07b funkcijske funkcije

January 28, 2024

0.1 Funkcijsko programiranje

Tule bomo spoznali nekaj tipičnih funkcij iz zakladnice *funkcijskega programiranja*. Tu ne gre za "programiranje s funkcijami", kot bi kdo naivno pomislil, temveč na nek način zlaganja programov. Boste videli.

Tema je kar napredna in namenjena bolj zagretim študentom.

Različni jeziki imajo različno sintakso. Zelo veliko jih sicer povzema C-jevsko - kdor zna C ali Javo, mu bosta domača tudi Javascript in php). Pythonova ali, recimo, Kotlinova sintaksa je precej drugačna. Medtem ko našteti jeziki bloke označujejo z zavitimi oklepaji, jih Python z bloki; Pascal ima raje `begin` in `end`. Programer v C-ju mora pomagati prevajalniku, tako da sam skrbno postavlja kupe oklepajev in podpičij; v Pythonu je nepotrebno, ker je jezik zasnovan drugače, moderni Javascript pa meje med stavki (kjer bi morali v C dati podpičje, v Pythonu pa iti v novo vrstico) preprosto ugiba in (navadno) ugane.

Od sloga sintakse je odvisno, kaj se bo v jeziku lepo povedalo, kaj ne.

- Python, recimo, nima zanke do-while, ki jo morda poznate iz drugih jezikov. Osnovni razlog je, da se je v Pythonovem načinu oblikovanja blokov, z dvopičji, preprosto ne da lepo opisati. Ker je zanka do-while obenem zelo redko dejansko uporabna (sam jo zapogrešam najbrž enkrat letno, če sploh) in ker jo je preprosto nadomestiti z `while True` ter pogojnim `break` na koncu zanke, so se odločili, da z njo ne bodo kazili jezika (lahko pa pogledate predlog in odgovor).
- Drugačen primer je ternarni operator, v C-jevskih jezikih zapisan z `?:`. Ta način bi v Pythonu izstopal, ker je Python bolj "beseden" jezik (`and` in `or`, ne `&&` in `||`). Ker je vendarle uporaben in predvsem, ker so se programerji pogosto zatekali k alternativi, ki včasih ne da pravega rezultata (C-jevski `p ? a : b` so v Pythonu zapisali z `p and a or b`, kar pa da napačen rezultat, če je `p` resničen, `a` pa je, recimo, `0` ali prazen seznam), je Python po dolgih diskusijah dobil operator `if-else`, ki pa se ga celo malo izogibamo. Ker je čuden.
- Nekaj, kar je v Pythonu - ki je sicer sodoben jezik - absolutno grozno, so lambda-funkcije. Pri Programiranju 1 bi jih najbrž morali predstaviti, vendar Pythonove tega skoraj niso vredne. Omejene so na en sam izraz, nepregledne, uporabne zgolj v butičnih primerih. V Javascriptu in Kotlin med funkcijo in lambda-funkcijo ni razlike - tako kot mora biti. Lisp je pa itak razred zase.

Vsaka sintaksa ima torej svoje slabosti in prednosti. Python smo v zgornjih točkah le kritizirali, vendar to, recimo, le zato, ker smo pri predmetu doslej spoznavali, kako izrazna, učinkovita je njegova sintaksa. Glede na tokratno temo pa … mešani občutki.

Pythonov sintaktični slog je fantastičen, ko gre za izpeljane sezname, slovarje, množice. Osnovno

inspiracijo je dobil v jeziku Haskell, vendar (tudi) v Pythonu to deluje sintaktično lepo, pregledno.

Druga sintaksa za podoben način programiranja temelji na nekaj funkcijah, konkretno `map`, `filter` in `reduce`. V tej obliki jih bomo našli v večini drugih jezikov, v katere bi bilo morda težje uvesti Pythonovo (no, Hasklovo) sintakso. In, predvsem, v jezikih z močnejšo lambdo. Te funkcije ima tudi Python, vendar niso tako zelo uporabne. Veliko več zakladov pa najdemo v modulu itertools, ki tudi nekako sodi v to zgodbo.

