Ходить вокруг, а не около

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

Снова о циклах

На предыдущей лекции мы познакомились с циклическими процессами и научились выполнять тело цикла заданное количество раз. Однако на практике бывает необходимо повторять некоторые действия сколь угодно долго – пока не выполнится какое-либо условие. Давайте представим, что мы ходим найти любимую цитату в книге. Саму цитату мы помним, а на какой странице она находится – нет.

- 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». «While» в переводе с английского языка – «до тех пор, пока». Это 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.

В общем случае цикл while выглядит так, как изображено на рисунке 1.

Рисунок 1 – Графическая интерпретация цикла while

В коде он будет изображен следующим образом.

while <логическое выражение>:

<тело цикла>

В теле цикла обязательно должно изменяться логическое выражение!

Давайте попробуем

Задача 1. Пусть нам известно, что между городами А и В ровно у км. В первый день марафона спортсмен пробежал х километров, а затем он каждый день увеличивал пробег на 10% от предыдущего значения. Необходимо узнать, через сколько дней спортсмен прибудет в город В? Схема алгоритма решения задачи представлена на рисунке 2.

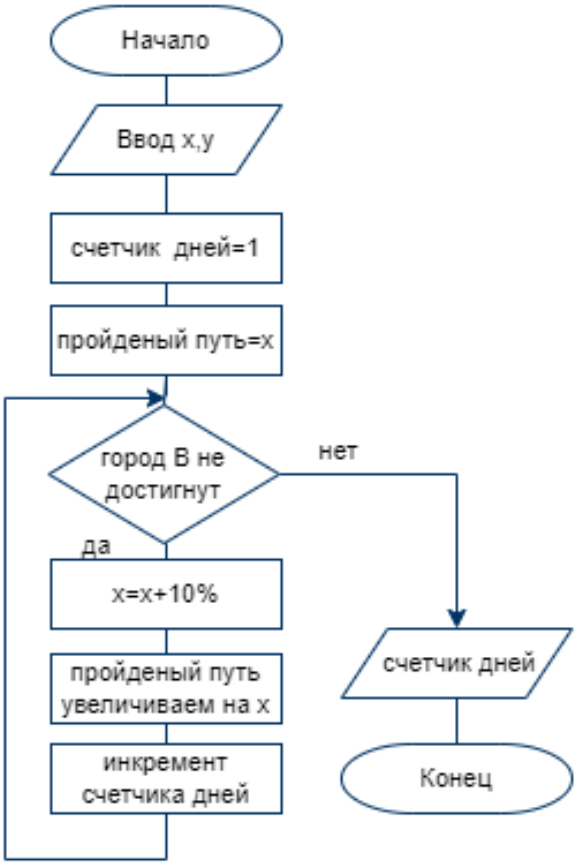

Рисунок 2 – Схема алгоритма решения задачи

Программа	На экране будет выведено

print('Введите через пробел расстояние между городами и величину первого участка пути')
y, x = map(int, input().split())
count = 1
way = x
while way <= y:
  x += x * 0.1
  way += x
  count += 1
	Введите через пробел расстояние между городами и величину первого участка пути
	100 10
	Спортсмен прибудет в город В на 8 день

print('Спортсмен прибудет в город B на ' , count, 'день')	

Внимание: если ввести значения 10 и 10, наш цикл не выполнится ни разу.

Задача 2. Есть список паролей пользователей некоторой системы. Давайте напишем программу допуска в систему только по верно введенному паролю.

Программа	На экране будет выведено

key=['123', 'asd', 'qwer', 'zxcv']
print('Пароль:')
password = input()
while password not in key:
  print('Не верно. Попробуйте еще раз.')
  password = input()
print('Добро пожаловать!')
	Пароль:
	55
	Не верно. Попробуйте еще раз.
	123
	Добро пожаловать!

Задача 3. Даны два натуральных числа. Определить их набольший общий делитель. Воспользуемся алгоритмом Евклида (рисунок 3).

