**2023/10/09 00:00|Лекция. Рекурсия**

## Рекурсия

На этом уроке мы с вами поговорим про рекурсию. Что это такое, чем она опасна?

**Рекурсия** — это такой способ определения функции или описание функции, когда эта самая функция вызывает саму себя.

**На практике давайте посмотрим на самый простой пример.**

Создадим функцию, которая будет находить суму чисел с помощью рекурсии(рис.1). Назовем ее summa - она будет принимать один параметр n и здесь при написании рекурсивной функции нас интересуют два момента: в какой момент функция будет вызывать саму себя и в какой момент все это дело будет останавливаться.

Останавливаться все будет тогда, когда этот параметр n будет равен 0. Соответственно, здесь мы используем оператор return для выхода из функции и будем говорить, что она вернет нам 0. Но когда оператор n будет неравен 0, в остальных случаях, мы будем возвращать этот самый параметр, при этом к нему мы будем добавлять вызов функция, но только с параметром, который меньше на единицу( n + summa(n-1))(рис.1).

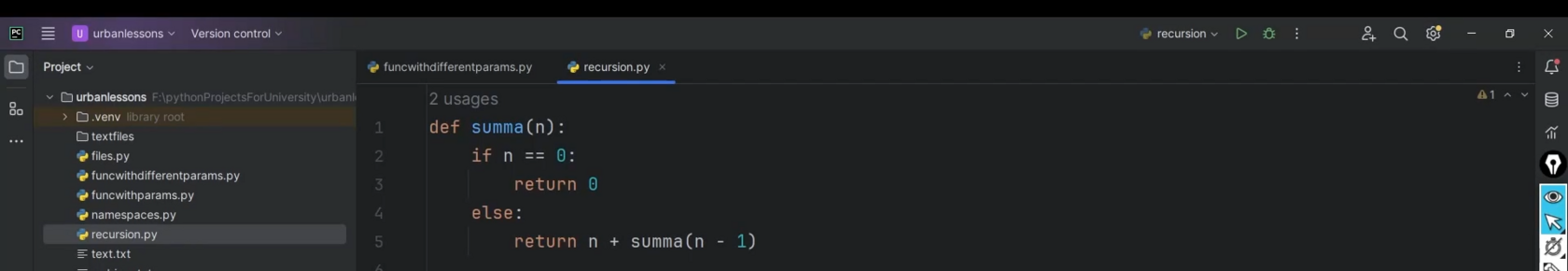


Рис.1

Давайте посмотрим. Вызовем нашу функцию summa для 5. Получили 15(рис.2). Это работает, то есть мы сделали вот такую реализацию, нахождение суммы чисел, используя рекурсивный подход.

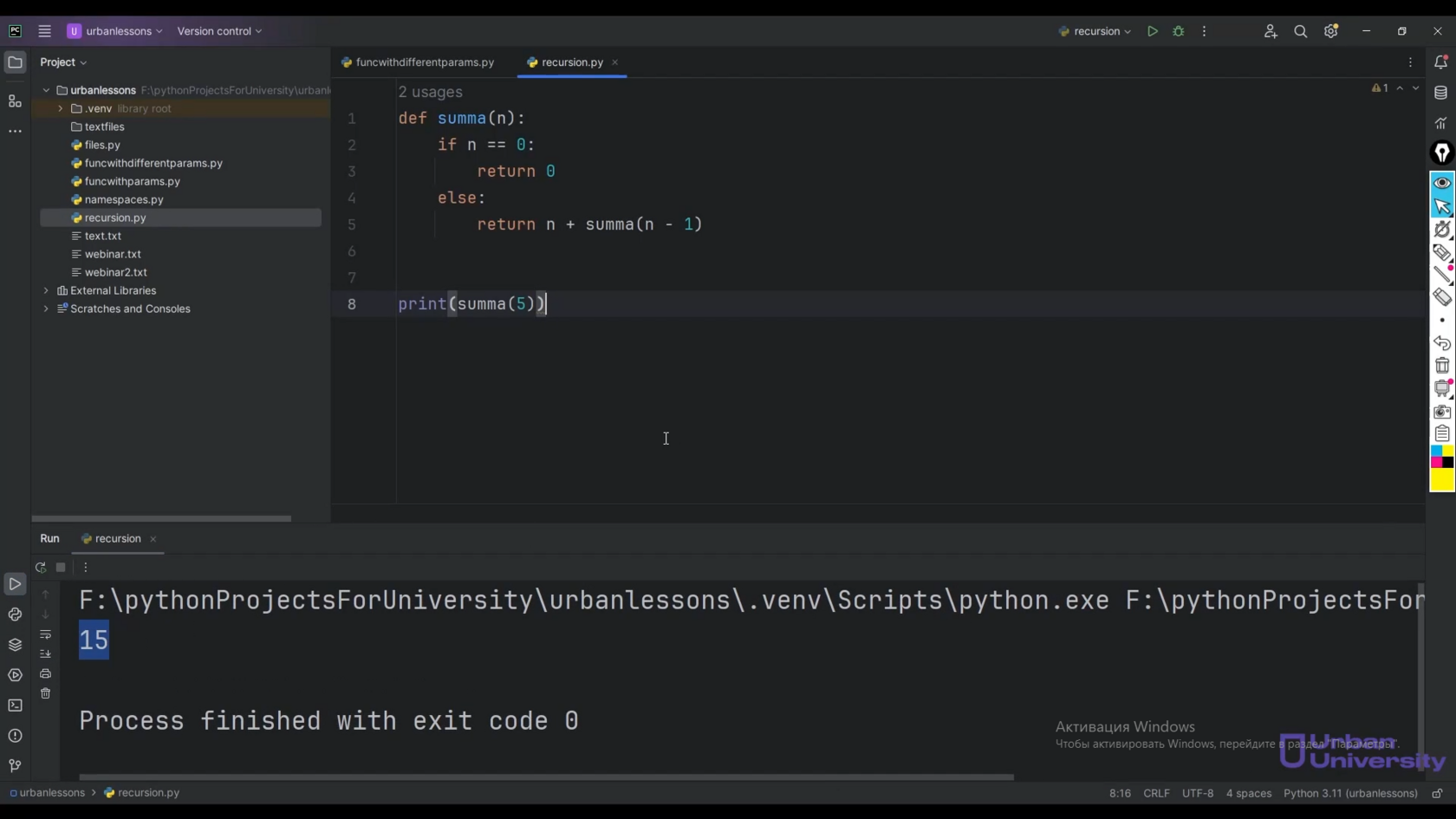


Рис.2

Чтобы понять, как это работает, давайте немного поговорим с вами про**стек(stack)**. Что такое стек?

**Стеком**называется способ организации данных в памяти компьютера. Этот способ работает по принципу последним пришел, первым вышел. Возможно, где-то уже на форумах видели аббревиатуру **“LIFO”(Last in first out)**— это значит, что последний элемент, добавленный в стек, будет взят из него первым. Добавлять новые элементы и удалять существующие мы можем только с одного конца, который называется вершиной.

