

Več funkcij z istim imenom in različnimi argumenti

Študente pogosto zanima, ali je mogoče v Pythonu napisati več različic iste funkcije; katera od njih se pokliče, bi bilo odvisno od tipa argumenta. Eno funkcijo, torej, za argumente tipa `int` in drugo za sezname, ter mogoče še tretjo za množice. Inspiracija izvira iz jezikov v slogu C++, ki poznajo function overloading in funkcije

```
int f(int x) {  
    return x * 2;  
}
```

```
int f(float x) {  
    return x * 3;  
}
```

```
int f(int x, int y) {  
    return x * y;  
}
```

pa bo klic `f(42)` vrnil 84, klic `f(3.14)` pa 9.42, če se prav spomnim C++-a. In klic `f(2, 3)` bo vrnil 6.

Je v Pythonu torej to mogoče?

Ne. In da.

Python - kot jezik - tega nima. Pač pa je možno delati čudeže z dekoratorji.

Dekorator `singledispatch`

```
from functools import singledispatch
```

```
@singledispatch  
def f(x):  
    print(f"Takega argumenta, {x}, nihče ne mara.")
```

```
@f.register  
def _(x: int):  
    print("0, glej no, int")
```

```
@f.register  
def _(a: float, b: str):  
    print("Česa ne poveste, float smo dobili!")
```

```
@f.register  
def _(s: set):  
    print("Množica? Tudi to obstaja?")
```

```
f(42)
0, glej no, int
f(3.14, "test")
Česa ne poveste, float smo dobili!
f({1, 2, 3})
Množica? Tudi to obstaja?
f([1, 2, 3])
```

Takega argumenta, [1, 2, 3], nihče ne mara.

Reč ima omejitve. O tem, katera funkcija se bo poklicala, odloča le prvi element. Tip pa mora biti razred; napisali smo `set`, `set[int]` pa ne deluje.

Razvajencu iz C++ to seveda ni dovolj. V praksi pa tega ne potrebujemo ravno velikokrat (vsaj jaz to rekdo uporabljam - pa ne zato, ker mi ne bi bilo blizu, v C++ sem imel to rad!) in omejitev na en sam argument sploh ni tako huda.

To je vse. Naprej naj berejo tisti, ki jih zanima, kako je to narejeno.

Parametrizirani dekoratorji

Bolj zaradi tega, kar sledi v naslednjem razdelku, povejmo, da gre tudi brez označevanje tipov argumentov. V tem primeru je potrebno tip podati dekoratorju `register`.

```
from functools import singledispatch

@singledispatch
def f(x):
    print(f"Takega argumenta, {x}, nihče ne mara.")

@f.register(int)
def _(x):
    print("0, glej no, int")

@f.register(float)
def _(a, b):
    print("Česa ne poveste, float smo dobili!")

@f.register(set)
def _(s):
    print("Množica? Tudi to obstaja?")
```

Kako je to narejeno

Napisal sem, da tega ne omogoča Python kot jezik, temveč je to možno narediti z dekoratorjem. Z drugimi besedami: če tega ne bi bilo, bi lahko to naredili sami. Ker je preprosteje (kar še ne pomeni, da je preprosto), pokažimo, kako bi sami naredili to, drugo, parametrizirano različico dekoratorja.

```
def moj_dispatch(f):
    funkcije = {}
    def g(prvi, *ostali):
        return funkcije.get(type(prvi), f)(prvi, *ostali)

    def register(tip):
        def registriraj(f):
            funkcije[tip] = f
        return registriraj

    g.register = register
    return g
```

Za začetek razlage se spomnimo, kako bomo uporabili dekorator:

```
@moj_dispatch
def f(x):
    print(f"Takega argumenta, {x}, nihče ne mara.")
```

`moj_dispatch` je dekorator, ki pripravi slovar, v katerega bo shranjeval registrirane funkcije (ključ je tip, vrednost je funkcija, ki prejme argument tega tipa). Nato sestavi funkcijo `g`, ki jo bo kasneje tudi vrnil - funkcija `f` (iz primera uporabe) bo v resnici ta `g`. Funkcija `g` bo prejela nek argument (enega ali več - ostali so v `*ostali`). V slovarju poišče funkcijo, ki ustreza tipu `x`; če take funkcije ni, kot privzeto vrednost uporabi podani `f`; zato torej `funkcije.get(type(prvi), f)`. To funkcijo pokliče z argumenti `x` in `*ostali`-mi ter vrne njen rezultat.

Potem pa se spomnimo, kako bomo registrirali nove funkcije:

```
@f.register(int)
def _(x):
    print("0, glej no, int")
```

Da bo to delovalo, mora imeti `f` metodo, atribut ali kakorkoli hočemo temu reči `register`. Na srečo lahko Pythonovim funkcijam prirejamo attribute; za `f.register` poskrbimo z `g.register = register`. Seveda je potrebno `register` prej definirati.

Na prvi pogled bo `f.register` dekorator. To je zmota. `f.register` *pokličemo* s tipom. `f.register` torej ni dekorator, pač pa mora `f.register` *vrniti* dekorator. Drži? Razumemo? V gornji kodi piše `@f.register(int)`, torej je `f.register(int)` dekorator. `register` mora vrniti dekorator, z drugimi besedami, `f.register` mora vrniti funkcijo, ki prejme neko funkcijo (v našem

primeru funkcijo `_`) in jo "dekorira". Kako jo dekorira? Pravzaprav ... zelo slabo. Vse kar naredi, je, da jo shrani v slovar. `register(int)` je tako zanič dekorator, da sploh ničesar ne vrne! Po tem "dekoriranju" `_` sploh ni funkcija temveč `None`. Vendar nas to ne moti, saj je ne bomo nikoli direktno klicali.

To je vse.

```
@moj_dispatch
def f(x):
    print(f"Takega argumenta, {x}, nihče ne mara.")

@f.register(int)
def _(x):
    print("0, glej no, int")

@f.register(float)
def _(a, b):
    print("Česa ne poveste, float smo dobili!")

@f.register(set)
def _(s):
    print("Množica? Tudi to obstaja?")

f(42)

0, glej no, int

f(3.14, "test")

Česa ne poveste, float smo dobili!

f({1, 2, 3})

Množica? Tudi to obstaja?
```

V resnici je `singledispatch` seveda malo bolj zapleten. Med drugim, recimo, dekorator `register` vrača funkcijo,

```
def register(tip):
    def registriraj(f):
        funkcije[tip] = f
        return f
    return registriraj
```

kar omogoča, da isto funkcijo uporabimo za več tipov,

```
@f.register(int)
@f.register(float)
def _(x):
    print("Prejeli smo int ali float")
```

Še več komplikacij povzroči to, da lahko uporabljamo `f.register` s parametrom (npr. `int`) ali brez.

A osnovni princip je pravilen; naš dekorator deluje.

Kaj lahko poskusite

Če razumete gornjo kodo, napišite dekorator, ki bo deloval za več argumentov: če navedemo, recimo, tri tipe

```
@f.register(int, int, float)
def _(x, y, z, w):
    pass
```

uporabi za izbor funkcije tipe prvih treh spremenljivk.