

## 2.1 JÄRJEND JA TSÜKKEL

### Järjend tsükli

Kui maatriks on esitatud kahemõõtmelise järjendina, siis märkame, et tegelikult iga maatriksi rida on omakorda eraldi järjend, veerg aga ei ole. Tegelikult saame isegi programmikoodis maatriksi ridu paremini välja tuua:

```
tabel = [  
    [1, 2, 4],  
    [1, 5, 0]  
]
```

Kui see tabel aga ekraanile väljastada (`print(tabel)`), siis ilmub ikkagi halvasti loetavalt `[[1, 2, 4], [1, 5, 0]]`. Tabelikujulise väljundi jaoks võime väljastada tsükli abil iga rea eraldi:

```
for rida in tabel:  
    print(rida)
```

Tsükli igal sammul väljastatakse üks rida järjendina ekraanile. Varasemast teame, et kui tahame iga elemendi ükshaaval järjendist väljastada, siis seda saab teha tsükli abil nii:

```
for element in rida:  
    print(element)
```

Need konstruktsioonid saab aga omavahel kokku panna nii, et iga rea korral väljastataks kõik selle rea elemendid ükshaaval:

```
for rida in tabel:  
    for element in rida:  
        print(element)
```

Siin teeb sisemine tsükkel kogu oma töö ära igal välimise tsükli sammul. Niimoodi kokku pandud tsükleid võib nimetada *kahekordseks tsükliks*.

Eelmise näitekoodi abil saame tõesti kõik elemendid maatriksist (või kahekordsest järjendist) väljastada, kuid tegelikult ei ilmu nad ekraanile siiski tabelina. Põhjus on selles, et funktsioon `print` väljastab pärast igat väljakutset lisaks elemendile ka reavahetuse. Seda käitumist saab muuta nii, et anda `print` funktsioonile *nimelise argumenina* nimega `end` ette sõne, mis tuleb alati lõpuks väljastada.

```
for rida in tabel:  
    for element in rida:  
        print(element, end=" ")
```

Selle programmijupi käivitades ilmuvad elementide vahele reavahetuste asemel tühikud. Samuti võiks me `end=" "` asemel kirjutada näiteks `end=""`, et elementide vahele pandaks tühjad sõned (ehk ei pandaks mitte midagi) või hoopis midagi muud - kõik sõned on sinna lubatud. Huvi korral saate rohkem lugeda funktsiooni `print` võimaluste kohta [Pythoni dokumentatsioonist](#).

Eelmist programmi käivitades ilmuvad elemendid küll ilusate vahedega, kuid erinevate ridade elemendid väljastatakse ikkagi samale reale. Selle parandamiseks saame väljastada tavalise `print` funktsiooniga ühe reavahetuse iga kord, kui minnakse elementide vaatamisel järgmise rea juurde:

```
for rida in tabel:  
    for element in rida:  
        print(element, end=" ")  
    print()
```

Siin on reavahetuse väljastav `print` just välimise tsükli sees, kuid mitte sisemise. Nii rakendub see ainult siis, kui sisemine tsükkel on kõik mingi antud rea elemendid väljastanud (tühikutega eraldatult) ja välimisel tsükliil on aeg minna järgmise rea (välimise tsükli uue sammu) juurde.

Kui tahame arvestada ka asjaolu, et elemendid võivad tabelis olla erineva pikkusega, siis võib tühiku asemel kasutada tabulatsioonimärki `"\t"` või huvi korral võite uurida käsu [rjust](#) kohta.

## Tsükkel ja indeksid

Eespool kasutasime Pythoni mugavaid `for`-tsükli võimalusi, et järjendit elementhaaval läbida ilma indeksite peale mõtlemata. Sarnaselt saaks leida kõigi antud arvude korrutise:

```
a = [1, 6, 3, 2, 1, 8, 3, 2]  
korrutis = 1 # 0 jätkaks korrutise 0-ks  
for el in a:  
    korrutis *= el  
print(korrutis)
```

Selline viis sobib hästi siis, kui midagi tuleb teha järjendi kõigi elementidega. Sageli on aga vaja tegeleda ainult teatud kohtadel paiknevate elementidega ja sellisel juhul on kasu indeksitest. Järgmine programm on eelmisega samaväärne, aga elementide krabamiseks järjendist kasutatakse indekseid.

```
a = [1, 6, 3, 2, 1, 8, 3, 2]
korrutis = 1
for i in range(len(a)):
    korrutis *= a[i]
print(korrutis)
```

`range` funktsiooni abil omandab tsüklimuutuja `i` järjest väärtusi lõigust `0` kuni `len(a) - 1`. Vajadusel tuletage meelde `range`-funktsiooni [eelmise kursuse materjalidest](#). Antud juhul saab muutuja `i` väärtuseks järjest `0`, `1`, `2`, `3`, `4`, `5`, `6` ja `7`.