Самый простой пример стека - детская пирамидка. Вспоминаем детскую игрушку, где имеется стержень и основания. Наша задача из колец собрать пирамидку. Мы начинаем на это стержень накидывать кольца. Первым кидаем самое большое кольцо, потом кольцо поменьше и продолжаем до тех пор, пока мы не дойдем до самого маленького кольца, пока пирамидка не будет готова(рис.3). Таким же образом устроен стек. То есть получается мы добавили элементы и посмотрите на первое кольцо. Кольцо, котором мы положили самым первым, при необходимости мы можем достать только в последнюю очередь, то есть, чтобы получить доступ к этому элементу, к самому первому кольцу, нам необходимо с вершины достать каждый элемент: сначала самое маленькое кольцо, потом побольше и так далее, до тех пор, пока эта пирамидка не будет полностью разобрана(рис.4).



Рис.3

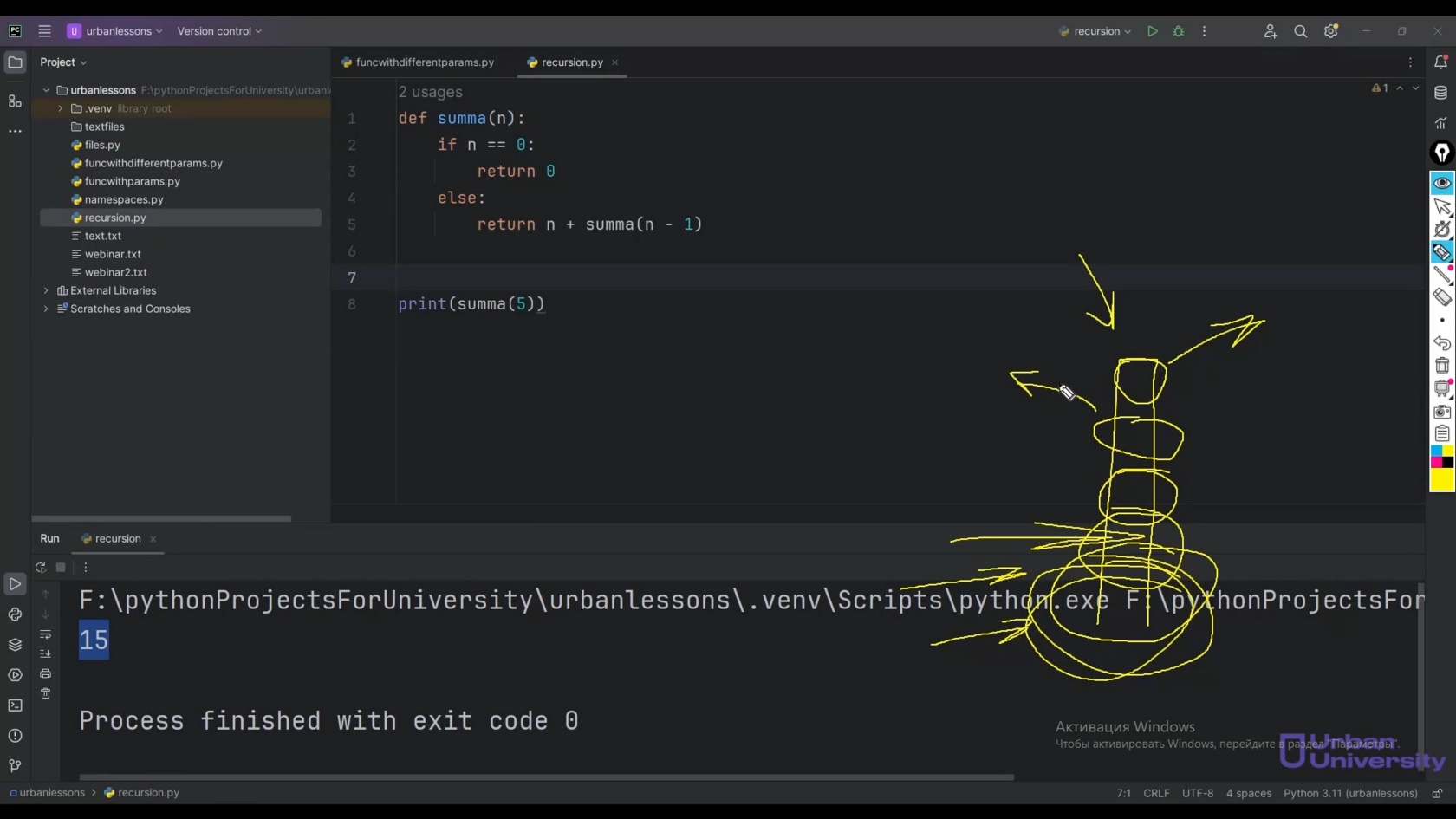


Рис.4

Самый простой пример, если смотреть это в программировании, потому что стек - это некая абстракция, чтобы нам смотреть на него, нам нужна какая-то реализация, при которой мы можем как-то с ней работать. Чаще всего вообще стеки реализуют с помощью динамических массивов и связанных списков. Динамический массив — это такой массив, размер которого может увеличиваться или уменьшаться в процессе выполнения программы. К сожалению, пока что, мы не можем удобно рассмотреть пример реализации стека с помощью динамического массива, но мы можем с помощью простого списка рассмотреть пример.

Отлично подходят **append и pop**. Если вспомнить append и pop — это методы для работы со списком: append - добавляет элемент в конец, а рор - возвращает нам элементы списка.

#### Давайте посмотрим, как выглядит stack.

Создадим переменную stack, изначально будет пустой список и вызовем print(‘Добавили элемент’, stack). Возьмем наш stack и добавим туда 1(stack.append(1)). Снова вызовем print(), добавим к стеку 2. Затем проделаем все действия снова и добавим 3(рис.5).

Запустим, посмотрим: первым элементом была добавлена 1, потом 2, потом 3. Получилась такая своеобразная пирамидка(рис.5).

