**21.3 Рекурсии**

**Подробнее о рекурсии**

В первую очередь надо понять, что рекурсия не использует новые инструменты и незнакомые особенности языка. Если откинуть лишнее, то рекурсия — это просто вызов функции.

Привычная простая функция без рекурсии:

def foo(x):  
     print(x)

Функция с рекурсией:

def foo(x):  
     foo(x)

Первая функция распечатает x, вторая будет бесконечно вызывать себя же и вызовет исключение. Результат их работы разный, но принцип — одинаковый.

И там, и там функция foo вызывает новую функцию (в первом случае print, во втором — foo).

И там, и там пайтон приостановит выполнение текущей функции foo, чтобы полностью выполнить вызов вложенной функции, а после завершения — продолжит выполнение текущей функции.

И там, и там, когда пайтон дойдёт до выполнения print(x) или foo(x), он возьмёт имя переменной (print или foo) и будет искать в памяти то, что хранится под этими именами. Далее найдёт функцию, а внутри функции найдёт код и выполнит его. Пайтон не различает, та же самая это функция или новая, он просто видит имя функции и выполняет код внутри этой функции.



Рекурсия встречается и в реальной жизни. Источник: <https://reddit.com/>

Система с приостановлением выполнения текущей функции для выполнения новой называется стеком вызовов и работает примерно так:

Пример с простой функцией:  
def foo(x):  
    print(x)

foo(5)

|  |  |
| --- | --- |
| Пайтон доходит до вызова foo(5) и foo(5), помещается внутрь стека вызовов.    Далее пайтон выполняет код функции, которая находится на вершине стека (у нас в стеке сейчас одна функция, она и выполняется).    Внутри функции пайтон встречает вызов другой функции:  print(x) | https://api.selcdn.ru/v1/SEL_72086/prodLMS/files/share/image_02.png |
| Она тоже добавляется в стек.    Сразу две функции пайтон выполнять не может, поэтому foo(5) приостанавливается, а print(5) запускается (запускается всегда та функция, которая выше в стеке вызовов). | https://api.selcdn.ru/v1/SEL_72086/prodLMS/files/share/image_03.png |
| Код внутри print(5) выполняется, число 5 выводится на консоль, функция print завершается и, при завершении, удаляется из стека. | https://api.selcdn.ru/v1/SEL_72086/prodLMS/files/share/image_04.png |

Управление возвращается к функции foo(5).  
Так как кода в этой функции больше нет, то и foo(5) завершается и удаляется из стека.

Таким образом стек помогает пайтону создать очередь из функций, которые надо выполнить. Только, в отличие от обычной очереди, то, что пришло раньше всего (foo(5) в нашем примере), уйдёт позже всего (foo(5) будет полностью выполнен только после того как будет выполнена функция print). Вот такая несправедливая очередь получается. Кто позже пришёл — тот быстрее выполняется.

### Стек вызова функции

Как это поможет разобраться с рекурсиями?

Давайте разберём тот же пример со стеком с рекурсией. Только немного изменим пример, чтобы наша рекурсия не вызывала исключения, а завершалась, спустя несколько повторных вызовов.

Пример:  
def foo(x):  
     if x <= 0:  
        print(x)  
    else:  
        foo(x - 1)

foo(3)

Тут у нас есть блок if, который поможет завершить вызовы новых рекурсий.   
У нас есть два варианта действий.   
Если x равен 0, то мы выполняем простой принт.  
Если x не равен 0, то мы вызываем рекурсию с x, значение которого уменьшено на 1.

В итоге, если мы передадим в функцию 3, то функция вызовет себя же с числом 2, потом ещё раз вызовет себя же с числом 1, ещё раз вызовет себя с нулевым значением, и все вызовы будут завершены.

