Вятский государственный университет
Цифровые кафедры
Разработка прикладного программного обеспечения

Ходить вокруг, а не около

Авторы
Долженкова Мария Львовна,
кандидат технических наук, заведующий кафедрой ЭВМ
Нижегородова Маргарита Владимировна,
кандидат педагогических наук, доцент кафедры САУ
Шмакова Наталья Александровна,
старший преподаватель кафедры САУ

Дата 20.10.2022

Повторение

Переменная – имя для доступа к области памяти компьютера, выделенной для хранения информации определенного типа.

Создание переменной происходит в момент выполнения операции присваивания значения.

Переменная в Python не хранит значение напрямую – она хранит лишь ссылку на объект (адрес).

Повторение

Тип данных определяет множество значений, набор операций, которые можно применять к таким значениям и способ реализации хранения значений и выполнения операций

К основным встроенным типам относятся:

Логические переменные (bool).

Числа (int, float, complex)

Списки (list, tuple)

Строки

Множества (set)

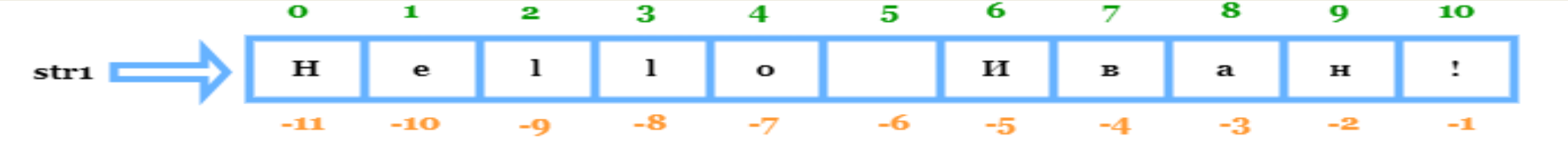
Словари (dict)

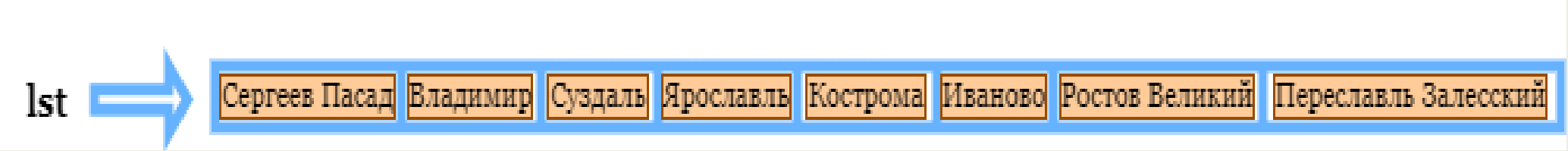
Неизменяемые типы не позволяют модифицировать содержимое ячейки памяти, на которую ссылается переменная. Для изменения значения создается новый объект. Память, занятая предыдущим значением, становится недоступной и освобождается сборщиком мусора.

Изменение мутируемого объекта не приводит к созданию копии

Повторение

Строки, списки и множества являются индексируемыми (итерируемые объекты)





dict1 = {'Сергиев Посад': 49654, 'Владимир': 4922, 'Углич': 48532, 'Муром': 49234,}

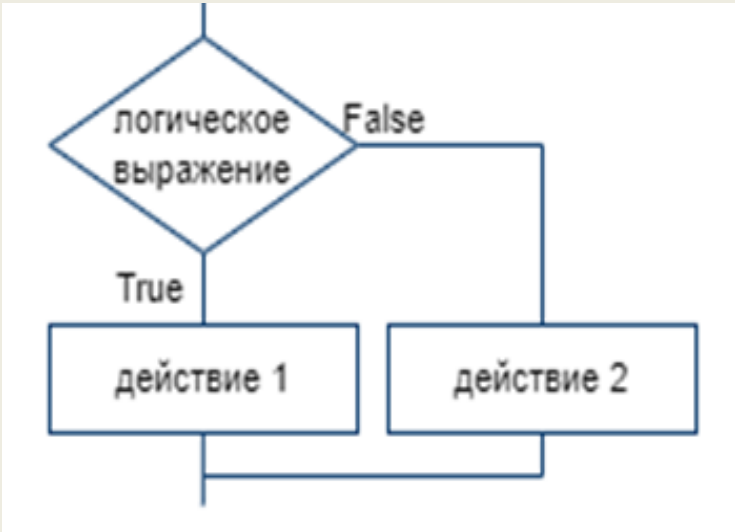
str1[0]='h' lst[1]:='Москва' dict['Владимир']=5000

Повторение

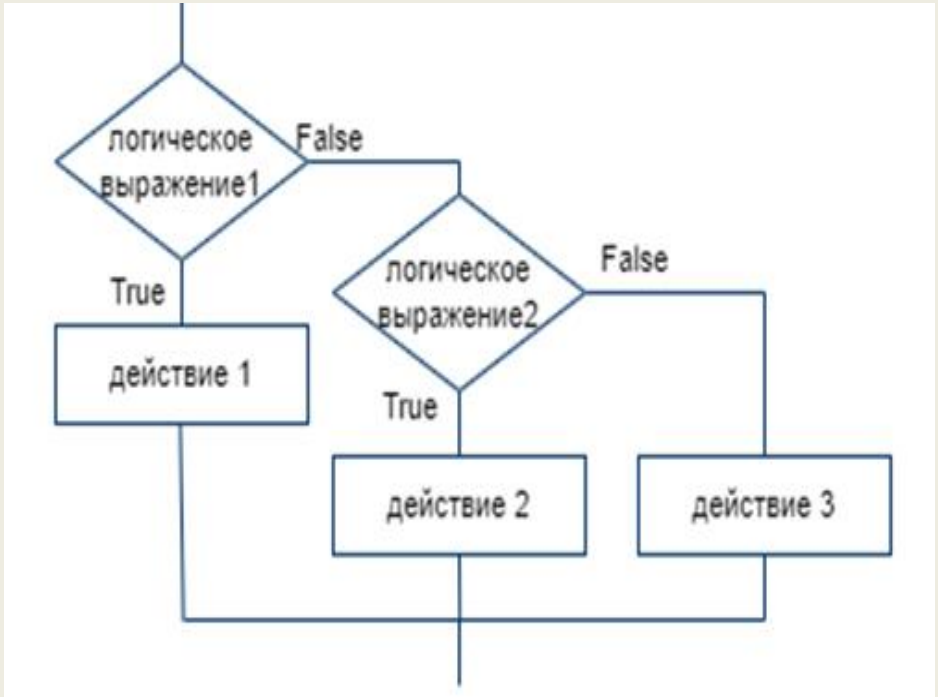
Управляющая конструкция УСЛОВНЫЙ ОПЕРАТОР *позволяет выбрать для дальнейшего решения только одну ветвь решения задачи в зависимости от значения заданного логического условия*

if <условное выражение>:
оператор

if <условное выражение>:
оператор 1
else:
оператор 2



if <условное выражение1>:
оператор 1
elif <условное выражение2>:
оператор 2
else:
оператор 3



Повторение

Циклические процессы обеспечивают многократного повторения заданного участка кода

Тело цикла – последовательность инструкций, которую нужно выполнить несколько раз.

Итерация – однократное выполнение тела цикла.

Завершение цикла – условие выхода из него.

