 Utter\_step 4 апреля 2012 в 22:19

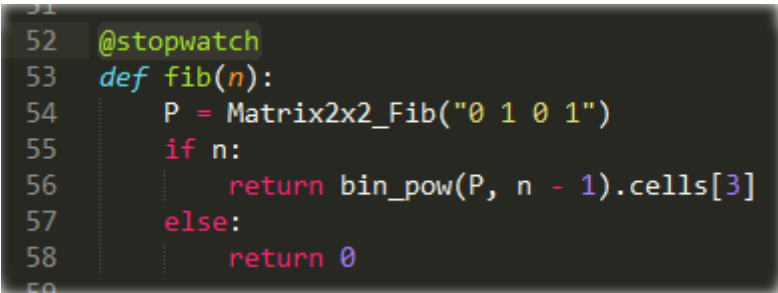
## Понимаем декораторы в Python'е, шаг за шагом. Шаг 1

Разработка веб-сайтов\*, Python\*

Перевод

Tutorial

Автор оригинала: Renaud Gaudin



На Хабре множество раз обсуждалась тема декораторов, однако, на мой взгляд, данная статья (выросшая из одного [вопроса на stackoverflow](#)) описывает данную тему наиболее понятно и, что немаловажно, является «пошаговым руководством» по использованию декораторов, позволяющим новичку овладеть этой техникой сразу на достойном уровне.

Итак, что же такое «декоратор»?

Впереди достаточно длинная статья, так что, если кто-то спешит — вот пример того, как работают декораторы:

```
def makebold(fn):
    def wrapped():
        return "<b>" + fn() + "</b>"
    return wrapped

def makeitalic(fn):
    def wrapped():
        return "<i>" + fn() + "</i>"
    return wrapped

@makebold
@makeitalic
def hello():
    return "hello habr"

print hello() ## выведет <b><i>hello habr</i></b>
```

Те же из вас, кто готов потратить немного времени, приглашаются прочесть длиииинный пост.

### Функции в Python'е являются объектами

Для того, чтобы понять, как работают декораторы, в первую очередь следует осознать, что в Python'е функции — это тоже объекты.

Давайте посмотрим, что из этого следует:

```
def shout(word="да"):
    return word.capitalize()+"!"

print shout()
# выведет: 'Да!'
```

# Так как функция - это объект, вы связать её с переменной,  
# как и любой другой объект

```
scream = shout
```

```

# Заметьте, что мы не используем скобок: мы НЕ вызываем функцию "shout",
# мы связываем её с переменной "scream". Это означает, что теперь мы
# можем вызывать "shout" через "scream":

print scream()
# выведет: 'Да!'

# Более того, это значит, что мы можем удалить "shout", и функция всё ещё
# будет доступна через переменную "scream"

del shout
try:
    print shout()
except NameError, e:
    print e
    #выведет: "name 'shout' is not defined"

print scream()
# выведет: 'Да!'

```

Запомним этот факт, скоро мы к нему вернёмся, но кроме того, стоит понимать, что функция в Python'e может быть определена... внутри другой функции!

```

def talk():
    # Внутри определения функции "talk" мы можем определить другую...
    def whisper(word="да"):
        return word.lower()+"...";

    # ... и сразу же её использовать!
    print whisper()

# Теперь, КАЖДЫЙ РАЗ при вызове "talk", внутри неё определяется а затем
# и вызывается функция "whisper".
talk()
# выведет: "да..."

# Но вне функции "talk" НЕ существует никакой функции "whisper":
try:
    print whisper()
except NameError, e:
    print e
    #выведет : "name 'whisper' is not defined"

```

## Ссылки на функции

Ну что, вы всё ещё здесь?:)

Теперь мы знаем, что функции являются полноправными объектами, а значит:

- могут быть связаны с переменной;
- могут быть определены одна внутри другой.

Что ж, а это значит, что одна функция может вернуть другую функцию!

Давайте посмотрим:

```

def getTalk(type="shout"):

    # Мы определяем функции прямо здесь
    def shout(word="да"):
        return word.capitalize()+"!"

    def whisper(word="да") :

```

```

        return word.lower()+"...";

# Затем возвращаем необходимую
if type == "shout":
    # Заметьте, что мы НЕ используем "()", нам нужно не вызвать функцию,
    # а вернуть объект функции
    return shout
else:
    return whisper

# Как использовать это непонятное нечто?
# Возьмём функцию и свяжем её с переменной
talk = getTalk()

# Как мы можем видеть, "talk" теперь - объект "function":
print talk
# выведет: <function shout at 0xb7ea817c>

# Который можно вызывать, как и функцию, определённую "обычным образом":
print talk()

# Если нам захочется - можно вызвать её напрямую из возвращаемого значения:
print getTalk("whisper")()
# выведет: да...

