03a seznami, terke in zanka for

January 28, 2024

Na prvih dveh predavanjih smo se pogovarjali predvsem o logiki, o tem, kako "usmerjati" program, da se bo vrtel in skakal. Mimogrede smo spoznali tri tipe podatkov, cela števila (`int`), realna števila (`float`), logične vrednosti (`bool`) in nizi (`str`).

Za te, ki že znate programirati: na predavanju bomo spoznali sezname, vendar bom karseda dolgo zamolčal, da lahko do njihovih elementov dostopamo z indeksi. Namerno. Ideja je v tem, da prej spoznate, kako se do njih dostopa z zanko in na ta način povečamo možnost, da ne boste prevečkrat uporabljali indeksov takrat, ko jih ni treba.

0.0.1 Seznami

Recimo, da smo zbrali teže šestih študentov. Seznam tež zapišemo takole:

[ ]:
```
teze = [74, 82, 58, 66, 61, 84]
```

Seznam (angl. *list*) je zaporedje česarkoli, recimo števil, lahko pa tudi česa drugega. Števila ali kaj drugega naštejemo, vmes pišemo vejice in vse skupaj zapremo v oglate oklepaje (`[` in `]`). Za primer sestavimo še seznam imen študentov:

[ ]:
```
imena = ["Anze", "Benjamin", "Cilka", "Dani", "Eva", "Franc"]
```

in seznam, ki bo povedal, ali gre za študenta ali študentko

[ ]:
```
studentka = [False, False, True, False, True, False]
```

Seznami lahko vsebujejo tudi še hujšo eksotiko. Imamo lahko, recimo, seznam seznamov - vanj bomo, prikladno, stlačili (pod)sezname teža-ime-spol:

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

To sicer navadno delamo malo drugače (pogosto celo precej drugače), a dokler tega še ne znamo, bo dobro tudi tako. Mimogrede opazimo še dve stvari: seznam lahko, če želimo, raztegnemo v več vrst, preprosto tako, da ne zapremo oklepaja. Naredili bi lahko tudi tole

[ ]:
```
teze = [74, 82, 58,
        66, 61, 84]
```

V takih primerih je v naslednjih vrsticah vseeno, koliko presledkov damo na začetek. Toliko pač, da dobro izgleda. (Se spomnite, ko smo na čisto prvem predavanju povedali, da python ve, da izraza še ni konec, če še niso zaprti vsi oklepaji? Zdaj vidimo, zakaj je to prikladno.)

Druga opazka: za zadnjim elementom v prejšnjem primeru smo naredili vejico. Ni treba, lahko pa jo. To pride prav, če bi kdaj kasneje spreminjali vrstni red elementov, na primer dali Franca na čelo vrste. V tem primeru bi le skopirali vrstico, vejice pa bi bile že OK.

Seznami lahko vsebujejo še hujšo eksotiko. V resnici lahko vsebujejo karkoli, celo, recimo, funkcije:

[ ]:
```
from math import *
par_funkcij = [sin, cos, radians]
```

Čemu bi kdo hotel narediti seznam funkcij?! Ne boste verjeli, kolikokrat pride to prav!

Nikjer tudi ne piše, da morajo biti vsi elementi seznama istega tipa. To smo pravzaprav že izkoristili: podseznami s težo, imenom in spolom so vsebovali število, niz in logično vrednost. Navrgli bi lahko še dve funkciji in en seznam; pa ne bomo, to se ne dela. Seznamov, ki imajo elemente različnih tipov, ne bomo tako pogosto videvali. (Če obratno parafraziramo Andreja Bauerja: *seznami z elementi različnih tipov se morda komu zdijo dobra ideja. V resnici je to zelo sla ba ideja.* Če želimo sezname različnih tipov, raje uporabimo terke. Kaj so terke, bomo izvedeli vsak čas.)

