05 izpeljani seznami, mnozice, slovarji

January 28, 2024

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

Tokratna tema bo pustila - napačen! - vtis, da se učimo pisati funkcije v eni vrstici. A ta ekshibicionizem res sodi v Programiranje 1?! Ni to pravzaprav grdo?

Ne sodi. In zna biti grdo.

Nekateri študenti so za ta šport, izziv, res zagnani. "A se da tudi za tole narediti oneliner?" je kar pogosto vprašanje. 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? To je kratko in jedrnato, povsem razumljivo in tudi hitrejše - če uporabljamo po pameti. Vsekakor torej nekaj, kar je koristno znati. Če v to silimo, kadar ni potrebe, pa bomo seveda napisali kodo, ki bo daljša, nerazumljiva in počasna.

Kar bomo počeli danes, je torej še eno orodje, ki nam je na voljo, uporabljati pa ga je potrebno, kot vsa druga orodja, v pravem trenutku.

Drugi namen tega je privajanje na drugačen način razmišljanja pri progrmairanju. Na programe - predvsem kodo, ki je sestavila kak seznam, slovar, množico - smo doslej gledali *postopkovno*. Opisali smo *korake* gradnje seznama, slovarja, množice – vsak element posebej smo izračunali in ga *dodali* z `append`, `add` ali kakorkoli že. Današnji programi bodo *deklarativni*: namesto, da bi povedali, kako sestaviti seznam, bomo povedali zgolj, kaj naj vsebuje. Koda niti ne bo tako zelo različna, le malo obrnjena bo. Popolnoma drugačen pa bo moral biti način razmišljanja ob pisanju in branju.

Poleg tega dveh razlogov pa je pisanje funkcij v eni vrstici seveda tudi zabaven šport - tudi, kadar je rezultat takšen, da ni primeren za produkcijsko kodo. :)

0.2 "Deklarativni zapis"

Kar bomo počeli, pravzaprav že poznate. Pri matematiki množico včasih opišemo tako, da naštejemo njene elemente.

$$S = \{1, 2, 3\}$$

Včasih pa zgolj opišemo njihove lastnosti (z nekaj previdnosti, da nas ne obrije Russell). Takole bi opisali množico dvakratnikov vseh naravnih števil, katerih kvadrat je manjši od 120, takole:

$$S = \{2x; \forall x \in \mathbb{N} \wedge x^2 < 120\}.$$

0.3 Izpeljani seznami

Začnimo s tistim, kar poznamo najdlje: s seznami. Doslej smo jih zapisali tako, da smo v oglatih oklepajih naštevali njihove elemente,

[1]:
```
k = [9, 16, 81, 25, 196]
```

Oglate oklepaje bomo uporabljali tudi poslej, le da znotraj njih ne bomo našteli elementov, temveč napisali izraz, ki jih sestavi. Recimo, da bi hotel seznam korenov števil iz gornjega seznama (po nekem čednem naključju gornji seznam vsebuje ravno same kvadrate). Naredil bi tako:

[2]:
```
from math import sqrt

[sqrt(x) for x in k]
```

Torej: napisal sem oglate oklepaje, kot vedno, kadar definiram seznam, a namesto, da bi naštel elemente, sem povedal, kakšni naj bodo. Rekel sem, naj bodo elementi mojega novega seznama *koren iz x*, pri čemer so `x` elementi seznama `k`. Kaj pa, če bi hotel namesto tega imeti kvadrate števil? Isti šmorn:

[3]:
```
[x ** 2 for x in k]
```

Oblika tega, kar pišemo, je vedno `[izraz for spremenljivka in nekaj]`, kjer je *spremenljivka* ime neke spremenljivke (npr. `x`, *izraz* nek izraz, ki (običajno, ne pa čisto nujno) vsebuje to spremenljivko (npr. `sqrt(x)` ali `x ** 2`), *nekaj* pa je nekaj, čez kar je možno spustiti zanko `for`, torej seznam, slovar, množica ali kaj, česar še ne poznamo, vendar bomo spoznali še danes.

Ko sestavljamo tak *izpeljan seznam*, moramo torej razmišljati:

- kaj hočemo v tem seznamu? S kakšnim izrazom opišem (izračunem) element?
- prek česa (katerega seznama, `range`-a…) moram iti, da bom dobil te elemente.

Poglejmo še nekaj primerov.

