B4

В этом модуле вы узнаете:

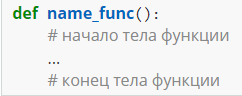
Что такое функция и как её определять.

Как вызывать функции с разными параметрами, в том числе запакованными.

Что такое рекурсия, как и для чего её использовать.

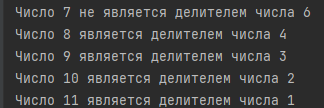
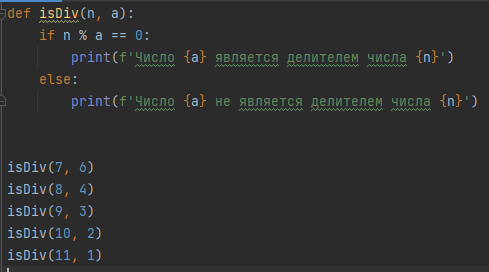
Что такое итераторы, генераторы и декораторы

**Функциональное программирование в Python. Понятие функции**



Если функция имеет аргументы, но их не передать или передать большее количество, то будет ошибка.



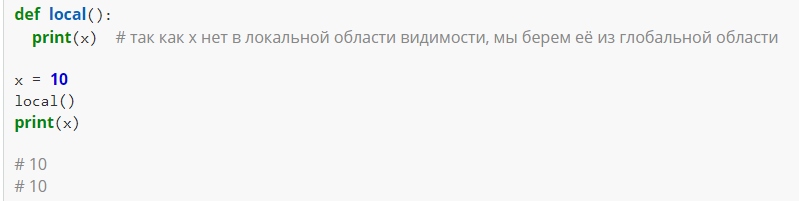


Любая функция в **Python** что-то возвращает. Если не указан return, то функция вернет **None.**

**Области видимости, запакованные переменные, аргументы по умолчанию и рекурсия**

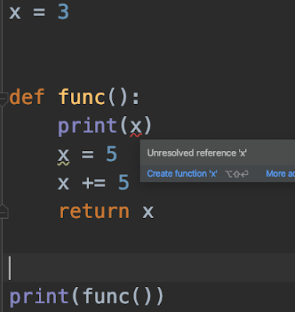
Области видимости определяют, в какой части программы мы можем работать с той или иной переменной, функцией и т. п., а от каких они «скрыты». Крайне важно понимать, как использовать только те значения и переменные, которые нам нужны, и как интерпретатор языка себя при этом ведет.

В python3 есть 3 области видимости. **Локальный, глобальная, нелокальная.**

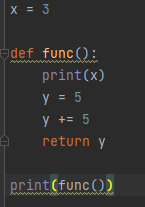


Тут в функцию local никакие переменные не передавались, тем не менее функция увидела переменную и напечатала ее. Просто сначала она проверил свою область видимости и когда не нашла там переменной, то обратилась уже к глобальной области видимости.

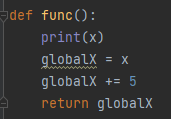
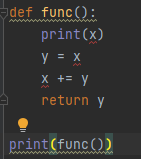
А такой код выполняться не может.



Так как в теле функции найдено объявление переменной, но осуществлена попытка распечатать ее до объявления. Вот как этот код можно исправить.

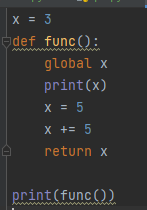
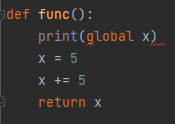


**Глобальные переменные внутри функций изменять нельзя.**

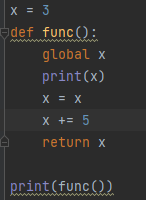
****

**Но обращаться к ним можно.**

Еще можно явно указать, что переменная является **глобальной**, тогда и проблем с названием не будет. Но нельзя сделать это внутри функции **print,** как минимум



**Это значение можно даже присвоить.**

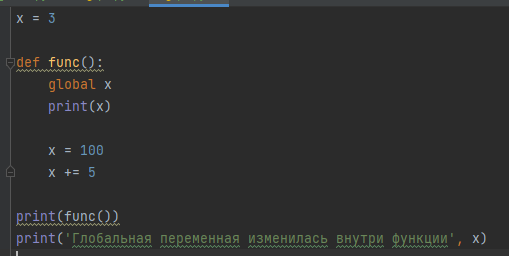
****

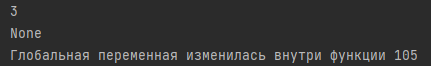
**Но так уже становится трудно воспринять этот код, так еще можно сделать еще хуже.** Этот код изменил переменную 3 на 8, вне функции.