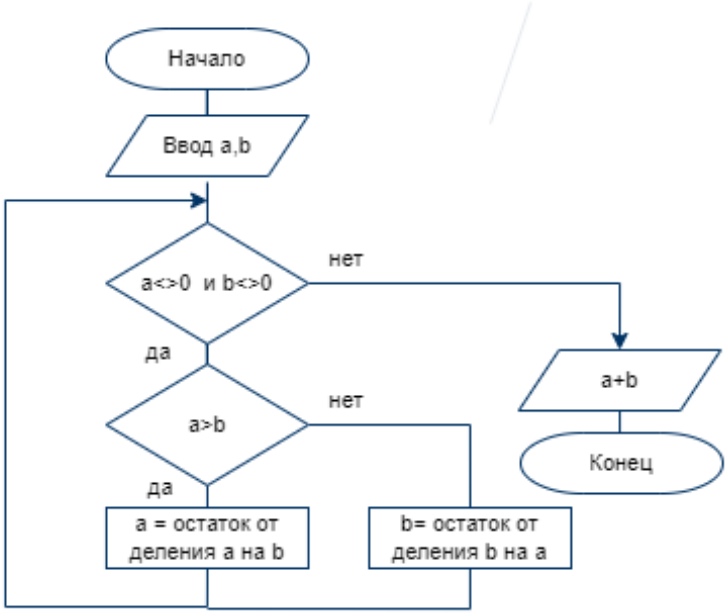

Рисунок 3 – Алгоритм Евклида

Программа	На экране будет выведено

a, b = map(int, input().split())
while a != 0 and b != 0:
  if a > b:
  a = a % b
  else:
  b = b % a
print(a + b)
	55 130
	5

Если условие, по которому нужно выйти из цикла, находится не в начале или конце цикла, а посередине или даже в нескольких местах его тела, можно воспользоваться бесконечным циклом в сочетании с инструкцией break.

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

Цикл while становится бесконечным, когда его условие не может быть ложным. Например, можно реализовать программу «Часы», которая бесконечно отображает время.

```
import time
from datetime import datetime

while True:
    cur_time = datatime.now()
    print(cur_time)
    time.sleep(1)
```

На предыдущих лекциях мы писали программу для ведения списка дел на вчера-сегодня-завтра. Давайте попробуем ее улучшить. Добавим возможность добавлять задачи на заданный день (день задается номером, вчера – 0, сегодня -1 и так далее) и определять, на какой день запланирована задача. Программа должна работать до тех пор, пока пользователь не введет команду «END». А еще перед началом работы предоставим пользователю инструкцию по доступным командам.

Программа	На экране будет выведено

print('Здравствуйте, я программа-планировщик')
print('У меня есть система команд',
'Введите ADD, если хотите добавить задачу',
'SHOW, если хотите посмотреть задачи на день',
'FIND, чтобы узнать день, на который запланирована задача')
tasks = []
while True:
command = input("Введите команду")
if command == 'ADD':
day = int(input('Выберите день'))
task = input('Задайте задачу')
if day <= len(tasks) - 1:
tasks[day].append(task)
else:
for i in range(len(tasks) - 1,day):
tasks.append([])
tasks[len(tasks)-1].append(task)
if command == 'SHOW':
day = int(input('Выберите день'))
if day <= len(tasks) - 1:
for i in tasks[day]:
print(i)
else:
	Здравствуйте, я программа-планировщик
	У меня есть система команд.
	Введите ADD, если хотите добавить задачу
	SHOW, если посмотреть задачи на день
	FIND, чтобы узнать день, на который запланирована задача
	Введите команду
	ADD
	Выберите день
	1
	Задайте задачу
	Тест
	Введите команду
	SHOW

```
                print('На этот день нет планов')
        if command == 'FIND':
                flag = 0
                task = input('Введите задачу для поиска')
                for i in range(len(tasks)):
                        for j in tasks[i]:
                                if j == task:
                                        print('Задача запланирована на день ', i)
                                        flag = 1
                                        break
                        if flag == 1:
                                break
                else:
                        print('Такая задача не запланирована')
        if command == 'END':
                break
        else:
                print('Такой команды нет')
```

```
Выберите день
1
Тест
Введите команду
ADD
Выберите день
0
Задайте задачу
кино
Введите команду
SHOW
Выберите день
0
кино
Введите команду
FIND
Введите задачу для поиска
кино
Задача запланирована на день 0
Введите команду
FIND
Введите задачу для поиска
музей
Такая задача не запланирована
Введите команду
STOP
Такой команды нет
Введите команду
END
```

Разделим, чтобы собрать

Наша программа работает, но выглядит неструктурно и отлаживать ее будет очень неудобно. Существует способ разбить программу на обособленные части, каждая из которых выполняет свою конкретную подзадачу. Такие фрагменты называют подпрограммами (функциями).