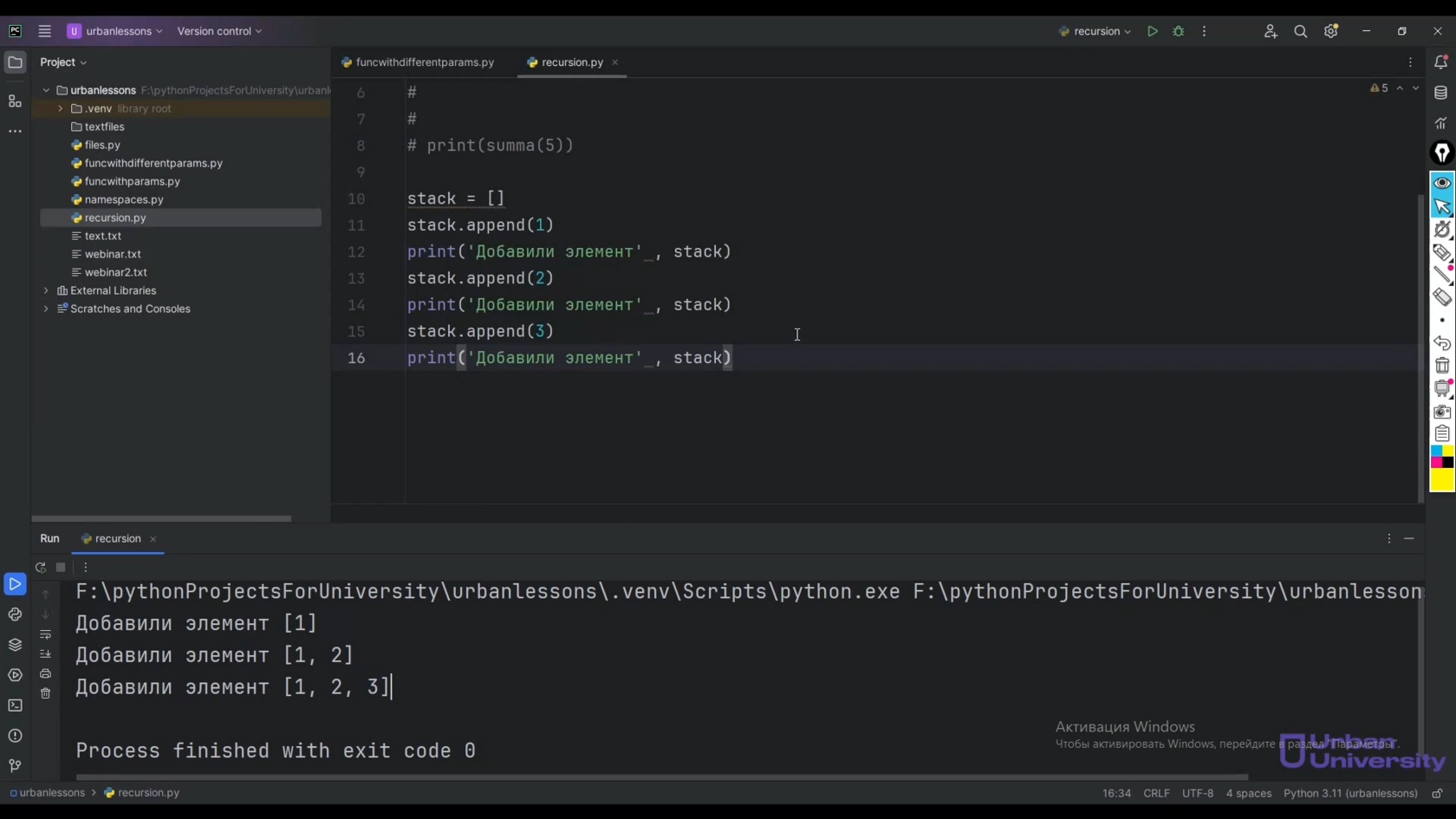


Рис.5

Далее выведем нашу пирамидку полностью(print(stack)) и начнем ее разбирать. Чтобы разобрать пирамидку, будем использовать pop(stack.pop()). Здесь не надо передавать никаких параметров, никаких аргументов, она будет возвращать последний элемент и будем выводить сообщение print(‘Убрали элемент’, stack). Проделаем эти действия еще 2 раза(рис.6).

Запускаем, смотрим: появилась наша пирамидка. Когда мы использовали рор, когда начали разбирать пирамидку, сделали мы это с самого конца, то есть получили сначала последний элемент, после этого следующий элемент и после этого самый первый элемент был убран последним. Вот так это выглядит(рис.6).