Два типа циклов в Python

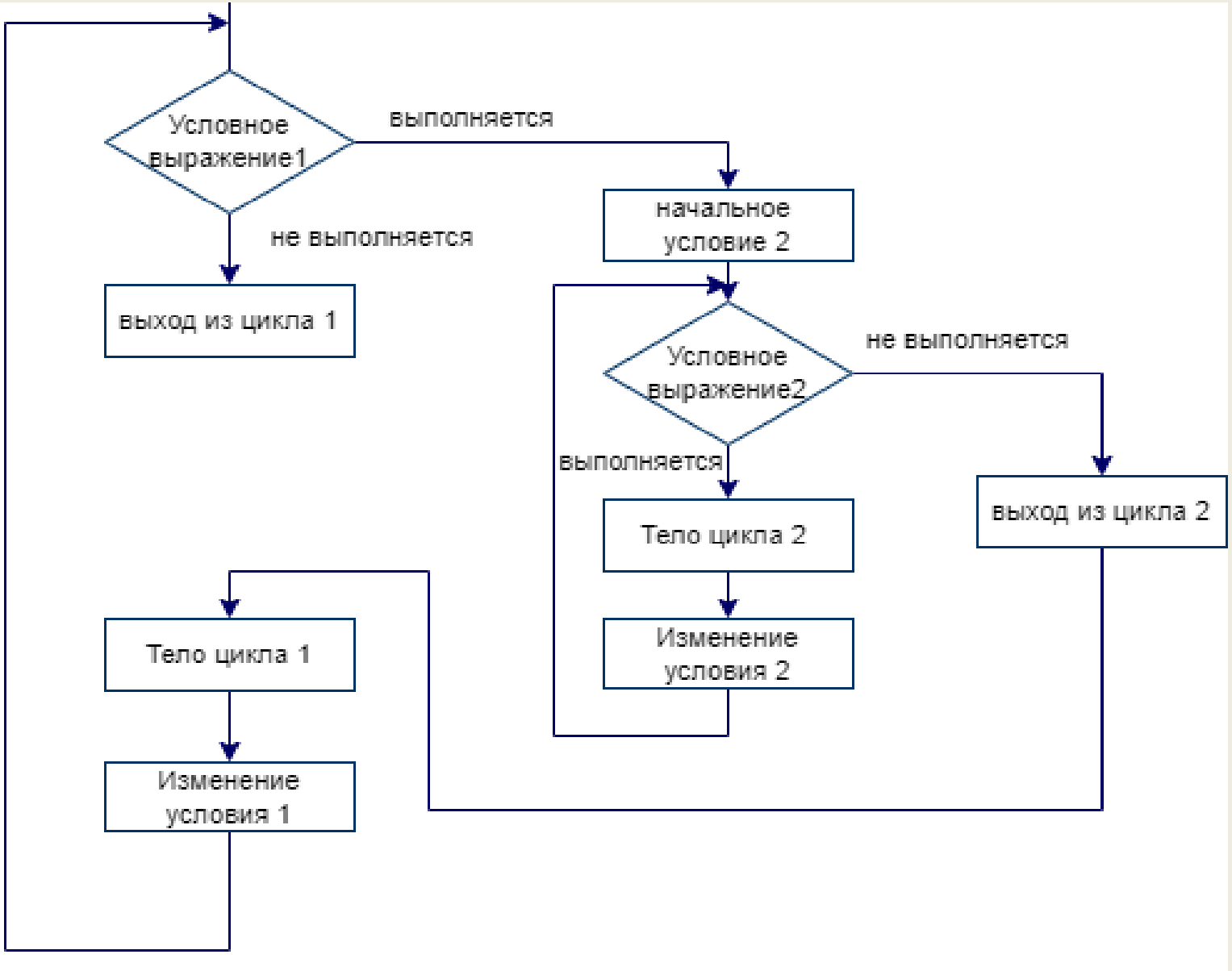
Цикл for – используется в тех случаях, когда некоторое действие необходимо выполнить определенное, известное перед его началом, число раз.

Цикл while – используется, когда количество повторений заранее не известно, и тело цикла должно выполняться, пока не будет достигнуто определенное условие.

Перебор последовательности – цикл с известным числом повторений

for <переменная> in <последовательность>:
    <тело цикла>

Вложенный цикл

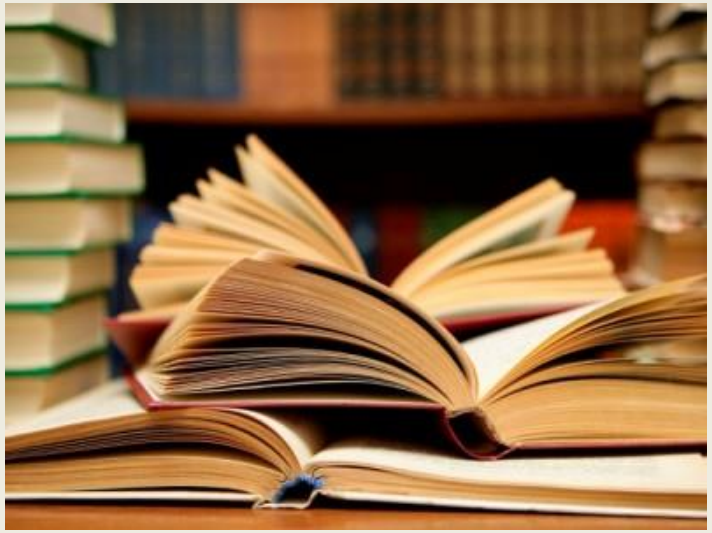



```
for i in range(10):
    for j in range(1,10):
        print(j , ‘в ’, i)
    print(i, ‘десяток’)
```

Повторять пока….

Давайте представим, что мы ходим найти любимую цитату в книге.

- Что мы будем делать?
- Перелистывать страницу за страницей.
- Сколько раз?
- Неизвестно. Пока не встретим нужную фразу.



Для решения подобной задачи в Python используется цикл while.

Повторять пока….

Цикл while – достаточно универсальный цикл, его условие записывается и проверяется до входа в тело цикла.

Циклический процесс завершается, когда условие цикла перестает быть истинным.

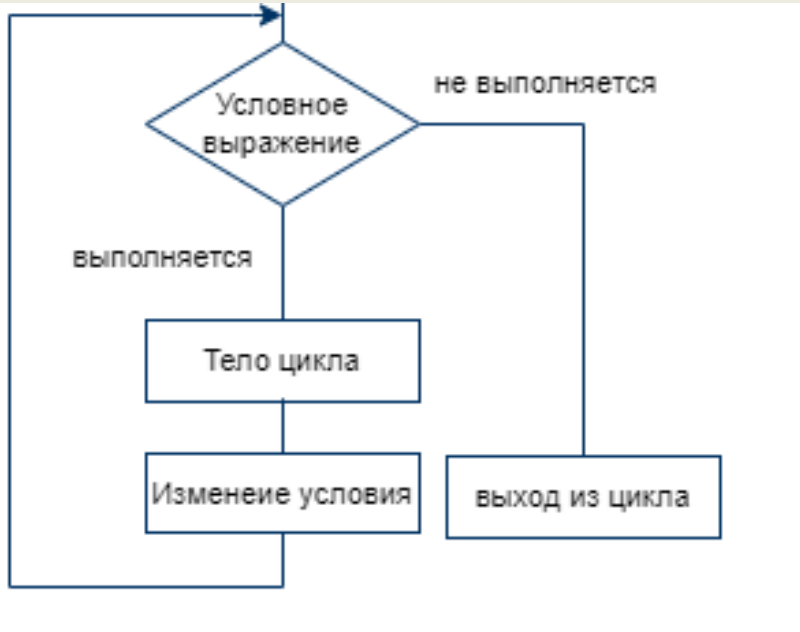
Такой цикл еще называют циклом с предусловием.

Цикл с предусловием

while <логическое выражение>:

