

Metode

Na teh predavanjih začetnikom prvič zares vidijo metode in vidijo, kako ta reč funkcionira z nizi, terkami in seznamami. Obenem pa seveda spoznajo kup metod nizov in seznamov, kar nam bo prišlo prav pri programiranju kasneje.

Na hitrejši izvedbi teh predavanj, bomo bolj kot ne preleteli seznam metod.

Še prej pa tem, ki že kaj vedo, povejmo nekaj o funkcijah in metodah.

Funkcije in metode

Tole v tem razdelku niti ni pomembno in pišem samo zaradi študentov, ki so se pogovarjali o tem, ali so funkcije v Pythonu ekvivalentne metoda ne-vem-več-kje.

Besedi *funkcija* in *metoda* imajo v različnih besedah nekoliko različne pomene. V Pythonu se beseda funkcija navadno uporablja za takšne funkcije, ki niso vezane na nek objekt. To so

- globalne funkcije (`print`, `abs`) oz. globalne funkcije modulov (`math.sqrt`, `os.path.split`)
- funkcije znotraj razredov, ki (še) niso vezane na noben objekt (in morda niti ne morejo biti, ker so statične).

Z besedo metoda bomo mislili funkcijo, ki je vezana na določen objekt, se pravi instanco razreda ali razred.

Če komu kaj pomeni, je tule primer. Kdor ga ne razume, naj se ne vznemirja.

```
class A:
    def f(self):
        ...

    @staticmethod
    def g():
        ...

    @classmethod
    def h(cls):
        ...
```

Za zdaj imamo le razred. `A.f` je metoda razreda, vendar pričakuje kot argument objekt. `A.f` torej ni vezana in ji Python pravi funkcija.

`A.f`

```
<function __main__.A.f(self)>
```

`A.g` smo definirali kot `staticmethod`, torej ne bo nikoli vezana na nič in ji bo Python vedno rekel funkcija. (Kar je nekoliko nekonsistentno, saj se dekorator, s katerim smo jo okrasili, imenuje prav `staticmethod`!).

A.g

```
<function __main__.A.g()>
```

h pa je `classmethod` in kot argument prejme razred. Zato je vedno vezan in mu Python pravi vezana metoda.

A.h

```
<bound method A.h of <class '__main__.A'>>
```

Če naredimo objekt razreda A, a, bo a.f (vezana) metoda.

```
a = A()
```

```
a.f
```

```
<bound method A.f of <__main__.A object at 0x10993e790>>
```

Za g in h pa se nič ne spremeni in je pravzaprav vseeno, ali do njiju pridemo z a.g in a.h ali A.g in A.h.

a.g

```
<function __main__.A.g()>
```

a.h

```
<bound method A.h of <class '__main__.A'>>
```

Kot sem napisal, pa so tole samo terminološki detajli, ki se jih tudi jaz navadno niti ne spomnim, ker ... so nepomembnei

Metode nizov

Pri metodah nizov predvsem ne smemo pozabiti, da so le-ti nespremenljivi. Metode, kot je `replace` ne spreminjajo nizov, temveč vračajo nove nize.

Precej spodnjih metod (`find`, `strip`...) ima še dodatne argumente, ki jih bomo tule zamolčali.

- `s.startswith(x)`, `s.endswith(x)`: `True`, če se niz začne oz. konča z x.
- `s.count(x)`: vrne število pojavitev podniza x znotraj niza s.
- `s.replace(o, n)`: vrne niz, v katerem so vse pojavitve podniza o zamenjane z nizom n.
- `s.lower()`, `s.upper()`, `s.capitalize()`, `s.title()`: vrne niz, v katerem so vse črke podanega niza zamenjane z malimi oz velikimi, ali pa je velika le prva črka ali pa prve črke vseh besed.
- `s.index(x)`, `s.rindex(x)`: vrne indeks prve oz. zadnje pojavitve podniza x v s. Če takega podniza ni, sproži izjemo.
- `s.find(x)`, `s.rfind(x)`: isto, le, da v primeru, da niza ni, vrne -1.
- `s.ljust(x)`, `s.rjust(x)`, `s.center(x)`: na začetek, konec ali začetek in konec niza doda toliko presledkov, da ima le-ta x znakov. Precej neuporabno, saj nize oblikujemo drugače.

- `s.strip()`, `s.lstrip()`, `s.rstrip()`: odluči bel prostor na obeh straneh, začetku ali koncu niza.
- `s.split(c)`: vrne seznam podnizov, ki ga dobi, če `s` loči glede na znak `c`. Argument navadno izpustimo in dobimo nize, ločene glede na beli prostor (presledek, tabulator, nova vrstica).

Posebej si oglejmo samo `join`.

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

'AnaBertaCilkaDaniEma'
```

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

```
" - ".join(imena)

'Ana - Berta - Cilka - Dani - Ema'

", ".join(imena)

'Ana, Berta, Cilka, Dani, Ema'

" in ".join(imena)

'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.

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

'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:

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

'Ana in Benjamin'
```

Pri enem samem pa malo zataji.

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

' in Ana'
```

Metode seznamov

Tako kot nizi imajo tudi seznammi 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:

```
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.

```
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.

```
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:

```
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`.

```
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.

```
imena
```

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

```
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`.

```
imena = ['Ana', 'Berta', 'Cilka', 'Dani', 'Ema']
```

```
imena.extend(["Fanči", "Greta"])
```

```
imena
```

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

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

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

```
imena
```

```
['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.

```
imena = ['Ana', 'Berta', 'Dani', 'Ema', 'Greta']
```

```
imena.insert(2, "Cilka")
```

```
imena.insert(-1, "Fanči")
```

```
imena
```

```
['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:

```
del imena[-2]
```

```
imena
```

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

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

```
imena.pop(2)
```

```
'Cilka'
```

```
imena
```

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

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

```
imena.pop()
```

```
'Greta'
```

```
imena
```

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

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

```
imena.append(imena.pop(0))
```

```
imena
```

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

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

```
imena.remove("Dani")
```

```
imena
```

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

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

```
s = [7, 1, 2, 3, 4, 1, 1, 2]
```

```
s.remove(1)
```

```
s
```

```
[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.

```
s = [7, 2, 5, 1, 1]
```

```
s.reverse()
```

```
s
```

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

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

```
s.sort()
```

```
s
```

```
[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.

```
s = [7, 2, 5, 1, 1]
```

```
for e in reversed(s):
```

```
    print(e)
```

```
1
```

```
1
```

```
5
```

```
2
```

```
7
```

```
s
```

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

```
sorted(s)
```

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

```
s
```

```
[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`).

```
sorted("berta")  
['a', 'b', 'e', 'r', 't']  
for e in reversed("berta"):  
    print(e)  
  
a  
t  
r  
e  
b
```

Metode terk

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

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