```

Подождите, раз мы можем возвращать функцию, значит, мы можем и передавать её другой функции, как параметр:

```

def doSomethingBefore(func):
    print "Я делаю что-то ещё, перед тем как вызвать функцию, которую ты мне передал"
    print func()

doSomethingBefore(scream)
#выведет:
# Я делаю что-то ещё, перед тем как вызвать функцию, которую ты мне передал
# Да!

```

Ну что, теперь у нас есть все необходимые знания для того, чтобы понять, как работают декораторы.

Как вы могли догадаться, декораторы — это, по сути, просто своеобразные «обёртки», которые **дают нам возможность делать что-либо до и после того, что сделает декорируемая функция, не изменяя её.**

**Создадим свой декоратор «вручную»**

```

# Декоратор - это функция, ожидающая ДРУГУЮ функцию в качестве параметра
def my_shiny_new_decorator(a_function_to_decorate):
    # Внутри себя декоратор определяет функцию-"обёртку".
    # Она будет (что бы вы думали?..) обёрнута вокруг декорируемой,
    # получая возможность исполнять произвольный код до и после неё.

    def the_wrapper_around_the_original_function():
        # Поместим здесь код, который мы хотим запускать ДО вызова
        # оригинальной функции
        print "Я - код, который отработает до вызова функции"

        # ВЫЗОВЕМ саму декорируемую функцию
        a_function_to_decorate()

        # А здесь поместим код, который мы хотим запускать ПОСЛЕ вызова
        # оригинальной функции
        print "А я - код, срабатывающий после"

```

```
# На данный момент функция "a_function_to_decorate" НЕ ВЫЗЫВАЛАСЬ НИ РАЗУ

# Теперь, вернём функцию-обёртку, которая содержит в себе
# декорируемую функцию, и код, который необходимо выполнить до и после.
# Всё просто!
return the_wrapper_around_the_original_function

# Представим теперь, что у нас есть функция, которую мы не планируем больше трогать
def a_stand_alone_function():
    print "Я простая одинокая функция, ты ведь не посмеешь меня изменять?.."

a_stand_alone_function()
# выведет: Я простая одинокая функция, ты ведь не посмеешь меня изменять?..

# Однако, чтобы изменить её поведение, мы можем декорировать её, то есть
# Просто передать декоратору, который обернет исходную функцию в любой код,
# который нам потребуется, и вернёт новую, готовую к использованию функцию:

a_stand_alone_function_decorated = my_shiny_new_decorator(a_stand_alone_function)
a_stand_alone_function_decorated()
#выведет:
# Я - код, который отработает до вызова функции
# Я простая одинокая функция, ты ведь не посмеешь меня изменять?..
# А я - код, срабатывающий после
```

Наверное, теперь мы бы хотели, чтобы каждый раз, во время вызова *a\_stand\_alone\_function*, вместо неё вызывалась *a\_stand\_alone\_function\_decorated*. Нет ничего проще, просто перезапишем *a\_stand\_alone\_function* функцией, которую нам вернул *my\_shiny\_new\_decorator*:

```
a_stand_alone_function = my_shiny_new_decorator(a_stand_alone_function)
a_stand_alone_function()
#выведет:
# Я - код, который отработает до вызова функции
# Я простая одинокая функция, ты ведь не посмеешь меня изменять?..
# А я - код, срабатывающий после
```

Вы ведь уже догадались, что это ровно тоже самое, что делают @декораторы.:)

## Разрушаем ореол таинственности вокруг декораторов

Вот так можно было записать предыдущий пример, используя синтаксис декораторов:

◆ +93    👁 376K    📖 1172    ➡

```
def another_stand_alone_function():
    print "Оставь меня в покое"

another_stand_alone_function()
#выведет:
# Я - код, который отработает до вызова функции
# Оставь меня в покое
# А я - код, срабатывающий после
```

Да, всё действительно так просто! @decorator — просто синтаксический сахар для конструкций вида:

```
another_stand_alone_function = my_shiny_new_decorator(another_stand_alone_function)
```

Декораторы — это просто pythonic-реализация паттерна проектирования «Декоратор». В Python включены некоторые классические паттерны проектирования, такие как рассматриваемые в этой статье декораторы, или привычные любому пайтонисту итераторы.

