
03. Декоратори

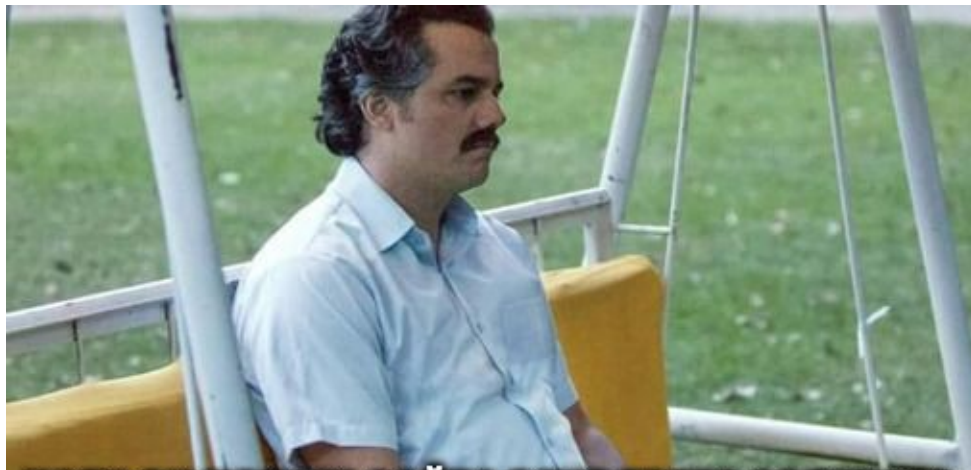
— 18 октомври 2022 —

Сайт/форум

Когато студентите с
влизането питат дали вече има сайт



Сайт/форум



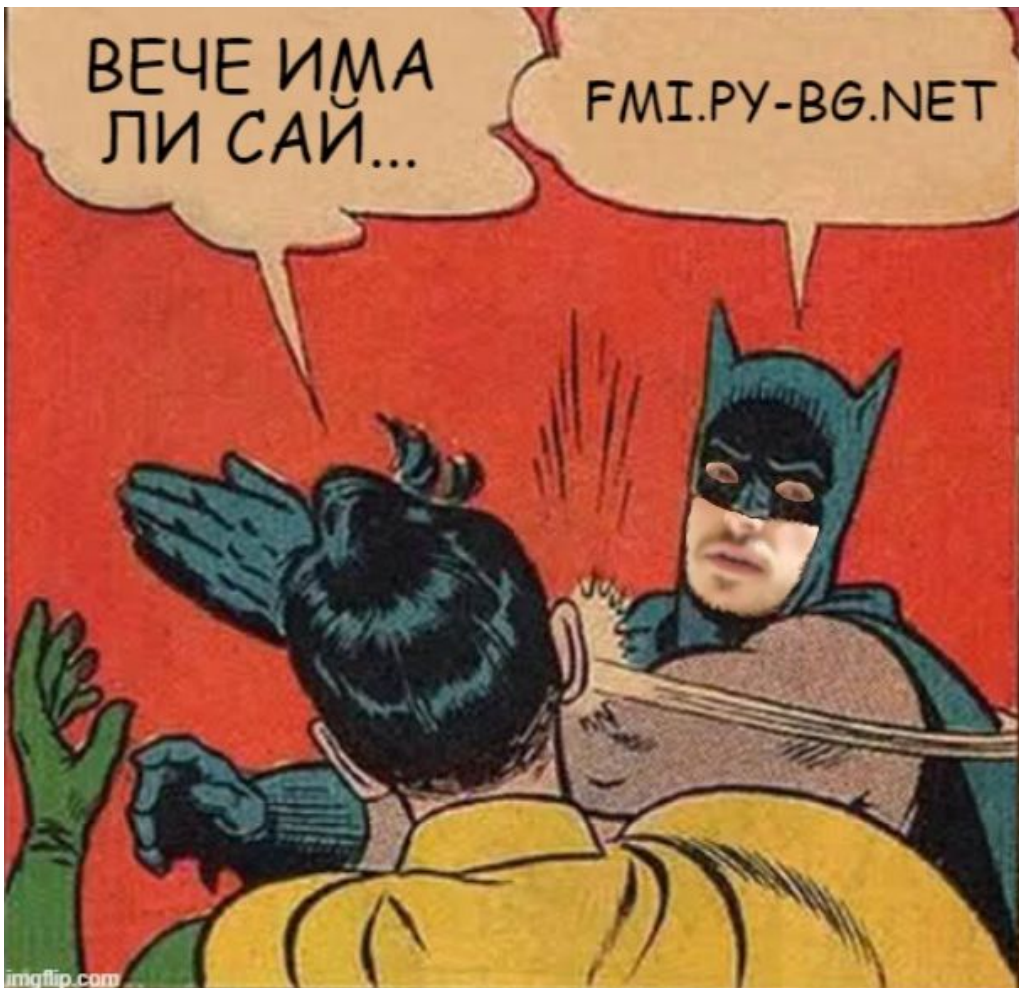
КОГИ СИ ЧАКАШ САЙТА ВЕЧЕ ТРЕТА СЕДМИЦА



imgflip.com



Сайт/форум



Какво видяхте в предишната лекция?

