05 metode nizov in seznamov

January 28, 2024

0.1 Kako kličemo metode

Funkcijam, ki smo jih srečevali doslej, smo, da so kaj naredile, morali dati podatke, na katerih so delale. Funkcija `len` kot argument zahteva seznam ali niz, katerega dolžino bi radi izvedeli. Funkcija `input` kot argument želi niz z vprašanjem, ki bi ga radi zastavili uporabniku. Razen argumentov, ki jih podamo, te funkcije nimajo drugih podatkov (vsaj ne, da bi mi vedeli zanje).

Ko smo se učili o seznamih, pa smo mimogrede naleteli na neko nenavadnost: za dodajanje novega elementa v seznam nismo poklicali funkcije, ki bi ji podali seznam in element (na primer `append(imena, "Ana")`), temveč smo morali napisati ime seznama, ki mu je sledila pika in `append`, kot argument pa smo navedli le element, ki smo ga želeli dodati, torej `imena.append("Ana")`.

[1]:
```
imena = []
imena.append("Ana")
imena.append("Berta")
imena
```

[1]:
```
['Ana', 'Berta']
```

`append` je očitno nenavaden, ne kličemo ga tako kot druge funkcije. Funkcija `append` "pripada" seznamu; zato mu pravimo `imena.append`. Naredimo še en seznam.

[2]:
```
teze = []
teze.append(55)
imena.append("Cilka")
```

[3]:
```
imena
```

[3]:
```
['Ana', 'Berta', 'Cilka']
```

[4]:
```
teze
```

[4]:
```
[55]
```

Vsak seznam ima svoj `append`: `imena` imajo `imena.append` in `teze` imajo `teze.append`.

Takšnim funkcijam pravimo *metode*. Klic `imena.append` si predstavljamo, kot da *seznamu* ***`imena`*** *rečemo "dodaj si element"* ***`"Cene"`*** in klic `teze.append(55)` pomeni, da *seznamu* ***`teze`*** *rečemo "dodaj si element"* ***`55`***.

Pika, `.` izraža nekakšno pripadnost, vsebovanost. Ko smo spoznali module, smo videli, da z `math.sqrt` pridemo do `math`-ove funkcije z imenom `sqrt`. Z `imena.append` zahtevamo "imenovo" metodo `append`. In s `teze.append` "tezovo" metodo `append`.

Takšnih metod je še veliko: takole prosimo niz `ime`, naj nam pove, koliko črk "n" vsebuje.

[5]:
```
ime = "Benjamin"
ime.count("n")
```

[5]:
```
2
```

Takole pa niz `fname` vprašamo, ali se konča s "torrent".

[6]:
```
fname = "en film.torrent"
fname.endswith(".torrent")
```

[6]:
```
True
```

Kako bi Benjamina spremenili v Benjamaxa? Težko: kot vemo, so nizi nespremenljivi (tako kot terke; od vsega, o čemer smo govorili prejšnji teden, lahko spreminjamo samo sezname). Pač pa lahko vprašamo niz, kako bi bil videti, če "min" zamenjano z "max".

[7]:
```
ime.replace("min", "max")
```

[7]:
```
'Benjamax'
```

Metode niso povezane s tem, ali je niz shranjen v spremenljivki ali ne. Metodo moremo poklicati tudi na nizu "kar tako".

[8]:
```
"Maja".lower()
```

[8]:
```
'maja'
```

Skoraj vsaka reč v Pythonu ima metode (ali vsaj nekaj podobnega). Celo običajna števila imajo metode, čeprav zgolj tehnično in z zelo čudnimi imeni, kot lahko vedoželjen študent hitro preveri.

[9]:
```
x = 3.14
x.is_integer()
```

[9]:
```
False
```

Med tipi, ki smo jih spoznali doslej, pa imajo uporabe vredne metode nizi in seznami. V okviru predavanj ne bomo podrobneje spoznavali vse mogočih podatkovnih tipov. Temeljiteje pa bomo pogledali metode nizov, seznamov in podobnih reči, ki jih pri programiranju v Pythonu vsakodnevno uporabljamo. Obenem bomo tako dobili vtis, kako te reči izgledajo.

