**Рекурсия**

**Введение**

Вот мы и подобрались еще к одной важной теме, очень пугающей начинающих разработчиков.

Рекурсия является формой итерации, как и цикл. Рекурсия и цикл взаимозаменяемы.

Вы спросите, а зачем использовать рекурсию, если существует цикл?

* Существую задачи, которые имеют достаточно сложный алгоритм, но благодаря рекурсии получают простое решение.
* Считается, что рекурсия дает более стильное, элегантное решение. Знание и умение применить рекурсию повышает статус разработчика.

Вроде пользу рекурсии обосновали. В чем же тогда сложность ее использования?

А в том, что мы привыкли мысленно представлять алгоритм с использованием циклов и потом перекладывать его на язык программирования. Здесь же нам придется мыслить совсем по-другому – рекурсивно. Т.е. сложность именно в правильном осмыслении рекурсивной работы алгоритма.

**Понятие рекурсии**

Под рекурсией понимается разбиение задачи на подзадачи до тех пор, пока не появляется возможность определить результат простым способом. Рекурсия подразумевает вызов функцией самой себя. Да, этот так! Функция может вызывать саму себя.

Начнем сразу с примера.

**Листинг 1. task\_1.py**

|  |
| --- |
| **def** get\_sum\_1(lst\_obj):  *"""Простой цикл"""* res = 0  **for** el **in** lst\_obj:  res = res + el  **return** res    print(get\_sum\_1([1, 3, 5, 7, 9])) |

Тут все понятно, но что, если нам нельзя использовать цикл?

Представим, что мы можем решить эту задачу в виде функции, которая на каждом шаге принимает два аргумента – числа.

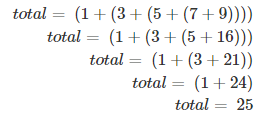
Тогда задачу можно обрисовать так:



Скобки можно расставить и так:



Заметим, что самое внутреннее выражение (7+9) вычисляется очень просто, без спец. алгоритмов и задачу мы можем представить так:



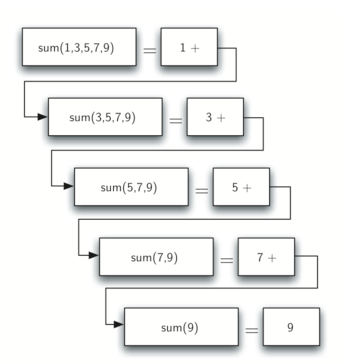
Можно сказать, что сумма списка при таком решении — это сумма первого элемента списка и уже посчитанной суммы остатка.

**Листинг 2. task\_1.py**

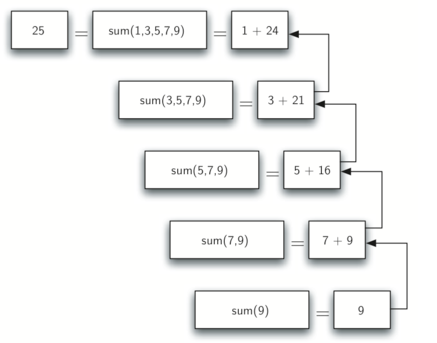
|  |
| --- |
| **def** get\_sum\_2(lst\_obj):  *"""Простая рекурсия"""* **if** len(lst\_obj) == 1:  **return** lst\_obj[0]  **else**:  **return** lst\_obj[0] + get\_sum\_2(lst\_obj[1:])   print(get\_sum\_2([1, 3, 5, 7, 9])) |

Во второй строке мы проверяем, что список имеет длину в один элемент. На этой проверке мы определяем условие завершения рекурсивных вызовов. В 5-й строке ф-ция вызывает саму себя. Вот те самые итерации, на каждой из которых ф-ция получает новое значение на вход.

А вот так выглядит сама последовательность рекурсивных вызовов:



При достижении точки максимального упрощения задачи начинаем собирать вместе кусочки решения, пока они не сформируют итоговое решение всей задачи.



**Законы рекурсии**

**Рекурсивный алгоритм должен иметь базовый случай.**

Рекурсивные вызовы не могут быть вечными и должны когда-то завершаться. Должны быть условия, завершающие вызовы функции самой себя. Базовый случай должен представлять собой простейшую задачу (выражение), которое можно решить без применения доп. средств. В примере выше базовым случаем является список длиной в 1 элемент.

**Рекурсивный алгоритм должен изменять свое состояние и «продвигаться» к базовому случаю.**

Цель алгоритма – достижение некоторого результата. И на пути к этому результату происходит изменение некоторых данных, на которые алгоритм опирается в процессе своей работы. В примере, рассмотренном выше, список, передаваемый в функцию, уменьшается на каждом шаге на один элемент.

**Рекурсивный алгоритм должен вызывать сам себя.**

В этом заключается сама суть рекурсии. Функция вызывает саму себя. В этом элегантность подхода. Мы разбиваем задачу на подзадачи. У всех подзадач одинаковая логика, но разные входные значения. Разбиение на подзадачи происходит до тех пор, пока не останется задача, которую можно решить простым способом.