> 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.

Все функции в языках программирования можно классифицировать на встроенные и пользовательские.

Мы уже сталкивались с некоторыми встроенными в Python функциями: print(), input(), int(), range(). Их код нам не виден, он «спрятан внутри языка». Нам же предоставляется только интерфейс – имя функции.

Пользовательские функции разработчики программ создают самостоятельно. Зачем нужны пользовательские функции?

– Чтобы избежать дублирования кода. Допустим, мы разрабатываем систему авторизации и регистрации, тогда нам потребуется, как минимум, два раза вызвать функцию шифрования пароля, иначе придется дважды прописывать цикл для работы с каждым символом, а затем дважды создавать ключ для шифрования.

– Чтобы обеспечить возможность повторного использования кода. Единожды прописав алгоритм выполнения нужных операций, можно множество раз обращаться к нему, но уже с разными значениями.

– Чтобы сократить время на написание кода и отладку, тем самым ускорить разработку программы. Программисты, работающие над большим общим проектом, могут разделить рабочую нагрузку и занимаясь разными функциями.

Пользовательские функции делятся на именованные и анонимные. В этом модуле мы будем рассматривать только именованные.

Определение функции.

– Заголовок функции должен начинаться с ключевого слова def и обязательно заканчиваться двоеточием.

– Функция должна иметь уникальное имя.

– Для обработки можно передать произвольное количество параметров. Но это необязательно.

– Желательно документировать функцию, то есть между тройными кавычками написать docstrings (что делает функция).

– Тело функции – это набор инструкций, отвечающий за логику ее работы.

– С помощью инструкции return можно вернуть результат работы функции, но это необязательно.

На рисунке 4 схематично выделены различные элементы функции.

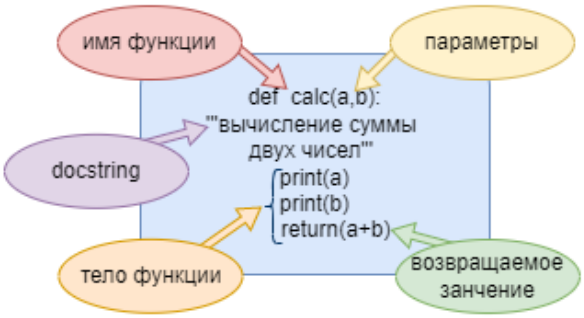

Рисунок 4 – Элементы функции

Вызов функции выглядит просто как упоминание ее имени со скобками. Если в функцию мы ничего не передаем – скобки пустые. Когда функция вызывается, поток выполнения программы переходит к ее определению и начинает исполнять ее тело. После того, как тело функции исполнено, поток возвращается в основной код, в точку вызова. Далее исполняется следующее за вызовом выражение. Схематично это показано на рисунке 5.

Определение функции должно предшествовать ее вызовам. Это связано с тем, что интерпретатор читает код строка за строкой и о том, что находится ниже, ему еще неизвестно. Поэтому возникает ошибка.

Функции придают программе структуру. Давайте приведем наш планировщик в структурный вид.

