1.1 SISSEJUHATUS

Järjend on Pythoni andmestruktuur, mille abil saab ühes muutujas hoida mitut väärtust. Näiteks muutujale `kursus` omistatakse järjend, mis koosneb kahest sõnest:

```
kursus = ['Programmeerimise alused', '2']
```

Varem oleme kasutanud peamiselt järjendeid, mille elemendid on arvud või sõned. Tegelikult võime järjendis hoida igasugust tüüpi väärtuseid, kusjuures järjendi elemendid võivad isegi eri tüüpi olla:

```
huvitav_jarjend = ['Taisi Telk', 29, True]
```

Muuhulgas võivad järjendi elementideks olla ka teised järjendid:

```
kahemootmeline_jarjend = [[30, 2], [42]]
```

Siin on tegemist järjendiga, millel on kaks elementi. Esimene element on järjend, mille elementideks on arvud `30` ja `2` ning teine on järjend, mille ainsaks elemendiks on `42`.

See on meie esimene kahemõõtmeline andmestruktuur. Kahemõõtmeliseks nimetatakse seda sellepärast, et sisemiste elementide (eelmises näites `30`, `2`, `42`) asukoht määratakse kahe mõõtme abil. Esiteks tuleb määrata, mitmendas sisemises järjendis element asub ja teiseks, mitmes element ta on selle väljavalitud sisemise järjendi elementide hulgas. Seejuures peame meeles, et järjendi elemente nummerdatakse indeksite abil ja indeksid algavad alati nullist. Näiteks kui tahame eelmises näites tekitatud järjendist saada kätte väärtust `2`, siis leiame, et ta asub sisemises järjendis, mille indeks välimise suhtes on `0` ja selles `0`-ndas sisemises järjendis on ta element indeksiga `1`. Koodis pannakse see kirja niimoodi:

```
kahemootmeline_jarjend = [[30, 2], [42]]
print(kahemootmeline_jarjend[0][1])  # Väljastab 2
```

Kahemõõtmelise järjendi sisemiste elementide poole pöördumiseks paigutatakse pärast muutuja nime kaks paari nurksulge. Esimesed nurksulud sisaldavad indeksit, mis määrab vaadeldava sisemise järjendi. Teiste nurksulgude sisse kirjutatakse indeks, mis määrab elemendi asukoha saadud sisemises järjendis.

Kahemõõtmelist järjendit võime visualiseerida ka kahe ruumimõõtme abil. Sel juhul võiks esimesest indeksist mõelda kui rea indeksist ja teisest kui veeru indeksist. Siis näeb meie näitejärjend välja selline:

```
30 2
42
```

Elemendi väärtusega `2` leiab tõesti sellest tabelist realt indeksiga `0` ja veerult indeksiga `1`.

Enesetest:

Mis väljastatakse ekraanile?

```
tulemused = [[3, -5, 5], [-4, -9, 2], [1, 0, -2], [4, 10, -3], [-1, 9, -6]]
print(tulemused[2][1])
```

Vali -4

Vali 0

Vali 2

Vali Midagi muud

Enesetest:

Mis tüüpi elemendid väljastatakse ekraanile?

```
lst = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
for el in lst:
  print(el)
```

Vali Järjendid/listid

Vali Täisarvud

Vali Ujukomaarvud

Vali Midagi muud

Enesetest:

Mis väljastatakse ekraanile?

```
lst = [[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]
print(len(lst))
```

Vali 3

Vali 4

Vali 12

Vali Midagi muud

Aga kui järjendite sees saab hoida järjendeid, siis kas nende sees saab hoida omakorda veel järjendeid? Jah, tehniliselt ei piira meid miski tekitamast ükskõik kui sügavaid seesuguseid struktuure. Võime isegi mõelda järjendite peale, kus igal sügavusastmel võib olla kas n-ö lõppelemente või veel järjendeid. Sellist andmestruktuuri võib nimetada *n-mõõtmeliseks järjendiks*. Kuidas tegutseda andmestruktuuriga, mille sügavust me ei tea? Sellest räägime selle kursuse viimastel nädalatel *rekursiivsete* andmestruktuuride juures.