Seznam je lahko tudi prazen

[ ]:
```
prazen = []
```

ali pa ima en sam element

[ ]:
```
samo_edini = [42]
```

Ali pa je poln praznih seznamov

[ ]:
```
polnoprazen = [[], [], [], [], [], []]
```

0.0.2 Terka

Terka (angl. *tuple*) je podobna seznamu. Kakšna je pomembna razlika, bomo še videli, manj pomembna pa je ta, da pri terki namesto oglatih oklepajev uporabljamo kar okrogle. (Skoraj) vse ostalo ostane enako:

[ ]:
```
teze = (74, 82, 58, 66, 61, 84)
imena = ("Anze", "Benjamin", "Cilka", "Dani", "Eva", "Franc")
```

Pri terkah se nam - zaradi tega, kje in kako jih uporabljamo, pogosteje zgodi, da vsebuje elemente različnih tipov. Recimo takole:

[ ]:
```
student = (74, "Anze", False)
```

Tudi seznam teža-ime-spol bi raje naredili takole:

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

Tako kot seznami so tudi terke lahko prazne (`()`) ali pa imajo po en sam element. Tu imamo težavo pri terki z enim samim elementom: če bi napisali `(42)` to ni seznam, temveč le 42 v oklepaju (odvečnem, seveda). Da bi povedali, da gre za terko, moramo dodati še vejico:

[ ]:
```
samo_en = (42, )
```

Terko smemo, zanimivo, pisati tudi brez oklepajev.

[ ]:
```
>>> t = 1, 2, 3, 4
>>> t
(1, 2, 3, 4)
```

Ker to ni preveč pregledno, pa to počnemo le v posebnih primerih, na katere bomo sproti opozorili.

0.0.3 Razpakiranje v elemente

Kako pridemo do elementov seznama ali terke? Se pravi, kako bi shranili elemente terke v posamezne spremenljivke? Recimo, da imamo

student = (74, "Anze", False)

Če želimo podatke prirediti spremenljivkam `teza`, `ime` in `je_zenska`, terko *razpakiramo*:

[ ]:
```
teza, ime, je_zenska = student
```

Doslej smo imeli pri prirejanju na levi strani vedno eno samo spremenljivko, ki smo ji želeli prirediti vrednost na desni strani enačaja. Tu pa smo na levi našteli več imen spremenljivk (tri), na desni pa mora biti za to terka z enakim številom elementov (tremi). Terka? No, lahko je tudi seznam ali niz.

[ ]:
```
>>> a, b, c = [1, 2, 3]
>>> a
1
>>> a, b, c = "xyz"
>>> a
'x'
>>> b
'y'
>>> c 'z'
```

Več ko boste programirali v Pythonu, bolj boste čutili moč terk. Omenimo, recimo, da je z njimi mogoče napisati funkcijo, ki ne vrača ene, temveč več stvari. Vzemimo funkcijo `splitext`, ki ji damo ime datoteke, pa vrne njeno osnovno ime in njeno končnico.

[ ]:
```
>>> from os.path import *
>>> film = "Babylon 5 - 3x04 - Passing through Gethsemane.avi"
>>> zac, konc = splitext(film)
>>> zac
'Babylon 5 - 3x04 - Passing through Gethsemane'
>>> konc
'.avi'
```

V resnici funkcija `splitext` vrne terko,

[ ]:
```
>>> from os.path import *
>>> splitext(film)
('Babylon 5 - 3x04 - Passing through Gethsemane', '.avi')
```

ki pa smo jo kar mimogrede razpakirali v `zac` in `konc`.

Seveda bi lahko rezultat funkcije priredili tudi eni sami spremenljivki - v tem primeru bi bila ta spremenljivka terka.

[ ]:
```
>>> t = splitext(film)
>>> t
('Babylon 5 - 3x04 - Passing through Gethsemane', '.avi')
```

In seveda bi lahko terko nato še razpakirali.

[ ]:
```
ime, koncnica = t
```

Kako v Pythonu zamenjamo vrednosti dveh spremenljivk?

[ ]:
```
>>> a = "Alenka"
>>> b = "Tina"
```

Želeli bi, da bi `a` postal "Tina" in `b` "Alenka". Naivnež bi napisal tale nesmisel:

[ ]:
```
>>> a = b
>>> b = a
```

To seveda ne deluje. V prvi vrstici `a`-ju priredimo `b`, se pravi, "Tina", v drugi vrstici pa `b`-ju `a`, ki pa je medtem postal "Tina". Tako sta po tem `a` in `b` enaka "Tina".

