GeekBrains

Объектно-ориентированные паттерны, группа структурные: Decorator

Введение

Декоратор – это **вызываемый объект**, основное назначение которого — служить оберткой для функции, метода класса и даже самого класса с целью изменить или расширить их возможности. Синтаксически декораторы оформляются добавлением специального символа **@** к имени декоратора.

Декоратор-функция

Рассмотрим пример.

Листинг 1. Урок 13. Коды к уроку/deco_func_1.py

```
"""Простейший декоратор-функция"""

from time import time


def decorator(func):
    """Декоратор"""
    def wrapper():
        """Дополненная функция"""
        start = time()
        func()
        end = time()
        print(f'Время выполнения исходной ф-ции:
                                        {round(end-start, 5)} секунд')
    return wrapper


@decorator
def get_wp():
    print('Выполняем расчет')


get_wp()
```

Декоратором должен быть **вызываемый объект**. Классическим примером вызываемого объекта является функция.

Представьте, что мы хотим преподнести кому-то подарок (функция **get_wp()**). При этом хотим его украсить яркой оберткой. Для этого мы обращаемся к специалисту (функция **decorator()**), чтобы он это сделал:

```
@decorator
def get_wp():
    print('Выполняем расчет')
```

Специалист принимает наш подарок, т.е. объект декорируемой функции:

```
def decorator(func):

...
```

Специалист выполняет декорирование и обертывает наш подарок (функцию) красивой оберткой:

```
…
def wrapper():
    """Дополненная функция"""
    start = time()
    func()
    end = time()
    print(f'Время выполнения исходной ф-ции: {round(end-start, 5)} секунд')
…
```

И возвращает нам не исходный подарок, а уже украшенный:

```
…

return wrapper
```

Т.е. возвращает объект не исходной функции, а улучшенной, декорированной.

Получается, что теперь мы преподнесем не исходный подарок, а украшенный.

Если говорить о функции **get_wp()**, то при ее вызове будет исполняться не ее исходная версия, а улучшенная.

Стоит отметить, что внутри дополненной функции **wrapper()** мы видим вызов исходной функции **func()**.

```
def decorator(func):
    """Декоратор"""
    def wrapper():
        """Дополненная функция"""
        start = time()
        func()
        end = time()
        print(f'Время выполнения исходной ф-ции: {round(end-start, 5)} секунд')
    return wrapper
```

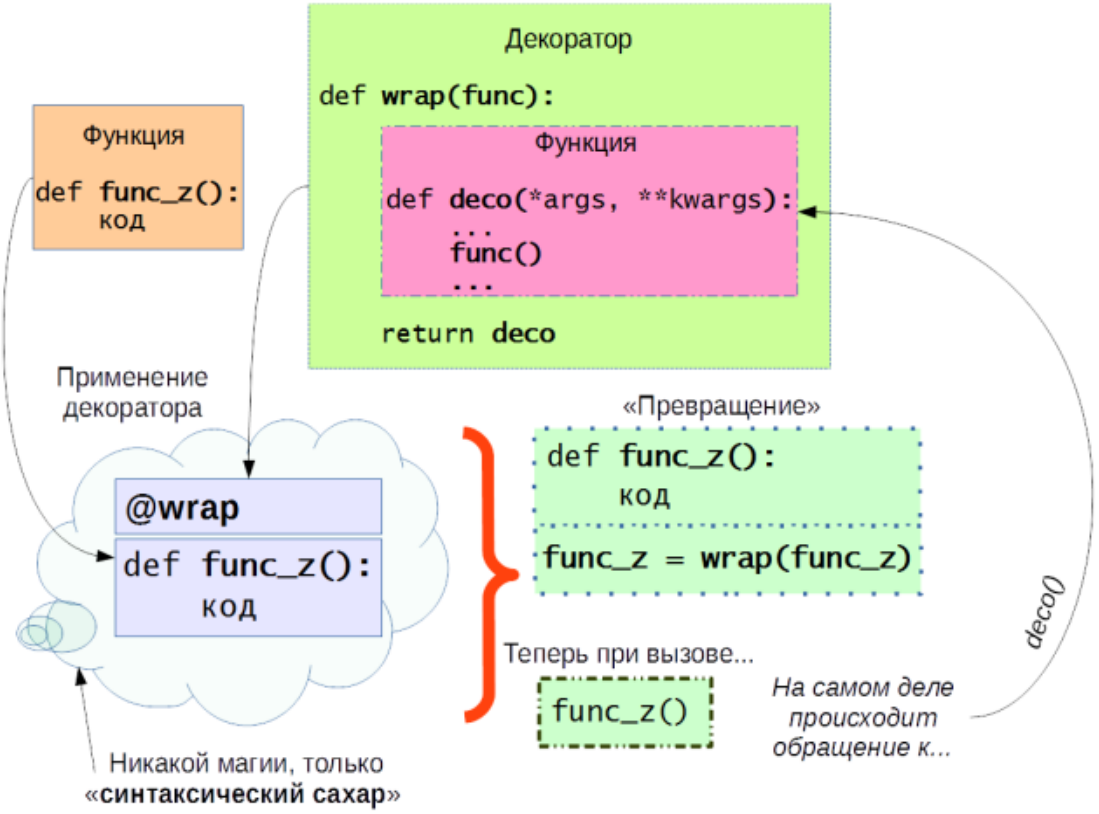
Вызов исходной функции обязательно нужен, ведь когда мы вызываем декорированную функцию нужно сохранить логику исходной версии функции.