- Какво е колекция (нещо итерируемо)
- List (“указатели”, индексирани и слайсвани, основни методи)
- Tuple (immutable, unpacking, променлив брой стойности (*), сравнения)
- Set (уникални стойности, константно търсене, сечения на множества)
- Dict (ключ-стойност, константно търсене, (не)подреден, хеш функции)
- Голи снимки
- Който видял - видял

Какво не видяхте в предишната лекция?

- Функции, които генерират и/или използват колекции
- Загатване за генератори

Map/Filter/All/Any

- `map(function, iterable)` създава колекция от резултатите от прилагането на `function` върху всеки елемент от `iterable`
 - `filter(function, iterable)` създава колекция само с елементите, за които `function` върне `True`
 - `all(iterable)` всички елементи се оценяват на истина
 - `any(iterable)` поне един от елементите се оценява на истина
-
- за любознателните: `map` и `filter` са мързеливи
 - за още по-любознателните - `lambda`
 - демонстрация?

Comprehensions

- Изрази, които *генерират* колекции
- Елегантен заместител на `map` и/или `filter`
- Колекциите могат да са динамични

List comprehension

[израз **for** променлива **in** поредица **if** условие]

```
>>> [x * x for x in range(0, 10)]  
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

```
>>> [x * x for x in range(0, 10) if x % 2]  
[1, 9, 25, 49, 81]
```

List comprehension

Един list comprehension може да се вложи в друг, защото връща нещо итерируемо

```
>>> [(x, y) for x in [1, 2, 3] for y in [3, 1, 4] if x != y]
```

```
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

Generator expression

- Кръгли скоби вместо квадратни
- Като list comprehension, но се изпълнява динамично (lazy evaluation)
- На всяка стъпка итераторът оценя(ва) условието и израза за следващата стойност

Set comprehension

Като list comprehension, но с {}

```
>>> import math
>>> {int(math.sqrt(x)) for x in range(1,100)}
{1, 2, 3, 4, 5, 6, 7, 8, 9}
```

`import math` ?

- има време

Dict comprehension

```
>>> {i: chr(65 + i) for i in range(10)}
```

```
{0: 'A', 1: 'B', 2: 'C', 3: 'D', 4: 'E', 5: 'F', 6: 'G', 7: 'H', 8: 'I',  
9: 'J'}
```

Колекции за хора без компания в петък вечер

- deque - двупосочни опашки
- OrderedDict - речник, който помни реда
- defaultdict - речник със стойност по подразбиране
- Counter - речник, който брои повтарящи се стойности
- namedtuple - кортеж с именовани полета

03. Декоратори

— Да се върнем на темата —

За да разберем какво са декоратори, трябва да знаем...

- Какво е функция
- Какво е област на видимост
- Какво са вложени функции (е, то е очевидно)
- Какво значи, че функциите са първокласни обекти
- Какво е closure

Едно по едно...

Предговор

- Какво различава функцията от повечето други обекти?
 - `__call__`
- Какви типове обекти може да връща една функция?
 - Всякакви.

Области на видимост

- Всяка променлива (име) може да бъде свързана със стойност (binding)
- Има операции, които променят свързването, например =

```
global_one = 1
```

```
def foo():  
    local_one = 2  
    print(locals())
```

```
print(globals())    # {..., 'global_one': 1}  
foo()               # {'local_one': 2}
```

locals/globals

Вградени функции

- `locals` – връща речник с всички имена в локалната област на видимост
- `globals` – връща речник с всички имена в глобалната област на видимост

Области на видимост**2

- Всеки блок от код (напр. функция, модул, дефиниция на клас) си има своя област на видимост, в която стоят локално дефинираните променливи
- Ако една функция не може да намери дадена променлива в локалния си скоуп, търси в обграждащия (глобалния) за променлива със същото име

```
global_one = 1
```

```
def foo():  
    print(global_one)
```

```
foo()
```

Области на видимост**2**2

А какво ще изведе следният код?

```
global_one = 1
```

```
def foo():  
    global_one = 2  
    print(global_one)  
    print(locals())
```

```
foo()  
print(globals())
```

- По подразбиране пренасочването на имена става в локалния скоуп
- Използването на ключовата дума `global` позволява пренасочването на глобални имена
- **НЕ ИСКАТЕ ДА ПОЛЗВАТЕ `global`**

Животът на една променлива

- Една променлива "умира" заедно със своя скоуп...
- но ние не страдаме, защото такъв е живота на кода
- Амин!