Namen tega predavanja ni, da si zapomnite vse te metode in jih znate na pamet za izpit. :) Namen je le, da dobite predstavo, kako to deluje in kakšne vse stvari obstajajo. Za konkretna imena pa vam bo vedno na voljo dokumentacija.

0.1.1 Metode nizov

Nekaj smo jih že spoznali: `count`, `replace` in `lower`. Zadnja ima sestro, `upper`, ki vrne niz, pri katerem so vse črke pretvorjene v velike črke. Podobna sta še `capitalize` in `title`.

[10]:
```
"Maja".lower()
```

[10]:
```
'maja'
```

[11]:
```
"Maja".upper()
```

[11]:
```
'MAJA'
```

[12]:
```
"maja".capitalize()
```

[12]:
```
'Maja'
```

[13]:
```
"stavek, KI GA bomo povelikočrkovali po anGLEško".title()
```

[13]:
```
'Stavek, Ki Ga Bomo Povelikočrkovali Po Angleško'
```

Da ne bi kdo pozabil, za vsak slučaj še enkrat spomnimo: te metode ne spreminjajo niza, temveč vračajo nove nize. Nizi ostanejo takšni, kot so bili. Po tem primeru bi moralo biti jasno, kaj ne deluje in kako je potrebno pisati, da bo delovalo.

[14]:
```
s = "Benjamin"
s.upper()
```

[14]:
```
'BENJAMIN'
```

[15]:
```
s
```

[15]:
```
'Benjamin'
```

[16]:
```
s = s.upper()
s
```

[16]:
```
'BENJAMIN'
```

Metodo `count` smo že spoznali. Povejmo le še, da lahko išče tudi daljše

[17]:
```
"Maja".count("a")
```

[17]:
```
2
```

[18]:
```
"Maja".count("aj")
```

[18]:
```
1
```

Poleg `count` imamo tudi nekaj funkcij, ki povedo, kje v nizu se nahaja določen podniz. Prvi sta `find` in `index`.

[19]:
```
"Benjamin".find("jam")
```

[19]:
```
3
```

[20]:
```
"Benjamin".index("jam")
```

[20]:
```
3
```

Edina razlika med njima je v tem, kaj storita, če iskanega niza ni. `find` vrne -1, `index` pa javi napako.

[21]:
```
"Benjamin".find("max")
```

[21]:
```
-1
```

[22]:
```
"Benjamin".index("max")
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-22-c435d9d16506> in <module>
----> 1 "Benjamin".index("max")

ValueError: substring not found
```

Če se dani podniz pojavi večkrat, bosta funkciji vrnili indeks prve pojavitve - razen, če jima z dodatnimi argumenti povemo, od kod naprej naj iščeta. Poleg `find` in `index` obstajata še funkciji `rfind` in `rindex`, ki iščeta s konca in tako vrneta zadnjo pojavitev podniza v nizu.

Mimogrede smo že omenili tudi preprosto, a zelo uporabno funkcijo `endswith`, ki pove, ali se niz konča s podanim podnizom. Poleg njega obstaja še `startswith`, ki pove ali se niz *začne* z danim podnizom.

Metode `ljust`, `rjust` in `center` dopolnijo niz s presledki z leve, z desne ali z obeh strani, tako da je dolg toliko, kot želimo. Takole napihnemo Benjamina na 15 znakov:

[27]:
```
"Benjamin".ljust(15)
```

[27]:
```
'Benjamin       '
```

[28]:
```
"Benjamin".rjust(15)
```

[28]:
```
'       Benjamin'
```

[29]:
```
"Benjamin".center(15)
```

[29]:
```
'    Benjamin   '
```

Še enkrat: te metode ne spreminjajo niza, temveč vrnejo nov niz! Nobena metoda ne more spremeniti niza.

Metoda `strip` naredi ravno nasprotno: odbije presledke (in tabulatorje in znake za prehod v novo vrstico) na levi in desni strani niza. `lstrip` in `rstrip` pospravita samo levo in samo desno stran.