1.2 JÄRJENDITE JÄRJEND

Eelmises osas sai räägitud, et kahemõõtmelisest järjendist saab elementidele ligi, kasutades kahte järjestikkust nurksulgudesse paigutatud indeksit:

```
jarjendite_jarjend = [[1, 2, 4], [-1, 5, 0], [], ['sõne']]
print(jarjendite_jarjend[0][2])  # Väljastab 4
```

Kuid samuti võime kahemõõtmelist järjendit vaadelda tavalise ühemõõtmelise järjendina, mille elemendid on lihtsalt mingid järjendid, mille sisu meid ei huvitagi. Näiteks võib meid huvitada ainult selle välimise järjendi elementide arv:

```
jarjendite_jarjend = [[1, 2, 4], [-1, 5, 0], [], ['sõne']]
print(len(jarjendite_jarjend))  # Väljastab 4
```

Või äkki hoopis tahame teha midagi tema viimase elemendiga (järjendiga), hoolimata otseselt sellest, mis elemendid selle sees on:

```
jarjendite_jarjend = [[1, 2, 4], [-1, 5, 0], [], ['sõne']]
viimane = jarjendite_jarjend[-1]
print(len(viimane))             # Väljastab 1
print(viimane.count(5))         # Väljastab 0
print(viimane.count('sõne'))    # Väljastab 1
print(viimane)                  # Väljastab ['sõne']
```

Kahemõõtmelist järjendit võime vastavalt vajadusele käsitleda kas

- teatud tüüpi tabelina, kus iga elemendi asukoht on määratud kahe indeksiga või
- tavalise ühemõõtmelise järjendina, mille elemendid on järjendi tüüpi väärtused.

Nii saame leida veel näiteks iga sisemise järjendi (rea) maksimaalse elemendi:

```
lst = [[1, 3, 2], [4, 5, 6], [7, 8, 9]]
for rida in lst:
    print(max(rida))
```

Enesetest:

Mis väljastatakse ekraanile?

```
lst = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
summa = 0
for el in lst:
  summa += len(el)
print(summa)
```

Vali 3

Vali 9

Vali 45

Vali Midagi muud

Tegelikult võime kahemõõtmelise järjendi teise taseme (sügavatele) elementidele ligi pääseda ka kahe sammuga: esmalt eraldame meid huvitava sisemise järjendi ning siis valime sellest omakorda sobiva elemendi.

```
jarjendite_jarjend = [[1, 2, 4], [-1, 5, 0], [], ['sõne']]
sisemine = jarjendite_jarjend[1]
print(sisemine[0])  # Väljastab -1
```

Pythoni "kõhus" toimub kõik täpselt samamoodi, olenemata sellest, kas valime eraldi välja rea ning sellest elemendi või kasutame kohe kahte indeksit järjest.

Kahemõõtmelised järjendid polegi tegelikult Pythoni jaoks midagi oluliselt erinevat ühemõõtmelisest järjendist. Seega saame ka kahemõõtmelisi järjendeid tekitada juba tuttavatel viisidel:

```
jarjendite_jarjend = []                # Praegu on see ühemõõtmeline
järjend
jarjendite_jarjend.append([1, 2, 4])  # Lisame sisemise järjendi -
nüüd on kahemõõtmeline
jarjendite_jarjend += [[-1, 5, 0]]    # Pane tähele! Kahekordsed
nurksulud!
tyhi = []
jarjendite_jarjend.append(tyhi)
viimane = []
viimane.append('sõne')
jarjendite_jarjend.append(viimane)
print(jarjendite_jarjend)              # Väljastab [[1, 2, 4], [-1,
5, 0], [], ['sõne']]
```

Enesetest:

Mis väljastatakse ekraanile?

```
meeste_nimed = ["Aleksander", "Vladimir", "Sergei", "Andrei", "Aleksei", "Andres"]
naiste_nimed = ["Olga", "Jelena", "Tatjana", "Irina", "Svetlana", "Valentina"]
nimed = [meeste_nimed, naiste_nimed]
print(nimed[1][4])
```

Vali Andrei

Vali Aleksei

Vali Svetlana

Vali Irina

Enesetest:

Mis väljastatakse ekraanile?

```
print([[4, 3], [2]][1][0])
```

Vali 4

Vali 3

Vali 2

Vali Veateade

Kuigi Python lubab hoida ühes järjendis mitut eri tüüpi väärtust, siis see on üpris veaohtlik, sest programmeerija peab siis meeles pidama, mis tüüpi väärtused kuskil asuvad. Kui võimalik, siis on mõistlik püüda hoida eri tüüpi väärtuseid erinevates järjendites.