Väga oluline on siin vahet teha indeksil `i` ja vastaval elemendil `a[i]`.

### Enesetest:

Mida teeb antud programm?

```
b = [1, 9, 1, 8]
korrutis = 1
for i in range(len(b)):
    korrutis *= i
print(korrutis)
```

Vali Leiab järjendi `b` kõigi elementide korrutise.

Vali Leiab kõikide `i` väärtuste korrutise.

Vali Annab veateate.

### Enesetest:

Mis arvud korrutab antud programm?

```
a = [1, 6, 3, 2, 1, 8, 3, 2]
korrutis = 1
for i in range(0, len(a), 2):
    korrutis *= a[i]
print(korrutis)
```

Vali 1, 3, 1, 3

Vali 1, 6, 3

Vali Kõik järjendis olevad arvud.

Vali Mingid muud.

### Kahekordne tsükkel ja indeksid

Samuti saame indeksite abil läbida kahekordseid järjendeid. Siis võetakse sageli (aga mitte alati) tsüklimuutujateks `i` ja `j`. Tabeli väljastamise saaksime indeksite abiga teha nii:

```
for i in range(len(tabel)):
    for j in range(len(tabel[i])):
        print(tabel[i][j], end=" ")
    print()
```

Välimine tsükkel käib ikka ridahaaval, `i` saab väärtusi lõigust `0` kuni `len(tabel) - 1`. Sisemine tsükkel võtab tol hetkel vaadeldava rea (`tabel[i]`) elemente (`tabel[i][j]`) vastavalt `j` väärtustele, järjest `0` kuni `len(tabel[i]) - 1`. See programm töötab ka siis, kui read ei ole ühepikkused - iga rea korral muutub `j` just vastavalt konkreetse rea pikkusele.

### Enesetest:

Mis ilmub ekraanile?

```
tabel = [[1, 3, 10], [], [2, 7, 5]]
esimesed = []

for i in range(len(tabel)):
    for j in range(len(tabel[i])):
        if j == 0:
            esimesed.append(tabel[i][j])
print(esimesed)
```

Vali [1, 3, 10]

Vali [1, 2]

Vali Veateade

Vahel on vaja teada, kus täpselt teatud tingimustele vastav element on. Järgmine programm väljastab ekraanile negatiivsete arvude asukohad (indeksid).

```
tabel = [[1, -4, 5], [-4, 6, 7], [5, 6, -7]]
for i in range(len(tabel)):
    for j in range(len(tabel[i])):
        if tabel[i][j] < 0:
            print("Negatiivne arv " + str(tabel[i][j]) +
                  " on kohal (" + str(i) + "; " + str(j) + ").")
```

### Enesetest:

Mida teeb antud programm?

```
tabel = [[1, -4, 5, 3, 7], [-4, 6, 7, 2, 3], [5, 6, -7, -1, 4]]
a = 0

for i in range(len(tabel)):
    for j in range(len(tabel[i])):
        if i == 1 and tabel[i][j] < 5:
            a += 1
print(a)
```

- Vali Loeb, mitu arvu on teises reas väiksemad kui 5.
- Vali Loeb, mitu arvu on esimeses reas väiksemad kui 5.
- Vali Liidab kokku kõik teises reas olevad arvud, mis on väiksemad kui 5.
- Vali Annab veateate.

### Enesetest:

Mida teeb antud programm?

```
tabel = [[1, -43, 5, 3, 7], [-4, 6, 7, 2, 3], [5, 6, -7, -1, 4]]
lst = []

for i in range(len(tabel)):
    loendaja = 0
    for j in range(len(tabel[i])):
        if tabel[i][j] % 2 == 0:
            loendaja += 1
    if loendaja >= 2:
        lst += [i]

print(lst)
```

- Vali Väljastab järjendi tabeli sisemiste järjendite indeksitest, milles on vähemalt 2 paarisarvu.
- Vali Väljastab järjendi kõikidest tabelis asuvatest paarisarvudest.
- Vali Väljastab paarisarvude koguarvu.
- Vali Midagi muud.