[30]:
```
"     Benjamin    ".strip()
```

[30]:
```
'Benjamin'
```

[31]:
```
"     Benjamin    ".lstrip()
```

[31]:
```
'Benjamin    '
```

[32]:
```
"     Benjamin    ".rstrip()
```

[32]:
```
'     Benjamin'
```

Če se zdi ta metoda komu neuporabna, se moti. Zelo. Svojo zmoto bo spoznal, ko pridemo do branja datotek.

Nizi imajo še veliko metod. Pogledali bomo le še dve najzanimivejši.

Metoda `split` razbije niz na seznam podnizov, kakor jih ločujejo presledki, tabulatorji in znaki za nove vrstice. (V prvem približku: razbije jih na besede.)

[33]:
```
s = "Metoda split razbije niz na seznam podnizov, kakor jih ločujejo presledki."
s.split()
```

[33]:
```
['Metoda',
 'split',
 'razbije',
 'niz',
 'na',
 'seznam',
 'podnizov,',
 'kakor',
 'jih',
 'ločujejo',
 'presledki.']
```

Presledkom, tabulatorjem in znakom za novo vrstico rečemo tudi *beli prostor* (*white space*). Namesto glede na beli prostor lahko `split` razbije niz tudi glede na kak drug znak, ki ga moramo v tem primeru podati kot argument.

[34]:
```
"123-4123-21".split("-")
```

[34]:
```
['123', '4123', '21']
```

[35]:
```
"123-4123-21".split("1")
```

[35]:
```
['', '23-4', '23-2', '']
```

Zadnja metoda, `join`, dela ravno obratno kot `split`: združuje nize. Način, na katerega je obrnjena, je nenavaden, a če malo razmislimo, vidimo, da drugače ne bi moglo biti.

[36]:
```
imena = ["Ana", "Berta", "Cilka", "Dani", "Ema"]
"".join(imena)
```

[36]:
```
'AnaBertaCilkaDaniEma'
```

Praznemu nizu, `""`, smo "naročili", naj združi nize iz podanega seznama. To sicer ni videti preveč lepo, lepše bo, če jih združimo s kakim ločilom.

[37]:
```
" - ".join(imena)
```

[37]:
```
'Ana - Berta - Cilka - Dani - Ema'
```

[38]:
```
", ".join(imena)
```

[38]:
```
'Ana, Berta, Cilka, Dani, Ema'
```

[39]:
```
" in ".join(imena)
```

[39]:
```
'Ana in Berta in Cilka in Dani in Ema'
```

Najlepše pa bo, če vsa imena do zadnjega združimo z vejicami, nato pa z "in" pripnemo še zadnje ime. Ker ravno poznamo rezine.

[40]:
```
", ".join(imena[:-1]) + " in " + imena[-1]
```

[40]:
```
'Ana, Berta, Cilka, Dani in Ema'
```

To, mimogrede, deluje celo z dvema imenoma - prvo bo pač pustil pri miru, saj v seznamu, kot je `["Ana"]` ni česa združevati:

[41]:
```
imena = ["Ana", "Benjamin"]
", ".join(imena[:-1]) + " in " + imena[-1]
```

[41]:
```
'Ana in Benjamin'
```

Pri enem samem pa malo zataji.

[42]:
```
imena = ["Ana"]
", ".join(imena[:-1]) + " in " + imena[-1]
```

[42]:
```
' in Ana'
```

0.1.2 Metode seznamov

Tako kot nizi imajo tudi seznami metodi `count` in `index`, o katerih ne bomo izgubljali besed.

Medtem ko vam pri nizih metoda `index` ne bo prišla velikokrat na misel, jo boste, sodeč po izkušnjah iz preteklih generacij, pridno zlorabljali na seznamih. Že kaka dva, tri tedne pred temi predavanji jo nekateri študenti odkrijejo in si z njeno pomočjo otežijo reševanje domačih nalog.