V drugih jezikih se navadno rešimo tako, da vrednost `a`-ja spravimo na varno, preden ga povozimo.

[ ]:
```
>>> tmp = a
>>> a = b
>>> b = tmp
```

V Pythonu pa to ni potrebno. Spremenljivki preprosto zapakiramo v terko, ki jo takoj razpakiramo nazaj, a v obratnem vrstnem redu.

[ ]:
```
>>> a, b = (b, a)
```

V resnici navadno ne pišemo tako, temveč izpustimo oklepaje.

[ ]:
```
>>> a, b = b, a
```

Oklepaje okrog terke izpuščamo preprosto zato, ker je takšno prirejanje tako pogosto, da ga programer vajen Pythona takoj prepozna. Pri tem tudi ne razmišljamo o terkah, temveč prirejanje preberemo preprosto tako, da `a`-ju priredimo `b` in `b`-ju `a`.

0.0.4 Zanka for

Python ima dve vrsti zank: zanki `while`, ki jo že poznamo, dela družbo `for`.

(Medklic za tiste, ki že znate programirati)

V C-ju in jezikih, izpeljanih iz njega (C++, C#, Java…) imamo običajno dve vrsti zank. Mogoče mislite, da so tri, ampak v resnici sta dve: `while` in `do-while`. C-jevski `for` je samo malo bolj zgoščeno zapisan `while`.

```
for(zacetek; pogoj; korak) {
    koda;
}
```

je isto kot

```
zacetek;
while(pogoj) {
    koda;
    korak;
}
```

V C, C++, C#, Javi sta zanka `for` in `while` le dva načina, na katera povemo eno in isto. Pa tudi med `while` in `do-while` ni tako velike razlike, čeprav je slednja včasih - a redko - praktična.

Pač pa obstajajo tudi [druge različice zanke for (http://en.wikipedia.org/wiki/For_loop#Kinds_of_for_loops), ki so v resnici drugačne od `while`. Novejši jeziki imajo - in starejši jeziki dobivajo - drugačno obliko zanke `for`, ki je v današnjih časih, ko vedno več uporabljamo funkcijske jezike, bolj uporabna. Nekateri, recimo C#, jo poznajo pod imenom `foreach`. Pythonov `for` vas bo spominjal nanj in ne na C-jevski `for`.

(Konec medklica)

Zastavimo si preprosto nalogo (in ne čisto smiselno) nalogo: izpišimo teže in kvadrate tež vseh študentov v seznamu. Se pravi (po slovensko):

```
za vsako težo s seznama teze stori tole:
    izpiši težo in težo na kvadrat
```

V Pythonu pa prav tako, samo brez sklonov:

[ ]:
```
for teza in teze:
    print(teza, teza ** 2)
```

V zanki `for` se skriva prirejanje. Zanka najprej *priredi* spremenljivki `teza` prvi element seznama `teze`. Nato se izvrši vsa koda bloka znotraj bloka `for` - tako, kot se je dogajalo v zanki `while`. V naslednji rundi priredi spremenljivki `teza` drugi element in spet izvede kodo znotraj bloka, … ter tako naprej do konca seznama.

Zanka `for` je torej nekoliko podobna zanki `while`. Razlika je v tem, - da se `while` izvede tolikokrat, dokler ji dopušča pogoj, `for` pa se izvede enkrat za vsak element seznama, - da `for` v vsaki ponovitvi (po tuje: *iteraciji*) priredi spremenljivki vrednost naslednjega elementa seznama.

V zvezi z obojim je potrebno še nekaj pripomniti, da nas ne bodo držali za besedo, češ da učimo polresnice. Zanka `for` ne gre nujno le prek seznamov, temveč tudi prek terk, nizov in še drugih reči, med katerimi so tudi kake tako imenitne, da si jih zdajle ne moremo niti predstavljati.

Glede *prirejanja vrednosti spremenljivkam* pa se moramo popraviti, da Python ne prireja vrednosti spremenljivkam, temveč daje imena rečem. Za tiste, ki jim ta razlika kaj pomeni. Tistim, ki jim ne, pa še bo.

Zdaj pa izpišimo vse teže, ki so večje od 70.

