 Utter_step 5 апреля 2012 в 21:49

Понимаем декораторы в Python'е, шаг за шагом. Шаг 2

Разработка веб-сайтов*, Python*

Перевод Tutorial

Автор оригинала: Renaud Gaudin

```
14
15     @staticmethod
16     def field_from_file(filename):
17         in_file = open(filename).readlines(SudokuSolver.__height)
18         array = [map(int, x.split()) for x in in_file]
19         for line in array:
20             if len(line) != SudokuSolver.__width:
21                 raise RuntimeError("Wrong file format")
22         return SudokuSolver(array)
23
```

*И снова доброго времени суток всем читателям!
Спасибо, за проявленный интерес к первой части перевода, надеюсь, вторая вас так же не разочарует.*

Итак, в первой части данной статьи мы совершили базовое знакомство с декораторами, принципами их работы и даже написали свой вручную.
Однако, все декораторы, которые мы до этого рассматривали не имели одного очень важного функционала — передачи аргументов декорируемой функции.
Что ж, исправим это недоразумение!

Передача («проброс») аргументов в декорируемую функцию

Никакой чёрной магии, всё, что нам необходимо — собственно, передать аргументы дальше!

```
def a_decorator_passing_arguments(function_to_decorate):
    def a_wrapper_accepting_arguments(arg1, arg2): # аргументы прибывают отсюда
        print "Смотри, что я получил:", arg1, arg2
        function_to_decorate(arg1, arg2)
    return a_wrapper_accepting_arguments

# Теперь, когда мы вызываем функцию, которую возвращает декоратор,
# мы вызываем её "обёртку", передаём ей аргументы и уже в свою очередь
# она передаёт их декорируемой функции

@a_decorator_passing_arguments
def print_full_name(first_name, last_name):
    print "Меня зовут", first_name, last_name

print_full_name("Питер", "Венкман")
# выведет:
# Смотри, что я получил: Питер Венкман
# Меня зовут Питер Венкман
# *
```

* — Прим. переводчика: Питер Венкман — имя одного из Охотников за приведениями, главного героя одноименного культового фильма.

Декорирование методов

первым параметром ссылку на сам объект (*self*). Это значит, что мы можем создавать декораторы для методов так же, как и для функций, просто не забывая про *self*.

```
def method_friendly_decorator(method_to_decorate):
    def wrapper(self, lie):
        lie = lie - 3 # действительно, дружелюбно - снизим возраст ещё сильнее :-)
        return method_to_decorate(self, lie)
    return wrapper

class Lucy(object):

    def __init__(self):
        self.age = 32

    @method_friendly_decorator
    def sayYourAge(self, lie):
        print "Мне %s, а ты бы сколько дал?" % (self.age + lie)

l = Lucy()
l.sayYourAge(-3)
# выведет: Мне 26, а ты бы сколько дал?
```

Конечно, если мы создаём максимально общий декоратор и хотим, чтобы его можно было применить к любой функции или методу, то стоит воспользоваться тем, что **args* распаковывает список *args*, а ***kwargs* распаковывает словарь *kwargs*:

```
def a_decorator_passing_arbitrary_arguments(function_to_decorate):
    # Данная "обёртка" принимает любые аргументы
    def a_wrapper_accepting_arbitrary_arguments(*args, **kwargs):
        print "Передали ли мне что-нибудь?:"
        print args
        print kwargs
        # Теперь мы распакуем *args и **kwargs
        # Если вы не слишком хорошо знакомы с распаковкой, можете прочесть следующую
        # http://www.saltycrane.com/blog/2008/01/how-to-use-args-and-kwargs-in-python/
        function_to_decorate(*args, **kwargs)
    return a_wrapper_accepting_arbitrary_arguments

@a_decorator_passing_arbitrary_arguments
def function_with_no_argument():
    print "Python is cool, no argument here." # оставлено без перевода, хорошая игра слов

function_with_no_argument()
# выведет:
# Передали ли мне что-нибудь?:
# ()
# {}
# Python is cool, no argument here.

@a_decorator_passing_arbitrary_arguments
def function_with_arguments(a, b, c):
    print a, b, c

function_with_arguments(1,2,3)
# выведет:
# Передали ли мне что-нибудь?:
# (1, 2, 3)
# {}
# 1 2 3

@a_decorator_passing_arbitrary_arguments
def function_with_named_arguments(a, b, c, platypus="Почему нет?"):
    print "Любят ли %s, %s и %s утконосов? %s" %\
        (a, b, c, platypus)
```

```

function_with_named_arguments("Билл", "Линус", "Стив", platypus="Определенно!")
# выведет:
# Передали ли мне что-нибудь?:
# ('Билл', 'Линус', 'Стив')
# {'platypus': 'Определенно!'}
# Любят ли Билл, Линус и Стив утконосов? Определенно!

class Mary(object):

    def __init__(self):
        self.age = 31

    @a_decorator_passing_arbitrary_arguments
    def sayYourAge(self, lie=-3): # Теперь мы можем указать значение по умолчанию
        print "Мне %s, а ты бы сколько дал?" % (self.age + lie)

m = Mary()
m.sayYourAge()
# выведет:
# Передали ли мне что-нибудь?:
# (<__main__ .Mary object at 0xb7d303ac>,)
# {}
# Мне 28, а ты бы сколько дал?