Рассмотрим еще несколько примеров с рекурсией.

**Листинг 2. task\_2.py**

|  |
| --- |
| *"""Рекурсия против цикла Вывод чисел по убыванию, начиная с текущего и до нуля """* **def** count\_cycle(i):  *"""Цикл"""* **while** i >= 0:  print(i)  i -= 1   count\_cycle(3)   **def** count\_recur(i):  *"""Рекурсия"""* print(i)  *# базовый случай* **if** i <= 0:  **return** *# рекурсивный случай* count\_recur(i-1)   count\_recur(3) |

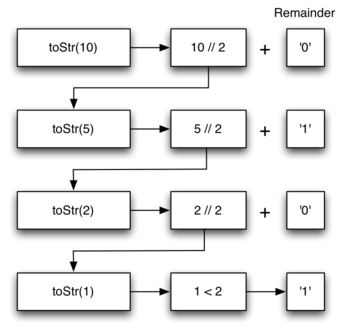
**Листинг 3. task\_3.py**

|  |
| --- |
| *"""Изменение значений переменных"""* **def** recursion(a, b):  *"""Рекурсия"""  # базовый случай  # последний шаг рекурсии* **if** a == b:  **return** str(a)  *# шаг рекурсии  # рекурсивное условие* **elif** a > b:  **return f'{**str(a)**} {**recursion(a - 1, b)**}'** *# шаг рекурсии  # рекурсивное условие* **elif** a < b:  **return f'{**str(a)**} {**recursion(a + 1, b)**}'** print(recursion(20, 15)) print(recursion(10, 15)) |

Рассмотрим еще одну интересную задачу: конвертирование целого числа в строку по любому основанию.

**Листинг 4. task\_4.py**

|  |
| --- |
| *"""Конвертация"""* **def** convert\_to\_str(n, base\_val):  convert\_str = **"0123456789ABCDEF"** *# Базовый случай, в котором n должно быть меньше,  # чем основание, по которому мы конвертируем* **if** n < base\_val:  **return** convert\_str[n]  *# Здесь выполняются 2-й и 3-й законы рекурсии  # выполняется рекурсивный вызов и происходит  # уменьшение размера задания с помощью деления* **else**:  **return** convert\_to\_str(n // base\_val, base\_val) + convert\_str[n % base\_val]   print(convert\_to\_str(10, 2)) |



**Рекурсия и стек**

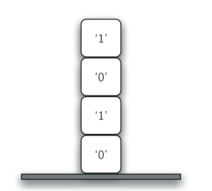
Рекурсивные функции связаны с понятием стека вызовов. Как только происходит вызов функции, функция помещается на вершину стека вызовов, это очень напоминает знакомый всем пример со стопкой книг. Далее, если мы готовы снять что-то обратно, происходит снятие верхнего элемента.

Изменим рассмотренный выше алгоритм таким образом, чтобы он помещал строки в стек перед выполнением рекурсивного вызова.

**Листинг 5. task\_5.py**

|  |
| --- |
| *"""Конвертация"""* **from** stack **import** StackClass  sc\_obj = StackClass()   **def** convert\_to\_str(n, base\_val):  convert\_str = **"0123456789ABCDEF"   while** n > 0:  **if** n < base\_val:  sc\_obj.push\_in(convert\_str[n])  **else**:  sc\_obj.push\_in(convert\_str[n % base\_val])  *# стек пополняется и достигает длины 4* print(sc\_obj.stack\_size())  n = n // base\_val   res = **""  while not** sc\_obj.is\_empty():  res = res + str(sc\_obj.pop\_out())  **return** res   print(convert\_to\_str(10, 2)) *# здесь стек уже пустой. все возвраты выполнены* print(sc\_obj.stack\_size()) |

Теперь при каждом вызове функции convert\_to\_str в стек помещается символ.



Теперь мы более-менее понимаем, как в Python реализованы рекурсивные вызовы. При вызове функции для управления ее локальными переменными выделяется фрейм стека. Возвращаемое значение к моменту завершения работы функции будет находиться на вершине стека и доступно для вызывающей части программы.

Как же выглядит теперь стек вызовов, если показать его подробнее?

convert\_to\_str(10, 2)

n = 10

base\_val = 2

convert\_to\_str(10//2, 2) + conver\_str[10%2]

convert\_to\_str(5, 2)

n = 5

base\_val = 2

convert\_to\_str(5//2, 2) + conver\_str[5%2]

convert\_to\_str(2, 2)

n = 2

base\_val = 2

convert\_to\_str(2//2, 2) + conver\_str[2%2]

В этой задаче вызов convert\_to\_str(2//2, 2) возвращает значение ‘1’ и оставляет его в стеке. Далее оно подставляется вместо функции convert\_to\_str(1, 2) в выражение "1" + convert\_str[2%2], которое оставляет на вершине стека ‘10’.

Можно сделать вывод, что стек рекурсии работает также, как и в примере, который мы рассмотрели на предыдущем уроке. Только здесь стек не программный, а аппаратный.

