# 04 prvorazredni objekti

January 28, 2024

## 1 Prvorazredni objekti

V programu imamo različne objekte - številke, nize, sezname, terke… V programu ustvarjamo nove objekte. Objekti izginejo iz pomnilnika, ko jih nihče več ne potrebuje. Objekte lahko zlagamo v sezname in terke. Objekte lahko podajamo funkcijam kot argumente. Objekte lahko dobimo kot rezultat funkcije.

Nekateri jeziki (Python je med njimi, klasični prevajani jeziki pa tipično ne) se odlikujejo po tem, da je tudi funkcija objekt. Prvorazredni objekt. To pomeni, da lahko s funkcijami počnemo vse, kar lahko počnemo z drugimi objekti. Lahko ustvarjamo nove funkcije; in funkcije lahko izginejo iz pomnilnika, ko jih nihče več ne potrebuje. Lahko jih zlagamo v seznem, podajamo kot argumente in dobimo kot rezultat.

Objekti se seveda razlikujejo po svojih zmožnostih: nekatere lahko odštevamo in delimo, druge le seštevamo, tretjih niti to. Funkcij pač ne moremo sešteti. Lahko pa jih pokličemo. Prek seznamov pa lahko gremo z zanko `for`, česar ni mogoče početi ne s funkcijami ne s števili. A to je stvar definicije tipa. Bistvo pa je, da lahko z vsakim objektom, najsibo število, niz ali funkcija, počnemo stvari, našetete v prejšnjem odstavku.

### 1.1 Kaj je definicija funkcije?

Ko Python vidi definicijo funkcije, priredi pripadajočo kodo (in še kaj zraven) imenu. Definicija funkcije, na primer,

[1]:
```
def f(x):
    return 2 * x
```

je le nekoliko drugače zapisano prirejanje. Lahko si predstavljamo, da je gornje ekvivalentno spodnjemu.

```
f = def(x):
    return x
```

To sicer ni pravilna koda v Pythonu, v, recimo, Javascriptu pa v resnici lahko dejansko napišemo

```
> f = function(t) { return 2 * t }
> f(5)
10
```

(ali pa kaj še krajšega, a nepoznavalcu Javascripta manj očitnega).

### 1.1.1 Za uvod in okus

Imena funkcij so samo imena. Tako kot imena kateregakoli drugega objekta. Torej ni nič narobe (a tudi nič posebej prav), če rečemo

[2]:
```
napiši = print
```

in potem

[3]:
```
napiši(42)
```

```
42
```

Zdaj se `napiši` nanaša na isti objekt kot `print`. Imenu `print` lahko celo priredimo drugo vrednost (ker je to seveda totalno smiselno).

[5]:
```
print = 42
napiši(print)
```

```
42
```

ali celo

[6]:
```
print = abs
abs(-5)
```

[6]:
```
5
```

V tej obliki je to čisto brez zveze, vendar nam bo morda pomagalo razumeti, kar sledi, v ponaslednjem razdelku.

Zdaj pa samo popravimo `print`, da ne bo kasneje štale.

[28]:
```
print = napiši
```

### 1.1.2 Lambda-funkcije

Tudi v Pythonu lahko takole, na hitro definiramo funkcijo. Takšnim funkcijam navadno rečemo lambda-funkcije.

Torej: `lambda x, y: x + y` je funkcija, ki prejme dva argumenta (poimenovali smo ju `x` in `y`) ter vrne njuno vsoto. Definicija lambda funkcije je sestavljena iz ključne besede `lambda`, ki ji sledi seznam argumentov, kot pri običajnih funkcijah (a brez oklepajev), nato dvopičje in nato izraz. Brez `return`-a. Samo izraz. In samo en izraz.

Pythonove lambda-funkcije so omejene, zelo omejene. Če bi si človek česa želel v Pythonu, bi bile to boljše lambda-funkcije. Vendar jih ni in najbrž nikoli ne bo. Sintaktično se ne ujamejo z njim. Kdor hoče mogočne lambde (in ostati v mainstream jezikih), naj gre programirat v Kotlinu.

Takšna lambda seveda ničemur ne služi. Če jo hočemo še kdaj videti in poklicati, ji moramo dati ime. Recimo tako:

[7]:
```
sestej = lambda x, y: x + y
```

To se ne dela. Namesto tega bomo vedno pisali

def sestej(x, y): return x + y

Lambdo lahko pokličemo tudi kar takoj, ne da bi ji dali ime.

[8]:
```
(lambda x, y: x + y)(5, 1)
```

[8]:
```
6
```

Tudi to se ne dela, ker je neumno. (Ne spreglejte pa oklepajev: samo funkcijo smo morali dati v oklepaj. Če bi napisali `lambda x, y: x + y(5, 1)`, bi bilo to videti, kot da kličemo funkcijo `y`. Točneje, to je definicija lambde, ki vrača `x + y(5, 1)`.

Če se lambd ne uporablja ne tako ne drugače - kje pa se jih? "*Le z menoj, bralec!*", je napisal Bulgakov.