```

Вызов декоратора с различными аргументами

Отлично, с этим разобрались. Что вы теперь скажете о том, чтобы попробовать вызывать декораторы с различными аргументами?

Это не так просто, как кажется, поскольку декоратор должен принимать функцию в качестве аргумента, и мы не можем просто так передать ему что либо ещё.

Так что, перед тем, как показать вам решение, я бы хотел освежить в памяти то, что мы уже знаем:

```

# Декораторы - это просто функции
def my_decorator(func):
    print "Я обычная функция"
    def wrapper():
        print "Я - функция, возвращаемая декоратором"
        func()
    return wrapper

# Так что, мы можем вызывать её, не используя "@"-синтаксис:

def lazy_function():
    print "zzzzzzzz"

decorated_function = my_decorator(lazy_function)
# выведет: Я обычная функция

# Данный код выводит "Я обычная функция", потому что это ровно то, что мы сделали:
# вызвали функцию. Ничего сверхъестественного

@my_decorator
def lazy_function():
    print "zzzzzzzz"

# выведет: Я обычная функция

```

Как мы видим, это два аналогичных действия. Когда мы пишем

```
@my_decorator
```

— мы просто говорим интерпретатору «вызвать функцию, под названием *my_decorator*». Это важный момент, потому что данное название может как привести нас напрямую к декоратору... так и нет!

Давайте сделаем нечто страшное!:)

```
def decorator_maker():

    print "Я создаю декораторы! Я буду вызван только раз: "+\
        "когда ты попросишь меня создать тебе декоратор."

    def my_decorator(func):

        print "Я - декоратор! Я буду вызван только раз: в момент декорирования функц

        def wrapped():
            print ("Я - обёртка вокруг декорируемой функции. "
                "Я буду вызвана каждый раз когда ты вызываешь декорируемую функц
                "Я возвращаю результат работы декорируемой функции.")
            return func()

        print "Я возвращаю обёрнутую функцию."

        return wrapped

    print "Я возвращаю декоратор."
    return my_decorator

# Давайте теперь создадим декоратор. Это всего лишь ещё один вызов функции
new_decorator = decorator_maker()
# выведет:
# Я создаю декораторы! Я буду вызван только раз: когда ты попросишь меня создать те
# Я возвращаю декоратор.

# Теперь декорируем функцию

def decorated_function():
    print "Я - декорируемая функция."

decorated_function = new_decorator(decorated_function)
# выведет:
# Я - декоратор! Я буду вызван только раз: в момент декорирования функции.
# Я возвращаю обёрнутую функцию.

# Теперь наконец вызовем функцию:
decorated_function()
# выведет:
# Я - обёртка вокруг декорируемой функции. Я буду вызвана каждый раз когда ты вызыва
# Я возвращаю результат работы декорируемой функции.
# Я - декорируемая функция.
```

Длинно? Длинно. Перепишем данный код без использования промежуточных переменных:

```
def decorated_function():
    print "Я - декорируемая функция."
decorated_function = decorator_maker()(decorated_function)
# выведет:
# Я создаю декораторы! Я буду вызван только раз: когда ты попросишь меня создать те
# Я возвращаю декоратор.
# Я - декоратор! Я буду вызван только раз: в момент декорирования функции.
# Я возвращаю обёрнутую функцию.

# Наконец:
decorated_function()
# выведет:
# Я - обёртка вокруг декорируемой функции. Я буду вызвана каждый раз когда ты вызыва
```

```
# Я возвращаю результат работы декорируемой функции.  
# Я - декорируемая функция.
```