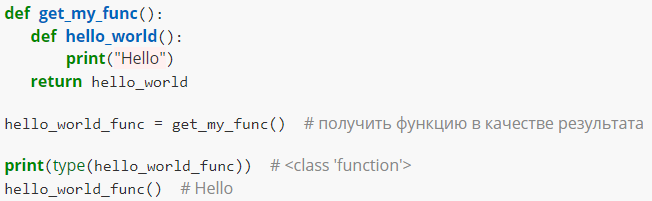
Да даже так, происходит изменение глобальной переменной!



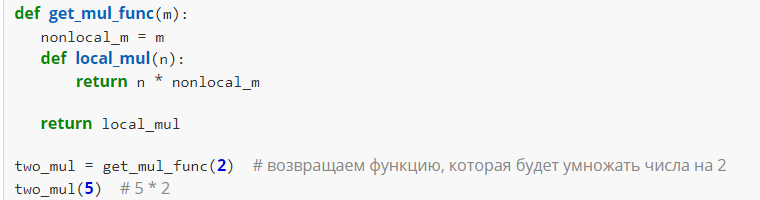


**Нелокальная область видимости**

Появилось это понятие в Python 3 вместе с ключевым словом nonlocal. Используется чаще всего во вложенных функциях, когда мы хотим дать интерпретатору понять, что для вложенной функции определенная переменная не является локальной, но она и не является глобальной в общем смысле. Вот пример функции, которая возвращает функцию.



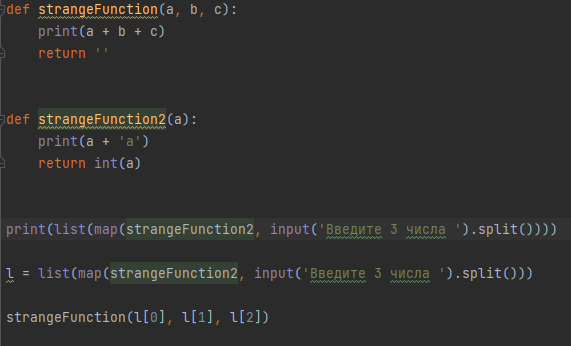
А nonlocal нужен вот для таких случаев.



**Использование переменных объявленных вне функции называется замыканием функций.**

Хочу написать функцию, в которую буду передавать распакованный список.

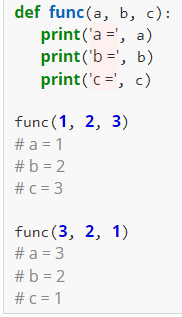
Пока получилось только так



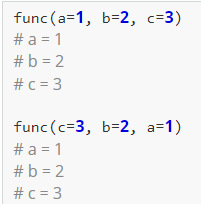
Как же мне распаковать список….

**Запакованные переменные, или что такое \*args и \*\*kwargs**

Это как раз по коду, который я не помню как оформить, тот что выше. Во первых в функцию можно отправлять значения просто через запятую. Это позиционный метод.

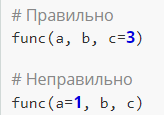


А можно строго по именам, именованный.

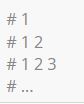
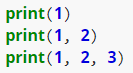


Используя именованный, порядок не нарушится.

**Важная особенность: все именованные аргументы должны идти строго после позиционных, как при объявлении функций, так и при их вызове.**

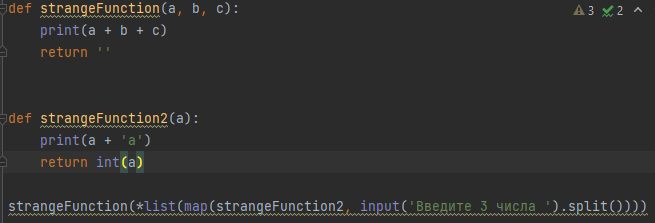
****

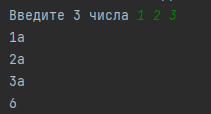
**А теперь. Как написать функцию, работающую как print()**



Оператор **\*** чаще всего ассоциируется с операцией умножения, но в Python он имеет и другой смысл. Этот оператор позволяет **«распаковывать»**

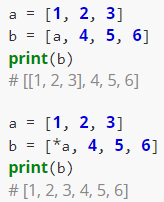
**О, то что надо.**





Хорошо, но ведь если я введу 4 числа, то жопа, верно? Нужно и в функцию передать их.