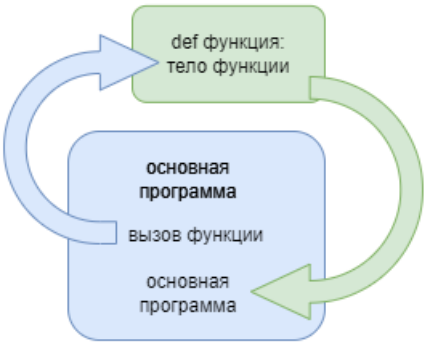

Рисунок 5 – Вызов функции

```
def info():
        #информация о командах программы
        print('Здравствуйте, я программа-планировщик')
        print('У меня есть система команд', 'Введите ADD, если хотите добавить задачу ',
'SHOW, если посмотреть задачи на день', 'FIND, чтобы узнать день, на который
запланирована задача')

def add_task():
        #запись новой задачи
        day = int(input('Выберите день'))
        task = input('Задайте задачу')
```

```
        if day <= len(tasks) - 1:
                tasks[day].append(task)
        else:
                for i in range(len(tasks) - 1, day):
                        tasks.append([])
                tasks[len(tasks) - 1].append(task)
        print(tasks)

def show_tasks():
        '''просмотр задачи на заданный день'''
        day = int(input('Выберите день'))
        if day <= len(tasks) - 1:
                for i in tasks[day]:
                        print(i)
        else:
                print('На этот день нет планов')

def find_day():
        '''определение в какой день запланирована задача'''
        flag = 0
        task = input('Введите задачу для поиска')
        for i in range(len(tasks)):
                for j in tasks[i]:
                        if j == task:
                                print('Задача запланирована на день ', i)
                                flag = 1
                                break
                if flag == 1:
                        break
        else:
                print('Такая задача не запланирована')

tasks = []
info()
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. Если будет необходимо использовать эти функции в другой программе, то достаточно импортировать их, сославшись на данных файл.

Откуда все видно?

Области видимости очень важное понятие любого языка программирования, они позволяют использовать одно и то же имя переменных в разных местах программы и делают программу более безопасной.

> 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 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.

Программа	На экране будет выведено
def fun(): x = 1000 print('локально', x) y = 1111 print('локально', y) x = 2000 fun() print('глобально', x) print('глобально', y)	локально 1000 локально 1111 глобально 2000 Traceback (most recent call last): File "deffunprintxx1000printxx2000funprintxUntitled2.py", line 9, in <module> print('глобально',y) NameError: name 'y' is not defined

Получается, что внешняя часть программы ничего не знает о переменной y, которая содержится в области видимости функции, при попытке обращения к переменной интерпретатор выдает ошибку.

Если требуется обратиться к переменной из любой части программы, следует сделать такую переменную глобальной, для этого ее необходимо объявить до вызова функции.

Программа	На экране будет выведено
def fun(): print('локально', x) y = 1111 print('локально', y) x = 2000 y = 2222 fun() print('глобально', x) print('глобально', y)	локально 2000 локально 1111 глобально 2000 глобально 2222

Однако это работает только в случае, если переменная внутри функции не изменяется. Происходит это потому, что с точки зрения интерпретатора у локальный и у глобальный являются разными переменными. Исправить ситуацию можно, используя ключевое слово global.

Программа	На экране будет выведено
def fun(): print('локально', x) global y y = 1111 print('локально', y) x = 2000 y = 2222 fun() print('глобально', x) print('глобально', y)	локально 2000 локально 1111 глобально 2000 глобально 1111

Задача 4. Необходимо написать программу для печати чека на покупку товара следующего вида.

```
_____________________________________
|___****_____******_____****___|
| Дата 2022-09-22 11:18:59.149138 |
|_____________________________________|
|___****_____******_____****___|
|_____________________________________|
| Наименование товара |
| Мышь компьютерная |
|_____________________________________|
| Цена 120 |
|_____________________________________|
| Количество товара 6 |
|_____________________________________|
| К ОПЛАТЕ |
| 720руб |
|_____________________________________|
|___****_____******_____****___|
|_____________________________________|
| СПАСИБО ЗА ПОКУПКУ |
|___****_____******_____****___|
|_____________________________________|
```

Явно видно, что на рисунке множество однотипных элементов, для отображения которых можно использовать функции.

Программа	На экране будет выведено
from datetime import datetime def in_chek(): print ('Информация о покупке: товар, цена, количество ') global tovar, cost, count tovar = input() cost = int(input()) count = int(input())	Информация о покупке: товар, цена, количество Мышь компьютерная 120 6 _____

