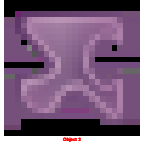
Лекция 7



Лекция 7

Рекурсия в Python

Рекомендуемая литература:

1. Functional Programming with Python, Pramode C.E., Linux Gazette, 2004

2. David Mertz. Functional Programming in Python. — O'Reilly, 2015. — ISBN 978-1-491-92856-1.

3. Steven Lott. Functional Python Programming. — Packt Publishing Ltd, 2015. — 360 p. — ISBN 978-1-78439-761-6.

4. https://www.linuxjournal.com/content/book-excerpt-chapter-6-functions-and-functional-programming

5. Плас Дж.В. Python для сложных задач. Наука о данных и машинное обучение/ - СПб.: Питер, 2018. — 576 с. https://www.twirpx.com/file/2353850/

Основы рекурсии в Python

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 Питоне:

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. 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 получится бесконечная цепочка.

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

ЗАДАЧА: "Напишите рекурсивную функцию" listSum ", которая принимает список целых чисел и возвращает сумму всех целых чисел в списке".

Пример:

```
>>>> listSum([1,3,4,5,6])
19
```

Посмотрим, как это сделать другим способом, но не рекурсивным способом.

```
def listSum(ls):
    i = 0
    s = 0
    while i < len(ls):
        s = s + ls[i]
        i = i + 1
    print s
```

Нужен базовый способ сделать это, поскольку специальные встроенные функции не разрешены.

Всякий раз, когда вы сталкиваетесь с такой проблемой, попробуйте выразить результат функции с той же функцией.

В вашем случае вы можете получить результат, добавив первый номер с результатом вызова той же функции с остальными элементами в списке.

Например,

```
listSum([1, 3, 4, 5, 6]) = 1 + listSum([3, 4, 5, 6])
                         = 1 + (3 + listSum([4, 5, 6]))
                         = 1 + (3 + (4 + listSum([5, 6])))
                         = 1 + (3 + (4 + (5 + listSum([6]))))
                         = 1 + (3 + (4 + (5 + (6 + listSum([])))))
```

Теперь, что должно быть результатом `listSum([])`? Это должно быть 0. Это называется базовым условием вашей рекурсии. Когда выполняется базовое условие, рекурсия подходит к концу. Теперь попробуйте реализовать его.

Главное здесь, разбивая список. Вы можете использовать slicing для этого.

Простая версия

```
>>> def listSum(ls):
...     # Base condition
...     if not ls:
...         return 0
...
...     # First element + result of calling `listsum` with rest of
the elements
...     return ls[0] + listSum(ls[1:])
>>>
>>> listSum([1, 3, 4, 5, 6])
19
```

Рекурсия хвостового вызова

Как только вы поймете, как работает вышеупомянутая рекурсия, вы можете попытаться сделать ее немного лучше. Теперь, чтобы найти фактический результат, мы также зависим от значения предыдущей функции. Оператор `return` не может сразу вернуть значение до тех пор, пока рекурсивный вызов не вернет результат. Мы можем избежать этого, передавая ток в параметр функции, как этот

```
>>> def listSum(ls, result):
...     if not ls:
...         return result
...     return listSum(ls[1:], result + ls[0])
...
>>> listSum([1, 3, 4, 5, 6], 0)
19
```

Здесь мы передаем начальное значение суммы в параметрах, которое равно нулю в `listSum([1, 3, 4, 5, 6], 0)`. Затем, когда выполняется базовое условие, мы фактически накапливаем сумму в параметре `result`, поэтому возвращаем его. Теперь последний оператор `return` имеет `listSum(ls[1:], result + ls[0])`, где мы добавляем первый элемент в текущий `result` и передаем его снова на рекурсивный вызов.

Это может быть подходящее время для понимания Tail Call. Это не имело бы отношения к Python, так как оно не оптимизировало Tail call.

Прохождение версии индекса

Теперь вы можете подумать, что мы создаем так много промежуточных списков. Можно ли этого избежать?