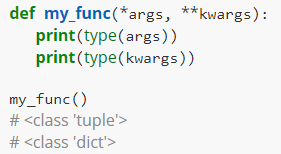
Чтобы было понятее, вот:



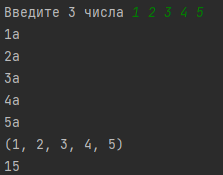
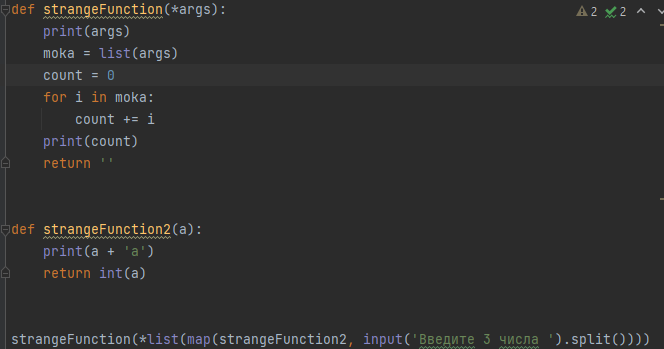
**Для неименованный аргументов \*args, для именованных \*\*kwargs**

Это не зарезервированные слова, просто хорошая практика. args — это сокращение от «arguments» (аргументы), а kwargs — сокращение от «keyword arguments» (именованные аргументы)

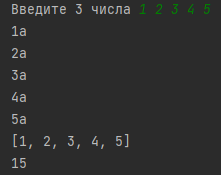
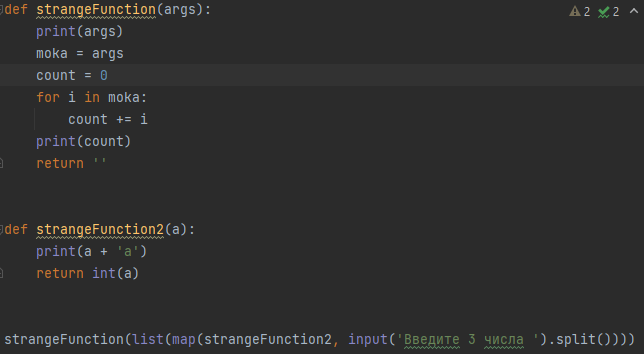
Разница в том, что одна обращает в список, другая в словарь.



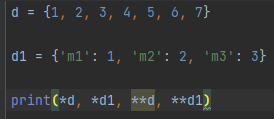
Ну вот:



Но если подумать, то вместо этого я просто мог бы отправлять в эту функцию список и все:

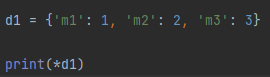


Ладно, просто попробовать распаковать словарь.



Ага, нужны именно две \*\*.

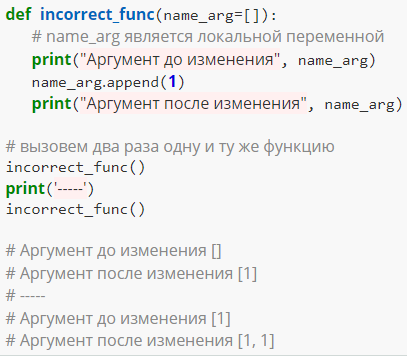
 Так тоже нельзя. Видимо только со словарем.

 Понятно, так можно получить ключи.

