Лекция №15.
Тема: **Рекурсия.**
Понятие и особенности рекурсии, виды рекурсии. Структура рекурсивной функции. Механизм работы рекурсивной программы со стеком. Управление глубиной рекурсии в языке Python. Целесообразность выбора рекурсии.
Ключевые слова: рекурсия, рекуррентное соотношение, тип рекурсии, стек, стек вызовов, глубина рекурсии.
Keywords: recursion, recurrence relation, type of recursion, stack, stack call, depth of recursion.

1 Понятие и особенности рекурсии, виды рекурсии

Объект называется рекурсивным, если он содержит в своём описании вызовы самого себя (см. https://en.wikipedia.org/wiki/Recursion_(computer_science)). Рекурсивную программу (функцию) P можно определить как последовательность операторов S_i, не содержащих рекурсии, и собственно рекурсивной части P: $P \equiv \Re[S_i, P]$.

Особенности рекурсии:

1. Происхождение рекурсии. Источником рекурсивно сформулированных задач выступают задачи, в которых процесс обработки информации представлен рекуррентными соотношениями (см. http://en.wikipedia.org/wiki/Recurrence_relation). Рекуррентные соотношения связывают результат каждого следующего шага с результатом текущего шага. Простым примером рекуррентного соотношения является функция факториала

$$n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n,$$

где следующий факториал $(n+1)!$ можно вычислить по текущему как

$$(n+1)! = n! \cdot (n+1)$$

Рекурсивное определение задачи позволяет свести ее решение к решению последовательности задач, связанных рекуррентно. Такая декомпозиция сложной задачи является естественной для многих, уже ставших классическими, алгоритмов. Метод рекурсивного решения задач теоретически опирается на метод математической индукции (см. https://en.wikipedia.org/wiki/Mathematical_induction).

2. Рекурсия и цикл. Рекурсия предполагает выполнение участка программы многократно, а значит рекурсивный процесс может быть представлен итерационно (с помощью цикла). При выполнении цикла или рекурсии проводится проверка некоторого условия (разбор случав). При рекурсии в результате разбора случаев должен реализовываться выбор, который сокращает выполнения. Рекурсия завершается на операторах, в которых нет больше ветвей, содержащих рекурсивные вызовы. Для заданного набора значений параметров программы рекурсивный процесс вычисления конечен, если он требует конечного числа выполнений.

3. Рекурсия и Python. Средство для рекурсивного представления программ в языке Python – описание функций, которые содержат вызовы самих себя по имени.

4. Назначение рекурсивных функций. Наглядно и компактно представить вычисление рекурсивно определённого алгоритма.

Пример 1. Реализация рекурсивной функции для вычисления факториала.

```
    def factorial(n):
        '''factorial(x) -> Integral

        Find x!.'''

        if type(n) is not int:
            raise  ValueError('factorial()  only  accepts  integral
values')
        if n < 0:
            raise  ValueError('factorial()  not  defined  for  negative
values')
        if n == 0:
            return 1
        else:
            return n * factorial(n - 1)

    x = input('F = ')
    print(factorial(x))
```

Рекурсивная форма организации данного алгоритма дает более компактный текст программы, чем итерационная форма. Но на выполнение требуется больше времени и может происходить переполнение стека.

5. Механизм работы рекурсивной программы со стеком. Тьюринг А.М. в 40-х годах 20 века разработал механизм стека, названный реверсивной памятью, для связывания подпрограмм, локальных переменных и параметров [Кнут, т.1, с.514]. При каждом вызове рекурсивной подпрограммой самой себя реализуется механизм распределения стековой памяти под следующий набор значений локальных переменных. Наборы значений помещаются в стек (push) до тех пор, пока не исчерпаны рекурсивные вызовы. Обратный процесс – извлечение данных из стека (pop), согласно принципу действия стековой памяти, выполняется, начиная с последнего набора значений, помещенного в стек.