В примере с подарком, если мы забудем добавить вызов **func()** во **wrapper()**, это будет как передать специалисту подарок, а он вернул нам только обертку.

В нашем примере декоратор выполняет роль средства для замеров времени работы любой функции. Плюс в том, что этот декоратор мы можем применить к любым функциям, это универсальный декоратор. Иначе нам пришлось бы прописывать выражения для замеров для каждой функции.

Пользовательские декораторы могут использоваться для замеров времени выполнения функций, затрачиваемой памяти, логирования и решения других задач.

Далее приведена диаграмма, описывающая логику вызовов при использовании декоратора-функции.



Теперь представим, что наша декорируемая функция принимает позиционные или именованные аргументы.

Листинг 2. Урок 13. Коды к уроку/deco_func_2.py

```
@decorator
def get_wp(url):
    """Делаем запрос"""
    res = requests.get(url)
    return res


print(get_wp('https://google.com'))
```

Как передать их декоратору? Попробуем напрямую:

```
def decorator(func, *args):
    """Сам декоратор"""
    def wrapper():
        """Обертка"""
        start = time.time()
        return_value = func(args[0])
        end = time.time()
        print(f'Отправлен запрос на адрес {args[0]}. '
              f'Время выполнения: {round(end-start, 2)} секунд')
        # обертка может возвращать и некий результат
        return f'возврат - {return_value}'
    return wrapper
```

Это даст ошибку:

```
Traceback (most recent call last):
  File "D:/Курсы GeekBrains. 2022/буткемп/урок 13/Урок 13. Коды примеров/deco_func_2.py", line 28, in <module>
    print(get_wp('https://google.com'))
TypeError: wrapper() takes 0 positional arguments but 1 was given
```

Эта ошибка неслучайна. Ведь декоратор должен принимать на вход только объект исходной функции, т.е.:

```
def decorator(func):
```

По сути в представленном выше скриншоте ответ на вопрос, как исправить ошибку.

Нам не нужно специально передавать декоратору выходные аргументы декорируемой функции.