[ ]:
```
for teza in teze:
    if teza > 70:
        print(teza)
```

Napišimo program, ki pove, kako težak je najlažji študent.

[ ]:
```
najlazji = 1000
for teza in teze:
    if teza < najlazji:
        najlazji = teza
print(najlazji)
```

Je potrebno prevesti v slovenščino? Pa dajmo.

```
za začetek naj bo najlažja teža 1000 (ker vemo, da je to preveč)
za vsako težo iz seznama tež:
    če je teža manjša od najmanjše  doslej:
        najlažja je ta teža
izpiši najmanjšo težo
```

Deluje ta program samo na seznamih števil? Ali pa bi znal poiskati tudi najmanjši niz? Kako pravzaprav primerjamo nize? Nize primerja, seveda, po abecedi. In, da: program deluje tudi na nizih, vrne "najmanjši" niz - prvi niz po abecedi. Le v začetku moramo napisati `najlazji = "ŽŽŽŽŽŽŽŽŽŽŽ"` namesto `najlazji = 1000`. (Tole se da sprogramirati tudi tako, da deluje za nize in za števila. A ne ubijajmo začetnikov s prezapletenimi programi.)

Mimogrede, Python ima že vdelano funkcijo `min`, ki vrne najmanjši element seznama.

Kako bi izračunali vsoto elementov seznama?

[ ]:
```
s = 0
for teza in teze:
    s += teza
```

Pa poprečno vrednost? Za to moramo poznati dolžino seznama. Pove nam jo funkcija `len` (okrajšava za *length*, če ji kot argument podamo nek seznam).

[ ]:
```
s = 0
for teza in teze:
    s += teza
s /= len(teze)
print(s)
```

Samo… malo previdni moramo biti. Seznam bi lahko bil tudi prazen. Dogovorimo se, da bo poprečna vrednost v tem primeru 0. Pravilno delujoč program bi bil takšen.

[ ]:
```
s = 0
for teza in teze:
    s += teza
if len(teze) > 0:
    s /= len(teze)
print(s)
```

Zanka `for` ne deluje le na seznamih. Kot smo omenili, jo lahko naženemo tudi prek terk, nizov in še prek mnogih drugih reči. V Pythonu celo za branje datotek pogosto uporabimo `for`, kot se bomo kmalu naučili.

[ ]:
```
>>> ime = "Cilka"
>>> for crka in ime:
...     print(crka)
'C' 'i' 'l' 'k' 'a'
```

0.0.5 Najdi takšnega, ki...

Zdajle, bolj proti začetku ure, ko smo še sveži, brž naskočimo najtežji oreh današnjega predavanja. Kar bomo počeli zdaj, boste srečali v sto in eni preobleki, na koncu pa skoraj gotovo tudi kot najpreprostejšo nalogo na izpitu.

Kako bi ugotovili, ali vsebuje seznam kako sodo število?

[ ]:
```
s = [11, 13, 5, 12, 5, 16, 7]
imamo_sodo = False
for e in s:
    if e % 2 == 0:
        imamo_sodo = True

if imamo_sodo:
    print("Seznam vsebuje sodo število")
else:
```

```
    print("Seznam ne vsebuje sodega števila")
```

V začetku si rečemo, da nimamo nobenega sodega števila. Nato gremo prek vseh števil in čim naletimo na kakega sodega, zabeležimo, da smo, aleluja, našli sodo število.

(

Kak duhovitež bi gotovo raje napisal

[ ]:
```
ostanki = 0
for e in s:
    ostanki += e % 2
if ostanki != len(e):
    ...
```

in nam tako pokvaril lekcijo. Ignorirajmo ga. :)

)

Ne naredite klasične napake. Tole je narobe:

[ ]:
```
s = [11, 13, 5, 12, 5, 16, 7]
imamo_sodo = False
for e in s:
    if e % 2 == 0:
        imamo_sodo = True
    else:
        imamo_sodo = False

if imamo_sodo:
    print("Seznam vsebuje sodo število")
else:
    print("Seznam ne vsebuje sodega števila")
```

Ta program se bo ob vsakem lihem številu delal, da doslej ni bilo še nobenega sodega - zaradi `imamo_sodo = False` bo pozabil, da je kdajkoli videl kako sodo število. V gornjem seznamu bo, recimo, pregledal vsa števila, končal bo s 7 in si ob tem rekel `imamo_sodo = False`. Rezultat bo tako napačen.