Конечно, можно вкладывать декораторы друг в друга, например так:

```
def bread(func):
    def wrapper():
        print "</-----\>"
        func()
        print "<\_____/>"
    return wrapper

def ingredients(func):
    def wrapper():
        print "#помидоры#"
        func()
        print "~салат~"
    return wrapper

def sandwich(food="--ветчина--"):
    print food

sandwich()
#выведет: --ветчина--
sandwich = bread(ingredients(sandwich))
sandwich()
#выведет:
# </-----\>
# #помидоры#
# --ветчина--
# ~салат~
# <\_____/>
```

И используя синтаксис декораторов:

```
@bread
@ingredients
def sandwich(food="--ветчина--"):
    print food

sandwich()
#выведет:
# </-----\>
# #помидоры#
# --ветчина--
# ~салат~
# <\_____/>
```

Следует помнить о том, что порядок декорирования ВАЖЕН:

```
@ingredients
@bread
def sandwich(food="--ветчина--"):
    print food

sandwich()
#выведет:
# #помидоры#
# </-----\>
# --ветчина--
# <\_____/>
# ~салат~
```

На этом моменте Вы можете счастливо уйти, с осознанием того, что вы поняли, что такое декораторы и с чем их едят.

Для тех же, кто хочет помучать ещё немного свой мозг, завтра будет допереведена вторая часть статьи, посвящённая продвинутому использованию декораторов.

Содержание:

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

Прим. переводчика:

Спасибо за внимание. Буду благодарен любым замечаниям по переводу и оформлению, постараюсь учесть их все во второй части перевода.

UPD: Во второй части будут разобраны такие вопросы, как: передача аргументов декорируемым функциям, декорация методов, декораторы с параметрами и пр.

UPD2: Выложена **вторая часть статьи**.

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

Хабы: Разработка веб-сайтов, Python

### Редакторский дайджест

Присылаем лучшие статьи раз в месяц

Электронная почта



94

0


Карма

Рейтинг

Владислав Степанов @Utter\_step

Пользователь


Комментарии 37

- 

dunki


04.04.2012 в 23:48



спасибо. ждем вторую часть



+4

Ответить





- 

krovatti


05.04.2012 в 01:21



Присоединяюсь.



+2

Ответить





- 

nvbn


05.04.2012 в 00:04



Что в декораторах такого волшебного и непостижимого, заставляющего писать очередной пост про них?



+19

Ответить




- 

Utter\_step


05.04.2012 в 00:13

Для людей, давно и серьёзно занимающихся программированием — почти ничего.  
Для новичков, как показывает практика, эта тема неочевидна.

Плюс, во второй части будет больше неочевидных вещей, таких как: фабрики декораторов, декораторы с параметрами, декораторы для методов и т.д.


Надеюсь, наличие данного материала на русском никому не повредит:)

 **+3** Ответить  

 **krovatti** 05.04.2012 в 00:51 


Я больше скажу. Некоторым еще и полезна будет.

 **+3** Ответить  

 **ddsl** 05.04.2012 в 06:48 

Напишите, если не трудно про у-комбинатор, а то статья на хабре просто мне мозг выломала( хотя там строчек кода штук 5 на пример). Так полностью и не понял.

 **+3** Ответить  

 05.04.2012 в 12:32 

НЛО прилетело и опубликовало эту надпись здесь

 **0** Ответить  

 **Astynax** 06.04.2012 в 09:24 

Y-комбинатор позволяет в анонимной функции использовать рекурсию с использованием самой себя, без необходимости назначения ей имени — функция остаётся анонимной.

Скажем, у нас есть неанонимная рекурсивная функция вычисления факториала:

```
def fact(n):
    return n * fact(n-1) if n > 0 else 1
print fact(10)
```

Это работает, т.к. у нашей функции есть имя fact и мы можем по нему функцию вызывать внутри неё самой.

Но если мы захотим написать анонимную функцию вычисления факториала, мы столкнемся с проблемой — как нам внутри функции вызвать её саму, не имея имени.