Конечно, вы можете. Вам просто нужен индекс элемента для последующей обработки. Но теперь базовое условие будет другим. Поскольку мы собираемся передавать индекс, как мы определяем, как весь список был обработан? Ну, если индекс равен длине списка, то мы обработали все элементы в нем.

```
>>> def listSum(ls, index, result):
...     # Base condition
...     if index == len(ls):
...         return result
...
...     # Call with next index and add the current element to result
...     return listSum(ls, index + 1, result + ls[index])
...
>>> listSum([1, 3, 4, 5, 6], 0, 0)
19
```

Версия внутренней функции

Если вы посмотрите на определение функции сейчас, вы передаете ему три параметра. Допустим, вы собираетесь выпустить эту функцию как API. Будет ли удобнее для пользователей передавать три значения, когда они действительно находят сумму списка?

Нет. Что мы можем с этим поделать? Мы можем создать еще одну функцию, которая является локальной для фактической функции `listSum`, и мы можем передать ей все связанные с реализацией параметры, такие как

```
>>> def listSum(ls):
...
...     def recursion(index, result):
...         if index == len(ls):
...             return result
...         return recursion(index + 1, result + ls[index])
...
...     return recursion(0, 0)
...
>>> listSum([1, 3, 4, 5, 6])
19
```

Теперь, когда вызывается `listSum`, он просто возвращает возвращаемое значение внутренней функции `recursion`, которая принимает параметры `index` и `result`. Теперь мы передаем только эти значения, а не пользователи `listSum`. Им просто нужно передать список, который нужно обработать.

В этом случае, если вы наблюдаете параметры, мы не передаем `ls` в `recursion`, но мы используем его внутри него. `ls` доступен внутри `recursion` из-за свойства закрытия.

Версия параметров по умолчанию

Теперь, если вы хотите сохранить его простым, не создавая внутренней функции, вы можете использовать параметры по умолчанию, например этот

```
>>> def listSum(ls, index=0, result=0):
...     # Base condition
...     if index == len(ls):
...         return result
...
...     # Call with next index and add the current element to result
...     return listSum(ls, index + 1, result + ls[index])
...
>>> listSum([1, 3, 4, 5, 6])
19
```

Теперь, если вызывающий объект явно не передает какое-либо значение, тогда `0` будет присвоен как `index`, так и `result`.

Рекурсивная проблема питания

Теперь давайте применим идеи к другой проблеме. Например, попробуйте реализовать функцию `power(base, exponent)`. Он вернет значение `base`, поднятое до мощности `exponent`.

```
power(2, 5) = 32
power(5, 2) = 25
power(3, 4) = 81
```

Теперь, как мы можем сделать это рекурсивно? Попробуем понять, как эти результаты достигнуты.

```
power(2, 5) = 2 * 2 * 2 * 2 * 2 = 32
power(5, 2) = 5 * 5             = 25
power(3, 4) = 3 * 3 * 3 * 3     = 81
```

Хммм, так мы поняли. Результат `base`, умноженный на себя, `exponent` дает результат. Хорошо, как мы к этому подходим. Попробуем определить решение с той же функцией.

```
power(2, 5) = 2 * power(2, 4)
            = 2 * (2 * power(2, 3))
            = 2 * (2 * (2 * power(2, 2)))
            = 2 * (2 * (2 * (2 * power(2, 1))))
```

Каким должен быть результат, если что-то поднято до власти 1? Результат будет такого же числа, не так ли? Мы получили базовое условие для нашей рекурсии: -)

```
            = 2 * (2 * (2 * (2 * 2)))
            = 2 * (2 * (2 * 4))
            = 2 * (2 * 8)
            = 2 * 16
            = 32
```

Хорошо, давайте его реализовать.

```
>>> def power(base, exponent):
...     # Base condition, if `exponent` is lesser than or equal to 1,
return `base`
...     if exponent <= 1:
...         return base
...
...     return base * power(base, exponent - 1)
...
>>> power(2, 5)
32
>>> power(5, 2)
25
>>> power(3, 4)
81
```