Аргументи

- Можем да ги подаваме като позиционни или като именовани
- След именуван аргумент не можем да подадем позиционен
- Подадените аргументи отиват в locals
- Очевидно умират с приключването на функцията си

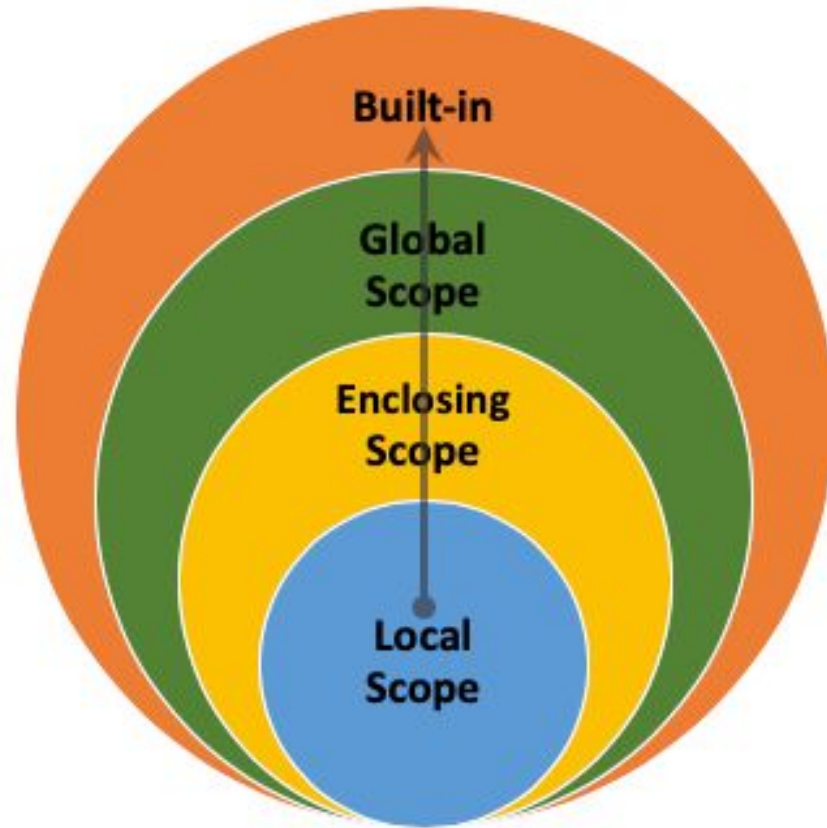
Вложени функции

- Можем да дефинираме функция тялото на друга функция
- Друга тема е кога това е добра идея
- Какво се случва тогава с променливите на двете функции и къде отиват?

```
def outer(x):  
    print(x)  
    def inner():  
        x = 0  
        print(x)  
    inner()  
    print(x)
```

- Името `inner` също отива в `locals()` на `outer`
- Ключовата дума `nonlocal` позволява пренасочване на име, дефинирано в обграждащ блок
- Познайте на какво мнение сме за ползването на `nonlocal`
- Още информация: [Naming and binding](#)

Local/Globals



Функциите са първокласни обекти

- Те са като всички останали обекти
- Можем да ги подаваме като аргументи
- Можем да ги връщаме като резултат
- Можем да ги записваме в колекции
- Можем да ги присвояваме на променлива
- Имат идентитет - `id()`

Closures

Имаме closure, когато вложена функция достъпва променлива, дефинирана в обграждаща функция






```
def start(x):  
    def increment(y):  
        return x + y  
    return increment
```

```
first_inc = start(0)  
second_inc = start(8)
```

```
first_inc(3)  
second_inc(3)
```

```
first_inc(1)  
second_inc(2)
```

[Edit] За да разберем какво са декоратори, трябва да знаем...

- Какво е функция 
- Какво е област на видимост 
- Какво са вложени функции (е, то е очевидно) 
- Какво значи, че функциите са първокласни обекти 
- Какво е closure 
- **+ Сериозен проблем**

Един сериозен проблем

- Занимаваме се с известен ресторант
- В него може да се поръчва храна със следните функции
- Игнорирайте това, че функциите ще се държат странно с $n \leq 1$
- Ще трябва да ни повярвате, че знаем как да се справим с проблема

```
def spam(n):  
    spams = ("spam", ) * (n - 1)  
    return "I would like {} and spam".format(", ".join(spams))
```

```
def eggs(n):  
    return "I would like {} eggs".format(n)
```

Обноси

- Внезапно се сещаме, че преувеличените обноси са хубаво нещо
- Искаме след поръчката да кажем "dear sir" или "dear madam" в зависимост от пола на обслужващия ни този ден

Начин 1

```
def spam(n, server):  
    spams = ("spam", ) * (n - 1)  
    return "I would like {} and spam, dear {}".format(", ".join(spams), server)  
  
def eggs(n, server):  
    return "I would like {} eggs, dear {}".format(n, server)  
  
spam(3, "sir")
```

- Easy! Caveman approach. Добавяме втори аргумент
- Да, но сега всеки път, когато си поръчваме нещо, ще трябва да се сещаме какъв беше полът на сервитьора. Би било хубаво ако можеше някакси само веднъж да се занимаваме с това.
- Ако ресторантът ни имаше 100 различни неща за поръчване? Ако искаме да сменим формата от "dear madam" на нещо друго?

