05 dekoratorji

January 28, 2024

0.1 Dekoratorji

Tale tema bo za mnoge naporna miselna vaja. Vendar koristna.

Najprej ponovimo nekaj, kar smo se naučili prejšnji teden.

0.1.1 Funkcija, ki vrača funkcijo

Tole je funkcija, ki predstavlja polinom $2x^2 + 4x + 3$.

[1]:
```
def poly243(x):
    return 2 * x ** 2 + 4 * x + 3
```

Trivialno. Pokličimo jo, samo za poskus.

[2]:
```
poly243(4)
```

[2]:
```
51
```

Zdaj pa bi rad naredil funkcijo, ki sestavlja takšne polinome. Torej: funkcija `create_poly(a, b, c)` bo kot argumente dobila koeficiente polinoma ter sestavila (in vrnila) polinom.

[3]:
```
def create_poly(a, b, c):
    def poly(x):
        return a * x ** 2 + b * x + c

    return poly
```

S to funkcijo lahko sestavimo gornjo funkcijo.

[5]:
```
poly243 = create_poly(2, 4, 3)
```

Če razmislimo, kaj se zgodi - očitno se zgodi praktično isto kot prej: ko pokličemo `create_poly(2, 4, 3)` se zgodi definicija funkcije `poly` (s koeficienti 2, 4, 3), ki prejme le en argument `x`. Funkcija `create_poly` vrne to funkcijo in to funkcijo priredimo imenu `poly243`.

[6]:
```
poly243(4)
```

[6]:
```
51
```

0.2 Pakiranje in razpakiranje argumentov

Ali: funkcije s poljubnim številom argumentov. In klicanje z neznanim številom argumentov.

Nadaljujmo kar s polinomom. Obstajajo tudi polinomi, ki niso druge stopnje. Obstajajo tudi takšni tretje, ali celo četrte. Naša funkcija `create_poly` pa dela le polinome druge stopnje. Posplošimo jo.

Najprej razmislimo, kako izračunati polinom poljubne stopnje, če imamo seznam (ali terko) koeficientov, recimo `coefs = [c3, c2, c1, c0]` in vrednost `x`. Napisali bomo funkcijo, ki kot argument dobi koeficiente in vrednost x - kar očitno ni isto kot funkcija, ki smo jo pisali zgoraj; v zgornjo so koeficienti že vdelani.

En način je:

[7]:
```
def poly_x(x, coefs):
    y = 0
    for i in range(len(coefs)):
        y += coefs[i] * x ** (len(coefs) - i - 1)
    return y
```

Upam, da sem vam tega že izbil iz glave, saj lahko uporabimo `enumerate`.

[8]:
```
def poly_x(x, coefs):
    y = 0
    for i, coef in enumerate(coefs):
        y += coef * x ** (len(coefs) - i - 1)
    return y
```

Tudi to je sicer dokaj neprivlačno; osnovni problem je, da so koeficienti našteti od višjih potenc proti nižjim, vendar je tako prav: tudi polinome vedno pišemo v takem vrstnem redu.

Uporabimo malo osnovnošolske matematike: $c_3x^3 + c_2x^2 + c_1x^1 + c_0x^0$ je isto kot $(((c_3x + c_2)x + c_1)x + c_0$. (Ta reč ima celo ime: Hornerjevo pravilo. Če boste imeli kdaj kakšen predmet v slogu numeričnih metod, vas bodo učili celo, da je ta oblika boljša, ker je numerično stabilnejša, to je, vodi v manjše zaokrožitvene napake.)

[9]:
```
def poly_x(x, coefs):
    y = 0
    for coef in coefs:
        y = x * y + coef
    return y

x = 4
poly_x(x, [2, 4, 3])
```

[9]:
```
51
```

Luštno bi bilo, če bi lahko podali številke kar kot argumente, na primer `poly_x(x, 2, 4, 3)`. Funkcija, ki prejme poljubno število argumentov, kot vemo od prejšnjega tedna, deklarira argument,

katerega ime se začne z * in ta argument bo terka z vsemi dodatnimi argumenti - torej našimi koeficienti.

[10]:
```
def poly_x(x, *coefs):
    y = 0
    for coef in coefs:
        y = x * y + coef
    return y

x = 4
poly_x(x, 2, 4, 3)
```

[10]: 51

Spremenilo se je samo to, da smo dodali zvezdico.

