03 seznami, terke, for

January 28, 2024

0.1 Seznami

Python ima podatkovni tip `list`. Jaz to prevedem v *seznam*, eni imajo raje *tabela*, ker se vede malo podobno kot *array* v kakšnem drugem jeziku.

- Zapremo ga v oglate oklepaje. Vsebuje lahko elemente poljubnih tipov, lahko tudi mešanico tipov, npr. `["Ana", 74, True]`. Ampak tega ponavadi ne počnemo. Za različne tipe ponavadi uporabljamo terke (glej spodaj).
- Gre lahko čez več vrstic, če pustimo odprt oklepaj. Zamik v drugi in nadaljnjih vrsticah je načelno poljuben, vendar se splača lepo poravnavati.

[1]:
```
teze = [74, 82, 58,
        66, 61, 84]
```

- Za zadnjim elementom lahko naredimo vejico. To je praktično predvsem, če bomo, recimo, spreminjali vrstni red.

[3]:
```
imena = [
    "Ana",
    "Berta",
    "Cilka",
]
```

- `prazen` je zelo slabo ime za seznam. Če kdo napiše `prazen = []`, grem stavit, da ta seznam ne bo *vedno* prazen. Tudi `seznam` je slabo ime za seznam.

0.2 Terke

Terka (`tuple`) je podobna seznamu, samo da jo zapremo v navadne oklepaje, `t = (1, 2, 3)`.

- Oklepaje lahko tudi izpustimo: `t = 1, 2, 3`. Tega ponavadi ne počnemo.
- Če terka vsebuje samo en element, to ni `(42)` temveč `(42, )`. Če napišemo samo `(42)`, je to samo 42 v oklepaju.
- Pač pa je `()` čisto lepa prazna terka.
- V terko pogosto dajemo stvari različnih tipov.
- Bistvena in skoraj edina razlika med terkami in seznami je, da terk ne moremo spreminjati. Terke so za razliko od seznamov *immutable*.
- Python interno pogosto uporablja terke. Ko, recimo, pokličemo funkcijo, Python spakira vse argumente v terko in funkcija v resnici dobi terko. Samo da programer tega niti ne opazi, to je Pythonova privatna fora.

0.2.1 Dolžina seznama, terke, niza

Funkcija `len(s)` vrne dolžino `s`. `s` je lahko karkoli, kar ima dolžino, npr. niz, seznam, terka, množica, slovar.

0.3 Razpakiranje v elemente

To je ena najboljših stvari v Pythonu (Kotlin jo delno imitira, JavaScript pa je tu še ful bolj kul).

Levo od enačaja je lahko več spremenljivk, če je desno od njega kaj, kar ima natančno toliko elementov. Če bi imeli

[4]:
```
student = ("Ana", 65, "Ž")
prastevila = [2, 3, 5]
crke = "abcdef"
```

lahko pišemo

[5]:
```
ime, teza, spol = student
prvo, drugo, tretje = prastevila
a, b, c, d, e, f = crke
```

Seveda lahko tudi

[6]:
```
ime, teza, spol = ("Ana", 65, "Ž")
```

vendar je to čudno, in celo

[7]:
```
ime, teza, spol = "Ana", 65, "Ž"
```

saj terk ni potrebno zapirati v oklepaje, vendar je to nečitljivo.

Pač pa je zanimivo

[10]:
```
x = 1
y = 2
x, y = y, x
```

[11]:
```
x
```

[11]:
```
2
```

[12]:
```
y
```

[12]:
```
1
```

Prirejanje `x, y = y, x` je v bistvu isto kot `x, y = (y, x)`, vendar to vedno pišemo brez oklepajev, ker je vsakemu jasno, da s tem menjamo vrednosti dveh spremenljivk. (Torej: sintakse `x, y = y, x` si niso izmislili za zamenjavo vrednosti spremenljivk, temveč je to slučajni stranski učinek tega, da terk ni potrebno zapirati v oklepaje in da lahko terke razpakiramo.)

