**Декораторы в Python за три минуты**

26.05.2021

*Декораторы* представляют собой удобный для восприятия человеком способ *расширения* возможностей функции, метода или класса извне. Использование декораторов особенно полезно при декорировании (т. е. расширении) похожих функций для выполнения ими одного и того же, но без добавления ненужных повторений в коде.

Вот простой пример того, как *расширить* функцию с помощью декоратора:

@guard\_zero

def divide(x, y):

return x / y

Декоратор @guard\_zero расширяет функциональность метода divide тем, что не допускает деления на 0. Однако пока что guard\_zero не существует, ведь он еще не реализован. Далее покажем, как именно это сделать.

**Как создать декоратор в Python**

Лучший способ продемонстрировать, каковы декораторы в действии,  —  создать и задействовать один из них. Реализуем наш декоратор @guard\_zero.

Вначале у нас есть только этот метод, который делит два числа:

def divide(x, y):

return x / y

Проблема с этим методом в том, что нет проверки, не равно ли 0 значение y. Очевидное решение здесь  —  задействовать простую проверку if. Но есть и альтернативное решение: *декораторы*.

Начнем с создания простой *функции-декоратора* guard\_zero:

def guard\_zero(operate):

def inner(x, y):

if y == 0:

print("Cannot divide by 0.")

return

return operate(x, y)

return inner

Декоратор в Python  —  это такая же обычная функция. Она принимает в качестве аргумента *функцию* operate. Затем расширяет функциональность operate, создавая *внутреннюю функцию и добавляя туда расширенное поведение*. После чего возвращает внутреннюю функцию inner, которая становится новой версией функции operate.

И вот декоратор guard\_zero готов. Теперь расширим (т. е. декорируем) функцию divide:

divide = guard\_zero(divide)

Выглядит прямо как добавление переменной нового значения. Но в этом случае существующей функции просто добавляется новая функциональность, что совершенно допустимо.

Есть и более распространенный синтаксис применения декоратора. Если в предыдущем синтаксисе функция заменялась новой своей версией, то здесь декоратор просто ставится перед определением функции:

@guard\_zero

def divide(x, y):

return x / y

Такой синтаксис более удобен для восприятия человеком, и суть его ясна: *метод divide расширяется, чтобы исключить деление на 0.*

Теперь пора протестировать метод divide различными входными данными и убедиться, что декоратор делает то, что должен:

print(divide(5, 0))

print(divide(5, 2))

Вывод:

Cannot divide by 0.

None

2.5

Обратите внимание на None в выводе. А все потому, что декоратор guard\_zero возвращает None, когда значение y равно 0.

Вот и все! Теперь вы знаете, что такое декораторы и как их использовать. А вот и весь код:

def guard\_zero(operate):

def inner(x, y):

if y == 0:

print("Cannot divide by 0.")

return

return operate(x, y)

return inner

@guard\_zero

def divide(x, y):

return x / y

print(divide(5, 0)) *# выводит Cannot divide by 0 («На 0 делить нельзя»)*

Напомним: метод декоратора guard\_zero принимает в качестве аргумента функцию divide и создает ее расширенную версию.

**Когда же используются декораторы?**

«К чему такие сложности?»  —  скажете вы. Достаточно написать простую проверку if внутри функции divide, сэкономив при этом несколько строк кода.

Мощь декораторов становится очевидной, *когда видишь, что они помогают избежать повторения кода*.

Представьте, что у вас десять подобных друг другу методов. Необходимо убедиться, что второй параметр каждого из них не равен 0. В этой ситуации вы:

* либо потратите время на написание десяти одинаковых проверок if для каждого метода отдельно,
* либо создадите декоратор и напишете @zero-guard перед каждой функцией.

Последний подход будет более предпочтительным, ведь в этом случае нужно просто написать декоратор в одном месте и добавить везде @zero-guard.

**Заключение**

В Python декораторы используются для расширения функциональных возможностей функции, метода или класса извне.

Например, чтобы не допустить деления на 0, задействуется декоратор guard\_zero. Просто добавляем его перед определением метода и таким образом используем этот декоратор в любом месте кода:

@guard\_zero

def divide(x, y):

return x / y

Но когда надо задействовать декораторы? Главным образом когда нужно избежать повторений при использовании похожих методов. Вместо того, чтобы вносить одно и то же изменение в несколько методов, куда как проще сделать одно изменение, а затем *декорировать* им все методы.

Спасибо за внимание! Надеюсь, статья была вам полезна.