<тело цикла>

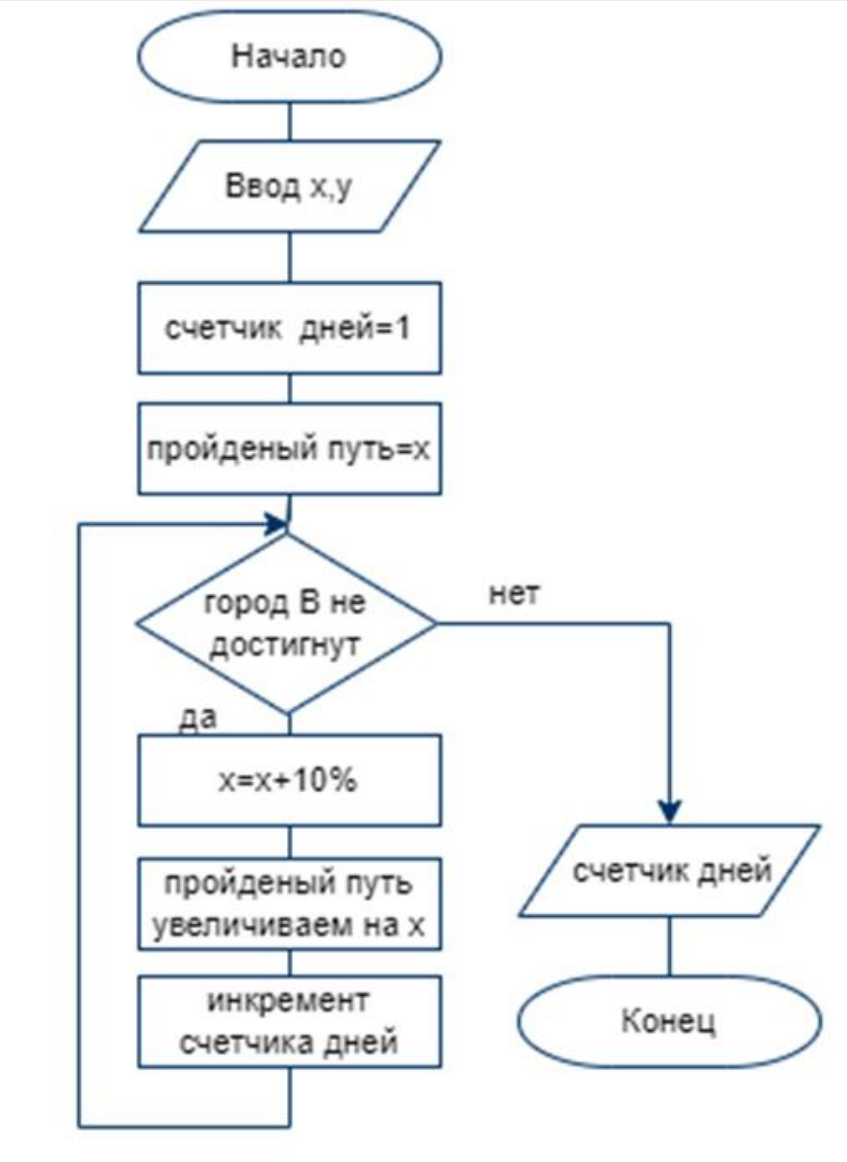
В теле цикла обязательно должно изменяться логическое выражение!



Давайте попробуем

Нам известно, что между городами A и B ровно y км. В первый день марафона спортсмен пробежал *x* километров, а затем он каждый день увеличивал пробег на 10% от предыдущего значения. Необходимо узнать через сколько дней спортсмен прибудет в город B.

Из города А и город В




```
way = x
while way<=y:
    x += x*0.1
    way += x
```

Задать начальное значение

Изменить условие

В каком случае циклический процесс не выполнится ни разу?

Допуск в систему

В нашей системе есть список зарегистрированных пользователей.

Необходимо разработать модуль аутентификации по паролю.

```
users={'boss':'123', 'bad': 'asd', 'good':'!@#'}
key= tuple(users.values())
<ввод пароля>
while <пароль> not in key:
        <дать попытку для ввода пароля>
```

НОД – алгоритм Евклида

Даны два натуральных числа. Определить их набольший общий делитель.

106 / 16 = 6, остаток 10

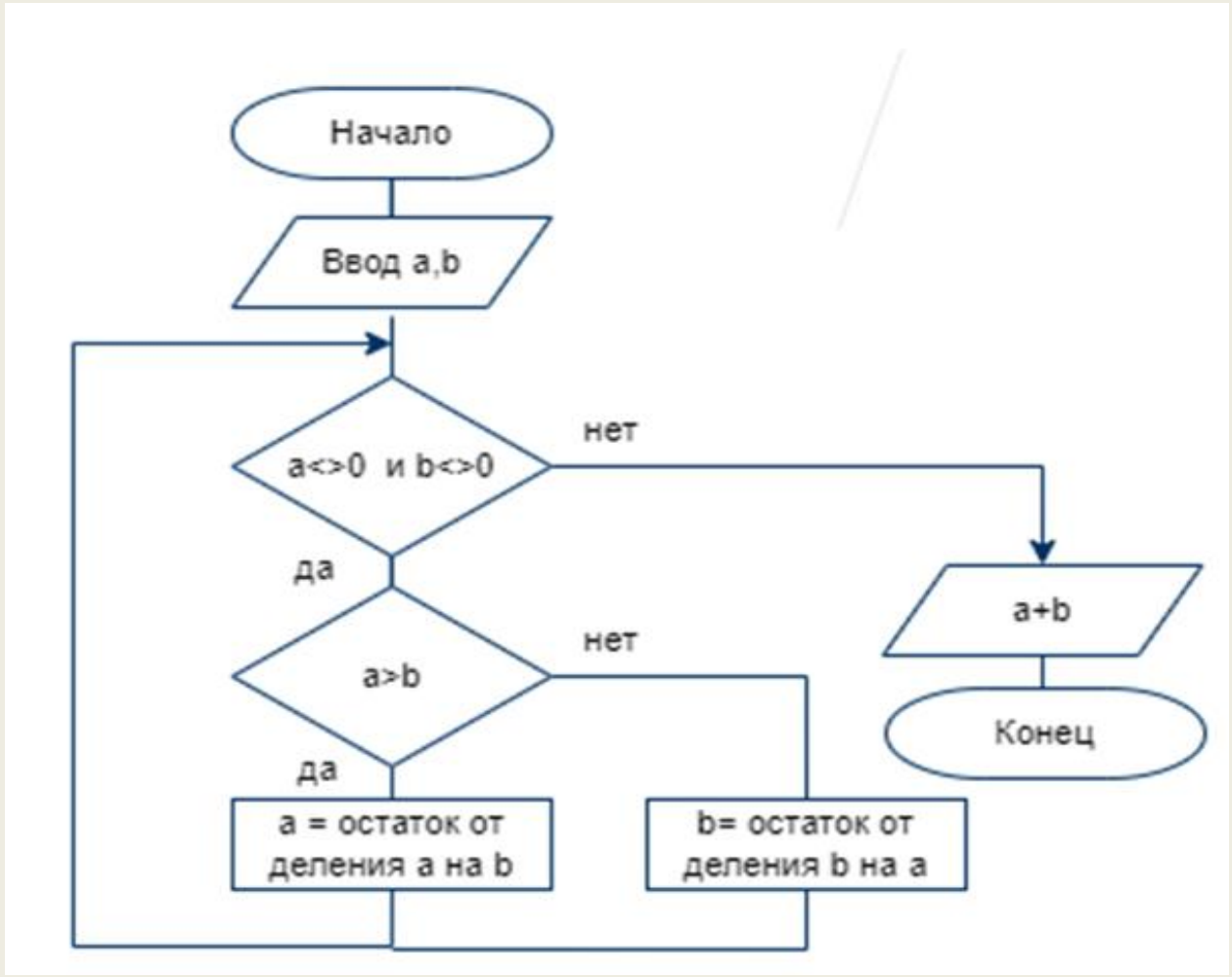
16 / 10 = 1, остаток 6

10 / 6 = 1, остаток 4

6 / 4 = 1, остаток 2

4 / 2 = 2, остаток 0

НОД



Пока стоит Земля

Бесконечными циклами в программировании называются те, в которых условие выхода никогда не выполняется.

Цикл while становится бесконечным, когда его условие не может быть ложным

Программа «купи слона».

Снова список дел

Расширим возможности:

- добавлять задачи на заданный день (день задается номером, вчера – 0, сегодня -1 и т.д.),
- определять на какой день запланирована задача.
- программа должна работать до тех пор, пока пользователь не введет команду END.
- перед началом работы предоставим пользователю инструкцию по доступным командам.