Ta funkcija prejme `x` in koeficiente. Radi bi takšno, ki prejme le `x`, koeficiente pa že ima. Torej, želeli bi `create_poly`, ki bo ustvarila `poly`.

[11]:
```
def create_poly(*coefs):
    def poly(x):
        y = 0
        for coef in coefs:
            y = x * y + coef
        return y

    return poly
```

[12]:
```
poly324 = create_poly(2, 4, 3)
poly324(4)
```

[12]: 51

0.3 Funkcija, ki ovije funkcijo

Recimo, da bi hoteli iz nekega neumnega razloga želeli, da `sqrt`, `sin` in `cos` vračajo zaokrožajo rezultate na tri decimalke. Lahko napišemo nove funkcije, ki povozijo stare.

[13]:
```
import math

def rounded_sqrt(x):
    y = math.sqrt(x)
    return round(y, 3)

def rounded_sin(x):
    y = math.sin(x)
    return round(y, 3)

def rounded_cos(x):
```

```
    y = math.cos(x)
    return round(y, 3)
```

[14]:
```
rounded_sqrt(5)
```

[14]:
```
2.236
```

Lahko pa napišemo funkcijo, ki sestavlja takšne funkcije. Prej smo napisali funkcijo, ki definira in vrne funkcijo, ki računa polinome. Zdaj pa bomo napisali funkcijo, ki definira funkcijo, ki pokliče neko funkcijo (`math.sqrt`, `math.sin`, `math.cos`…) in vrne njen rezultat zaokrožen na tri decimalke. Tako kot je bila `poly` prej funkcija znotraj `create_poly`, bo zdaj funkcija, kot so zgornje, znotraj funkcije … no, imenujmo jo `rounder`.

[15]:
```
def rounder(f):
    def rounded_f(x):  # funkcija, ekvivalentna gornjim rounded_sqrt,␣
 ↪rounded_sin...
        y = f(x)
        return round(y, 3)

    return rounded_f

rounded_sqrt = rounder(math.sqrt)
rounded_sin = rounder(math.sin)
rounded_cos = rounder(math.cos)
```

Seveda lahko na podoben način ovijamo tudi svoje funkcije.

[16]:
```
from math import pi

def circumf(r):
    return 2 * pi * r

rounded_circumf = rounder(circumf)
```

`rounder` je imenitna in gotovo izjemno uporabna funkcija (no, to pač ne), še imenitnejša pa bo, če ne bo pričakovala, da ovita funkcija (`rounded_f`) vedno prejme samo en argument. Takole jo popravimo.

[17]:
```
def rounder(f):
    def rounded_f(*args):  # funkcija, ekvivalentna gornjim rounded_sqrt,␣
 ↪rounded_sin...
        y = f(*args)
        return round(y, 3)

    return rounded_f
```

Podobno kot smo se učili zgoraj, tudi tu pridemo do poljubnega števila argumentov tako, da uporabimo zvezdico; za takšne argumente navadno uporabimo ime `args`. Zgoraj pa nismo ponovili

(pač pa smo prejšnji teden vseeno videli), kako pokličemo funkcijo, če imamo argumente v terki: spet tako, da damo pred terko zvezdico, torej `y = f(*args)`.

0.4 Vohun

Zdaj pa naredimo nekaj, kar bi bilo morda lahko celo uporabno: funkcijo, ki, podobno kot `rounder`, ovije funkcijo, vendar pusti rezultat pri miru. Pač pa ob vsakem klicu funkcije izpiše argumente in rezultat.

[18]:
```
def spied(f):
    def spied_f(*args):
        y = f(*args)
        print("Function", f.__name__, "called with", args, ", returned", y)
        return y

    return spied_f

def add(a, b):
    return a + b

add = spied(add)
```

[19]:
```
v = []
for x in range(5):
    v.append(add(42, x))
```

```
Function add called with (42, 0) , returned 42
Function add called with (42, 1) , returned 43
Function add called with (42, 2) , returned 44
Function add called with (42, 3) , returned 45
Function add called with (42, 4) , returned 46
```

Spet, na podoben način bi lahko ovili tudi Pythonove funkcije.