Imam seznam imen, recimo,

[4]:
```
imena = ["Ana", "Berta", "Cilka", "Dani", "Ema"]
```

Kako bi izračunal poprečno dolžino imena? Imam funkcijo `sum`, ki zna sešteti števila v seznamu. Potrebujem torej le še seznam dolžin (namesto seznama imen). Pa poglejmo gornji "točki". Kaj želim v seznamu? Dolžine imen. Kako jih dobim? Očitno z nekim `len(ime)`, pri čemer je `ime` pač vsako ime iz seznama `imena`. V tem stavku smo odgovorili na obe vprašanji – kaj bom računal (`len(ime)`) in čez kaj bom spustil zanko (čez `imena`). Torej vemo: potrebujemo

[5]:
```
[len(ime) for ime in imena]
```

Zdaj pa tole le še seštejem in delim z dolžino seznama.

[6]:
```
sum([len(ime) for ime in imena]) / len(imena)
```

Poprečna dolžina imena ni nič posebej zanimivega. Računajmo raje poprečno težo in, da nam ne bo dolgčas, jo bomo računali iz teh troj `(teža, ime, je_ženska)`:

[7]:
```
podatki = [
    (74, "Anže", False),
    (82, "Benjamin", False),
    (58, "Cilka", True),
    (66, "Dani", False),
    (61, "Eva", True),
    (84, "Franc", False)]
```

Takole nam je storiti: z zanko se zapeljemo čez podatke, izračunamo vsoto prvih elementov in jo delimo z dolžino seznama.

Najprej poglejmo, kako bi to počeli po starem. Lepo prosim, ne tako:

[8]:
```
vsota = 0
for i in range(len(podatki)):
    vsota += podatki[i][0]

print(vsota / len(podatki))
```

Tole je že boljše:

[9]:
```
vsota = 0
for podatek in podatki:
    vsota += podatek[0]

print(vsota / len(podatki))
```

Če hočete, da vas vaš profesor ohrani v trajnem lepem spominu, pa boste sprogrmirali tako:

[10]:
```
vsota = 0
for teza, ime, zenska in podatki:
    vsota += teza

print(vsota / len(podatki))
```

Zdaj pa naredimo tako, kot smo se naučili danes. Naredimo lahko

[11]:
```
vsota = sum([podatek[0] for podatek in podatki])

print(vsota / len(podatki))
```

ali, po boljši, zadnji različici

[12]:
```
vsota = sum([teza for teza, ime, zenska in podatki])

print(vsota / len(podatki))
```

0.3.1 Izpeljani seznami in pogoji

Pri izpeljevanju seznamov lahko dodamo še pogoj, s katerim izbiramo elemente, ki naj se dodajo.

Kako bi sestavili seznam kvadratov vseh lihih števil do 100? Seznam kvadratov vseh števil do 100 je trivialen, napisati nam je treba le:

[13]:
```
[x ** 2 for x in range(10)]
```

Če želimo pobrati samo liha števila, lahko v izpeljavo seznama dodamo še pogoj.

[14]:
```
[x ** 2 for x in range(10) if x % 2 == 1]
```

Tole pravzaprav ni čisto potrebno, lahko bi rekli preprosto

[15]:
```
s = [x ** 2 for x in range(1, 10, 2)]
```

Kaj pa, če bi hoteli seznam vseh števil do 30, ki niso deljiva s 7 in ne vsebujejo števke 7?

[16]:
```
[x for x in range(30) if x % 7 != 0 and "7" not in str(x)]
```

[16]:
```
[1,
 2,
 3,
 4,
 5,
 6,
 8,
 9,
 10,
 11,
 12,
 13,
 15,
 16,
 18,
 19,
 20,
 22,
 23,
 24,
 25,
 26,
 29]
```

Prvi del pogoja poskrbi, da število ni deljivo s 7 (ostanek po deljenju s 7 mora biti različen od 0). Drugi del poskrbi, da število ne vsebuje števke 7: število preprosto pretvorimo v niz in preverimo, da ne vsebuje sedmice.

Kako bi izračunali vsoto tež moških v gornjem seznamu `podatki`? Za zanko dodamo pogoj. Za

začetek sestavimo le seznam imen vseh moških.

[17]:
```
[ime for teza, ime, zenska in podatki if not zenska]
```

[17]:
```
['Anže', 'Benjamin', 'Dani', 'Franc']
```

V definicijo našega seznama lepo na konec, za `for`, dodamo še `if` in pogoj, ki mu mora zadoščati element, da nas zanima.

Na enak način naberemo tudi teže.