Список дел

```
while True:
        <ввод команды>
        if <команда> == 'ADD':
                tasks[day].append(task)
        elif <команда> == 'SHOW':
                for i in tasks[day]:
                        print(i)
        elif <команда> == 'FIND':
                for i in range(len(tasks)):
                        for j in tasks[i]:
                                if  j == task:
                                        print('Задача запланирована на день ', i)
        elif <команда> == 'END':
                break
        else:
                print('Такой команды нет')
```

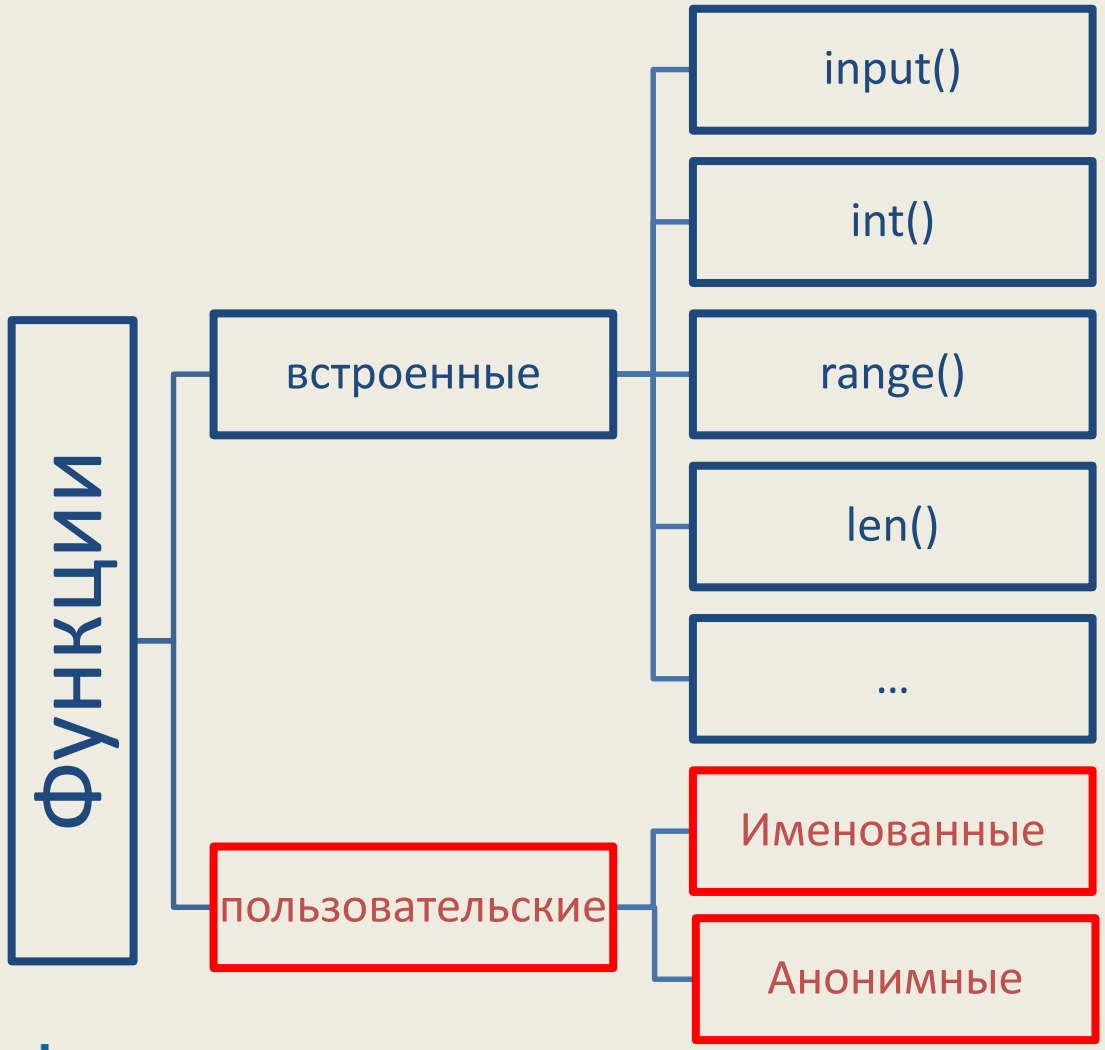
```
tasks = [['посмотреть фильм', 'сдать зачет'],
['навестить маму', 'купить книгу']]
```

Разделим, чтобы собрать

Существует способ разбить программу на обособленные части, каждая из которых выполняет свою конкретную подзадачу.

Функция - это обособленный именованный участок программного кода, предназначенный для решения какой-либо задачи, который может быть вызван в любом месте программы по имени, что позволяет повысить структурированность и избежать дублирования кода при его многократном использовании.

Классификация функций



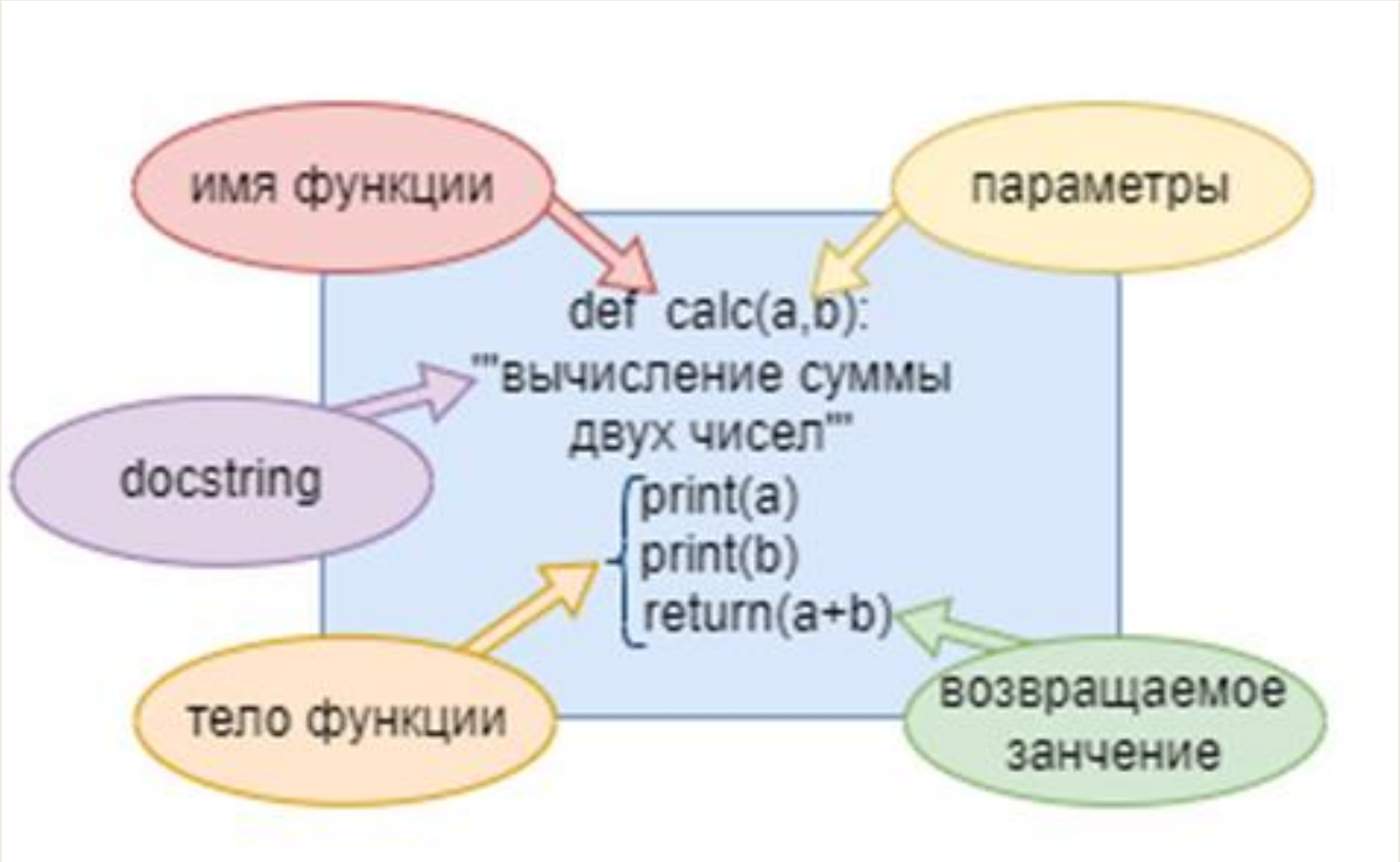
Код "спрятан внутри языка". Нам предоставляется только интерфейс - имя функции.