6. Глубина и текущий уровень рекурсии. Глубина рекурсии определяется числом наборов значений переменных, одновременно находящихся в стековой памяти. Глубина рекурсии меняется при помещении и извлечении данных из стека. Расчет максимальной глубины рекурсии и объема памяти под один набор значений дает возможность оценить требуемый размер стека. Текущий уровень рекурсии – число рекурсивных вызовов в конкретный момент времени выполнения программы. В интерпретаторе Python по умолчанию установлено ограничение глубины рекурсии, равное 1000.

7. Недостаток рекурсии. Серьезным недостатком рекурсивного решения задачи может быть ограничение на размер стека, в котором хранятся все рекурсивные вызовы. Временные затраты рекурсивного решения выше, чем итерационного за счет затрат на обработку вызовов функций. Дополнительные

сложности (неэффективное заполнение стека, некорректности при возврате значений в точку вызова) возникают при использовании сложных видов рекурсии.

8. Характерные признаки классической рекурсии:

- функция вызывает саму себя с меньшими значениями аргумента;
- функция содержит условие завершения, при выполнении которого непосредственно вычисляется результирующее значение.

9. Виды рекурсии:

- прямая рекурсия – функция содержит явное обращение к самой себе

```
def Rec_1(k):
    ...
    Rec_1(k - 1)
    ...
```

- косвенная рекурсия – функция S_1 содержит обращение к функции S_2, которая содержит прямые или косвенные обращения к S_1.

```
def Rec_B(k):
    ...
    Rec_A(k - 1)
    ...

def Rec_A(k):
    ...
    Rec_B(k - 1)
    ...
```

- линейная (повторительная) рекурсия – рекурсивный вызов встречается только один раз в отдельных ветвях разбора случаев, т.е. при каждом рекурсивном вызове порождается только один новый рекурсивный вызов, и порядок порождения вызовов определен однозначно (рекурсивный вызов при линейной рекурсии называется гладким).
- нелинейная рекурсия – в отдельных ветвях разбора случаев имеется несколько рекурсивных вызовов, т.е. при определённых условиях возможны различные порядки порождения и даже параллельная работа.

Рассмотрим случай взаимных рекурсий: $f(x) = p(x, g(x))$, $g(x) = f(x, f(x))$. Взаимное расположение может быть организовано так

```
def F(y):
    # Вызов функции G.
    G(y)

def G(x):
    # Вызов функции F.
    F(x)
```

2 Структура рекурсивной функции

Структура рекурсивной функции определяет взаимное расположение в теле функции операторов без рекурсивных вызовов (операторов действий) и операторов с рекурсивными вызовами. Выделяют следующие варианты структуры:

1. Действия на спуске. Операторы действий предшествуют операторам с рекурсивным вызовом.

```
def Recursion():
    Si      # Некоторые действия.
    if expression:
        Recursion()
```

2. Действия на возврате. Операторы действий следуют за операторами с рекурсивным вызовом.

```
def Recursion():
    if expression:
        Recursion()
    Si      # Некоторые действия.
```

3. Действия и до, и после. Операторы действий расположены до и после операторов с рекурсивным вызовом.

```
def Recursion():
    Si      # Некоторые действия.
    if expression:
        Recursion()
    Si      # Некоторые действия.
```

3 Механизм работы рекурсивной программы со стеком

Рассмотрим простую рекурсивную функцию:

```
def Recursion(a):
    if a > 0:
        Recursion(a - 1)
    print(a)
```

Если в основной программе поставить вызов, например, вида Recursion (3), то последовательность выполнения операторов можно проиллюстрировать с помощью рисунков 1, 2.

Функция Recursion ( ) вызывается с параметром a = 3. В ней содержится вызов функции Recursion ( ) с параметром a = 2. Предыдущий вызов еще не завершился, поэтому создается еще один вызов функции и до окончания его работы первый вызов свою работу не заканчивает. Процесс вызова заканчивается, когда параметр a = 0. В этот момент одновременно выполняются 4 вызова функции. Количество одновременно выполняемых вызовов функции описывает глубину рекурсии. Четвертый вызов функции (Recursion (0)) напечатает число 0 и закончит свою работу. После этого управление возвращается к вызову процедуры, который создал завершившийся вызов (Recursion (1)) и печатается число 1. И так далее пока не завершатся все вызовы функции. Результатом исходного вызова будет печать четырех чисел: 0, 1, 2, 3.