[20]:
```
sqrt = spied(math.sqrt)

v = []
for x in range(5):
    v.append(sqrt(x))
```

```
Function sqrt called with (0,) , returned 0.0
Function sqrt called with (1,) , returned 1.0
Function sqrt called with (2,) , returned 1.4142135623730951
Function sqrt called with (3,) , returned 1.7320508075688772
Function sqrt called with (4,) , returned 2.0
```

0.5 Dekoratorji

Takšnih funkcij bomo, za vajo, napisali še nekaj. Zdaj pa se naučimo drugačnega načina za njihovo rabo. Omogočil nam bo, da se izognemo temu.

[21]:
```
def add(a, b):
    return a + b

add = spied(add)

def sub(a, b):
    return a - b

sub = spied(sub)

def circumf(r):
    return 2 * pi * r

circumf = rounder(circumf)
```

Vzorec je povsod isti: imamo funkcijo, ki jo hočemo oviti v neko drugo funkcijo, tako da uporabimo funkcijo za ovijanje. `rounder` je funkcije, ki ovije `circumf` v neko lokalno funkcijo (ki kliče originalno). Ker je to še kar pogosta zadeva, obstaja krajši, preglednejši način: dekoratorji. Namesto zgornjega lahko pišemo:

[22]:
```
@spied
def add(a, b):
    return a + b

@spied
def sub(a, b):
    return a - b

@rounder
def circumf(r):
    return 2 * pi * r
```

Z

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

definiramo neko funkcijo `f`, jo podamo funkciji `decorator` in "pravi" `f` bo tisto, kar vrne `decorator`. Kot v gornjih primerih.

0.6 Dekorator, ki šteje klice funkcije

Napišimo dekorator, ki bo štel, kolikokrat je funkcija poklicana.

Najprej moramo izvedeti tole: Pythonove funkcije so objekti in imajo lahko tudi atribute.

[23]:
```
def f(x):
    return 2 * x
```

```
f.foo = 42

print(f.foo)
```

```
42
```

Funkcija bi lahko štela, kolikokrat smo jo poklicali.

[24]:
```
def f(x):
    f.called += 1
    return 2 * x

f.called = 0

for x in range(5):
    f(x)

print(f.called)
```

```
5
```

Vendar zdaj poznamo zabavnejši način: znamo napisati dekorator. Ta bo ovil funkcijo v funkcijo, ki šteje, kolikokrat je poklicana. Sicer pa bo le poklical in vrnil ovito funkcijo, ne da bi se vtikal v argumente ali rezultat.

[25]:
```
def count_calls(f):
    def wrapper(*args):
        y = f(*args)
        wrapper.called += 1
        return y

    wrapper.called = 0
    return wrapper

@count_calls
def add(x, y):
    return x + y

for x in range(5):
    add(x, x ** 2)

print(add.called)
```

```
5
```

Še bolj imenitne stvari lahko počnemo: lahko naredimo cel log klicev! Seznam parov (argumenti, rezultat) za vse klice funkcije!

[26]:
```
def logged(f):
    def wrapper(*args):
        y = f(*args)
        wrapper.log.append((args, y))
        return f(*args)

    wrapper.log = []
    return wrapper

@logged
def add(x, y):
    return x + y

for x in range(5):
    add(x, x ** 2)

print(add.log)
```

```
[((0, 0), 0), ((1, 1), 2), ((2, 4), 6), ((3, 9), 12), ((4, 16), 20)]
```

0.7 Funkcije s spominom

Čas je, da naredimo kaj uporabnega.

Tule imamo funkcijo, ki računa Fibonaccijeva števila. Napisana je rekurzivno; z njo lahko izračunamo prvih nekaj Fibonaccijevih števil, potem pa postane prepočasna. Ker znamo logirati klice, lahko ugotovimo tudi, zakaj.