Пользовательские функции разработчики программ создают самостоятельно

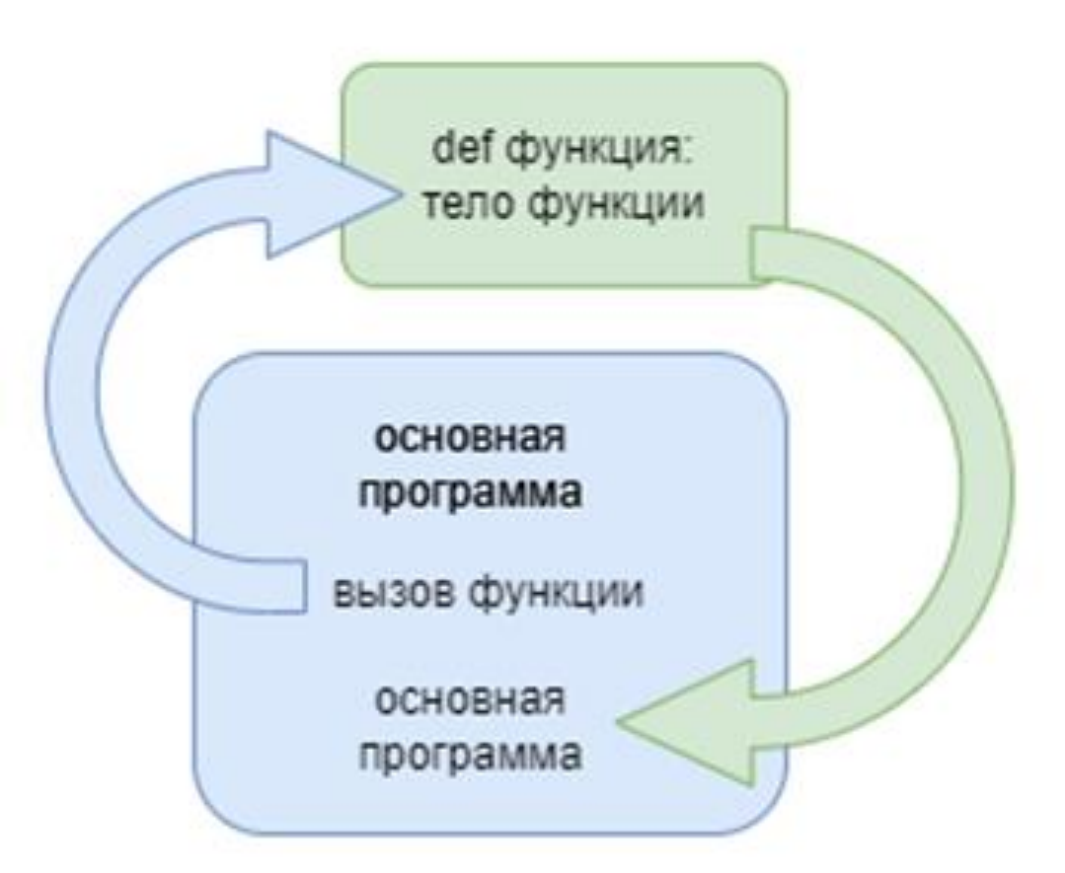
Зачем нужна пользовательская функция?

- Чтобы избежать дублирования кода.
- Чтобы обеспечить возможность повторного использования кода.
- Чтобы сократить время на написание кода и отладку, тем самым ускорить разработку программы.

Определение функции



Вызов функции



ВАЖНО!!

Определение функции должно предшествовать ее вызовам.

Функции придают программе структуру

```
def  info():
'''информация о командах программы '''
def add_task():
 '''запись новой задачи'''
def show_tasks():
'''просмотр задачи на заданный день'''
def  find_day():
 '''определение в какой день
запланирована задача'''
```

```
while True:
    command = input("Введите команду")
    if command == 'ADD':
            add_task()
    elif command == 'SHOW':
            show_tasks()
    elif command == 'FIND':
            find_day();
    elif command == 'END':
            break
    else:
            print('Такой команды нет')
```

Программа теперь состоит их отдельных кирпичиков, которые мы можем комбинировать в зависимости от решаемой задачи. Основная ветвь играет роль управляющего механизма.

Откуда все видно?

Область видимости переменной – это блок программы внутри которого переменная существует и доступна.

Время жизни – это промежуток времени, в течение которого переменная хранится в памяти.

Существует две области видимости: глобальная и локальная.

Локальные переменные

Локальные переменные видны и доступны только в рамках блока (функции) программы в котором они объявлены. Во время каждого вызова функции все ее локальные переменные инициализируются вновь. Значения таких переменных из предыдущих вызовов функция не сохраняет.

Глобальные переменные

Если требуется обратиться к переменной из любой части программы, следует сделать такую переменную глобальной

- объявить до вызова функции, если переменная внутри функции не изменяется
- указать ключевое слово global

Печать чека

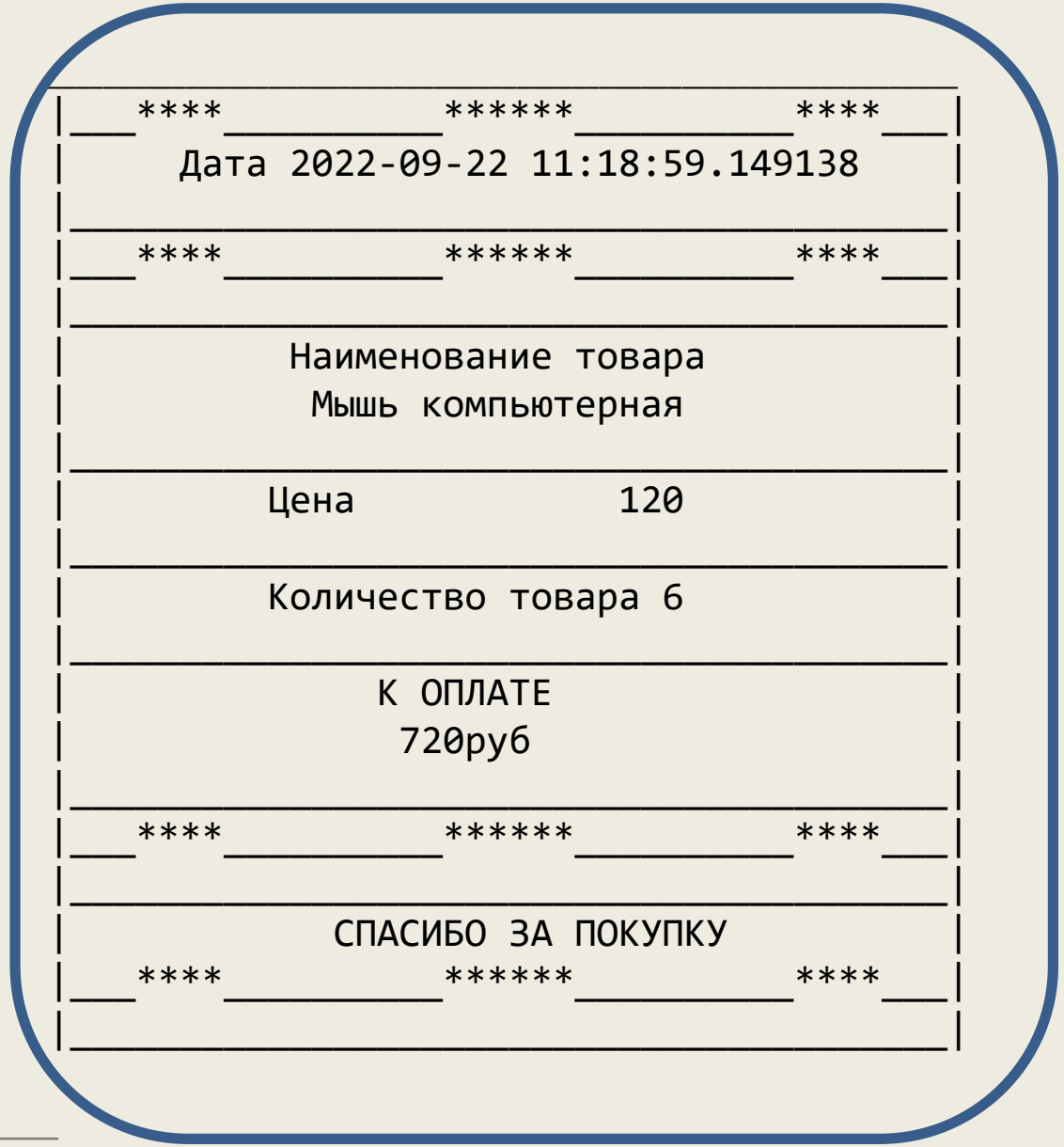
Написать программу для печати чека на покупку товара следующего вида:

```
|___****__________******__________****___|
```

```
print('|'+'_'*3+
          '*'*4+'_'*10+
          '*'*6+'_'*10+
          '*'*4+'_'*3+'|')
```

```
|_______________________________________|
```

```
print(']'+'_'*40+'|')
```