В примере рекурсивный вызов стоит внутри условного оператора. Это необходимое условие для того, чтобы рекурсия закончилась. Также процедура вызывает сама себя с другим параметром, отличающимся от параметра самого первого вызова. Так как в процедуре не используются глобальные переменные, то это также необходимо, чтобы избежать бесконечной рекурсии.

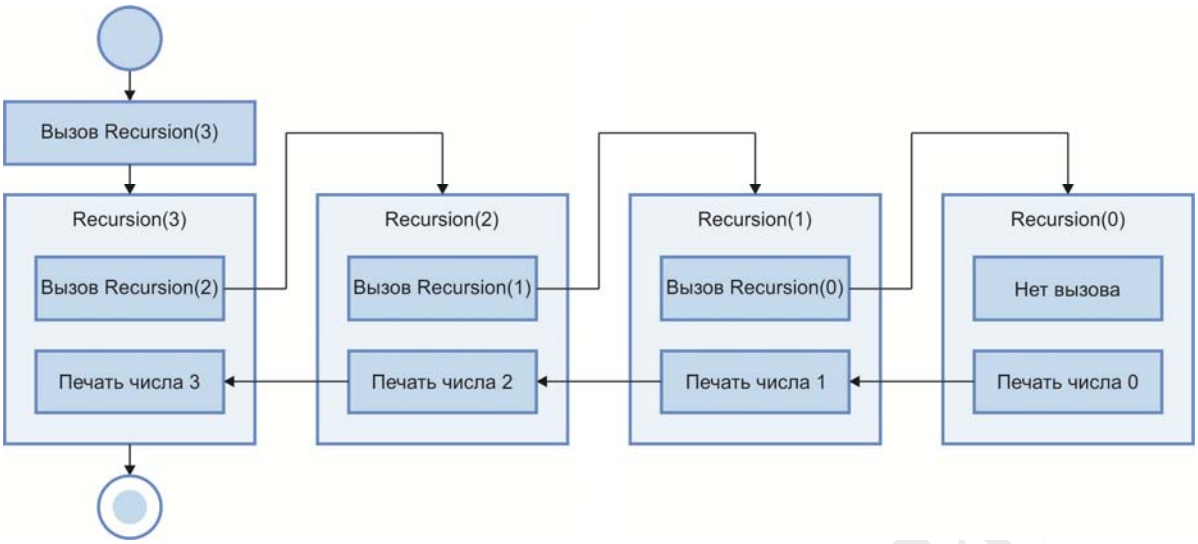

Рис. 1. Блок-схема выполнения функции Recursion ( ) с параметром 3

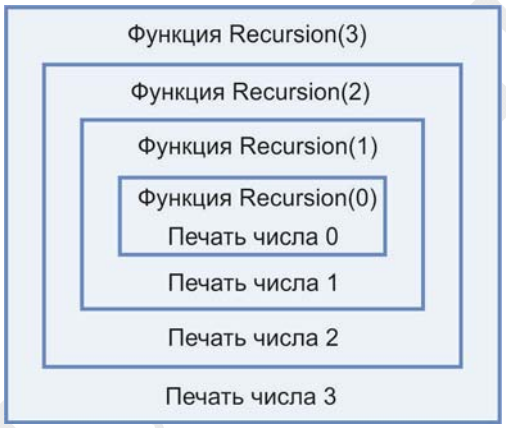

Рис. 2. Вложенность рекурсивных вызовов функции Recursion ( )

Пример 2. Реализация функции возведения числа в целочисленную степень.

```
def PowIter(x, n):
    y = 1
    for i in range(n):
        y *= x
    return y
```

В стек загружаются x, n, y, i. Рассмотрим рекурсивную функцию, которая позволяет решить эту же задачу.

```
def PowRec(x, n):
    if n == 0:
        return 1
    else:
        y = x * PowRec(x, n - 1)
        return y
```

Состояние значений переменных в процессе вызова PowRec(3,2) представлено в табл. 1.