Как это выглядит изнутри:

|  |  |
| --- | --- |
| foo(3) — стартует оригинальный вызов.  Внутри выполняется проверка 3 <= 0?  Если ответ false, то сработает код в else.    foo(3-1) → пайтон сперва посчитает 3-1, получит 2 и запустит → foo(2). |  |
| foo(2) запустит проверку 2 <= 0?  Опять ответ будет false,  значит сработает else и вызов  foo(2-1) → foo(1) |  |
| foo(1) запустит проверку 1<= 0?  Будет false, сработает else.    foo(1-1) → foo(0) |  |
| foo(0) запустит проверку 0<= 0? Наконец-то ответ true!  Значит, else не сработает и следующего вызова не будет!  Сработает функция print(0), которая, опять же, будет добавлена в стек. |  |
| Вот так мы заполнили несправедливую очередь функциями, дали пайтону много работы, и он начинает (правильнее будет сказать, «продолжает») выполнять эти функции с самой верхней — с принта.    Выполняется принт: 0 выводится в консоль, функция завершается, принт удаляется из стека. |  |
| Управление возвращается к foo(0).  На чём мы остановились?    def foo(x):      if x <= 0:          print(x) — этот принт выполнился как  else:          foo(x-1)  вот сюда.    Else не сработает. Если сработал if, то выполнение кода переходит на строку «вот сюда». Кода в функции больше нет, а значит функция завершается и удаляется из стека. | https://api.selcdn.ru/v1/SEL_72086/prodLMS/files/share/image_10_FlhAden.png |

\*Интересный факт: если бы после if/else был ещё какой-то код, то он бы выполнился после срабатывания принта. То есть функция выполнила бы часть кода до другой функции, остановилась бы на время выполнения вложенной функции, а, по возвращении к работе, — выполнила бы остальную часть кода.

|  |  |
| --- | --- |
| После завершения и удаления foo(0) из стека, он выглядит так: | 1234 https://api.selcdn.ru/v1/SEL_72086/prodLMS/files/share/image_11_%D0%BA%D0%BE%D0%BF%D0%B8%D1%8F.png |
| Теперь пайтон возвращается к функции foo(1).  В ней тоже нет кода, который нужно выполнить, поэтому функция просто завершается. | https://api.selcdn.ru/v1/SEL_72086/prodLMS/files/share/image_12_UrOdzOF.png |
| foo(2) тоже завершается и удаляется. В стеке остаётся только оригинальный вызов foo(3).    Мы называем его оригинальным, потому что он был вызван изначально, а остальные вызовы были начаты рекурсивно. | https://api.selcdn.ru/v1/SEL_72086/prodLMS/files/share/image_13_EjNhFoQ.png |

Пайтон возвращается к этой функции, смотрит, есть ли ещё код, который надо выполнить в этой функции.

Если не находит кода, то функция завершается и удаляется из стека, оставляя его полностью пустым.

Очень важно продумывать условие выхода из рекурсии, иначе она будет работать бесконечно, и пайтон вызовет ошибку, чтобы она не съела всю память вашего компьютера.



Источник: <https://reddit.com/>

Таким образом, при помощи стека мы можем проследить всю цепочку «жизни» рекурсии и увидеть, что вся эта цепочка — повторяющийся вызов функций. Значит, рекурсия — это не новый инструмент, а, скорее, новый способ использования уже знакомых функций.

### Вернуть результат рекурсии — невыполнимо?

Если это так, то с вызовами рекурсии работают те же правила, что и с вызовами обычных функций.

В примере мы передавали в каждый новый вызов новое число, чтобы каждый новый вызов сработал иначе. Так мы делаем и с обычными функциями, выполняя повторяющийся код с новыми значениями переменных.

Но что делать, если мы захотим вернуть значение из рекурсии?

В первую очередь, нужно вспомнить, как возврат работает с обычными функциями. return возвращает какой-либо объект в место, в котором была запущена функция.

Если мы создадим функцию:  
def sum\_test(x, y):  
    return x + y

и вызовем её:  
result = sum\_test(1, 2)    
print(result)

то пайтон:

1. найдёт по названию объект sum\_test;
2. поймёт, что это функция;
3. создаст переменные x=1 и y=2;
4. посчитает сумму x+y с этими переменными;
5. вызовет return с получившимся ответом;
6. в итоге число, которое мы передали return, заменит вызов функции;

то есть sum\_test(1, 2) будет заменён на число 3;

значит:   
result = sum\_test(1, 2),   
теперь будет  
result = 3.

В конце переменная будет распечатана в консоль.

Функция вернула своё значение именно в ту строку, в которой была вызвана.

Попробуем выполнить то же самое с рекурсией:  
def foo(x):  
    if x <= 0:  
        return 0  
    else:  
        new\_result = foo(x - 1)  
        result = x + new\_result  
        return result

По задумке функция будет получать число x (пусть оно будет целым и положительным) до тех пор, пока оно больше 0 — мы будем вызывать рекурсивный вызов этой функции с x, уменьшенным на 1.

