07a - zakaj se to učimo

January 28, 2024

1 Zakaj se učimo pisati "onelinerje"?! Ni to grdo?!

Nekateri študenti so zagnani za programe v enih vrsticah. To je šport, izziv, … Ampak - mar to res sodi v Programiranje 1?

Ne.

In vsako leto me skrbi, ko sprašujejo, "ali se da tudi za tole narediti oneliner?". Da. Vse se da. Ampak je navadno brez zveze. Potem, ko znaš, pogosto ni niti več izziv. Kot da bi se vprašal, ali se da iti peš v Pariz. Ja, samo ne vem, zakaj bi to počel.

Zakaj se potem učimo "pisati onelinerje"? Saj se ne. Učimo se pisati generatorje, iteratorje, izpeljane sezname, množice in slovarje. Zakaj? Namen vaje je privajanje na drugačen način razmišljanja, ki je potreben za to.

Poglejmo na primeru. Recimo, da imamo nek seznam in naloga je, da preštejemo, kolikokrat se pojavi kateri element.

[1]:
```
s = [4, 5, 1, 3, 2, 1, 6, 3, 3, 4, 3, 1, 1, 4]
```

"Po starem" bi naredili tako.

[2]:
```
stevci = {}
for x in s:
    if x not in stevci:
        stevci[x] = 0
    stevci[x] += 1

stevci
```

[2]:
```
{4: 3, 5: 1, 1: 4, 3: 4, 2: 1, 6: 1}
```

Tipično vprašanje, ki ga vsako leto dobim vsaj nekajkrat, je: a se da znotraj izpeljanega slovarja spreminjati njegove elemente? Dopolnjevati, prištevati … karkoli?

1.0.1 Spreminjanje slovarja (z `reduce`)

V principu se da, vendar je točno to tisto, česar nočemo. Če vemo, da je gornje isto kot

[3]:
```
stevci = {}
for x in s:
```

```
    stevci = {**stevci, x: 1 + stevci.get(x, 0)}

stevci
```

[3]: {4: 3, 5: 1, 1: 4, 3: 4, 2: 1, 6: 1}

(ta trik mi je na predavanjih pokazal eden od študentov; sam sem počel nekaj bolj zapletenega)

lahko vse skupaj spremenimo v

[4]:
```
from functools import reduce

stevci = reduce(lambda acc, x: {**acc, x: 1 + stevci.get(x, 0)}, s, {})

stevci
```

[4]: {4: 4, 5: 2, 1: 5, 3: 5, 2: 2, 6: 2}

Tega niti ne želim prav posebej razlagati - le toliko, da boste razumeli, kaj v osnovi počne. V začetku imamo prazen slovar, potem pa mu v vsakem koraku dodamo ključ `x` z vrednostjo, ki je za `1` večja od vrednosti, ki je bila pod ključem `x` shranjena prej (ali `0`, če je še ni).

Ta `reduce` torej počne natančno isto kot prva rešitev, tista, ki ni v eni vrstici in uporablja čisto običajno zanko. Zato je to brez zveze. Na ta način se učimo le pisati težje razumljive programe.

1.0.2 Pravilna rešitev: izpeljani slovar

Pravilna rešitev je

[5]:
```
stevci = {x: s.count(x) for x in set(s)}

stevci
```

[5]: {1: 4, 2: 1, 3: 4, 4: 3, 5: 1, 6: 1}

S tem povemo, da hočemo slovar, katerega ključi so elementi seznama `s` (mimogrede jih damo v množico, zato da se z vsakim ukvarjamo le enkrat), pripadajoče vrednosti pa števil pojavitev tega elementa v `s`-u.

Razlika je v tem: zgornja rešitev je "*proceduralna*". Opisali smo postopek. Pythonu smo rekli, naj naredi prazen slovar, nato gre čez vse elemente `s`-ja in tako naprej.

Ta pa je bolj "*deklarativna*": povedali smo, kakšen slovar želimo. Seveda se zavedamo tudi, kako bo Python to dobil. Šel bo čez vse vrednosti in za vsako ustvaril ključ in vrednost. A narava, izgled programa je deklarativen. Povemo, kaj hočemo. Opišemo končni rezultat in ne nekih korakov in sprememb, ki bodo vodile do njega.

Zato se učimo pisati takšne programe.

Če nekdo reši nalogo na gornji način, z `reduce`, to štejem(o) kot pravilno samo zato, ker je težko dobro razmejiti, kaj je dovoljeno in kaj ne, ter to preskušati s testi. A zares pravilno rešena je, če je rešena na drugi način.

1.0.3 Vzporedno računanje

Rešitev, ki sem jo razglasil za pravilno, je pravzaprav očitno precej slabša. Prvotna rešitev gre prek seznama `s` enkrat samkrat. Ta gre najprej enkrat samo zato, da sestavi množico, potem pa za vsak element te množice prepotuje celoten seznam in šteje.

V tem kontekstu to ni uporabno, zato bi bila "produkcijska" rešitev te naloge takšna, kot smo jo napisali na začetku. Ali pa bi poklicali kar `itertools.Counter(s)` (ki dela isto).

Zdaj pa si predstavljamo, da naša naloga ni šteti elemente, temveč ugotoviti, koliko elementov ima določeno lastnost. Torej nekaj takega

```
{lastnost: imajo_lastnost(s, lastnost) for lastnost in lastnosti}
```

pri čemer je `imajo_lastnost(s, lastnost)` funkcija, ki pove, koliko elementov seznama `s` ima lastnost `lastnost`. (Lahko je, recimo `sum(ima_lastnost(x, lastnost) for x in s)`, pri čemer `ima_lastnost(x, lastnost)` pove, ali ima `x` lastnost lastnost.)

Nadalje predpostavimo, da imamo na voljo toliko procesorjev (jeder, računalnikov …), kolikor je lastnosti. Ali, če že ne ravno toliko, vsaj "več kot enega". V tem primeru se lahko vsak procesor ukvarja z eno lastnostjo (in če jih je manj, kot je lastnosti, procesor najprej računa eno lastnost, nato se loti druge.

Z drugimi besedami: elemente tega slovarja je možno računati vzporedno, istočasno na več procesorjih.

Osnovna prednost izpeljanih seznamov, slovarjev, množic je prav v tem, da so njihovi elementi med seboj neodvisni. Vrednost enega elementa je neodvisna od vrednosti ostalih.

Pri rešitvi z `reduce` to ni očitno. Tam slovar dopolnjujemo. Stvari tečejo serijsko. Tisto je nemogoče paralelizirati.

1.0.4 Moderni jeziki in ogrodja

Takšni triki (hm, no, ja, vključno z `reduce`) se na veliko uporabljajo v *funkcijskih jezikih*. Ti imajo precej dobrih lastnosti. Pogosto jih je preprosto paralelizirati; če jih pravilno uporabljamo, so varnejši; znajo biti tudi preglednejši. Še ena zanimiva lastnost: pogosto nimajo spremenljivk, temveč samo konstante. Na vprašanje "Ali se da spreminjati elemente slovarja znotraj izpeljanega slovarja" odgovorimo, čim izgovoritmo "spreminjati". Ne.

To je torej razlog, da se učimo teh stvari. V zadnjem času imajo vsi jeziki vedno več funkcijskega pridiha (Python malo, Kotlin pa je tozadevno že prav super), tako da je dobro, da se vadite v razmišljanju na ta način.