Он уже о них «знает», главное прописать эти аргументы в обертке, т.е. в нашем случае во **wrapper()** и конечно не забыть передать аргументы в саму исходную функцию, которую мы декорируем:

```
def decorator(func,):
    """Сам декоратор"""
    def wrapper(*args, **kwargs):
        """Обертка"""
        start = time.time()
        return_value = func(args[0])
        end = time.time()
        print(f'Отправлен запрос на адрес {args[0]}. '
              f'Время выполнения: {round(end-start, 2)} секунд')
        # обертка может возвращать и некий результат
        return f'возврат - {return_value}'
    return wrapper
```

Теперь с передачей аргументов все в порядке:

```
def decorator(func):
    """Сам декоратор"""
    def wrapper(*args, **kwargs):
        """обертка"""
        start = time.time()
        return_value = func(args[0])
        end = time.time()
        print(f'Отправлен запрос на адрес {args[0]}. '
              f'Время выполнения: {round(end-start, 2)} секунд')
        # обертка может возвращать и некий результат
        return f'возврат - {return_value}'
    return wrapper
```

Если рассмотреть аналогию с подарком, то аргументы декорируемой функции — это, например, яркая открытка, которую специалист должен прикрепить к подарку. И он действительно ее прикрепляет:

```
def decorator(func):
    """Сам декоратор"""
    def wrapper(*args, **kwargs):
        """Обертка"""
        start = time.time()
        return_value = func(args[0])
        end = time.time()
        print(f'Отправлен запрос на адрес {args[0]}. '
              f'Время выполнения: {round(end-start, 2)} секунд')
        # обертка может возвращать и некий результат
        return f'возврат - {return_value}'
    return wrapper
```

Стоит отметить еще один важный момент. Использование служебного символа @ это так называемый «синтаксический сахар», т.е. задекорировать функцию можно и без этого символа. А ведь действительно, декоратор, это всего лишь функция-обертка для объекта другой функции. Поэтому декорирование можно выполнить и без синтаксического сахара.

Реализуем первый пример без синтаксического сахара:

Листинг 3. Урок 13. Коды к уроку/deco_func_3.py

```
from time import time


def decorator(func):
    """Декоратор"""
    def wrapper():
        """Дополненная функция"""
        start = time()
        func()
        end = time()
        print(f'Время выполнения исходной ф-ции: {round(end-start, 5)} секунд')
    return wrapper


def get_wp():
    print('Выполняем расчет')


get_wp = decorator(get_wp)
get_wp()
```

Результат выполнения будет аналогичен результату выполнения варианта с синтаксическим сахаром.

Нам интересны последние две строки:

```
get_wp = decorator(get_wp)
get_wp()
```

С переменной `get_wp` мы связываем объект функции **decorator**, которой передали объект исходной функции `get_wp`.

Получается, что переменная `get_wp` будет ссылаться на объект задекорированной функции `get_wp`.

А т.к. это объект функции, его можно вызвать, что мы и делаем.

Вы можете удивиться, почему у нас и переменная слева от знака равно и сама исходная функция имеют одинаковые имена. Но здесь все просто. Ведь в Python динамическая типизация и такой трюк допустим.

Переменная это ссылка и мы можем в любой момент связать эту ссылку с любым объектом.

Можно было записать последние две строки и таким образом:

```
res = decorator(get_wp)
res()
```

Результат будет аналогичным.

Декоратор-объект класса

Вспомним еще раз утверждение, что декоратором должен быть вызываемый объект. В Python им может быть не только функция, но и объект класса.

Выполним простой эксперимент:

Листинг 4. Урок 13. Коды к уроку/deco_class_1.py

```
class NewClass:
    pass


obj = NewClass()
obj()
```

Мы создали объект класса и пытаемся его вызвать:

```
Traceback (most recent call last):
  File "D:/Курсы GeekBrains. 2022/буткемп/урок 13/Урок 13. Коды примеров/deco_class_1.py", line 6, in <module>
    obj()
TypeError: 'NewClass' object is not callable
```

Но нам говорят, что этого делать нельзя и объект класса не является вызываемым объектом. Неужели мы ошиблись и объект класса действительно нельзя вызвать, как обычную функцию?