То есть, если вызвали foo(2), то рекурсивно будут вызваны сначала foo(1), а потом — foo(0).

Чтобы лучше проследить весь путь рекурсии, добавим принты!



Big brother is watching. Фото: David Blackwell

def foo(x):  
    if x == 0:  
       print("Вызов foo(0) возвращает 0")  
        return 0  
    else:  
        print(f"Вызов foo({x - 1}) начинается и добавляется в стек")  
        new\_result = foo(x - 1)  
        print(f"Вызов foo({x - 1}) завершился и удаляется из стека")  
        result = x + new\_result  
        return result

print(f"Вызов foo(2) начинается и добавляется в стек")  
result = foo(2)  
print(f"Вызов foo(2) завершается и удаляется из стека")  
print("Итоговый ответ — ", result)

При запуске мы получим такой вывод в консоль:  
Вызов foo(2) начинается и добавляется в стек.  
Вызов foo(1) начинается и добавляется в стек.  
Вызов foo(0) начинается и добавляется в стек.  
Вызов foo(0) возвращает 0.  
Вызов foo(0) завершился и удаляется из стека.  
Вызов foo(1) завершился и удаляется из стека.  
Вызов foo(2) завершается и удаляется из стека.  
Итоговый ответ — 3.

Мы можем увидеть сходство с прошлым примером.  
Сперва стек заполняется вызовами foo(2), foo(1), foo(0).  
Затем foo(0) завершается.  
Пайтон начинает выполнять все приостановленные функции:  
сначала foo(1),   
потом foo(2).

Этот элемент является важным в примере:  
new\_result = foo(x - 1).

Мы рекурсивно вызываем функцию с числом, уменьшенным на 1, и записываем результат выполнения этой функции в переменную new\_result.

Если бы мы написали что foo(x-1), то результат бы посчитался, вернулся, но не был бы сохранён в переменную и в итоге был бы потерян. Часто делают эту ошибку, думая, что return завершает сразу все вызовы рекурсии и отправляет ответ в то место, где была вызвана оригинальная функция «снаружи». Но это не так.

Если бы мы вызывали другие функции с return внутри нашей функции, то их return не завершил бы нашу. Это логично, ведь мы сами хотим завершить её, когда получим нужный ответ.

Например:  
def sum\_test(x, y):  
    return x + y

def test\_return(a, b, c):  
    new\_summ = sum\_test(a, b) — вызывая эту функцию, мы хотим получить промежуточный результат; мы не хотим, чтобы наша функция завершилась;  
    result = new\_summ + c — получив результат вложенной функции, мы его изменяем  
 и возвращаем (return result).

value = test\_return(1, 2, 3)  
В итоге в value будет записано то число, которое попало в return result, но ни в коем случае не то число, которое вернулось при помощи return x+y.

Так и с рекурсией: каждый вложенный вызов делает промежуточные расчёты, которые мы МОЖЕМ (если захотим) сохранить и использовать в наших дальнейших расчётах.

Рассмотрим нашу функцию подробнее:  
def foo(x):  
    if x <= 0:  
        return 0  
    else:  
        new\_result = foo(x - 1)  
        result = x + new\_result  
        return result

print(foo(2))

Запускается foo(2).  
Срабатывает else, так как 2 больше 0.

Пайтон начинает выполнять первую строку внутри else:  
new\_result = foo(2-1).

В этой строке у нас целых три действия.

1. 2-1
2. foo(1)
3. new\_result =

И, что интересно, пайтон выполнит первое действие и посчитает 2-1.  
Это нужно, чтобы запустить функцию foo(1). Но после запуска функция foo(2) будет поставлена на паузу, и действие new\_result = будет выполнено после того, как завершится действие foo(1). Следующие строки тоже не будут выполнены, пока функция foo(2) приостановлена!

Идём дальше. Запускается foo(1).  
Опять срабатывает else, так как 1 больше 0.

Пайтон доходит до строки:  
new\_result = foo(1-1)  
Выполняет foo(1-1) → foo(0).