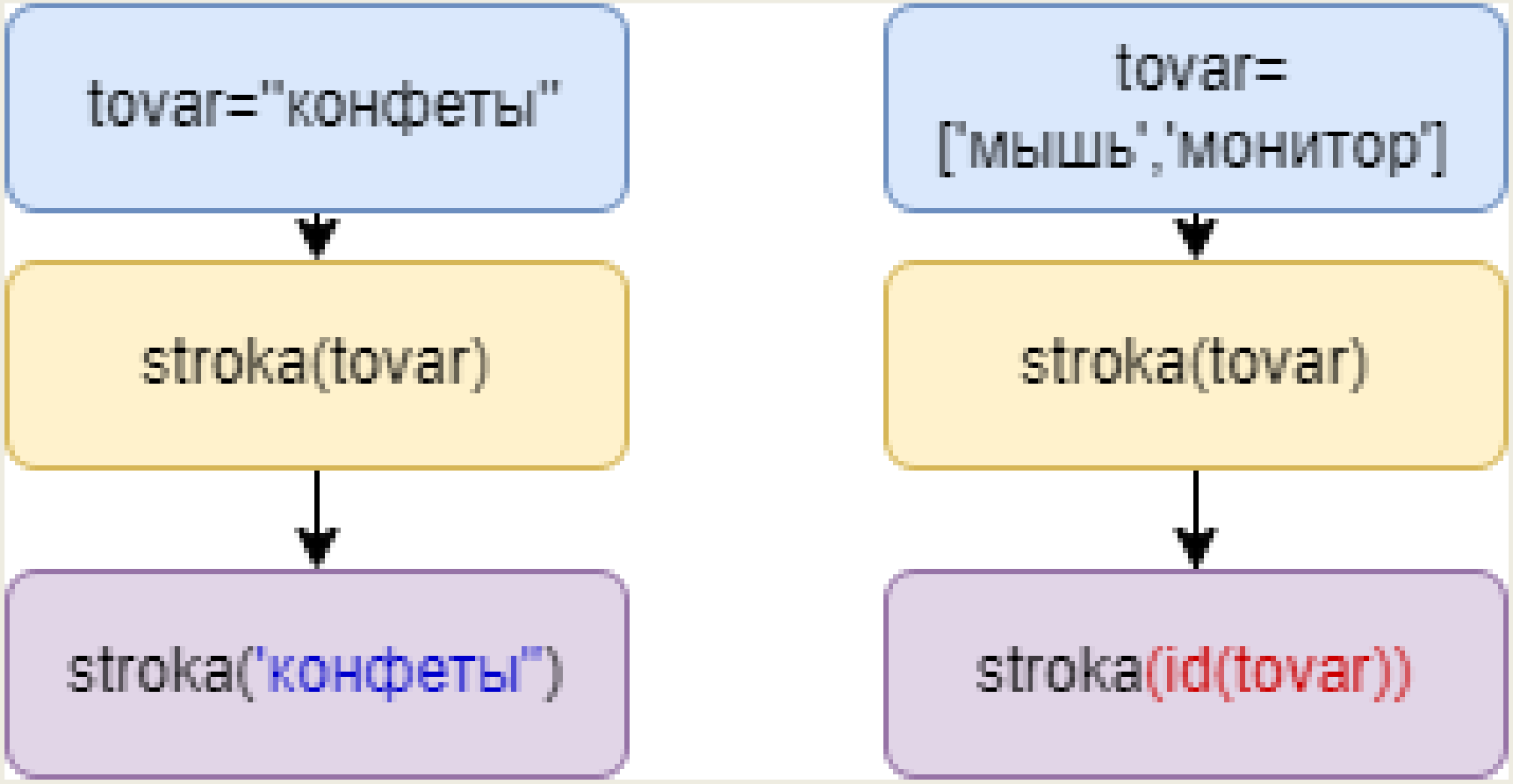


Параметры vs аргументы

stroka(s), которая имеет непустой список параметров, это позволяет нам каждый раз выводить в строку чека новое значение

Когда функция вызывается, интерпретатор перед началом выполнения ее кода присваивает переменным параметрам переданные в скобках значения, для неизменяемых типов, или ссылку на значение, в случае изменяемых.

Параметры vs аргументы



Параметры vs аргументы

Параметры являются локальными переменными, которым в момент вызова присваивается значение. Такие значения в Python называют аргументами, в других языках можно встретить понятие фактические параметры.

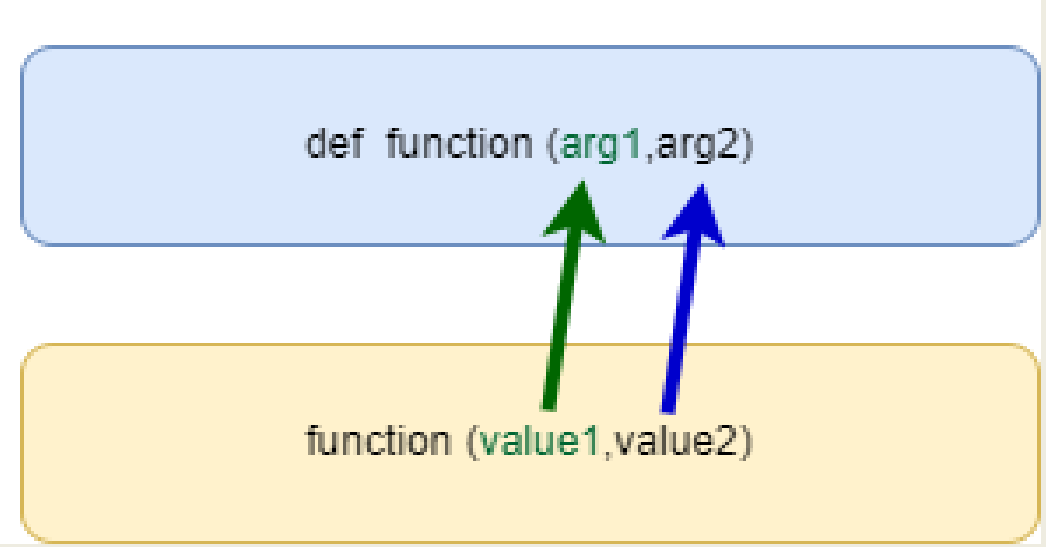
Аргументы используются для того чтобы ввести переменную в функцию.

Типы аргументов

- Обязательные (позиционные)
- Ключевые
- Аргументы по умолчанию

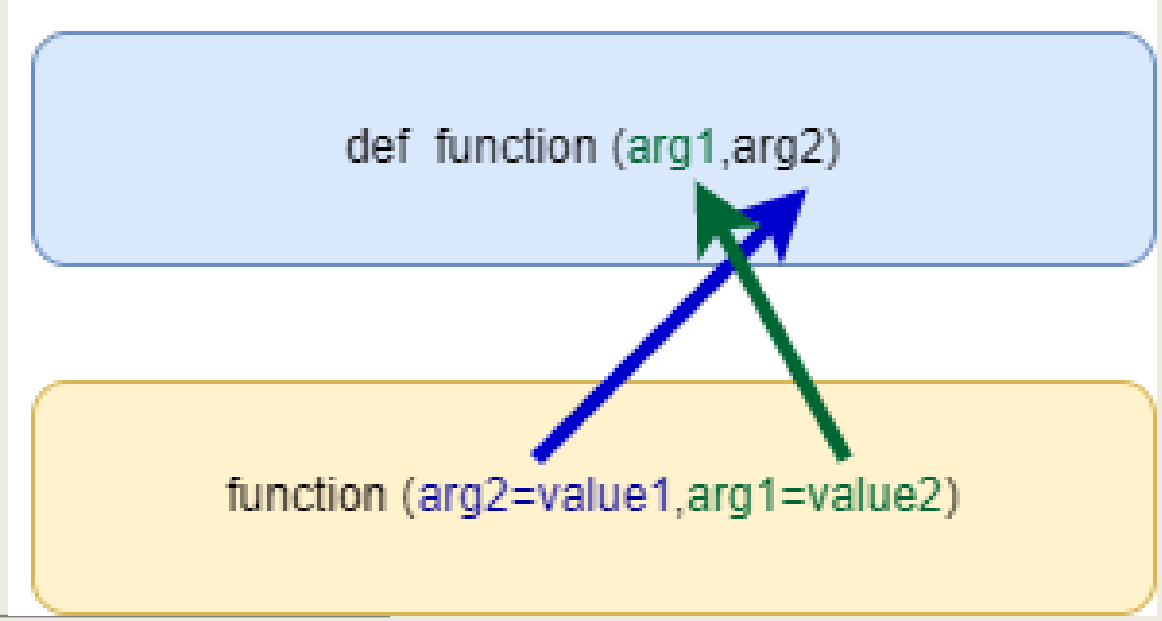
Параметры vs аргументы

Обязательные аргументы передаются в функцию в строгом позиционном порядке, их количество и порядок при вызове должен совпадать с описание функции