А теперь ещё раз, ещё короче:

```
@decorator_maker()  
def decorated_function():  
    print "I am the decorated function."  
  
# выведет:  
# Я создаю декораторы! Я буду вызван только раз: когда ты попросишь меня создать те  
# Я возвращаю декоратор.  
# Я - декоратор! Я буду вызван только раз: в момент декорирования функции.  
# Я возвращаю обёрнутую функцию.  
  
# И снова:  
decorated_function()  
# выведет:  
# Я - обёртка вокруг декорируемой функции. Я буду вызвана каждый раз когда ты вызо  
# Я возвращаю результат работы декорируемой функции.  
# Я - декорируемая функция.
```

Вы заметили, что мы вызвали функцию, после знака "@"?:)

Вернёмся, наконец, к аргументам декораторов, ведь если мы используем функцию, чтобы создавать декораторы «на лету», мы можем передавать ей любые аргументы, верно?

```
def decorator_maker_with_arguments(decorator_arg1, decorator_arg2):  
  
    print "Я создаю декораторы! И я получил следующие аргументы:", decorator_arg1,  
    decorator_arg2  
  
    def my_decorator(func):  
        print "Я - декоратор. И ты всё же смог передать мне эти аргументы:", decorator_arg1,  
        decorator_arg2  
  
        # Не перепутайте аргументы декораторов с аргументами функций!  
        def wrapped(function_arg1, function_arg2) :  
            print ("Я - обёртка вокруг декорируемой функции.\n"  
                  "И я имею доступ ко всем аргументам: \n"  
                  "\t- и декоратора: {0} {1}\n"  
                  "\t- и функции: {2} {3}\n"  
                  "Теперь я могу передать нужные аргументы дальше"  
                  .format(decorator_arg1, decorator_arg2,  
                          function_arg1, function_arg2))  
            return func(function_arg1, function_arg2)  
  
        return wrapped  
  
    return my_decorator  
  
@decorator_maker_with_arguments("Леонард", "Шелдон")  
def decorated_function_with_arguments(function_arg1, function_arg2):  
    print ("Я - декорируемая функция и я знаю только о своих аргументах: {0}"  
          " {1}".format(function_arg1, function_arg2))  
  
decorated_function_with_arguments("Раджеш", "Говард")  
# выведет:  
# Я создаю декораторы! И я получил следующие аргументы: Леонард Шелдон  
# Я - декоратор. И ты всё же смог передать мне эти аргументы: Леонард Шелдон  
# Я - обёртка вокруг декорируемой функции.  
# И я имею доступ ко всем аргументам:  
#   - и декоратора: Леонард Шелдон  
#   - и функции: Раджеш Говард  
# Теперь я могу передать нужные аргументы дальше  
# Я - декорируемая функция и я знаю только о своих аргументах: Раджеш Говард
```

* — Прим. переводчика: в данном примере автор упоминает имена главных героев популярного сериала «Теория Большого взрыва».

Вот он, искомый декоратор, которому можно передавать произвольные аргументы.

Безусловно, аргументами могут быть любые переменные:

```
c1 = "Пенни"
c2 = "Лесли"

@decorator_maker_with_arguments("Леонард", c1)
def decorated_function_with_arguments(function_arg1, function_arg2):
    print ("Я - декорируемая функция и я знаю только о своих аргументах: {0}"
          " {1}".format(function_arg1, function_arg2))

decorated_function_with_arguments(c2, "Говард")

# выведет:
# Я создаю декораторы! И я получил следующие аргументы: Леонард Пенни
# Я - декоратор. И ты всё же смог передать мне эти аргументы: Леонард Пенни
# Я - обёртка вокруг декорируемой функции.
# И я имею доступ ко всем аргументам:
#   - и декоратора: Леонард Пенни
#   - и функции: Лесли Говард
# Теперь я могу передать нужные аргументы дальше
# Я - декорируемая функция и я знаю только о своих аргументах: Лесли Говард
```

Таким образом, мы можем передавать декоратору любые аргументы, как обычной функции. Мы можем использовать и распаковку через **args* и ***kwargs* в случае необходимости.

Но необходимо всегда держать в голове, что декоратор вызывается **ровно один раз**. Ровно в момент, когда Python импортирует Ваш скрипт. После этого мы уже не можем никак изменить аргументы, с которыми.

