1. Лекция 4. Функции. Рекурсия

1.1. Функции

Давайте вспомним, что такое функция

Функция — фрагмент программного кода (подпрограмма), к которому можно обратиться из другого места программы. Заметим, что имя должно быть выбрано таким образом, чтобы нам не нужно было смотреть в реализацию, чтобы понять, что делает функция.

Что нужно для использования функции:

- Объявление функции
- Вызов функции

Функция состоит из:

- Названия
- Списка аргументов (параметров)
- Тела функции
- Результата

Название — отражает, что функция должна делать.
Список аргументов — то, с чем это делать.
Тело функции — что она делает на самом деле.
Результат — то, что она сделала. Возвращается с помощью $return$. Результат не обязательно одно число или строка, им может быть список, $tuple$, список списков и т.д. Если после $return$ ничего не стоит, или вообще $return$ отсутствует, то ничего не будет возвращено, а точнее, будет возвращен $None$:

```
def print_square(a):
    print(a ** 2)

a = 5
b = print_square(a)
print(b)
print(type(b))
```

```
25
None
<class 'NoneType'>
```

Рассмотрим следующий пример:

```
def gcd(a, b):
    while b != 0:
        a, b = b, a % b
    return a

def gcd3(a, b, c):
    return gcd(gcd(a, b), c)

a, b, c = [int(i) for i in input().split()]
print(gcd3(a, b, c))
```

Заметим, что у функций должны быть разные имена, иначе произойдет ошибка. Хотя, хотелось бы, для функций, которое делают одно и то же, иметь одинаковые имена. Позже мы разберем, как это сделать.

1.1.1. Область видимости локальных переменных и аргументов. Ключевое слово global

Давайте посмотрим еще раз на код выше. В нем в нескольких местах присутствуют переменные a, b, c, причем, они имеют разный смысл: одни используются для вычисления в функциях, а другие — просто входные данные, которые мы затем должно передать в функцию. Возникают вопросы: почему здесь не происходит пересечения имён и почему числа, которые мы передали в функцию, не "портятся" ею.

Различают три типа областей видимости переменных:

- Локальная — переменные видны только в определенном месте кода
- Глобальная — глобальные видны везде
- Перекрытие — происходит перекрытие этих областей видимости

Рассмотрим пример:

```
def reverse(a, b, c): # (1)
    print(c, b, a)

print(a, b, c)
```

```
NameError Traceback (most recent call last)
<ipython-input-1-452abd77e9b8> in <module>()
2 print(c, b, a)
3
—> 4 print(a, b, c)
NameError: name 'a' is not defined
```

В строчке (1) переменные являются параметрами и становятся **локальными** аргументами для функции. При попытке обратиться к ним извне появится ошибка, что эти **переменные не определены извне**. То есть **они существуют только внутри функции** $reverse$.

Рассмотрим более сложный пример:

```
def reverse(a, b, c): # (1)
    print(c, b, a) # (2)

def watch_var():
    print(a, c, b)

a, b, c = 15, 10, 5 # (3)
reverse(a, c, b) # (4)
watch_var() # (5)
```

В строчке (3) мы определили тоже переменные a, b, c, причем они не лежат в какой-либо функции. Их область видимости теперь **глобальная**. В строчке (4) мы передаем параметры в функцию, причем **имена этих**

параметров в объявлении функции никак не связаны с глобальными именами. Внутри функции $reverse$ $a = 15, b = 5, c = 10$. Значит, **локальная область видимости перекрывает глобальную область видимости**.
В строчке (5) мы вызываем функцию $watch\_var$, но в ней печатаем переменные a, b, c. Казалось бы, они не определены в локальной области видимости этой функции. Но, здесь одно важное правило: **Если в любом месте функции не определена переменная, но она определена в глобальной области видимости, и внутри этой функции она используется только на чтение, то python будет искать эту переменную в глобальной области видимости. В случае, если он не найдет её, будет ошибка. Если найдет, то будет использовать её значение.** Заметим еще раз, что это верно только на **чтение**. Что же в итоге выведет функция $wath\_var$?

Рассмотрим теперь такой пример:

```
def f():
    print(s)
    s = 0
    print(s)

s = 5
f()
```

```
UnboundLocalError Traceback (most recent call last)
<ipython-input-7-6e9ade36b97b> in <module>()
5
6 s = 5
—-> 7 f()
<ipython-input-7-6e9ade36b97b> in f()
1 def f():
—-> 2 print(s)
3 s = 0
4 print(s)
5
UnboundLocalError: local variable 's' referenced before assignment
```

Будет ошибка. Давайте разберемся, почему. Что в этом коде есть:

- глобальная переменная $s = 5$
- локальная переменная функции f $s = 0$

Что мы только что говорили? В функции f есть определение **локальной** переменной s ($s = 0$), поэтому все s, которые будут встречаться в теле функции, будут локальными. Поэтому первый $print(s)$ захочет распечатать **локальную** переменную s. Но! Она к моменту печати еще не определена. Поэтому будет ошибка.