[18]:
```
[teza for teza, ime, zenska in podatki if not zenska]
```

[18]:
```
[74, 82, 66, 84]
```

Elemente tega seznama seštejemo s `sum`, pa smo.

[19]:
```
sum([teza for teza, ime, zenska in podatki if not zenska])
```

[19]:
```
306
```

Zdaj veste, zakaj sem tako utrujal, da ne uporabljajte `for i in range(len(s))`: zato, ker z njim namesto lepega, kratkega preglednega `[teza for teza, ime, zenska in podatki if not zenska]` dobimo `[podatki[i][0] for i in range(len(podatki)) if not podatki[i][2]]`. Seveda lahko delate tudi na ta, drugi način, če vam je v užitek. V svobodi živimo; noben zakon ne prepoveduje mazohizma.

Sestavimo seznam vseh deliteljev danega števila `n`. Vzemimo, recimo `n = 60`. Seznam deliteljev je tedaj seznam vseh `x`, pri čemer `x` prihaja z intervala 1 x n, za katere velja, da je ostanek po deljenju `n` z `x` enak 0 (torej `x` deli `n`).

[20]:
```
def delitelji(n):
    return [x for x in range(1, n + 1) if n % x == 0]
```

Zdaj lahko hitro ugotovimo, ali je dano število popolno: popolna števila so (se še spomnimo?) števila, ki so enaka vsoti svojih deliteljev (brez sebe samega).

[1]:
```
def popolno(n):
    return n == sum(delitelji(n)) - n
```

Še bolj imenitna števila so praštevila, števila brez deliteljev. Se pravi tista, ki so deljiva le z 1 in s samim seboj.

[22]:
```
def prastevilo(n):
    return delitelji(n) == [1, n]
```

Če funkcije `delitelji` še ne bi imeli - nič hudega. Ali je število praštevilo, bi še vedno dognali v eni sami vrstici. Poleg tega bi lahko šli le od 2 do n (brez n); tako bi izpustili 1 in n ter število razglasili za praštevilo, če je je deljivo le z 1 in s samim seboj.

[23]:
```
def prastevilo(n):
    return [x for x in range(1, n + 1) if n % x == 0] == [1, n]
```

Seveda je preprosteje, če spremenimo meje v `range` tako, da ne vključujejo `1` in `n`.

[24]:
```
def prastevilo(n):
    return [x for x in range(2, n) if n % x == 0] == []
```

Še boljše pa je, če se spomnimo, da so prazni seznami neresnični.

[25]:
```
def prastevilo(n):
    return not [x for x in range(2, n) if n % x == 0]
```

Za bis dodajmo še nekaj, česar ne bomo razložili.

[26]:
```
def prastevilo(n):
    return all(n % x for x in range(2, n))
```

S to funkcijo - v katerikoliže obliki - hitro sestavimo seznam vseh praštevil med 2 in 100:

[27]:
```
[n for n in range(2, 101) if prastevilo(n)]
```

[27]:
```
[2,
 3,
 5,
 7,
 11,
 13,
 17,
 19,
 23,
 29,
 31,
 37,
 41,
 43,
 47,
 53,
 59,
 61,
 67,
 71,
 73,
 79,
 83,
 89,
 97]
```

Znamo pa, jasno, tudi brez funkcije; tisto, kar dela funkcija, pač kar prepišemo v pogoj.

[28]:
```
[n for n in range(2, 101) if not [x for x in range(2, n) if n % x == 0]]
```

[28]:
```
[2,
 3,
 5,
 7,
 11,
 13,
 17,
 19,
 23,
 29,
 31,
 37,
 41,
 43,
 47,
 53,
 59,
 61,
 67,
 71,
 73,
 79,
 83,
 89,
 97]
```

Takšnih reči sicer raje ne pišemo, saj hitro postanejo nepregledne.

0.3.2 Splošni vzorec

V splošnem: vsak kos kode, ki izgleda takole

```
r = []
for e in s:
    if pogoj(e):
        r.append(izraz)
```

lahko prepišemo v

```
r = [izraz for e in s if pogoj(e)]
```

0.4 Množice

Množice sestavljamo natančno tako kot sezname, le da namesto oglatih oklepajev uporabimo zavite. Takole dobimo vse delitelje 60

[29]:
```
{i for i in range(1, 61) if 60 % i == 0}
```

[29]:
```
{1, 2, 3, 4, 5, 6, 10, 12, 15, 20, 30, 60}
```

0.5 Slovarji

Ista reč. Če hočemo narediti slovar, ki bo kot ključe vseboval števila do 10, kot vrednosti pa njihove kvadrate, napišemo

[30]:
```
{i: i ** 2 for i in range(11)}
```

[30]:
```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81, 10: 100}
```

0.6 Nekaj uporabnih funkcij

Te stvari res zaživijo, če jih uporabljamo v kombinaciji z nekaj funkcijami. Nekatere že poznamo, nekatere bodo postale smiselne šele danes.