Когда мы пишем *"import x"* все функции из *x* **декорируются сразу же**, и мы уже не сможем ничего изменить.

Немного практики: напишем декоратор декорирующий декоратор

Если вы дочитали до этого момента и ещё в строю — вот вам бонус от меня.

Это небольшая хитрость позволит вам превратить любой обычный декоратор в декоратор, принимающий аргументы.

Изначально, чтобы получить декоратор, принимающий аргументы, мы создали его с помощью другой функции.

Мы обернули наш декоратор.

Есть ли у нас что-нибудь, чем можно обернуть функцию?

Точно, декораторы!

Давайте же немного развлечёмся и напишем декоратор для декораторов:

```
def decorator_with_args(decorator_to_enhance):
    """
    Эта функция задумывается КАК декоратор и ДЛЯ декораторов.
    Она должна декорировать другую функцию, которая должна быть декоратором.
    Лучше выпейте чашку кофе.
    Она даёт возможность любому декоратору принимать произвольные аргументы,
    избавляя Вас от головной боли о том, как же это делается, каждый раз, когда это
    """

    # Мы используем тот же трюк, который мы использовали для передачи аргументов:
    def decorator_maker(*args, **kwargs):

        # создадим на лету декоратор, который принимает как аргумент только
        # функцию, но сохраняет все аргументы, переданные своему "создателю"
        def decorator_wrapper(func):

            # Мы возвращаем то, что вернёт нам изначальный декоратор, который, в се
```



```
        # ПРОСТО ФУНКЦИЯ (возвращающая функцию).
        # Единственная ловушка в том, что этот декоратор должен быть именно так
        # decorator(func, *args, **kwargs)
        # вида, иначе ничего не сработает
        return decorator_to_enhance(func, *args, **kwargs)

    return decorator_wrapper

return decorator_maker
```

Это может быть использовано так:

```
# Мы создаём функцию, которую будем использовать как декоратор и декорируем её :- )
# Не стоит забывать, что она должна иметь вид "decorator(func, *args, **kwargs)"
@decorator_with_args
def decorated_decorator(func, *args, **kwargs):
    def wrapper(function_arg1, function_arg2):
        print "Мне тут передали...", args, kwargs
        return func(function_arg1, function_arg2)
    return wrapper

# Теперь декорируем любую нужную функцию нашим новеньким, ещё блестящим декоратором

@decorated_decorator(42, 404, 1024)
def decorated_function(function_arg1, function_arg2):
    print "Привет", function_arg1, function_arg2

decorated_function("Вселенная и", "всё прочее")

# выведет:
# Мне тут передали...: (42, 404, 1024) {}
# Привет Вселенная и всё прочее

# Уфффффф!
```

Думаю, я знаю, что Вы сейчас чувствуете.

Последний раз Вы испытывали это ощущение, слушая, как вам говорят: «Чтобы понять рекурсию необходимо для начала понять рекурсию».

Но ведь теперь Вы рады, что разобрались с этим?;)

Рекомендации для работы с декораторами

- Декораторы были введены в Python 2.4, так что узнавайте, на чём будет выполняться Ваш код.
- Декораторы несколько замедляют вызов функции, не забывайте об этом.
- Вы не можете «раздекорировать» функцию. Безусловно, существуют трюки, позволяющие создать декоратор, который можно отсоединить от функции, но это плохая практика. Правильней будет запомнить, что если функция декорирована — это не отменить.
- Декораторы оборачивают функции, что может затруднить отладку.

Последняя проблема частично решена в Python 2.5, добавлением в стандартную библиотеку модуля *functools* включающего в себя *functools.wraps*, который копирует всю информацию об оборачиваемой функции (её имя, из какого она ~~ранее~~ модуля, её docstrings и т.п.) в функцию-обёртку.

Забавным фактом является то, что *functools.wraps* — сам по себе декоратор.

```
# Во время отладки, в трассировочную информацию выводится __name__ функции.
def foo():
    print "foo"

print foo.__name__
```

```

# выведет: foo

# Однако, декораторы мешают нормальному ходу дел:
def bar(func):
    def wrapper():
        print "bar"
        return func()
    return wrapper

@bar
def foo():
    print "foo"

print foo.__name__
# выведет: wrapper

# "functools" может нам с этим помочь

import functools

def bar(func):
    # Объявляем "wrapper" оборачивающим "func"
    # и запускаем магию:
    @functools.wraps(func)
    def wrapper():
        print "bar"
        return func()
    return wrapper

@bar
def foo():
    print "foo"

print foo.__name__
# выведет: foo

```

Как можно использовать декораторы?