Попробуем убрать первый $print$:

```
def f():
    d = 0
    print(d)

d = 5
f()
```

> 0

Мы определили внутри функции f новую **локальную** переменную d, которая и будет передаваться в $print$, перекрывая область видимости глобальной переменной $d = 5$.

С записью/изменения значения переменной все несколько сложнее: **Если в любом месте функции не определена переменная, но она определена в глобальной области видимости, и внутри этой функции она используется на запись, то произойдет ошибка.**:

```
def f():
    w += 2
    print(w)

w = 0
f()
```

```
UnboundLocalError Traceback (most recent call last)
<ipython-input-12-b8a74390fc8c> in <module>()
4
5 w = 0
—-> 6 f()
<ipython-input-12-b8a74390fc8c> in f()
1 def f():
—-> 2 w += 2
3 print(w)
4
5 w = 0
UnboundLocalError: local variable 'w' referenced before assignment
```

Как же тогда изменять значение глобальной переменной внутри функции? Нам на помощь приходит ключевое слово $global$. В коде выше надо писать так:

```
def f():
    global w
    # теперь python знает, что w глобальная
    w += 2
    print(w)

w = 0
f()
print(w)
```

```
2
2
```

При изменении глобальной переменной меняется и её значение "извне", что является логичным поведением.

```
def f():
    z = 5
    print(z)
    global z
    z += 2
    print(z)

z = 0
f()
print(z)
```

```
5
7
7
<ipython-input-15-4b8837a18bd6>:4: SyntaxWarning: name 'z' is assigned to before global declaration global z
```

Попытаемся объяснить, что произошло. Вначале обратим внимание на предупреждение: переменная z объявляется перед глобальным объявлением z. Заметим, что это не ошибка, а предупреждение, поэтому код все равно будет работать. **Если в функции в любом месте стоит $global$ переменная, то эта переменная будет считаться глобальной во всем теле функции, независимо от того, была ли она объявлена раньше, как локальная.** Поэтому при присваивании $z = 5$ z уже считается как глобальная переменная, значит, её значение извне тоже изменяется. Дальше очевидно, почему мы получаем такой вывод.

```
def reverse(a, b, c):
    global a
    print(c, b, a)
```

```
File "<ipython-input-16-31ade39ac2b2>", line 2
global a
SyntaxError: name 'a' is parameter and global
```

Нельзя использовать одну и ту же переменную в качестве и параметра, и глобальной переменной. Такое поведение кажется вполне логичным.

Вообще, старайтесь использовать глобальные переменные, только когда вам очень-очень нужно. Ведь вы с легкостью можете забыть, что у вас была глобальная переменная, где-нибудь изменить её, или, наоборот, не изменить, а потом долго искать ошибку коде. Передавайте все необходимые переменные в параметры функции.

Также между объявлениями функций ставьте 2 пустых строчки (PEP8).

Теперь проговорим про параметры функции. Выделим два типа параметров:

Формальные параметры — это имена аргументов функции, за которыми могут скрываться произвольные значения. **Фактические параметры** - это конкретные значения, которые связываются с формальными параметрами при вызове функции.

```
def f(a, b):
    print(a + b)

f(3, 4)
```

a и b — это формальные параметры, а $3, 4$ — фактические параметры.

Помимо стандартного вида параметров, рассмотренных выше, в них могу встречаться следующие конструкции:

- Именованные параметры
- Списки параметров произвольной длины
- Значения по умолчанию

1.1.2. Именованные параметры

Любой параметр в объявлении функции — это имя аргумента. В коде выше, например, python понимает, что первый аргумент функции имеет имя a, а второй — имя b. Поэтому можно писать так:

```
def f(a, b):
    # a, b --- имена параметров
    print(a, b)

f(a=3, b=4) # a, b --- именованные параметры
f(b=3, a=4)
```

```
3 4
4 3
```

Заметим, при втором вызове функции мы поменяли местами параметры a и b. Но python понимает, какому аргументу какое значение присваивается.

