
04. Декоратори

— 17 октомври 2024 —

По темата с домашните



A man in a dark tuxedo and white bow tie sits behind a dark wooden desk on a beach. The desk is positioned on a bed of dark pebbles. In the background, the ocean waves are breaking onto the shore. On the desk, there is a vintage-style microphone and a typewriter. The scene is lit with a warm, golden light, suggesting late afternoon or early morning. A semi-transparent dark horizontal band is overlaid across the middle of the image, containing white text.

But first
~~And now~~ for something completely different.”

Monty Python

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 итератора оценява условието и израза за следващата стойност

Set comprehension

Както 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}
```


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'}
```

Какво (не) видяхте в предишната лекция?

- Контролни структури - `if`, `while`, `for`, `switch` (`break`, `continue`)
- 4 интервала индентация и `“:”` за начало на блока
- Дефиниране на функции и аргументи на функции
- Функциите (и не само) са обекти и могат да бъдат подавани като аргументи на други функции (`map`, `filter...`)
- Анонимни функции (`lambda`)
- Голи снимки
- Който видял - видял

Въпроси?



Коя колекция може да бъде итерирана с for?

A: Dict

B: Set

C: List

D: Tuple

Побългарен код - какво ще изведе?

```
value = '_|_'  
кажи = print  
Вальо = value  
тръбата = '|'  
дължината = len
```

```
if тръбата in Вальо:                # Ако тръбата е във Вальо  
    if дължината(тръбата) > 20:    # Ако тръбата е по-дълга от 20  
        _ = Вальо, кажи('0оох!')  # Кажи, Вальо, 0оох!  
    else:                            # Инак  
        _ = Вальо, кажи('Ееех!')  # Кажи, Вальо, Ееех!  
    _ = Вальо, кажи('Въх!')        # Кажи, Вальо, Въх!
```

```
>>> Ееех!
```

```
>>> Въх!
```

Поангличен код

Какво ще изведе следната любовна история?

```
tool = slowly = ready = bored = something = 0
you = action = []
she = she_moans_like_a = {0}

for play in action:
    you.extend([tool])

while she is not ready:
    while she is not bored:
        you.insert(tool, slowly)
        you.remove(tool)
        if she.discard(something):
            she is ready
        else:
            (she_moans_like_a := (she := 0-0-0))
        if she is bored:
            break
        you.reverse()
    if she is ready:
        you.clear()

print(she is ready)
```

Foreplay in action.
You extend tool.

While she is not ready, while she is not bored, you insert tool slowly, you remove tool.

If she discard(s) something, she is ready.
Else, she moans like a walrus.
She-walrus: 0-0-0.

If she is bored, break - you reverse.


If she is ready, you clear.

За десерт - една скучна функция

```
def boring_function(boring_argument, *args, **kwargs):  
    return len(args) + len(kwargs)
```

```
print(boring_function() + boring_function(0) + boring_function(0, 0))
```

```
# TypeError: boring_function() missing 1 required positional  
argument: 'boring_argument'
```

A man in a dark tuxedo and white bow tie sits behind a dark wood desk on a beach. The desk is positioned on a bed of dark pebbles. On the desk, there is a typewriter and a vintage microphone. The background shows the ocean with waves breaking onto the shore. A semi-transparent dark grey horizontal band is overlaid across the middle of the image, containing white text.

“And now for something completely different.”

Monty Python

04. Декоратори

— 17 октомври 2024 —

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)
```

Обноски

- Внезапно се сецщаме, че преувеличените обноски са хубаво нещо
- Искаме след поръчката да кажем "dear sir" или "dear madam" в зависимост от пола на обслужващия ни този ден
- No offense, LGBT, но всяка жаба да си знае гьола

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?

Трябва да има и по-добър начин...

- Но първо трябва да обсъдим:
 - 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...

Какво е функция

В python функциите са обекти!

```
def baba():  
    print('баница')  
  
