# 08 Osnove numpyja

January 28, 2024

## 1 Osnove `numpy`-ja

Osnovni element - osnovni podatkovni tip - v `numpy`-ju je tabela (`array`). Ta je nekoliko podobna Pythonovemu seznamu. (V slovenščini mu nekaterimi pravimo *seznam*, ker je njegovo uradno angleško ime `list`, drugi pa to slovenijo v *tabela*, ker bi bilo to ime iz določenih razlogov dejansko boljše. Tu pa nam pride prav, da je ime *tabela* še nezasedeno in ga lahko uporabimo za `numpy`-jeve *array*-e.)

Med numpyjevimi tabelami in Pythonovimi seznami je več pomembnih razlik. Za prvo srečanje z numpyjem se bomo posvetili dvema: razlikam v načinu indeksiranja in razlikam med tem, kako nanju delujejo različne operacije.

Pripravimo si dva seznama v Pythonu.

[1]:
```
p = [3, 8, 9, 2]
r = [8, 0, 1, -3]
```

Z indeksiranjem lahko pridemo do posamičnih elementov. Indeksi morajo biti seveda cela števila (`int`).

[2]:
```
p[2]
```

[2]:
```
9
```

Sezname lahko tudi "seštevamo". Besedo *seštevanje* pravzaprav uporabljamo zgolj zato, ker uporabimo operator `+`. V resnici ne gre za seštevanje (v matematičnem pomenu), temveč za *stikanje*.

[3]:
```
p + r
```

[3]:
```
[3, 8, 9, 2, 8, 0, 1, -3]
```

Ker `+` v resnici ni seštevanje, odštevanje seznamov nima nobenega smisla.

[4]:
```
p - r
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[4], line 1
----> 1 p - r
```

```
TypeError: unsupported operand type(s) for -: 'list' and 'list'
```

Prav tako nima smisla k seznamu prišteti 1, saj ne moremo stakniti seznama in števila.

[5]:
```
p + 1
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[5], line 1
----> 1 p + 1

TypeError: can only concatenate list (not "int") to list
```

Pač pa lahko seznam pomnožimo s celim številom; rezultat je tak, kot da bi seznam tolikokrat sešteli.

[6]:
```
p * 3
```

[6]:
```
[3, 8, 9, 2, 3, 8, 9, 2, 3, 8, 9, 2]
```

### 1.1 Uvažanje modula `numpy`

Če hočemo modul uporabiti, ga moramo najprej uvoziti. Modula `numpy` navadno ne uvažamo z `import numpy`, ker bomo njegove funkcije uporabljali tako pogosto, da bi bil ves program poln `numpy.` – ime `numpy` je preprosto predolgo. Po drugi strani ne uvažamo posamičnih funkcij `from numpy import hstack, sum, max`, ker tipično potrebujemo preveč različnih funkcij, poleg tega pa imajo nekatere enaka imena kot vdelane Pythonove funkcije (na primer `sum` in `max`; funkcije, uvožene iz `numpy` bi jih izpodrinile in povzročile težave, saj pričakujejo drugačne podatke, delajo malo drugače in vračajo drugačne rezultate kot Pythonove.

`numpy` zato običajno uvozimo z

[8]:
```
import numpy as np
```

To uvozi modul, vendar ga potem ne vidimo pod imenom `numpy`, temveč pod krajšim, prijaznejšim imenom `np`.

Zdaj pa naredimo dve tabeli. Najpreprosteje bo poklicati funkcijo `array` in ji podati seznam elementov.

[9]:
```
a = np.array([3, 8, 9, 2])
b = np.array([8, 0, 1, -3])
```

Tako.

[10]:
```
a
```

[10]:
```
array([3, 8, 9, 2])
```

### 1.1.1 Indeksiranje tabel

Tabele indeksiramo tako kot sezname. Z indeksi, ki morajo biti cela števila.

[11]:
```
a[2]
```

[11]:
```
9
```

Delujejo tudi rezine in vse, kar smo se naučili v zvezi z njimi.