Если не прописывать имена параметров при вызове функции, то значения будут присваиваться последовательно, в привычном понимании.

```
def f(a, b):
    print(a, b)

f(3, b=4) # все хорошо
f(3, a=4)
# ошибка, два раза присваиваем
# значение переменной a
f(a=3, a=4)
# ошибка, два раза используем
# одно и то же имя
```

```
def f(a, b, c):
    print(a, b, c)

f(c=3, 4)
""" ошибка. именнованные параметры можно использовать
только после неименованых. В данном случае
непонятно, 4 нужно присвоить a или b"""
```

Заметим, что вокруг $=$ в присвоении фактического параметра формальному не стоят пробелы — это часть соглашения PEP8.

Имена параметров и имена переменных не связаны:

```
def reverse(a, b, c):
    print(a, b, c)

a, b, c = 15, 10, 5
reverse(a, c, b)
reverse(a=b, b=a, c=c)
```

```
15 5 10
10 15 5
```

Вспомним, что у функции *print* были именованные параметры *sep*, *end*, *file* и мы как раз прописывали их всегда в конце.

1.1.3. Параметры по умолчанию

Вспомним, что мы писали функцию *print* и без именованных параметров, и всё хорошо работало. Для этого у параметров функции существуют значения по умолчанию. Когда вы вызываете функцию *print* без параметров *sep*, *end*, *file*, то в них подставляются значения, заданные по умолчанию: для *sep* это ' ', для *end* это '\n', а для *file* это stdin.

Вот так мы можем задавать значения по умолчанию:

```
def f(a, b=1):
    # по умолчанию b = 1
    print(a + b)

f(3, 4) # a = 3, b = 4
f(5) # a = 5, b = 1
f(a=5) # a = 5, b = 1
f(b=5)
# ошибка, значение a не присвоено
f() # ошибка, нет значения для a
f(c=6) # ошибка, неизвестное имя параметра
```

Теперь можно писать функцию f с одним или двумя параметрами, причем второй будет подставлен по умолчанию, в случае его отсутствия.

А вот так писать нельзя:

```
def f(a=5, b):
    print(a + b)

f(7)
```

> SyntaxError: non-default argument follows default argument

Писать параметры по умолчанию перед параметрами не по умолчанию нельзя. Например, если мы вызовем $f(7)$, то какому параметру присваивать 7? Не понятно.

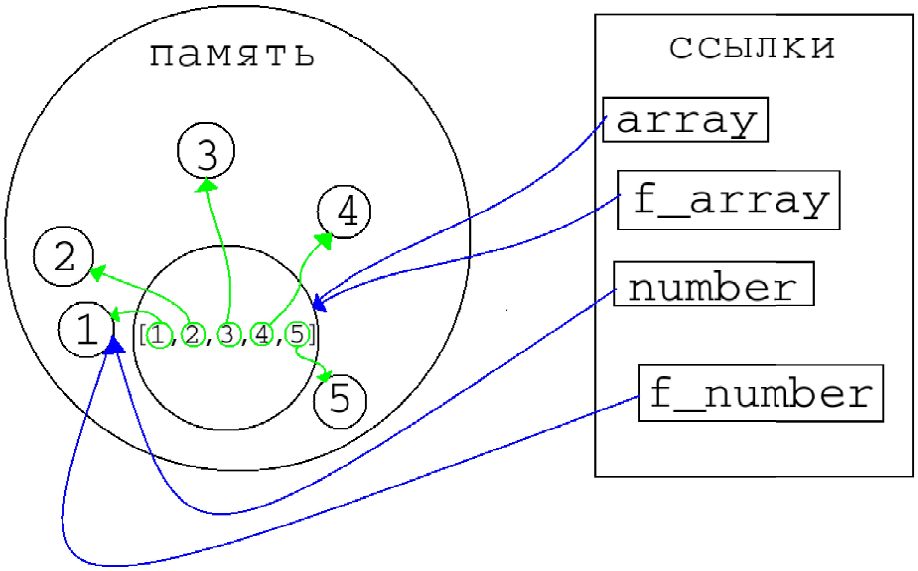
Вспомним, как хранятся объекты в памяти. Имена переменных — это на самом деле ссылки на объекты, хранящиеся в памяти. Каждое число и строка в памяти — это отдельный объект. При изменении числа (например, +=) изменяется не сам объект, а на самом деле ссылка переприсваивается получившемуся числу. Строки вообще менять нельзя, только переприсваивать. Кортежи тоже неизменяемые. В списках хранятся не

сами объекты, а ссылки на объекты. Соответственно, если поменять, например, 0-ой элемент в списке, сам список как объект в памяти останется, но поменяется ссылка на на 0-ой элемент.