**Изменяемые типы данных как аргументы по умолчанию**

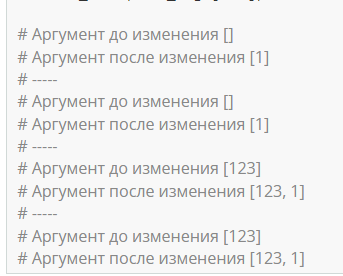
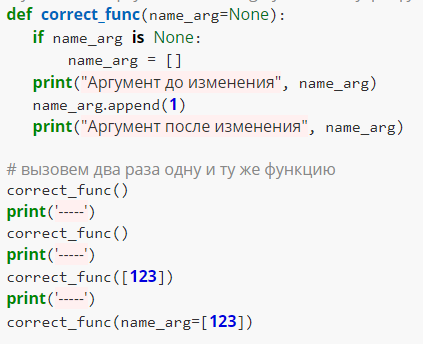
Если в качестве аргументов по умолчанию использовать изменяемые типы данных (списки, словари), то они создаются один раз и будут использоваться на протяжении времени жизни функции. Так как при передаче словаря передаются не все его значения, а указатель на первый его элемент.

**Вот пример неправильной функции.**

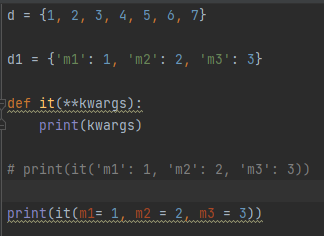
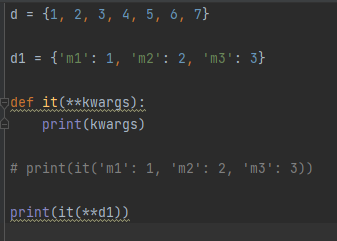
****

Функция при первом запуске создала в памяти пустой список. И он так и остался в памяти, поэтому после вызова функции второй раз используется все тот же список, а не новый.

Если в функции все же нужно использовать список, то следует писать так:



Кварги только так можно писать:





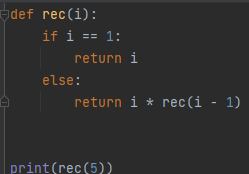
**Полезная информация.**

**Рекурсивные функции**

Рекурсивная функция — это функция, вызывающая сама себя и обрабатывающая полученный результат до тех пор, пока не достигнем терминального условия (условия остановки).

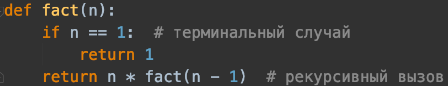
Количество раз, сколько функция вызвала сама себя, называется глубиной рекурсии.

Ладно, как обычно, напишу рекурсивную функцию нахождения факториала.

Эм… вышло быстрее чем я думал.

У них чуть симпатичней



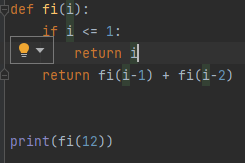
**Память компьютера не бесконечна, поэтому после определенного числа рекурсивных вызовов, если задача будет не выполнена, вы получите ошибку RecursionError, которая защищает вас от переполнения памяти.**

**Числа фибоначчи и золотое сечение**

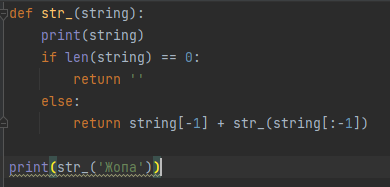
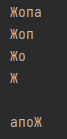
****

Число является результатом сложения двух предыдущих чисел

Если поделить два соседних числа, то всегда будет число в районе 1.6Это золотое число, или золотое сечение.

**С помощью рекурсивной функции развернуть строку.**

** **

**B4.4. Итераторы и генераторы, оператор yield**

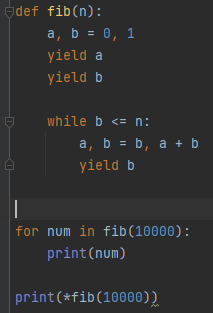
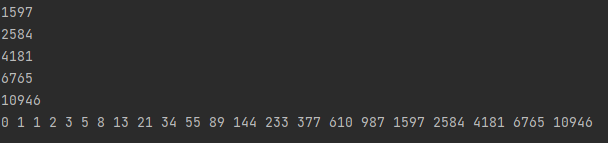
**Функции-генераторы**

Их задача выполнять те действия, которые прописаны в теле функции-генератора, дальше уходить в «спячку» и ждать следующего вашего обращения. Такой процесс выполнения называют ещё «ленивыми вычислениями», то есть вычислениями только в момент обращения, а не всего сразу, как это делают обычные функции.