Metoda `index` je v resnici uporab(lje)na zelo redko. Metode `index` prav gotovo ne gre uporabljati znotraj zanke `for`, ki gre prek seznama in ob tem poskuša dobiti indeks trenutnega elementa.

Ena tipičnih zlorab je ta:

[43]:
```
imena = ["Ana", "Berta", "Cilka", "Dani", "Ema"]
teze = [72, 78, 70, 65, 68]
for ime in imena:
    i = imena.index(ime)
    print(ime, "tehta", teze[i], "kilogramov.")
```

```
Ana tehta 72 kilogramov.
Berta tehta 78 kilogramov.
Cilka tehta 70 kilogramov.
Dani tehta 65 kilogramov.
Ema tehta 68 kilogramov.
```

Tole tule slučajno deluje. Neha pa delovati takoj, ko imamo v seznamu dve enaki imeni: `index` bo vrnil prvo.

[44]:
```
imena = ["Ana", "Ana", "Ana"]
teze = [72, 78, 70]
for ime in imena:
    i = imena.index(ime)
    print(ime, "tehta", teze[i], "kilogramov.")
```

```
Ana tehta 72 kilogramov.
Ana tehta 72 kilogramov.
Ana tehta 72 kilogramov.
```

Tu se študenti tipično pritožijo: ko se `ime` nanaša na drugo Ano v seznamu, bi morala metoda `index` po njihovem vrniti indeks drugega elementa, `1` in ne `0`.

Če bi `index` res delal tako: kaj bi dobili, če bi poklicali `imena.index("Ana")`? V tem primeru se niz `"Ana"` ne nanaša na nek specifičen element v seznamu.

V resnici je `ime` v gornjem primeru vedno samo `"Ana"`. Ne neka konkretna Ana s tega seznama. Samo Ana. In `index` vrne prvo Ano.

Vem, vem, zakaj študenti pišejo programe, kot je gornji. Zaradi moje reklame proti `range(len(...))`. Radi bi napisali tole, a si ne upajo.

[45]:
```
imena = ["Ana", "Berta", "Cilka", "Dani", "Ema"]
teze = [72, 78, 70, 65, 68]

for i in range(len(imena)):
    print(imena[i], "tehta", teze[i], "kilogramov.")
```

```
Ana tehta 72 kilogramov.
Berta tehta 78 kilogramov.
Cilka tehta 70 kilogramov.
Dani tehta 65 kilogramov.
Ema tehta 68 kilogramov.
```

In prav je, da si ne upajo. To se naredi tako:

[46]:
```
for ime, teza in zip(imena, teze):
    print(ime, "tehta", teza, "kilogramov.")
```

```
Ana tehta 72 kilogramov.
Berta tehta 78 kilogramov.
Cilka tehta 70 kilogramov.
Dani tehta 65 kilogramov.
Ema tehta 68 kilogramov.
```

Kadar potrebujemo element in indeks, pa uporabimo `enumerate`.

[47]:
```
for i, ime in enumerate(imena):
    print(i, ":", ime)
```

```
0 : Ana
1 : Berta
2 : Cilka
3 : Dani
4 : Ema
```

Malo smo skrenili s poti, ampak je pomembno. Zato bom povedal še enkrat **preden uporabite `index`, razmislite, ali ga res potrebujete**. Metoda `index` je

- nevarna, ker vedno vrne prvi element s podano vrednostjo, in tega morda ne pričakujemo,
- počasna, ker mora Python dejansko pregledati seznam do iskanega elementa,
- navadno nepotrebna, ker lahko pridemo do indeksa po preprostejši poti,
- poleg tega pa navadno brez potrebe zapleta program.

Če je tako slab, zakaj obstaja? Ker ga včasih čisto legalno potrebujemo. Zakaj pa o njem govorim programerjem - začetnikom? Zato, ker ga sicer odkrijejo sami. `index` je ena od stvari, za katere je boljše, da otroci o njih slišijo od staršev, ne vrstnikov.

Zdaj pa nazaj k metodam.