[12]:
```
a[2:4]
```

[12]:
```
array([9, 2])
```

[13]:
```
a[1:]
```

[13]:
```
array([8, 9, 2])
```

[14]:
```
a[:-1]
```

[14]:
```
array([3, 8, 9])
```

Prva razlika primerjavi s seznami: če potrebujemo več elementov, lahko podamo več indeksov. Ne kar tako, da jih naštejemo (s tem bomo dosegli nekaj drugega), temveč tako, da kot indeks podamo seznam ali tabelo indeksov.

Če imamo torej

[15]:
```
a
```

[15]:
```
array([3, 8, 9, 2])
```

je

[16]:
```
a[[2, 0, 1, 0, 0, 1]]
```

[16]:
```
array([9, 3, 8, 3, 3, 8])
```

tabela, ki vsebuje drugi, ničti, prvi, potem še dvakrat ničti in spet prvi element tabele `a`.

Namesto seznama `int`-ov, lahko podamo seznam (ali tabelo) `bool`-ov. Ta seznam mora biti enako dolg kot tabela, ki jo indeksiramo, saj pove, katere elemente bi radi in katerih ne. Takole, recimo, dobimo prvi in zadnji element tabele:

[17]:
```
a[[True, False, False, True]]
```

[17]:
```
array([3, 2])
```

Oboje - predvsem zadnje - je videti … ne preveč uporabno. V resnici bo fenomenalno uporabno. Počakajmo na primer.

## 1.2 Operacije nad tabelami

Imamo torej

[18]:
```
a
```

[18]:
```
array([3, 8, 9, 2])
```

[19]:
```
b
```

[19]:
```
array([ 8,  0,  1, -3])
```

Seštejmo ju.

[20]:
```
a + b
```

[20]:
```
array([11,  8, 10, -1])
```

Da, to je v resnici seštevanje. Potemtakem lahko tudi v resnici odštevamo.

[21]:
```
a - b
```

[21]:
```
array([-5,  8,  8,  5])
```

In množimo.

[22]:
```
a * b
```

[22]:
```
array([24,  0,  9, -6])
```

Vse operacije nad tabelami, delujejo po elementih. `+`, `-`, `*` ... vsaka operacija se izvede na vsakem elementu posebej. Zato lahko tabele množimo tudi s števili, jim prištevamo števila...

[23]:
```
a
```

[23]:
```
array([3, 8, 9, 2])
```

[24]:
```
a + 1
```

[24]:
```
array([ 4,  9, 10,  3])
```

[25]:
```
a * 2.5
```

[25]:
```
array([ 7.5, 20. , 22.5,  5. ])
```

[26]:
```
np.array([2]) ** range(10)
```

[26]:
```
array([  1,   2,   4,   8,  16,  32,  64, 128, 256, 512])
```

Še več. Celo operacije, kot so `<` delujejo nad posameznimi elementi.

[27]:
```
b
```

[27]:
```
array([ 8,  0,  1, -3])
```

[28]:
```
b > 0
```

[28]:
```
array([ True, False,  True, False])
```

Pa funkcije tudi!

[29]:
```
np.abs(b)
```

[29]:
```
array([8, 0, 1, 3])
```

[30]:
```
np.sqrt(a)
```

[30]:
```
array([1.73205081, 2.82842712, 3.        , 1.41421356])
```

Kjer je to smiselno, jasno. Funkcije, kot so `sum` in `max` bodo seveda seštevale in množile.

[31]:
```
np.sum(a)
```

[31]:
```
22
```

[32]:
```
np.max(a)
```

[32]:
```
9
```

Tule smo uporabljali `numpy`-jeve funkcije `abs`, `sqrt`, `sum` in `max`. Namesto nekaterih od njih bi lahko uporabili tudi Pythonove funkcije. `sum` zna seštevati, kar jima pride pod roko (točneje: vse, čez kar lahko nažene zanko `for`), pa tudi `max` in `min` nista izbirčna.

[33]:
```
sum(a)
```

[33]:
```
22
```

Vendar: ko delamo s tabelami, se nam vedno splača uporabiti `numpy`-jeve ekvivalente funkcije. Če ne drugega, bodo hitrejše, včasih pa vdelane Pythonove funkcije ne bodo delovale pravilno ali pa ne sploh (`sqrt(a)` javi napako). Poleg tega imajo `numpy`-jeve funkcije pogosto dodatne argumente, specifične za delo s tabelami.

