

# Python Decorators

<https://www.geeksforgeeks.org/decorators-in-python/>

**Декораторы** — очень мощный и полезный инструмент в Python, поскольку он позволяет программистам изменять поведение функции или класса. Декораторы позволяют нам обернуть другую функцию, чтобы расширить поведение обернутой функции, не изменяя ее навсегда. Но прежде чем углубиться в декораторы, давайте разберемся с некоторыми понятиями, которые пригодятся при изучении декораторов.

Объекты первого класса В Python функции являются [объектами первого класса](#), что означает, что функции в Python могут использоваться или передаваться в качестве аргументов. Свойства функций первого класса:

- Функция — это экземпляр типа Object.
- Вы можете сохранить функцию в переменной.
- Вы можете передать функцию в качестве параметра другой функции.
- Вы можете вернуть функцию из функции.
- Вы можете хранить их в структурах данных, таких как хэш-таблицы, списки и т. д.

Рассмотрим приведенные ниже примеры для лучшего понимания.

**Пример 1:** Обработка функций как объектов.

```
# Python program to illustrate functions
# can be treated as objects
def shout(text):
    return text.upper()
print(shout('Hello'))
yell = shout
print(yell('Hello'))
```

Output:  
ПРИВЕТ  
ПРИВЕТ

В приведенном выше примере мы присвоили функцию Shout переменной. Это не вызовет функцию, вместо этого он берет объект функции, на который ссылается shout, и создает второе имя, указывающее на него, yell.

**Пример 2:** Передача функции в качестве аргумента

```
# Python program to illustrate functions
# can be passed as arguments to other functions
def shout(text):
    return text.upper()
def whisper(text):
    return text.lower()
def greet(func):
```

```
# storing the function in a variable
greeting = func("""Hi, I am created by a function passed as an argument.""")
print (greeting)
greet(shout)
greet(whisper)
```

Output:

ПРИВЕТ, Я СОЗДАН ФУНКЦИЕЙ, ПЕРЕДАННОЙ В КАЧЕСТВЕ АРГУМЕНТА.

привет, меня создала функция, переданная в качестве аргумента.

В приведенном выше примере функция приветствия принимает в качестве параметра другую функцию (в данном случае shout и whisper). Затем функция, переданная в качестве аргумента, вызывается внутри функции приветствия.

**Пример 3:** Возврат функций из другой функции.

```
# Python program to illustrate functions
# Functions can return another function
def create_adder(x):
    def adder(y):
        return x+y
    return adder
add_15 = create_adder(15)
print(add_15(10))
```

Output:

25

В приведенном выше примере мы создали функцию внутри другой функции, а затем вернули функцию, созданную внутри.

Приведенные выше три примера иллюстрируют важные концепции, необходимые для понимания декораторов. Пройдя через них, давайте теперь углубимся в декораторы.

## Декораторы

Как указано выше, декораторы используются для изменения поведения функции или класса. В декораторах функции передаются в качестве аргумента другой функции, а затем вызываются внутри функции-оболочки.


**Syntax for Decorator:**

```
@gfg_decorator
def hello_decorator():
    print("Gfg")
'''Above code is equivalent to -
def hello_decorator():
    print("Gfg")

hello_decorator = gfg_decorator(hello_decorator)'''
```

оболочку.

**Декоратор может изменить поведение :**



```
# defining a decorator
def hello_decorator(func):

    # inner1 is a Wrapper function in
    # which the argument is called

    # inner function can access the outer local
    # functions like in this case "func"
    def inner1():
        print("Hello, this is before function execution")

        # calling the actual function now
        # inside the wrapper function.
        func()

        print("This is after function execution")

    return inner1


# defining a function, to be called inside wrapper
def function_to_be_used():
    print("This is inside the function !!")


# passing 'function_to_be_used' inside the
# decorator to control its behaviour
function_to_be_used = hello_decorator(function_to_be_used)


# calling the function
function_to_be_used()
```

Output:

```
Hello, this is before function execution
This is inside the function !!
This is after function execution
```

Let's see the behaviour of the above code and how it runs step by step when the "function\_to\_be\_used" is called.

step 2 `def hello_decorator(func):`

step 3 `def inner1():`  
`print("Hello, this is before function execution")`

`func()`

step 4 `print("This is after function execution")`  
`return inner1`

`def function_to_be_used():`  
`print("This is inside the function!!")`

step 1 `function_to_be_used = hello_decorator(function_to_be_used)`

step 5 `function_to_be_used()`

`def hello_decorator(func):`

step 6 `def inner1():`  
step 7 `print("Hello, this is before function execution")`

step 8 `func()`

step 11 `print("This is after function execution")`  
`return inner1`

step 9 `def function_to_be_used():`  
step 10 `print("This is inside the function!!")`

`function_to_be_used = hello_decorator(function_to_be_used)`

step 12 `function_to_be_used()`