1.3 MAATRIKS

Väga tihti on praktikas vaja kahemõõtmelisi järjendeid kasutada selliste andmete hoidmiseks, mis ongi olemuselt kahemõõtmelised, näiteks tabelid. Ilmselt on kõigile tuttav see tuntud tabel:

1	2	3	4	5	6	7	8	9	10
2	4	6	8	10	12	14	16	18	20
3	6	9	12	15	18	21	24	27	30
4	8	12	16	20	24	28	32	36	40
5	10	15	20	25	30	35	40	45	50
6	12	18	24	30	36	42	48	54	60
7	14	21	28	35	42	49	56	63	70
8	16	24	32	40	48	56	64	72	80
9	18	27	36	45	54	63	72	81	90
10	20	30	40	50	60	70	80	90	100

Selliste andmete puhul saame tavaliselt teha olulise eelduse: igas reas on täpselt sama palju elemente. Kuna mõiste *tabel* on siiski liiga mitmetähenduslik, siis mõnikord kasutame täpsuse huvides sünonüümina matemaatilist mõistet maatriks.

Maatriksiks nimetame kahemõõtmelist järjendit, mille igas sisemises järjendis (reas) on samapalju elemente.

Maatriksi mõõtmed antakse tavaliselt nii, et ridade arv on enne ja veergude arv pärast. Näiteks kirjapilt *2x3 maatriks*tähendab, et maatriksil on 2 rida ja 3 veergu.

Eeltoodud korrutustabelit võiks veel täpsemalt nimetada *ruutmaatriksiks*, sest tema ridade ja veergude arv on võrdne ehk ta on ruudukujuline. Ruutmaatriksite puhul on kasulikud mõisted *peadiagonaal* ja *kõrvaldiagonaal*.

Ruutmaatriksi peadiagonaaliks nimetame järjendit, mis sisaldab kõiki elemente maatriksi diagonaalilt, mis jookseb vasakust ülemisest nurgast paremasse alumisse nurka. Peadiagonaalil paiknevate elementide indeksid on alati võrdsed.

```
A = [[1, 2, 3],
     [4, 5, 6],
     [7, 8, 9]]
print(A[0][0])  # Väljastab 1
print(A[1][1])  # Väljastab 5
print(A[2][2])  # Väljastab 9
```

Kõrvaldiagonaaliks nimetame analoogiliselt järjendit paremalt ülevalt vasakule alla jooksva diagonaali elementidega.

Pange tähele! Diagonaalidest räägime vaid ruutmaatriksite puhul.

Enesetest:

Mis on 3x3 maatriksi kõrvaldiagonaalil olevate elementide indeksid? Tähistame indeksipaare `(i, j)`, kus `i` on reaindeks ja `j` veeruindeks.

- Vali `(0, 2), (1, 1), (2, 0)`
- Vali `(0, 0), (1, 1), (2, 2)`
- Vali `(0, 2), (1, 2), (2, 2)`

Enesetest:

Mida saame öelda iga ruutmaatriksi `A` kõrvaldiagonaalil olevate elementide indeksite kohta? Olgu reaindeks `i` ja veeruindeks `j`.

- Vali `i+j == len(A)-1` iga kõrvaldiagonaali elemendi korral
- Vali `i+j == len(A)` iga kõrvaldiagonaali elemendi korral
- Vali `i+j == len(A)+1` iga kõrvaldiagonaali elemendi korral
- Vali `i-j == 0` iga kõrvaldiagonaali elemendi korral

Enesetest:

Kui teame, et kahemõõtmeline järjend `A` on maatriks, siis mis avaldise väärtuseks on kindlasti `True`?

- Vali Kui `A` sisaldab vähemalt 1 elementi, siis `len(A[0]) == len(A[1])`
- Vali Kui `A` sisaldab vähemalt 1 elementi, siis `len(A) == len[A[0]]`
- Vali Kui `A` sisaldab vähemalt 2 elementi, siis `len(A[0]) == len(A[1])`

Järgmisel nädalal kahekordsete tsüklitega kahekordseid järjendeid analüüsides näeme, kuidas õpitud teadmisi praktiliselt rakendada.

1.4 MASINÕPE

Rasked ülesanded

Kursusel *Programmeerimise alused* ja sel kursusel oleme lahendanud juba mitmeid ülesandeid: kuidas arvutada optimaalset teleri kaugust diivanist, kuidas koostada peo eelarvet, kuidas tunda ära, kas perekonnanimi kuulub leedulannale või kuidas arvutada kala pikkuse järgi tema kaalu. Nende ülesannete jaoks on olemas olnud (või me oleme välja mõelnud) algoritmid ehk programmeerimiskeeles kirjapandavad formaalsed juhendid, mis määravad, kuidas programm peab käituma. Küll aga leidub ülesandeid, mida oleks tore programselt lahendada, kuid mille jaoks on "tavapärast" algoritmi väga keeruline kirja panna.