### 1.3 Vektorske operacije

Ko delamo z `numpy`-jem, se poskušamo predvsem izogniti pisanju zank.

Kako bi dobili vsoto vseh pozitivnih elementov `b`-ja? Najprej sestavimo "masko", tabelo `bool`-ov, ki vsebuje `True` na mestih, kjer ima `b` pozitivne elemente.

[34]:
```
b > 0
```

[34]:
```
array([ True, False,  True, False])
```

S to masko lahko izberemo pozitivne elemente.

[35]:
```
b[b > 0]
```

[35]:
```
array([8, 1])
```

Ker nas zanima vsota, jih seštejemo.

[36]:
```
np.sum(b[b > 0])
```

[36]:
```
9
```

Če bi nas zanimalo samo, koliko pozitivnih elementov ima `b`, bi preprosto sešteli masko, saj tudi v `numpy` velja, da je `True` toliko kot `1`, `False` pa toliko kot `0`.

[37]:
```
np.sum(b > 0)
```

[37]:
```
2
```

Če bi bili še manj zahtevni in bi nas zanimalo le, ali ima `b` kakšen pozitiven element, bi uporabili `any` (spet vzamemo `numpy`-je ekvivalent in ne Pyhonovega vdelanega `any`, ki smo ga spoznali prejšnjo uro):

[38]:
```
np.any(b > 0)
```

[38]:
```
True
```

Da `b` nima samih pozitivnih elementov, pa nam pove `all`:

[39]:
```
np.all(b > 0)
```

[39]:
```
False
```

Podobno preprosto je poiskati (šteti, preverjati) sode elemente `a`-ja.

[40]:
```
a[a % 2 == 0]
```

[40]:
```
array([8, 2])
```

Ta, zadnji primer je kar zanimivo prebrati: `a % 2 == 0`. Kar ta formula pravi o `a`-ju, se v bistvu nanaša na vsak element `a`-ja. Ko rečemo `a % 2 == 0` dobimo to, kar bi v golem Pythonu, s seznami, dosegli z `[x % 2 == 0 for x in a]`. Pogovor o izpeljanih seznamih je bil - če ne zaradi drugega - potreben zato, da lažje razumemo idejo "vektorskih operacij" - operacij, ki se, popolnoma enake, zgodijo na vsakem elementu tabele. Nekje znotraj `numpy`-ja se seveda še vedno skriva neka zanka, vendar je zaradi načina, na katerega je `numpy` narejen takšna zanka veliko (kjer "veliko" zlahka pomeni petdesetkrat ali celo stokrat) hitrejša, kot če bi zapisali zanko v Pythonu.

### 1.4 Primer: višinske razlike

Kolesar je na vsakih sto metrov (neke izgleda kar razgibane :) vožnje zabeležil svojo nadmorsko višino.

[41]:
```
h = np.array([345, 355, 360, 364, 378, 370, 360, 355, 360, 361])
```

Zdaj ga, kot vsakega kolesarja, ki počne takšne stvari, zanima skupni dvig.

Za začetek je potrebno dobiti seznam sprememb višine med zaporednimi pari meritev. S seznami bi napisali nekaj takšnega:

[42]:
```
d = [x - y for x, y in zip(h[1:], h)]

d
```

[42]:
```
[10, 5, 4, 14, -8, -10, -5, 5, 1]
```

Vendar se želimo izogniti zanki. Operator `-` želimo uporabiti na tabeli, ne na njenih posamičnih elementih. Gornji `zip` - oziroma njegovi argument - so že pokazali, kaj je potrebno odšteti.

[43]:
```
print(h[1:])
print(h)
```

```
[355 360 364 378 370 360 355 360 361]
[345 355 360 364 378 370 360 355 360 361]
```

Odšteti moramo, preprosto drugo tabelo od prve.

[44]:
```
h[1:] - h
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[44], line 1
----> 1 h[1:] - h

ValueError: operands could not be broadcast together with shapes (9,) (10,)
```

Da, mali detajl: ker `-` deluje po elementih, morata biti tabeli enako dolgi. Druga je za en element predolga. Zadnji element zato odrežemo.