Синтаксис отличается от обычных только наличием оператора **yield. Он заменяет return.**

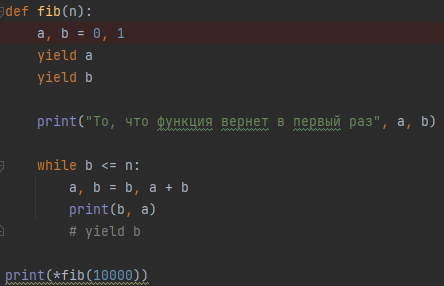
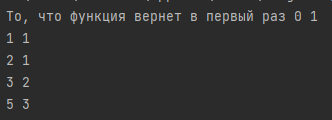
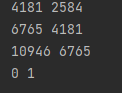
Встречая же **yield**, программа понимает, что функцию-генератор, закрывать не нужно, а нужно приостановить и затем вернуться к ней.

Хм.. тут через нее написали нахождение числа фибоначи, но кажется в виде массива? Ща…

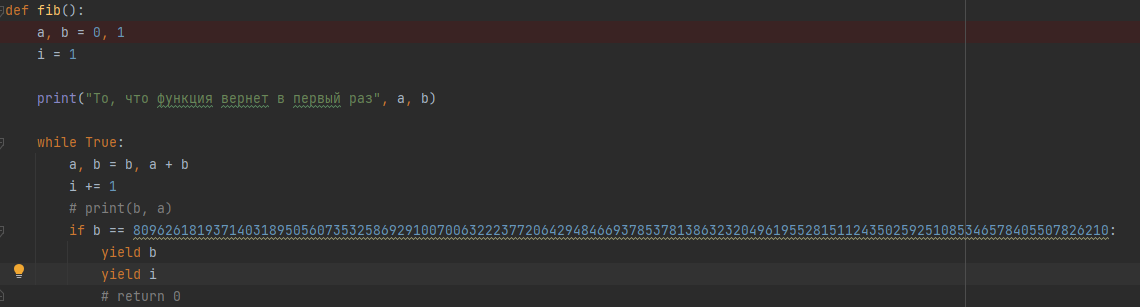
Да, не плохо, хотя суть логики работы не до конца понятна. А если пройти в отладчике.

После первой инициализации к первым строкам при вызове функций не возвращались, а сразу переходили к циклу. А и b в каждом новом цикле сохраняют свои значения. Функция вообще не вызывается заново. Цикл нужен, чтобы выводить получаемые из функции значения. Так вот каждое значение **yield** будет возвращаться за пределы функции.

То есть сначала выполнились все действия, а уже потом вывелся результат, несмотря на то, что yield был вызван до цикла, выполнению нужно окончание. Если окончания не будет, то функция ничего и не выведет.

К примеру вот код:



Он будет выполняться бесконечно и так и не выведет результат Кроме первого сообщения:



Хотя условие b и выполнилось, чтобы этот результат получить, нужно остановить функцию. Это можно сделать даже обычным **return. Тот что закомменчен.**

Тогда результат выведется.

Кому интересно, это число фибоначи в 585 итерации.

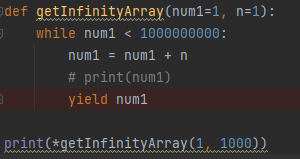
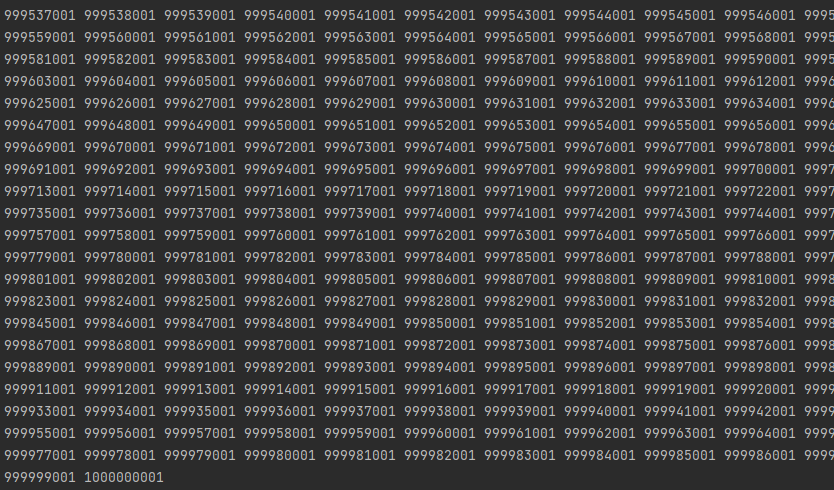
Также хочу отметить, что для вывода числа из такой функции, а это функция напомню **генератор,** нужно либо использовать цикл или указатель. Но, через цикл можно сразу вывести значения, а указатель не вернет значение до тех пор, пока функция не остановится.