Število imen levo od enačaja mora biti enako številu elementov stvari na desni. Lahko pa dodamo še spremenljivko, v katero gre višek. Označimo jo z zvezdico.

[13]:
```
prvo, drugo, *ostala = [2, 3, 5, 7, 11, 13]
```

[14]:
```
prvo
```

[14]:
```
2
```

[15]:
```
drugo
```

[15]:
```
3
```

[16]:
```
ostala
```

[16]:
```
[5, 7, 11, 13]
```

Lahko celo

[17]:
```
prvo, drugo, *ostala, zadnje = [2, 3, 5, 7, 11, 13]
```

[18]:
```
ostala
```

[18]:
```
[5, 7, 11]
```

Zvezdica nima zveze s kazalci, temveč prihaja iz drugega vica. To vlogo je prvič dobila ob klicih funkcij, ampak to bomo videli drugič.

Razpakiranje v elemente uporabljamo tudi zato, da navidez dosežemo, da lahko funkcija vrača več stvari. Funkcija `splitext(fname)`, ki prejme ime datoteke ter vrne osnovni del in končnico, bi lahko bila definirana tako:

[19]:
```
def splitext(fname):
    dot_idx = fname.rfind(".")
    base = fname[:dot_idx]
    ext = fname[dot_idx:]
    return base, ext
```

in bi jo klicali z

[20]:
```
osnova, koncnica = splitext("en_film.avi")
```

V resnici `return` vrača `return (base, ext)`, samo da tu ne pišemo oklepajev, ob klicu pa to razpakiramo nazaj. Tako sploh ni videti, da delamo s terkami, ampak je videti, kot da vračamo in dobimo dve stvari. Se pravi, v počasnem posnetku se zgodi to:

[21]:
```
def splitext(fname):
    dot_idx = fname.rfind(".")
    base = fname[:dot_idx]
    ext = fname[dot_idx:]
```

```
    t = (base, ext)
    return t
```

in

[22]:
```
ime_konc = splitext("en_film.avi")
osnova = ime_konc[0]
koncnica = ime_konc[1]
```

0.4 Zanka for

C-jevska zanka `for` je samo bolj kompaktno zapisan `while`. Zanka `for(zacetek; pogoj; korak) { nekaj }` je isto kot `zacetek; while (pogoj) { nekaj; korak }`. C ima torej dvakrat isto zanko.

Pythonova zanka `for` pa je drugačna od `while`. Je taka kot zanke, ki se ponekod `for each` ali kaj podobnega. Zanka `for` gre vedno čez neko zbirko stvari, recimo čez seznam, terko, niz, množico, slovar, vrstice datoteke, generator…

[23]:
```
imena = ["Ana", "Berta", "Cilka", "Dani", "Ema"]
for ime in imena:
    print(ime)
```

```
Ana
Berta
Cilka
Dani
Ema
```

Prosim, ne pišite

```
for i in range(len(imena)):
    ime = imena[i]
    print(ime)
```

0.4.1 Razpakiranje v zanki `for`

Navadite se tako:

```
studenti = [("Ana", 65), ("Berta", 80), ("Cilka", 78)]
for ime, teza in studenti:
    ...
```

in celo

```
studenti = [("Ana", (65, 175)), ("Berta", (80, 180)), ("Cilka", (78, 160))]
for ime, (teza, visina) in studenti:
    ...
```

ne pa

```
for i in range(len(studenti)):
    ime = studenti[i][0]
    teza = studenti[i][1][0]
    visina = studenti[i][1][1]
```

Ker očitno.

V obeh zankah, `for` in `while` lahko uporabljamo `break` in `continue`.

0.4.2 Else po zanki

Zanki v Pythonu lahko sledi `else`. Kar zapišemo vanj, se izvede, če zanka ni bila prekinjena z `break`.

```
for x in stevila:
    if x % 2 == 1:
        print("Prvo liho število v seznamu je", x)
        break
else:
    print("Seznam ne vsebuje nobenega lihega števila")
```

Pozor: `else` je poravnan s `for`, ker se nanaša na zanko, in ne z `if`, kar bi bilo očitno narobe.