[27]:
```
@logged
def fibo(n):
    if n < 2:
        return 1
    return fibo(n - 2) + fibo(n - 1)

fibo(5)

print(fibo.log)
```

```
[((1,), 1), ((0,), 1), ((1,), 1), ((2,), 2), ((0,), 1), ((1,), 1), ((3,), 3),
((1,), 1), ((0,), 1), ((1,), 1), ((2,), 2), ((0,), 1), ((1,), 1), ((0,), 1),
((1,), 1), ((2,), 2), ((0,), 1), ((1,), 1), ((1,), 1), ((0,), 1), ((1,), 1),
((2,), 2), ((0,), 1), ((1,), 1), ((3,), 3), ((1,), 1), ((0,), 1), ((1,), 1),
((2,), 2), ((0,), 1), ((1,), 1), ((4,), 5), ((0,), 1), ((1,), 1), ((2,), 2),
((0,), 1), ((1,), 1), ((1,), 1), ((0,), 1), ((1,), 1), ((2,), 2), ((0,), 1),
((1,), 1), ((3,), 3), ((1,), 1), ((0,), 1), ((1,), 1), ((2,), 2), ((0,), 1),
((1,), 1), ((5,), 8), ((1,), 1), ((0,), 1), ((1,), 1), ((2,), 2), ((0,), 1),
((1,), 1), ((3,), 3), ((1,), 1), ((0,), 1), ((1,), 1), ((2,), 2), ((0,), 1),
((1,), 1), ((0,), 1), ((1,), 1), ((2,), 2), ((0,), 1), ((1,), 1), ((1,), 1),
((0,), 1), ((1,), 1), ((2,), 2), ((0,), 1), ((1,), 1), ((3,), 3), ((1,), 1),
```

```
((0,), 1), ((1,), 1), ((2,), 2), ((0,), 1), ((1,), 1), ((4,), 5), ((0,), 1),
((1,), 1), ((2,), 2), ((0,), 1), ((1,), 1), ((1,), 1), ((0,), 1), ((1,), 1),
((2,), 2), ((0,), 1), ((1,), 1), ((3,), 3), ((1,), 1), ((0,), 1), ((1,), 1),
((2,), 2), ((0,), 1), ((1,), 1)]
```

[28]:
```
len(fibo.log)
```

[28]:
```
101
```

Da izračuna peto Fibonaccijevo število, funkcija 101-krat pokliče samo sebe. Za šesto se pokliče že 277-krat.

[29]:
```
fibo.log = []
fibo(6)
len(fibo.log)
```

[29]:
```
277
```

Problem je v tem, da za izračun petega potrebuje tretje in četrto. Za izračuna četrtega potrebuje drugo in tretje – torej bo dvakrat računalo tretje Fibonaccijevo število. Za izračun tretjega potrebuje drugo in prvo (vsakega po dvakrat - poleg tega pa bo računala še enkrat, namreč takrat, ko bo računala tretjega) … Skratka, teh klicev je več in več.

Fibonaccijeva števila je seveda možno računati učinkoviteje, namreč naprej in ne nazaj. A to ni bistvo. Bistvo je, da imamo počasno funkcijo, ki bi se jo dalo pospešiti tako, da stvari, ki jih je že enkrat izračunala, ne bi računala ponovno. Recimo tako, da bi imela slovar, katerega ključi bi bili pretekli argumenti, vrednosti pa rezultat pri teh argumentih. Ob vsakem klicu bi preverila, če je bila s temi argumenti že klicana. Če, potem le vrne že izračunani rezultat. Če ne, računa in shrani.

[30]:
```
cache = {}

@logged
def fibo(n):
    # Če rezultat za te argumente še ni v slovarju, ga izračuamo in dodamo v␣
 ↪slovar
    if n not in cache:
        if n < 2:
            cache[n] = 1
        else:
            cache[n] = fibo(n - 2) + fibo(n - 1)

    return cache[n]

fibo(5)

print(fibo.log)
```

```
[((1,), 1), ((0,), 1), ((1,), 1), ((2,), 2), ((3,), 3), ((2,), 2), ((3,), 3),
((4,), 5), ((5,), 8)]
```

Tole je seveda samo za demo. Uporablja globalne spremenljivke in to se ne dela.

Vendar ni problema. To bomo itak posplošili: naredili bomo dekorator, ki ovije funkcijo v funkcijo, ki shranjuje rezultate ovite funkcije. In ovito funkcijo kliče le, če in kadar je to potrebno.

