# **Менеджеры контекстов. Оператор with**

Курс по Python ООП: <https://stepik.org/a/116336>

[Смотреть материал на видео](https://www.youtube.com/watch?v=FkhnVkl0EgM&list=PLA0M1Bcd0w8zPwP7t-FgwONhZOHt9rz9E)

На этом занятии речь пойдет о, так называемых, менеджерах контекста. С менеджером контекста мы с вами уже сталкивались, когда рассматривали работу с файлами. Дело в том, что когда открываем файловый поток с помощью функции open(), то в конце работы с ним, его желательно закрыть с помощью метода close(). Если реализовать эту логику через конструкцию try/except/finally, то получим примерно вот такой текст программы:

fp = None**try**:     fp = open("myfile.txt")     **for** t **in** fp:         **print**(t)**except** Exception **as** e:     **print**(e)**finally**:     **if** fp **is** **not** None:         fp.close()

Благодаря блоку finally мы гарантированно закрываем файл, даже если в блоке try возникло какое-либо исключение. Но, если воспользоваться файловым менеджером контекста, то программа принимает вид:

**try**:     **with** open("myfile.txt") **as** fp:         **for** t **in** fp:             **print**(t)**except** Exception **as** e:     **print**(e)

Видите, она фактически отличается от первой только тем, что не реализует блок finally. Но почему? Нам же нужно закрыть файл после работы с ним? Да, и файловый менеджер контекста делает это автоматически вне зависимости от возникновения возможных исключений. Как же в деталях это происходит? Для этого мы вначале посмотрим, как вообще создаются свои собственные менеджеры контекстов.

В целом менеджер контекста – это класс, в котором реализованы два магических метода:

\_\_enter\_\_() и \_\_exit\_\_()

Когда происходит создание менеджера контекста с помощью оператора with, то автоматически вызывается метод класса \_\_enter\_\_. А когда менеджер контекста завершает свою работу (программа внутри него выполнилась или произошло исключение), то вызывается метод \_\_exit\_\_. Как видите, все предельно просто. И, как вы догадались, в файловом менеджере происходит закрытие файлового потока именно в методе \_\_exit\_\_.

Далее, общий синтаксис вызова менеджера, следующий:

with <менеджер контекста> as <переменная>:  
      список конструкций языка Python

Здесь «переменная» - это ссылка на экземпляр менеджера контекста, через которую, мы потом с ним можем работать. При необходимости ее можно опустить и записать все вот в таком виде:

with <менеджер контекста>:  
      список конструкций языка Python

Но тогда мы не сможем обратиться к объекту менеджера контекста.

Давайте создадим свой класс менеджера, который бы контролировал работу при изменении списка: если программа в теле менеджера приводит к исключению (ошибке), то список должен оставаться прежним (без изменений):

v1 = [1, 2, 3] v2 = [1, 2]**with** DefenerVector(v1) **as** dv:     **for** i **in** enumerate(dv):         dv[i] += v2[i]   **print**(v1)

А класс DefenderVector менеджера контекста будет выглядеть так:

**class** DefenerVector:     **def** \_\_init\_\_(self, v):         self.\_\_v = v       **def** \_\_enter\_\_(self):         self.\_\_temp = self.\_\_v[:]  *# делаем копию вектора v*         **return** self.\_\_temp       **def** \_\_exit\_\_(self, exc\_type, exc\_val, exc\_tb):         **if** exc\_type **is** None:             self.\_\_v[:] = self.\_\_temp         **return** False

Мы здесь в приватном свойстве сохраняем ссылку на вектор, который следует «защитить». Далее, в методе enter создаем копию этого вектора и возвращаем его. То есть, переменная dv будет ссылаться на эту копию и обработка внутри менеджера будет происходить с элементами этой копии, а не исходным вектором. Затем, в методе exit мы проверяем: если исключений не произошло, то заменяем все элементы вектора \_\_v на преобразованные \_\_temp. В результате, при выходе из менеджера, мы получим измененный вектор v1. Если же было какое-либо исключение, то запись новых данных выполняться не будет и у нас останется прежний вектор v1.

Метод exit у нас возвращает значение False, что означает обработку исключения (если оно произошло) вышестоящим блоком. Обычно именно так и делают, чтобы не скрывать возможные ошибки и в обработчике верхнего уровня реагировать должным образом на ошибочные ситуации. Кстати, оператор return можно вовсе опустить, тогда метод exit возвратит None, а оно интерпретируется как False. Так что, часто его не пишут.

Давайте для примера возвратим значение True и смотрите, при возникновении исключения, оно было перехвачено менеджером и далее уже не распространялось. Снова вернем False, запустим и теперь видим это исключение снова.

## **Вложенные менеджеры контекстов**

При необходимости, менеджеры контекстов можно вкладывать друг в друга. Например, при работе с файлами, можно выполнить такое вложение:

**try**:     **with** open("myfile.txt") **as** fin:         **with** open("out.txt", "w") **as** fout:             **for** line **in** fin:                 fout.write(line)**except** Exception **as** e:     **print**(e)

Работает все очевидным образом. Сначала завершается (отрабатывает) вложенный менеджер, а затем, внешний (первый). Во всем остальном логика работы такого вложения сохраняется.

Думаю, теперь вы лучше узнали, что такое менеджеры контекста и зачем они нужны, а также сможете создавать свои менеджеры, если в этом возникнет необходимость.

Курс по Python ООП: <https://stepik.org/a/116336>