0.4.3 Zanka po dveh seznamih

Če imamo

```
imena = ["Ana", "Berta", "Cilka", "Dani", "Ema"]
teze = [72, 80, 68, 72, 67]
```

ne pišemo

```
for i in range(len(imena)):
    print(imena[i], teze[i])
```

temveč

[24]:
```
for ime, teza in zip(imena, teze):
    print(ime, teza)
```

```
Ana 74
Berta 82
Cilka 58
Dani 66
Ema 61
```

`zip` vrne približno tole:

```
>>> zip(imena, teze)
[('Ana', 72), ('Berta', 80), ('Cilka', 68), ('Dani', 72), ('Ema', 67)]
```

Tako je bilo pred različico 3. V različici 3 vrne nekaj malo drugačnega, vendar se vede (skoraj) enako.

Funkciji `zip` lahko damo poljubno število seznamov in drugih stvari.

```
for ime, teza, crka, stevilka in zip(imena, teze, "abcdef", (1, 2, 3, 4, 5)):
    ...
```

Če argumenti nimajo enakega števila elementov, se `zip` ustavi, ko je konec najkrajšega.

0.4.4 Zanka prek številskega intervala

Pogosta rabe zanke `for` v C-ish jezikih je `for(i = 0; i < 10; i++)`. V Pythonu je to `for i in range(10)`. Lahko si predstavljamo, da `range` vrne seznam števil. V Pythonu 2.7 bi to bilo tako

```
>>> range(5)
[0, 1, 2, 3, 4]
>>> range(7, 12)
[7, 8, 9, 10, 11]
>>> range(10, 25, 4)
[10, 14, 18, 22]
>>> range(20, 10, -2)
[20, 18, 16, 14, 12]
```

Spodnja meja je vključena, zgornja ni, zato da `range(a, b)` vrne `b - a` elementov in zato da

```
>>> range(5, 8) + range(8, 10)
[5, 6, 7, 8, 9]
```

Od Pythona 3 naprej, `range` ne vrača seznamov, vendar vse deluje zelo podobno, kot da bi jih. Kaj vrača, izvemo čez par tednov.

0.4.5 Indeksiranje

Tole je namerno proti koncu, zato da se čimbolj navadite uporabljati `for` tako, kot je v Pythonu in podobnih jezikih treba.

"Običajni" indeksi

[27]:
```
imena
```

[27]:
```
['Ana', 'Berta', 'Cilka', 'Dani', 'Ema']
```

[28]:
```
imena[0]
```

[28]:
```
'Ana'
```

[29]:
```
imena[1]
```

[29]:
```
'Berta'
```

Indeksi s konca Pogosto potrebujemo zadnji element. Ali predzadnjega. Ne pišemo `imena[len(imena) - 1]` ali `imena[len(imena) - 2]` temveč:

[30]:
```
imena[-1]
```

[30]:
```
'Ema'
```

[31]:
```
imena[-2]
```

[31]:
```
'Dani'
```

Rezine S `s[i:j]` dobimo elemente seznama `s` od `i`-tega do `j`-tega. Z `i`-tim, a brez `j`-tega. Torej `j - i` elementov. Enako deluje tudi s terkami in nizi (Python nima nobenih `substr` in podobnih funkcij).

[32]:
```
s = "Benjamin"
```

[33]:
```
s[2:5]
```

[33]:
```
'nja'
```

Če izpustimo spodnjo mejo, gre od začetka, tako da lahko `s[:i]` preberemo kar "*prvih i elementov*".

[36]:
```
s[:3]
```

[36]:
```
'Ben'
```

Če izpustimo zgornjo mejo, gre do konca. V bistvu `s[i:]` pomeni "*brez prvih i*".

[37]:
```
s[3:]
```

[37]:
```
'jamin'
```

Zdaj pa še negativni indeksi: `s[:-i]` pomeni vse do i-tega z desne, se pravi "*brez zadnjih i*".

[38]:
```
s[:-3]
```

[38]:
```
'Benja'
```

In `s[-i:]` je potem "*zadnjih i elementov*".