## 2.2 JÄRJEND JA FUNKTSIOON

### Järjend funktsiooni argumendina

Programmide kirjutamisel on mõistlik püüda tööd jaotada erinevatest osadeks - alamprogrammideks ehk funktsioonideks. Funktsiooni teemat saate vajadusel korrata [eelmise kursuse materjalidest](#). Pythonis on juba mitmeid defineeritud funktsioone, millele saab anda argumendiks järjendi, näiteks `max`, `min` ja `len`, aga ka näiteks `print`, kuigi talle saab ka paljusid teisi tüüpe argumendiks anda.

```
a = [2, -3, 5, 1]
print(max(a))
print(min(a))
print(len(a))
print(a)
```

Järgmine funktsioon kontrollib, kas esimese argumendina antud järjendis on elemente, mis on suuremad teisest argumendist. Kui on, siis tagastatakse tõeväärtus `True` ja kui pole, siis tõeväärtus `False`.

```
def on_suuremaid(jarjend, piir):
    for i in range(len(jarjend)):
        if jarjend[i] > piir:
            return True
    return False
```

Oluline on tähele panna, et seda, et piirist suuremaid elemente ei leidu (`return False`) saame öelda alles siis, kui kõik on läbi vaadatud. Leidumist saame kinnitada kohe, kui sellise leiame.

Pange tähele, et tegelikult saaks sama ülesannet lihtsamini lahendada funktsiooni `max` abil. Kui maksimaalne element järjendist on piirist suurem, siis järelikult peaks tulemus olema tõene, vastasel juhul väär:

```
def on_suuremaid(jarjend, piir):
    return max(jarjend) > piir
```

### Enesetest:

Mis ilmub ekraanile?

```
tabel = [[1, -43, 5, 3, 7], [-4, 6, 7, 9, 8], [5, 6, -7, -1, 4]]

loendaja = 0
for el in tabel:
    if on_suuremaid(el, 6):
        loendaja += 1

print(loendaja)
```

Vali 2

Vali 4

Vali 3

Vali Veateade

Funktsioonina võime realiseerida ka mõne varemtoodud konstruktsiooni. Tabelit väljastava funktsiooni puhul ei tagastata midagi (tegelikult tagastatakse `None`). Küll aga toimub väljastamine.

```
def valjasta_tabel(tabel):
    for i in range(len(tabel)):
        for j in range(len(tabel[i])):
            print(tabel[i][j], end=" ")
        print()

arvude_tabel = [[1, 3, 5], [4, 6, 6], [3, 6, -3]]
valjasta_tabel(arvude_tabel)
print()
arvude_tabel2 = [[-1, 3, 5], [4, -8, 6]]
valjasta_tabel(arvude_tabel2)
print()
riimitabel = [['karu', 'maru', 'taru'], ['haru', 'varu', 'naru']]
valjasta_tabel(riimitabel)
```

Loomulikult võib funktsioon ka mingi väärtuse tagastada. Näiteks võib loendada, mitu positiivset elementi on esimese argumendina etteantud tabeli teise argumendina etteantud indeksiga veerus:



```
def positiivsete_arv_veerus(tabel, veeru_indeks):  
    loendaja = 0  
    for rida in tabel:  
        if rida[veeru_indeks] > 0:  
            loendaja += 1  
    return loendaja
```

### Enesetest:

Mida teeb antud funktsioon?

```
def fun(tabel, rea_indeks):  
    loendaja = 0  
    for el in tabel[rea_indeks]:  
        if el > 0:  
            loendaja += 1  
    return loendaja
```

Vali Tagastab positiivsete arvu antud indeksiga veerus.

Vali Tagastab positiivsete arvu tabelis.

Vali Tagastab positiivsete arvu antud indeksiga reas.

Vali Tagastab midagi muud.

### Järjend funktsiooni väärtusena

Funktsioon võib tagastada samuti järjendi, kasvõi kahemõõtmelise järjendi.