На самом деле можно, но для этого нужно в рамках класса выполнить перегрузку встроенного метода `__call__`:

Листинг 5. Урок 13. Коды к уроку/deco_class_1.py

```
class NewClass:
    def __call__(self, *args, **kwargs):
        pass
```

```
obj = NewClass()
obj()
```

Все получилось, все работает.

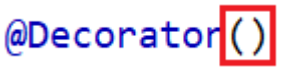
Перепишем наш самый первый пример с декоратором-функцией на вариант, когда декоратором будет объект класса.

Листинг 6. Урок 13. Коды к уроку/deco_class_2.py

```
from time import time


class Decorator:
    """Класс-декоратор"""

    def __call__(self, func):
        def wrapper(*args, **kwargs):
            start = time()
            func()
            end = time()
            print(
                f'Время выполнения исходной ф-ции: {round(end - start, 5)} секунд')

        return wrapper


@Decorator()
def get_wp():
    print('Выполняем расчет')


get_wp()
```

Код изменился незначительно. Теперь объект функции мы передаем методу **__call__()**.

Вы можете задаться вопросом, откуда появились эти скобки:



Ведь когда декоратором была функция, скобок не было. Дело в том, что функция сама по себе является вызываемым объектом и скобки не требуется.

А когда мы говорим об объекте класса, то перед тем, как вызывать этот объект, его нужно создать. Для этого скобки и нужны.

Декораторы с параметром

Нам может потребоваться передать в декоратор некоторый параметр, и это именно параметр для декоратора, а не для функции, к ней он не имеет отношения.

Например, мы хотим подсчитать среднее время выполнения 10 запусков функции, а в другом случае 20 запусков, а в третьем – 100. Число запусков не имеет отношения к самой функции, но имеет отношение к декоратору, т.к. он проводит замеры. Так как же передать в декоратор это постоянно меняющееся число.

Нам нужно реализовать декоратор с параметром. Причем реализовать его и в виде функции, и в виде объекта класса.

Для реализации декоратора-функции с параметром нам нужно создать внешнюю функцию, которая будет принимать параметр, а управление передавать уже декоратору, который в свою очередь будет принимать объект нашей декорируемой функции. Получается своего рода «матрешка».