Таблица 1 – Рекурсивные вызовы функции PowRec с параметрами (3, 2)

Вход в функцию	Уровни рекурсии						Выход из процедуры
	0		1		2		
	x	n	x'	n'	x''	n''	y
PowRec(3,2)	3	2					?
PowRec(3,1)	3	2	3	1			?
PowRec(3,0)	3	2	3	1	3	0	1
PowRec(3,1)	3	2	3	1			3
PowRec(3,2)	3	2					9

4 Управление глубиной рекурсии в языке Python

В языке Python на рекурсивную функцию накладываются два ограничения:

1. Ограничение на размер стека, которое устанавливается операционной системой. Переполнение стека приводит к ошибки памяти (MemoryError) Stack overflow.
2. Ограничение на глубину рекурсии (по умолчанию равно 1000). Превышение допустимой глубины рекурсии приводит к ошибки времени выполнения (RuntimeError) maximum recursion depth exceeded in comparison. Для программного управления ограничением на глубину рекурсии в языке Python используются следующие функции:

- `sys.getrecursionlimit( ) –`

возвращает текущий лимит на глубину рекурсии.

- `sys.setrecursionlimit(n) –`

устанавливает лимит на глубину рекурсии.

Рассмотрим на примере работу функций по управлению глубиной рекурсии.

Пример 3. Реализация функции вычисления суммы N положительных целых чисел.

Вариант рекурсивной функции без управления глубиной рекурсии:

```
def SummaRec(n):
    if n == 1:
        return 1
    elif n > 1:
        return n + SummaRec(n - 1)

n = int(input('N = '))
print('Res = ', SummaRec(n))
```

Результат работы программы

```
N = 2000
…
RuntimeError: maximum recursion depth exceeded in comparison
```

Вариант рекурсивной функции с управлением глубины рекурсии:

```
import sys

def SummaRec(n):
    if n == 1:
```

```
            return 1
        elif n > 1:
            return n + SummaRec(n - 1)

    m = sys.getrecursionlimit()
    n = int(input('N = '))
    if n >= m - 1:
        sys.setrecursionlimit(n + 2)
        print("Глубина рекурсии переустановлена в",
sys.getrecursionlimit())
    print('Res = ', SummaRec(n))
```

Результат работы программы

```
N = 2000
Глубина рекурсии переустановлена в 2002
Res = 2001000
```

5 Целесообразность выбора рекурсии

Рассмотрим вопрос целесообразности рекурсивных функций на примерах.

Пример 4. Реализация функции вычисления чисел Фибоначчи.

```
def FibRec(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return FibRec(n - 1) + FibRec(n - 2)
```

Схема рекурсивных вызовов функции FibRec с параметром 5 представлена на рисунке 3 (Call stack, см. https://en.wikipedia.org/wiki/Call_stack).