Harjutamiseks koostame funktsiooni, millele antakse argumentidena kaks arvu: esimene näitab, kui suur ruutmatriks tehakse ja teine, millega täidetakse peadiagonaal. Tahame, et funktsiooni rakenduse `loo_diagonaalmaatriks(3, 1)` väärtuseks oleks `[[1, 0, 0], [0, 1, 0], [0, 0, 1]]`.

```
def loo_diagonaalmaatriks(jark, sisu):
    maatriks = []
    for i in range(jark):      # välimine tsükel tekitab ridu
        rida = []
        for j in range(jark):  # sisemine hoolitseb veergude eest
            if i == j:         # tegemist on peadiagonaali
elemendiga
                rida.append(sisu)
            else:
                rida.append(0)
        maatriks.append(rida)
    return maatriks
```

Matemaatiliselt pole diagonaalmaatriksil tegelikult piirangut, et kõik peadiagonaali elemendid peaksid võrdsed olema.

### Enesetest:

Millega tuleks asendada rida `if i == j:`, et peadiagonaali asemel täidetakse nii pea- kui kõrvaldiagonaal etteantud elemendiga?

```
def loo_diagonaalmaatriks(jark, sisu):
    maatriks = []
    for i in range(jark):      # välimine tsükel tekitab ridu
        rida = []
        for j in range(jark):  # sisemine hoolitseb veergude eest
            if i == j:         # tegemist on peadiagonaali elemendiga
                rida.append(sisu)
            else:
                rida.append(0)
        maatriks.append(rida)
    return maatriks
```

Vali `if i == j or i == jark - j:`

Vali `if i == j or i == jark - 1:`

Vali `if i == j or i == jark - j - 1:`

Vali Ükski eelnev variant ei ole õige.

### Enesetest:

Millises järjekorras tuleb panna sisemine ja välimine for-tsükkel, et saaksime kätte veerud?

```
tabel = [[3, 2, 5], [9, 5, 1], [4, 5, 2]]
```

1)...

2)...

```
print(tabel[rida][veerg])  
print()
```

Vali `1) for veerg in range(len(tabel[0]),2) for rida in range(len(tabel)):`

Vali `1) for rida in range(len(tabel)),2) for veerg in range(len(tabel[0])):`

Vali Mingis muus järjekorras.

Vali Nende tsüklitega ei ole võimalik veergudele ligi pääseda.

### Enesetest:

Missugune tingimuslause tuleks asendada lünka, et väljastataks ruutmatriksi tabel nurkmised elemendid?

```
for i in range(len(tabel)):  
    for j in range(len(tabel[i])):  
        ...  
        print(tabel[i][j])
```

Vali `if i == 0 or i == len(tabel)-1 or j == 0 or j == len(tabel[i])-1:`

Vali `if (i == 0 or i == len(tabel)-1) and (j == 0 or j == len(tabel[i])-1):`

Vali `if i == 0 and i == len(tabel)-1 or j == 0 and j == len(tabel[i])-1:`

Vali `if i == 0 and i == len(tabel)-1 and j == 0 and j == len(tabel[i])-1:`

Vali Ükski eelnev ei sobi.

## 2.3 REAALSETE ANDMETE KASUTAMINE

### Andmed failist

Seni oleme tegutsenud kahemõõtmeliste järjenditega, mille elemendid kirjutasime programmiteksti sisse, näiteks:

```
tabel = [[1, 2, 4], [-1, 5, 0]]
```

või käsureale, kui funktsiooni rakendasime, näiteks:

```
>>> on_bingo_tabel([[1, 30, 34, 55, 75], [10, 16, 40, 50, 67],  
[5, 20, 38, 48, 61], [4, 26, 43, 49, 70], [15, 17, 33, 51, 66]])
```

Vähegi suuremate mahtude juures on see ebamugav ja praktikas ka tihti võimatu, sest tahame sama programmi abil töödelda palju erinevaid andmeid. Failist lugemist käsitlesime [eelmisel kursusel](#), aga nüüd vaatame, kuidas failist andmed mugavalt kahemõõtmelisse järjendisse saada.