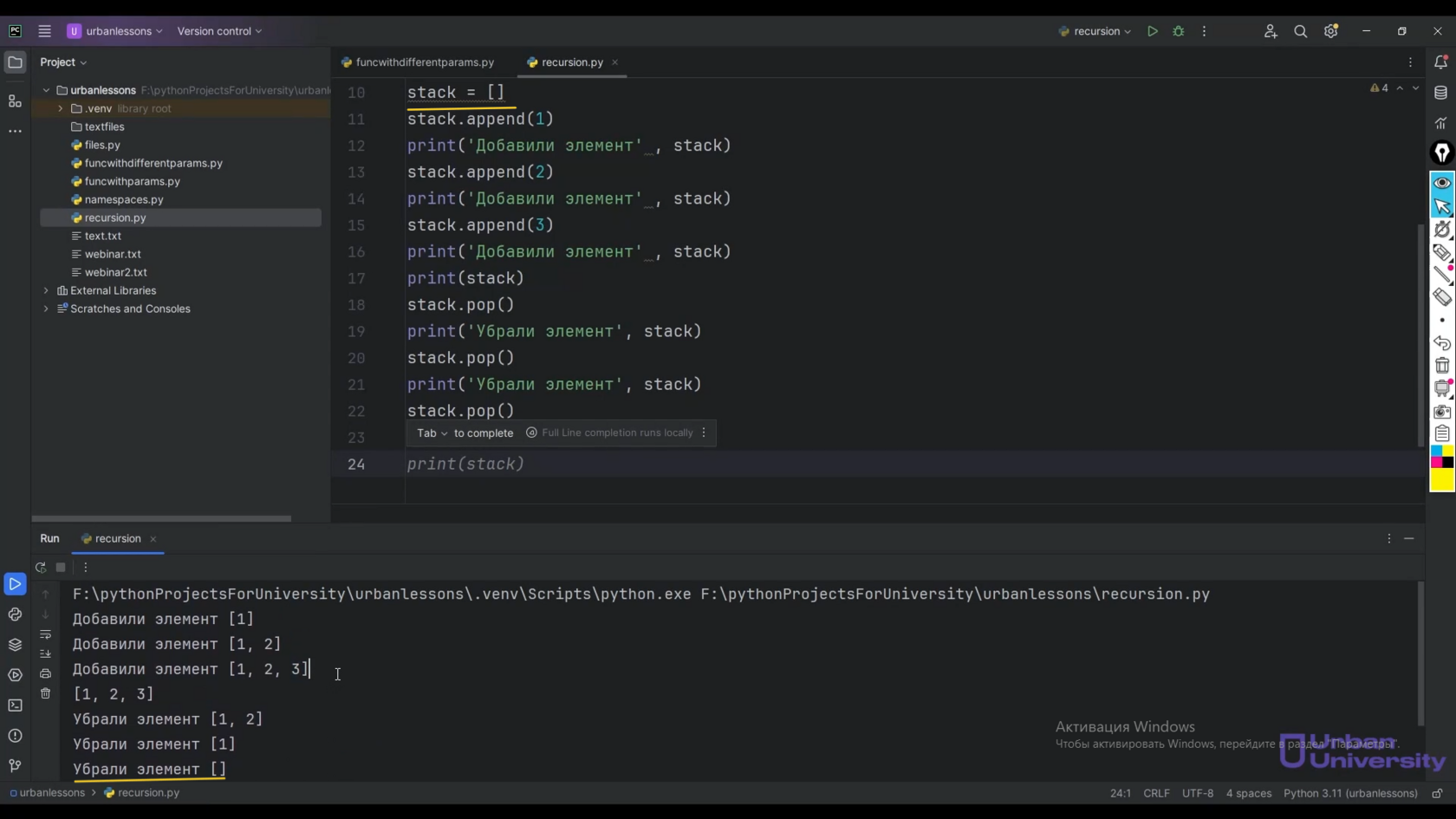


Рис.6

#### Что это такое и что это нам дает?

Здесь дело обстоит в том, что каждый раз, когда мы вызываем функцию, мы забиваем стек вызовов. Стек вызовов — это такая структура данных, которая управляет вызовами функций в момент выполнения программы. То есть, когда компьютер выполняет какую-то программу и доходит до вызова какой-то функции, ему нужно ненадолго переключиться, чтобы эту самую функцию выполнить. Чтобы запомним, где он остановился, компьютер как бы сохраняет в памяти специальные точки перехода и область памяти, где хранятся точки перехода, как раз и называется стеком вызовов. В этой точке будет храниться все, чтобы компьютер быстро смог вернуться к выполнению какого-то основного кода, то есть там могут храниться, например: значения переменных, какие-то аргументы, функции и адрес возврата, то есть, то место, куда компьютер должен перейти после окончания какой-то подпрограммы. Если внутри одной функции, как в данном примере, произойдет вызов другой( n+summa(n-1)), то компьютер ставит на паузу выполнения первой нашей функции( def summa(n): if n==0: return 0). Он сохраняет это в точку перехода стеки вызовов и переходит к выполнению уже функции n+summa(n-1).

Такой принцип работает для любой степени вложенности, то есть если у нас такая рекурсия, тем самым работает как раз этот стек вызовов.

Можно представить это в виде чаши определенной, то есть компьютер дошел до вызова функции. Стек вызовов сохраняется в точке возврата к основной программе. Сейчас должна отработать функция print(summa(5)), и потом дальше мы её уже соответственно выведем(рис.7). В первом вызове функции мы проверяем условия и либо же снова вызываем функцию, либо из нее выходим. В нашем случае мы передали аргумент 5, функция вызывается снова(рис.8). В стек вызовов добавляется точка перехода к нашей первой функции. Потом вызывается вторая функция и в ней снова вызывается функция и снова появляется в стеке вызовов точка перехода и так до тех пор, пока функции не перестанут вызываться, то есть до определенного момента(рис.9).