И в заключение, я бы хотел ответить на вопрос, который я часто слышу: зачем же нужны декораторы? Как их можно использовать?

Декораторы могут быть использованы для расширения возможностей функций из сторонних библиотек (код которых мы не можем изменять), или для упрощения отладки (мы не хотим изменять код, который ещё не устоялся).

Так же полезно использовать декораторы для расширения различных функций одним и тем же кодом, без повторного его переписывания каждый раз, например:

```

def benchmark(func):
    """
    Декоратор, выводящий время, которое заняло
    выполнение декорируемой функции.
    """
    import time
    def wrapper(*args, **kwargs):
        t = time.clock()
        res = func(*args, **kwargs)
        print func.__name__, time.clock() - t
        return res
    return wrapper

def logging(func):
    """
    Декоратор, логирующий работу кода.
    (хорошо, он просто выводит вызовы, но тут могло быть и логирование!)
    """

```



```

def wrapper(*args, **kwargs):
    res = func(*args, **kwargs)
    print func.__name__, args, kwargs
    return res
return wrapper

def counter(func):
    """
    Декоратор, считающий и выводящий количество вызовов
    декорируемой функции.
    """
    def wrapper(*args, **kwargs):
        wrapper.count += 1
        res = func(*args, **kwargs)
        print "{0} была вызвана: {1}x".format(func.__name__, wrapper.count)
        return res
    wrapper.count = 0
    return wrapper

@benchmark
@logging
@counter
def reverse_string(string):
    return str(reversed(string))

print reverse_string("A роза упала на лапу Азора")
print reverse_string("A man, a plan, a canoe, pasta, heros, rajahs, a coloratura, maps")

# выведет:
# reverse_string ('A роза упала на лапу Азора',) {}
# wrapper 0.0
# reverse_string была вызвана: 1x
# арозА упал ан алапу азор А
# reverse_string ('A man, a plan, a canoe, pasta, heros, rajahs, a coloratura, maps') {}
# wrapper 0.0
# reverse_string была вызвана: 2x
# !amanaP :lanac a ,noep a ,stah eros ,raj a ,hsac ,oloR a ,tur a ,mapS ,snip ,eper

```

Таким образом, декораторы можно применить к любой функции, расширив её функционал и не переписывая ни строчки кода!

```

import httpplib

@benchmark
@logging
@counter
def get_random_futurama_quote():
    conn = httpplib.HTTPConnection("slashdot.org:80")
    conn.request("HEAD", "/index.html")
    for key, value in conn.getresponse().getheaders():
        if key.startswith("x-b") or key.startswith("x-f"):
            return value
    return "Эх, нет... не могу!"

print get_random_futurama_quote()
print get_random_futurama_quote()

#outputs:
#get_random_futurama_quote () {}
#wrapper 0.02
#get_random_futurama_quote была вызвана: 1x
#The laws of science be a harsh mistress.
#get_random_futurama_quote () {}

```

```
#wrapper 0.01
#get_random_futurama_quote была вызвана: 2x
#Curse you, merciful Poseidon!
```

В Python включены такие декораторы как *property*, *staticmethod* и т.д.
В Django декораторы используются для управления кешированием, контроля за правами доступа и определения обработчиков адресов. В Twisted — для создания поддельных асинхронных inline-вызовов.
Декораторы открывают широчайший простор для экспериментов! И надеюсь, что данная статья поможет Вам в его освоении!
Спасибо за внимание!

Содержание:

- Шаг 1. Базовое понимание декораторов
- **Шаг 2. Продвинутое использование декораторов** (вы сейчас прочитали эту статью)

Прим. переводчика:
Буду благодарен любым замечаниям по переводу и оформлению. Надеюсь, вторая часть статьи покажется Вам более неочевидной и полезной, чем первая.


Теги: python, decorator, decorators, декораторы, step-by-step, перевод

Хабы: Разработка веб-сайтов, Python

Редакторский дайджест

Присылаем лучшие статьи раз в месяц

Электронпочта



94


0


Карма Рейтинг

Владислав Степанов @Utter_step

Пользователь


Комментарии 25



- 


05.04.2012 в 22:27
- НЛО прилетело и опубликовало эту надпись здесь
- 

0

Ответить







- 

05.04.2012 в 22:29
- НЛО прилетело и опубликовало эту надпись здесь
- 

0

Ответить




- 

Utter_step 05.04.2012 в 22:41
- Всё верно.
- Я и указал, что, возможно, стоит «освежить память».

