**Аннотация**

*Сегодня мы познакомимся с понятием рекурсии, покажем ее связь с уже известными нам конструкциями (циклами и функциями). Разберем наиболее часто встречающиеся ошибки и классические примеры.*

**Факториал и число сочетаний**

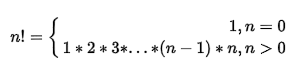
Задача на сочетания — простейшая комбинаторная задача на сочетания без повторений: сколькими способами можно из данных n предметов выбрать некоторые k предметов, если порядок их выбора не важен?

Ответом на эту задачу является величина:



называемая числом сочетаний из n элементов по k.

Запись n! обозначает произведение 1 · 2 · 3 · ... · n, называемое факториалом числа n (мы уже неоднократно сталкивались с данным понятием), при этом считается, что 0! = 1. Приведем «красивое» математическое определение факториала:



Давайте напишем функцию, вычисляющую факториал числа n классическим способом, и проверим ее работу.

# Вычисление факториала

def factorial(number):

result = 1

for index in range(2, number + 1):

result \*= index

return result

for i in range(10):

print(i, factorial(i))

Обратите внимание на красивый способ вычисления факториала в стиле языка Python:

from functools import reduce

def cool\_factorial(number):

return reduce(lambda x, y: x \* y, range(2, number + 1), 1)

for i in range(10):

print(i, cool\_factorial(i))

**Определение рекурсии, принцип работы**

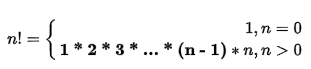
Однако задачу вычисления факториала можно решить иначе. В математике очень часто для упрощения вычислений исходную задачу сводят к более простым.

**Рекурсия**

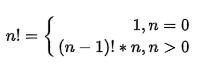
В итоге можно прийти к тому, что будет вызвана первоначальная задача, но в несколько упрощенной форме. Такой прием называется рекурсией (от лат. recurcio — «возвращение»).

Итак, **рекурсия** в программировании — прием, когда функция может вызывать сама себя прямо либо косвенно (через другую функцию, при этом обе функции являются рекурсивными).

Давайте еще раз посмотрим на определение факториала и обратим внимание на выделенный жирным фрагмент:



Можно увидеть, что **1 · 2 · 3 · ... · (n − 1)**, не что иное, как факториал числа n − 1. Поэтому определение факториала можно записать в сокращенном виде:



Вернемся к нашей задаче и рассмотрим функцию вычисления факториала с несколько другой стороны, постараемся применить рекурсию. Известно, что 0! = 1, 1! = 1. А как вычислить величину n! для большого n? Если бы мы могли вычислить величину (n − 1)!, тогда мы легко вычислим n!, поскольку n! = n · (n − 1)!. Но как вычислить (n − 1)!? Если бы мы вычислили (n − 2)!, мы сможем вычислить 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.

Давайте напишем соответствующую функцию:

# Рекурсивное вычисление факториала

def rec\_factorial(number):

if number == 0:

return 1

else:

return number \* rec\_factorial(number - 1)

Логическая сложность рекурсивных функций заключается в изменении параметров и особенностях получения промежуточных результатов при последовательном обращении подпрограммы к себе. Выполняется две серии шагов. Первая серия — шаги рекурсивного погружения подпрограмм в себя до тех пор, пока выбранный параметр не достигнет граничного значения (**глубина рекурсии**). Вторая серия — шаги рекурсивного выхода до тех пор, пока значение выбранного параметра не достигнет начального. Она, как правило, и обеспечивает получение промежуточных и конечных результатов.

Вот так работает рекурсивная функция вычисления факториала:

В общем случае рекурсия тяготеет к [декларативному](https://ru.wikipedia.org/wiki/%D0%94%D0%B5%D0%BA%D0%BB%D0%B0%D1%80%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) стилю программированию. Если в двух словах: когда мы пишем императивную функцию (как делали все время до этого), отвечаем на вопрос, **как** достигнуть необходимого результата, а когда создаем декларативную — на вопрос, **что** такое наш результат.

Поэтому в **любой** рекурсивной функции должно быть как минимум **две** ветки развития «сюжета»:

1. Основная.
2. Точка выхода.