[45]:
```
d = h[1:] - h[:-1]

d
```

[45]:
```
array([ 10,   5,   4,  14,  -8, -10,  -5,   5,   1])
```

Če to seštejemo, bomo seveda dobili `0`, saj je kolesar končal, kjer je začel.

[46]:
```
np.sum(d)
```

[46]: 16

Vsota spustov je pač enaka vsoti dvigov. Kolesarja zanimajo le dvigi,

[47]:
```
d[d > 0]
```

[47]:
```
array([10,  5,  4, 14,  5,  1])
```

To je torej tisto, kar želimo sešteti. Za bonus navrzimo še največji dvig. In zložimo vse skupaj, da bomo imeli pravi vtis, kako kratek program smo napisali.

[48]:
```
h = np.array([345, 355, 360, 364, 378, 370, 360, 355, 360, 361, 345])
d = h[1:] - h[:-1]
print(np.max(d))
print(np.sum(d[d > 0]))
```

```
14
39
```

### 1.5 Primer: Numpy na dražbi

Zdaj pa se spomnimo domače naloge Dražba: rešili jo bomo s numpy-ja. Videli bomo, kako bo pokazal mišice!

Za začetek uvozimo podatke. O funkcijah, kot je `genfromtxt` se bomo pogovarjali prihodnjič. Tu jo le pokličimo.

[49]:
```
cene = np.genfromtxt("../domace-naloge/02-drazba/drazba.txt", dtype=int)
```

[50]:
```
cene
```

[50]:
```
array([11, 17, 24, 30, -1, 13, 27, 33, -1, 12, 27, 34, 40, -1,  9, -1,  8,
       20, 30, 31, -1])
```

Na prvi dve vprašanji - koliko izdelkov so prodali in koliko je stal najdražji, lahko že odgovorimo.

Prodali so toliko izdelkov, kolikor je `-1`. S `cene == -1` dobimo tabelo `False` in `True`; `True`, kjer so `-1`.

[51]:
```
cene == -1
```

[51]:
```
array([False, False, False, False,  True, False, False, False,  True,
       False, False, False, False,  True, False,  True, False, False,
       False, False,  True])
```

Ker je `True` enak `1` in `False` enak `0`, elemente seznama preprosto seštejemo.

[52]:
```
np.sum(cene == -1)
```

[52]: 5

Cena najdražjega izdelka je kar največja številka v tabeli.

[53]:
```
np.max(cene)
```

[53]:
```
40
```

Odgovor na zadnje vprašanje je pravilen, z estetskega vidika pa nekoliko zmoti, da smo računali maksimum prek vsega, vključno z vmesnimi ponudbami in celo `-1`. Nalogo bomo brez težav rešili tudi elegantneje, saj nas bo v to prisililo naslednje vprašanje: kakšna je vsota cen prodanih izdelkov.

Funkcija `np.flatnonzero` nam vrne tabelo z indeksi ne-ničelnih elementov. Če gre za tabelo `True`-jev in `False`-ov, vrne indekse `True`-jev. V našem primeru bodo to ravno indeksi elementov z vrednostjo `-1`.

[54]:
```
indeksi = np.flatnonzero(cene == -1)

indeksi
```

[54]:
```
array([ 4,  8, 13, 15, 20])
```

Hitro se prepričamo, da so na teh indeksih ravno `-1`-ke.

[55]:
```
cene[indeksi]
```

[55]:
```
array([-1, -1, -1, -1, -1])
```

To je nezanimivo: zanimajo nas ravno elementi pred njimi. Od indeksov torej odštejemo `1`.

[56]:
```
koncne = cene[indeksi - 1]

koncne
```

[56]:
```
array([30, 33, 40,  9, 31])
```

Zdaj lahko ponovno rešimo prvi dve nalogi in še tretjo:

[57]:
```
print(f"Število prodanih predmetov: {len(koncne)}")
print(f"Cena najdražjega predmeta: {np.max(koncne)}")
print(f"Vsota končnih cen: {np.sum(koncne)}")
```

```
Število prodanih predmetov: 5
Cena najdražjega predmeta: 40
Vsota končnih cen: 143
```

Zadnji vprašanji sprašujeta, koliko predmetov je kupila Ana in koliko Berta ter koliko je zapravila katera od njiju.