Olgu näiteks õpilaste saadud (täisarvulised) punktid iga õpilase korral failis eraldi ridadel tühikutega eraldatult. Näiteks:

```
1 4 5  
3 6 8  
2 4 8  
10 5 0
```

### Enesetest:

Olgu failis `andmed.txt` samad andmed nagu eelnevas näites. Mis ilmub sellise programmi käivitamisel ekraanile?

```
fail = open("andmed.txt", encoding="UTF-8")
read = []
for rida in fail:
    read.append(rida)
fail.close()

print(read)
```

Vali ['1 4 5 \n', '3 6 8 \n', '2 4 8 \n', '10 5 0 \n']

Vali ['1 4 5', '3 6 8', '2 4 8', '10 5 0']

Vali ['145\n', '368\n', '248\n', '1050\n']

Vali [[1, 4, 5], [3, 6, 8], [2, 4, 8], [10, 5, 0]]

Vali Veateade

Kahemõõtmelisse järjendisse saab need andmed aga näiteks nii:

```
punktide_tabel = []

for rida in fail: # iga rea jaoks failist
    op_punktid = [] # kogume ühe õpilase punkte
    osad = rida.split()
    for osa in osad: # osade kaupa
        op_punktid.append(int(osa))

    punktide_tabel.append(op_punktid)
```

## Enesetest:

Kuidas tuleks muuta järgmist tekstifaili, et eelnev programm annaks veateate?

```
1 4 5
3 6 8
2 4 8
10 5 0
```

- Vali Lisada ühele reale üks element juurde.
- Vali Kustutada kogu sisu ning käivitada programm tühja failiga.
- Vali Lisada kahe rea vahele tühi rida.
- Vali Muuta vähemalt üks arv sõnaks/täheks.
- Vali Tekstifaili muutmine ei saa põhjustada programmi töös vigu.

Eelmises näites olid ühel real olevad andmed eraldatud tühikutega. Ajalooliselt on eraldajaks olnud ka näiteks koma. Sellest tuleb ka lühend CSV - [ingl Comma-separated values](#). Tegelikult ei pruugi koma alati hästi eraldajaks sobida, kuna võib andmetes tähendada midagi muud, näiteks komadega arve. Levinumad eraldajad on tabulatsioonimärk `"\t"` ja semikoolon. Ka meie edasistes näidetes on eraldajaks just semikoolon.

## Reaalsete andmete kasutamine

Väga palju andmeid elust enesest on tegelikult täiesti vabalt kättesaadavad ja kasutatavad. Eesti kohta leiab huvitavaid andmeid näiteks [statistikaameti veebilehelt](#). Haridusteemalisi andmeid saab [Haridussilmast](#).

Näiteks leidub statistikaameti kodulehel andmeid Eesti rahvastiku soolisest koossesisust erinevates vanuserühmades.

RV021: RAHVASTIK, 1. JAANUAR --- Aasta, Sugu ning Vanuserühm																				
	1-4	5-9	10-14	15-19	20-24	25-29	30-34	35-39	40-44	45-49	50-54	55-59	60-64	65-69	70-74	75-79	80-84	85-89	90-94	95-99
2016																				
Mehed	28 950	39 323	32 940	30 876	38 215	51 769	48 625	46 048	46 188	42 355	41 991	40 928	36 463	29 973	20 131	17 867	10 232	5 122	1 315	132
Naised	27 606	37 269	31 310	29 062	36 274	47 371	45 658	43 888	44 874	43 499	45 243	47 800	47 047	43 836	34 117	37 307	26 198	16 954	6 107	906

Selle tabeli põhjal saaks lahendada erinevaid ülesandeid. Näiteks saab leida:

- mis vanuses elanikke on kõige rohkem,
- mis vanuserühmas on meeste ja naiste arvuline erinevus kõige suurem,
- teatud vanuserühma elanike arvu.

Haridussilmast võiksime leida näiteks nelja ülikooli vilistlaste keskmised palgad erialade kaupa. Haridussilmast saab andmeid alla laadida küll ainult (Exceli) xls-formaadis, kuid tabelitöötlusprogrammis saab selle faili ka csv-formaati salvestada.