Že nekaj časa vemo za `append`, ki v seznam doda nov element. Pazite, tole je zelo drugače kot pri nizih! Metoda `append` ne vrača novega seznama, temveč v resnici spreminja seznam.

Kako pa bi k nizu pripeli seznam? Tule nam `append` ne bo pomagal: kar naredi, je povsem narobe.

[48]:
```
imena
```

[48]:
```
['Ana', 'Berta', 'Cilka', 'Dani', 'Ema']
```

[49]:
```
imena.append(["Fanči", "Ema", "Greta"])
```

Kaj smo pa pričakovali? Metoda `append` prejme en argument. In to kar prejme, doda k seznamu. Torej: kar ji podamo kot argument, bo zadnji element seznama. Če torej `append`-u podamo seznam, bo seznam zadnji element seznama.

Metoda, ki bo naredila, kar hočemo, je `extend`.

[50]:
```
imena = ['Ana', 'Berta', 'Cilka', 'Dani', 'Ema']
imena.extend(["Fanči", "Greta"])
imena
```

[50]:
```
['Ana', 'Berta', 'Cilka', 'Dani', 'Ema', 'Fanči', 'Greta']
```

V resnici `extend`-a praktično ne potrebujemo. Ker je sezname možno seštevati, raje uporabimo kar `+=`.

[51]:
```
imena += ["Helga", "Iva"]
imena
```

[51]:
```
['Ana', 'Berta', 'Cilka', 'Dani', 'Ema', 'Fanči', 'Greta', 'Helga', 'Iva']
```

Metodi `insert` podamo dva argumenta, indeks in element, pa bo vstavila element *pred* element s podanim indeksom.

[52]:
```
imena = ['Ana', 'Berta', 'Dani', 'Ema', 'Greta']
imena.insert(2, "Cilka")
imena.insert(-1, "Fanči")
imena
```

[52]:
```
['Ana', 'Berta', 'Cilka', 'Dani', 'Ema', 'Fanči', 'Greta']
```

Očitno deluje tudi indeksiranje s konca.

Elemente seznama lahko odstranjujemo na tri načine.

Prvi je `del`. `del` ni metoda in prihaja iz povsem drugega vica, a vseeno ga pač omenimo, ker se ravno učimo o spreminjanju seznamov. Uporabimo ga tako:

[53]:
```
del imena[-2]
imena
```

[53]:
```
['Ana', 'Berta', 'Cilka', 'Dani', 'Ema', 'Greta']
```

Metoda `pop` vrne element s podanim indeksom in ga pobriše s seznama.

[54]:
```
imena.pop(2)
```

[54]:
```
'Cilka'
```

[55]:
```
imena
```

[55]:
```
['Ana', 'Berta', 'Dani', 'Ema', 'Greta']
```

Metodo `pop` redko uporabljamo za pobiranje elementov sredi seznamov. Navadno na ta način poberemo prvi ali zadnji element. Pravzaprav najpogosteje zadnjega, zato lahko `pop` pokličemo tudi brez argumentov, pa bomo dobili zadnji element.

[56]:
```
imena.pop()
```

[56]:
```
'Greta'
```

[57]:
```
imena
```

[57]:
```
['Ana', 'Berta', 'Dani', 'Ema']
```

Če bi hoteli, recimo, prestaviti prvi element na konec, bi napisali

[58]:
```
imena.append(imena.pop(0))
imena
```

[58]:
```
['Berta', 'Dani', 'Ema', 'Ana']
```

V obeh primeri, pri `del` in `pop` smo morali podati indeks elementa, ki ga želimo odstarniti. Včasih nimamo indeksa, pač pa poznamo vrednost elementa, ki ga želimo odstraniti. V tem primeru uporabimo `remove`.

[59]:
```
imena.remove("Dani")
imena
```

[59]:
```
['Berta', 'Ema', 'Ana']
```

Pri tem ne odstrani vseh takšnih elementov, temveč le prvega, na katerega naleti, kakor lahko vidimo v spodnjem primeru.