До преди малко говорихме за функции

```
def spam(n):  
    spams = ("spam",) * (n - 1)  
    return "I would like {} and spam".format(", ".join(spams))
```

```
def eggs(n):  
    return "I would like {} eggs".format(n)
```

```
def served_by(func, server):  
    def cached_server(n):  
        return "{}, dear {}".format(func(n), server)  
    return cached_server
```

```
eggs = served_by(eggs, "sir")  
spam = served_by(spam, "sir")
```


Да благодарим

Когато поръчваме яйца, винаги да благодарим

```
def thank_you(func):  
    def with_thanks(n):  
        return "{}. Thank you very much!".format(func(n))  
    return with_thanks
```

```
eggs = thank_you(served_by(eggs, "sir"))  
spam = served_by(spam, "sir")
```

Започна да става сложно

- Доста се натовари създаването на нашите функции
- Има по - добър начин. Ще го покажем след малко
- Но преди това един друг пример

Фибоначи

```
def fibonacci(x):  
    if x in [0, 1]:  
        return 1  
    return fibonacci(x - 1) + fibonacci(x - 2)
```

Рекурсивната версия на fibonacci, освен че е бавна, е много бавна. Особено усезаемо, когато $x \geq 40$.

Проблемът е, че fibonacci се извиква стотици пъти с един и същ аргумент. Можем спокойно да прегенерираме първите стотина резултата в един речник или...

Да изчисляваме всеки резултат само по веднъж...

```
if x not in memory:  
    memory[x] = fibonacci(x)
```

```
print(memory[x])
```

Разбира се, тази идея може да се използва и на много повече места! Можем да я направим още по-елегантно.

Функции, които опаковат други функции

- $f(\text{функция}) \rightarrow \text{функция}$
- резултатът е нова функция, която "опакова" старата и може да разшири нейната функционалност

memoize

```
def memoize(func):  
    memory = {}  
    def memoized(*args):  
        if args in memory:  
            return memory[args]  
        result = func(*args)  
        memory[args] = result  
        return result  
    return memoized
```

```
fibonacci = memoize(fibonacci)
```

Красивият синтаксис

```
def fibonacci(x):  
    if x in [0,1]:  
        return 1  
    return fibonacci(x-1) + fibonacci(x-2)
```

```
fibonacci = memoize(fibonacci)
```

Декорацията става след дефиницията на функцията.

```
@memoize  
def fibonacci(x):  
    if x in [0, 1]:  
        return 1  
    return fibonacci(x - 1) + fibonacci(x - 2)
```

Друг пример за декоратор

```
def notifyme(f):  
    def logged(*args, **kwargs):  
        print(f.__name__, ' called with', args, 'and', kwargs)  
        return f(*args, **kwargs)  
    return logged
```

```
@notifyme
```

```
def square(x):  
    return x * x
```

```
res = square(25) # 625
```

```
# square was called with (25,) and {}.
```


Яйца?

```
def served_by(server):
    def decorator(func):
        def cached_server(n):
            return "{} , dear {}".format(func(n), server)
        return cached_server
    return decorator

def thank_you(func):
    def with_thanks(n):
        return "{} . Thank you very much!".format(func(n))
    return with_thanks

@served_by("sir")
def spam(n):
    spams = ("spam", ) * (n - 1)
    return "I would like {} and spam".format(", ".join(spams))

@thank_you
@served_by("sir")
def eggs(n):
    return "I would like {} eggs".format(n)
```

Има сайт, now what?

- Регистрирайте се
- Сложете си снимка (1 точка)
- Качете си Python и качете скрийншот във форума - има пост, там има инструкции (1 точка)
- Който има Snickers - вече може да въведе кода (1 точка)
- Имате първо домашно
- Домашното е със срок 1 седмица (без 4 часа)... **Постарайте се да предадете в срок, решения след крайния срок няма как да приемем**
- Ако имате въпроси по домашните - ще има тема във форума
- **PER8**... Сериозно, казахме, че държим на качеството на кода
- Ако имате въпроси по каквото и да е друго - можете да правите теми във форума

Въпроси?