Можно попробовать сделать так:

```
fact = lambda self, n: n * self(self, n) if n > 0 else 1
print fact(fact, 10)
```

В данном случае имя функции не встречается внутри неё. Но при внешнем вызове функции приходится передавать в качестве одного из аргументов ссылку на неё саму. Это можно обернуть так:

```
wrap = lambda fn, *args: fn(fn, *args)
fact = lambda self, n: n * self(self, n) if n > 0 else 1
print wrap(fact, 10)
```

Теперь мы её ещё немного дообернем:

```
y = lambda fn: lambda *args: fn(fn, *args)
fact = y(lambda self, n: n * self(self, n) if n > 0 else 1)
```

Вот у и есть Y-комбинатор, к тому же анонимный. Пользуем inline как то так:

```
# применяем анонимную рекурсивную функцию
map(
    (lambda fn: lambda *args: fn(fn, *args))(
        lambda self, n: n * self(self, n-1) if n > 0 else 1),
```

```
[1, 2, 3]
)
```

PROFIT! :)

 **+3** [Ответить](#)  

 **monolithed** 05.04.2012 в 08:21 

Видимо тем, что остальные статьи заканчивались на первом примере этого поста.  
Иными словами автор выбрал несколько иной подход, сперва привел итоговый пример, а затем разложил все по полочкам, так сказать сохранил интригу и не дал заснуть.

 **+2** [Ответить](#)  

 **lightman** 05.04.2012 в 12:32 



Что в декораторах такого волшебного и непостижимого, заставляющего писать очередной пост про них?

Для человека, издревле работающего с питоно-подобными языками ничего.

А вот программера, взрощенного на светлых идеалах паскале- и си-подобных языков, поначалу несколько обескураживает фраза «каждая функция есть объект». А последующее знакомство с теми же декораторами вообще вводит в некоторое замешательство, требующее осознания и последующей перестройки некоторых принципов, укрепившихся в мозге. Вот тут-то подобные статьи и оказываются очень кстати.



Ведь чем более разнообразным количеством слов и статей будет описана некая сущность, тем проще её разложить по полочкам и осознать.

 **0** [Ответить](#)  

 **VoICh** 05.04.2012 в 12:55 

На Си всё не так плохо, если не требуется метапрограммирование и прочий манкипатчинг. Указатели на функцию в данном контексте являются практически полным аналогом ссылки на объект Function — можно присваивать переменным, можно вызывать из переменной, можно передавать в другие функции, а значит можно использовать паттерн «декоратор».

 **+2** [Ответить](#)  

 **qmax** 05.04.2012 в 23:10 



хаскель-барби смотрит на питоновцев и точно также (только с большими глазами) недоумевает — чего такого волшебного в монадах, что о них пишет какждый, освоивший хаскель и умеющий писать.

 **0** [Ответить](#)  

 **niko83** 05.04.2012 в 00:32


В статье не упоминается что декораторы могут быть применены не только к функциям, но и к целым классам! Таким образом они могут управлять вызовами классаов с целью создания экземпляров, и самими объектами классов, например добавлять новые методы в классы.

 **0** [Ответить](#)  

 **niko83** 05.04.2012 в 00:36 

так же нет инфы что декораторы могут вызываться с аргументами

 **0** [Ответить](#)  

 **Utter\_step** 05.04.2012 в 00:43 

Если бы вы посмотрели по ссылке на [оригинал](#), то заметили бы, что после данного материала идёт как раз «Passing arguments to the decorated function» («Передаем аргументы декорируемым функциям») и «Decorating methods» («Декорируем методы»). Вызов декораторов с параметрами так-




же будет во второй части, как я успел упомянуть в этом комментарии.

Резюмируя: не спешите, всё будет:)

И спасибо за критику!

0 Ответить


 **mc\_dir** 05.04.2012 в 00:52

[habrahabr.ru/post/74838/](http://habrahabr.ru/post/74838/)  
[habrahabr.ru/post/46306/](http://habrahabr.ru/post/46306/)  
[habrahabr.ru/post/86255/](http://habrahabr.ru/post/86255/)  
[habrahabr.ru/post/139866/](http://habrahabr.ru/post/139866/)

Вот ей богу не надоело еще ??? Мне даже кармы не жалко — но... прям rrr... Я сам php разработчик, начал недавно изучать python. Так вот в питоне куча всего интересного, помимо декораторов. И ими начинающие вряд ли со старта пользоваться будут, а если будут то еще быстрее запутаются (особенно полезен совет вложить декоратор в декоратор — вордпрес напоминает 2 версии, там не декораторы но пол движка на хуках...).