Теперь функция foo(1) тоже приостановлена!  
Управление переходит к foo(0).  
Срабатывает условие if x<= 0, так как 0 равен 0.   
Срабатывает return 0.  
И этот return:

1. завершает работу функции foo(0);
2. возвращает число 0 туда, где была вызвана функция foo(0), а значит возвращает 0 в вызов foo(1).

Пайтон снимает с паузы функцию foo(1)  
и продолжает с того места, на котором остановился:  
new\_result = 0;  
result = 1+0  (x в текущем вызове по-прежнему равен 1);  
return 1 — этот return завершает функцию foo(1).

Функция foo(1) удаляется из стека, а пайтон снимает с паузы оригинальный вызов foo(2).

Снова выполнение продолжается с того места, на котором мы остановились:

new\_result = 1 — тут мы вызывали foo(1); выше мы посчитали, что результат foo(1) был равен числу 1;  
result = 2+1  (x в текущем вызове по-прежнему равен 2);  
return 3 — этот return завершает функцию foo(2).

Из стека удаляется функция foo(2),  
а результат её выполнения попадает в строку, в которой была вызвана эта функция:  
print(foo(2)) — то есть сюда.

И, наконец, мы распечатываем число 3 :)

Ещё один важный момент — визуализация.

Пониманию рекурсии (и других сложных идей) очень помогает «подключение» органов чувств. В нашем случае можем использовать визуальную информации о ходе выполнения рекурсивного алгоритма.

Есть хороший пример — рисование дерева при помощи рекурсии.

Если мы будем рисовать функцией одну ветвь, а в конце функции вызывать себя же дважды (чтобы нарисовалось две ветви из конца одной), то получим что-то подобное:

Play Video

В видео возможно упоминание Инстаграм и Фейсбук. Деятельность компании Meta Platforms Inc., которой принадлежит Инстаграм / Фейсбук, запрещена на территории РФ в части реализации данных социальных сетей на основании осуществления ею экстремистской деятельности

Дерево — очень хорошая метафора. Каждая ветвь на рисунке — это один вызов функции. Мы можем проследить, какой вызов откуда начался и в какой момент эти вызовы заканчиваются. Например, если мы в условии выхода из рекурсии поставим определённый размер ветви (каждый раз ветвь немного уменьшается; мы можем перестать рисовать ветви, если они станут меньше указанного размера), то такое условие обрежет часть ветвей, и мы наглядно увидим, как сократилось количество вызовов.

Кроме того, интересной может быть работа с параметрами. Каждый вызов должен получить точку, из которой рисуется ветка. Если во все вызовы передавать начальную точку, то у нас получится странный куст. Чтобы получилось дерево, нужно в конце вызова передавать следующим веткам текущую точку, в которой закончилась ветка текущего вызова. И на практике, при использовании рекурсий, зачастую надо в каждый новый вызов передавать новые параметры. Так, в нашей прошлой задаче, мы передавали x в каждый новый вызов, уменьшая его на единицу. В итоге каждый вызов функции обрабатывал своё значение x, а когда приходила очередь нуля, рекурсия завершалась.

Больше примеров визуализации вы найдете по ссылкам ниже:

* [Recursion Visualization - Algorithms (англ.)](https://www.youtube.com/watch?v=mFb1Fj4sVcU)
* Ссылка [на инструмент, с которым вы можете самостоятельно визуализировать свои функции](https://quanticdev.com/tools/recursion-visualization/), но на другом языке программирования. Можно подогнать свой код под него или воспользоваться [другим ресурсом](https://www.recursionvisualizer.com/).

### Резюме

##### **Основные моменты**

1. Рекурсия — это многократный вызов обычных функций.
2. Чтобы она сработала, в первую очередь, надо добавить условие выхода из рекурсии.
3. С рекурсивными вызовами функции работают те же правила, что и с обычными функциями. В них можно передавать значения параметрами и получать результат работы посредством return.



Источник: <https://reddit.com/>

Дополнительные источники информации:

* [Знакомимся с рекурсией](https://techrocks.ru/2019/09/15/recursion-demystified/).
* Книга «Грокаем алгоритмы» Адитья Бхаргава, глава 3. Эта глава о рекурсии, но можете ознакомиться и с другими главами книги, в ней много полезной информации.
* Хороший пример разбора рекурсивной задачи известной команды KhanAcademy: [Пошаговое объяснение рекурсивной функции Фибоначчи](https://www.youtube.com/watch?v=F4HNO4rCQ2c).