def call(function, times):  
    for _ in range(times):  
        function()
```

```
call(baba, 4)  
# баница  
# баница  
# баница  
# баница
```

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`**

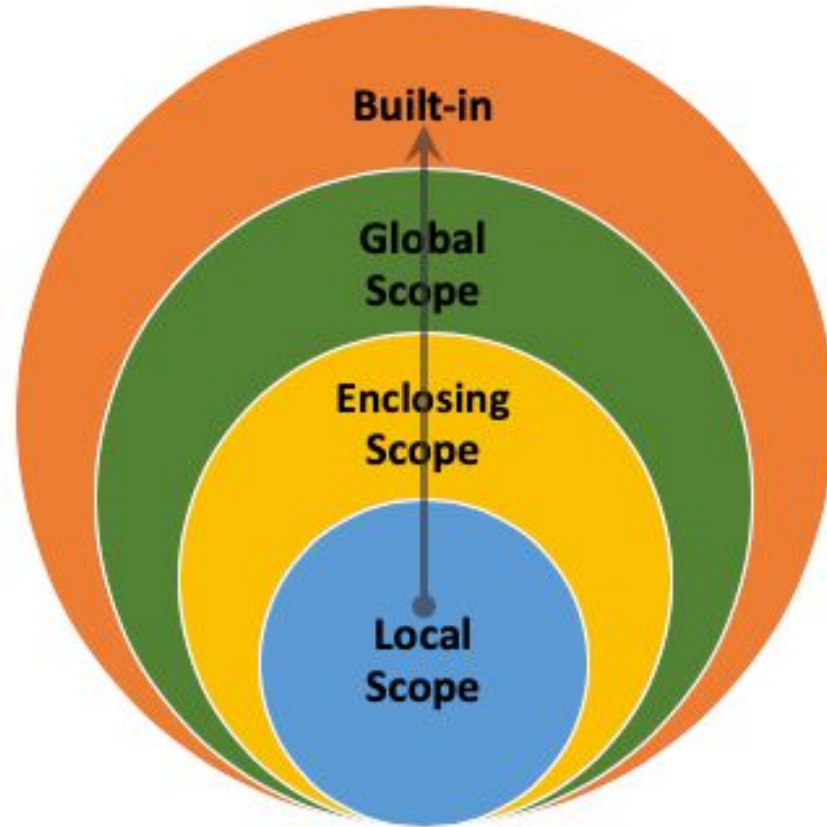
Животът на една променлива

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

Пре(д)говор - аргументи

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

Local/Globals



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`

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)  # 3  
second_inc(3) # 11
```

```
first_inc(1)  # 1  
second_inc(2) # 10
```

Да си дойдем на думата

```
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")
```

Без да се усетите ви направихме сефтето

- Току що написахме първият си декоратор заедно
- Декоратор е функция, която приема функция и връща функция
- `f(g) -> g'`
- Идеята е, че декораторът `f` "опакова" функцията `g` и разшири нейната функционалност, давайки ни като резултат `g'`

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

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

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

Да благодарим

А ако искаме 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!

Решението

Нека за момент игнорираме `served_by` и да се фокусираме само върху `thank_you`

```
def thank_you(func):  
    def with_thanks(n):  
        return "{}. Thank you very much!".format(func(n))  
    return with_thanks  
  
eggs = thank_you(eggs)  
eggs(10) # 'I would like 10 eggs. Thank you very much!'
```

Красивият синтаксис

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

```
eggs = thank_you(eggs)
```

Е същото като...

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



Sugar Bae

Дотук добре

Обаче `served_by` дефинира (и съответно очаква) параметър `server`

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

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

Колко би било яко да можем да направим следното:

```
@served_by("sir")  
def spam(n):  
    ...
```


Е, можем!

За целта трябва да направим малка промяна:

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

```
@served_by("sir")
```

```
def spam(n):
```

```
    ...
```

Откройте различия

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

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

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

```
@served_by("sir")  
def spam(n):  
    ...
```

Открийте разликите

```
def served_by(func, server):
```

```
:
```

```
    return cached_server
```

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

```
def served_by(server):  
    def decorator(func):
```

```
        return cached_server  
    return decorator
```

```
@served_by("sir")  
def spam(n):  
    ...
```



Разяснение

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

```
@served_by("sir")  
def spam(n):  
    ...
```

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

```
served_by_sir = served_by("sir")
```

```
@served_by_sir  
def spam(n):  
    ...
```

- `served_by` не е *“баш”* декоратор, защото не приема функция като аргумент
- `served_by` е функция, която създава и връща декоратор (виж `served_by_sir`)
- Просто не е нужно да създаваме име, което да държи декоратора, а можем направо да извикаме функцията след символа @

Искате по-дълбоко?

```
def fun1(arg1):
    def fun2(arg2):
        def fun3(arg3):
            def decorator(func):
                def wrapper(*args, **kwargs):
                    decorated = func(*args, **kwargs)
                    decorated.extend([arg1, arg2, arg3])
                    return decorated
                return wrapper
            return decorator
        return fun3
    return fun2
```

```
@fun1("arg1")("arg2")("arg3")
```

```
def fun():
    return ['static value']
```

```
print(fun()) # ['static value', 'arg1', 'arg2', 'arg3']
```

- Можете да дефинирате произволен брой вложени функции на същия принцип

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)
```

Бонус декоратор

```
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.

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
```

```
@memoize  
def fibonacci(x):  
    ...
```

Бонус бонус декоратор

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

```
@notifiyme
```

```
def square(x):
```

```
    return x * x
```

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

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

Въпроси?