```
def linen():
        print('_' * 42)  print('|' + '_' * 3 + '*' * 4 + '_' * 10 + '*' * 6 + '_' * 10 + '*' * 4 + '_' * 3 + '|')

def linek():
        print('|' + '_' * 3 + '*' * 4 + '_' * 10 + '*' * 6 + '_' * 10 + '*' * 4 + '_' * 3 + '|')
        print(']' + '_' * 40 + '|')

def line_1():
        print('|' + '_' * 40 + '|')

def line_2():
        print('|' + '_' * 40 + '|')
        print('|' + '_' * 3 + '*' * 4 + '_' * 10 + '*' * 6 + '_' * 10 + '*' * 4 + '_' * 3 + '|')
        print('|' + '_' * 40 + '|')

def stroka(s):
        print('|' + ' ' * (20 - len(s) // 2) + s + ' ' * (20 - len(s) // 2 - len(s) % 2) + '|')

tovar = ''
cost, count = 0, 0
in_chek()
linen()
stroka('Дата '+ str(datetime.now()))
line_2()
stroka('Наименование товара')
stroka(tovar)
line_1()
stroka('Цена ' + str(cost))
line_1()
stroka('Количество товара ' + str(count))
line_1()
stroka('К ОПЛАТЕ')
stroka(str(count * cost) + ' руб')
line_2()
stroka('СПАСИБО ЗА ПОКУПКУ')
linek()
```

```
|___****_____******_____****___|
| Дата 2022-09-22 11:18:59.149138 |
|_____________________________________|
|___****_____******_____****___|
|_____________________________________|
| Наименование товара |
| Мышь компьютерная |
|_____________________________________|
| Цена 120 |
|_____________________________________|
| Количество товара 6 |
|_____________________________________|
| К ОПЛАТЕ | | 720руб |
|_____________________________________|
|___****_____******_____****___|
|_____________________________________|
| СПАСИБО ЗА ПОКУПКУ |
|___****_____******_____****___|
]_____________________________________|
```

Среди функций нашей программы есть stroka(s), которая имеет непустой список параметров, это позволяет нам каждый раз выводить в строку чека новое значение.

Как это происходит? 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.

То есть когда мы вызывали функцию stroka(tovar), при этом выполняли объявление tovar = 'usb' (неизменяемый тип), на самом деле выполнялся следующий вызов stroka('usb'), а в случае tovar = ['мышь', 'клавиатура'] – stroka(id(tovar)).

Таким образом, 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.

Функцию можно вызвать, используя следующие типы аргументов.

- Обязательные (позиционные).
- Ключевые.
- Аргументы по умолчанию.

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

Программа	На экране будет выведено

def login(log, passw):
#вход по имени и паролю
print('логин', log)
print('пароль', passw)
name = input('Введите имя')
password = input('Введите пароль')
login(password, name)
	Введите имя
	User
	Введите пароль
	1234
	логин 1234
	пароль User

Не всегда удобно передавать аргументы в той последовательности, в которой они указаны описании функции. В этом случае удобно использовать ключевые или именованные аргументы.

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

Программа	На экране будет выведено

def ranger(min, max):
#генератор диапазона
ranger_list = range(min, max + 1)
print(list(ranger_list))
 	
d = ranger(min = 4, max = 8)
d = ranger(max = 8, min = 4)
	[4, 5, 6, 7, 8]
	[4, 5, 6, 7, 8]

Используя ключевые аргументы, вы точно знаете куда были переданы данные и делаете код более понятным для других разработчиков.

Существуют ситуации, когда одному из аргументов нужно задавать значение по умолчанию, которое сохранится, пока не будет передано другое значение.

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

Программа	На экране будет выведено
def say_goodbye(name, msg = "Сладких снов!"): print("До свидания,", name + '.', msg) say_goodbye('друг') say_goodbye('друг', 'Была рада встрече')	До свидания, друг. Сладких снов! До свидания, друг. Была рада встрече

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.

Если мы во время разработки программы точно не знаем, сколько переменных будет отправлено на обработку, можно воспользоваться аргументами переменной длины.

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

Допустим, мы должны написать функцию, которая составляет координаты точек. Причем, отправить в нее можно бесконечное множество координат, например многоугольника.

Программа	На экране будет выведено

def figure(*args):
#построение пар координат
i = 0
string_coords = ""
while (i < len(args)):
if(i % 2 == 0):
string_coords += "(x=" + str(args[i]) + ","
else:
string_coords += "y="+str(args[i]) + ") "
i += 1
print (string_coords)
figure(3, 5, 1, 5, 7, 4, 2, 2)
	(x=3,y=5) (x=1,y=5) (x=7,y=4) (x=2,y=2)

Вернуть откуда взяли

Особенностью функции является то, что она способна не только выполнить какой-то набор выделенных в отдельный блок инструкций, но и передать результаты своей работы в точку, откуда она была вызвана. Говорят – «функция возвращает значение».

> 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 – например, записано во внешнюю переменную.

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

Когда нужно прервать работу функции и вернуться к той точке программы, из которой была вызвана эта функция. Появление return в коде обозначает завершение функции, вне зависимости от наличия дальнейших инструкций. В функции может быть несколько операторов return, но в каждый запуск выполняется только один из них.