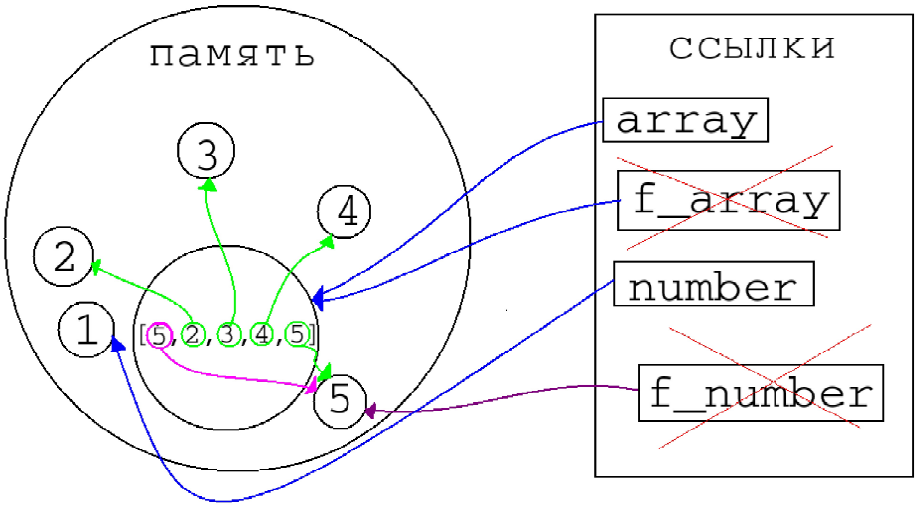
При передаче объектов в параметры функции передается не сам объект, а копируется ссылка на него.

```
def f1(f_array):
    f_array[0] = 5
    print(f_array)

def f2(f_number):
    f_number = 5
    print(f_number)

array = [1, 2, 3, 4, 5]
number = 1
f1(array)
f2(number)
print(array)
print(number)
```

```
[5, 2, 3, 4, 5]
5
[5, 2, 3, 4, 5]
1
```



В момент входа в каждую из функций



После выхода из каждой функции

Еще одно очень важное замечание: **При работе программы значения именованным параметрам присваиваются один раз, в месте определения функции. Если присваемый объект изменяемый, то**

измененный во время работы функции(!!!, а не при передаче параметра), он при в следующих вызовах будет иметь не то, значение, которое указано как значение в определении функции, а то, которое было присвоено во время предыдущего вызова.

Рассмотрим следующие примеры:

```
def f(a, b=[1, 2, 3]):
    print(a)
    print(b)


f(5)
f(5, [4, 5, 6])
f(5)
```

```
5
[1, 2, 3]
5
[4, 5, 6]
5
[1, 2, 3]
```

```
def f(a, b=[1, 2, 3]):
    print(a)
    print(b)
    b = [4, 5, 6]


f(5)
f(5)
```

```
5
[1, 2, 3]
5
[1, 2, 3]
```

```
def f(a, b=[1, 2, 3]):
    print(a)
    print(b)
    b[0] = 5
    b.append(4)


f(5)
f(5)
```

```
5
[1, 2, 3]
5
[5, 2, 3, 4]
```

В первом случае при втором вызове функции мы меняем лишь то место, на что ссылается ссылка b, при этом сам аргумент по умолчанию остается неизменным. Поэтому функция, при следующем её вызове, использует неизменное значение аргумента по умолчанию.

Во втором случае мы переприсваиваем ссылку b другому списку, но, опять же, исходное значение по умолчанию остается неизменным.

В третьем случае мы изменяем значение по умолчанию: вспомните ссылочное устройство элементов списка. Мы меняем лишь то место, на что ссылается значение по умолчанию, сам же объект в целом не меняется (не создается нового списка).

1.1.4. Изменяемое число параметров, *

Так же непонятно, как функции $print$ удается принимать различное число печатаемых переменных. Для этого есть специальная конструкция для

переменного числа параметров: *

```
def sum(*args):
    # теперь функция принимает произвольное
    # число параметров и возвращает их сумму
    # обычно пишут именно args
    print(type(args))
    # args --- кортеж(\tuple) из переданных
    # значений
    result = 0
    for x in args:
        result += x
    return result

print(sum(1))
print(sum(1, 2, 3))
L = [1, 2, 4, 8, 16, 32]
print(sum(*L)) (1)
```

```
<class 'tuple'>
1
<class 'tuple'>
6
<class 'tuple'>
63
```

В строчке (1) приведена замечательная конструкция, называемая, **аргументом со звездочкой**. Она распаковывает список L так, что в параметры функции передаются, в данном случае, 6 отдельных аргументов.