Хорошо, как будет определяться оптимизированная версия Tail call? Позволяет передать текущий результат в качестве параметра самой функции и вернуть результат при выполнении базового условия. Пусть это упрощает и напрямую использует подход по умолчанию.

```
>>> def power(base, exponent, result=1):
...     # Since we start with `1`, base condition would be exponent
reaching 0
...     if exponent <= 0:
...         return result
...
...     return power(base, exponent - 1, result * base)
...
>>> power(2, 5)
32
>>> power(5, 2)
25
>>> power(3, 4)
81
```

Теперь мы уменьшаем значение `exponent` в каждом рекурсивном вызове и несколько `result` с `base` и передаем его рекурсивному вызову `power`. Начнем со значения `1`, потому что мы обратимся к проблеме в обратном порядке. Рекурсия произойдет так:

```
power(2, 5, 1) = power(2, 4, 1 * 2)
               = power(2, 4, 2)
               = power(2, 3, 2 * 2)
               = power(2, 3, 4)
               = power(2, 2, 4 * 2)
               = power(2, 2, 8)
               = power(2, 1, 8 * 2)
               = power(2, 1, 16)
               = power(2, 0, 16 * 2)
               = power(2, 0, 32)
```

Так как `exponent` обращается в нуль, базовое условие выполняется и возвращается `result`, поэтому мы получаем `32`: -)

Ранний выход типичен для рекурсивных функций. `seq` является ложным, когда пуст (поэтому, когда нет цифр, оставшихся для суммирования).

Синтаксис фрагмента позволяет передавать последовательность в рекурсивно называемую функцию без целого, потребляемого на текущем шаге.

```
def listSum(seq):
    if not seq:
        return 0
    return seq[0] + listSum(seq[1:])

print listSum([1,3,4,5,6])  # prints 19
```

```
def power(a,b): #a^b
    if b==0:
        return 1
    elif b>0:
        return a * power(a,b-1)
    elif b<0:
        return power(a, b+1)/a
```

```
def listSum(L):
    """Returns a sum of integers for a list containing
    integers.
    input: list of integers
    output: listSum returns a sum of all the integers
    in L.
    """
    if L == []:
        return []
    if len(L) == 1:
        return L[0]
    else:
        return L[0] + listSum(L[1:])
print listSum([1, 3, 4, 5, 6])
print listSum([])
print listSum([8])
```

В некоторых случаях описание функции элегантнее всего выглядит с применением вызова этой же функции. Такой прием, когда функция вызывает саму себя, называется рекурсией. В функциональных языках рекурсия обычно используется много чаще, чем итерация (циклы).

В следующем примере переписывается функция bin() в рекурсивном варианте:

```
def bin(n):
"""Цифры двоичного представления натурального числа """
if n == 0:
return []
n, d = divmod(n, 2)
return bin(n) + [d]
print bin(69)
```

Здесь видно, что цикл while больше не используется, а вместо него появилось условие окончания рекурсии: условие, при выполнении которого функция не вызывает себя.

Конечно, в погоне за красивым рекурсивным решением не следует упускать из виду эффективность реализации. В частности, пример реализации функции для вычисления n-го числа Фибоначчи это демонстрирует:

```
def Fib(n):

if n < 2:

return n

else:

return Fib(n-1) + Fib(n-2)
```

В данном случае количество рекурсивных вызовов растет экспоненциально от числа n, что совсем не соответствует временной сложности решаемой задачи.

В качестве домашнего упражнения предлагается написать итеративный и рекурсивный варианты этой функции, которые бы требовали линейного времени для вычисления результата.

Предупреждение:

При работе с рекурсивными функциями можно легко превысить глубину допустимой в Python рекурсии. Для настройки глубины рекурсии следует использовать функцию setrecursionlimit(N) из модуля sys, установив требуемое значение N.