Сама функция при этом получается **декларативной**: она повторяет практически один в один определение факториала.

**Опасность использования рекурсивных алгоритмов или что может пойти не так**

**Важно!**

Наиболее распространенной ошибкой при использовании рекурсии является бесконечная рекурсия, когда цепочка вызовов функций никогда не завершается и продолжается, пока не кончится свободная память в компьютере.

Определим две наиболее распространенные причины для бесконечной рекурсии на примере некорректно написанной функции нахождения факториала числа.

def rec\_factorial(number):

return number \* rec\_factorial(number - 1)

def rec\_factorial(number):

if number == 0:

return 1

else:

return number \* rec\_factorial(number)

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

**Важно!**

Еще одна проблема, связанная с использованием рекурсивных функций, — нетривиальность задачи оценки сложности и эффективности алгоритма. Сложность этих алгоритмов зависит не только от сложности внутренних циклов, но и от количества итераций рекурсии. Рекурсивная процедура может выглядеть достаточно простой, но она может серьезно усложнить программу, многократно вызывая себя.

**Красота требует жертв?**

Напишем функцию перевода числа из десятичной системы счисления в двоичную (а на самом деле и в любую другую позиционную систему).

Для начала вспомним базовый алгоритм перевода:

**Шаг 1.** Разделить число на основание системы счисления, в которую осуществляется перевод (в нашем случае — два). Записать остаток.

**Шаг 2.** Если результат деления больше двух или равен двум, продолжать делить его на два до тех пор, пока результат деления не станет равен единице.

**Шаг 3.** Выписать результат последнего деления и все остатки от деления в обратном порядке в одну строку.

Рассмотрим на примере перевода числа 136 в двоичную систему счисления:

13610 = 100010002

Декларативное описание этой функции звучит так:

1. Если число a больше единицы, напечатаем перевод в двоичную систему числа, равного целой части от деления a на два.
2. В противном случае напечатаем остаток от деления a на два.

С помощью рекурсии реализовать этот алгоритм можно очень красивым и лаконичным кодом.

def bin(a):

if a > 1:

bin(a // 2)

print(a % 2, end="")

Теперь напишем функцию, которая вычисляет НОД (наибольший общий делитель) пары чисел А и B.

Определение:

1. Если число b равно 0, вернем число a.
2. В противном случае вернем значение функции (рекурсия) от числа b и остатка от деления числа a на число b.

def gcd(a, b):

if b == 0:

return a

else:

return gcd(b, a % b)

print(gcd(16, 24))

И последний пример: функция, которая удаляет из строки s все вхождения символа e.

def del\_all\_e(s, e):

if not s:

return s

elif e == s[0]:

return del\_all\_e(s[1:], e)

else:

return s[0] + del\_all\_e(s[1:], e)

print(del\_all\_e("мама мыла раму", "а"))

**Несколько рекурсивных веток. Деревья**

Рассмотрим в качестве примера функцию, вычисляющую числа Фибоначчи.

Числа Фибоначчи — ряд чисел: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89..., в котором два первых элемента равны 1, а каждый следующий — сумме двух предыдущих. Удивительно, что отношение двух соседних чисел Фибоначчи стремится к числу золотого сечения: 1,6180339887.

Составим рекурсивное определение этих чисел и сразу запишем его в виде функции:

def rec\_fib(n):

if 0 < n <= 2:

return 1

else:

return rec\_fib(n - 1) + rec\_fib(n - 2)

Это удивительно, но программа почти слово в слово совпадает с определением чисел Фибоначчи!

Однако в этом примере мы столкнулись с новым типом рекурсии, в котором функция порождает целых **две** рекурсивные ветки. Неявно во время выполнения программы мы обходим дерево в глубину. Проиллюстрируем это на примере fib(6):

Важно понимать, что экземпляры функции выполняются не параллельно, а в детерминированной (то есть определенной) последовательности: сначала левое поддерево, а потом все правое поддерево из любой вершины.

Интересно и то, что дерево очень быстро разрастается при росте номера числа Фибоначчи, что влечет замедление программы и трату памяти.