[60]:
```
s = [7, 1, 2, 3, 4, 1, 1, 2]
s.remove(1)
s
```

[60]:
```
[7, 2, 3, 4, 1, 1, 2]
```

Za metodo `remove` velja vse, kar smo napisali za `index`. V resnici za `remove` sicer obstaja več situacij, ko ga "legalno uporabimo", vseeno pa je nevaren, ker odstrani prvi takšen element; počasen, ker mora element najprej poiskati; pogosto nepotreben (ker lahko poznamo indeks). Poleg tega pa je še zelo nevaren: študenti ga radi uporabijo znotraj zanke, v kateri gredo čez prav ta isti

seznam. Brisati elemente seznama, čez katerega greš ravnokar z zanko, je vedno zelo slaba ideja. Če uporabljaš `for` ali če nisi pošteno previden.

Metoda `s.copy()` naredi kopijo seznama `s`, tako kot da bi napisali `s = s[:]`. Navadno uporabimo `s.copy()`, ker bolj eksplicitno pove, kaj počnemo.

Metoda `s.clear()` ga izprazni, kar je isto kot `s[:] = []` ali `del s[:]` in *ni isto kot* `s = []`. Razlika je tako prefinjena, da bo vredna posebnega predavanja. Navadno uporabimo `s.clear()`.

Le še dve metodi sta nam ostali. `reverse` obrne vrstni red elementov v seznamu.

[61]:
```
s = [7, 2, 5, 1, 1]
s.reverse()
s
```

[61]:
```
[1, 1, 5, 2, 7]
```

Zadnja je `sort`, ki uredi elemente po vrsti.

[62]:
```
s.sort()
s
```

[62]:
```
[1, 1, 2, 5, 7]
```

Metoda deluje nad vsakršnimi elementi, ki jih je mogoče primerjati - z njo lahko uredimo seznam števil, nizov… Podamo ji lahko tudi kup argumentov, ki pa jih v tem trenutku še nismo zmožni razumeti.

Ne spreglejte razlike med metodami nizov in seznamov. Metode nizov vračajo nove nize; `s.replace("min", "max")` ni spremenil niza `s`, temveč vrnil nov niz. Metode seznamov spreminjajo seznam; `s.insert(2, "Ana")` ne vrne novega seznama, temveč spremeni `s`. Enako velja za `sort` in `reverse`.

Tule omenimo še dve Pythonovi funkciji, ki nista metodi, sta pa podobni gornjima in študente pogosto bega razlika.

[63]:
```
s = [7, 2, 5, 1, 1]
for e in reversed(s):
    print(e)
```

```
1
1
5
2
7
```

[64]:
```
s
```

[64]:
```
[7, 2, 5, 1, 1]
```

[65]:
```
sorted(s)
```

[65]: `[1, 1, 2, 5, 7]`

[66]:
```
s
```

[66]: `[7, 2, 5, 1, 1]`

`sort` in `reverse` sta metodi seznamov. Imeni sta v velelniku - uredi!, obrni! - torej uredita oz. obrneta seznam. Torej: spremenita ga.

`sorted` in `reversed` sta funkciji. Imeni sta urejen in obrnjen, vrneta torej nov seznam (`reversed` v resnici ne vrača seznama, vrne pa nekaj, kar se vede malo podobno kot seznam). Seznam, ki ga podamo kot argument, ostane nespremenjen. Obe funkciji skoraj vedno uporabimo za zanko `for`.

Eden od razlogov, zakaj `sorted` in `reversed` nista metodi, je tudi, da lahko na ta način sprejemata različne argumente. Zadovoljni sta tudi z nizom ali terko (in vsem drugim, čez kar je možno iti z zanko `for`).

[67]:
```
sorted("berta")
```

[67]: `['a', 'b', 'e', 'r', 't']`

[68]:
```
for e in reversed("berta"):
    print(e)
```

```
a
t
r
e
b
```

0.2 Metode terk

Terke imajo enake metode kot seznami, manjkajo jim le metode, ki spreminjajo seznam.

In, presenečenje: vse metode seznamov spreminjajo seznam. Terke imajo le metodi: `count` in `index`.