[31]:
```
def cached(f):
    cache = {}
    def wrapped(x):
        if not x in cache:
            cache[x] = f(x)
        return cache[x]

    return wrapped

@logged
@cached
def fibo(n):
    if n < 2:
        return 1
    return fibo(n - 2) + fibo(n - 1)

fibo(5)
print(fibo.log)
```

```
[((1,), 1), ((0,), 1), ((1,), 1), ((2,), 2), ((3,), 3), ((2,), 2), ((3,), 3),
((4,), 5), ((5,), 8)]
```

To je potrebno le še posplošiti tako, da deluje s funkcijami s poljubnim številom argumentov. A ne bomo. Vemo, samo `wrapped(x)` zamenjamo z `wrapped(*x)` in `f(x)` z `f(*x)`. A vseeno ne bomo. Zato ker so takšen dekorator že naredili namesto nas. Imenuje se `lru_cache` in je v modulu `functools`. Dekorator sprejme tudi argument: povedati mu je potrebno, koliko zadnjih klicev naj si zapomni.

[32]:
```
from functools import lru_cache

@logged
@lru_cache(10)
def fibo(n):
    if n < 2:
        return 1
    return fibo(n - 2) + fibo(n - 1)

fibo(5)
print(fibo.log)
```

```
[((1,), 1), ((0,), 1), ((1,), 1), ((2,), 2), ((3,), 3), ((2,), 2), ((3,), 3),
((4,), 5), ((5,), 8)]
```

Kako napisati dekorator, ki sprejema tudi argumente? Poleg funkcije?

No, `lru_cache` v resnici ni dekorator, temveč funkcija, ki vrne dekorator. Če je dekorator funkcija, ki vrača funkcijo, je `lru_cache` parametriziran dekorator, torej funkcija, ki vrača funkcijo, ki vrača funkcijo. Ni tako kompliciran, vendar smo danes že dovolj zvijali možgane.

0.8 Dekoratorji, ki prčkajo po argumentih

Napisali smo par dekoratorjev, ki se ukvarjajo z rezultati funkcije, še nobenega pa v zvezi z argumenti. Storimo še to.

Recimo, da bi nek modrijan prišel na modro idejo, da bi bilo fino, če bi funkcije, ki sicer sprejemajo samo `float`-e, sprejele tudi nize - seveda takšne nize, ki se lahko pretvorijo v števila.

[33]:
```
from math import sqrt

def to_float(f):
    def wrapped_f(*args):
        new_args = []
        for arg in args:
            new_args.append(float(arg))
        return f(*new_args)
    return wrapped_f

sqrt = to_float(sqrt)

sqrt("25")
```

[33]:
```
5.0
```

Tule smo "popravili" že vdelano funkcijo `sqrt`. Vsak dekorator je možno uporabiti tudi na ta način. Običajno pa jih uporabljamo za dekoriranje lastnih funkcij, torej tudi tu poskusimo še to.

[34]:
```
@to_float
def circumf(r):
    return 2 * pi * r

circumf("1")
```

[34]:
```
6.283185307179586
```

0.9 "Overloadanje" funkcij

V nekaterih jezikih je mogoče napisati več različic funkcije z istim imenom, vendar različnimi tipi (ali številom) argumentov. Ob klicu funkcije se prevajalnik na osnovi argumentov odloči, katero funkcijo bo poklical.

V Pythonu to ne gre, saj v definiciji funkcije ne deklariramo tipov argumentov. No, lahko, vendar jih Python ignorira in jih, kot prisegajo avtorji, tudi vedno bo.

Vseeno pa obstaja dekorator, s katerim lahko dosežemo nekaj takšnega: `singledispatch`.

[35]:
```
from functools import singledispatch

help(singledispatch)
```

```
Help on function singledispatch in module functools:

singledispatch(func)
    Single-dispatch generic function decorator.

    Transforms a function into a generic function, which can have different
    behaviours depending upon the type of its first argument. The decorated
    function acts as the default implementation, and additional
    implementations can be registered using the register() attribute of the
    generic function.
```

Uporabljamo jo tako.

[36]:
```
from functools import singledispatch

@singledispatch
def add(x, y):
    return x, y

@add.register(str)
def _(x, y):
    return x + " " + y

@add.register(list)
def _(x, y):
    r = []
    for e, f in zip(x, y):
        r.append(e + f)
    return r
```

Osnovna različica preprosto sešteva.

[37]:
```
add(5, 6)
```

[37]:
```
(5, 6)
```

Če kot argument podamo niz, bo mednju vtaknila presledek.

[38]:
```
add("Ana", "Berta")
```

[38]:
```
'Ana Berta'
```

Če podamo seznama, pa ju bo seštela po elementih.