Кстати, следующее число Фибоначчи вычисляется ровно в золотое сечение раз медленнее, чем предыдущее. Таким образом, fib(500) будет получено уже после того, как исчезнет Солнечная система.

Следующий пример демонстрирует этот факт. Запустите его и убедитесь:

from time import time

for i in range(30, 35):

s = time()

print(i, rec\_fib(i), "%.03f" % (time() - s))

А теперь запустите простой императивный вариант:

from time import time

def fib(n):

a, b = 1, 1

for \_ in range(n - 2):

a, b = b, a + b

return b

for i in range(30, 35):

s = time()

print(i, fib(i), "%.03f" % (time() - s))

Второй вариант работает существенно быстрее.

А все потому, что в рекурсивном случае мы много раз вычисляем одно и то же число Фибоначчи. Никакого **кеширования** (запоминания предыдущих вычислений) не происходит. Кстати, оптимизации типа кеширования присутствуют по умолчанию в функциональных языках (LISP, Haskell). В Python включать такой функционал надо вручную.

В следующем примере запоминаются последние 1000 вызовов функции fib.

from time import time

from functools import lru\_cache

@lru\_cache(maxsize=1000)

def rec\_fib(n):

if 0 < n <= 2:

return 1

else:

return rec\_fib(n - 1) + rec\_fib(n - 2)

for i in range(30, 35):

s = time()

print(i, rec\_fib(i), "%.03f" % (time() - s))

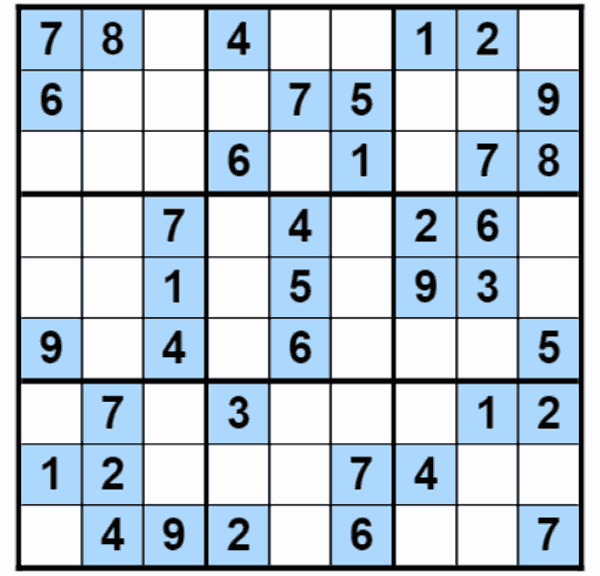
Ну вот, теперь все работает быстро.

Перейдем к главному выводу.

**Важно!**

Рекурсивный метод обеспечивает удобный обход списка, дерева или графа, при этом контролируя перемещение по элементам и возвращение к предыдущим состояниям. Этим можно пользоваться во многих математических и прикладных задачах.

**Бонус. Решаем судоку**



Вы уже могли сталкиваться с этой задачей раньше. Вспомните, как мы ее решали? Попробуем теперь предложить иной способ.

Предположим, что нам нужно сделать программу, которая разгадывает судоку. Пусть поле моделируется списком списков с целыми числами:

field = [

[0,0,0,0,0,0,0,0,0],

[0,1,0,0,2,0,0,3,0],

[0,0,0,0,0,0,0,0,0],

[0,0,0,0,0,0,0,0,0],

[0,4,0,0,5,0,0,6,0],

[0,0,0,0,0,0,0,0,0],

[0,0,0,0,0,0,0,0,0],

[0,7,0,0,8,0,0,9,0],

[0,0,0,0,0,0,0,0,0],

]

Сформулируем рекурсивный алгоритм решения судоку.

