

## Od Numpyja inu še česa

Numpy je ena taka knjižnica za delo s poljubnodimenzijskimi tabelami. V takih časteh jo imajo, da so svojčas modrovali, ne bi li je bilo smiselno med vdelane knjižnice dodati in vkup s Pythonom distribuirati. Ako bi avtor Pythona ne sklenil, da ne bode tako, današnji dan bi ... (Kaj pa, če bi nehal? Prav, bom jenjal. Ni naporno samo brati, tudi pisati je nemogoče. Ne vem, kako je Trubar to zdržal.)

Python je v osnovi počasen in kuri ogromno pomnilnika. Toliko pomnilnika:

```
import sys
sys.getsizeof(3.14)
```

24

```
sys.getsizeof(42)
```

28

28 bajtov za en `int` me pravzaprav nič ne boli. 28 MB za milijon `int`-ov je pa veliko. Ampak milijon `int`-ov bo vedno milijon `int`-ov v seznamu ali neki večdimenzijski tabeli. Podobno je z zankami. Bolijo me zanke, ki se velikokrat obrnejo. To pa so pogosto zanke prek nekih daljših seznamov in tabel.

Numpy reši prvi problem in, v večini primerov, tudi drugega. Zato je del obvezne opreme Pythona. Vsak, ki resno dela s pythonom, bo prej ko slej potreboval tudi numpy.

Numpy navadno uvozimo z

```
import numpy as np
```

To je oblika importa, ki modul ob importu še preimenuje. Namesto da bi ga potem videli pod imenom `numpy`, ga vidimo kot `np`. To je kar konvencija, vsi počnejo tako. Tudi mi bomo.

### Tabele

Osnovni podatkovni tip v `numpy`-ju je `np.array`, to je, poljubno dimenzijska tabela, ki vsebuje elemente istega tipa. (Poljubno dimenzijska: lahko je tudi 0-dimenzijska. 0-dimenzijska tabela je število. :)

Tabelo lahko sestavimo na več načinov. Dobimo jo lahko iz seznama; v spodnjem primeru iz seznama seznamov.

```
np.array([[1, 2, 3], [4, 5, 6]])
```

```
array([[1, 2, 3],
       [4, 5, 6]])
```

Lahko pa sestavimo tabelo samih ničel, enic ali česa drugega, ali pa neinicializirano tabelo. Kot argument podamo terko z dimenzijami. Če terka vsebuje dva elementa, bo tabela dvodimenzionalna. Če pet, bo petdimenzionalna.

```
np.zeros((2, 4))
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.]])

np.ones((2, 4))
array([[1., 1., 1., 1.],
       [1., 1., 1., 1.]])

np.full((2, 4), 42)
array([[42, 42, 42, 42],
       [42, 42, 42, 42]])

np.empty((2, 4))
array([[1., 1., 1., 1.],
       [1., 1., 1., 1.]])
```

Slednje, `np.empty` vrne neinicializirano tabelo. V njej je lahko karkoli, kar se pač slučajno nahaja v koščku pomnilnika, ki je bil dodeljen tabeli. Če dobimo slučajno same enice, je to ... pač slučajno.

Pogosto potrebujemo tudi tabelo zaporednih števil.

```
np.arange(12)
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
```

Ali, recimo, 15 enako razmaknjenih števil med 0 in 3.5.

```
np.linspace(0, 3.5, 15)
array([0.   , 0.25, 0.5  , 0.75, 1.   , 1.25, 1.5  , 1.75, 2.   , 2.25, 2.5  ,
       2.75, 3.   , 3.25, 3.5  ])
```

V gornjih tabelah so bili `int`-i ali `float`-i; kdaj se je zgodilo kaj, vidimo iz izpisa. Če sestavimo tabelo iz seznama ali z `np.full`, je tip odvisen od argumenta. Če je v seznamu kak `float`, bo to tabela `float`-ov, če sami `int`-i, tabela `int`-ov.

