# **Магический метод \_\_call\_\_. Функторы и классы-декораторы**

[Смотреть материал на видео](https://www.youtube.com/watch?v=UqX5Qekb9sU&list=PLA0M1Bcd0w8zPwP7t-FgwONhZOHt9rz9E)

На этом занятии мы познакомимся с очередным магическим методом \_\_call\_\_. Магические методы еще называют:

dunder-методы (от англ. сокращения double underscope)

В дальнейшем я буду говорить магические методы. Итак, когда вызывается метод \_\_call\_\_ и для чего он нужен? Как вы уже знаете, после объявления любого класса:

**class** Counter:     **def** \_\_init\_\_(self):         self.\_\_counter = 0

Мы можем создавать его экземпляры командой:

c = Counter()

Обратите внимание на круглые скобки после имени класса. В общем случае – это оператор вызова, например, так можно вызывать функции. Но, как видите, так можно вызывать и классы. В действительности, когда происходит вызов класса, то автоматически запускается магический метод \_\_call\_\_ и в данном случае он создает новый экземпляр этого класса:

Это очень упрощенная схема реализации метода \_\_call\_\_, в действительности, она несколько сложнее, но принцип тот же: сначала вызывается магический метод \_\_new\_\_ для создания самого объекта в памяти устройства, а затем, метод \_\_init\_\_ - для его инициализации. То есть, класс можно вызывать подобно функции благодаря встроенной для него реализации магического метода \_\_call\_\_. А вот экземпляры классов так вызывать уже нельзя. Если записать команду:

c()

то возникнет ошибка: «TypeError: 'Counter' object is not callable».

Как вы уже догадались, мы можем поправить этот момент, если явно в классе Counter пропишем магический метод \_\_call\_\_, например, так:

**class** Counter:     **def** \_\_init\_\_(self):         self.\_\_counter = 0       **def** \_\_call\_\_(self, \*args, \*\*kwargs):         **print**("\_\_call\_\_")         self.\_\_counter += 1         **return** self.\_\_counter

Здесь мы выводим сообщение, что был вызван данный метод, затем увеличиваем счетчик counter для текущего объекта на 1 и возвращаем его.

Запустим программу снова и теперь никаких ошибок нет, а в консоли отобразилась строка «\_\_call\_\_», что означает вызов магического метода \_\_call\_\_. То есть, благодаря добавлению этого магического метода в наш класс, теперь можно вызывать его экземпляры подобно функциям через оператор круглые скобки. Классы, экземпляры которых можно вызывать подобно функциям, получили название **функторы**.

В нашем случае метод \_\_call\_\_ возвращает значение счетчика, поэтому с объектом можно работать, следующим образом:

c = Counter() c() c() res = c()**print**(res)

Мы здесь три раза вызвали метод \_\_call\_\_ и счетчик \_\_counter трижды увеличился на единицу. Поэтому в консоли мы видим значение 3. Мало того, если создать еще один объект-счетчик:

c = Counter() c2 = Counter() c() c() res = c() res2 = c2()**print**(res, res2)

То они будут работать совершенно независимо и подсчитывать число собственных вызовов.

Давайте еще раз посмотрим на определение метода \_\_call\_\_. Здесь записаны параметры \*args, \*\*kwargs. Это значит, что при вызове объектов мы можем передавать им произвольное количество аргументов. Например, в нашем случае можно указать значение изменения счетчика при текущем вызове. Для этого я перепишу метод \_\_call\_\_, следующим образом:

**def** \_\_call\_\_(self, step=1, \*args, \*\*kwargs):         self.\_\_counter += step         **return** self.\_\_counter

Здесь появился в явном виде первый параметр step с начальным значением 1. То есть, можно вызывать объекты, например, так:

c(2) c(10) res = c() res2 = c2(-5)

Вот общий принцип работы магического метода \_\_call\_\_. Но здесь остается, как всегда, один важный вопрос: зачем это нужно, где может пригодиться? Давайте я приведу несколько примеров его использования.