0.2 Map

Funkcija `map` kot argument prejme funkcijo in nekaj, prek česar je možno nagnati zanko. Vsak element tega, nečesa, "premapira" čez funkcijo. Če imamo

[1]:
```
from math import sqrt

k = [9, 25, 16, 81]
```

bo `map(sqrt, k)` vrnil korene vseh števil v `k`:

[2]:
```
for x in map(sqrt, k):
    print(x)
```

Funkcija `map` dela, približno tole:

[3]:
```
def map(func, s):
    return [func(x) for x in s]
```

Do Pythona 3.0 je funkcija `map` v resnici vračala seznam, od različica 3.0 naprej pa vrne *iterator*. Za tiste, ki ne veste, kaj je to: vede se kot seznam, samo da ni; čezenj lahko gremo z zanko `for`. Za tiste, ki ne veste, pa bi radi izvedeli: preberite zapiske o generatorjih in iteratorjih. Za tiste, ki veste: ja, takšen:

[4]:
```
def map(func, s):
    for x in s:
        yield func(s)
```

Ali (isto, le krajše)

[2]:
```
def map(func, s):
    return (func(x) for x in s)
```

Funkcijo `map` smo pogosteje uporabljali do Pythona 2.0. Ta pa je uvedel izpeljane sezname. Prednost novejše različice je v tem, da

- ne zahteva funkcije, temveč izraz; generatorja `(x ** 2 for x in s)` ne moremo prepisati v `map(**2, s)`, temveč potrebujemo lambdo: `map(lambda x: x ** 2, s)`;
- je `map` počasnejši, ker vedno kliče funkcijo, medtem ko je novejši zapis, generator, ne (vsak, dokler lahko vse opravimo z izrazom).

Osebno `map` rad uporabim, kadar imam funkcijo ravno pri roki in kadar izgleda sintaktično lepše.

Se pravi: redko.

0.3 Filter

Funkcija `filter` je druga funkcija, ki so jo izpeljani seznami spravili ob delo. `filter(func, s)` vrne vse tiste elemente `s`, pri katerih `func` vrne `True`.

[5]:
```
def vsebuje_i(s):
    return "i" in s

imena = ["Ana", "Berta", "Cilka", "Dani", "Ema"]

for x in filter(vsebuje_i, imena):
    print(x)
```

To je seveda isto kot `(x for x in imena if vsebuje_i(x))`, kar je tako ali tako le bolj zapletena različica `(x for x in imena if "i" in x)`. Resnici na ljubo tudi `filter` ne potrebuje poprej definirane funkcije, saj bi lahko pisali `filter(lambda x: "i" in x, imena)`. Vendar je očitno, zakaj filtra ne vidimo več velikokrat.

Izpeljani seznami, slovarji, množice in generatorji v enem zamahu naredijo oboje, mapirajo in filtrirajo.

0.4 Reduce

Funkcija `reduce` je edina iz te družbe, ki ni ostala brezposelna. No, hkrati pa tudi najmanj uporabna od njih, saj Python ni ravno jezik za te hece. Mogoče je tudi to razlog, da jo dobimo v modulu `functools` in ne kar tako, na prostem.

`reduce(func, s)` je nekako ekvivalenten temu `func(func(func(func(s[0], s[1])), s[2]), s[3]), s[4])` - če je `s` seznam s petimi elementi. Ali, v kodi (ki sicer ne zna vsega, kar zna `reduce`):

[6]:
```
def reduce(func, s):
    acc = s[0]
    for x in s[1:]:
        acc = func(acc, x)
    return acc
```

Po domače: `reduce` pokliče funkcijo na prvih dveh elementih, nato na rezultatu tega klica in tretjem elementu, nato na rezultatu tega klica in četrtem elementu… Spremenljivko `acc` pa smo poimenovali po njeni vlogi: akumulator.

Če vemo, kaj so iteratorji in kaj počne `next`, znamo bolj natančno (če ne, pa nič narobe, tudi gornje je dovolj dobro za razumevanje, ki ga potrebujemo za uporabo funkcije):

[7]:
```
def reduce(func, s, acc=None):
    t = iter(s)
    if acc is None:
        acc = next(t)
```

```
    for x in t:
        acc = func(acc, x)
    return acc
```

Z `reduce` se da početi zanimive stvari. Pripravimo si nekaj funkcij (ki bi lahko bile tudi lambde, ampak recimo, da jih ne znamo pisati).

[8]:
```
def sestej(a, b):
    return a + b

def zmnozi(a, b):
    return a * b

def vrni_vecjega(a, b):
    if a > b:
        return a
    else:
        return b

def oba_resnicna(a, b):
    return a and b
```

Pripravimo si še priložnostni seznam števil.