Плюс, в данном месте в текущей статье указывается на тот факт, что при декорировании функции декоратором декоратор будет обязательно вызван, так как это просто функция.
Момент, при его осознании, очевидный, но не самоочевидный.)


0 Ответить

 **qmax** 06.04.2012 в 00:00

А на КДПВ реальный код?


@staticmethod декорирует статические методы класса, первым аргументом котрых является переменная связанная с классом (обычно называется cls).
Тоесът, аргумент filename будет связан с определением класса.

-1 Ответить

 **qmax** 06.04.2012 в 00:00 ^


Я туплю. Попутал с @classmethod.

0 Ответить

 06.04.2012 в 00:23

НЛО прилетело и опубликовало эту надпись здесь

0 Ответить

 **Deerwalker** 06.04.2012 в 01:37 ^

Это антипаттерн для питона — класс с одним методом __call__.

0 Ответить

 06.04.2012 в 02:04 ^

НЛО прилетело и опубликовало эту надпись здесь

0 Ответить

 **Monnoroch** 06.04.2012 в 05:09 ^

Вообще в питоне функция — это объект, как раз и поддерживающий инициализацию (ну оно же определение, кстати с той же реализацией примерно, что и у вас) и вызов. Таким образом вы обернули объект (функцию) в точно такой же по структуре объект (класс с двумя методами) без модификации поведения изначального (функции). То есть вы ввели ничего не делающую сущность. Ну и нафига она такая ничего не делающая?
Фу-фу-фу вам :)

+1 Ответить

 06.04.2012 в 11:44 ^

НЛО прилетело и опубликовало эту надпись здесь

0 Ответить

 06.04.2012 в 12:26 ^

НЛО прилетело и опубликовало эту надпись здесь

0 Ответить

 06.04.2012 в 12:44 ^


НЛО прилетело и опубликовало эту надпись здесь

0 Ответить

 06.04.2012 в 12:48 ^

НЛО прилетело и опубликовало эту надпись здесь

0 Ответить

 **svetlov** 06.04.2012 в 16:19 ^


Это — общепринятый способ. Странно, почему вас напрягает повсеместно распространенный подход.

0 Ответить

 06.04.2012 в 17:15 ^

НЛО прилетело и опубликовало эту надпись здесь

0 Ответить

 **svetlov** 06.04.2012 в 16:17 ^


1. Декоратор, реализованный классом, длиннее записывается и медленнее работает.
2. Если вам захотелось вернуть None из декоратора — скорее всего что-то не так в архитектуре и декоратор используется не по назначению.

0 Ответить

 06.04.2012 в 17:11 ^

НЛО прилетело и опубликовало эту надпись здесь

0 Ответить

 **svetlov** 06.04.2012 в 17:25 ^

Декоратор, возвращающий переданную ему функцию без изменений — это нормально. Если же он возвращает None — это обескураживает.

```
@service(123)
def foo(a, b, c='d'):
    pass
```

превращается в `foo = None`

Согласитесь, не самая очевидная запись. Я стараюсь избегать такого кода. Лучше вернуть функцию с совместимой сигнатурой, которая будет выбрасывать исключение при вызове. Приемлемым вариантом было бы ``del foo``, но из декоратора такое не сделаешь.

Если пишете декоратор на C — придется создавать классы. Для Питона это избыточно в подавляющем большинстве случаев.

+3 Ответить

 **Warlock2** 06.04.2012 в 01:19

Извините за оффтоп, но не могли бы вы сказать что за цветовая схема для idea (?)

0 Ответить

 **Monnoroch** 06.04.2012 в 02:04 ^

Это sublime-text 2, насколько я понимаю. А схема — что-то похожее на Dark in pastels, но не уверен.

+2 Ответить

 **hellman** 06.04.2012 в 10:11 ^

Схема Monokai

+1 Ответить

 **Monnoroch** 06.04.2012 в 05:13

напишем декоратор декорирующий декоратор, декорирующий функции

Что-то это мне очень сегодняшнюю статью напоминает...