Окончательная версия НОД от произвольного числа параметров:

```
def gcd2(a, b):
    while b != 0:
        a, b = b, a % b
    return a
```

```
def gcd(*args):
    answer = 0
    for arg in args:
        answer = gcd2(answer, arg)
# свойство gcd(a_1, ..., a_n) = gcd(a_1, gcd(a_2, ..., a_n))
    return answer

numbers = [int(i) for i in input().split()]
print(gcd(*numbers))
```

1.2. Рекурсия

Рекурсия — см. рекурсия
Рекурсия — вызов функции из самой себя или через другие функции. Выражение решения задачи через решения подзадач.

1.2.1. Рекурсивный факториал. Стек вызовов функции. Глубина рекурсии

Для того, чтобы понять, что такое рекурсия, рассмотрим следующую задачу: посчитать факториал введённого числа $n \geq 0$.
Факториал n — произведение всех чисел от 1 до n. Обозначается как $n!$. По определению 0! = 1.

Заметим, что $n! = n \cdot (n-1)!$. Значит, если у нас есть некоторая функция $factorial$, которая умеет вычислять факториал числа, то эта функция выражается сама через себя: $factorial(n) = n \cdot factorial(n-1)$, причем $factorial(0) = 1$.

Давайте напишем такую функцию:

```
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n - 1)

print(factorial(0))
print(factorial(3))
```

Получим: $n *$ factorial$(n-1) = n * (n-1) *$ factorial$(n-2) = ... = n! *$ factorial$(0) = n!$

Теперь немного отвлечемся и узнаем, что такое **стек**.

Стек — такая **структура данных** ("коробочка"), в которую значения кладутся последовательно, а вынимаются с конца (первый вошел — последний вышел).

Пример из жизни: пирамидка, на которую надевают диски. Сначала вы надеваете самый большой диск, потом меньше, потом еще меньше и так далее, пока не дойдете до самого маленького. А чтобы снова взять самый большой диск, сначала нужно снять самый маленький и так далее, пока не дойдете до самого большого.

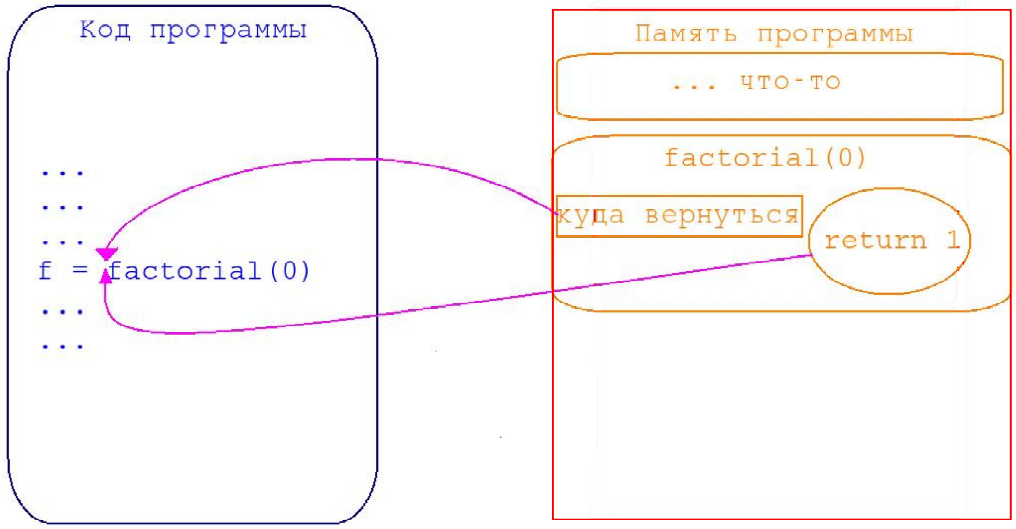
Пример из программирования:

```
stack = []
stack.append(2)
stack.append(3)
stack.append(5)
print(stack)
print(stack.pop())
print(stack.pop())
print(stack.pop())
print(stack)
```