### 1.1.3 Funkcija kot argument

Pri Programiranju 1 stalno ponavljamo tole nalogo: iščemo element seznama, ki je največji po določenem kriteriju. Bodisi po velikosti, kot jo razume Python, bodisi po dolžini, po absolutni vrednosti, po številu a-jev, ki jih vsebuje (če gre za nize) … Vsakič napišemo novo funkcijo.

[9]:
```
def najdaljsi(s):
    # Vrne najdaljši element s
    naj_e = None
    naj_dol = None
    for e in s:
        dol = len(e)
        if naj_dol is None or dol > naj_dol:
            naj_dol = dol
            naj_e = e
    return naj_e

def najvecji_abs(s):
    # Vrne element s z
    naj_e = None
    naj_abs = None
    for e in s:
        abse = abs(e)
        if naj_abs is None or abse > naj_abs:
            naj_abs = abse
            naj_e = e
    return naj_e

def najvec_a_jev(s):
    # Vrne element (niz) z največ a-ji
    naj_e = None
    naj_ajev = None
    for e in s:
```

```
        ajev = e.count("a") + e.count("A")
        if naj_ajev is None or ajev > naj_ajev:
            naj_ajev = ajev
            naj_e = e
    return naj_e
```

Vse to je očitno eno in isto, razlikuje se le po tem, da enkrat pokličemo `len`, enkrat `abs`, enkrat pa seštevamo število malih in velikih a-jev.

Splošna funkcija `najvecji` bi bila videti tako.

[12]:
```
def max(s, key):
    max_e = None
    max_key = None
    for e in s:
        k = key(e)
        if max_key is None or k > max_key:
            max_key = k
            max_e = e
    return max_e
```

Vse je enako kot prej, le da namesto `len` ali `abs` kličemo `key`, pri čemer je `key` argument funkcije `max`.

Zdaj lahko naredimo tako

[13]:
```
imena = ["Ana", "Berta", "Dani"]
max(imena, len)
```

[13]:
```
'Berta'
```

kot

[15]:
```
stevila = [22, 5, -6, -42, 1]
max(stevila, abs)
```

[15]:
```
-42
```

V, recimo, prvem primeru, je znotraj funkcije `s` isto kot `imena` (en in isti seznam), `key` isto kot `len`. Ko pokličem `key(e)`, je to isto, kot če bi klicali `len(e)`, saj sta `key` in `len` ena in ista reč.

Kaj pa tretji primer, v katerem ne kličemo funkcije, temveč računamo `e.count("a") + e.count("A")`? Tu moramo funkcijo narediti sami. Lahko bi pisali

[16]:
```
def stevilo_ajev(s):
    return s.count("a") + s.count("A")

max(imena, stevilo_ajev)
```

[16]:
```
'Ana'
```

vendar je to nepraktično. Dolgo. Definirati celo funkcijo za tako preprosto reč. Tu je torej mesto, kjer uporabimo lambde.

[17]:
```
max(imena, lambda s: s.count("a") + s.count("A"))
```

[17]:
```
'Ana'
```

Python seveda natančno takšno funkcijo `max` že ima. Razlikuje se le po tem, da moramo argument s ključem nujno podati tako, da ga poimenujemo, na primer. Še en lep primer: v seznamu seznamov poišči seznam z največjo vsoto elementov:

[23]:
```
tabela = [ [1, 2, 3], [1, 2],[10, 1]]
max(tabela, key=sum)
```

[23]:
```
[10, 1]
```

Podoben argument ima tudi funkcija `min`. Imata ga celo metoda `sort` in funkcija `sorted`.

[24]:
```
sorted(stevila, key=abs)
```

[24]:
```
[1, 5, -6, 22, -42]
```

[25]:
```
sorted(tabela, key=sum)
```

[25]:
```
[[1, 2], [1, 2, 3], [10, 1]]
```

Tule smo uredili števila po njihovi absolutni vrednosti (kot "ključ" za urejanje smo uporabili funkcijo `abs`) in potem še po vsoti njihovih elementov.

### 1.1.4 Primer: seznami funkcij

Napišimo funkcijo, ki tabelira podani seznam funkcij na vrednostih od 0 do 1, s korakom 0.1.

[29]:
```
def tabeliraj(fs):
    for x10 in range(11):
        x = x10 / 10
        print(x, end="\t")
        for f in fs:
            print(f(x), end="\t")
        print()

from math import *
tabeliraj([sqrt, sin, lambda x: x**2])
```

```
0.0     0.0     0.0     0.0
0.1     0.31622776601683794     0.09983341664682815     0.010000000000000002
0.2     0.4472135954999579      0.19866933079506122     0.04000000000000001
0.3     0.5477225575051661      0.29552020666133955     0.09
0.4     0.6324555320336759      0.3894183423086505      0.16000000000000003
```

```
0.5     0.7071067811865476      0.479425538604203       0.25
0.6     0.7745966692414834      0.5646424733950354      0.36
0.7     0.8366600265340756      0.644217687237691       0.48999999999999994
0.8     0.8944271909999159      0.7173560908995228      0.6400000000000001
0.9     0.9486832980505138      0.7833269096274833      0.81
1.0     1.0     0.8414709848078965      1.0
```

Z argumentom `end="\t"` smo funkcijo `print` prosili, naj po vsakem izpisu ne gre v novo vrstico temveč le doda tabulator. To ni bistveno. Bistvena je notranja zanka: funkcija `tabeliraj` je kot argument dobila seznam funkcij, torej gremo v notranji zanki čez ta seznam in `f` je vsakič druga funkcija.