[9]:
```
s = [4, 2, 6, 3]
```

Z `reduce` lahko zdaj izračunamo vsoto elementov seznama

[10]:
```
reduce(sestej, s)
```

[10]: 15

produkt

[11]:
```
reduce(zmnozi, s)
```

[11]: 144

in poiščemo največji element

[12]:
```
reduce(vrni_vecjega, s)
```

[12]: 6

mimogrede pa še 10!, se pravi produkt števil do 10

[13]:
```
reduce(zmnozi, range(1, 11))
```

[13]:
```
3628800
```

Če imamo seznam `True`-jev in `False`-ov, lahko z `reduce` izračunamo njegovo konjunkcijo (`and` prek vseh elementov').

[14]:
```
reduce(oba_resnicna, [True, True, True, True, True])
```

[14]:
```
True
```

[15]:
```
reduce(oba_resnicna, [True, True, True, False, True])
```

[15]:
```
False
```

Imenitna reč, problem je le, da se nam teh funkcij ne da definirati vnaprej, Pythonove lambde, s katerimi lahko funkcijo definiramo kar sproti, znotraj klica `reduce`, pa so zelo kilave in tudi nikoli ne bodo drugačne kot kilave.

1 Zakladi iz modula `itertools`

Za dve funkciji iz modula `iterools` smo že povedali v "glavnem" delu predavanja: `chain` in `count`. Tidve boste potrebovali najpogosteje. Poleg njih vsebuje še veliko drugih - zanimivih in uporabnih, če se spomnimo nanje.

1.0.1 `pairwise`

Zaporedne elemente seznama dobimo, vemo, z `zip(s, s[1:])`. Od različice 3.10 lahko uporabimo `pairwise`:

[7]:
```
s = ["Ana", "Berta", "Cilka", "Dani", "Ema", "Fanči", "Greta", "Helga"]
```

[8]:
```
from itertools import pairwise

for x, y in pairwise(s):
    print(x, y)
```

```
Ana Berta
Berta Cilka
Cilka Dani
Dani Ema
Ema Fanči
Fanči Greta
Greta Helga
```

1.0.2 `cycle in repeat`

`cycle` preprosto ponavlja seznam v neskončnost. Zanke

```
for x in cycle(s):
    print(x)
```

raje ne poganjajmo, saj bi v neskončnost ponavljala gornjih osem imen. Zanko čez `cycle` bomo vedno z nečim prekinili. Recimo tako:

[9]:
```
from itertools import cycle

for ime, smer in zip(s, cycle(["levo", "desno"])):
    print(ime, smer)
```

```
Ana levo
Berta desno
Cilka levo
Dani desno
Ema levo
Fanči desno
Greta levo
Helga desno
```

Ta primer tudi nakazuje rdečo nit tega, kar bomo počeli tu - in kar se v splošnem trudimo početi s takimi funkcijami - opraviti čimveč dela s smiselnim gnezdenjem teh funkcij.

`repeat` uporabimo, če želimo v neskončnost ponavljati en sam element. V resnici ga ne potrebujemo velikokrat; tule je malo umeten primer.

[11]:
```
from itertools import repeat, chain

for ime, vrata in zip(s, chain(range(1, 4), repeat(4))):
    print(ime, "gre skozi vrata", vrata)
```

```
Ana gre skozi vrata 1
Berta gre skozi vrata 2
Cilka gre skozi vrata 3
Dani gre skozi vrata 4
Ema gre skozi vrata 4
Fanči gre skozi vrata 4
Greta gre skozi vrata 4
Helga gre skozi vrata 4
```

Če zmanjka vrat (ki jih generira `range(1, 4)`), morajo vsi skozi zadnja vrata, 4.

1.0.3 `zip_longest`

Gornje sicer ni nič drugega kot

[28]:
```
from itertools import zip_longest

for ime, vrata in zip_longest(s, range(1, 4), fillvalue=4):
    print(ime, "gre skozi vrata", vrata)
```

```
Ana gre skozi vrata 1
Berta gre skozi vrata 2
```

```
Cilka gre skozi vrata 3
Dani gre skozi vrata 4
Ema gre skozi vrata 4
Fanči gre skozi vrata 4
Greta gre skozi vrata 4
Helga gre skozi vrata 4
```

Funkcija `zip` vedno vrne toliko reči, kolikor jih je v krajšem izmed podanih seznamov (ali česarkoli že). Funkcija `zip_longest` generira reči toliko časa, kolikor zmore najdaljši od podanih argumentov, manjkajoče vrednosti pa nadomešča s podano `fillvalue`.

