



# Применение функций



# Рекурсивные алгоритмы

# Что такое рекурсия?

## Натуральные числа:

- 1 – натуральное число
- если  $n$  – натуральное число, то  $n + 1$  – натуральное число

индуктивное  
определение

**Рекурсия** — это способ определения множества объектов через само это множество на основе заданных простых базовых случаев.

Рекурсивный алгоритм разбивает задачу на части, которые по своей структуре являются такими же как исходная задача, но более простыми.

# Рекурсия

- При обращении подпрограммы к самой себе происходит то же самое, что и при обращении к любой процедуре или функции:
  - в стек записывается адрес возврата,
  - резервируется место под локальные переменные,
  - происходит передача параметров,
  - после чего управление передается первому исполняемому оператору программы

Любое рекурсивное определение содержит две части:  
базисную часть и рекурсивную

Базисная часть является не рекурсивным утверждением, которое задает определение для некоторой фиксированной группы объектов

Рекурсивная часть определения записывается таким образом, чтобы при цепочке повторных применений утверждение из рекурсивной части приводилось бы к базисной части

# Рекурсия. Пример. Нечетное число

## ■ Базисная часть:

- число 1 является нечетным целым числом

## ■ Рекурсивная часть:

- если какое либо число  $K$  является нечетным целым числом, то нечетными целыми будут числа, определяемые выражениями  $K - 2$  и  $K + 2$

Докажем, что число  $K = 7$  является нечетным целым числом.

Для этого применим рекурсивную часть определения:

число  $K = 7$  является нечетным целым числом, если нечетным целым является число  $K - 2 = 7 - 2 = 5$ ;

число  $K = 5$  является нечетным целым числом, если нечетным целым является число  $K - 2 = 5 - 2 = 3$ ;

число  $K = 3$  является нечетным целым числом, если нечетным целым является число  $K - 2 = 3 - 2 = 1$ ;

число  $K = 1$  является нечетным целым числом – базовое утверждение.

Таким образом, рекурсивное утверждение удалось привести к базе, следовательно, первоначальное утверждение о том, что число  $K = 7$  - нечетное целое число является истинным.

# Что такое рекурсия?

Примеры:

Факториал:

$$N! = \begin{cases} 1, & N = 1 \\ N \cdot (N-1)!, & N > 1 \end{cases}$$

Числа Фибоначчи:

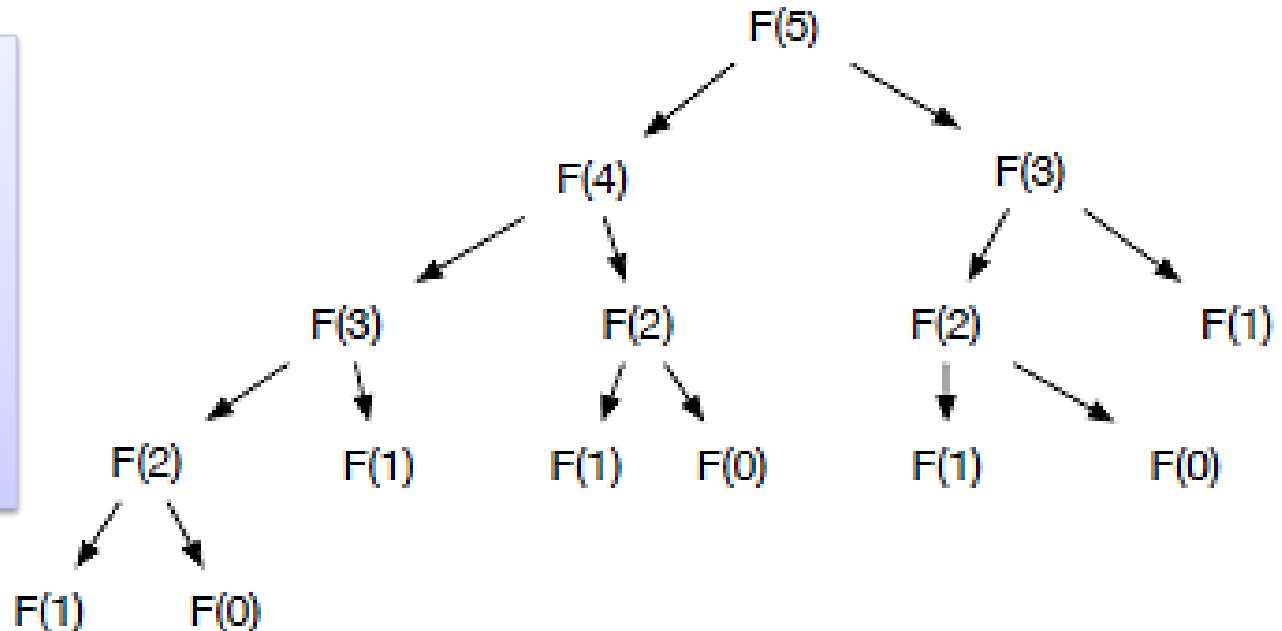
- $F_1 = F_2 = 1$
- $F_n = F_{n-1} + F_{n-2}$  при  $n > 2$