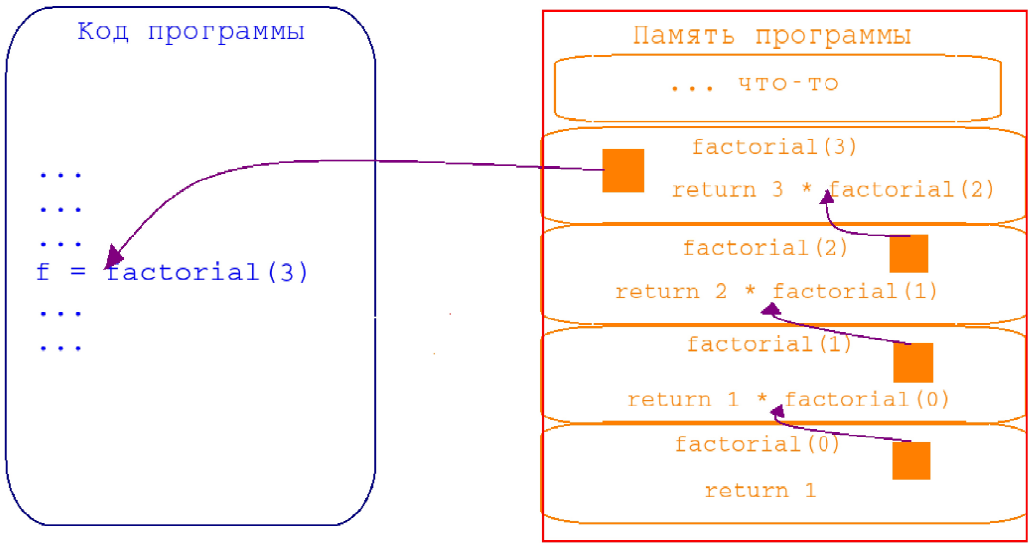
```
[2, 3, 5]
5
3
2
[ ]
```

Когда в программе вызывается функция, то под функцию выделяется некоторый кусок памяти, доступный интерпретатору. В этом куске памяти хранятся передаваемые параметры, локальные переменные, а так же некоторые дополнительные переменные (поступите в университет — узнаете, что там на самом деле ;). По окончании работы функции, выделенная

память полностью освобождается. Вся память, выделяющаяся под функцию, называется **фреймом функции**. Причем, что очень важно, функция "знает", из какого места программы её вызвали, поэтому по завершении работы функции вы попадаете на то место, откуда функция была вызвана. А также интрепретатор умеет возвращать значение из функции (если она должна возвращать его) (а как, вы тоже узнаете в университете;). Это свойство как раз и используется при применении рекурсии. Нарисуем, как будет выглядеть память, если мы вызовем $factorial$ от 0 и от 3.



Стек вызовов функций $factorial(0)$



Стек вызовов функций $factorial(3)$

Вызов $factorial(0)$ работает как вызов обычной функции: мы тут же возвращаем значение 1. Причем, благодаря специальной переменной, функция знает, куда его нужно возвращать.

Вызов $factorial(3)$ работает уже гораздо хитрее: чтобы мы смогли что-то вернуть из функции $factorial(3)$, нам нужно посчитать $factorial(2)$, а чтобы посчитать его, нужно посчитать $factorial(1)$, для которого нужен $factorial(0)$. Поэтому мы будем делать вызовы тех функций, которые нам необходимы, что посчитать текущую. Как только мы посчитали $factorial(0)$, функция завершает свою работу и возвращает значение 1 в функцию $factorial(1)$, а сама память, выделенная под функцию, освобождается. И так далее будет происходить до тех пор, пока мы не вернемся с изначальному месту вызова функции $factorial(3)$. Причем, функции будут располагаться в памяти "друг под другом" (так написали умные бородатые дяди). Такая конструкция вызовов функций называется **стек вызовов функций**. Процесс освобождения памяти, путем перехода к вызывающей функции, называется **сверткой стека**. Заметим, что стек здесь очень кстати: ведь та функция, которую мы взывали первой, будет завершена последней. (Показать на картинке). Максимальное кол-во функций, которое мы вызвали подряд, называется **глубиной рекурсии**. В данном случае для 0 глубина равна 1, а для 3 глубина равна 4.

1.2.2. Ограничение памяти. Функция sys.setrecursionlimit(). База рекурсии

А что будет, если мы вызовем $factorial$ от оочень большого числа? Память у компьютера и у, тем более, интерпретатора, не бесконечная, поэтому при слишком большом числе вызываемых подряд функций (большая глубина рекурсии) память может закончится и программа выдаст ошибку. В python, по умолчанию, совсем все печально: интерпретатор позволяет иметь глубину рекурсии не более 1000. Для того, чтобы увеличить это число, надо проделать такую магию:

```
import sys
# специальный модуль для взаимодействия
# с интепретатором
print(sys.getrecursionlimit())
# узнать текущую глубину рекурсии
sys.setrecursionlimit(1000000)
# установить новую глубину рекурсии
```

Заметим одну важную вещь: если бы не было вызова функции, которая возвращает просто число (а не выражается через саму себя же), то мы бы всё время вызывали новую функцию пока бы не закончилась память. Такие вызовы функции, для подсчета которых не требуется еще вызовы аналогичных функций, называются **базами рекурсии**. В данном примере базой рекурсии является $factorial(0)$ (только в этом случае, мы возвращаем число, а не вызываем еще одну функцию $factorial$). Другими словами, базы рекурсии позволяют рекурсии остановиться через конечное число шагов.