```
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”
```

В Python можно возвращать из функции несколько значений, перечислив их через запятую после return.

Программа	На экране будет выведено

def cylinder(r, h):
#вычисление площадей цилиндра
side = 2 * 3.14 * r * h
circle = 3.14 * r ** 2
full = side + 2 * circle
return side, full
 	
sc, fsc = cylinder(float(input()), float(input()))
print("Площадь боковой поверхности %.2f" % sc)
print("Полная площадь %.2f" % fsc)
	10
	10
	Площадь боковой поверхности 628.00
	Полная площадь 1256.00

Мы вернули два значения: первое из них присваивается переменной sc, а второе – fsc. 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).

Внутри любой функции может быть вызвана другая функция, причем она может быт как встроенной, так и пользовательской. Пусть мы хотим создать функцию для решения квадратного уравнения. Как известно, для нахождения корней нам потребуется вычислить дискриминант. Оформим поиск дискриминанта, как отдельную функцию.

Программа	На экране будет выведено

import math
def SquareRoot(a, b, c):
#функция для решения квадратного уравнения
D = Discr(a, b, c)
if (D < 0):
return
#уравнение не имеет решения, возвращается None
else:
x1 = (-b - math.sqrt(D)) / (2 * a)
x2 = (-b + math.sqrt(D)) / (2 * a)
return x1, x2
 	
def Discr(a, b, c):
#функция, которая вычисляет дискриминант
return b * b - 4 * a * c
 	
a, b, c = map(float, input().split())
roots = SquareRoot(a, b, c)
if roots != None:
print("x1 = ", roots[0])
print("x2 = ", roots[1])
else:
print("Действительных корней нет")
	2 1 -1
	x1 = -1.0
	x2 = 0.5

Но функция может вызывать и саму себя! Такая функция называется рекурсивной.

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

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

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.

Такую задачу перебора можно решить, сведя ее к задаче меньшего размера. Будем последовательно подставлять буквы к слову. Так, на первой позиции слова может стоять одна из 4-х букв алфавита (А, В, Р, О). Предположим, что сначала первой поставим букву 'А'. Тогда для того, чтобы получить все варианты с первой буквой 'А', нужно перебрать все возможные комбинации букв на оставшихся N - 1 позициях, затем на N - 2 и так далее, пока не будет подставлена последняя буква (простейший случай).

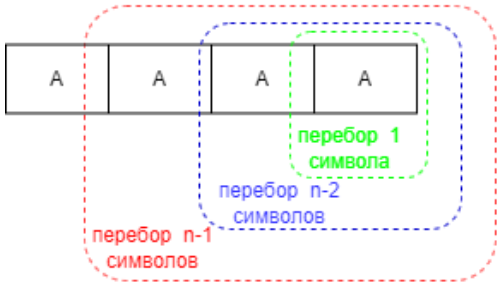

Рисунок 6 – Процесс перебора

Таким образом, мы видим возможность рекурсивного решения: в цикле перебрать все возможные первые буквы (ставя по очереди каждую букву алфавита на первое место) и для каждого случая построить все возможные «хвосты» длиной N - 1 и так далее.