Seveda pa

1. Lahko o tem sami odločamo
2. `numpy` ne pozna samo enega `int`-a in enega `float`-a.

Tip določamo z dodatnim argumentom `dtype`. Njegova vrednost je lahko `np.int8`, `np.int16`, `np.int32`, `np.int64`, `np.uint8`, `np.uint16`, ..., `np.float16`, ... OK, razumemo idejo - številka je število bitov, u je unsigned in tako naprej. Samo vseh skalarnih tipov je 24; lahko si ogledate seznam. Tule je tridimenzionalna tabela 16-bitnih floatov.

```

np.ones((2, 4, 3), dtype=np.float16)
array([[[1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.]],

       [[1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.]]], dtype=float16)

```

### Lastnosti tabele; spreminjanje oblike

Mimogrede spoznajmo še en način za ustvarjanje tabele. Kar sprašujemo in izpisujemo zatem, je menda očitno.

```

a = np.random.randint(0, 100, (3, 4))
a
array([[20, 35, 82, 94],
       [73, 67, 42, 52],
       [89,  3, 46, 33]])

```

```
a.ndim
```

```
2
```

```
a.shape
```

```
(3, 4)
```

```
a.size
```

```
12
```

```
a.dtype
```

```
dtype('int64')
```

```
a.itemsize
```

```
8
```

`itemsize` vrne velikost posameznega elementa v bajtih. 64-bitni `int`-i očitno vzamejo 8 bajtov.

Tabele lahko preoblikujemo.

```
b = a.reshape(2, 6)
```

```
b
```

```
array([[20, 35, 82, 94, 73, 67],
       [42, 52, 89,  3, 46, 33]])

```

In tu se začne numpy-jeva magija. `b` je le drugačen pogled na pomnilnik, v katerem je shranjen `a`. Podatki se tu niso kopirali, torej nismo izgubljali ne časa ne dodatnega pomnilnika. A to zdajle še ni tako pomembno, zares bo postalo pomembno ob indeksiranju.

## Operacije nad tabelo

Dvoje sem obljubil v začetku. Prvo: `numpy` bode prijazen do pomnilnika. (Pravzaprav sem obljubil troje: prvo je bilo, da ne bom več imitiral Trubarja.) In vidimo: je prijazen. Vsak element tabele vzame le toliko pomnilnika, kot je potrebno. Drugo: rešil nas bo počasnih Pythonovih zank. To pride zdaj.

Namreč: operacije nad tabelami se izvajajo po elementih, z zanko, ki je napisana v C-ju (v katerem je napisan `numpy`).

```
a = np.arange(12).reshape(3, 4)
```

```
a
```

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

```
a * 10
```

```
array([[ 0, 10, 20, 30],
       [40, 50, 60, 70],
       [80, 90, 100, 110]])
```

To množenje vseh elementov z 10 se je zgodilo tako hitro, kot če bi to naredili v C-ju.

To se da poljubno zaplesti.

```
(a - 5) * 10 + 1000
```

```
array([[ 950,  960,  970,  980],
       [ 990, 1000, 1010, 1020],
       [1030, 1040, 1050, 1060]])
```

```
a ** 2
```

```
array([[ 0,  1,  4,  9],
       [16, 25, 36, 49],
       [64, 81, 100, 121]])
```

Delo po elementih velja tudi za operatorje, kot so `==`, `<` in `>`.

```
a ** 2 > 40
```

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

S tem smo seveda dobili novo tabelo, katere `dtype` je `bool`.

Numpy ima tudi vse žive matematične funkcije. Namesto `math.sqrt`, recimo, pokličemo `np.sqrt`, pa bo kot argument sprejela in pokorenila celo tabelo.

```
np.sqrt(a)
array([[0.          , 1.          , 1.41421356, 1.73205081],
       [2.          , 2.23606798, 2.44948974, 2.64575131],
       [2.82842712, 3.          , 3.16227766, 3.31662479]])

r = np.random.randint(0, 100, (3, 4))
r
array([[51, 12, 16, 75],
       [46, 61, 37, 46],
       [27, 65, 78, 35]])

np.max(r)
78

np.sum(r)
549

np.mean(r)
45.75
```

Funkcije, ki takole seštevajo in podobno "akumulirajo" elemente, lahko izvajamo tudi po oseh". Os 0 je prve dimenzija, os 1 druga, in tako naprej.

```
np.max(r, axis=0)
array([51, 65, 78, 75])

np.max(r, axis=1)
array([75, 61, 78])

np.sum(r, axis=1)
array([154, 190, 205])
```