Olgu meie programmi ülesanne lugeda failist pilt ja leida sõnastikust seda pilti kõige paremini kirjeldav sõna. Failist pilti lugedes saab programm teada iga piksli värvikoodi (näiteks kuueteistkümnendsüsteemis). Kuidas leida sõnastikust sõna, mille tähendus vastab kõige paremini nende pikslite poolt kirjeldatud pildile?

```
[['#474747', '#4c4b4b', '#474545', ... ],
 ['#494646', '#444141', '#444242', ... ],      -->     'kiisu'
 ['#444242', '#444141', '#444242', ... ],
 ...]
```

Mida saaksime pikslitega teha? Näiteks saaks arvutada nende pikslite keskmise värvi. See lahendus võib töötada piiratud maailmas. Näiteks Robotexi robotid võivadki kasutada oma kaamera pildistatud pildi mingi ala keskmist värvi, et tuvastada näiteks, kas nende ees on oranž pall, vastase kollane värav või hoopis hall sein. Kui aga soovime tuvastada, kas foto peal on kiisu või kutsu, siis jääb see meetod hätta. Üldine probleem peitub selles, et üksikud pikslid ei ütle suurt midagi pildi tervikliku sisu kohta. Isegi kui me suudaks kirjeldada algoritmi, mis uurib pildi piksleid ja püüab nende põhjal ennustada, kas pildil on kass, siis ei tuleks kuidagi kõne alla teha seda kõikvõimalike sõnade ja objektide jaoks.

Programm õpib

Kui me ei suuda programmi algoritmi sisse kirjutada tarkusi, kuidas üht objekti teisest eristada, siis äkki suudame kirjutada algoritmi, mille abil saaks programm neid teadmisi ise ammutada olemasolevatest *treeningnäidetest*. Just sellist algoritmi, mis oskab etteantud sisendite ja väljundite põhjal õppida sooritama mingit teisendust, nimetatakse *masinõppe* algoritmiks.

Seda, mis masinõppe algoritmi juures määrab, kuidas teisendus sisendist väljundiks tehakse, nimetatakse algoritmi *mudeliks* (mõnikord ka *meetodiks* või *algoritmiks*). Igas mudelis on defineeritud *mudeli parameetrid* (või *kaalud*), mis hoiavadki treeningnäidetest õpitud "teadmisi". Need mudeli parameetrid on tavalised ujukomaarvud, mida kasutatakse mõnes valemis mudeli sees.

Järgneb üks väga lihtne masinõppe algoritm, mis oskab näidete põhjal õppida sooritama lihtsaid aritmeetilisi tehteid. Mudeli sisendiks on üks ujukomaarv ja väljundiks üks ujukomaarv. Täpsemalt oskab mudel õppida selgeks mistahes *lineaarteisenduse*. See tähendab, et sisendist saab väljundi, kasutades valemit, mis sisaldab vaid liitmist-lahutamist ja arvuga (mitte sisendväärtusega) korrutamist-jagamist. Näiteks üks lineaarteisendus on selline:

```
väljund = (42 * sisend - 4) * 2 + 1
```

ehk lihtsustatult:

```
väljund = 84 * sisend - 7
```

Programm loeb failist `treeningandmed.txt` treenimiseks kasutatavad sisend- ja väljundväärtused, treenib nende põhjal mudelit, küsib kasutajalt uue sisendväärtuse ning ennustab sellele vastava väljundi. Failis on igal real üks treeningnäide, kus sisend ja väljund (sellises järjekorras) on eraldatud komaga. Selle mudeli treenimiseks on vaja ainult kahte treeningnäidet, seega failis peab olema vähemalt kaks rida näidetega. Kuna rohkem näiteid pole vaja, siis arvestataksegi ainult kahte esimest rida.