0.6.1 `enumerate`

Doslej sem sitnaril, da jo uporabljajmo, kadar potrebujemo tako indeks elementa kot njegovo vrednost. Zdaj bo še toliko uporabnejša, saj bo bistveno izboljšala preglednost izpeljanih seznamov, množic in slovarjev.

Zdaj, ko smo ravno tako pametni, vemo tudi, kako bi napisali `enumerate`, če le-tega še ne bi bilo.

[31]:
```
def enumerate(s):
    i = 0
    for x in s:
        yield i, x
        i += 1

list(enumerate("Ana"))
```

[31]:
```
[(0, 'A'), (1, 'n'), (2, 'a')]
```

0.6.2 `zip`

Tudi za `zip` že vemo: podamo ji nekaj seznamov in vrne nov seznam - seznam terk z elementi seznamov, ki smo jih poslali kot argumente.

No, od danes vemo: v resnici vrne generator parov, ne seznama parov.

[32]:
```
zip("Benjamin", "Margareta")
```

[32]:
```
<zip at 0x7ff2830931e0>
```

Ignorirajte tiste `0x10193c308` (ali karkoli že se prikaže) - izpis je povedal, da smo naredili `zip` in ne seznama. Seveda lahko iz njega "izgeneriramo" seznam.

[33]:
```
list(zip("Benjamin", "Margareta"))
```

[33]:
```
[('B', 'M'),
 ('e', 'a'),
 ('n', 'r'),
 ('j', 'g'),
 ('a', 'a'),
 ('m', 'r'),
 ('i', 'e'),
 ('n', 't')]
```

Napišimo funkcijo, ki pove, v koliko črkah se ujemata dani besedi. Tako se, recimo, MELODIJA in DELODAJALEC ujemata v šestih črkah.

```
MELODIJA
DELODAJALEC
 ELOD JA
```

Da bo naše delo lažje, se najprej spomnimo, da se `True` vede, kot da bi bil `1` in `False`, kot da bi bil 0.

[34]:
```
True + 7
```

[34]:
```
8
```

Vzemimo torehj besedi MELODIJA in DELODAJALEC.

[35]:
```
b1 = "MELODIJA"
b2 = "DELODAJALEC"
```

Združimo ju v seznam parov črk (`list` pa dodamo samo v izpisu, da vidimo, kaj smo v resnici setavili).

[36]:
```
list(zip(b1, b2))
```

[36]:
```
[('M', 'D'),
 ('E', 'E'),
 ('L', 'L'),
 ('O', 'O'),
 ('D', 'D'),
 ('I', 'A'),
 ('J', 'J'),
 ('A', 'A')]
```

Poženimo zanko čez seznam parov črk in sestavimo nov seznam, ki bo vseboval `True`, če sta črki enaki in `False`, če različni.

[37]:
```
[c == d for c, d in zip(b1, b2)]
```

[37]:
```
[False, True, True, True, True, False, True, True]
```

Zdaj pa preštejmo, koliko je `True`-jev. Kako? Če je `False` isto kot 0 in `True` isto kot 1, potem preprosto izračunamo vsoto elementov seznama.

[38]:
```
sum(c == d for c, d in zip(b1, b2))
```

[38]:
```
6
```

Če bi torej hoteli napisati funkcijo, ki pove, v koliko črkah se besedi ujemata, bi rekli kar

[39]:
```
def ujemanj(b1, b2):
    return sum(c==d for c, d in zip(b1, b2))
```

Napišimo funkcijo, ki izračuna Evklidsko razdaljo med dvema točkama, katerih koordinate so predstavljene s seznamom. Imejmo

[40]:
```
a = [2, 1, -3]
b = [3, 5, 4]
```

Nalogo bi včasih rešili takole:

[41]:
```
def euclid(a, b):
    s = 0
    for x, y in zip(a, b):
        s += (x - y) ** 2
    return sqrt(s)
```

Zdaj znamo veliko veliko preprosteje. Znamo sestaviti seznam parov? Znamo.