Рассмотрим еще один яркий пример использования рекурсии – вычисление факториала числа.

**Листинг 6. task\_6.py**

|  |
| --- |
| *"""Факториал через рекурсию"""* **import** sys  print(sys.getrecursionlimit()) sys.setrecursionlimit(10000) print(sys.getrecursionlimit())   **def** factorial(n):  **if** n <= 1:  **return** 1  **else**:  **return** n \* factorial(n - 1)   print(factorial(100)) print(factorial(1000))  **""" Рекурсивные функции используют так называемый «Стек вызовов».  Когда программа вызывает функцию, функция отправляется на верх стека вызовов.  Это похоже на стопку книг, вы добавляете одну вещь за одни раз.  Затем, когда вы готовы снять что-то обратно, вы всегда снимаете верхний элемент. """  ''' 0 шаг. Вызов функции: fac(5) 1. fac(5) возвращает fac(4) \* 5 2. fac(4) => fac(3) \* 4 3. fac(3) => fac(2) \* 3 4. fac(2) => fac(1) \* 2 5. fac(1) => fac(0) \* 1 (завершение рекурсивных вызовов) 6. 1 \* 1 - возврат в вызов fac(1) (fac(0) \* 1 -> 1 \* 1) 6. 1 \* 2 - fac(2) 7. 2 \* 3 - fac(3) 8. 6 \* 4 - fac(4) 9. 24 \* 5 – fac(5) 10. Возврат в основную ветку программы значения 120 '''** |

Рассмотрим еще один яркий пример использования рекурсии – вычисление чисел Фибоначчи.

**Листинг 7. task\_7.py**

|  |
| --- |
| *"""Числа Фибоначчи"""* **import** sys   sys.setrecursionlimit(10000)   **def** fib(n, summ):  **if** n < 1:  **return** summ  **return** fib(n-1, summ+n)   c = 998 *# c = 998 - уже не работает # необузданная рекурсия вызывает переполнение стека* print(fib(c, 0)) |

Если говорить о реальных задачах, то рекурсия очень органично вписывается в задачи «сканирования содержимого» директорий, в том числе вложенных.

**Листинг 8. task\_8.py**

|  |
| --- |
| **import** os   **def** get\_directory\_files(path):  *"""Функция вывода содержимого директории"""* struct = []  **for** file\_or\_directory **in** os.listdir(path):  full\_name = os.path.join(os.path.abspath(path), file\_or\_directory)  **if** os.path.isfile(full\_name):  struct.append((os.path.abspath(path), file\_or\_directory))  **else**:  struct.extend(get\_directory\_files(full\_name))  **return** struct   my\_res = get\_directory\_files(**'mainapp'**) print(my\_res) |

Рассмотрим еще один яркий пример использования рекурсии – определение НОД через алгоритм Евклида.

**Листинг 9. task\_9.py**

|  |
| --- |
| *"""Определение НОД"""* **def** first\_method(a, b):  *"""Цикл"""* **while** a != b:  **if** a > b:  a = a - b  **else**:  b = b - a  print(a)   first\_method(36, 60)   **def** second\_method(a, b):  *"""Рекурсия"""* **if** b == 0:  **return** a  **else**:  **return** second\_method(b, a % b)   print(second\_method(36, 60))   **def** third\_method(a, b):  *"""Тоже цикл"""* **while** b != 0:  a, b = b, a % b  **return** a   print(third\_method(36, 60)) |

Рассмотрим еще один яркий пример использования рекурсии – расчет сложного процента

**Листинг 10. task\_10.py**

|  |
| --- |
| *"""Рассчитаем сложный процент по кредиту  Нам нужны сведения:  Срок кредита в годах Процентная ставка Количество платежей в год Сумма кредита  """* period = 10 rate = .06 pay\_counts = 12 credit\_sum = 4000   **def** calc\_1(credit\_sum\_val, pay\_counts\_val, period\_val, rate\_val):  *"""Цикл"""* total = period\_val \* pay\_counts\_val  **for** \_ **in** range(1, (total+1)):  credit\_sum\_val = credit\_sum\_val\*(1+(rate\_val/pay\_counts\_val))  **return** credit\_sum\_val   print(calc\_1(credit\_sum, pay\_counts, period, rate))   **def** calc\_2(credit\_sum\_val, pay\_counts\_val, period\_val, rate\_val, number\_of\_recursions):  *"""Рекурсия"""* **if** number\_of\_recursions == 0:  total = pay\_counts\_val \* period\_val  **elif** number\_of\_recursions != 0:  total = period\_val  **if** period\_val == 0:  **return** credit\_sum\_val  **else**:  new\_duration = total - 1  amount = credit\_sum\_val\*(1+(rate\_val/pay\_counts\_val))  **return** calc\_2(amount, pay\_counts\_val, new\_duration, rate\_val, 1)   print(calc\_2(credit\_sum, pay\_counts, period, rate, 0)) |