Без него получим информацию, что это **генератор**.

**Создать функцию-генератор, возвращающую бесконечную последовательность натуральных чисел. По умолчанию, она начинается с единицы и шагом 1, но пользователь может указать любой шаг и любое число в качестве аргумента функции, с которого будет начинаться последовательность.**

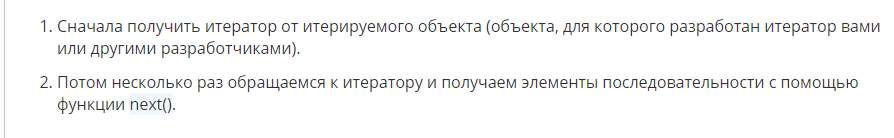
** **

**Итераторы**

Это объект, который возвращает свои элементы по одному за раз.

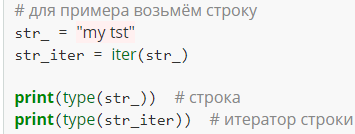
Функция **iter()** возвращает **iterator** если объект является итератором.

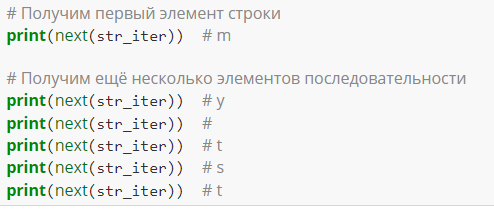




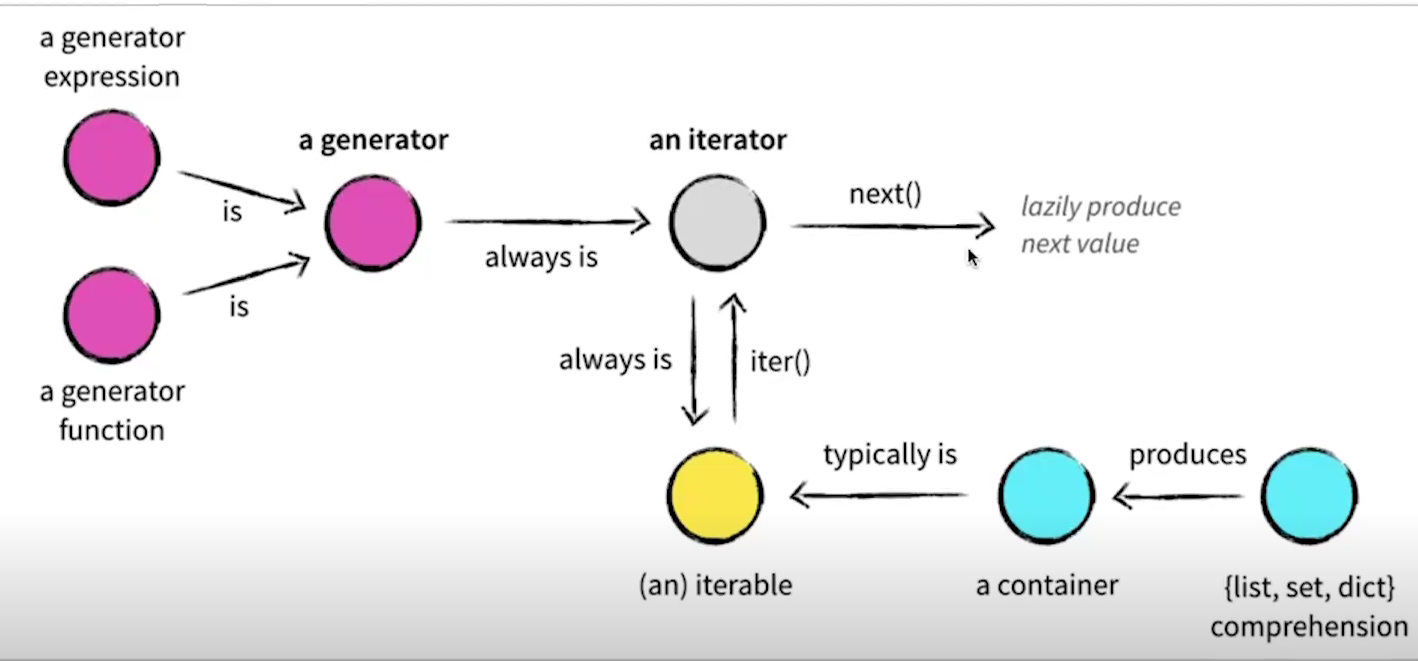
А было ведь такое, кажется **next** можно использовать как **continue.**