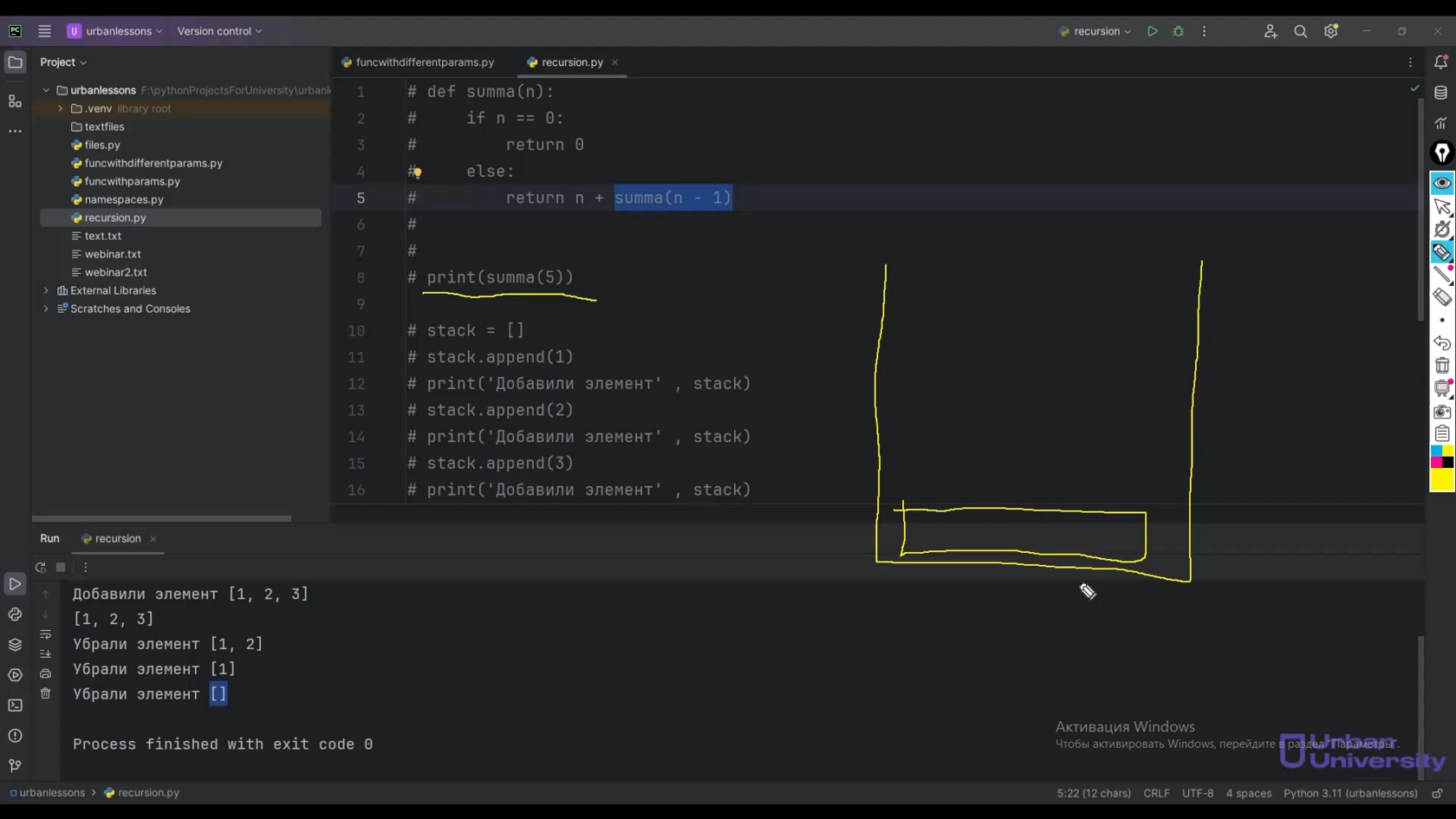


Рис.7

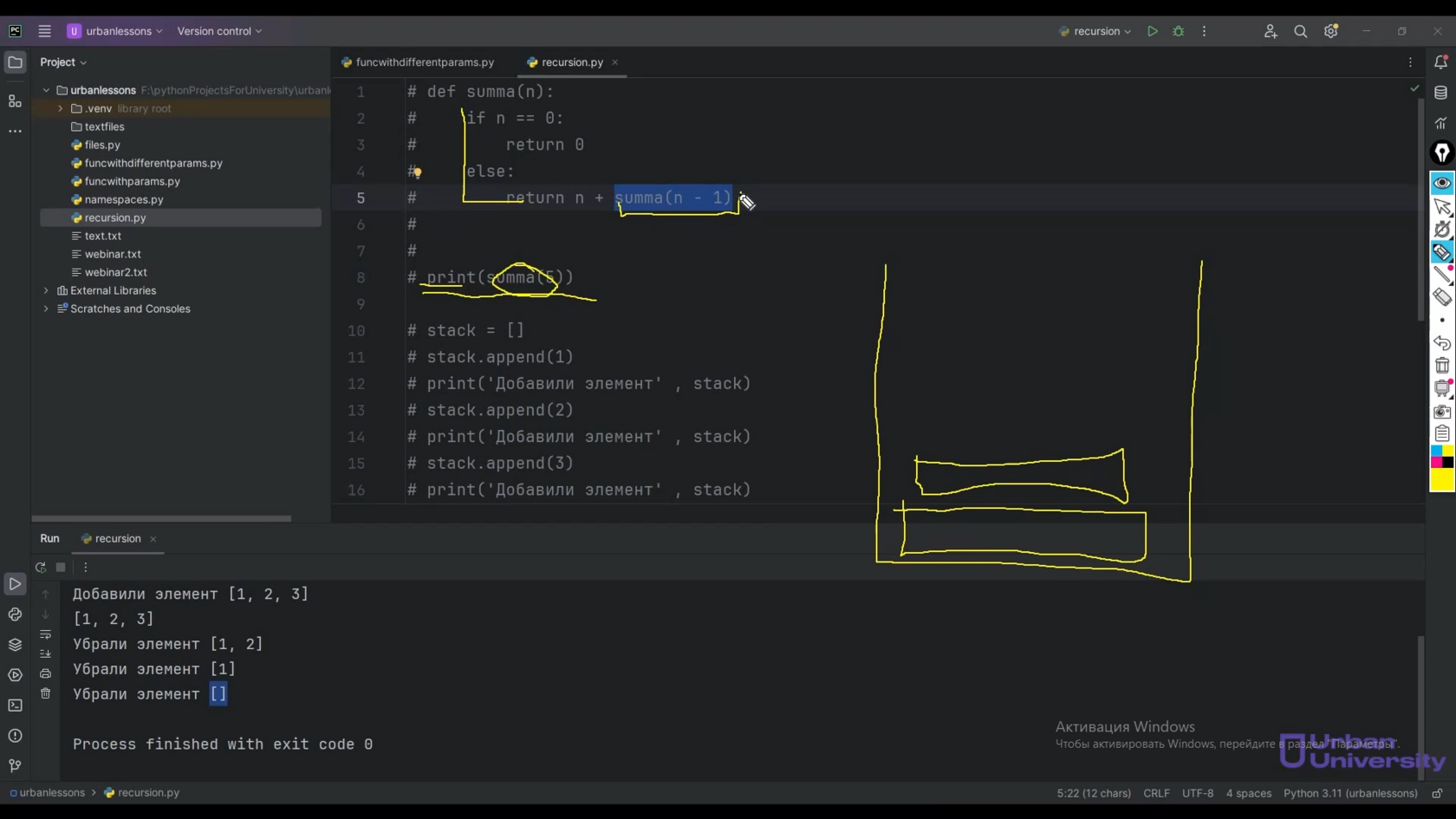


Рис.8

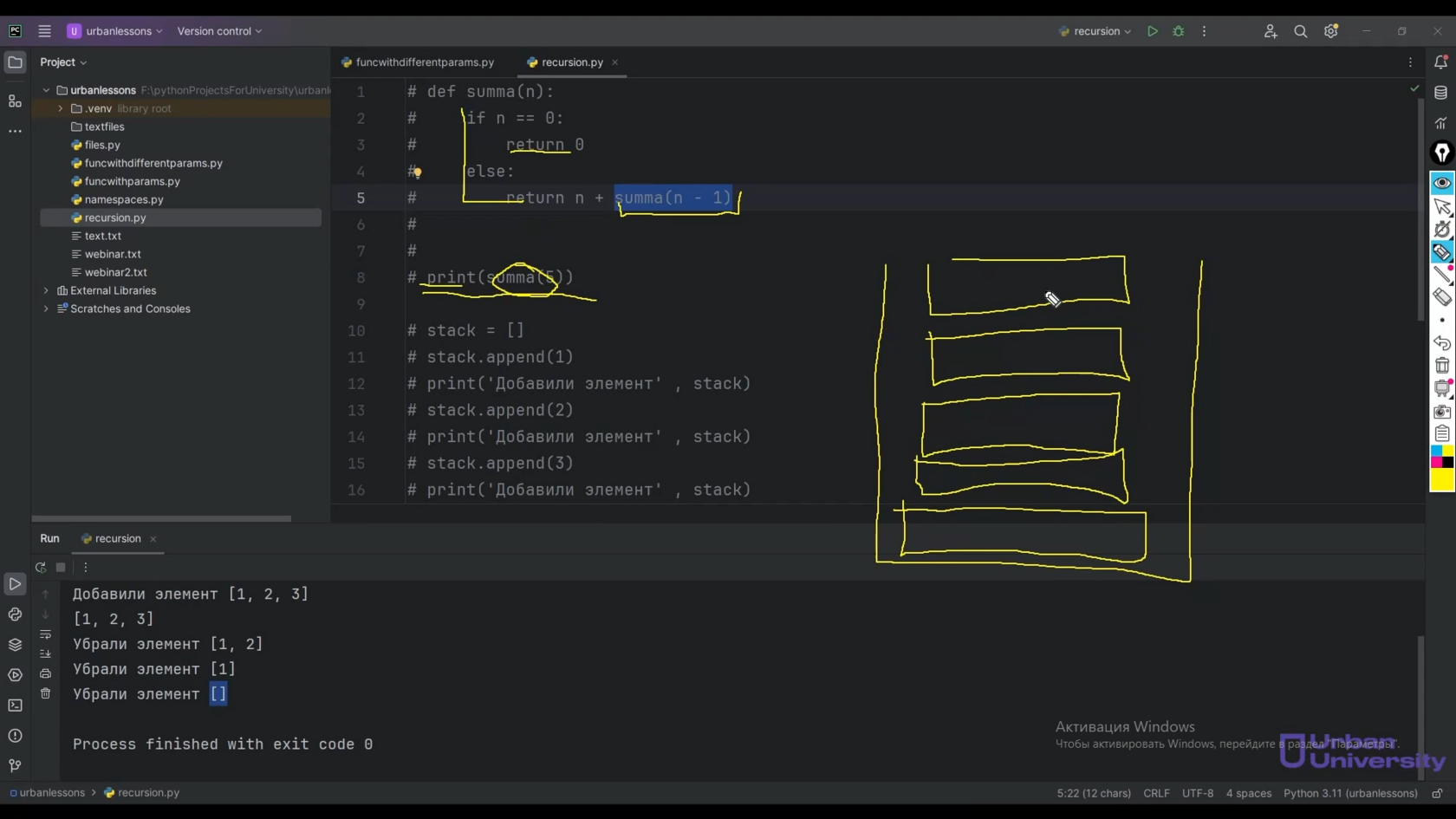


Рис.9

Когда функции перестают вызываться, этот стек вызовов начинает опустошаться сверху вниз, то есть элемент, который был вызван последним, убирается первым и так до тех пор, пока мы не вернемся к нашей основной программе(рис.10).