Lõpetamise aasta	Õppeasutus	Sugu	Õppease	Õppeasund	Lõpetajate sündmuste arv	isikute arv, kelle jaoks see on kõrgem haridus	isikute arv, kelle sissetulekut arvestati	Keskmine sissetulek
2	2006-2013	Eesti Maalikool	KOKKU	KOKKU	6830	5494	4324	1004
3	2006-2013	Eesti Maalikool	KOKKU	KOKKU	918	771	643	1097
4	2006-2013	Eesti Maalikool	KOKKU	KOKKU	149	115	87	978
5	2006-2013	Eesti Maalikool	KOKKU	KOKKU	1	1	1	930
6	2006-2013	Eesti Maalikool	KOKKU	KOKKU	1260	967	720	930
7	2006-2013	Eesti Maalikool	KOKKU	KOKKU	1918	1431	1104	1068
8	2006-2013	Eesti Maalikool	KOKKU	KOKKU	700	584	517	1069
9	2006-2013	Eesti Maalikool	KOKKU	KOKKU	130	126	104	1069
10	2006-2013	Eesti Maalikool	KOKKU	KOKKU	11	10	9	1454
11	2006-2013	Eesti Maalikool	KOKKU	KOKKU	276	275	136	1198
12	2006-2013	Eesti Maalikool	KOKKU	KOKKU	18	16	12	895
13	2006-2013	Eesti Maalikool	KOKKU	KOKKU	1368	1200	991	1109
14	2006-2013	Tallinna Tehnikaülikool	KOKKU	KOKKU	16063	13451	10763	1458
15	2006-2013	Tallinna Tehnikaülikool	KOKKU	KOKKU	1234	1177	1024	1448
16	2006-2013	Tallinna Tehnikaülikool	KOKKU	KOKKU	2411	1963	1584	1849
17	2006-2013	Tallinna Tehnikaülikool	KOKKU	KOKKU	389	283	206	1226
18	2006-2013	Tallinna Tehnikaülikool	KOKKU	KOKKU	508	353	277	1126
19	2006-2013	Tallinna Tehnikaülikool	KOKKU	KOKKU	107	101	71	895
20	2006-2013	Tallinna Tehnikaülikool	KOKKU	KOKKU	168	136	102	1000
21	2006-2013	Tallinna Tehnikaülikool	KOKKU	KOKKU	862	738	565	1291
22	2006-2013	Tallinna Tehnikaülikool	KOKKU	KOKKU	3656	2808	2351	1543
23	2006-2013	Tallinna Tehnikaülikool	KOKKU	KOKKU	1200	969	744	1135
24	2006-2013	Tallinna Tehnikaülikool	KOKKU	KOKKU	463	362	297	1422
25	2006-2013	Tallinna Tehnikaülikool	KOKKU	KOKKU	362	340	262	1290
26	2006-2013	Tallinna Tehnikaülikool	KOKKU	KOKKU	149	129	118	1340
27	2006-2013	Tallinna Tehnikaülikool	KOKKU	KOKKU	4464	4093	3162	1402
28	2006-2013	Tallinna Ülikool	KOKKU	KOKKU	13622	11777	9695	1071
29	2006-2013	Tallinna Ülikool	KOKKU	KOKKU	646	577	487	1118
30	2006-2013	Tallinna Ülikool	KOKKU	KOKKU	301	274	229	1547
31	2006-2013	Tallinna Ülikool	KOKKU	KOKKU	250	163	133	1065
32	2006-2013	Tallinna Ülikool	KOKKU	KOKKU	238	156	120	973
33	2006-2013	Tallinna Ülikool	KOKKU	KOKKU	2029	1629	1193	926
34	2006-2013	Tallinna Ülikool	KOKKU	KOKKU	716	639	490	1011
35	2006-2013	Tallinna Ülikool	KOKKU	KOKKU	207	168	132	1053

Järgmise näite puhul püüame teada saada, mis aastatel on Tartu elanike arv kasvanud. Statistikaameti [veebilehelt](#) saab Tartu elanike arvud kätte järgmiste valikutega: *Statistika andmebaas -> Rahvastik -> Rahvastikunäitajad ja koosseis -> Rahvaarv ja rahvastiku koosseis -> RV0282 Rahvastik soo, vanuserühma ja haldusüksuse või asustusüksuse liigi järgi, 1. jaanuar* ([otselink](#)).

