

Tule je še nekaj stvari, ki sicer niso nič posebnega, le na predavanju ni bilo časa zanje.

Generator z dvema zankama

Recimo, da bi kdo iz nekega neznanega razloga hotel napisati takšen program.

```
tipi = ["Anže", "Benjamin", "Cene"]
tipinje = ["Ana", "Berta", "Cilka", "Dani"]
```

```
pari = []
for tip in tipi:
    for tipinja in tipinje:
        pari.append((tip, tipinja))
print(pari)
```

```
[('Anže', 'Ana'), ('Anže', 'Berta'), ('Anže', 'Cilka'), ('Anže', 'Dani'), ('Benjamin', 'Ana'),
```

Če hočeš to napisati z izpeljanim seznamom, tega **ne narediš** tako:

```
print([(x, y) for y in tipinje] for x in tipi])
```

```
[('Anže', 'Ana'), ('Anže', 'Berta'), ('Anže', 'Cilka'), ('Anže', 'Dani')], [('Benjamin', 'Ana'), ('Benjamin', 'Berta'), ('Benjamin', 'Cilka'), ('Benjamin', 'Dani')], [('Cene', 'Ana'), ('Cene', 'Berta'), ('Cene', 'Cilka'), ('Cene', 'Dani')]]
```

To namreč vrne seznam, ki vsebuje sezname.

Izpeljani seznam lahko vsebujejo tudi več zank. Pišemo jih z leve proti desni, najprej zunanjo.

```
[(tip, tipinja) for tip in tipi for tipinja in tipinje]
```

```
[('Anže', 'Ana'),
 ('Anže', 'Berta'),
 ('Anže', 'Cilka'),
 ('Anže', 'Dani'),
 ('Benjamin', 'Ana'),
 ('Benjamin', 'Berta'),
 ('Benjamin', 'Cilka'),
 ('Benjamin', 'Dani'),
 ('Cene', 'Ana'),
 ('Cene', 'Berta'),
 ('Cene', 'Cilka'),
 ('Cene', 'Dani')]
```

Seveda je dovoljen tudi if.

```
[(tip, tipinja) for tip in tipi for tipinja in tipinje if tip[0] < tipinja[0]]
```

```
[('Anže', 'Berta'),
 ('Anže', 'Cilka'),
 ('Anže', 'Dani'),
 ('Benjamin', 'Cilka'),
```

```
('Benjamin', 'Dani'),  
( 'Cene', 'Dani')]
```

Generator za generatorjem

Zgodi pa se tudi - ali pa se zdaj dogaja prvič - da bi kdo hotel napisati takšen program.

```
stevila = "5 9 4 12"
```

```
s = []  
for stevilo in stevila.split():  
    x = int(stevilo)  
    s.append(x * (x + 1))
```

```
print(s)
```

```
[30, 90, 20, 156]
```

Če bi se mu zahotelo to napisati z izpeljanim seznamom, bi bil navidez prisiljen storiti tako.

```
[int(stevilo) * (int(stevilo) + 1) for stevilo in stevila.split()]  
[30, 90, 20, 156]
```

Pri tem bi ga upravičeno iritiralo, da mora dvakrat poklicati `int`.

Morda bi kdo pripomnil, da lahko uporabimo Walrusa.

```
[(x := int(stevilo)) * (x + 1) for stevilo in stevila.split()]  
[30, 90, 20, 156]
```

Kaj pa vem. Mogoče je prav to eden od primerov, kjer ga nočem videti. Zato ga tule niti ne bom razlagal. Prava rešitev je takšna:

```
[x * (x + 1) for x in (int(stevilo) for stevilo in stevila.split())]  
[30, 90, 20, 156]
```

Pravzaprav ne, prava rešitev tule je uporabiti `map`.

```
[x * (x + 1) for x in map(int, stevila.split())]  
[30, 90, 20, 156]
```

Vendar se o `map`-u nismo kaj prida pogovarjali, namen tega besedila pa je povedati, da lahko v primeru, da je potrebno pripraviti neko vrednost tako, da pokličemo funkcijo (tule le `int`, lahko pa gre za kaj bolj zapletenega), to storimo tako, da generator spustimo čez drug generator).

Zanka prek seznama, ki nastaja

Zanka prek slovarja, v katerega znotraj zanke dodajamo elemente ali pa jih brišemo, je prepovedana. Če poskušamo kaj takšnega, bo Python javil napako.

Zanka prek seznama, ki spreminja dolžino znotraj zanke, je običajno slaba ideja. Oziroma, je skoraj gotovo slaba ideja, če dodajamo ali brišemo elemente pred trenutno pozicijo. V nekaterih situacijah pa zna biti dobra ideja, če seznam dopolnjujemo, medtem ko gremo čezenj.

```
s = [1]
for x in s:
    x *= 2
    if x > 100:
        break
    s.append(x)
```

s

[1, 2, 4, 8, 16, 32, 64]

Ni, da se ne bi dalo živeti brez tega, včasih pa pride prav.

Razpakiranje seznamov, slovarjev, terk, ... zaporedij

Če imamo seznam `s`, bo `*s` (kot unarni operator, ne kot množenje) v vseh kontekstih, v katerih to smemo napisati, pomenilo isto kot `s[0]`, `s[1]`, `s[2]`, ... in tako do konca. To velja tudi za terke. In tudi za slovarje in množice in karkoli, čez kar lahko gremo z zanko `for` (ali bolj učeno, za poljubno zaporedje `s`), le da tam seveda ne moremo pisati o `s[0]` in tako naprej.

```
def f(x, y, z, u):
    print(x, y, z, u)
```

```
a = [1, 2]
b = (3, 4)
d = {1, 2, 3}
```

```
f(0, *a, 4)
```

```
0 1 2 4
```

```
f(*a, *b)
```

```
1 2 3 4
```

```
f(14, *d)
```

```
14 1 2 3
```

```
f(*range(3), 5)
```

```
0 1 2 5
```

Pa tudi:

```
[1, 2, 3, *a, 5, 6, *range(10, 12), *d]
[1, 2, 3, 1, 2, 5, 6, 10, 11, 1, 2, 3]
{*a, *b}
{1, 2, 3, 4}
```

Veriženje

Če imamo več generatorjev (ali seznamov, terk ipd) in bi radi šli z zanko prek njih, enega za drugim, uporabimo `chain` iz modula `itertools`.

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

from itertools import chain

for x in chain(s, t, range(7, 10)):
    print(x)

1
2
3
4
5
6
7
8
9

for x in chain((x ** 2 for x in range(3)), (x ** 3 for x in range(3))):
    print(x)

0
1
4
0
1
8
```

Veliki hack: veriženje z razpakiranjem

Imejmo seznam seznam seznamov. Ali pa, recimo, množico terk. Skratka, imejmo nekaj, česar elementi vsebujejo elemente. Nekaj gnezdenega.

```
s = [(1, 2, 3), (4, 5), (6, 7, 8, 9)]

for x in chain(*s):
    print(x)
```

```
1  
2  
3  
4  
5  
6  
7  
8  
9
```

```
list(chain(*s))
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Razumemo? `chain`-u podamo `*s`, torej kot argument dobi vse, kar vsebuje `s`.

```
chain(*s)
```

je v gornjem primeru isto kot

```
chain((1, 2, 3), (4, 5), (6, 7, 8, 9))
```

`chain` bo izgeneriral vse elemente teh terk.

V jezikih, ki imajo `flatten`, se temu reče `flatten`. :)