Вот как это выглядит:





Если после этого попробовать получить еще один объект будет ошибка

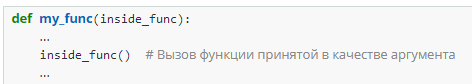


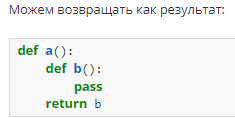
**B4.5. Замыкания и декораторы**

Ладно, продолжим полагаю. Два новых понятия:

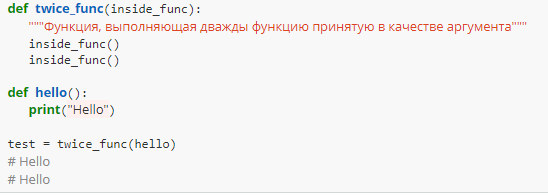
**Функции высших порядков** и **замыкание функций.**

**Функция высшего порядка –** в программировании это функция, принимающая в качестве аргументов другие функции или возвращающая другую функцию в качестве результата.





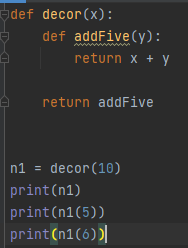
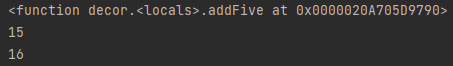
Так к примеру можно сделать функцию, которая выполняет другую функцию два раза.

****

**Замыкание в программировании –** это функция, в теле которой присутствуют ссылки на переменные, объявленные вне тела этой функции в окружающем коде и не являющиеся ее аргументами.

**(Эм..** а я думал это когда функция вызывает саму себя и там рекурсия**).**

Ну вот функция в которое записывается число и это числу складывается с другим числом, которое передается в другую функцию, которую эта функция возвращает.

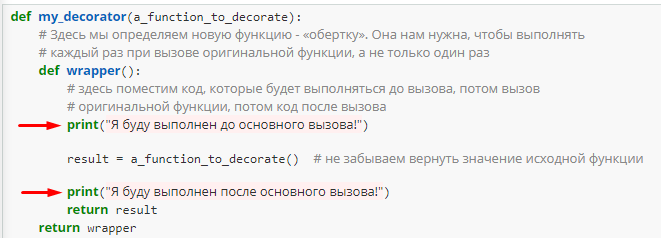
 

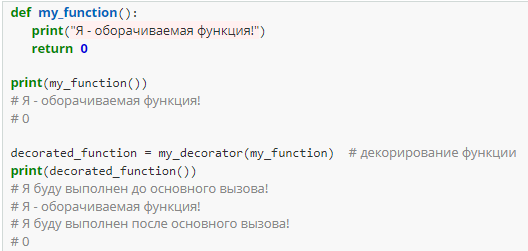
Звучит слегка сложно. Но по факту, просто возврат функции.

**Декораторы** предназначены для подключения любого дополнительного поведения к основной функции, называемой декорируемой функцией. Оно может выполняться до, после или вместо самой декорируемой функцией. Исходный код декорируемой функции никак не изменяется.

Так можно к примеру к функции подключить декоратор, который будет вычислять время выполнения декорируемой функции.

Все же мне не нравятся декораторы, больно они путают. Но вот пример:





**Если на место двух красных стрелок вставить начала таймера и конец таймера, то можно узнать, сколько выполнялась оборачиваемая функция.**

Еще из возможностей декораторов, это то, что с помощью них можно также сохранять результаты вычисления (кеширование).