Листинг 7. Урок 13. Коды к уроку/deco_func_4.py

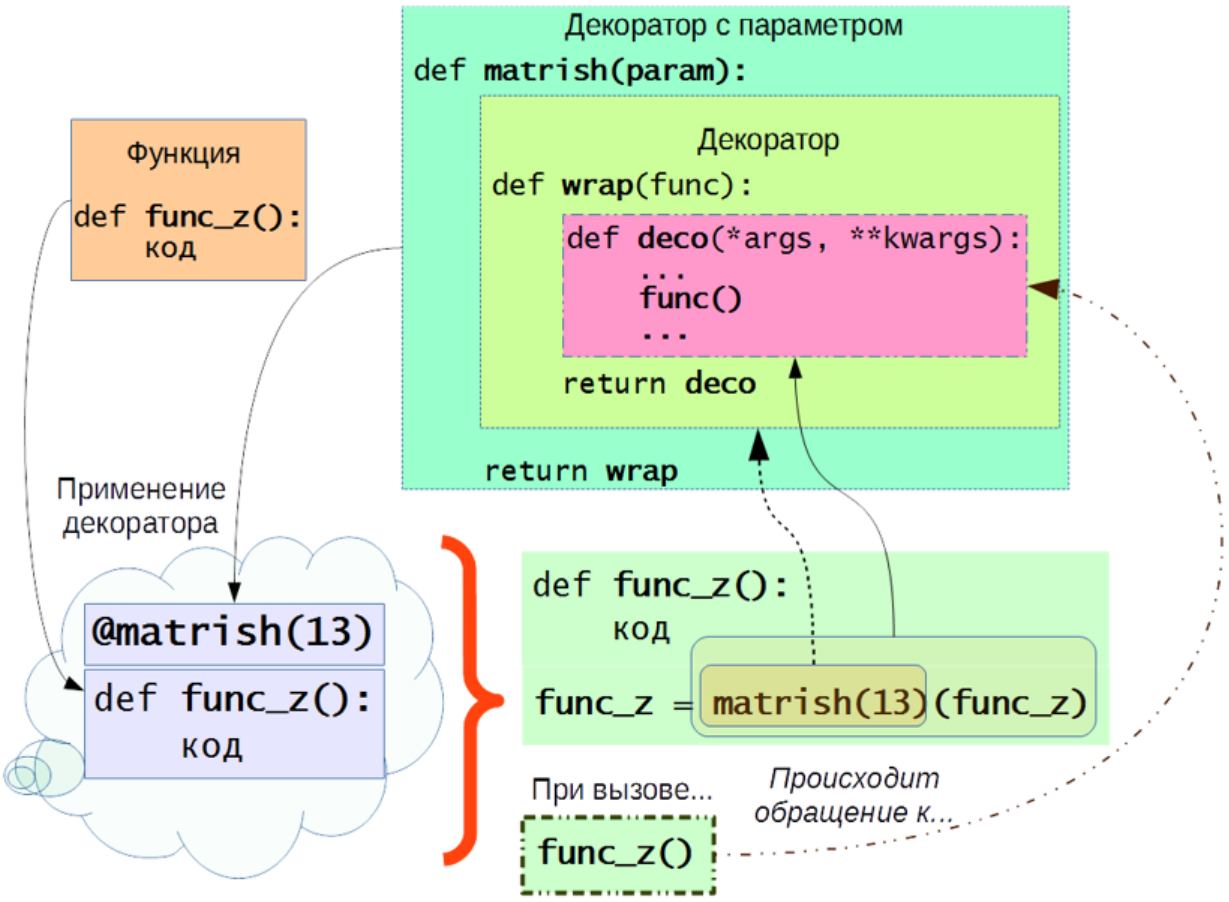
```
"""Простейший декоратор-функция"""

from time import time


def out(num):
    def decorator(func):
        """Декоратор"""

        def wrapper():
            """Дополненная функция"""
            res = 0
            for el in range(num):
                start = time()
                func()
                end = time()
                delta = round(end - start, 5)
                res += delta

            print(
                f'Среднее выполнения исходной ф-ции: {round(res / num, 5)} секунд')

        return wrapper

    return decorator


@out(10)
def get_wp():
    pass


get_wp()
```

В приведенном примере формально декоратором является функция **out()**, но по факту, она всего лишь принимает параметр, и управление передает на реальный декоратор – **decorator()**, а он в свою очередь передает управление на функцию-обертку **wrapper()**, т.е. возвращает объект задекорированной функции, которая далее и вызывается.

Без синтаксического сахара этот механизм выглядел бы так:

```
def get_wp():
    pass

get_wp = out(10)(get_wp)
get_wp()
```

Схема работы декоратора-функции с параметром:



Декоратор-объект класса с параметром выглядел бы следующим образом:

Листинг 8. Урок 13. Коды к уроку/deco_class_3.py

```
from time import time


class Decorator:
    """Класс-декоратор"""
    def __init__(self, num):
        self.num = num

    def __call__(self, func):
        def wrapper(*args, **kwargs):
            res = 0
            for el in range(self.num):
                start = time()
                func()
                end = time()
                delta = round(end - start, 5)
```

```
                res += delta

            print(
                f'Среднее выполнения исходной ф-ции:
                                {round(res / self.num, 5)} секунд')

        return wrapper


@Decorator(10)
def get_wp():
    pass


get_wp()
```

В данном случае мы не создаем отдельную пользовательскую функцию для приема аргумента. Достаточно выполнить перегрузку стандартного встроенного метода инициализации объекта класса (**__init__()**) и передать параметр в этот метод.

И, как всегда, реализуем и вариант без синтаксического сахара:

```
def get_wp():
    pass


get_wp = Decorator(10)(get_wp)
get_wp()
```