**1, 1, 2, 3, 5, 8, 13, 21, 34, ...**

## Пример. Числа Фибоначчи

- Вычислить  $N$  чисел в последовательности Фибоначчи, — 1, 1, 2, 3, 5, 8, ... — в которой первые два члена равны единице, а все остальные представляют собой сумму двух предыдущих.  $N$  меньше 100.
- Решение задачи: рекурсивная функция

Проблема в том, что мы много раз повторно вычисляем значение функции от одних и тех же аргументов



# Вычисление суммы цифр числа

Задача. Написать рекурсивную функцию, которая вычисляет сумму цифр числа.

```
s = sumDigits ( 1234 )
```



n

```
sumDigits ( 123 ) + 4
```

$n // 10$

$n \% 10$

$\text{sumDigits} ( n ) = \text{sumDigits} ( n // 10 ) + ( n \% 10 )$

$\text{sumDigits} ( n ) = n$  для  $n < 10$



Что дальше?



# Вычисление суммы цифр числа

последняя  
цифра

```
def sumDigits ( n ) :  
    if n < 10 : return n  
    d = n % 10  
    sum = d + sumDigits ( n // 10 )  
    return sum
```

рекурсивный вызов



Где условие окончания рекурсии?

sumDigits ( 1234 )

↓  
4 + sumDigits ( 123 )

↓  
4 + 3 + sumDigits ( 12 )

↓  
4 + 3 + 2 + sumDigits ( 1 )

↓  
4 + 3 + 2 + 1

# Вычисление суммы цифр числа

```
sumDigits ( 123 )
```

```
d = 3
```

```
sum = 3 + sumDigits ( 12 )
```

```
sumDigits ( 12 )
```

```
d = 2
```

```
sum = 2 + sumDigits ( 1 )
```

```
sumDigits ( 1 )
```

```
return 1
```

```
return 3
```

```
return 6
```

```
graph TD; subgraph Frame1 [sumDigits ( 123 )]; d1 = 3; sum1 = 3 + sumDigits ( 12 ); end; subgraph Frame2 [sumDigits ( 12 )]; d2 = 2; sum2 = 2 + sumDigits ( 1 ); end; subgraph Frame3 [sumDigits ( 1 )]; return1 = 1; end; Frame3 -- 1 --> Frame2; Frame2 -- 3 --> Frame1; Frame1 -- 6 --> Out((6));
```

# Алгоритм Евклида

**Алгоритм Евклида.** Чтобы найти НОД двух натуральных чисел, нужно вычитать из большего числа меньшее до тех пор, пока меньшее не станет равно нулю. Тогда второе число и есть НОД исходных чисел.

```
def NOD ( a, b ) :  
    if a == 0 or b == 0 :  
        return a + b ;  
    if a > b :  
        return NOD ( a - b, b )  
    else :  
        return NOD ( a, b - a )
```

условие окончания  
рекурсии

рекурсивные вызовы

# Факториал

Пример recursion\_fun.py

$$N! = \begin{cases} 1, & N = 1 \\ N \cdot (N-1)!, & N > 1 \end{cases}$$

```
def Fact(N):
    print ( "->", N )
    if N <= 1: F = 1
    else:
        F = N * Fact ( N - 1 )
    print ( "<- ", N )
    return F
```

```
-> N = 3
    -> N = 2
        -> N = 1
            <- N = 1
        <- N = 2
    <- N = 3
```



Как сохранить состояние функции перед рекурсивным вызовом?

**Стек** – область памяти, в которой хранятся локальные переменные и адреса возврата.


# Рекурсия – «за» и «против»

- с каждым новым вызовом расходуется память в стеке (возможно переполнение стека)
- затраты на выполнение служебных операций при рекурсивном вызове
- ⊕ программа становится более короткой и понятной
- ⊖
  - возможно переполнение стека
  - замедление работы

! Любой рекурсивный алгоритм можно заменить итерационным!