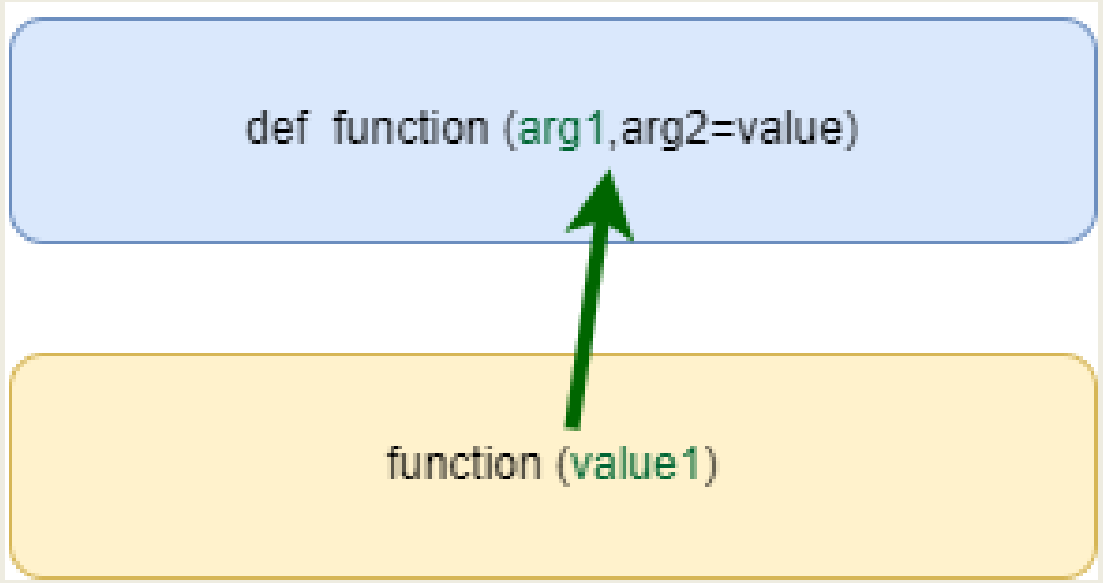
Параметры vs аргументы

Ключевые аргументы при вызове могут пописываться в любом порядке путем указания связки «имя_аргумента=значение».



Параметры vs аргументы

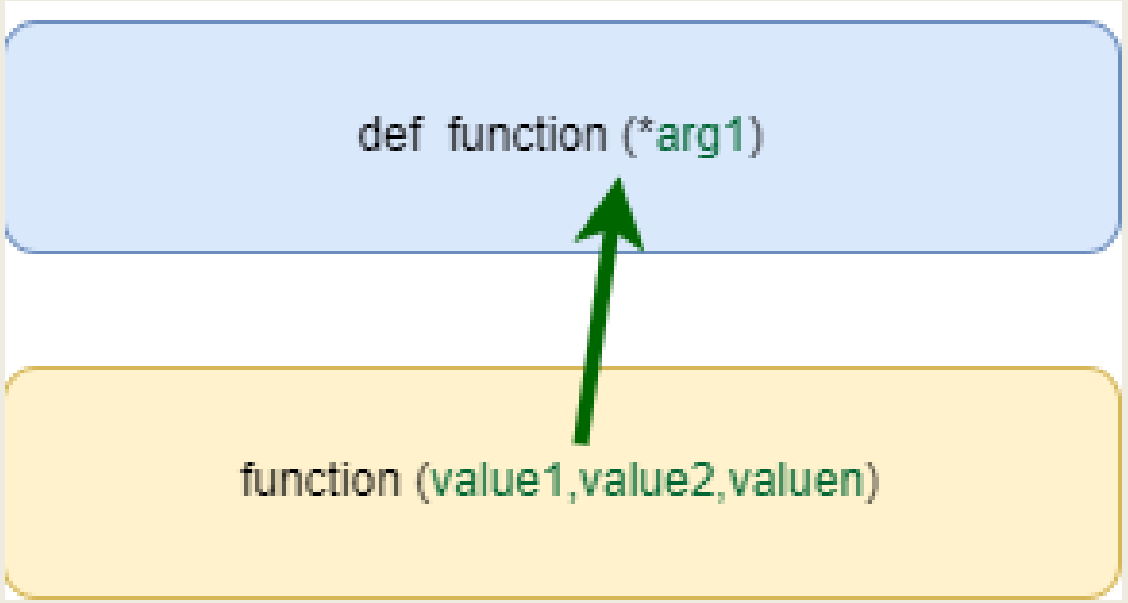
Аргумент по умолчанию может не указываться при вызове функции, значение будет взято из описания функции.



Замечание! Справа от аргумента со значением по умолчанию могут располагаться только такие же аргументы, имеющие значение по умолчанию. Обычные аргументы без значений по умолчанию должны находиться слева.

Параметры vs аргументы

Чтобы передать неизвестное число аргументов, достаточно перед именем аргумента поставит символ «*», тогда аргумент превратится в список.



Вернуть откуда взяли

С помощью оператора return, можно вернуть из функции одно или несколько значений.

Когда нужен return?

- Когда нужно вернуть результат работы функции в виде значения, которое может использовано вне функции.
- Когда нужно прервать работу функции и вернуться к той точке программы, из которой была вызвана эта функция.

Вернуть откуда взяли

```
def sum(a,b):
        return a+b
s = sum(a,b)


def division(a,d):
        if d == 0:
                return "Нельзя делить на ноль"
        return a / d

D= division(10,5) # результат 2
D= division(10,0)# результата “Нельзя делить на ноль”
```

Вернуть откуда взяли

С помощью оператора return, можно вернуть из функции одно или несколько значений.

Возможность такого группового присвоения – особенность Python, обычно не характерная для других языков. На самом деле перед возвратом из функции эти несколько значений упаковываются в кортеж (tuple).

Вернуть откуда взяли

Внутри любой функции может быть вызвана другая функция, причем она может быт как встроенной, так и пользовательской.

Создадим функцию для решения квадратного уравнения.

- Для нахождения корней нам потребуется вычислить дискриминант.
- Оформим поиск дискриминанта, как отдельную функцию.

У попа была собака

```
def short_story():
    print("У попа была собака, он ее любил.")
    print("Она съела кусок мяса, он ее убил,")
    print("В землю закопал и надпись написал:")
    short_story()
```

Рекурсивной называется функция, которая вызывает сама себя с измененными значениями аргументов.