**import time**

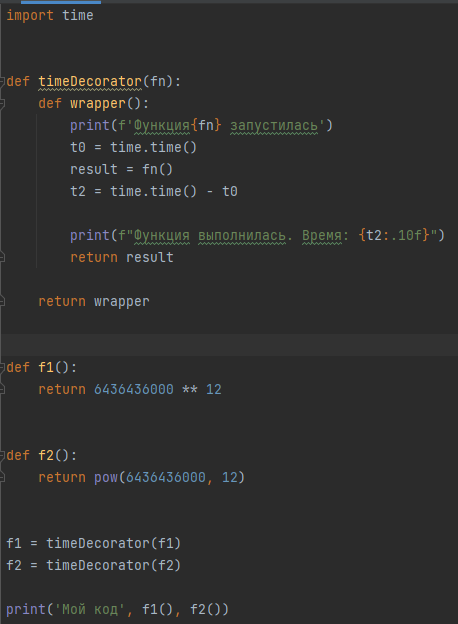
**time.time()**

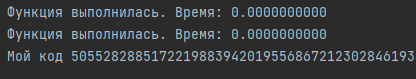
Вот эти две функции строки и с помощью них, напишу декоратор, который сравнивает время функций pow() и оператора \*\*. Оба для возведения числа в степень

Так вот. Временной декоратор у меня не получился(

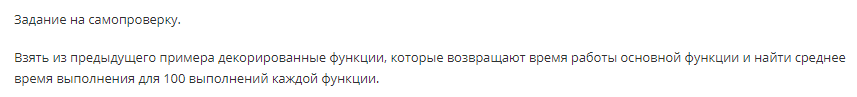
Точнее вроде все корректно, но разница между началом и концом все время ноль. Вот сравнение кода:



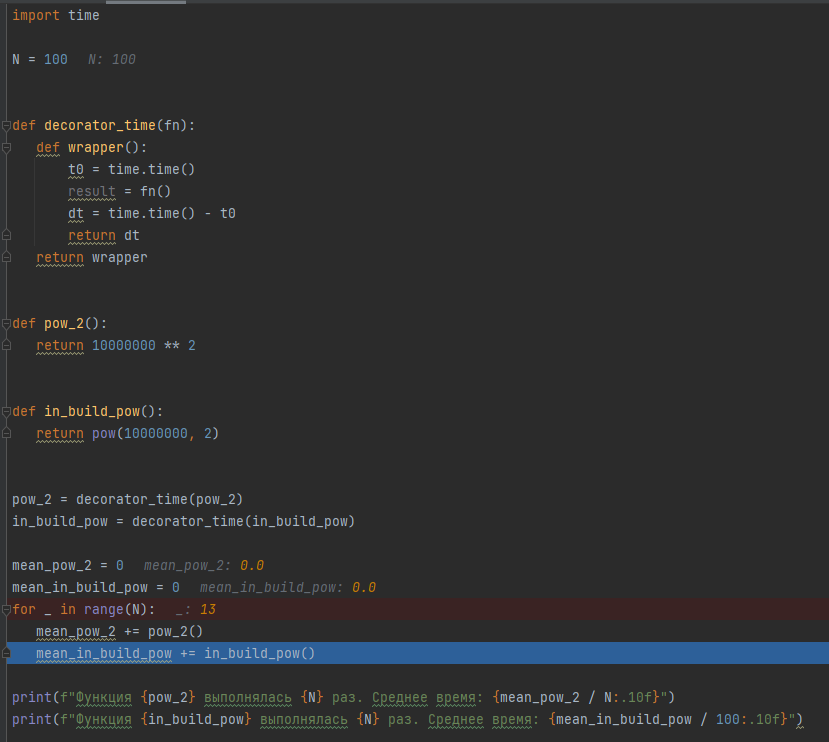




Ладно. Задания:



Честно, было лень, я сразу посмотрел решение. Но просто не было мотивации, т.к. у меня результат ноль даже с их кодом, видимо дело в интерпретаторе python. Мб в версии.

 Интересен тут только **for,** я гадал, что это за **\_** такой. А как оказалось, это такая практика, если нужен цикл **i** раз, но при этом сам **i** не используется. Ну.. я так думаю.