1.2.3. Рекурсивные числа Фибоначчи. Дерево рекурсии

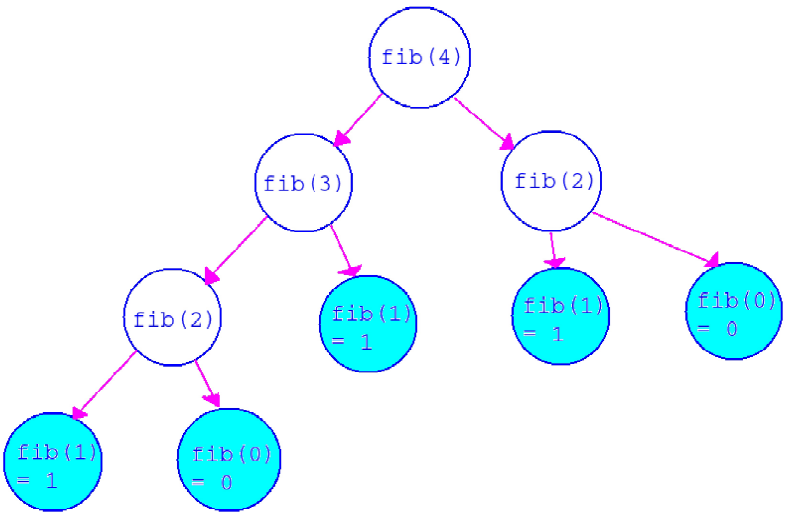
Число Фибоначчи F_n — такое число, что $F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2} \forall n \geq 2$.
Пример: $F_0 = 0, F_1 = 1, F_2 = F_0 + F_1 = 1, F_3 = F_2 + F_1 = 1 + 1 = 2, F_4 = 2+1 = 3, F_5 = 3+2 = 5, F_6 = 5+3 = 8, F_7 = 8+5 = 13, F_8 = 13+8 = 21, ...$

Напишем рекурсивный подсчет n-ого числа Фибоначчи:

```
def fib(n):
    if n == 0:
        return 0
    if n == 1:
        return 1

    return fib(n - 1) + fib(n - 2)

n = int(input())
print(fib(n))
```

Базы рекурсии — $fib(0)$ и $fib(1)$. В этом примере рисовать стек вызовов чуть сложнее, поэтому пока нарисуем более простую конструкцию, называемую **деревом рекурсии**. При $n = 4$.

$fib(4) = fib(3) + fib(2) = fib(2) + fib(1) + fib(1) + fib(0) = fib(1) + fib(0) + 1 + 1 + 0 = 1 + 0 + 2 = 3$.



Стрелочка из вершины A в B означает, что функция A вызывает функцию B. Вызываем функции в порядке слева направо. Максимальное кол-во вызванных функций подряд — 4. Всего было сделано вызовов 9 функций. Заметим, что мы много раз вызывали одинаковые функции: от 0 — 2 раза, от 1 — 3 раза, от 2 — 2 раза. При возрастании n кол-во одина-

ково вызванных функций растет очень быстро. Это один из недостатков рекурсии — возможны **"лишние" вычисления**. Выделение памяти под функцию (**создание фрейма**) довольно долгая и затратная операция, поэтому чем больше вызовов в рекурсии вы сделаете, особенно лишних, тем дольше будет работать программа.

1.2.4. Время работы рекурсивных функций

Как мы уже говорили раннее, время работы рекурсивных функций зависит от кол-ва вызываемых функций в рекурсии. Давайте посчитаем, сколько примерно нужно совершить вызовов функций, чтобы посчитать n-ое число Фибоначчи. Будем использовать прием **математической индукции**:

- Доказать предположение для базы
- Предположить, что предположение верно для некоторого n или для всех чисел от базы до n.
- Доказать верность предположения для $n + 1$

Для $n = 0$ или $n = 1$ будет сделан 1 вызов. Для $n = 2$ — 3 вызова, $n = 3$ — 5 вызовов, $n = 4$ — 9 вызовов. Уже видно, что кол-во вызовов растет быстро.