Kako pa ugotovimo, ali ima seznam *sama soda števila*?

[ ]:
```
sama_soda = True
for e in s:
    if e % 2 != 0:
        sama_soda = False

if sama_soda:
    print("Seznam vsebuje sama soda število")
else:
    print("Seznam ne vsebuje samo sodih števil")
```

(

Duhovitež ne lenari in napiše:

[ ]:
```
ostanki = 0
for e in s:
    ostanki += e % 2
if ostanki == 0:
    ...
```

)

Spet bomo naredili podobno napako kot prej, če bomo pisali:

[ ]:
```
s = [11, 13, 5, 12, 5, 16, 72]
sama_soda = True
for e in s:
    if e % 2 != 0:
        sama_soda = False
    else:
        sama_soda = True
```

Program že takoj, ko vidi 1, ugotovi, da niso vsa števila v seznamu soda, in postavi `sama_soda = False`. Vendar melje seznam naprej in ob vsakem koraku znova nastavlja `sama_soda`. Ko pride do 72, postavi `sama_soda` na `True`. To pa je ravno zadnje število; `sama_soda` ostane `True`… pa smo tam.

0.0.6 Prekinjanje zank

Ko smo prejšnji teden pisali zanko `while`, smo postavili pogoj, do kdaj naj se izvaja. Zanka for bo, če se vmes ne pripeti kaj posebnega, šla vedno od začetka do konca seznama (terke, niza, datoteke…)

Včasih to ni potrebno. Pravzaprav smo pravkar videli takšen primer: rezultat gornjega programa je znan že, čim naletimo na prvo liho število. Torej bi bilo čisto vseeno, če računalnik v tistem trenutku konča pregledovanje. To mu v resnici lahko naročimo.

[ ]:
```
sama_soda = True
for e in s:
    if e % 2 != 0:
        sama_soda = False
        break

if sama_soda:
    print("Seznam vsebuje sama soda število")
else:
    print("Seznam ne vsebuje samo sodih števil")
```

Ukaz `break` pomeni, da želimo prekiniti zanko. Program skoči “ven” iz zanke in nadaljuje z izvajanjem ukazov, ki sledijo zanki.

Na podoben način lahko prekinemo tudi `while` - kot ste nekateri uspešno počeli že prejšnji teden.

Tule je bil `break` bolj zaradi lepšega - da računalnik ne izgublja časa brez potrebe, češ elektrika je draga, pa globalno segrevanje pa te stvari. Poskusimo napisati program, ki takrat, ko seznam ni vseboval samo sodih števil, izpiše prvo liho število.

[ ]:
```
sama_soda = True
for e in s:
    if e % 2 != 0:
        sama_soda = False
        liho = e

if sama_soda:
    print("Seznam vsebuje samo soda števila")
else:
    print("Seznam ne vsebuje samo sodih števil:",
          " prvo liho število je", liho)
```

Brez `break` tole ne deluje: namesto prvega izpiše zadnje liho število. Rešimo se lahko z dodatnim pogojem:

[ ]:
```
sama_soda = True
for e in s:
    if e % 2 != 0 and sama_soda:
        sama_soda = False
        liho = e

if sama_soda:
    print("Seznam vsebuje samo soda števila")
else:
    print("Seznam ne vsebuje samo sodih števil:",
          " prvo liho število je", liho)
```

S tem, ko smo dodali `and sama_soda` smo poskrbeli, da se bosta `sama_soda = False` in `liho = e` izvedla le prvič. Ko bomo postavili `sama_soda` na `False`, bomo dosegli, da pogoj ne bo nikoli nikoli več resničen.

(Mimogrede opozorimo: napisali smo `and sama_soda` in ne `and sama_soda == True`. Rezultat izraza `sama_soda == True` je enak `True` natančno takrat, ko je `sama_soda` enak `True`. Torej tisti `== True` ne naredi ničesar. Če kdo ne razume, naj raje razmisli, ali bi bilo smiselno pisati `(sama_soda == True) == True)` ali celo `((sama_soda == True) == True) == True`. Če meni, da ne, potem naj tudi `sama_soda == True` ne piše, saj je enak nesmiselno.)

