УДК 004.423
DOI 10.51691/2541-8327_2023_6_4

ПРИМЕНЕНИЕ ДЕКОРАТОРОВ ДЛЯ УЛУЧШЕНИЯ ПРОИЗВОДИТЕЛЬНОСТИ И УДОБСТВА РАБОТЫ В PYTHON

Халевин Т.А.
студент направления подготовки информатика и вычислительная техника,
Хакасский государственный университет имени Н.Ф. Катанова,
г. Абакан, Россия [1]

Аннотация: Данная статья рассматривает концепцию декораторов в Python и их многочисленные применения. Она раскрывает, как декораторы могут повысить эффективность и удобство работы с функциями, позволяя добавлять дополнительное поведение без изменения их исходного кода. В результате, программисты получают мощный инструмент для модификации и расширения функциональности программ.
Ключевые слова: Python, декораторы, функции, декорирование функций, функции высшего порядка

USING DECORATORS TO IMPROVE PERFORMANCE AND USABILITY IN PYTHON

Khalevin T.A.
student of computer science and computer engineering department,
N.F. Katanov Khakass State University,
Abakan, Russia

Abstract: This article discusses the concept of decorators in Python and their many applications. It reveals how decorators can improve the efficiency and usability of functions by allowing additional behavior to be added without changing their source

[1] Научный руководитель: Голубничий А.А. старший преподаватель кафедры ПОВТиАС, Хакасский государственный университет имени Н.Ф. Катанова, г. Абакан, Россия

code. As a result, programmers have a powerful tool to modify and extend the functionality of programs.

Keywords: Python, decorators, functions, decorating functions, higher-order functions

В Python декораторы являются специальным типом функций, которые позволяют изменять поведение других функций без необходимости изменять их код непосредственно. Вместо этого мы можем определить декоратор, который принимает функцию в качестве аргумента, оборачивает ее в новую функцию и возвращает эту функцию. Таким образом, декоратор добавляет новое поведение к исходной функции, не затрагивая ее код [1].

Рассмотрим простой пример декоратора (Рис. 1).

```
def my_decorator(func):
    def wrapper():
        print("Декоратор до выполнения функции")
        func()
        print("Декоратор после выполнения функции")
    return wrapper
```

Рисунок 1 – Пример декоратора [разработано автором]

В данном примере мы определяем декоратор my*_decorator*, который принимает функцию *func* в качестве аргумента. Декоратор определяет внутреннюю функцию *wrapper*, которая выводит сообщение перед вызовом функции *func*, вызывает ее и выводит сообщение после ее вызова.

Чтобы применить декоратор к функции, мы используем следующий синтаксис (Рис. 2).

```
@my_decorator
def my_function():
    print("Выполнение функции обернутой в декоратор")


my_function()
Декоратор до выполнения функции
Выполнение функции обернутой в декоратор
Декоратор после выполнения функции
```

Рисунок 2 – Применение декоратора к функции [разработано автором]

Еще один пример применения декоратора к функции (Рис. 3).

```
def my_function():
    print("Выполнение функции обернутой в декоратор")

my_function = my_decorator(my_function)
my_function()
Декоратор до выполнения функции
Выполнение функции обернутой в декоратор
Декоратор после выполнения функции
```

Рисунок 3 – Применение декоратора к функции другим способом

[разработано автором]

Таким образом, мы передаем функцию *my_function* в качестве аргумента декоратору *my_decorator*, который оборачивает ее в новую функцию и возвращает ее. Затем мы присваиваем результат вызова декоратора переменной *my_function*, теперь при вызове *my_function* будет выполняться не исходная функция, а обернутая в декоратор.

Декораторы являются важным инструментом в Python, и они часто используются для добавления новой функциональности к существующему коду без необходимости его изменения [2]. Декораторы используются во многих фреймворках и библиотеках Python, таких как Flask, Django, SQLAlchemy и многих других [3].