1.0.4 `batched`

Tole je čisto sveža pridobitev, iz Pythona 3.12. Ker v času sestavljanja zapiskov poganjam Python 3.11 (nova različica Pythona vedno pride v začetku oktobra, zato pri predmetu vedno uporabljamo prejšnjo), tega primera niti e ne moremo pognati:

```
from itertools import batched

for skupina in batched(s, 3):
    print(skupina)
```

v različici 3.12 izpiše

```
["Ana", "Berta", "Cilka"]
["Dani", "Ema", "Fanči"]
["Greta", "Helga"]
```

To funkcijo smo v resnici pogrešali in se zatekali celo k takšnim norostim, kot je

[24]:
```
for skupina in zip(*[iter(s)] * 3):
    print(skupina)
```

```
('Ana', 'Berta', 'Cilka')
('Dani', 'Ema', 'Fanči')
```

Rezultat je podoben, le da manjka zadnja, nepopolna skupina. Razumeti, zakaj to deluje, pa naj bo v izziv tistim, ki imajo radi izzive.

1.0.5 `compress`

Funkcija `compress` "kompresira" sezname tako, da odstrane neželene elemente.

[25]:
```
from itertools import compress

primerna = [True, True, False, True, False, False, True]

list(compress(s, primerna))
```

[25]:
```
['Ana', 'Berta', 'Dani', 'Greta']
```

V osebnoizpovedni noti naj povem, da te funkcije nisem opazil vse do teh predavanj in zato redno pridno pisal

[26]:
```
[x for x, p in zip(s, primerna) if p]
```

[26]:
```
['Ana', 'Berta', 'Dani', 'Greta']
```

Nekako isto, vendar brez potrebe daljše.

Istočasno naj to služi kot še en primer, ko izpeljani seznami nudijo isti mehanizem kot te funkcije.

1.0.6 `takewhile`, `dropwhile`

`takewhile(func, s)` sprejme funkcijo in neko zaporedje (recimo seznam) ter vrača njegove člene, do prvega, za katerega `func` vrne `False` (oziroma neresnično vrednost).

Tole bi rešilo nalogo, ki bi spraševala, do kod se lahko pripeljemo po podani poti, če ne moremo voziti, po poteh, ki jih ni in po poteh, ki ne zahtevajo nobene veščine.

[40]:
```
from itertools import takewhile

A, B, C, D = "ABCD"
zemljevid = {(A, B): "trava", (A, C): "avtocesta", (A, D): "robnik", (C, B):␣
 ↪"bolt", (B, A): "trava"}
pot = "ABACDA"

for iz, v in takewhile(zemljevid.get, pairwise(pot)):
    print("Gremo iz", iz, "v", v)
print("Pot se konča v ", v)
```

```
Gremo iz A v B
Gremo iz B v A
Gremo iz A v C
Pot se konča v  C
```

Funkciji `takewhile` smo podali `zemljevid.get`. Ta bo prejemala pare, ki jih vrača `pairwise`. Če par obstaja, `get` vrne pripadajočo vrednost. Če je ta prazna, je neresnična in pot se ustavi (ker smo rekli, da ne bomo šli po povezavah, ki ne zahtevajo nobene veščine). Če povezava ne obstaja, pa `get` vrne `None`, kar je prav tako neresnično.

Če nas vmesni koraki ne zanimajo, pišemo kar

[41]:
```
from itertools import takewhile

A, B, C, D = "ABCD"
zemljevid = {(A, B): "trava", (A, C): "avtocesta", (A, D): "robnik", (C, B):␣
 ↪"bolt", (B, A): "trava"}
pot = "ABACDA"

for iz, v in takewhile(zemljevid.get, pairwise(pot)):
```

```
    print("Gremo iz", iz, "v", v)
print("Pot se konča v ", v)
```

```
Gremo iz A v B
Gremo iz B v A
Gremo iz A v C
Pot se konča v  C
```

Za rešitev domače naloge, v kateri nas zanima, do katere točke na zemljevidu pridemo s podanimi veščinami in katere veščine nam manjkajo, pa lahko uporabimo `dropwhile`. Ta izpušča člene, dokler zanje funkcije vrača `True` (oziroma resnično vrednost). Nas zanima le prva vrednost - in izvabimo jo z `next`.