[39]:
```
add([1, 5, 4], [2, -3, 8])
```

[39]:
```
[3, 2, 12]
```

Kako uporabimo `singledispatch`, vidimo v gornjem primeru. Osnovno različico dekoriramo s `singledispatch`, nadaljnje pa z `<ime-osnovne-funkcije>.register`. Ime nadaljnjih *ne sme biti* enako osnovni, temveč mora biti kaj neumnega, po možnosti `_`.

Osebno mislim, da je to, da nadaljnje funkcije ne morejo imeti normalnih imen, grdo. In mi smo tu zato, da naredimo boljše.

0.10 Domač dekorator za overload

Tole, kar sledi, je *way beyond*, ne, to je *way way way beyond* prvi letnik. Take stvari delajo na enem težjih predmetov magistrskega študija. Kdor prebere in razume, naj bo kar ponosen nase.

[40]:
```
def overloadable(base_func):
    overloads = {}

    def func(*args):
        tpe = type(args[0])
        return overloads.get(tpe, base_func)(*args)

    def overload(tpe):
        def overloaded(over_f):
            overloads[tpe] = over_f
            return func
        return overloaded

    func.overload = overload
    return func
```

Najprej se prepričajmo, da deluje.

[41]:
```
@overloadable
def add(x, y):
    return x + y

@add.overload(str)
def add(x, y):
    return x + " " + y

@add.overload(list)
def add(x, y):
    r = []
    for e, f in zip(x, y):
        r.append(e + f)
    return r
```

```
[42]: add(5, 6)
```

```
[42]: 11
```

```
[43]: add("Ana", "Berta")
```

```
[43]: 'Ana Berta'
```

```
[44]: add([5, 4, 2], [1, -2, 3])
```

```
[44]: [6, 2, 5]
```

Zdaj pa še, zakaj deluje.

```
def overloadable(base_func):
    overloads = {}

    def func(*args):
        tpe = type(args[0])
        return overloads.get(tpe, base_func)(*args)

    def overload(tpe):
        def overloaded(over_f):
            overloads[tpe] = over_f
            return func
        return overloaded

    func.overload = overload
    return func
```

Slovar `overloads` bo vseboval vse različice funkcije (razen osnovne). Ključi slovarja bodo tipi, pripadajoče vrednosti pa funkcije, ki jih je potrebno poklicati za posamezen tip.

Naš dekorator `overloadable` bo zamenjal podano funkcijo (osnovno, tisto, ki jo bomo kasneje "overloadali") s funkcijo `func`. Funkcija `func` pogleda tip prvega argumenta (da, gledamo le prvi argument, a tudi `singledispatch` počne isto!). V slovarju poišče pripadajočo funkcijo, vendar ne uporablja običajnega indeksiranja (`overloads[tpe]`) temveč `get`, ki mu poda privzeto vrednost. Privzeta vrednost pa je kar `base_func`. To funkcijo potem pokliče s podanimi argumenti.

Poleg tega pa v dekoratorju definiramo funkcijo `overload`, ki jo pripnemo funkciji, ki jo bomo vrnili. Ta skrbi za to, da bomo lahko kasneje izvedli

```
@add.overload(str)
def add(x, y):
    return x + " " + y
```

Funkcijo `overload` bomo pripeli kot atribut k funkciji `func`, ki jo vračamo. Ta, ki uporablja naš dekorator, bo funkcijo `overload` torej videl pod imenom `add.overload`. Ta funkcija, `add.overload` kot argument prejme tip, v gornjem primeru `str`. Kaj pa vrne? Dekorator! Imamo namreč `@add.overload(str)`, torej pričakujemo, da bo rezultat klica `add.overload(str)` dekorator.

Funkcija `overloaded` (v gornji kodi) je torej dekorator, ki prejme novo različico funkcije (argument `over_f`). V slovar `overloads` pod ključ `tpe` zabeleži tole funkcijo. Vrne pa dekorirano osnovno funkcijo, `func`!

To, slednje, je tisto, po čemer se Pythonov `singledispatch` razlikuje od našega (poleg tega, da Pythonov omogoča tudi uporabo za registriranje že napisanih funkcij, ne le dekoriranja, in da je nekoliko hitrejši). Pythonov register namreč vrne `func`, ki pa se nanaša na novo funkcijo, ne staro.