Namesto dodatnega pogoja lahko napišemo `break`.

[ ]:
```
sama_soda = True
for e in s:
    if e % 2 != 0:
        sama_soda = False
        liho = e
```

```
        break

if sama_soda:
    print("Seznam vsebuje samo soda števila")
else:
    print("Seznam ne vsebuje samo sodih števil:",
          " prvo liho število je", liho)
```

S tem se program lahko pravzaprav še bolj poenostavi. Ko zanko prekinemo, `e` ostane, kar je bil. Torej ne potrebujemo več spremenljivke `liho`.

[ ]:
```
sama_soda = True
for e in s:
    if e % 2 != 0:
        sama_soda = False
        break

if sama_soda:
    print("Seznam vsebuje samo soda števila")
else:
    print("Seznam ne vsebuje samo sodih števil:",
          " prvo liho število je", e)
```

0.1 Else po zanki

Medtem, ko je ukaz `break` zelo običajna žival v vseh programskih jezikih, ima Python še eno posebnost, povezano z zankami. V večini programskih jezikov lahko `else` uporabimo le kot alternativo `if`-u. V Pythonu pa lahko `else` sledi tudi zanki `for` ali `while`. V pogojnem stavku (`if`) se koda v `else` izvede, če pogoj ni bil resničen. Po zanki se `else` izvede, če se zanka *ni prekinila* zaradi `break`. Torej, `else` se izvede pri zankah, ki so se iztekle "po naravni poti".

Oglejmo si še enkrat gornji program. Kar smo počeli v njem, je kar pogosto: v zanki nekaj iščemo (recimo kako liho število). Če to reč najdemo, nekaj storimo (ga izpišemo) in prekinemo zanko. Sicer (torej, če ga ne najdemo), storimo kaj drugega (izpišemo, da ga nismo našli).

Zgornji odstavek se približno, a ne čisto povsem prilega zadnjemu kosu programa. V resnici je napisan po spodnjem programu:

[ ]:
```
for e in s:
    if e % 2 != 0:
        print("Seznam ne vsebuje samo sodih števil:",
              "prvo liho število je", e)
        break
else:
    print("Seznam vsebuje samo soda števila")
```

Se pravi: če najdemo liho število, napišemo, da seznam vsebuje liho število in prekinemo zanko. Ta del je jasen. Ne spreglejte pa, kje je `else`: poravnan je s `for` ne z `if`! Ta `else` se torej nanaša na `for`. Kar napišemo v `else`-u za `for`, se zgodi, če se zanka ni prekinila z `break`om.

0.1.1 Razpakiranje v zanki `for` in še malo telovadbe

Nekoč na začetku predavanja smo imeli seznam študentov in njihovih tež:

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

Recimo, da bi radi izpisali imena študentov in njihove teže.

Če gremo prek seznama z zanko `for`, bomo dobivali terke. Te lahko, vemo, razpakiramo.

[ ]:
```
for student in podatki:
    teza, ime, spol = student
    print(ime, ": ", teza)
```

Gre pa še dosti elegantneje: razpakiranje lahko opravimo kar znotraj glave zanke, brez (nepotrebne) terke `student`:

[ ]:
```
for teza, ime, je_zenska in podatki:
    print(ime, ": ", teza)
```

Program torej pravi

```
za vsako trojko (teza, ime, je_zenska) iz seznama podatki:
    izpisi ime in tezo
```

Lahko pa se tudi malo igramo in izrišemo graf:

[ ]:
```
for teza, ime, je_zenska in podatki:
    print(ime + " " + "*" * teza)
```

Skoraj, ampak ne čisto. Vse skupaj bi radi še poravnali. O tem, kako se v resnici oblikuje izpis, se bomo pogovarjali drugič, danes pa bomo s tem opravili nekoliko po domače. Predpostavimo, da so imena dolga največ 15 znakov. Pred vsako ime bomo dopisali toliko presledkov, da bo skupna dolžina enaka 15. (Mimogrede bom izdal, da tudi dolžino niza, ne le seznamov, dobimo s funkcijo `len`.)