Первый пример – это использование класса с методом \_\_call\_\_ вместо замыканий функций. Смотрите, мы можем объявить класс StripChars, который бы удалял вначале и в конце строки заданные символы:

**class** StripChars:     **def** \_\_init\_\_(self, chars):         self.\_\_chars = chars       **def** \_\_call\_\_(self, \*args, \*\*kwargs):         **if** **not** isinstance(args[0], str):             **raise** ValueError("Аргумент должен быть строкой")           **return** args[0].strip(self.\_\_chars)

Для этого, в инициализаторе мы сохраняем строку \_\_chars – удаляемые символы, а затем, при вызове метода \_\_call\_\_ удаляем символы через строковый метод strip для символов \_\_chars. То есть, теперь можно создать экземпляр класса и указать те символы, которые следует убирать:

s1 = StripChars("?:!.; ")

А, затем, вызвать объект s1 подобно функции:

res = s1(" Hello World! ")**print**(res)

В результате объект s1 будет отвечать за удаление указанных символов в начале и конце строки. Но нам ничто не мешает определять другие объекты этого класса с другим набором символов:

s1 = StripChars("?:!.; ") s2 = StripChars(" ") res = s1(" Hello World! ") res2 = s2(" Hello World! ")**print**(res, res2, sep='**\n**')

То есть, объект s2 уже отвечает только за удаление пробелов, тогда как s1 и некоторых других символов. Достаточно элегантное решение задачи, где нам требуется сохранять символы для удаления.

## **Классы-декораторы**

Второй пример – это реализация декораторов с помощью классов. Ранее мы с вами создавали декоратор для вычисления значения производной функции в определенной точке x. Я повторю эту реализацию, но с использованием класса. Вначале запишем следующий класс:

**class** Derivate:     **def** \_\_init\_\_(self, func):         self.\_\_fn = func       **def** \_\_call\_\_(self, x, dx=0.0001, \*args, \*\*kwargs):         **return** (self.\_\_fn(x + dx) - self.\_\_fn(x) ) / dx

Здесь в инициализаторе сохраняем ссылку на функцию, которую декорируем, а в методе \_\_call\_\_ принимаем один обязательный параметр x – точку, где вычисляется производная и dx – шаг изменения при вычислении производной.

Далее, определим функцию, например, просто синус:

**def** df\_sin(x):     **return** math.sin(x)

и вызове ее пока без декорирования:

**print**(df\_sin(math.pi/4))

После запуска программы увидим значение примерно 0.7071. Давайте теперь добавим декоратор. Это можно сделать двумя способами. Первый, прописать все в явном виде:

df\_sin = Derivate(df\_sin)

Теперь df\_sin – это экземпляр класса Derivate, а не исходная функция. Поэтому, когда она будет вызываться, то запустится метод \_\_call\_\_ и вычислится значение производной в точке math.pi/4.

Второй способ – это воспользоваться оператором @ перед объявлением функции:

@Derivate **def** df\_sin(x):     **return** math.sin(x)

Получим абсолютно тот же самый результат. Вот принцип создания декораторов функций на основе классов. Как видите, все достаточно просто – запоминаем ссылку на функцию, а затем, расширяем ее функционал в магическом методе \_\_call\_\_.

Надеюсь, из этого занятия вы поняли, как работает метод \_\_call\_\_ и где он может быть использован. Конечно, я привел всего два простых примера, чтобы продемонстрировать принцип его работы. В реальных задачах, проектах, вы должны сами, используя свои знания, решать, какие механизмы, приемы следует применять для решения текущих задач. Привести алгоритмы решений на все случаи жизни просто невозможно – это уже навык алгоритмизации, которым должен владеть каждый программист. И вырабатывается он, в основном, на решении практических задач. Так что больше практикуйтесь параллельно с изучением возможностей языка Python.