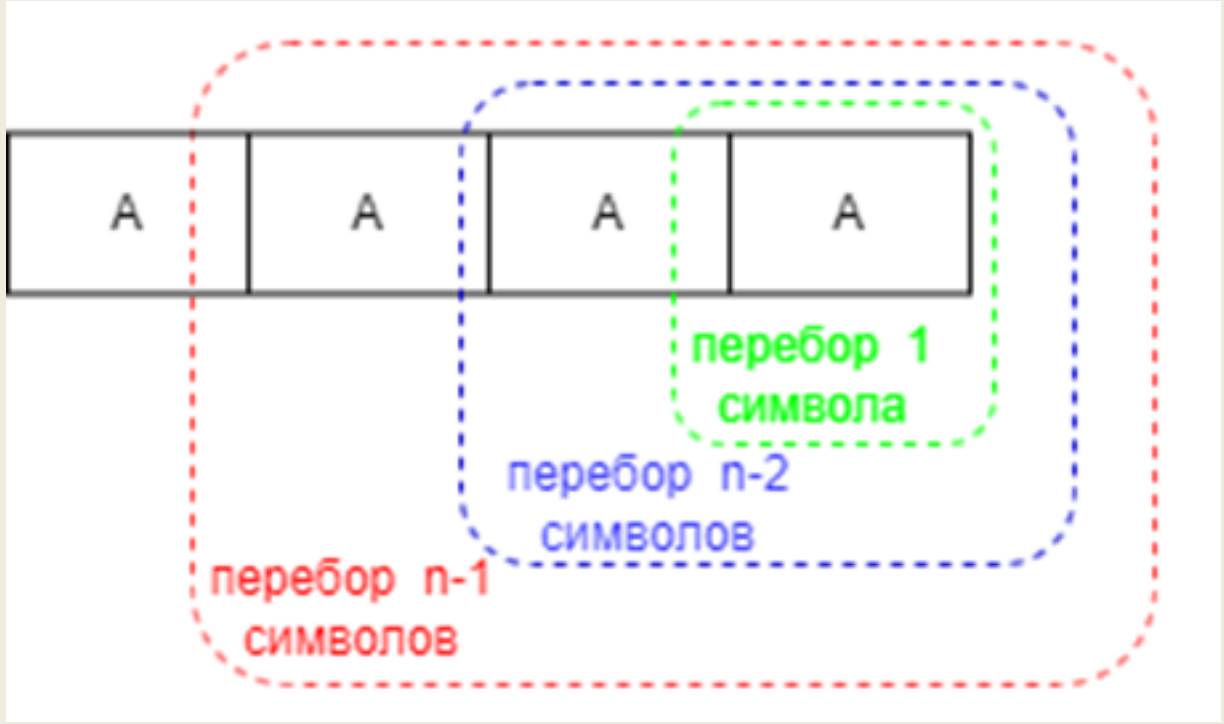
Рекурсия

Рекурсивные решение предполагают сведение поставленной задача к более простой, используя которую можно получить полное решение.

База рекурсии – простейший случай, при котором не происходит рекурсивного вызова, а происходим выход из текущей функции.

Давайте попробуем

В алфавите языка племени «Тумба-Юмба» всего четыре буквы: «А», «Б», «Р» и «О». Выведем на экран все слова, состоящие из N букв, которые могут употребить туземцы.

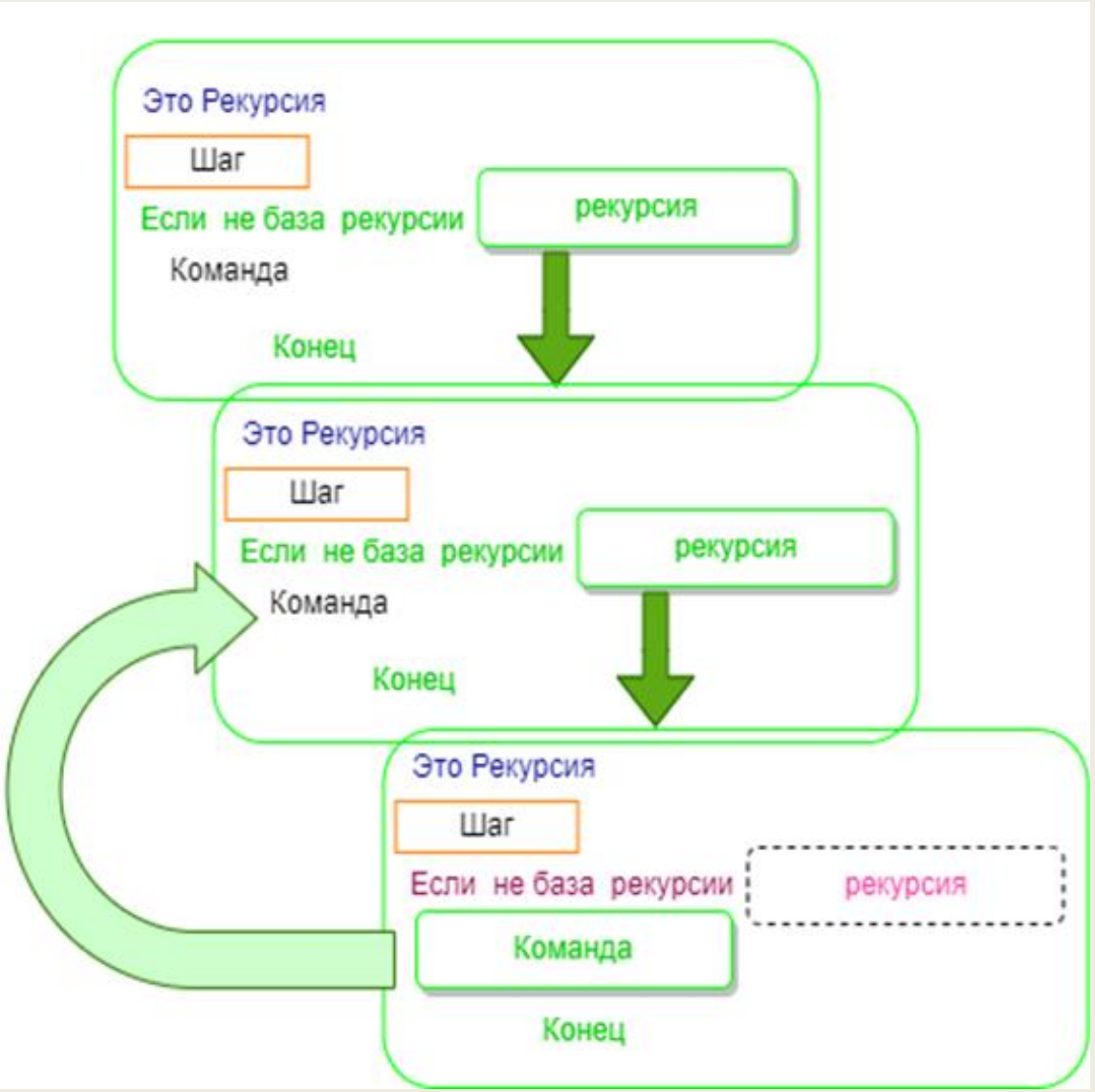


Давайте попробуем

```
def TumbaWords(word, alphabet, n):
    if n < 1:
        # база рекурсии
        print(word)
        return
    for c in alphabet:
#Рекурсивный вызов функции с измененными
значениями аргументов
        TumbaWords(word+c, alphabet, n - 1)

N = int(input())
TumbaWords("", "АБРО", N)
```

```
TumbaWords("", "АБРО", 3)
    TumbaWords("А", "АБРО", 2)
        TumbaWords("АА", "АБРО", 1)
            TumbaWords("ААА", "АБРО", 0)
```



Постепенно мы будем возвращаться функций. Подниматься вверх по цепочке вызов.

Возвращаемся к TumbaWords("АА", "АБРО", 1) и выполним следующий вызов, но уже с новым значением с в цикле

TumbaWords("ААБ", "АБРО", 0)

И так пока не завершится строка АБРО. А значит завершится цикл for.

Поднимаемся на уровень вверх И теперь следующий вызов:

TumbaWords("АБ", "АБРО", 1)

И так далее.

Можете посчитать сколько раз вызовется функция TumbaWords()?

Рекурсия

В Python имеется ограничение на максимальную глубину рекурсии. Но лимит можно увеличить если использовать функцию из библиотеки sys.

```
import sys
sys.setrecursionlimit(10**10)
```

Подведем итоги

- Функции очень важный и нужный элемент программы

Они позволяют структурировать код, избавиться от дублирования кода, предоставляют возможность повторного использования кода.

- Что будет получать функция для обработки решает механизм параметров

Аргументы при вызове подставляются на место параметров и могут быть обязательными, ключевыми, групповыми.

Могут использоваться аргументы по умолчанию.

- Если функция вызывает сама себя – она называется рекурсивной

Продолжение следует
Спасибо за внимание