0 Ответить

 **arm0** 14.11.2013 в 02:31

```
def counter(func):
    """
    Декоратор, считающий и выводящий количество вызовов
    декорируемой функции.
    """
    counter.count[func.__name__] = 0
    def wrapper(*args, **kwargs):
        counter.count[func.__name__] += 1
        res = func(*args, **kwargs)
        print "{0} была вызвана: {1}x".format(func.__name__, counter.count[func.__name__])
        return res
    return wrapper
```


Похоже, не хватает строки после декоратора

```
counter.count = {}
```

Посмотрел на StackOverflow. Ответ исправили, и получилось изящней:


```
def counter(func):
    """
    A decorator that counts and prints the number of times a function has been executed
    """
    def wrapper(*args, **kwargs):
        wrapper.count = wrapper.count + 1
        res = func(*args, **kwargs)
        print "{0} has been used: {1}x".format(func.__name__, wrapper.count)
        return res
    wrapper.count = 0
    return wrapper
```

+1 Ответить

 **Utter_step** 14.11.2013 в 14:39

Спасибо за замечание, действительно была ошибка + не самый разумный подход со словарём, исправил текст поста.)

+1 Ответить

 **arm0** 14.11.2013 в 15:15

Еще меня смущает запись:

```
str(reversed(string))
```

Выполнение этого выражения дает что-то вроде: "<reversed object at 0x10b0b2dd0>"

Предполагаю, что в старых версиях интерпретатора все работало верно. А возможно, автор использовал такую запись как псевдокод для наглядности. Попрошу знатиков разъяснить.

Возможные решения:

```
string[::-1]
# или
"".join(reversed(string))
```

+1 Ответить

Только полноправные пользователи могут оставлять комментарии. Войдите, пожалуйста.

ПОХОЖИЕ ПУБЛИКАЦИИ

15 июня в 14:59
Как UX ресерчеру найти ключ к сердцу РО? Step by step

+4 425 6 0

5 июня 2013 в 06:46
Step-by-step: настройка SpecFlow для русскоязычного проекта при написании тестов в среде .Net

+13 23K 59 6 +6

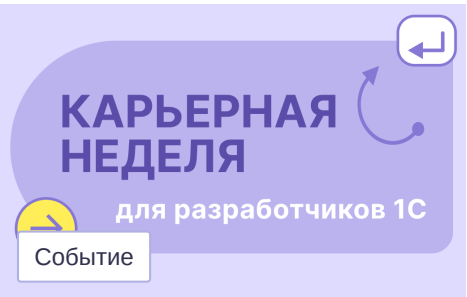
4 апреля 2012 в 22:19
Понимаем декораторы в Python'е, шаг за шагом. Шаг 1

+93 1 1172 37

МИНУТОЧКУ ВНИМАНИЯ



Тетрис на стероидах: тестируем War Robots на Steam Deck



Карьерная неделя для разработчиков 1С в телеграме



Помогите динозаврику добыть контент по Data Mining

КУРСЫ

Python для анализа данных
23 сентября 2022 · 42 000 ₽ · Нетология

Python: анализ данных и машинное обучение
3 октября 2022 · 31 000 ₽ · Loftschool

Программирование на языке Python. Уровень 1. Базовый курс
18 сентября 2022 · 30 990 ₽ · Специалист.ру

Программирование на Python. Уровень 1. Основы программирования
19 сентября 2022 · 25 000 ₽ · АИС

Основы программирования на Python. Уровень 2
20 сентября 2022 · 19 500 ₽ · Level UP

Больше курсов на Хабр Карьере

сегодня в 03:45

Отечественная микроэлектроника — как выдать нищету за добродетель

 +127  17K  36  155 +155

сегодня в 08:22

Паровой мотоцикл своими руками

 +105  6.6K  28  57 +57

вчера в 14:01

Пациенты, которым удаляли зубы, а боль не проходила

 +54  6.8K  39  23 +23


вчера в 15:20

Зачем работодатели требуют наличие ВО и почему это оправданно

 +40  26K  49  237 +237

вчера в 16:49

Почему аспирантура не только зло, но и...

 +38  6.4K  26  48 +48

DAST ist fantastisch: отечественный динамический анализатор к взлету готов

Мегапост

РАБОТА

Python разработчик
156 вакансий

Django разработчик
54 вакансии

Data Scientist
138 вакансий

Все вакансии



 [Настройка языка](#)

[Техническая поддержка](#)

[Полная версия](#)

[Вернуться на старую версию](#)