Декораторы также позволяют повторно использовать код, что уменьшает объем его дублирования [4]. Например, мы можем определить декоратор для логирования вызовов функций и применять его к различным функциям вместо того, чтобы вручную добавлять логирование в каждую функцию [5]. Декораторы также могут улучшить производительность кода, путем кэширования результатов выполнения функций (Рис. 4).

```
def memoize(func):
    cache = {}
    def wrapper(*args):
        if args not in cache:
            cache[args] = func(*args)
        return cache[args]
    return wrapper

@memoize
def fib(n):
    if n in (0, 1):
        return n
    return fib(n-1) + fib(n-2)

fib(30)
832040
fib(35)
9227465
fib(40)
102334155
```

Рисунок 4 – Применение декоратора кэширования [разработано автором]

В данном примере мы определяем функцию *fib*, которая вычисляет числа Фибоначчи. Чтобы улучшить производительность функции, мы применяем декоратор *memoize*, который кэширует результаты выполнения функции. Таким образом, если функция вызывается с теми же аргументами, что и раньше, результат будет возвращен из кэша, без необходимости повторного вычисления.

Так же можно считать время выполнения функции с помощью декоратора (Рис. 5).

```
import time

def total_time(func):
    global start_time
    start_time = None
    def wrapper(*args):
        global start_time
        if start_time is None:
            start_time = time.time()
        result = func(*args)
        return result
    def total():
        end_time = time.time()
        print(f"Общее время выполнения функции {func.__name__}: {end_time - start_time:.5f} секунд")
    wrapper.total = total
    return wrapper
```

Рисунок 5 – Декоратор для вычисления времени выполнения рекурсивной последовательности [разработано автором]

В этом декораторе мы сначала объявляем глобальную переменную *start_time*, которая будет хранить время начала выполнения рекурсивной последовательности вызовов функции. Затем мы определяем обертку *wrapper*, которая проверяет, установлено ли значение *start_time*, и, если нет, устанавливает его на текущее время. Затем мы вызываем функцию с переданными аргументами и возвращаем ее результат.

Кроме того, мы определяем функцию *total*, которая вычисляет общее время выполнения всей рекурсивной последовательности, вычитая время начала выполнения из текущего времени. Эта функция вызывается после завершения всех вызовов функции и выводит общее время выполнения в консоль.

Наконец, мы присваиваем функцию *total* атрибуту *total* обертки *wrapper*. Это позволит нам вызывать функцию *total* для получения общего времени выполнения функции.

Теперь применяем наш декоратор к вычислению чисел Фибоначчи (Рис. 6).

```
@total_time
def fib(n):
    if n in (0, 1):
        return n
    return fib(n-1) + fib(n-2)

print(fib(35))
9227465
fib.total()
Общее время выполнения функции fib: 8.71296 секунд
```

Рисунок 6 – Применение декоратора для вычисления времени выполнения всей рекурсивной последовательности [разработано автором]

Заключение

Декораторы – это мощный инструмент для улучшения производительности и удобства работы с функциями в Python. Они позволяют добавлять функциональность к существующим функциям, не изменяя при этом их исходный код. Мы рассмотрели несколько примеров декораторов и обсудили

их применение. Но следует помнить, что декораторы не всегда являются лучшим решением, и в некоторых случаях использование классов или других конструкций может быть более эффективным.

Библиографический список:

1. Бэрри, П. Изучаем программирование на Python [Текст] / П. Бэрри. – М.: Вильямс, 2014. – 243 с.
2. Кондратьева, В.А. Основы программирования на языке Python [Текст] / В.А. Кондратьева. – М.: МГП, 2022. – 68 с.
3. Гэддис, Т. Начинаем программировать на Python [Текст] / Т. Гэддис. – СПб.: БХВ-Петербург, 2021. – 768 с.
4. Макаренко Л.Ф. Программирование на языке Python [Текст] / Л.Ф. Макаренко, И.С. Шувалова. – М.: Московский автомобильно-дорожный государственный технический университет (МАДИ), 2022. – 88 с.
5. Гуриков, С.Р. Основы алгоритмизации и программирования на Python [Текст] / С.Р. Гуриков. – М.: ИНФРА-М, 2023. – 343 с.

Оригинальность 78%