Proovige välja mõelda üks selline valem, koostage treeningandmete fail ja katsetage, kas programm oskab valemi ära arvata. Vajadusel võite malliks võtta eelneva näiteteisenduse kohta loodud faili treeningandmed.txt.

```
def opi(x_list, y_list):
    # Modelleerime sõltuvust sisendi ja väljundi vahel
    # lineaarfunktsiooni (sirge) abil: valjund = a * sisend + b, kus
    # a ja b on kordajad ehk kaalud ehk mudeli parameetrid.
    # Kordajate arvutamiseks kasutame sirge võrrandit kahe punktiga.
    # Kuna sirge on määratud kahe punktiga, siis õppimiseks
rakendamegi
    # ainult kahte esimest näidet isegi siis, kui neid on antud
rohkem.

    a = (y_list[1] - y_list[0]) / (x_list[1] - x_list[0])
    b = (y_list[0] * x_list[1] - y_list[1] * x_list[0]) /
(x_list[1] - x_list[0])
    return [a, b]  # Tagastame kordajad listina

def ennusta(x, kaalud):
    # Ennustamiseks on vaja arvutada lineaarfunktsiooni väärtus
    # kohal x, mis vastab kasutaja sisendile, kasutades õpitud
kordajaid.
    a = kaalud[0]
    b = kaalud[1]
    return a * x + b

# Loeme treeningandmed failist.
f = open('treeningandmed.txt')
sisendvaartused = []
valjundvaartused = []
for rida in f:
    jupid = rida.split(',')
    sisendvaartused.append(float(jupid[0]))
    valjundvaartused.append(float(jupid[1]))
f.close()

# Õpime treeningandmete põhjal selgeks kaalud ehk mudeli
parameetrid.
kaalud = opi(sisendvaartused, valjundvaartused)

# Ennustame uue sisendi väärtust, kasutades õpitud kaale.
kasutaja_sisend = float(input('Sisend: '))
ennustus = ennusta(kasutaja_sisend, kaalud)
print('Ennustus: ' + str(ennustus))
```

Selle algoritmi mudeliks on lineaarfunktsioon, millel on kaks treenitavat parameetrit. Mudeli treenimiseks on vaja ka vaid kahte näidet sisendist ja väljundist, sest nagu koolist teate, siis iga sirge määramiseks tasandil on piisav fikseerida kaks erinevat punkti, mida see sirge läbib (vt kahe punktiga määratud sirge võrrand). Täpselt seda seal mudelis tehaksegi: eeldatakse, et sisendi ja väljundi vahel on lineaarne sõltuvus ning joonestatakse kahe antud punkti põhjal sirge. Uuele sisendile arvutatakse väljund selle leitud sirge (funktsiooni) abil.

Masinõpe päris rakendustes

Masinõppe rakendustes kasutatakse sageli palju keerulisemaid sõltuvusi sisendi ja väljundi vahel. Samuti on selle arvelt treenitavaid parameetreid rohkem ning mudelite treenimiseks on vaja väga palju treeningandmeid. Ent siiski on kõik masinõppe rakendused oma põhimõtetelt väga sarnased eelnevale näitele.

Märkus. Tegelikult räägime siin täpsemalt *juhendatud masinõppest*, kus treenitakse (juhendatakse) algoritm käituma meile sobivalt. On olemas ka *juhendamata masinõpe*, mille eesmärk on leida sisendandmetest struktuure ilma, et talle oleks midagi ette öeldud selle kohta, mis väljundiks peaks olema.

Pildituvastuse või masintõlke valdkondades on tavaline kasutada mudeleid, millel on kümneid (või isegi sadu) miljoneid treenitavaid parameetreid ning mida treenitakse kümnete miljonite treeningnäidete peal. Samuti ei leidu enamasti parameetrite jaoks valemeid, millega nende väärtust täpselt määrata (n-ö *analüütiline lahendus*), või need lahendused on liiga keerulised. Sel juhul leitakse parameetrite umbkaudsed väärtused mõne optimeerimisalgoritmiga (nt *gradientlaskumine* - ingl *gradient descent*), mis püüab väikeste sammudega liikuda "õigemate" väärtuste poole nii kaua, kuni tulemus sellest paraneb.

Internetti kasutades puutute tõenäoliselt iga päev kokku masinõppega. Kui kasutate Google'i, Facebooki või Apple'i teenuseid, näidatakse teile sisu, mis on kujundatud masinõppe meetodite abiga ning samuti kasutatakse teie käitumist ja andmeid uute masinõppemudelite treenimiseks. Ülesanded, mida masinõppemeetodid lahendavad, võivad tunduda programmeerimise seisukohast hoomamatud ja nende lahendused lausa maagilised. Kuid tegelikult peitub iga sellise programmi taga kavalate matemaatiliste-statistiliste nippidega algoritm, mis suudab õppida teisendusi sisendi ja väljundi vahel. Võibolla on inimese geenides defineeritud sarnane algoritm, mis õpib teisendama meelte abil saadud sensatsioone liikumiseks, rääkimiseks, söömiseks ja programmeerimiseks?