итерационный  
алгоритм

```
def Fact ( n ) :  
    f = 1  
    for i in range ( 2 , n+1 ) :  
        f *= i  
    return f
```



# Введение в функциональное программирование

# Функциональное программирование

- Вычисления в ФП понимаются не как последовательность изменения состояний, а как вычисление значений функций в их математическом понимании
  - Функции в ФП – это не подпрограммы, а отображения элементов одного множества на другое по определенным правилам
- ФП основывается на взаимодействии с функциями

Функциональная программа – это совокупность определений функций

Функции содержат вызовы других функций, а также инструкции, которые управляют последовательностью этих вызовов

# Функциональное программирование

Вычисления  
начинаются с  
вызова некоторой  
функции



Эта функция  
вызывает функции,  
которые входят в  
её определение в  
соответствии с  
внутренней  
иерархией

Каждый вызов возвращает значение, но помещается оно не в переменную, а в саму функцию, которая этот вызов совершила

Процесс продолжается до того момента, как та функция, с которой начались вычисления, не вернёт пользователю конечный результат

```
my_list = list(map(lambda x, y: x + y, [1, 2], [3, 4]))
```



# Функциональное программирование

- Функциональное программирование определяют как программирование, в котором нет побочных эффектов
- Отсутствие побочного эффекта означает, что функция:
  - ☐ полагается только на данные внутри себя,
  - ☐ не меняет данные, находящиеся вне функции.
- Вычисленный результат – есть единственный эффект выполнения любой функции

Вычисленный результат – есть единственный эффект выполнения любой функции

# Функциональное программирование

- Не функциональная функция

```
a = 0
def increment1():
    global a
    a += 1
```

- Функциональная функция

```
def increment2(a):
    return a + 1
```

Функциональный код не полагается на данные вне текущей функции, и не меняет данные, находящиеся вне функции



# Преимущества ФП

- Надёжность
- Лаконичность
- Удобство тестирования
- Параллелизм

# def или лямбда-выражение?

- Python полностью поддерживает парадигму функционального программирования

```
def add_one (a, b):  
    return a + b + 1
```



```
add_one = lambda a, b: a + b + 1
```

# Что такое функция высшего порядка?

- Функция высшего порядка – это функция, которая может принимать в качестве аргумента другую функцию и/или возвращать функцию как результат работы

В Python есть несколько встроенных функций высшего порядка

# Функция `map`

`fun _map_filter....py`

Функция **`map`** принимает функцию и итератор, возвращает итератор, элементами которого являются результаты применения функции к элементам входного итератора

```
a = [1, 2, 3, 4, 5]
list(map(lambda x: x**2, a))
```

`[1, 4, 9, 16, 25]`

```
def fun(x):
    if x % 2 == 0:
        return 0
    else:
        return x*2
list(map(fun, a))
```

`[2, 0, 6, 0, 10]`

# Функция filter

```
fun _map_filter....py
```

Функция **filter** принимает функцию предикат и итератор, возвращает итератор, элементами которого являются данные из исходного итератора, для которых предикат возвращает True.

```
list(filter(lambda x: x > 0, [-11, 11, -21, 21, 0]))
```

```
[11, 21]
```

# Функция zip

Пример Demo\_zip.py

Упаковывает итерируемые объекты в один список кортежей

```
word_list = ['Elf', 'Dwarf', 'Human']  
digit_tuple = (3, 7, 9, 1)  
ring_list = ['ring', 'ring', 'ring', 'ring', 'ring']  
  
my_list = list(zip(word_list, digit_tuple, ring_list))
```

```
[(('Elf', 3, 'ring'), ('Dwarf', 7, 'ring'), ('Human',  
9, 'ring'))]
```

При работе ориентируется на объект меньшей длины



# Функция `reduce`

`fun _map_filter....py`

Функция **`reduce`** сворачивает переданную последовательность с помощью заданной функции.

```
from functools import reduce
```

Прототип функции:

```
reduce(function, iterable[, initializer])
```

```
def add(x1, x2):  
    return x1 + x2
```

```
a = [1, 2, 3, 4, 5]  
sum_all = reduce(add, a)
```

15

```
mul_all = reduce(lambda x, y: x * y, a)
```

120

`reduce()` запускает цепь вычислений, применяя функцию ко всем элементам последовательности

# Замыкания

Замыкания.ру

- Смысл замыкания состоит в том, что определение функции "замораживает" окружающий её контекст на момент определения

```
def multiplier( n ):  
    def mul( k ):  
        return n * k  
    return mul
```

multiplier возвращает  
функцию умножения на n

```
mul3 = multiplier(3)  
mul5 = multiplier(5)
```

получение функций  
умножения на 3 и на 5