[42]:
```
[(x, y) for x, y in zip(a, b)]
```

[42]:
```
[(2, 3), (1, 5), (-3, 4)]
```

Znamo sestaviti seznam razlik parov? Znamo.

[43]:
```
[x - y for x, y in zip(a, b)]
```

[43]:
```
[-1, -4, -7]
```

Pravzaprav potrebujemo seznam kvadratov razlik.

[(x - y) ** 2 for x, y in zip(a, b)]

In zdaj moramo vse to le še sešteti.

[44]:
```
sum((x - y) ** 2 for x, y in zip(a, b))
```

[44]:
```
66
```

Funkcija, ki izračuna Evklidsko razdaljo, je torej

[45]:
```
def euclid(a, b):
    return sqrt(sum((x - y) ** 2 for x, y in zip(a, b)))
```

Te reči so, ko se jih navadiš, prav imenitno berljive. Program je skoraj dobesedno prepisan opis v naravnem jeziku: evklidska razdalja je koren vsote kvadratov razlik elementov, ki jih "vzporedno" jemljemo iz dveh seznamov.

0.6.3 `all` in `any`

Funkciji `all` in `any` sta videti trivialno, v resnici pa sta prav imenitni in uporabni. Prva, `all`, prejme generator (ali, seveda, seznam, množico, niz, slovar…) in vrne `True`, če le-ta generira same resnične reči (recimo same `True`).

Število `n` je praštevilo, če so vsi ostanki po deljenju z `i` (za vsak `i` od `2` do `n`) različni od 0. Z `all` se tole prevede skoraj dobesedno v Python.

[46]:
```
def prastevilo(n):
    return all(n % i != 0 for i in range(2, n))
```

Funkcija `any` vrne `True`, če tisto, kar podamo kot argument, vsebuje vsaj kakšno resnično vrednost. Kako deluje, pokažimo kar na obrnjenih praštevilih: število je sestavljeno, če je ostanek po deljenju s kakim drugim številom enak 0.

[47]:
```
def sestavljeno(n):
    return any(n % i == 0 for i in range(2, n))
```

Mimogrede opazimo, da sta `all` in `any` povezana prek de Morganovega pravila, ki se bi ga morali spomniti iz matematike. Število je praštevilo, če ni sestavljeno,

[48]:
```
def prastevilo(n):
    return not sestavljeno(n)
```

Če namesto, da bi poklicali `sestavljeno`, vstavimo kar kodo funkcije `sestavljeno`, dobimo

[49]:
```
def prastevilo(n):
    return not any(n % i == 0 for i in range(2, n))
```

Če le obrnemo pogoj, vidimo, da je

```
not any(n % i == 0 for i in range(2, n))
```

isto kot

```
all(not n % i == 0 for i in range(2, n))
```

0.6.4 `itertools.count`

Še več cukra se nahaja v modulu `itertools`.

Funkcija `count` iz modula `itertools` je kot `range`, ki ji podamo kvečjemu spodnjo mejo ali pa še te ne. `count()` generira vsa števila od 0 do neskončno, `count(n)` pa vsa števila od `n` naprej.

Sam jo najpogosteje uporabljam v zvezi z `zip`. `zip(count(), s)` je isto kot `enumerate(s)`. To sicer ni prav uporabno - očitno je `enumerate(s)` preprosteje in jasneje kot `zip(count(), s)`. Pač pa se dogaja, da greš z `zip` čez tri sezname hkrati, poleg tega pa potrebuješ še indekse. To se da pisati kot

```
for i, (x, y, z) in enumerate(zip(s, t, u)):
```

lahko pa pišemo kar

```
for i, x, y, z in zip(count(), s, t, u)
```

Pa še kje vam bo prišel prav, če se spomnite nanj.

0.6.5 `itertools.chain`

Če imamo več generatorjev (ali seznamov, slovarjev…) in bi radi šli z eno zanko čez vse, torej čez enega za drugim, jih sestavimo s `chain`.

[50]:
```
from itertools import chain

for c in chain("Ana", "Berta", "Cilka"):
    print(c)
```

```
A
n
a
B
e
r
t
a
C
i
l
k
a
```

Seveda je to isto kot `for c in "Ana" + "Berta" + "Cilka":`, vendar je lepota v tem, da nizov nismo sešteli, temveč je `chain`. To je lepo predvsem, kadar ne verižimo nizov, temveč kaj, česar se ne da seštevati.

0.6.6 Itertools

Vsi, ki vam je bilo tole všeč, bodo uživali tudi v ostalih dobrotah iz modula `itertools`.