Andmete salvestamisel on võimalik valida, mis kujul me neid faili tahame. Hetkel on sobiv *Ilma pealkirjata Semikooloneeraldusega pealkirjata tekst (.csv)*. Nii saame faili [RV0282sm.csv](#), mille esimene rida on

"..Tartu linn";"Mehed ja naised";"2000";106200

Näeme, et meid eriti huvitavad andmed on rea lõpus mõnevõrra erineval kujul - aastaarvul on jutumärgid ümber ja elanike arvul mitte. Kui nüüd eeltoodud moel fail sisse lugeda ja rida osadeks jaotada, siis saame, et `osad[2]` on sõne `"2000"` ja `osad[3]` on sõne `"106200"`. Meie tahame neid mõlemaid täisarvudena, mida annab funktsioon `int`. Küll aga tuleb `"2000"` puhul arvu ümbert sõnesisesed jutumärgid eemaldada, mida saame teha funktsiooni `strip` abil: `osad[2].strip()`.

Järgmises programmis paneme saadud andmed järjendisse nii, et ühes reas (`andmed[0]`) on kõik aastaarvud ja teises (`andmed[1]`) kõik elanike arvud. Failis olid andmed eri ridadel.

```
fail = open("RV0282sm.csv", encoding="UTF-8")
andmed = [[], []]
for rida in fail: # loeme ridahaaval
    osad = rida.split(";") # semikoolonitega eraldatud sõne
    järjendiks
        andmed[0].append(int(osad[2].strip('"')))
        andmed[1].append(int(osad[3]))
fail.close()
```

Kui nüüd tahame leida aastad, mil elanike arv kasvas, siis võrdleme iga aasta korral, kas järgmise aasta elanike arv on suurem. Kuna andmed on 1. jaanuari seisuga, siis täpselt ühe aasta kaupa muudatusi saamegi nii jälgida.

```
for i in range(len(andmed[0]) - 1):
    if andmed[1][i + 1] > andmed[1][i]:
        print(andmed[0][i])
```

Siin on oluline, et tsükkel lõpeb eelviimase elemendiga, sest viimasel elemendil poleks järgmist (`andmed[1][i + 1]`), millega võrrelda.

Kuna CSV-failidega tegutsemist võib ikka ette tulla, siis on selleks tarbeks olemas spetsiaalne Pythoni moodul, mille nimi ongi `csv`, millega võite huvi korral ise tutvuda. Tegemist on Pythoni standardmooduliga, mille kasutamine tuleb programmi alguses näidata `import csv` abil. Selle abil saaks sama programmi kirjutada näiteks nii:

```
import csv

andmed = [[], []]

csvfail = open('RV0282sm.csv', encoding='UTF-8')
loetudCSV = csv.reader(csvfail, delimiter=';')

for rida in loetudCSV:
    andmed[0].append(int(rida[2]))
    andmed[1].append(int(rida[3]))

csvfail.close()

for i in range(len(andmed[0]) - 1):
    if andmed[1][i + 1] > andmed[1][i]:
        print(andmed[0][i])
```



## 2.4 JUHUSLIKKUS

### Mis on juhuslik?

```
def juhuslik_arv():  
    # Valitud ausa täringuviske põhjal, juhuslikkus garanteeritud  
    return 4
```

Eelnev programmijupp on tuntud programmeerijate anekdoot juhuslikkuse kohta ([ingliskeelne originaal](#)).

Nali seisneb selles, et tavakeeles tähendab juhuslikkus tihti midagi muud kui programmeerijate jaoks. Kui paluda kaaslasel öelda üks juhuslik arv, siis mida saame tema valiku kohta öelda? Mis teeb sellest valikust juhusliku valiku? Kui laseme tal teist korda valida ja ta valib sama arvu, kas siis tema valik polegi juhuslik? Või oli esimene valik juhuslik ja teine mitte?

Mida mõtleme, kui ütleme, et mingi sündmus on juhuslik? Loomulik tundub, et sündmus võiks olla juhuslik siis, kui me ei osanud enne toimumist seda ette ennustada. Sel juhul paistab justkui sündmuse juhuslikkus sõltuks vaatelejust - oleks subjektiivne. Või hoopis on juhuslikud ainult need sündmused, mida on fundamentaalselt võimatu ette ennustada, olenemata vaatelejust?

### Juhuslikkuse eesmärgid

Programmeerimises eristatakse tihti mitmeid erinevaid juhuslikkuse tüüpe sõltuvalt rakendustest.