[39]:
```
s[-3:]
```

[39]:
```
'min'
```

Jasno delajo tudi kombinacije.

[40]:
```
s[1:-1]
```

[40]:
```
'enjami'
```

In še korak.

[41]:
```
s[1:10:2]
```

[41]:
```
'ejmn'
```

In celo

[43]:
```
s[::-1]
```

[43]:
```
'nimajneB'
```

`s[:]` vrne vse elemente od prvega do zadnjega, torej kopijo. To je uporabno predvsem za sezname. Terke in nizi so nespremenljivi, torej jih nima smisla kopirati.

0.4.6 Trik: zaporedni elementi

Če je `s` nek seznam, bomo z `zip(s, s[1:])` dobili pare zaporednih elementov. Recimo, da nas zanima, ali števila v nekem seznamu naraščajo.

[25]:
```
s = [2, 5, 8, 6, 10, 11, 15]

narascajo = True
for prej, potem in zip(s, s[1:]):
    if potem <= prej:
        narascajo = False
```

Še bolj pa mi je pravzaprav všeč

[26]:
```
for prej, potem in zip(s, s[1:]):
    if potem <= prej:
        narascajo = False
        break
else:
    narascajo = True
```

Spreminjanje seznamov z indeksiranjem in rezanjem

[45]:
```
imena
```

[45]:
```
['Ana', 'Berta', 'Cilka', 'Dani', 'Ema']
```

[47]:
```
imena[2] = "Cecilija"
imena[-1] = "Emilija"
imena
```

[47]:
```
['Ana', 'Berta', 'Cecilija', 'Dani', 'Emilija']
```

[49]:
```
imena[1:3] = ["Benjamin", "Boštjan", "Brane", "Cene"]
imena
```

[49]:
```
['Ana',
 'Benjamin',
```

```
 'Boštjan',
 'Brane',
 'Cene',
 'Brane',
 'Cene',
 'Dani',
 'Emilija']
```

[51]:
```
imena[5:] = []
imena
```

[51]:
```
['Ana', 'Benjamin', 'Cene', 'Brane', 'Cene']
```

Vse to je možno početi le s seznami, ne pa tudi z nizi in terkami, saj so nespremenljivi.

0.5 Računske operacije na seznamih, nizih in terkah

Glede seštevanja se seznami in terke vedejo enako kot nizi: operator `+` stakne dva seznama oz. terki.

[52]:
```
"Ana" + "marija"
```

[52]:
```
'Anamarija'
```

[54]:
```
[5, 7, 2] + [1, 8, 3]
```

[54]:
```
[5, 7, 2, 1, 8, 3]
```

S seštevanjem si pomagamo tudi, če želimo spremeniti element terke ali niza.

[61]:
```
s = "Be_jamin"
s = s[:2] + "n" + s[3:]
s
```

[61]:
```
'Benjamin'
```

Tudi množenje seznamov se vede enako kot množenje nizov.

[55]:
```
"Ana" * 3
```

[55]:
```
'AnaAnaAna'
```

[56]:
```
[5, 7, 2] * 3
```

[56]:
```
[5, 7, 2, 5, 7, 2, 5, 7, 2]
```

Z operatorjem `in` (in `not in`) preverimo, ali seznam vsebuje (in ne vsebuje) določenega elementa.

[57]:
```
5 in [5, 7, 2]
```

[57]:
```
True
```

[58]:
```
5 not in [5, 7, 2]
```

[58]:
```
False
```

Za nize operator `in` preverja, ali vsebujejo podniz.

[59]:
```
"min" in "Benjamin"
```

[59]:
```
True
```

[60]:
```
"min" not in "Benjamin"
```

[60]:
```
False
```

0.6 Dodajanje elementov v seznam

[62]:
```
imena
```

[62]:
```
['Ana', 'Benjamin', 'Cene', 'Brane', 'Cene']
```

[63]:
```
imena.append("Franz")
imena
```

[63]:
```
['Ana', 'Benjamin', 'Cene', 'Brane', 'Cene', 'Franz']
```