Spomnimo se, kaj dela operator `>`.

```
t = r > 30
t
array([[ True, False, False,  True],
       [ True,  True,  True,  True],
       [False,  True,  True,  True]])
```

Torej z `np.all(r > 30, axis=1)` izvemo, v katerih vrsticah so vsi elementi večji od 30.

```
np.all(r > 0.3, axis=1)
array([ True,  True,  True])
```

## Operacije med tabelami

Množenje, primerjanje ... tabele s številom je samo poseben primer množenja in primerjanja tabel.

```
a = np.array([2, 6, 3])
b = np.array([8, 2, 1])

a + b
array([10,  8,  4])

a * b
array([16, 12,  3])

a > b
array([False,  True,  True])

a ** b
array([256,  36,   3])
```

Spet: vse te operacije so enako hitre, kot če bi to počeli v C-ju. Saj v resnici počnemo v C-ju. Razlika se seveda pozna predvsem, ko tabele postanejo velike.

Enako se vedejo večdimenzionalne tabele.

```
a = np.array([[4, 5, 1], [1, 2, 4]])
b = np.array([[1, 2, 8], [3, 1, 5]])

a
array([[4, 5, 1],
       [1, 2, 4]])

b
array([[1, 2, 8],
       [3, 1, 5]])

a + b
array([[5, 7, 9],
       [4, 3, 9]])

a * b
array([[ 4, 10,  8],
       [ 3,  2, 20]])

a > b
```

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

a

```
array([[4, 5, 1],
       [1, 2, 4]])
```

K tabeli lahko prištejemo tudi tabelo (ali kar seznam), katerega dimenzija ustreza zadnji dimenziji tabele.

```
a + [100, 200, 300]
```

```
array([[104, 205, 301],
       [101, 202, 304]])
```

Za primer zdaj vzemimo, da imamo koordinate nekih točk na ravnini.

```
k = np.array([
    [5, 2],
    [4, 1],
    [-1, 4],
    [5, 6],
    [0, 2]
])
```

Kakšne so njihove razdalje od koordinatnega izhodišča? Za vsako točko je potrebno izračunati  $\sqrt{x^2 + y^2}$ , pri čemer sta  $x$  in  $y$  prvi in drugi stolpec. Prav. Najprej kvadriramo.

```
k ** 2
```

```
array([[25,  4],
       [16,  1],
       [ 1, 16],
       [25, 36],
       [ 0,  4]])
```

Seštejemo vsako vrstico, da dobimo  $x^2 + y^2$ .

```
np.sum(k ** 2, axis=1)
```

```
array([29, 17, 17, 61,  4])
```

Korenimo.

```
np.sqrt(np.sum(k ** 2, axis=1))
```

```
array([5.38516481, 4.12310563, 4.12310563, 7.81024968, 2.          ])
```

Kaj pa, če nas zanima razdalja točk od (1, 2)? In moramo torej računati  $\sqrt{(x-1)^2 + (y-2)^2}$ ? Pač odštejemo (1, 2) od prvega in drugega stolpca, ne?

```
k - (1, 2)
```

```

array([[ 4,  0],
       [ 3, -1],
       [-2,  2],
       [ 4,  4],
       [-1,  0]])

np.sqrt(np.sum((k - (1, 2)) ** 2, axis=1))
array([4.          , 3.16227766, 2.82842712, 5.65685425, 1.          ])

```

Lepota vsega tega je, spet, učinkovitost. Četudi bi imeli milijon točk, bi se tole izračunalo enako hitro, kot če bi programirali v Cju. Le da bi tam morali alocirati pomnilnik, pisati zanke ... tu pa se vse zanke skrivajo v numpyju, ki, kakor se zdi iz Pythona, dela z vsemi elementi tabele hkrati.

## Indeksiranje, rezanje ... in pogledi

Še zadnji košček sestavljanke, zaradi katere je numpy tako učinkovit in praktičen.

```

a = np.arange(12).reshape(4, 3)
a
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])

```

`a[3]` bo, očitno, vrnil tretjo vrstico matrike.

```

a[3]
array([ 9, 10, 11])

```

Da pridemo do drugega elementa tretje vrstice, bi, če bi bil to Pythonov seznam seznamov, potrebovali dvojne indekse.

```

a[3][2]
11

```

V numpyju pa smemo v iste oglate oklepaje napisati več indeksov.

```

a[3, 2]
11

```

In, najbolj imenitno: ti indeksi so lahko tudi rezine. In celo samo `:`. Začnimo kar z njim.

```

a[3, :] # sicer isto kot a[3] ...
array([ 9, 10, 11])
a[:, 2]

```



```
array([ 2,  5,  8, 11])
```

Razumemo? `a[3, :]` vrne vse, kar je v tretji vrstici, `a[:, 2]` vrne vse (vrstice, ki so) v tretjem stolpcu.

Seveda delujejo tudi običajne rezine.

```
a[1:3]
```

```
array([[3, 4, 5],
       [6, 7, 8]])
```

```
a[1:3, 1:]
```

```
array([[4, 5],
       [7, 8]])
```

Obljubil sem, da bo indeksiranje in rezanje pomagalo k Pythonovi učinkovitosti in praktičnosti. Tole je bila praktično. Kje je učinkovitost? Oh, še začeli nismo!

Prvo, kar se tiče učinkovitosti je: podatki se ne kopirajo. Ko napišemo `a[1:3, 1:]` dobimo le nov pogled na iste podatke.

```
a
```

```
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])
```

```
b = a[1:3, 1:]
```

```
b += 100
```

```
a
```

```
array([[ 0,  1,  2],
       [ 3, 104, 105],
       [ 6, 107, 108],
       [ 9, 10, 11]])
```

Seveda gre tudi brez `b`.

```
a[1:3, 1:] = 0
```

```
a
```

```
array([[ 0,  1,  2],
       [ 3,  0,  0],
       [ 6,  0,  0],
       [ 9, 10, 11]])
```

Kaj pa to: vsako drugo vrstico `a`-ja hočemo spremeniti v `[1, 2, 3]`?

```
a[::2] = [1, 2, 3]
```

```
a
```

```
array([[ 1,  2,  3],
       [ 3,  0,  0],
       [ 1,  2,  3],
       [ 9, 10, 11]])
```

## Indeksiranje tabel s tabelami

Indeksi v numpyju niso nujno samo števila ali rezine. Indeksiramo lahko tudi s tabelo (celih) števil.

```
a = np.random.random(5)
a
array([0.12857635, 0.44611773, 0.80872688, 0.73088645, 0.35649974])
b = np.array([1, 2, 4])
a[b]
array([0.44611773, 0.80872688, 0.35649974])
a[b] = 42
a
array([ 0.12857635, 42.,          , 42.,          , 0.73088645, 42.,          ])
```

No, v resnici ni potrebno da so indeksi tabela. Lahko je tudi običajen seznam.

```
a[[1, 2, 4]] -= 13
a
array([ 0.12857635, 29.,          , 29.,          , 0.73088645, 29.,          ])
```

Če je potrebno, se lahko isti indeks pojavi tudi večkrat.

```
a[[0, 1, 0, 0, 4, 0, 1]]
array([ 0.12857635, 29.,          , 0.12857635, 0.12857635, 29.,          ,
       0.12857635, 29.,          ])
```

Tabele, ki jih dobimo, ko tabelo indeksiramo s tabelo, pa zahtevajo kopiranje podatkov. Drugače ne gre.

Posebej imenitno pa je indeksiranje s tabelo `bool`-ov. Takšna tabela mora biti enako dolga, kot tabela, ki jo indeksiramo. Izbrala bo tiste elemente (vrstice, karkoli), pri katerih imamo vrednost `True`.

```
a = np.array([5, 3, 10, 1, 12])
t = a >= 5
t
array([ True, False,  True, False,  True])
a[t]
array([ 5, 10, 12])
```

V praksi seveda ne delamo tako, temveč pišemo

```
a[a >= 5]
array([ 5, 10, 12])
```

To seveda deluje s poljubno dimenzionalnimi tabelami. Vrnimo se k primeru s točkami. Znali smo izračunati oddaljenost točk od točke (1, 2).

```
k
array([[ 5,  2],
       [ 4,  1],
       [-1,  4],
       [ 5,  6],
       [ 0,  2]])

dist = np.sqrt(np.sum((k - (1, 2)) ** 2, axis=1))
dist
array([4.          , 3.16227766, 2.82842712, 5.65685425, 1.          ])

Zdaj me zanimajo vse točke, ki so oddaljene manj kot 4.

dist < 4
array([False,  True,  True, False,  True])

k[dist < 4]
array([[ 4,  1],
       [-1,  4],
       [ 0,  2]])
```

## Kaj se je še treba naučiti

Zdaj poznamo osnovno orožarno. Odtod naprej se moramo naučiti dvoje.

1. **numpy** ima ogromno funkcij. Sam ga uporabljam že dolgo, a stalno odkrivam nove. Avtorje imam celo na sumu, da jih sproti dodajajo. :) Res pa je, da so bile nekatere, ki jih odkrijem na novo, v **numpy**-ju že od začetka. Očitno bomo **numpy** uporabljali tem boljše, čim več funkcij bomo poznali.
2. Urjenje spretnosti vektorskih operacij. Vektorska operacija je tisto, kar naredimo v Pythonu brez zanke. Recimo **a + b**, če sta **a** in **b** tabeli. Ko programiramo z velikimi podatki in v **numpy**-ju, se za vsako ceno izognemo temu, da bi napisali zanko v Pythonu. Dokler so se zanke v "skrite" v **numpy**-ju, se bodo izvedle v C-ju in bodo hitre.

Prvo in drugo gre seveda z roko v roki. Nekaj izjemno uporabnih funkcij (ki se take morda ne zdijo na prvi pogled) je

- **nonzero**, ki vrne indekse elementov, različnih od 0. Praktičnejše uporabna sestra te funkcije je **flatnonzero**.

```
np.flatnonzero(dist < 4)
array([1, 2, 4])
```

- **argmin**, **argmax** vrnete indekse najmanjših in največjih elementov, **argsort** pa indekse urejene po velikosti elementov. V resnici je **argsort** veliko uporabnejša od **sort**. **sort** ne uporabim skoraj nikoli, **argsort** uporabljam stalno.
- **hstack** in **vstack** sklapljata tabele horizontalno oz. vertikalno, pri čemer morajo biti dimenzije seveda skladne.
- **tranpose** transponira tabelo (vrstice v stolpce in obratno). Enako dosežemo tudi s **.T** (kot npr. **a.T**),
- funkcije povezane z linearno algebro,
- in še veliko drugih.

## Nadaljnje knjižnice

Vse knjižnice, povezane z uporabo Pythona v znanosti, temeljijo na **numpy**-ju.

- **scipy** vsebuje podatkovne strukture za redke matrike (matrike, v katerih so skoraj same ničle, zato jih je možno shranjevati učinkoviteje), funkcije povezane z optimizacijo, analizo signalov (npr. Fourierovo transformacijo), statistiko ...
- **matplotlib** je knjižnica z vsakim grafom, ki si ga poželijo srce. Odlično se ujame tudi z Jupyter notebookom.
- **pandas** je knjižnica za delo s tabelaričnimi podatki, se pravi podatki, kjer stolpci predstavljajo neke spremenljivke, vrstice pa primere. Se pravi nekaj takega, kot navadno vidimo v Excelu ali podatkovnih bazah.
- **scikit-learn** je najbolj razširjena knjižnica za strojno učenje.
- Globoke nevronske mreže so doma v Pythonu. Aktualne odprtokodne knjižnice -- **keras**, **theano**, **Googlov Tensor Flow**, **Facebookov pytorch** -- so bodisi v celoti napisane v Pythonu, ali pa jih je vse možno uporabljati iz Pythona. In tipično temeljijo na **numpy**-ju.

Condo so naredili predvsem, da bi bilo lažje nameščati in nadzorovati vse te mnoge knjižnice. **Anaconda** pa se od **miniconde** razlikuje po tem, da je skupaj z **Anacondo** že poberemo stotine knjižnic, tako da jih ob nameščanju ni potrebno downloadati, saj so že na disku.