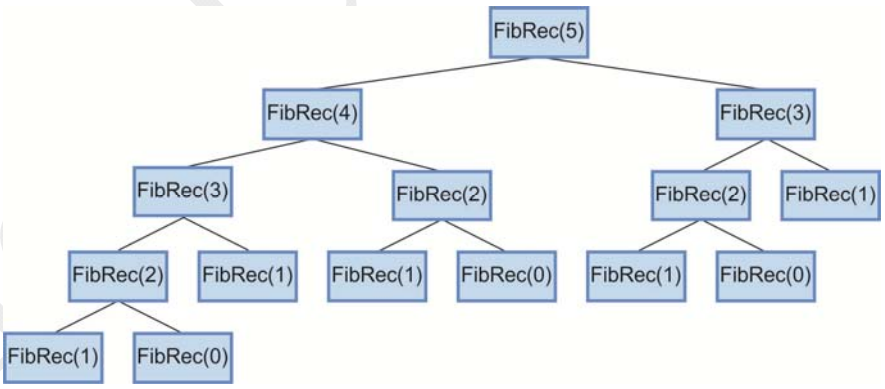

Рис. 3. Схема рекурсивных вызовов FibRec с параметром 5

Раскрутка вызовов будет выполняться вглубь сначала по правой, а затем по левой ветви. Стек будет заполняться вызовами, а затем раскручиваться в обратном направлении. Каждое обращение приводит к двум дальнейшим обращениям (при n>1), то есть общее число обращений растёт экспоненциально.

Очевидно, что числа Фибоначчи можно вычислять по итеративной схеме, при которой использование вспомогательных переменных x, y позволяет избежать повторного вычисления одних и тех же значений.

```
def FibIter(n):
    x, y = 0, 1
    for i in range(n):
        x, y = y, x + y
    return x
```

Предложенный рекурсивный алгоритм вычисления чисел Фибоначчи неэффективен как с точки зрения временной сложности (функция тратит время на обращения к себе самой), так и с точки зрения пространственной сложности (расходуется память на сохранение среды незавершенного вызова). Таким образом, следует избегать рекурсии, если имеется очевидное итеративное решение поставленной задачи.

Пример 5. Реализация функции, которая при нечетности аргумента n выполняет рекурсивный вызов с аргументом (3 * n + 1), а при четности – с аргументом (n // 2).

```
def Puzzle(n):
    if n == 1:
        return 1
    elif n % 2 == 0:
        return Puzzle(n // 2)
    else:
        return Puzzle(3 * n + 1)
```

Цепочка рекурсивных вызовов Puzzle с параметром 3 будет следующей:

```
Puzzle(3)
  Puzzle(10)
    Puzzle(5)
      Puzzle(16)
        Puzzle(8)
          Puzzle(4)
            Puzzle(2)
              Puzzle(1)
```

Данная вложенная последовательность вызовов функции со временем завершается, однако нельзя гарантировать, что рекурсивная функция не будет иметь произвольную глубину вложенности для какого-либо аргумента.

Желательно использовать рекурсивные функции, которые вызывают сами себя с меньшими значениями аргументов.

Пример 6. Реализация рекурсивной функции C(m, n) для вычисления биномиального коэффициента C_n^m по формуле $C_n^0 = C_n^n = 1;\ C_n^m = C_{n-1}^m + C_{n-1}^{m-1}$, где $0 \le m \le n$.

```
def C(m, n):
    if m == 0 or m == n:
        return 1
    else:
        return C(m, n - 1) + C(m - 1, n - 1)
```

Рекурсия наиболее пригодна для случаев, когда поставленная задача определена рекурсивно.

Пример 7. Ханойские башни. Даны 3 стержня (A, B, C) и n дисков разного размера. Пусть диски вначале находятся на стержне A в порядке убывания. Нужно переместить n дисков на стержень B, чтобы они оставались в том же порядке, выполняя правила:

- на каждом шаге перемещается ровно один диск с одного стержня на другой;
- диск большего размера нельзя переместить на меньший;
- один из трёх дисков можно использовать как промежуточный.

```
def Hanoi(n, A, B, C):
    '''Задача о Ханойских башнях

    Hanoi(n, A, B, C)
    n - количество дисков,
    A - стержень-источник,
    B - стержень-приемник,
    C - промежуточный стержень'''

    if n > 0:
        Hanoi(n - 1, A, C, B)
        print(A, ' to ', B)
        Hanoi(n - 1, C, B, A)

# Вызов функции.
Hanoi(3, 'A', 'B', 'C')
```

Результат работы программы:

```
A  to  B
A  to  C
B  to  C
A  to  B
C  to  A
C  to  B
A  to  B
```

Читабельность рекурсивной функции для решения задачи о ханойских башнях значительно превосходит итеративное решение.