Tuttav Pythoni funktsioon `randint` genereerib juhusliku täisarvu antud lõigust. Kuid kuidas suudab arvuti kui deterministlik ja rangelt defineeritud käitumisega masin tekitada midagi juhuslikku? Ei suudagi, vähemalt mitte otseselt. Selle funktsiooni realiseerimiseks kasutatakse kavalaid matemaatilisi ja statistilisi võtteid, et jätta programmeerijale või rakenduse kasutajale mulje justkui tegemist oleks juhusliku arvuga. Tegelikult tekib see arv läbi keeruliste (kuid deterministlike) valemite või algoritmide, mis võtavad sisendiks mingi väikese hulga infot ja muundavad selle näivaks juhuslikkuseks. Selliset juhuslikkust nimetatakse *pseudojuhuslikkuseks*, kusjuures eelnevalt mainitud sisendinfot kutsutakse juhuslikkuse *seemneks* (ingl *random seed*).

Pythonis on pseudojuhuslikkuse generaatorina kasutusel algoritm nimega Mersenne Twister, konkreetse Pythoni implementatsiooniga on võimalik tutvuda [siin](#). Juhuslikkuse seemnest räägitakse tavaliselt mitte kui sõnest ega arvust, vaid bittide järjestist, sest bitid on kõige fundamentaalsemad informatsiooniühikud. Seemnena kasutatakse pseudoarvude generaatorite juures tihti väärtuseid, mis väga tihti muutuvad ja millel on palju erinevaid

olekuid. Python kasutab selles rollis kas arvuti operatsioonisüsteemi poolt pakutud juhuslikkuse allikat või sekundite arvu, mis on möödunud 1. jaanuarist 1970. a (vt [Forte: UNIXi ajaarvamine jõuab 1234567890ni](#) või ingl [Unix time](#)).

Enamik operatsioonisüsteeme pakuvad programmidele kasutamiseks juhuslikku infot, mis on pärit mingist füüsilisest allikast. Näiteks võib süsteem mõõta miniatuurseid temperatuuri kõikumisi mõne riistvarakomponendi juures. Sellist tüüpi juhuslikkust nimetatakse *tõeliseks juhuslikkuseks* ning seda iseloomustab informatsiooni kordumatus. Kui pseudojuhuslikult genereeritud arve on võimalik seemne põhjal uuesti tekitada, siis tõeline juhuslikkus ei sõltu mingist teadaolevast infost (seemnest). Tõelist juhuslikkust tekitatakse ka näiteks, mõõtes atmosfäärilist müra või muid füüsikalisi nähtuseid. Tõeliselt juhuslike [arvude](#) või [sõnede](#) genereerimist pakub tasuta random.org. [Krüptograafiliste rakenduste](#) juures on äärmiselt oluline, et kasutatav juhuslikkus oleks just tõeline, mitte pseudojuhuslikkus. Vastasel juhul võib ründajal olla võimalus näivald juhuslike andmeid ära arvata.

## Erinevad jaotused

Vahel arvatakse juhuslikkusest rääkides, et võimalikud sündmused või väärtused peavad olema võrdvõimalikud, kuid see ei pruugi sugugi nii olla. Kui viskame palju kordi tavalist kuuetahtulist täringut, siis näeme tulemusi kokku võttes, et iga väärtus 1-6 esines keskmiselt sama palju kordi. Mis juhtub aga, kui viskame kahte kuuetahtulist täringut korraga ja liidame saadud silmade arvud kokku?

```
from random import randint

# Tekitame 12-elementilise listi, kus iga element on 0
vaartused = [0] * 12

for i in range(10000):
    taring_1 = randint(1, 6)
    taring_2 = randint(1, 6)
    summa = taring_1 + taring_2
    # Lisame ühe juurde elementidele, mis tähistab saadud tulemust
    vaartused[summa-1] += 1

print(vaartused)
```

Enam pole tulemuste jaotus üldse ühtlane, vaid palju rohkem on tulnud tulemusi, mis jäävad keskmiste väärtuste hulka. See on ka intuiitvne, sest summa 2 saamiseks on ainult üks võimalus, kuidas täringud peavad jääma: 1+1. Aga näiteks summa 7 saamiseks on variante lausa kuus: 1+6, 2+5, 3+4, 4+3, 5+2, 6+1.

Samad ideed on kasutusel ka loteriides ja hasartmängudes. Näiteks Bingo Loto puhul on iga palli loosimine võrdvõimalik, kuid õnneratta keerutamisel on ratta peal väiksemaid summasid vähem kui suuremaid.