Funkcija je seveda na moč neelegantna. O tem, kako to narediti lepše, se bomo še učili; za zdaj le pokažimo, da kdo ne bi imel po krivici slabega mnenja.

[30]:
```
def tabeliraj(fs):
    for x10 in range(11):
        print("\t".join(map(str, [x] + [f(x) for x in fs])))
```

S številom decimalk v izpisu pa se res ukvarjajmo kdaj drugič.

## 1.2 Ustvarjanje in vračanje funkcij

Rekli smo, da definicija funkcije ustvari objekt-funkcijo in jo priredi imenu. Potemtakem smemo narediti tudi to.

[31]:
```
def f():
    def g(x):
        return x ** 2

    return g
```

So jeziki, v katerih ni mogoče pisati funkcij znotraj funkcij. So tudi jeziki, v katerih je to dovoljeno in takšnim funkcijam rečejo lokalne funkcije. In teh funkcij se ne da klicati iz zunanjega okolja. In te funkcije so ustvarjene le enkrat. So pa jeziki, v katerih to deluje boljše.

[32]:
```
t = f()
```

Kaj bom dobil, če pokličem `f`? Kaj je `t`? Pač, funkcija, ne.

[33]:
```
t(7)
```

[33]:
```
49
```

Zdaj pa naredim še

[34]:
```
u = f()
```

Sta `t` in `u` ista funkcija? V mnogih jezikih bi - če kaj temu podobnega sploh dopuščajo - bili. V mnogih pa ne. Python je med slednjimi. Najboljši dokaz bo tak:

```
def f(k):
    def g(x):
        return x ** k

    return g

kvadriraj = f(2)
kubiraj = f(3)

kvadriraj(5)

kubiraj(5)
```

Funkcija `f` vsakič ustvari novo funkcijo `g` in jo vrne. Ta funkcija ima dostop do okolja, v katerem je nastala. (Pravo ime za to je: *closure*.) Zato lahko uporablja `k`.

Deluje celo to

```
def f(k):
    k2 = k

    def g(x):
        return x ** k2

    return g
```

In celo

```
def f(k):
    k2 = k

    def g(x):
        return x ** k2

    return g
```

Zgoraj smo definirali funkcijo `max`, ki prejme seznam in ključ, podobno kot v Python vdelana funkcija `max`. Recimo, da bi potrebovali funkcije `najdaljse_ime`, `najvecja_vsota` in `naj_absolutist`.

```
def najdaljse_ime(s):
    return max(s, len)

def najvecja_vsota(s):
    return max(s, sum)

def naj_absolutist(s):
    return max(s, abs)
```

Tole se da narediti tudi preprosteje. (Ja, še preprosteje!) Napišemo lahko funkcijo za sestavljanje takšnih funkcij.

```
def naj_funkcija(f):
    def naj_f(s):
        return naj(s, f)
    return naj_f
```

Preden razložimo, kaj dela to čudo, povejmo, kako se uporablja. Kar smo napisali, je funkcija za sestavljanje funkcij.

```
najdaljse_ime = naj_funkcija(len)
najvecja_vsota = naj_funkcija(sum)
naj_absolutist = naj_funkcija(abs)
```

S to `naj_funkcija` lahko sestavljamo nove funkcije, ki vračajo največji element glede na podani kriterij.

Kaj naredi `naj_funkcija`? Definira funkcijo in jo vrne. Nova funkcija se imenuje `naj_f`, vendar je to samo lokalno ime znotraj funkcije `naj_funkcija`. Torej, še enkrat: `naj_funkcija` sestavi neko funkcijo in jo vrne. To funkcijo nato priredimo "spremenljivki" ("imenu" … kakorkoli že hočete reči temu) `najdaljse_ime`, `najvecja_vsota`, `naj_absolutist`… Te funkcije, `najdaljse_ime`, `najvecja_vsota`, `naj_absolutist` so torej funkcije, ki jih je sestavila in vrnila `naj_funkcija`.

Kaj pa dela funkcija, ki jo sestavi `naj_funkcija`? Funkcija, ki jo bo vrnila `naj_funkcija`, sprejme en argument, seznam `s`, in vrne največji element tega seznama, pri čemer ne primerja elementov "direktno", temveč glede na vrednost funkcije `f`. Katere funkcije `f`? Tiste, ki smo jo podali kot argument funkciji `naj_funkcija`.

### 1.3 Konec

Kje pa, s tem smo šele začeli. Prihodnjič: dekoratorji!

[ ]: