(a,b) не меняется. Несложно заметить, что алгоритм будет корректно работать и в случае, когда a<b, тогда на первом же шаге значения a и b будут просто поменяны местами.

Можно показать, что наибольшее количество операций алгоритм Евклида будет выполнять, когда на вход ему даны два числа, являющиеся соседними членами ряда Фибоначчи, то есть ряда 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181 и т. д. Известно, что n-й член последовательности Фибоначчи можно вычислить по формуле

Fn=(1+52)n−(1−52)n5, то есть Fn≈15(1+52)n . Таким образом, сложность алгоритма Евклида: O(log⁡min(a,b)).

братный ход рекурсии. Глубина = 1