[43]:
```
from itertools import dropwhile, pairwise

def koncna_tocka(pot, zemljevid, vescine):
    zemljevid = dvosmerni_zemljevid(zemljevid)
    pov = next(dropwhile(lambda pov: pov in zemljevid and zemljevid[pov] <=␣
 ↪vescine, pairwise(pot)))
    return pov[0], zemljevid[pov] - vescine
```

Uporabili smo še slavno lambdo: `lambda pov: pov in zemljevid and zemljevid[pov] <= vescine` je "sproti definirana funkcija", ki prejme en argument (`pov`) in vrne vrednost izraza `pov in zemljevid and zemljevid[pov] <= vescine`. Ta je resničen, če smemo prehoditi to povezavo. `dropwhile`-u damo to funkcijo in pare točk na poti. Ko ga `next` pozove, naj vrne naslednji element, ta preskoči vse povezave, ki jih smemo ubrati in vrne prvo, ki je ne moremo. Vrnemo prvo točko te povezave in, seveda, veščine, ki jih zahteva ta povezava ni jih kolesar nima (`zemljevid[pov] - vescine`).

1.1 Zaključne misli :)

Menda smo videli, za kaj gre: kup funkcij, ki jih lahko nizamo in vsaka procesira – predeluje, filtrira, preskakuje, grupira – zaporedja in jih podaja naslednji funkciji. To včasih vodi v elegantne rešitve, včasih pa v nerazumljive.

Slednje je v Pythonu kar pogosto. Problem je, da se funkcije nizajo odznotraj navzven, argumenti pa izgubljajo nekje na koncu. Primer smo videli v eni domačih nalog.

[19]:
```
from operator import itemgetter
from itertools import groupby

def zapisi(ovire):
    return "\n".join(zapisi_vrstico(y, sorted(x[:2] for x in group))
                     for y, group in groupby(sorted(ovire, key=itemgetter(2)),␣
 ↪itemgetter(2)))
```

Deluje, ni pa berljivo. Python je lep jezik, ni pa vsak jezik lep za vsak slog programiranja. Če se kdo potrudi to prebrati, bo videl, da mora brati nazaj - najprej se zgodi (drugi) `sorted`, nato `groupby`, potem `zapisi_vrstico`, katere rezultati se združijo z `join`. To bi se bralo veliko lepše v

jezikih, v katerih se to zapiše naprej, recimo v Kotlinu ali Javascriptu.

Tule je primer funkcije, ki vrne ime podanega direktorija (`slug`) in vseh poddirektorijev znotraj njega (vgnezdeno, torej s podpodpodpodpodpoddirektoriji). Funkcija je napisana v enakem slogu v JavaScriptu in Pythonu.

Javascript:

```
const traversePaths = (slug) => fs
    .readdirSync(slug)
    .map((name) => path.join(slug, name))
    .filter((subslug) => fs.statSync(subslug).isDirectory())
    .reduce((acc, subslug) => [...acc, ...traversePaths(subslug)], [slug]);
```

Python:

```
from os import path, listdir
from itertools import reduce

def traversePaths(slug):
    return reduce(lambda acc, subslug: acc + traversePaths(subslug),
                  filter(path.isdir,
                         map(lambda name: path.join(slug, name),
                             listdir(slug)
                            )
                        ),
                  [slug]
                 )
```

Funkcija dela tako:

- preberemo direktorij (`readDirSync` v JavaScriptu, `listdir` v Pythonu),
- v `map` vsakemu imeno dodamo pot do njega (`path.join`),
- s `filter` obdržimo le imena, ki se nanašajo na direktorije (`isDirectory`, `isDir`),
- z `reduce` k `[slug]` prištejemo sezname vseh imen, ki jih dobimo z rekurzivnim klicem `traversePath`.

Vidimo problem? V Javascriptu so `map`, `filter`, `reduce` in podobno metode seznamov (no, arrayev), zato se berejo v pravilnem vrstnem redu. V Pythonu so to funkcije, ki dobijo sezname kot argument, zato mora biti tisto, kar se zgodi prej, napisano kasneje. K temu dodajmo še zoprne lambde … in ta funkcija v Pythonu pač ne sodi v produkcijsko kodo.