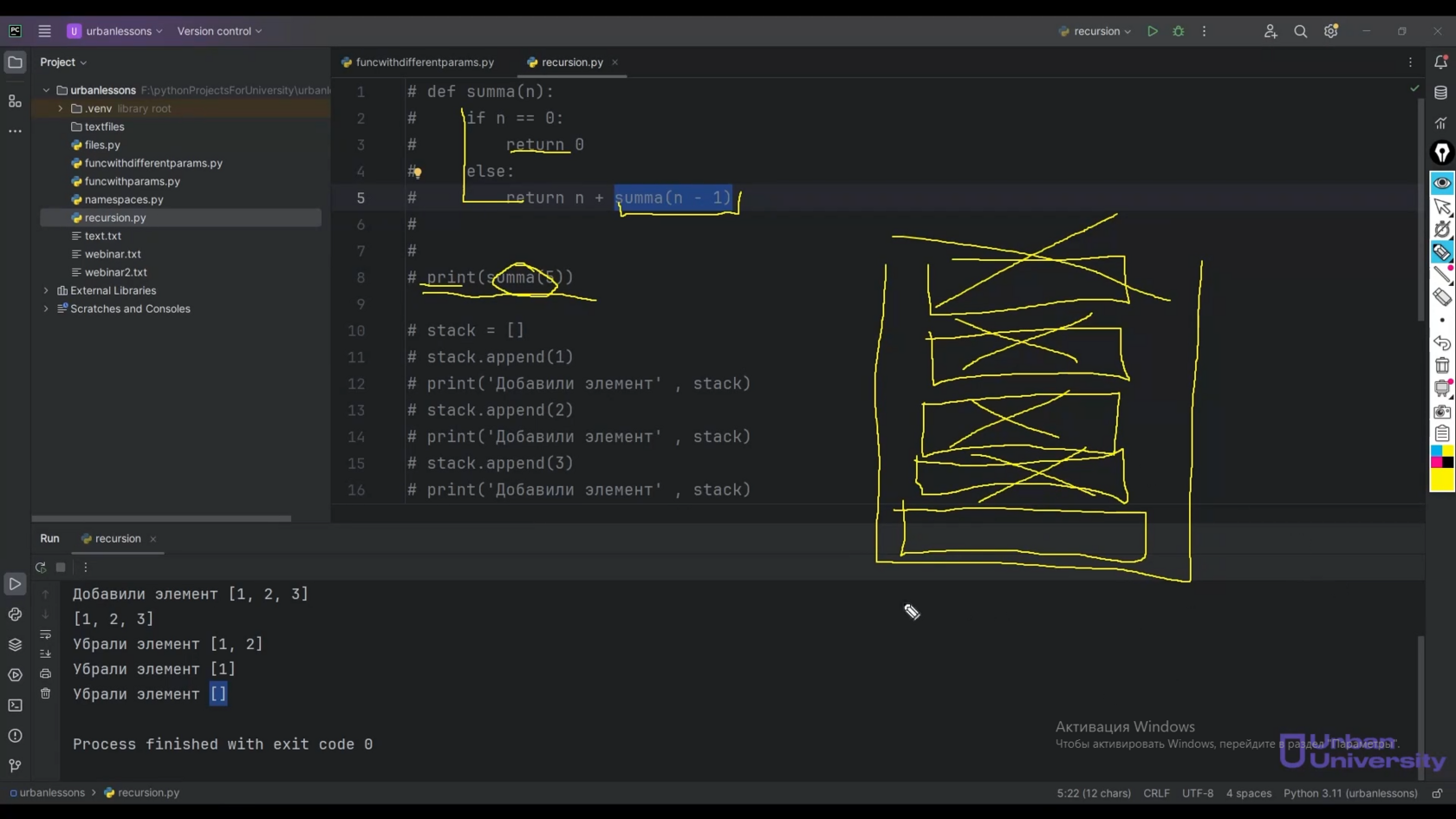


Рис.10

Получается, что мы передали параметр, передали аргумент 5 в нашу функцию, снова сработал вызов этой функции, только с аргументом меньше на единицу, и образуется лесенка, где у нас снова вызывается функция. Так будет работать до тех пор, пока мы не получим в конечном счете 0(рис.11).

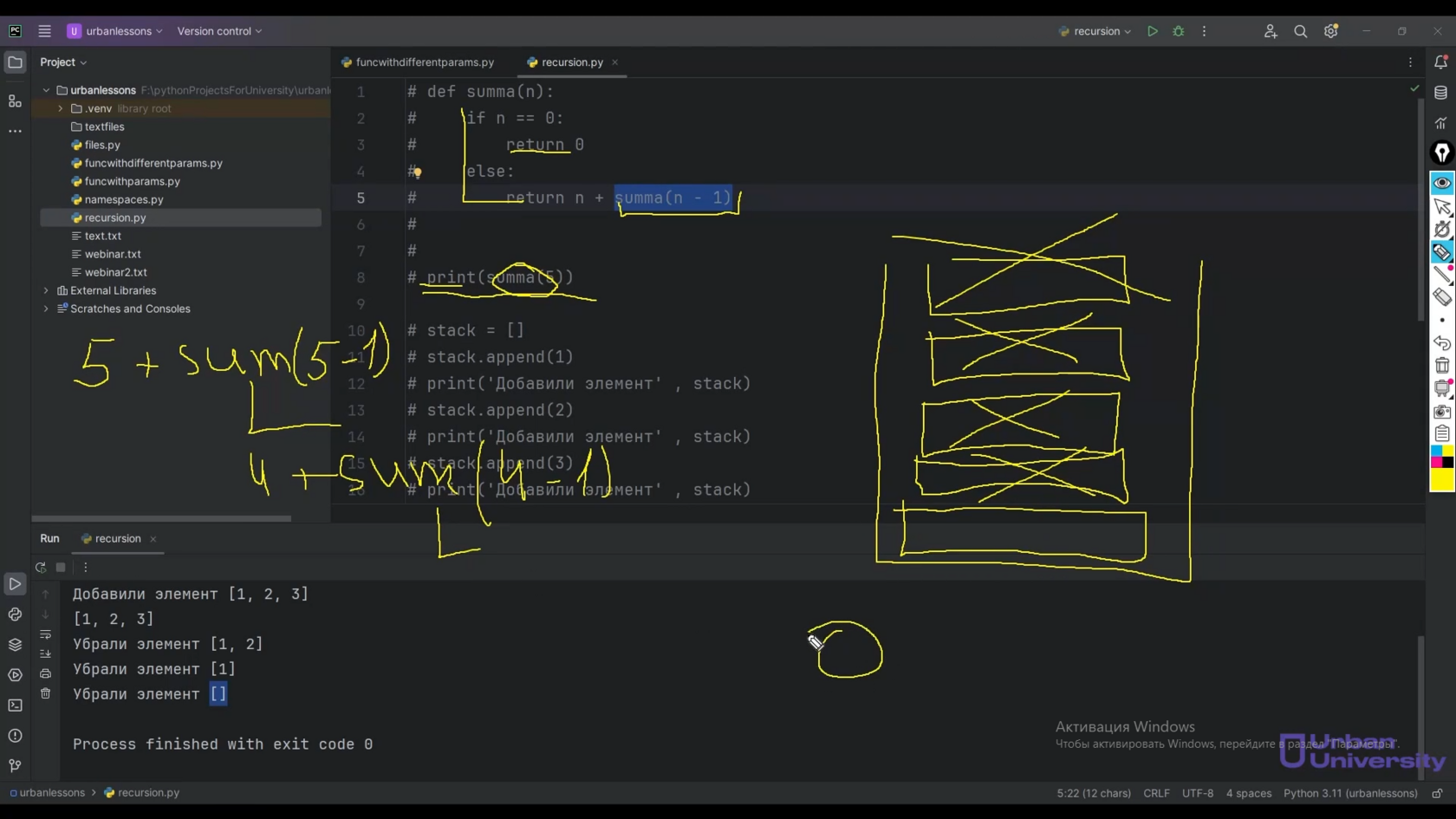


Рис.11

Как только получили 0, мы начинаем возвращаться к нашей программе(рис.12), передавая вот эту самую информацию. Передали наверх 0, здесь передали единичку, затем передали 2, и это все дело будет возвращаться в обратном порядке. В итоге мы получим эту самую заветную сумму.

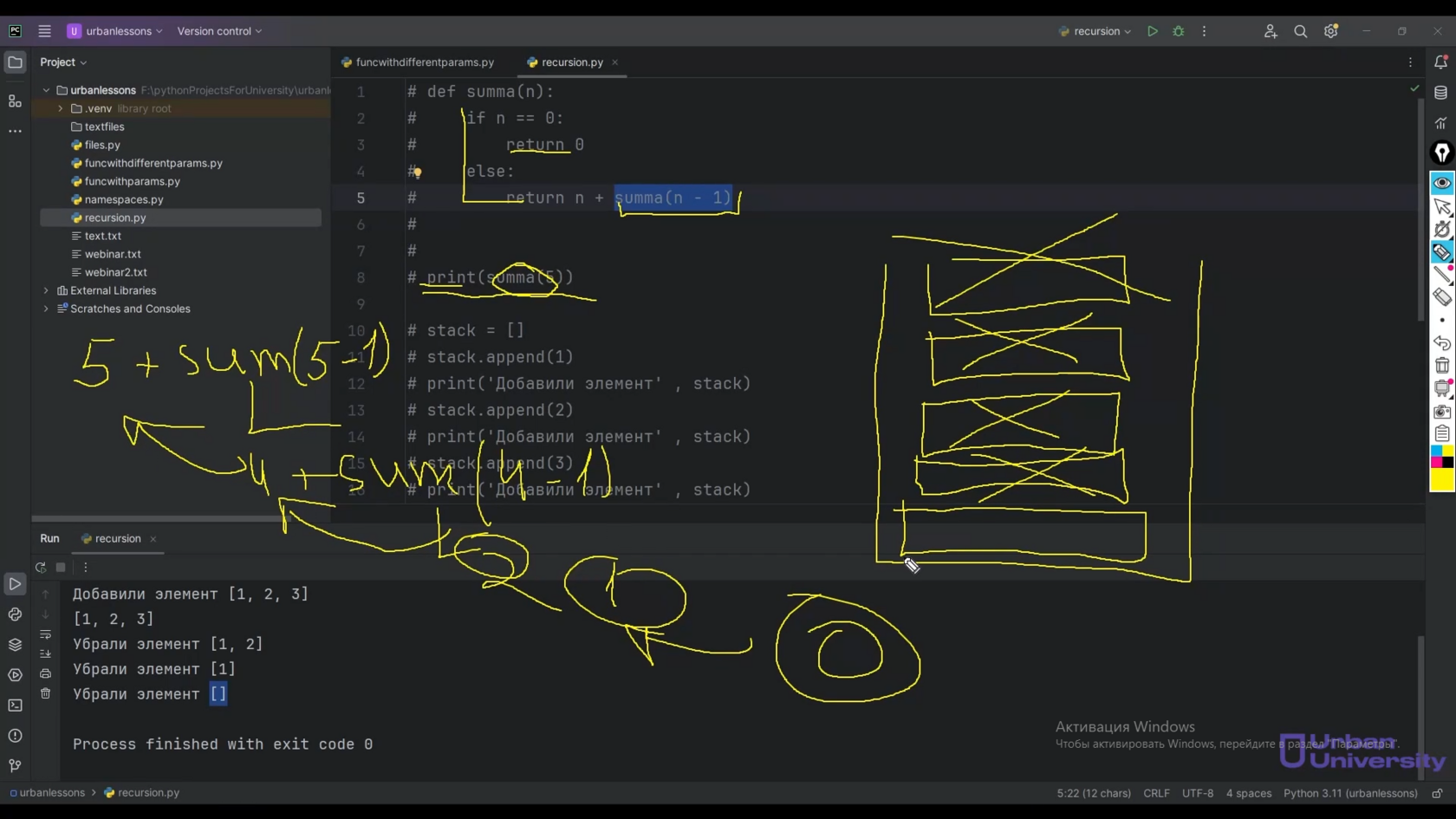


Рис.12

На самом деле, когда есть возможность избежать использования рекурсии, мы избегаем рекурсию. Если говорить другим языком, то в принципе любую рекурсию можно заменить циклом, как и любой цикл, можно заменить рекурсией. Некоторые языки программирования, например, не поддерживают функцию, соответственно там нет рекурсивного подхода. Решить задачу, которую необходимо использовать, скажем так, с помощью рекурсивного подхода мы можем за счёт циклов. Так и в некоторых языках, например, нет циклов и, соответственно, решить задачи, связанные с какими-то циклами мы можем с использованием функции и рекурсивного подхода.

Когда вы знаете о вот этих точках необходимых при создании рекурсивной функции — это границы, когда будет осуществлен выход из функции/из рекурсии и соответственно сама рекурсия, когда функция будет вызывать саму себя(рис.13). Если этих моментов придерживаться, никаких проблем при написании программы при работе с рекурсивными функциями у вас не будет.

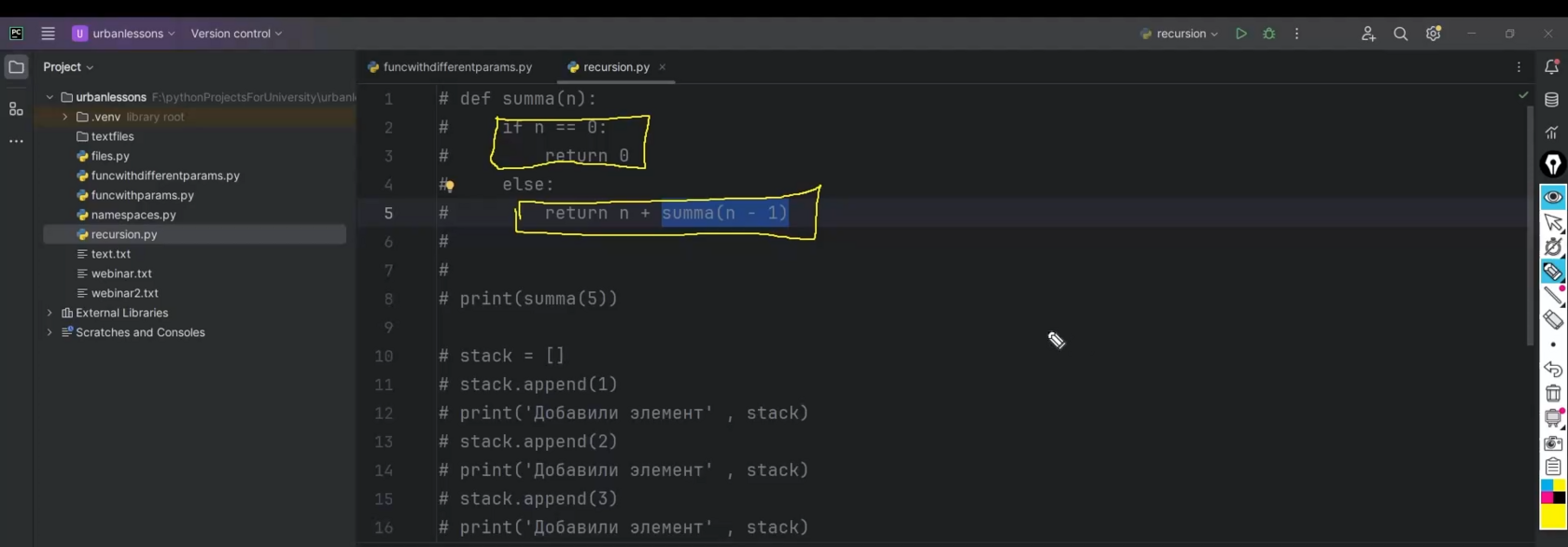


Рис.13

Может быть, кто-то сталкивался с такими проблемами, когда случайно вызвал функцию внутри себя и получали в конечном счёте ошибку. Эта ошибка получается из-за того, что просто переполняется этот самый стек вызовов. Постепенно компьютеру же нужно хранить где-то информацию о том, как вернуть информацию о данных и место в нашей оперативной памяти, где мы храним эту информацию. Оно, естественно, ограничено и рано или поздно этот стек может переполниться, если не будет осуществляться выход из этих самых функции.

Когда мы вызываем функцию внутри себя и при этом нигде из нее не выходим, в конечном счете мы просто получаем выход из программы по случаю заполнения этого самого стека вызовов. Наша задача просто быть готовыми.

В будущем мы рассмотрим примеры, где ресурсы применяются повсеместно, где без них не обойтись. Сейчас наша задача была познакомиться с рекурсией, чтобы мы могли с этим бороться. Поэтому, если вы видите ошибку maximum recursion depth exceeded(рис.14), значит где-то вы не вышли из функций.

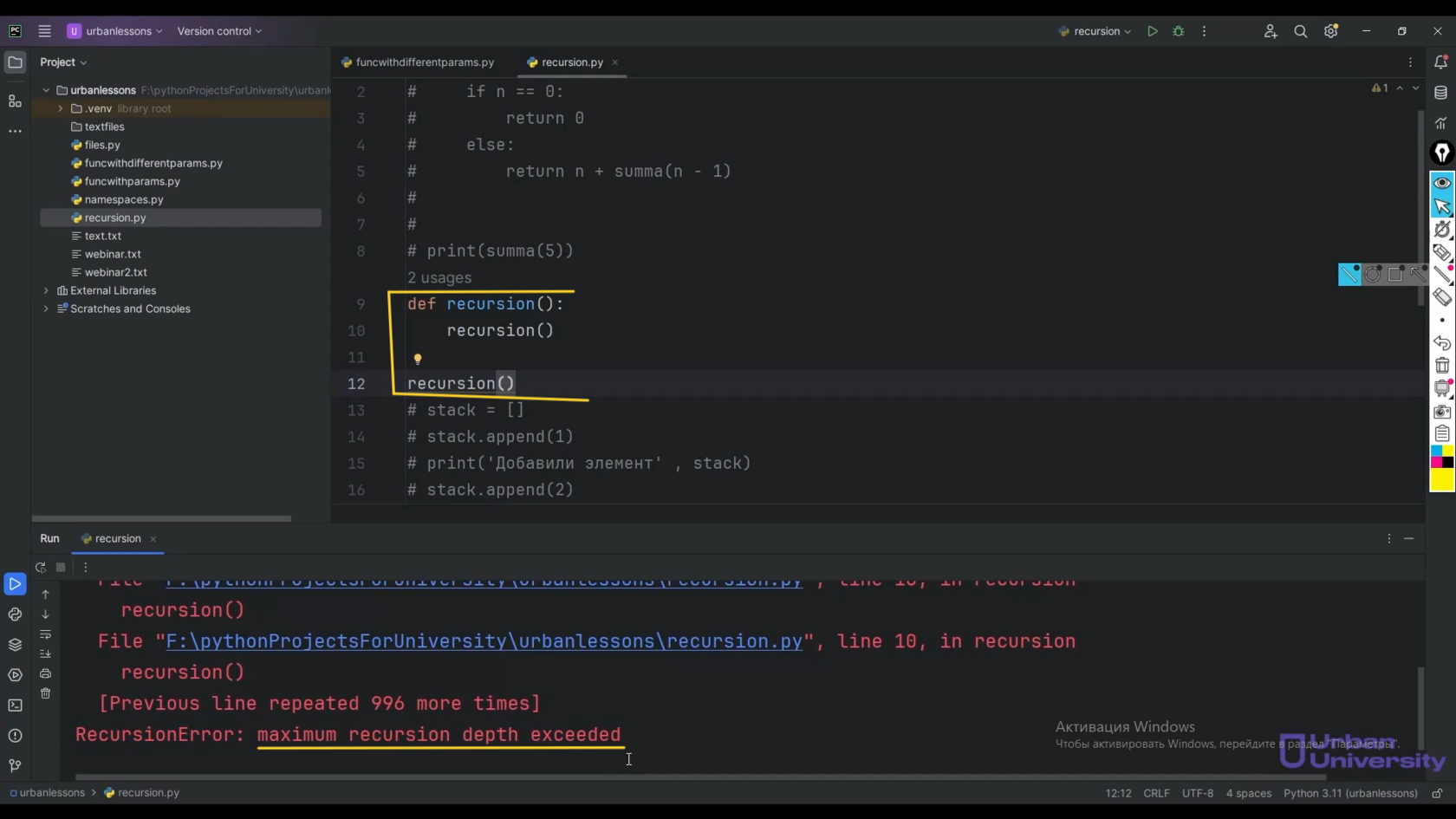


Рис.14