Одновременное использование нескольких версий на одной машине (linux), параллельные вычисление, отладка кода, оптимизация, подсчет занимаемой памяти и куча всего всего — довольно ж сильный язык жеж? Что мусолить одну тему по 10 раз?

+8 Ответить

 **Utter\_step** 05.04.2012 в 01:07

Спасибо за комментарий!

Попробую ответить на всё.

[habrahabr.ru/post/74838/](http://habrahabr.ru/post/74838/)  
[habrahabr.ru/post/46306/](http://habrahabr.ru/post/46306/)  
[habrahabr.ru/post/86255/](http://habrahabr.ru/post/86255/)  
[habrahabr.ru/post/139866/](http://habrahabr.ru/post/139866/)

1. На Хабре ≈140000 статей, не удивительно, что есть статьи по схожим темам. Уверяю вас, перед тем, как браться за перевод я поискал материалы по данной теме на Хабре и, по моему мнению, тот, что перевожу я, не является дублирующим в целом.

ими начинающие вряд ли со старта пользоваться будут, а если будут то еще быстрее запутаются

2. Со старта — не будут. Но когда-нибудь ведь начнут:)


особенно полезен совет вложить декоратор в декоратор

3. В данной статье нету советов, лишь описание того, как это работает.

Что мусолить одну тему по 10 раз?

4. Надеюсь, вторая часть Вам покажется более «незамусолоной».

+4 Ответить

 **Yanzay** 05.04.2012 в 01:15

Отличная статья, спасибо! Просто и понятно.

0 Ответить

 **TBilyn** 05.04.2012 в 02:04

У меня есть вопрос, возможно глупый, но все же:  
зачем разработчики python сделали что функция-декоратор возвращает функцию-обёртку, которая

затем вызывается? Как мне это можно было сделать следующим образом: когда вызывается декорированная функция, вместо нее просто вызывается функция-декоратор, который получает первым аргументом данную функцию.

Демонстрация кодом:

```
def decorator ():  
    ...
```

```
@ decorator  
def func ():  
    ...
```

Сейчас сделано так:


```
decorator (func) ()
```

Мой вариант:

```
decorator (func)
```

Я не говорю что мой вариант лучше.

 0    Ответить        ...

 **VolCh** 05.04.2012 в 02:32    ^

Потому что func без скобок возвращает объект Function. Если мы декорируем его, то тип возврата не должен изменяться, декорация должна быть прозрачной для вызывающего кода. Везде где используется func, должна использоваться и decorator(func). Например, вложенные декораторы — как вы в своем варианте их реализуете?

 +1    Ответить        ...

 **Utter\_step** 05.04.2012 в 02:34    ^

Ваш вариант сработает для декораторов простейших функций.  
Но, предположим, Вам нужно декорировать такую ф-ию:

```
def func(var):  
    print "I've got", var
```

Тогда, вызвав

```
decorator (func) ("10$")
```

мы получим ожидаемое поведение.

В случае

```
decorator (func)
```

нам просто «некуда» передать аргумент (как пробрасывать аргументы через декоратор — в скором времени будет показано).

Надеюсь, я верно понял суть Вашего вопроса?:)

 0    Ответить        ...

 **KlonKaktusa** 05.04.2012 в 08:49

```
# Возьмём функцию и запишем её в переменную  
talk = getTalk()
```

Скобки же не нужны?

 +1    Ответить        ...

Utter\_step 05.04.2012 в 08:55 ^

Функция getTalk возвращает другую функцию.

+1 Ответить

qzer 05.04.2012 в 10:55

что-за редактор, шрифт, тема на скрине?

0 Ответить

Utter\_step 05.04.2012 в 11:03 ^

Sublime Text 2

0 Ответить

LighteR 05.04.2012 в 11:49

Теперь мы знаем, что функции являются полноправными объектами

Из ваших примеров этого не следует. В своих примерах вы лишь показываете, что в питоне есть функции высшего порядка. Они, например, есть и в php, но, тем не менее, в php функция не является объектами.

+1 Ответить

fata1ex 05.04.2012 в 12:12

А еще там есть отличные ответы этого же человека про итераторы и метаклассы. Непонятно только, зачем все это переводить в бесконечном количестве. Все подобные хорошие посты в первых 2-3 ссылках по запросам в поисковике.

+1 Ответить

funca 05.04.2012 в 18:58

Да, всё действительно так просто! @decorator — просто синтаксический сахар для конструкций вида:

Декораторы — это просто pythonic-реализация паттерна проектирования «Декоратор».