[ ]:
```
for teza, ime, je_zenska in podatki:
    print(" " * (15 - len(ime)) + ime + " " + "*" * teza)
```

0.1.2 Zanka po dveh seznamih

Zdaj pa izpišimo imena in teže, pri tem, da se le-te nahajajo v ločenih seznamih.

[ ]:
```
teze = [74, 82, 58, 66, 61, 84]
imena = ["Anze", "Benjamin", "Cilka", "Dani", "Eva", "Franc"]
studentka = [False, False, True, False, True, False]
```

Naivno bi se lotili takole

[ ]:
```
for ime in imena:
    print(ime, ":", ????
```

in potem obstali, ker na tem mestu ne moremo priti do teže študenta s tem imenom. Do običajne in zelo zelo zelo napačne rešitve nas pripelje ta zgrešeni premislek: zanko moramo speljati prek imen in prek tež, torej

[ ]:
```
for ime in imena:
    for teza in teze:
        print(ime, ": ", teza)
```

Rezultat je nepričakovan za vse tiste, ki ga niso pričakovali. Program namreč izpiše

[ ]:
```
Anze : 74
Anze : 82
Anze : 58
Anze : 66
Anze : 61
Anze : 84
Benjamin : 74
Benjamin : 82
Benjamin : 58
Benjamin : 66
Benjamin : 61
Benjamin : 84
Cilka : 74
Cilka : 82
... in tako naprej
```

Računalniki imajo to nadležno navado, da vedno naredijo natanko tisto, kar jim naročimo. In v tem primeru smo mu naročili tole:

```
za vsako ime na seznamu ime stori tole:
    za vsako tezo na seznamu tez stori tole:
        izpisi ime in tezo
```

Kdor ne razume, kaj se je zgodilo in zakaj, naj to nujno premisli, da se ne bo kasneje učil na lastnih napakah.

Ko bo to opravljeno, bo razumel: potrebujemo samo eno zanko, ki gre istočasno po obeh seznamih. Aha, takole?

[ ]:
```
for ime in imena:
for teza in teze:
```

```
    print(ime, ": ", teza)
```

Ummm, ne. Ena zanka je ena zanka. To sta dve.

Nekaterim študentom so tule prišli na misel indeksi. (Nekateri zanje ne vejo, kar je dobro zanje. Odstavek lahko preskočijo.) Ta rešitev je slaba. V modernih jezikih razmišljamo na malo drugačen način, zato se ji, če se le da, izognemo, saj vodi v manj pregledne, manj učinkovite, slabše programe.

To je tako pomembno, da bom kar še enkrat ponovil: tega **ne rešujemo z** `for i in range(len(imena)):`.

Funkcija `zip` združi dva seznama (ali več seznamov) v seznam terk.

[ ]:
```
>>> zip(teze, imena)
[(74, 'Anze'), (82, 'Benjamin'), (58, 'Cilka'), (66, 'Daniel'), (61, 'Eva'),␣
 ↪(84, 'Franc')]
>>> zip(teze, imena, studentka)
[(74, 'Anze', False), (82, 'Benjamin', False), (58, 'Cilka', True),
(66, 'Daniel', False), (61, 'Eva', True), (84, 'Franc', False)]
>>> zip(teze, imena, studentka, teze)
[(74, 'Anze', False, 74), (82, 'Benjamin', False, 82),
(58, 'Cilka', True, 58), (66, 'Daniel', False, 66),
(61, 'Eva', True, 61), (84, 'Franc', False, 84)]
```

Opomba: tule goljufam. Tak izpis dobimo v starejših Pythonih. Od različice 3.0 naprej si lahko le predstavljamo, da `zip` dela takole. V resnici naredi nekaj majčkeno drugačnega (kaj, je za nas ta mesec še prezapleteno), rezultat pa je za nas (ta mesec) isti.

Zipamo lahko vse, prek česar lahko naženemo zanko `for`. Torej ne le seznamov, temveč tudi nize, terke in še kaj.