Na dražbi sta le onidve in prva ponudba je vedno Anina. Za odgovor na vprašanji moramo prešteti, za katere predmete je bilo število ponudb liho in za koliko sodo. Za to pa moramo za začetek prešteti število ponudb za vsak predmet. Pravilni odgovor bo `[4, 3, 4, 1, 4]`.

Število ponudb je (skoraj) enako razlikam med indeksi.

[58]:
```
indeksi
```

[58]:
```
array([ 4,  8, 13, 15, 20])
```

Za prvi predmet so bile dane štiri ponudbe. Naprej gledamo razlike: indeks druge -1 je `8`, indeks prve pa `4`; vmes so bile 8 - 4 - 1 = 3 ponudbe. (Še 1 je potrebno odšteti, ker imamo v tabeli poleg cen še vmesne elemente `-1`.) Naslednja indeksa sta 13 in 8; 13 - 8 - 1 = 4.

Število ponudb za posamični predmet bomo izvedeli, če od

[59]:
```
indeksi
```

[59]:
```
array([ 4,  8, 13, 15, 20])
```

odštejemo

[60]:
```
np.hstack(([-1], indeksi[:-1]))
```

[60]:
```
array([-1,  4,  8, 13, 15])
```

in še `-1`. Na začetek smo dodali -1. Na ta način bomo po odštevanju dobili ravno pravo številko za nesrečni, posebni prvi predmet, saj bomo imeli `4 - (-1) - 1 = 4`. Ne spreglejte tudi dvojnih oklepajev: funkciji `np.hstack` kot argument podamo terko s tabelama, ki jo želimo speti skupaj.

[61]:
```
ponudb = indeksi - np.hstack(([-1], indeksi[:-1])) - 1

ponudb
```

[61]:
```
array([4, 3, 4, 1, 4])
```

Natančno, kar potrebujemo.

Ana je kupila tiste predmete, za katere je bilo število ponudb liho; Berta tiste, za katere je bilo sodo.

[62]:
```
ana = ponudb % 2 == 1
berta = ponudb % 2 == 0

ana
```

[62]:
```
array([False,  True, False,  True, False])
```

[63]:
```
print(f"Ana je kupila {np.sum(ana)}, Berta pa {np.sum(berta)} reči.")
```

```
Ana je kupila 2, Berta pa 3 reči.
```

In koliko je zapravila katera? Tole so cene reči, ki so končale pri Ani:

[64]:
```
koncne[ana]
```

[64]:
```
array([33,  9])
```

Torej, očitno,

[65]:
```
print(f"Ana je zapravila {np.sum(koncne[ana])}, Berta pa {np.
  ↪sum(koncne[berta])}.")
```

```
Ana je zapravila 42, Berta pa 101.
```

### 1.5.1 Vse skupaj

Da se zavemo, kako elegantno kratko je vse skupaj, napišimo celoten program v kosu.

[66]:
```
import numpy as np

cene = np.genfromtxt("../domace-naloge/02-drazba/drazba.txt", dtype=int)
indeksi = np.flatnonzero(cene == -1)
koncne = cene[indeksi - 1]
ponudb = indeksi - np.hstack(([-1], indeksi[:-1])) - 1
ana = ponudb % 2 == 1
berta = ponudb % 2 == 0

print(f"Število prodanih predmetov: {len(koncne)}")
print(f"Cena najdražjega predmeta: {np.max(koncne)}")
print(f"Vsota končnih cen: {np.sum(koncne)}")
print(f"Ana je kupila {np.sum(ana)}, Berta pa {np.sum(berta)} reči.")
print(f"Ana je zapravila {np.sum(koncne[ana])}, Berta pa {np.
  ↪sum(koncne[berta])}.")
```

```
Število prodanih predmetov: 5
Cena najdražjega predmeta: 40
Vsota končnih cen: 143
Ana je kupila 2, Berta pa 3 reči.
Ana je zapravila 42, Berta pa 101.
```