Пусть $call(n)$ — это количество вызовов функций подсчета числа n-ого числа Фибоначчи. Тогда, глядя на дерево, можно сказать, что:

$$call(n) = call(n-1) + call(n-2) + 1$$

Докажем, что $call(n) \geq fib(n)\ \forall n \geq 0$.

- База: для $n = 0 : call(0) = 1 > fib(0) = 0, n = 1 : call(1) = 1 \geq fib(1) = 1, n = 2 : call(2) = 3 \geq fib(2) = 1$ — верно.
- Пусть $\forall\ 1 \leq k \leq n$ предположение верно: $call(k) \geq fib(k)$
- Докажем для $n + 1$:

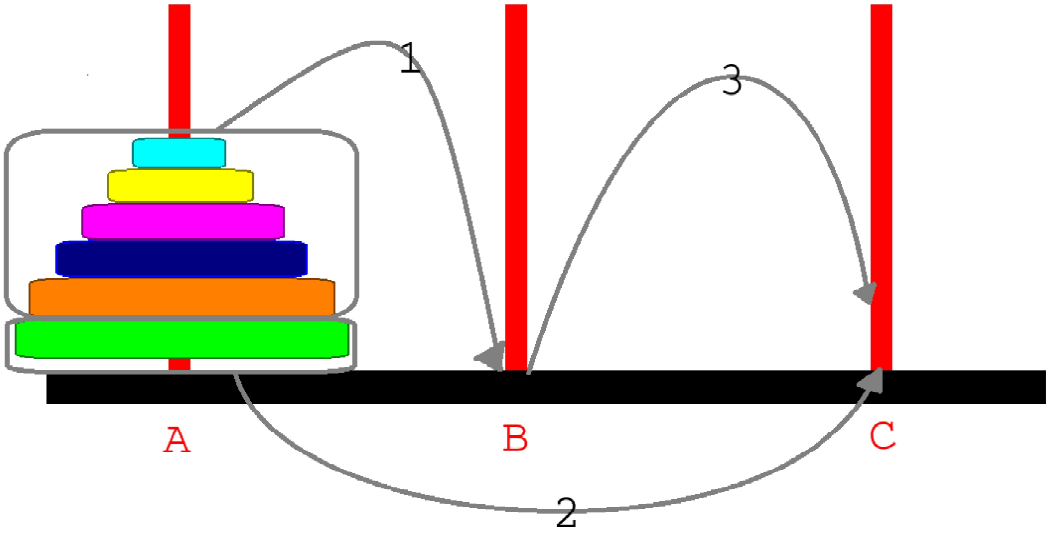
$$call(n+1) = call(n) + call(n-1) + 1 \geq fib(n) + fib(n-1) + 1 =$$

$$= fib(n+1) + 1 > fib(n+1)$$

Заметим, что числа Фибоначчи растут очень быстро, например, $fib(30) = 832040$, поэтому время работы программы тоже будет расти очень быстро.
Замечание: Рекурсивный НОД напишите самостоятельно на основе соотношения для НОД от двух чисел: НОД(a, b) = НОД$(b, a\%b)$, $a \geq b$.
Подсказка: Если $a < b$, то НОД(a, b) = НОД$(b, a\%b)$ = НОД(b, a), $b > a$

1.2.5. Ханойские башни

Задача: Дано три стержня и на одном из них (стержень A) находится пирамидка из n дисков, как на рисунке. Большее кольцо не может лежать выше меньшего. Надо переставить эту пирамиду со стержня A на стержень C.
Идея: Пусть нам известна функция $hanoi(n, source, middle, destination)$, которая умеет перекладывать n последовательных дисков со стержня $source$ на стержень $destination$ через стержень $middle$, причем делает это без нарушения правил задачи. Тогда, чтобы решить задачу, надо переложить $n - 1$ верхних дисков со стержня A на стержень B, отдельно переложить самый большой диск по стержня A на стержень C, а затем переложить $n - 1$ дисков со стержня B на стержень C



Если мы вызываем нашу функцию от 1 диска, то, очевидно, что с ним надо сделать: сразу переложить с $source$ на $destination$. Поэтому это будет базой рекурсии. Заметим, что класть диски можно не обязательно

строго по убыванию, можно, например, положить самый маленький диск на самый большой.

Докажем, что кол-во вызовов рекурсий для параметра $call(n) = 2^n - 1$:

- База: $n = 1$: $call(1) = 1 = 2^1 - 1$
- Пусть для n кол-во вызовов составляет $call(n) = 2^n - 1$
- Докажем для $n+1$: $call(n+1) = call(n) + call(n) + 1 = 2 \cdot (2^n - 1) + 1 = 2^{n+1} - 1$