второе утверждение не верное. пруф.

конечно, это перевод и придирка относится к оригинальному тексту.

0 Ответить

qmax 05.04.2012 в 23:02

«вы можете назначить её переменной», «помещаем в переменную», «записаны в переменную»  
Есть хороший термин «связать с перменной», без риска свалиться в вопросы — где располагается то место в которое что-то помещается, сколько места занимает ссылка на объект, итп.

0 Ответить

Utter\_step 05.04.2012 в 23:37 ^

Спасибо, тоже резал глаз этот момент, но про связку забыл.:)


Fixed

0 Ответить

qmax 05.04.2012 в 23:07

«раз мы можем возвращать функцию, значит, мы можем и передавать её другой функции, как параметр»  
Это два разных явления, являющимися следствиями того, что функции являются объектами «первого класса».

 0    Ответить        ...

 28.04.2012 в 04:09

НЛО прилетело и опубликовало эту надпись здесь

 0    Ответить        ...

 **lpeasocks** 24.05.2015 в 22:28

великолепный пример с седвичем.

 0    Ответить        ...

 **thornni** 12.03.2019 в 09:14

Просто отличная оригинальная статья и замечательный ее перевод! Спасибо огромное. Самый приятный, нескучный и понятный стиль объяснения.

 0    Ответить        ...

 **shaman4d** 26.07.2019 в 15:05



Спасибо огромное самое вменяемое объяснение.

 0    Ответить        ...

 **hulitolku** 27.08.2022 в 23:54 

А как завернуть декоратор, начинающийся с собаки @ в if ?  
На сколько я понимаю, декоратор работает только без отступов от начала строки.

 0    Ответить        ...

 **suhanoves** 02.09.2022 в 17:42 

В питоне всё есть объект. Думайте о декораторе как об обычном `decorator(func)` , и если надо используйте хоть в `if`, хоть где-либо ещё. Единственное что, ветвление `if` будет оценивать булевый тип.

 0    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

5 апреля 2012 в 21:49

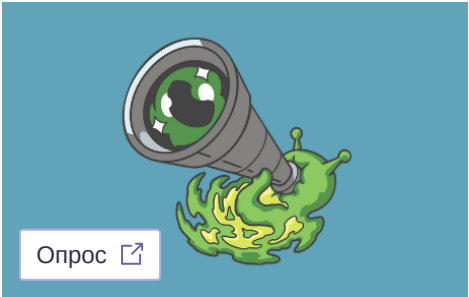
Понимаем декораторы в Python'е, шаг за шагом. Шаг 2

+61 193K 745 25

МИНУТОЧКУ ВНИМАНИЯ



Промокод — твой билет в общество потребления



Хотите рассказать о себе в наших социальных сетях?



Назад в сезон Java: читаем лучшие статьи

ВОПРОСЫ И ОТВЕТЫ

Как запустить x2gobroker в gunicorn?  
Python · Простой · 0 ответов

Как отследить изменения поля класса, если это список?  
Python · Простой · 0 ответов

Как отсортировать вывод значений согласно списку (list)?  
Python · Простой · 2 ответа

Что сделать чтобы скрипт работал при блокировке?  
Python · Простой · 0 ответов

Как проверить прокси с авторизацией на валидность через python?  
Python · Простой · 1 ответ

Больше вопросов на Хабр Q&A

ЛУЧШИЕ ПУБЛИКАЦИИ ЗА СУТКИ

сегодня в 03:45

Отечественная микроэлектроника — как выдать нищету за добродетель

+120 16K 31 152 +152

сегодня в 08:22

Паровой мотоцикл своими руками

+101 6.2K 26 51 +51

вчера в 14:01

Пациенты, которым удаляли зубы, а боль не проходила

+54 6.7K 37 22 +22

вчера в 13:04

МойОфис выпустил релиз 2.2. Более 700 улучшений в Mailion, редакторах документов и других продуктах компании

+39 3.2K 28 11 +11

вчера в 16:49

Руководство по крафту: во что можно превратить статью по Data Mining

Мегалост

РАБОТА


Python разработчик  
157 вакансий

Data Scientist  
138 вакансий

Django разработчик  
54 вакансии

Все вакансии



 [Настройка языка](#)

[Техническая поддержка](#)

[Полная версия](#)

[Вернуться на старую версию](#)