[ ]:
```
>>> zip("abcd", "ABCD")
[('a', 'A'), ('b', 'B'), ('c', 'C'), ('d', 'D')]
```

Rešimo torej nalogo: izpišimo imena in teže iz ločenih seznamov.

[ ]:
```
for ime, teza in zip(imena, teze):
    print(ime, ": ", teza)
```

Za konec pa izpišimo še poprečno težo študentk.

[ ]:
```
skupna_teza = 0
studentk = 0
for teza, je_zenska in zip(teze, studentka):
    if je_zenska:
        skupna_teza += teza
        studentk += 1
print(skupna_teza / studentk)  # ... ob predpostavki, da studentk > 0
```

(Tisti, ki so prejšnjo uro slišali tale trik, ne potrebujejo predpostavke, da imamo kakšno študentko, saj napišejo `skupna_teza / (studentk or 1)`.)

0.1.3 Z zip-om prek parov zaporednih elementov

Recimo, da imam seznam in bi rad naredil seznam razlik zaporednih elementov. Torej, če imam seznam `s = [5, 3, 8, 2, 1]`, bi rad pridelal seznam `[2, -5, 6, 1]` (ker je 5 - 3 = 2, 3 - 8 = -6, 8 - 2 = 6, 2 - 1 = 1).

Neučakani naredijo tole:

[ ]:
```
t = []
for i in range(1, len(s)):
    t.append(s[i - 1] - s[i])</xmp>
```

To je pravilno, ni pa elegantno in ne vodi daleč. Obstaja boljši način.

Vemo, da `s[1:]` odbije prvi element seznama. Imamo torej

```
>>> s
[5, 3, 8, 2, 1]
>>> s[1:]
[3, 8, 2, 1]
```

Seznama zazipamo.

```
>>> zip(s, s[1:])
[(5, 3), (3, 8), (8, 2), (2, 1)]
```

Dobili smo seznam parov zaporednih elementov. Z zanko gremo čez te pare, jih razpakiramo, odštevamo in zlagamo v nov seznam. Obljubljena boljša rešitev te naloge je torej:

[ ]:
```
t = []
for e1, e2 in zip(s, s[1:]):
    t.append(e1 - e2)
```

Če znamo razmišljati tako, bomo kasneje, ko pride na vrsto, tudi lažje razumeli krajšo rešitev:

[ ]:
```
t = [e1 - e2 for e1, e2 in zip(s, s[1:])]
```

0.1.4 Dolžina seznama, terke, niza

Seznami, terke in naši stari znanci nizi imajo nekaj skupnega. Pravzaprav veliko skupnega. Razlikujejo se le v podrobnostih. Skupno jim je, recimo, da imajo vsi trije dolžino. Gornji seznami šestih tež, šestih imen in šestih indikatorjev spola so dolžine 6. Terka `(1, 2, 3, 4)` ima dolžino 4 in niz "Benjamin" je dolžine 8. Dolžine reči, ki imajo dolžino, nam pove funkcija `len`. To sem napisal tako imenitno, da bom moral še enkrat, da bo sploh kdo razumel: funkcija `len`sprejme en argument, recimo seznam, terko ali niz in kot rezultat vrne dolžino tega seznama, terke ali niza.

[ ]:
```
>>> b = 'Benjamin'
>>> len(b)
8
```

```
>>> len(podatki)
6
>>> len((1, 2, 3))
3
>>> len(12)
Traceback (most recent call last):
File "", line 1, in TypeError: object of type 'int' has no len()
```

Morda je koga presenetilo zadnje: številka 12 bi lahko bila dolga 12, ne? Ali pa 2, ker ima dve števki? Ne. Funkcija `len` v bistvu pove *število elementov, ki jih vsebuje podani argument.* Niz "Benjamin" vsebuje 8 znakov, seznam `podatki` vsebuje 6 podseznamov in terka `(1, 2, 3)` ima tri elemente. Število 12 pa nima elementov.

Še ena prikladna značilnost seznamov (nizov, terk in še česa): prazni seznami so, tako kot prazni nizi, neresnični. Seznam lahko uporabimo v pogoju.

[ ]:
```
if s:
    print("Seznam s ni prazen")
else:
    print("Seznam s je prazen")
```