```
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)
```

Что происходит? При первом вызове функции мы передаем в нее пустую строку, так как еще нет сформированных слов, строку, содержащую допустимый набор символов, и длину слов (например, 3), которая нас интересует. Заходим в тело функции и постепенно начинаем выполнять вызовы той же функции. Переменная с содержит текущий символ из допустимой строки. Тогда следующий вызов будет с аргументами TumbaWords("А", "АБРО", 2) и мы снова войдем в функцию. Говорят,

что «образуется рекурсивная цепочка вызовов». Длина цепочки определяет глубину рекурсии.

TumbaWords("", "АБРО", 3)

TumbaWords("А", "АБРО", 2)

TumbaWords("АА", "АБРО", 1)

TumbaWords("ААА", "АБРО", 0)

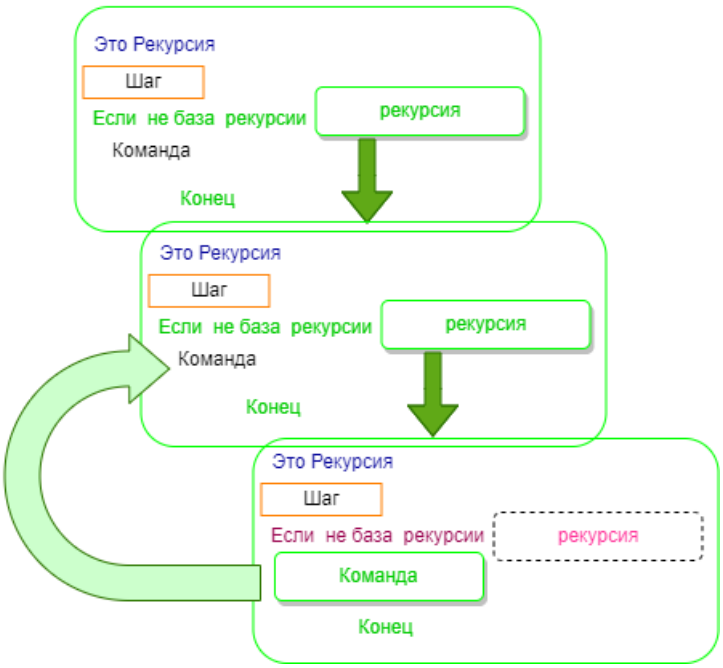

Рисунок 7 – Процесс рекурсивных вызовов

Первое слово получено, и оно выведется на экран за счет входа в базу рекурсии.

Теперь постепенно мы будем возвращаться из наших функций. Говорят – «подниматься вверх по цепочке». 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 и не 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()?

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

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

Мы подключили функцию из внешнего модуля. Мы уже делали это, когда вызывали функцию генерации случайного числа и функцию получения квадратного корня. Если в будущем вы захотите использовать свои функции во вновь разрабатываемых вами программах, то вам достаточно будет поместить их в отдельный файл, файл разместить в вашу текущую папку, импортировать этот файл, так же, как и стандартные файлы языка.

Практическая работа

1) Напишите программу, в которой вызывается функция, запрашивающая от пользователя две строки и возвращающая в программу результат их конкатенации. Выведите результат на экран.

2) Напишите функцию, которая считывает с клавиатуры числа и перемножает их до тех пор, пока не будет введен 0. Функция должна возвращать полученное произведение. Вызовите функцию и выведите на экран результат ее работы.

Контрольные вопросы

1) Где вы будете использовать while вместо for?
2) Является ли функция допустимой, если она не имеет оператора return?
3) Почему инструкцию while называют «циклом с предусловием»?
4) Для чего используется инструкция break?
5) Что такое локальные и глобальные переменные?
6) В чем особенность использования ключевых аргументов?
7) Какая функция называется пользовательской?
8) Опишите процесс создания функций в Python.
9) Что такое рекурсивная цепочка?
10) Как организовать бесконечный цикл? Зачем он может потребоваться?