* Если на поле судоку нет пустых клеток, оно уже решено и надо просто вернуть поле в качестве решения
* Если есть пустые клетки, надо вычислить какую-либо пустую клетку, для которой количество возможных вариантов минимально. Попробовать по очереди проверять эти варианты, и, если будет найдено решение, вернуть его

Сама функция будет не сильно больше данного описания:

from pprint import pprint

from copy import deepcopy

from random import shuffle

from time import clock

"""

Для всех клеток на основе ограничений

возвращает список возможных чисел, например:

(0, 0, {2, 3, 4, 5, 6, 7, 8, 9})

(0, 1, {2, 3, 5, 6, 8, 9})

(0, 2, {2, 3, 4, 5, 6, 7, 8, 9})

(0, 3, {1, 3, 4, 5, 6, 7, 8, 9})

(0, 4, {1, 3, 4, 6, 7, 9})

(0, 5, {1, 3, 4, 5, 6, 7, 8, 9})

(0, 6, {1, 2, 4, 5, 6, 7, 8, 9})

(0, 7, {1, 2, 4, 5, 7, 8})

(0, 8, {1, 2, 4, 5, 6, 7, 8, 9})

(1, 0, {4, 5, 6, 7, 8, 9})

"""

def get\_variants(sudoku):

variants = []

for i, row in enumerate(sudoku):

for j, value in enumerate(row):

if not value:

# значения в строке

row\_values = set(row)

# значения в столбце

column\_values = set([sudoku[k][j] for k in range(9)])

# в каком квадрате 3x3 находится клетка?

# Координаты этого квадрата

sq\_y = i // 3

sq\_x = j // 3

square3x3\_values = set([

sudoku[m][n]

for m in range(sq\_y \* 3, sq\_y \* 3 + 3)

for n in range(sq\_x \* 3, sq\_x \* 3 + 3)

])

exists = row\_values | column\_values | square3x3\_values

# какие значения остались?

values = set(range(1, 10)) - exists

variants.append((i, j, values))

return variants

def solve(sudoku):

# Если судоку заполнено, это ответ

if all([k for row in sudoku for k in row]):

return sudoku

# Иначе посмотрим все варианты

variants = get\_variants(sudoku)

# Выберем тот, у которого меньше всего возможностей.

x, y, values = min(variants, key=lambda x: len(x[2]))

# Попробуем все по очереди

for v in values:

# deepcopy создает полную копию списка с учетом всех вложенностей

new\_sudoku = deepcopy(sudoku)

new\_sudoku[x][y] = v

# Если оно решилось, возвратим ответ.

s = solve(new\_sudoku)

if s:

return s

return None

s = clock()

pprint(

solve([

[0, 0, 0, 0, 0, 0, 0, 0, 0],

[0, 1, 0, 0, 2, 0, 0, 3, 0],

[0, 0, 0, 0, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 0, 0, 0, 0, 0],

[0, 4, 0, 0, 5, 0, 0, 6, 0],

[0, 0, 0, 0, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 0, 0, 0, 0, 0],

[0, 7, 0, 0, 8, 0, 0, 9, 0],

[0, 0, 0, 0, 0, 0, 0, 0, 0],

]))

print('Затраченное время:', clock() - s, 'сек')

А вот еще один пример рекурсивного решения задачи. Он гораздо короче, но в нем есть **питоновские** штуки.

from random import shuffle

from copy import deepcopy

from pprint import pprint

def make\_assumptions(sudoku):

for i, row in enumerate(sudoku):

for j, value in enumerate(row):

if not value:

values = set(row) \

| set([sudoku[k][j] for k in range(9)]) \

| set([sudoku[m][n] for m in range((i // 3) \* 3, (i // 3) \* 3 + 3)

for n in range((j // 3) \* 3, (j // 3) \* 3 + 3)])

yield i, j, list(set(range(1, 10)) - values)

def solve(sudoku):

if all([k for row in sudoku for k in row]):

return sudoku

assumptions = list(make\_assumptions(sudoku))

shuffle(assumptions)

x, y, values = min(assumptions, key=lambda x: len(x[2]))

for v in values:

new\_sudoku = deepcopy(sudoku)

new\_sudoku[x][y] = v

s = solve(new\_sudoku)

if s:

return s

return None

pprint(solve(field))

Два предыдущих примера демонстрируют преимущество рекурсии — написание коротких и легко читаемых программ.

Попробуйте сравнить эти программы с императивным вариантом (без использования рекурсии).

С рекурсией вы еще встретитесь много раз. Помните: это **не панацея**, но позволяет элегантно и эффективно решать широкий круг задач.

А пока — все, переходите к задачам!